Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
https://qce.ewi.tudelft.nl/

2024

MSc THESIS

A Dynamically Reconfigurable RISC-V Processor Based on the MOLEN Paradigm

D.M. van den Berg

Abstract

In this thesis, we present a RISC-V processor that is extended with the MOLEN ISA extension, thereby granting it dynamic reconfiguration capabilities. The reconfigurable microcode ($\rho\mu-$code) of the MOLEN paradigm is modified to be suitable for (FPGA) implementation in the 64-bit Linux-capable CVA6 RISC-V processor. The *set* instruction performs reconfigurations by pointing it to a partial bitstream address, after which the *execute* instruction can perform operations on the reconfigured hardware. To this end, the concept of nested $\rho\mu-$code is presented, in which the reconfigurable opcodes are encapsulated in regular RISC-V instructions. Furthermore, a *status* instruction is introduced to enable the reconfiguration to be performed in the background. Consequently, the reconfiguration latency can be hidden, by allowing the CPU to do useful work during the reconfiguration. Using various experiments, it is demonstrated that the proposed implementation has a near-optimal reconfiguration performance and that the reconfiguration latency can be effectively hidden in typical cases.

Keywords: MOLEN Processor, RISC-V, reconfigurability, microcode

Faculty of Electrical Engineering, Mathematics and Computer Science

A Dynamically Reconfigurable RISC-V Processor Based on the MOLEN Paradigm

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

D.M. van den Berg
born in Dordrecht, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Dynamically Reconfigurable RISC-V Processor Based on the MOLEN Paradigm

by D.M. van den Berg

Abstract

In this thesis, we present a RISC-V processor that is extended with the MOLEN ISA extension, thereby granting it dynamic reconfiguration capabilities. The reconfigurable microcode ($\rho\mu-$code) of the MOLEN paradigm is modified to be suitable for (FPGA) implementation in the 64-bit Linux-capable CVA6 RISC-V processor. The *set* instruction performs reconfigurations by pointing it to a partial bitstream address, after which the *execute* instruction can perform operations on the reconfigured hardware. To this end, the concept of nested $\rho\mu-$code is presented, in which the reconfigurable opcodes are encapsulated in regular RISC-V instructions. Furthermore, a *status* instruction is introduced to enable the reconfiguration to be performed in the background. Consequently, the reconfiguration latency can be hidden, by allowing the CPU to do useful work during the reconfiguration. Using various experiments, it is demonstrated that the proposed implementation has a near-optimal reconfiguration performance and that the reconfiguration latency can be effectively hidden in typical cases.

Laboratory	:	Computer Engineering
Committee Members	:	

Advisor:	Dr. ir. J.S.S.M. Wong, CE, TU Delft
Chairperson:	Dr. ir. J.S.S.M. Wong, CE, TU Delft
Member:	Dr. ir. T.G.R.M van Leuken, SPS, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

CCA Configurable Compute Accelerator

CCU Custom Configured Unit

CGRA Coarse-Grained Reconfigurable Array

CPU Central Processing Unit

CSR Control and Status Register

DMA Direct Memory Access

FIFO First-In-First-Out

FPCA Fully Pipelined Composable Architecture

FPGA Field Programmable Gate Array

FPU Floating Point Unit

FSM Finite State Machine

HDL Hardware Description Language

HLS High-Level Synthesis

IC Integrated Circuit

ICAP Internal Configuration Access Port

ILA Integrated Logic Analyzer

ISA Instruction Set Architecture

LSU Load Store Unit

LUT Lookup Table

OS Operating System

PC Program Counter

PE Processing Element

PULP Parallel Ultra Low Power

RAM	Random Access Memory
RTL	Register-Transfer Level
RTOS	Real-time Operating System
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
XREG	Exchange Register

Acknowledgements

At the end of the journey that this MSc. project has been, there are some people I would like to express my gratitude to. Without them, completing this thesis would have been much harder, or perhaps even impossible.

First of all, I want to thank dr. ir. Stephan Wong, my thesis supervisor, for all of his advice and guidance. I would also like to thank him for his patience and understanding when I was putting too much pressure on myself.

I also want to thank dr. ir. René van Leuken for being part of my thesis committee, as well as for accommodating my thesis defense close to his retirement.

Furthermore, I want to thank my family for their continued support, without which I probably would not have been able to see this project through to the end.

Finally, I would like to thank Eline, whose insights have helped me to get back on track when things got tough.

D.M. van den Berg
Delft, The Netherlands
November 12, 2024

1 Introduction

In this thesis, the design of a dynamically reconfigurable processor is documented. The MOLEN paradigm is applied to the open-source RISC-V Instruction Set Architecture (ISA), resulting in a reconfigurable design that could aid the adoption of reconfigurable architectures in multiple domains.

In Section 1.1, the motivation of this project is discussed. Subsequently, the main research question and goals are formulated in Section 1.2. These goals are then adapted into the methodology of the project in Section 1.3. Finally, the structure of this thesis is outlined in Section 1.4.

1.1 Motivation

Historically, the widely known observation known as Moore's law ensured a continuous progression of the Integrated Circuit (IC) design process, by setting the industry goals. Back in 1965, it described the transistor scaling as a doubling of the IC transistor count every year [4]. Ten years later, it was adjusted to a doubling every two years [5], a trend that continued even in recent years.

Another observation that aided in this progression is Dennard Scaling, which describes that the power density of the transistors stays constant as the size decreases [6]. As a result, the (single-core) processor performance was able to continuously increase by developing larger and faster cores.

However, Dennard Scaling ceased to hold true around 2005 [7], resulting in increased power and heat buildups that halted single-core improvements [8]. This problem was mitigated by switching to multi-core processors, allowing the additional available transistors (resulting from Moore's law) to be used for additional cores instead of larger and faster cores [8] [9].

Though the switch to multi-core has extended Moore's law, it has become clear in recent years that CMOS transistor scaling is coming to an end due to physical limitations. This limit is expected to be a channel length of around 3 nm, but the practical limit could be even higher [10].

In addition to the absolute scaling limit, recent chips already suffer from another effect known as dark silicon: due to the end of Dennard scaling, power constraints limit the amount of transistors that can be used at the same time, causing parts of the chip to remain “dark” (turned off) [9]. Decreasing the transistor sizes worsens this effect due to the higher relative power consumption.

Finally, the limited inherent parallelism of typical software applications poses limits on the simultaneously usable chip area, in which case the power limitations do not even come into play [9].

The previous observations make it clear that new innovations are necessary in order to continue the performance increases and power decreases. Such innovations can take place in the following abstraction layers:

1. The **architecture** deals with the conceptual structure of the design and the functional behavior as observed by the user or developer. This includes the ISA.

2. The **implementation (micro-architecture)** defines how the architecture is implemented on a logical level.

3. The **realization** provides the physical structure of the micro-architecture. This includes the process technology and the physical layout of the transistors in that technology.

Potentially, changes in the lower abstraction levels will also require changes to the higher levels. For example, in a shift to quantum computing, all levels would be drastically changed. By extending the architecture with reconfigurability, compatibility with regular architectures can be maintained, while at the same time being able to use the available hardware more efficiently for the specific task at hand. Instead of having a lot of fixed, specialized units that cannot all be used at the same time due to the dark silicon effect, a reconfigurable unit provides an effective method of using the available area for functional units or accelerators that can be used at the same time. For that reason, reconfigurability is the chosen improvement avenue in the MSc. project described in this thesis. Moreover, associated with reconfigurability is always the reconfiguration latency of the underlying (reconfigurable) hardware fabric. Consequently, reconfiguration latencies (or the hiding thereof) will also be part of our project.

1.2 Problem statement and goals

Based on the motivation of the previous section, the research question can be formulated as follows:

> *Will the application of the MOLEN paradigm to a modern processor allow for hiding reconfiguration latencies?*

From this research question, the main goal can be derived, namely to introduce reconfigurability to an existing processor by implementing the MOLEN instruction set extension. Specifically, a functional reconfigurable design will be implemented, by using an existing RISC-V core and integrating the MOLEN scheme into it. In order to make this feasible, a minimal working design is considered sufficient. In other words, the goal of the implementation is to provide a proof of concept rather than a full-fledged design. This means the MOLEN scheme will be simplified to a minimal version. Specifically:

- Having a single reconfigurable area in which custom designs can be loaded is sufficient. Having multiple reconfigurable areas from the start would increase the complexity of the implementation towards a full-fledged design, which is not the goal of this project as stated before.

- The reconfigurable unit is intended for executing single (custom) instructions (or small sequences of instructions). No branching capabilities will be added.
- A single location will be used for storing the reconfigurable code (the main memory). No distinction will be made between often and less often used code, simplifying the loading of the programs.

In contrast with the original MOLEN scheme, the $\rho\mu$-code (as introduced in Section 2.2) is intended for use by developers/programmers instead of being hidden from them like traditional microcode. The reason for that is that the proposed design is a proof-of-concept, which should enable developers to experiment with the reconfigurable hardware, instead of locking them out from the specifics.

The desired outcome of this project is an FPGA-based implementation of a (soft-) RISC-V core with the described reconfiguration capabilities, using the MOLEN instruction set extension.

1.3 Methodology

In order to achieve the goals from the previous section and be able to answer the research question, the following set of milestones is followed during the MSc. project described in this thesis:

1. **Literature Study:** Compare existing reconfigurable architectures, such that the use of the MOLEN paradigm for this project can be justified.
2. **Minimal working design:** A minimal working implementation of a RISC-V core that is able to run (C) programs. No reconfiguration is performed in this stage. This design will also function as reference implementation during the final milestone.
   (a) **Select the RISC-V core:** Choose a core implementation that is suitable for extension with reconfigurability.
   (b) **Design the basic SoC:** Integrate the chosen core into a SoC containing memory and all necessary interfaces/buses. If possible, an existing SoC design could also be chosen, simplifying this milestone.
   (c) **Test the design:** Verify correct working of the design by compiling and running some basic code.
3. **Reconfigurable design:** Extend the minimal design with reconfigurability such that it can execute $\rho\mu$-code.
   (a) **Design the reconfigurable unit:** Design a standalone reconfigurable unit that is able to reconfigure the hardware and perform instructions on that hardware, using $\rho\mu$-code.
   (b) **Integrate the reconfigurable unit:** Integrate the reconfigurable unit into the core by implementing the arbiter that distinguishes between $\rho\mu$-code and

regular instructions (as introduced in Section 2.2). This may include a minimal compiler/assembler modification such that the *set/execute* instructions can be performed.

4. **Result verification:** Once the design is completed and functional, the performance and (resource) costs need to be compared to the standalone RISC-V implementation without reconfigurability.

   (a) **Select Application:** Choose a suitable application that can highlight the benefits of the reconfigurable design by allowing the reconfiguration latency to be hidden. For example, JPEG compression could be chosen in line with the previous work on the MOLEN architecture. Instead of a real-world application, a simplified benchmark may also be chosen.

   (b) **Implement software-only application:** Implement the chosen application on the regular RISC-V design.

   (c) **Implement soft-/hardware application:** Implement the chosen application on the reconfigurable RISC-V design.

   (d) **Result comparison:** Compare the results of both designs in terms of performance and resource usage.

1.4 Thesis structure

The remainder of this thesis is structured in the following way: in Chapter 2, the concepts that this work builds on are introduced, a RISC-V core is selected for the modifications of this project, and related work is explored. In Chapter 3, the MOLEN paradigm is adapted to the RISC-V ISA, resulting in a proposed ISA extension. In Chapter 4, the implementation of the proposed extension into the selected RISC-V core is discussed. The resulting implementation is subjected to a number of experiments in Chapter 5. Finally, this thesis is summarized and concluded in Chapter 6.

Background 2

As the research question stated in Chapter 1 pertains to reconfigurable architectures, this chapter will briefly introduce related concepts and related work to better understand the work of this MSc thesis project.

First, a general introduction to reconfigurable architectures is given in Section 2.1. Subsequently, the MOLEN paradigm is introduced in Section 2.2, followed by the introduction of the RISC-V architecture in Section 2.3. The Field Programmable Gate Array (FPGA) platform that was used for this project is introduced in Section 2.4, after which an existing RISC-V implementation is selected for modification with the MOLEN paradigm in Section 2.5. Following that, related work is explored in Section 2.6. Finally, this chapter is concluded in Section 2.7.

2.1 Reconfigurable architectures

With the need for reconfigurable architectures being explained in Section 1.1, the current section can focus on the types of reconfigurable architectures and their use cases. First, two common application domains are described. Subsequently, the primary architectural classification of reconfigurable systems is introduced. Finally, an introduction to a common type of reconfigurable architecture is given.

2.1.1 Application domains

Reconfigurability seems to be primarily applied in the following two domains: high performance computing and embedded systems. This section provides an introduction into both domains.

In the case of **high performance computing**, it is usually applied by means of reconfigurable accelerators (FPGAs). There, speedups of a factor 32 and energy reductions of a factor 36 have been achieved [11]. Logically, the reconfigurable hardware increases the (programming) complexity, but frameworks and high level languages can help mitigate this added complexity. For example, OpenCL can be used for these kind of accelerators.

The use of a separate reconfigurable accelerator poses a bandwidth limitation on the speed and amount of data that can be processed at once. A bus must be used to interface the processor and accelerator, resulting in communication overhead.

In the case of **embedded systems**, performance and energy efficiency is not necessarily the motivation for applying reconfigurability. Typically, embedded systems need to deal with time-critical operations, in addition to non time-critical ones. Traditionally, embedded processors took care of the non time-critical tasks, whereas Application Specific Integrated Circuits (ASICs) performed the time-critical ones. However, ASIC development results in lengthy design cycles. With FPGA performance approaching ASIC performance over the years, FPGAs became the component of choice in typical embedded systems [12]. Not only did this shorten the design cycle, it also allowed for loading new configurations onto the reconfigurable fabric when needed.

More recently, embedded processors became capable enough to meet the timing constraints of the time-critical tasks, reducing the need for ASICs or FPGAs next to the processor. However, specialized tasks such as multimedia processing will still require hardware implementation in order to meet timing constraints. In addition to that, reconfigurable hardware increases the flexibility of the system and allows for future changes without needing a new platform. Finally, in cases where reconfigurability would not be required for performance reasons, it could still be useful for energy efficiency, as it allows a smaller processor to be used by implementing the time-critical functions in the reconfigurable fabric.

2.1.2 Reconfigurable granularity

Different kinds of reconfigurable systems exist. An important architectural distinction can be made between fine-grained and coarse-grained reconfigurable systems. Fine-grained reconfigurable systems allow for complete reconfiguration at the lowest level (gate level), whereas coarse-grained reconfigurable systems only allow a higher-level reconfiguration on a block level.

Though fine-grained systems offer more flexibility, they typically use more power and run at lower clock speeds. Coarse-grained systems decrease the flexibility but can run at higher clock speeds and are more energy efficient. However, recent FPGAs often provide coarse-grained blocks in addition to fine-grained ones. Therefore, the choice of FPGA already influences the reconfigurable granularity.

2.1.3 Coarse-Grained Reconfigurable Arrays (CGRAs)

A specific, common type of coarse-grain reconfigurable architecture is the Coarse-Grained Reconfigurable Array (CGRA). Its name is derived from its structure: a CGRA consists of an array of Processing Elements (PEs). Each PE can be seen as a tiny processor that has access to some memory, logic and arithmetic units. The precise structure of the PEs varies between CGRA implementations and is determined by the use case of the system. As such, the PE structure can vary from a small processor with its own instruction set to a data flow unit that only has a few instructions. Such a data flow unit can quickly compute and accumulate results once the inputs are ready. The results can then be passed around to other PEs.

Though the variety in the PE structure results in an equal variety in the CGRA use cases and performance, the parallel nature of the structure implies that it is mostly suitable for speeding up computations on large amounts of data.

2.2 The MOLEN polymorphic processor

Historically, microcode was used to implement (emulate) instructions that could not be implemented in hardware at that time. Thanks to Moore's law, those instructions could then be converted to hardware instructions in later designs. This way, the ISA could be developed independently from the technological state of the hardware. Even though some instructions required emulation using microcode, software developers were presented with a uniform set of instructions, without being able to distinguish between microcoded and hardware instructions.

The MOLEN polymorphic processor takes this concept to a new level, by defining reconfigurable microcode ($\rho\mu$-code) [1]. This new kind of microcode can be used to perform the reconfiguration of the reconfigurable fabric, as well as to execute custom operations on the configured fabric. Because microcode is used for these operations, a one-time ISA extension is sufficient. This extension contains a *set* and *execute* instruction. The *set* instruction configures the reconfigurable fabric, wheres the *execute* instruction performs operations on the configured hardware. Both of these instructions work by taking the address of the to-be-executed microcode. Microcode execution will then commence at that address, after which it continues until a special *end_op* micro-instruction is encountered. *set* instructions are derived from the reconfiguration file (bitstream) by splitting that file into equally sized blocks, to which the *set* opcode is then appended.

2.2.1 MOLEN ISA extension

The one-time architectural extension required for implementing the MOLEN machine can be one of the following [1]:

- **Minimal extension:** this extension contains only the essential instructions that are required for a working implementation. These are *set*, *execute*, *movtx* and *movfx*. Here, *set* and *execute* reconfigure the Custom Configured Unit (CCU) (as introduced in Section 2.2.2) and execute code on the reconfigured CCU, respectively. The *movtx* and *movfx* instructions allow arguments to be written to and loaded from the exchange registers, respectively. The *set* instruction is actually a *c-set* instruction that configures the entire CCU.

- **Preferred extension:** This extension adds a *p-set* instruction that (pre-)configures parts of the CCU. The *c-set* instruction that was already present then configures only the remaining part of the CCU. Additionally, this extension defines *prefetch* instructions for the *set* and *execute* instructions, that enable microcode loading before it is needed. This diminishes the loading times.

- **Complete extension:** This final extension adds a *break* instruction that allows synchronization between the Central Processing Unit (CPU) and reconfigurable unit, such that parallel execution can take place. Now, the CPU and reconfigurable unit will run in parallel when an *execute* instruction is issued. Subsequently, the *break* instruction is used to wait for the termination of the CPU and reconfigurable instructions.

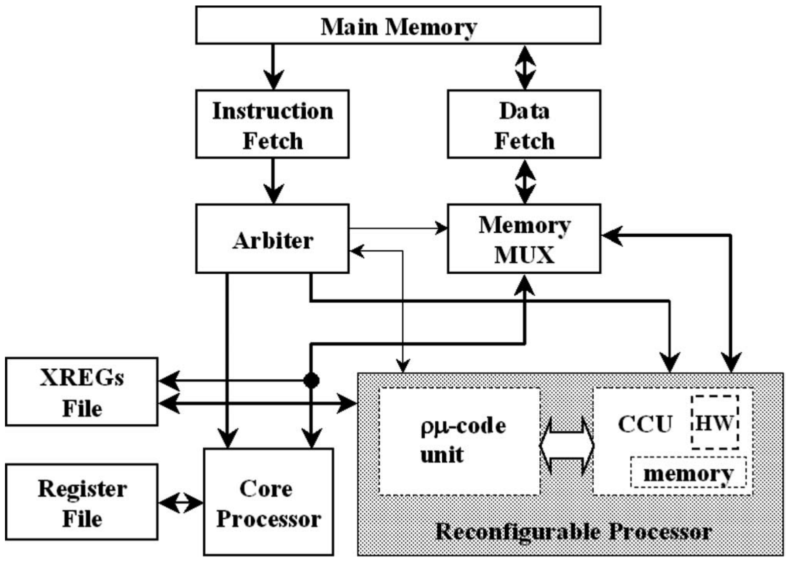

Figure 2.1: The MOLEN machine organization. Image courtesy of [1].

2.2.2 MOLEN organization

An overview of the MOLEN machine organization is depicted in Figure 2.1. The MOLEN machine extends a regular von Neumann architecture with the following components:

- The **Reconfigurable Processor** provides (as the name implies) the actual reconfigurable hardware and the control of that hardware. It consists of a **CCU** and a $\rho\mu$**-code unit**. The CCU provides the reconfigurable fabric, whereas the $\rho\mu$-code unit deals with the microcode initialization, loading, execution and storage. In order to have a tight integration of the reconfigurable hardware with the other parts of the system and boost performance, the CCU has Direct Memory Access (DMA). It should also be noted that the $\rho\mu$-code unit does not decode the actual micro-instructions, leaving this to the CCU instead (or optionally to a fixed or hybrid decoder).

- The **Arbiter** decodes the fetched instructions and issues them to either the Core Processor (regular instructions) or the $\rho\mu$-code unit (*set*/*execute* instructions).

- The **Exchange Registers (XREGs)** enable the Core Processor to pass function arguments to the reconfigurable unit, and to retrieve the corresponding results.

2.3 The RISC-V architecture

The RISC-V architecture is an open-source ISA that originated in the Parallel Computing Laboratory (Par Lab) at UC Berkeley [13]. The Par Lab was funded by Intel and Microsoft to advance parallel computing and ran from 2008 to 2013. Halfway through, in May 2010, this resulted in the inception of the RISC-V instruction set. This was followed by the first specification of the ISA in 2011 [14]. Influenced by many previous

ISAs [15], the RISC-V specification was not originally a goal of the project, but rather a means to achieve the goals relating to research into parallel processing systems [13]. However, the open nature of the specification attracted worldwide attention, because it allows anyone to develop their own hardware, while being able to share the software. Since its inception, work on the RISC-V ISA has continued, and in 2015 the RISC-V Foundation was formed to direct the development and stimulate the adoption of the ISA [16].

Despite parallel processing systems being the initial motivation for the creation of the ISA, it supports a wide range of applications. This is due to the highly modular design of the ISA with multiple bases and extensions [3]. Specifically, the following base instructions sets are defined:

- **RV32I**: A 32-bit (integer) instruction set covering 40 basic instructions such as load and store operations.

- **RV32E**: A 32-bit (integer) instruction set for embedded systems. This instruction set limits the number of registers to 16, as opposed to 32 for the RV32I instruction set. It is equal otherwise.

- **RV64I**: A 64-bit (integer) instruction set. This instruction set is also based on the RV32I instruction set, but widens the registers to 64 bits. Furthermore, 15 instructions are added to the base RV32I instruction set.

- **RV128I**: A 128-bit (integer) instruction set. Like the RV64I instruction set, it expands the register widths, albeit to 128 bits instead of 64. It is primarily intended for future use cases, when address spaces larger than 64 bits might be required [3].

Because these base instruction sets define only simple operations, several standard extensions are also defined, providing operations such as integer multiplication and division or atomic instructions. By making the instruction set modular in this way, it is suitable for a wide range of applications, ranging from high-performance computing to general purpose computing and embedded systems.

In addition to its modularity, the instruction set also supports custom extensions and has special opcodes reserved for such extensions [3]. This enables hardware developers to include their own (specialized) instructions to make their processor implementation suitable for the envisioned application domain.

These advantages, as well as the available compiler and simulator support, make the RISC-V architecture very suitable for extension with reconfigurability. By basing such a design on the RISC-V ISA, it is implicitly made suitable for a broad range of applications. The ability to add custom instructions ensure that the one-time ISA extension as defined by the MOLEN paradigm can also be added. Finally, the open nature of the RISC-V ISA enables developers to built on previous work and re-use existing open-source processor implementations instead of having to reinvent the wheel.

2.4 FPGA platform

The hardware platform that is used for the implementation of this project is the Xilinx VC707 evaluation kit [17], which is built around the Virtex-7 XC7VX485T-2FFG1761C FPGA. It has 485760 logic cells and 37080 KB of on-chip memory. The board also includes 1 GB DDR3 memory with speeds up to 1600 Mbps. All in all this should provide enough resources for the implementation of the project. Partial reconfiguration is also supported, allowing the intended reconfiguration of this project.

2.5 RISC-V implementation

As mentioned in Section 2.3, a major advantage of the RISC-V ISA is that it is open-source and can be implemented by anyone. As a result, multiple open-source core implementations have been developed. Instead of having to reinvent the wheel, it is possible to select an existing RISC-V core and extend it with the proposed reconfigurability. Since the design of a RISC-V core is not one of the goals of this thesis, the additional efforts in implementing and testing such a core can be saved.

Instead of just cores, open-source Systems on Chip (SoCs) are available as well. These SoCs use one of the cores and add memory, peripherals and interconnects to the chip. This results in a ready-to-use or nearly ready-to-use design. Such a design is preferable for this project, for the same reason as stated before.

2.5.1 Requirements

Because multiple implementations are available, the most suitable one needs to be selected. To do so, a set of requirements must be specified. Most of those requirements focus on the core (and SoC) being small or simple, in order to aid the implementation of the reconfigurable extension. The following list of requirements is used:

- **Hardware support:** The core should be compatible with the Xilinx VC707 board that is used in this project. Having official support in addition to that would be a nice-to-have.
- **Instruction width:** Preferably, the core should be 32-bit to reduce the complexity of the design. However, a 64-bit core would be acceptable, as long as it fits on the FPGA.
- **Single core:** Instead of using multiple cores and complicating the design, a single core is sufficient for this project.
- **Extensibility:** The SoC must accommodate the extension with the proposed reconfigurability. This includes the ability for DMA from within the reconfigurable part of the system.
- **Simulation and debugging support:** In order to successfully develop hardware designs, proper simulation and debugging support is essential.

- **Scalar execution:** Instead of using redundant functional units to achieve superscalar execution, the core should be scalar to reduce its complexity.
- **Hardware Description Language (HDL) implementation:** The core should be implemented in one of the classical HDL languages (VHDL, Verilog or SystemVerilog). Compared to High-Level Synthesis (HLS) languages, these are more widely supported and enable more reusability.
- **Documentation:** In order to be usable, the core should have proper documentation or at least use clean and well-written code.
- **Community backing:** in any open-source project, including SoC implementations, sufficient community backing is vital to the viability of the project. Projects with single developers behind them are more prone to being abandoned or insufficiently maintained than projects with large communities behind them. To ensure future work on the proposed design is feasible, the selected core should have sufficient community support.
- **Compliance Suite:** The implementation should preferably pass the (draft) RISC-V compliance suite, since doing so provides guarantees about the correctness and compatibility of the core.
- **ASIC support:** In addition to supporting FPGA synthesis, the SoC should be suitable for ASIC implementation, in order to accommodate future work on the proposed design.
- **Permissive license:** For the SoC to be usable in this project, the license under which it is released must allow using, modifying and distributing the source code. This means that no commercial implementations will be considered.
- **Operating System (OS) support:** To allow a future continuation of this project, readily available support for an OS is preferable. This opens the possibility for future research into OS integration of the current design.

Using this set of requirements, the available SoC implementations can be evaluated. An overview of available cores and SoCs on the RISC-V website is used as starting point [18]. The first step consists of evaluating the available cores. When the suitable cores are identified, the SoC implementations using those cores (or custom cores) can be evaluated.

2.5.2 Core selection

Inspecting the list of available cores, the first observation that can be made is that a large number of cores is released under a commercial or restrictive license. Following the requirements, these cores will not be considered for this project. At the time of writing, 66 RISC-V cores are available, of which 35 are released under a permissive license. Of these 35 cores, 12 are implemented in (novel) high-level languages, such as Chisel or Bluespec. These cores will not be considered per the requirements, which leaves the 23 cores listed in Table 2.1 for consideration.

It can be observed from the table that only a few cores offer OS support out of the box. Most of these cores support the Zephyr OS, which is an (embedded) Real-time Operating System (RTOS). Preferred over an RTOS would be a full-fledged OS such as Linux, due to its larger range of use cases. Two of the available cores offer support for Linux: The CVA6 and the biRISC-V. The biRISC-V core appears to be the work of a single developer and lacks community backing. By looking further into the CVA6 core, we can now decide if it suitable or if the requirement of having OS support must be dropped.

Name	**Instruction width**	**Privileged specification**	**User specification**	**ISA**	**Implementation language**	**OS support**
CVA6	64	1.11	2.3	RV64GC	SystemVerilog	Linux
CV32E40P	32	1.11	2.1	RV32IMC	SystemVerilog	-
Ibex	32	1.11	2.1	RV32I[M]C/RV32E[M]C	SystemVerilog	-
Kronos	32	1.11	2.1	RV32I	SystemVerilog	-
Roa Logic RV12	32/64	1.10	2.2	RV32I/RV64I	SystemVerilog	-
RSD	32	unknown	unknown	RV32IM	SystemVerilog	Zephyr
SCR1	32	1.10	2.2	RV32I[MC]/RV32E[MC]	SystemVerilog	Zephyr
SweRV EH1	32	1.11	2.1	RV32IMC	SystemVerilog	-
SweRV EH2	32	1.11	2.1	RV32IMAC	SystemVerilog	-
SweRV EL2	32	1.11	2.1	RV32IMC	SystemVerilog	-
Taiga	32	unknown	unknown	RV32IMA	SystemVerilog	-
biRISC-V	32	1.11	2.1	RV32I[M]	Verilog	Linux
DarkRISCV	32	-	incomplete	RV32I	Verilog	-
Hummingbird E200	32	1.10	2.2	RV32IMAC	Verilog	-
mRISC-V	32	unknown	unknown	RV32IM	Verilog	-
PicoRV32	32	-	unknown	RV32I[MC]/RV32E[MC]	Verilog	-
SERV	32	incomplete	unknown	RV32I	Verilog	Zephyr
SSRV	32	1.10	unknown	RV32IMC	Verilog	-
Maestro	32	-	incomplete	RV32I	VHDL	-
ORCA	32	unknown	unknown	RV32IM	VHDL	-
ReonV	32	unknown	unknown	RV32I	VHDL	-
RV01	32	1.7	2.0	RV32IM	VHDL	-
RPU	32	-	unknown	RV32I	VHDL	Zephyr

Table 2.1: Overview of RISC-V cores that have a permissive license and use a classical HDL.

The CVA6 processor [2] is a 64-bit, 6-stage in-order single-core processor with a focus on efficiency. It was developed as part of the Parallel Ultra Low Power (PULP) platform, a collaboration between ETH Zürich and the University of Bologna to research ultra-low-power architectures. The core was originally named Ariane, but it was donated to the OpenHW Group and renamed to CVA6 in June 2020. The core has a wide community and extensive documentation. It is actively maintained at the time of writing. ASIC synthesis is also supported, which was demonstrated by creating a 1.7 GHz implementation in 22 nm technology. The modular and well-structured design make it suitable for the addition of custom extensions. **As such, the requirements of the previous section are met, and the CVA6 processor is selected for the ISA extension of this project**.

The CVA6 is not just a core, but a SoC platform containing the core. This means other SoC platforms do not need to be considered, and no SoC has to be developed manually around the core. It also means that the second milestone of Section 1.3 is simplified to loading the CVA6 design onto the FPGA and verifying that the OS can boot.

It should be noted that, during the course of the MSc. project described in this thesis, a 32-bit version of the CVA6 core was developed, which can be implemented by selecting the appropriate configuration options in the design files. However, because the development of the proposed implementation had already started before that time, no attempt was made to transform the design into a 32-bit version. Although doing so would be a trivial task, it would not substantially change the results of this project.

2.6 Related work

In order to justify the MOLEN paradigm as the architecture to implement, other reconfigurable architectures must first be reviewed. Though the original MOLEN dissertation [12] also reviews related work, almost two decades have passed since then. Therefore, a new literature study must be performed to look into alternative reconfigurable designs. In this section, an overview of existing architectures is first presented, after which they are compared to each other and to the MOLEN paradigm.

2.6.1 Overview of existing architectures

The **Fully Pipelined Composable Architecture (FPCA)** [19] is a loosely coupled CGRA, meaning it acts as a separate co-processor instead of being integrated into the processor pipeline. It consists of an array of PE clusters, each containing a set of PEs. It aims to parallelize all operations from the user application, or even dynamically duplicate single operations if resources remain unused.

The **Configurable Compute Accelerator (CCA)** [20] is a CGRA that is designed to efficiently implement many common dataflow subgraphs. Such a subgraph is essentially a slow execution path in the application code. By collapsing these subgraphs into new instructions that can be executed on the CCA, the execution bottlenecks can be removed. This process can happen dynamically at run time, or statically at compile time. The CCA consists of a matrix of two types of functional units. One type performs addition, subtraction and logical operations, whereas the other type can only perform the logical operations.

Blocks [21] is a novel CGRA design that separates the data and control paths. There is a focus on energy efficiency, including the reconfiguration energy overhead. Blocks primarily performs well in applications where data-level parallelism is present. Its CGRA consists of 6 different kinds of functional units, connected by a data network for direct data transfers. The control network allows the instruction fetcher and decoder units to be arbitrarily connected to one or more functional units, allowing Single Instruction, Multiple Data (SIMD) processors to be constructed. As such, VLIW-like instructions can be created and used.

The **Dynamically Specializing Execution Resources (DySER)** [22] architecture is a CGRA focusing on both data-level parallelism and functionality specialization. Functionality specialization means that custom hardware is used to optimize the application-level performance (in the same way as the CCA architecture). The CGRA consists of 6 functional units, that perform integer and floating point operations. DySER was compared to a GPU and SIMD implementation, and outperformed both of them while significantly reducing the energy consumption.

The **Advanced Space-Time Reconfigurable Architecture (ASTRA)** [23] is an FPGA based design that uses CLBs as (fine-grained) building blocks, instead of the coarse-grained blocks of the previous architectures. The unique feature of this architecture is that these ASTRA cells can be configured in either spatial or temporal mode. In temporal mode, the least amount of cells is used, at the cost of a higher latency. In spatial mode, the amount of cells is maximized such that the latency is minimized. Different applications can use the different modes at the same time.

The **Chameleon** [24] architecture is a CGRA that consists of multiple PE arrays. The configuration of the arrays is specified using 32-bit words, that can be translated from high-level instructions. The Chameleon architecture is tailored for MapReduce (big data) applications, in which it is able to achieve significant energy savings.

SmartCell [25] is a power efficient CGRA that is specialized towards data streaming applications. It consists of blocks containing four PEs and a switch box. The cells are connected by a three-level interconnection network. A Serial Peripheral Interface (SPI) is used to reconfigure the cells, enabling low reconfiguration times. On a set of 7 benchmarks, SmartCell was able to use 75% less power than an FPGA implementation.

The **Polymorphic Pipeline Array (PPA)** [26] is a CGRA that focuses on mobile multimedia applications. It consists of an array of simple cores that are coupled using a mesh-style interconnect. Each core can execute its own instruction stream and consists of four PEs. However, multiple cores can be combined into a larger (logical) core, such that fine-grained parallelism can be accommodated more easily. In that case, the logical cores can be used to create pipelines, such that multiple consecutive operations can be performed without additional communication overhead in between. The complexity of this design is high, as schedules need to be generated for the execution, and dynamic hardware allocation is applied.

Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing (HARTMP) [27] is an architecture consisting of Dynamic Adaptive Processors (DAPs). Each DAP is a single-threaded processor, using a reconfigurable datapath.

In this way, thread-level parallelism is available as result of the total amount of DAPs. Instruction-Level Parallelism (ILP) is achieved by means of the reconfigurable datapaths. Larger cores allow for a higher ILP. Despite having different core sizes, a homogeneous ISA is used, hiding the heterogeneity of the cores from the programmers.

The **Dual-Track Coarse-Grained Reconfigurable Architecture (DT-CGRA)** [28] is a CGRA design tailored for stream processing in the machine learning domain. It consists of a set of Stream Buffer Units (SBUs) and a Computing Array (CA). The SBUs cache the input and output data, as well as the intermediate results. The CA forms the actual CGRA and consists of different types of reconfigurable cells, some of which are specialized to calculate interpolations and power functions. The rows of cells are connected with a multi-channel data bus, which in turn are connected to the SBUs using a crossbar.

2.6.2 Comparison

From the presented set of architectures, two observations stand out. The first one is that coarse-grained architectures seem to be the preferred choice in recent designs. This is because they allow for a lower power consumption and higher performance than fine-grained systems. Furthermore, the reconfiguration times can be significantly lower [24]. However, they also limit the flexibility of the system significantly: most designs seem to be focused on optimizing inner loops of user programs, typically present in computations on large datasets. The functional units can be seen as small general-purpose processors. In some cases, they approach regular processor cores, or can be used to construct SIMD cores [21]. Because of this, they mostly improve the performance by exploiting parallelism. However, recent CPUs already consist of multiple cores and (hardware) threads. Often, user applications tend to be sequential in nature, and current multi-core processing remains under-utilized (within single applications). In this situation, adding more parallelism will not be useful. The MOLEN paradigm resolves this issue by not necessarily adding parallelism, but by adding support for any kind of custom operation that can be synthesized to the reconfigurable fabric. This ensures a greatly increased flexibility, without compromising the ability to exploit parallelism. As many parallel units can be implemented in the reconfigurable fabric as allowed by the size of the fabric.

The other observation is that most designs are tailored for specific applications. This might improve the performance and energy efficiency for their respective applications, but reduces their overall usability. Especially when used with a general-purpose processor can this observation be disappointing. By implementing the MOLEN paradigm on a RISC-V processor, the design of this thesis is applicable to a broad range of applications. Fine-grained reconfigurability and the modular design of the RISC-V ISA enable this.

The clear disadvantages of the proposed design are the expected reconfiguration overhead and energy efficiency penalty of using a fine-grained reconfigurable fabric. However, recent FPGAs contain an increasing amount of coarse-grained blocks (such as DSPs). Therefore, these problems have already diminished to a certain degree and are expected to continue to do so in the future.

A final remark on the disadvantages further supports the proposed design as viable architecture: MOLEN does not enforce using fine-grain reconfigurability. All that is defined by the machine organization is that a CCU is available. By making that CCU coarse-grain reconfigurable, the reconfiguration overhead diminishes and the energy efficiency penalty disappears. At the same time, the power of the MOLEN paradigm is preserved, depending on the design of the PEs. The currently proposed, fine-grained design, is an initial working version of the design, that can be expanded in the future to a coarse-grained design. Alternatively, a more efficient fine-grain fabric [29] can be used to increase the energy efficiency and performance.

2.7 Conclusions

In this chapter, relevant background information was provided on reconfigurable architectures, the MOLEN processor, and the RISC-V architecture.

Two main application domains for reconfigurable architectures were first identified: accelerators for **high-performance computing** and specialized or time-critical tasks in **embedded systems**. Reconfigurable architectures can be distinguished in terms of granularity. Fine-grained systems use the smallest building blocks and can be fully configured. Coarse-grained systems are less flexible and use larger building blocks, allowing for increased clock speeds and a greater energy efficiency.

After establishing a basic understanding of reconfigurable architectures, the MOLEN polymorphic processor could be introduced. In this paradigm, a one-time ISA extension introduces reconfigurable microcode ($\rho\mu$-code), which is used to perform reconfigurations and execute custom operations on the reconfigured hardware. This extension mainly consists of a *set* instruction that loads a reconfigurable design, and an *execute* instruction that performs the custom operations. The von Neumann architecture is extended with a reconfigurable processor that performs the custom instructions, an arbiter that passes the instructions to either the core processor or reconfigurable processor, and a number of exchange registers that are used for storing the input and output operands of the custom instructions.

After introducing the MOLEN paradigm, the RISC-V architecture was introduced. This open-source ISA offers multiple base instruction sets and extension instruction sets, thereby making it suitable for a wide range of applications. Support for custom instructions is also included, enabling its use for this project.

Having introduced the RISC-V architecture and the MOLEN paradigm, existing RISC-V processor implementations were compared, such that the most suitable one could be selected for extension with the MOLEN paradigm. The CVA6 core was chosen for this purpose, because it supports the Linux OS, is actively maintained and has a community with extensive documentation.

Before pursuing the actual implementation of the MOLEN paradigm on the RISC-V architecture, related work had to be examined, in order to substantiate the potential benefits of a novel implementation. Several existing reconfigurable implementations were compared. It was observed that most existing architectures are coarse-grained and application-specific. The proposed design improves upon this by allowing a wide range of applications, while not imposing a coarse-grained or fine-grained design.

Design 3

Now that the MOLEN paradigm and RISC-V architecture have been introduced in Chapter 2, this chapter focuses on adapting the MOLEN design to the RISC-V architecture such that it can be implemented in Chapter 4. Specifically, the design is discussed from the perspective of the RISC-V ISA and the reconfigurable microcode ($\rho\mu$-code) of the MOLEN paradigm.

In Section 3.1, the different instruction formats of the RISC-V ISA are introduced. The available opcode space is investigated in Section 3.2. Following that, both the reconfiguration and execution microcode of the MOLEN design are adapted to the RISC-V architecture in Section 3.3. The initial adaptation of the microcode is then shaped into an actual proposed ISA extension in Section 3.4, after which this chapter is concluded in Section 3.5.

3.1 RISC-V instruction formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Table 3.1: 32-bit RISC-V instruction formats. Adapted from the RISC-V ISA specification[3].

Before the MOLEN ISA extension can be designed in the following sections, we first need to examine the main instruction types of the RISC-V instruction set, as depicted in Table 3.1. Suitable instruction formats for the custom instructions can then be identified in the following sections. All instruction types apart from the R4-type are defined by the RV32I base instruction set. The R4-type is defined by the F/D/Q standard extensions, which specify the single-/double-/quad-precision floating point instructions, respectively.

Before the different types can be compared, the following fields are first distinguished in the instruction formats:

- *opcode*: The field that (fully or partially) specifies the operation to perform.
- *rd*: The destination register that the result of the operation should be written to.

- *rs1*, *rs2*, *rs3*: The source registers that supply the operands that the operation is performed on.
- *funct7*, *funct3*, *funct2*: if present, these function values are combined with the opcode to fully specify the operation to perform.
- *imm*: An immediate value that is passed into the instruction itself instead of loaded from a register.

Now, the different instruction types can be explained:

- **R-type (R4-type)**: This type is a register-register instruction. It takes two (R-type) or three (R4-type) source registers to perform the operation on and the destination register to put the result into. An example of an R-type instruction is the RV32I ADD instruction, which adds the values in *rs1* and *rs2* and puts the result in *rd*.
- **I-type**: A register-immediate instruction that takes a 12-bit immediate value, a single source register and the destination register. An example of an I-type instruction is the RV32I ADDI instruction, which adds the sign-extended immediate value *imm* to the value in *rs1* and puts the result in *rd*. The I-type is also used for loads, such as the RV32I LW instruction that loads the value at memory address *rs1* + *imm* (with *imm* being sign-extended) and puts the result in *rd*.
- **S-type**: A store-type instruction that takes a 12-bit immediate value and two source registers. Note that the immediate value is split up across two portions. The reason for that is to allow all other fields to be in the same position as for the other instructions that have those fields. This simplifies the hardware design by allowing for as much overlap in the logic as possible.

  An example of an S-type instruction is the RV32I SW instruction, which stores the value in *rs2* at memory address *rs1* + *imm* (with *imm* being sign-extended).
- **B-type**: A branch-type instruction that is a variation of the S-type instruction. The difference with the S-type is the interpretation of the immediate value. Instead of it being 12 bits like for the S-type, the immediate value is 13 bits, but the lowest bit is an implicit 0. In other words, it is a 12 bit immediate that is left-shifted by one bit. As depicted in Table 3.1, the immediate bits are ordered differently than for the S-type. This maximizes the overlap of the bits with those of the immediate value of the S-type, allowing as much re-use of the same logic as possible. The exception to that is that the bit at position 31 is not bit 11 of the immediate (like for the S-type), but bit 12 instead. This is done because bit 12 is the most significant bit of the immediate, and as such, no changes to the sign-logic are needed compared to that of the S-type.

  An example of a B-type instruction is the RV32I BEQ instruction, which adds the sign-extended immediate value *imm* to the program counter (Program Counter (PC)) if the values in *rs1* and *rs2* are equal. This also reveals why the lowest bit of the immediate value is always 0: we can only jump in multiples of 2 bytes to end up at a potentially valid instruction address. Normal instructions are 32 bits, or 4 bytes, but the compressed instructions that are introduced in Section 3.1 are 16 bits, or 2 bytes.

- **U-type**: This type is an upper immediate instruction. It takes a 20-bit (upper) immediate value and the destination register *rd*. The most basic RV32I example is the LUI instruction, which loads the immediate value into bits 31-12 of *rd* and fills the lowest 12 bits of *rd* with zeroes. In the case of the RV64I instruction set, it is sign-extended to 64 bits.

- **J-type**: A jump-type instruction that is a variation of the U-type instruction. Like for the B-type, the difference can be found in the interpretation of the immediate value: PC-relative jumps can only (viably) target an offset that is a multiple of 2 bytes. Therefore, the immediate value is interpreted as 21-bit value instead of a 20 bit value, with the least significant bit being an implicit 0. Likewise, the bit ordering of the immediate value is chosen to match that of the U-type where possible, while placing the sign bit at the most significant position.

  The only example of the J-type instruction is the RV32I JAL instruction, which adds the sign-extended immediate value *imm* to the PC, and loads the address of the instruction after the jump into *rd*.

3.2 RISC-V opcode space

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

Table 3.2: RV32/64G opcode map, inst[1:0]=11. Adapted from Table 24.1 of the RISC-V ISA specification[3].

After highlighting the different instruction types in the previous section, we first need to investigate the available opcode space for our one-time ISA extension, before proceeding with the design in the following sections. Table 3.2 highlights the RISC-V instructions and instruction groups for the different opcodes. An instruction group refers to a set of operations for which the opcode alone does not fully specify the operation (instead being complemented by one or more *funct* fields).

First, it should be noted that the lowest two bits of instruction (and opcode) are both 1 in this table. Any other variation of the lowest two bits is reserved for the compressed 16-bit instructions that are specified in the “C” standard extension [14]. Since the opcode space defined in that extension is fully exhausted, no custom ISA extension can be added there. As such, only the regular opcode space is investigated.

Inspecting Table 3.2 reveals the *custom-0*, *custom-1*, *custom-2* and *custom-3* opcode spaces. These spaces are reserved for custom extensions and are guaranteed to be avoided by future standard extensions [14]. The exception to that are the future RV128 standard extensions that will occupy the *custom-2* and *custom-3* opcode spaces. It is also possible to identify instructions or instruction groups that are not used in the chosen CPU design, and use those for the custom extension. However, doing so would limit the possibilities for implementing the proposed extension in other micro-architectures. To allow for the most future possibilities, the *custom-0* or *custom-1* opcode space should be chosen. Since no further advantages in favor of either one could be identified, the *custom-0* opcode space is chosen for the proposed design. Therefore, the opcode for the one-time custom extension will be 0001011.

3.3 Microcode design

Following the overview of the RISC-V instruction formats in Section 3.1 and the exploration of the available opcode space in Section 3.2, we can now design the MOLEN reconfigurable microcode ($\rho\mu$-code), as introduced in Section 2.2. This microcode is used for both the act of reconfiguration, as well as the execution of operations on the reconfigured hardware. In order to adapt this microcode to the RISC-V processor, we need to take into account the hardware limitations of the chosen FPGA. Because of their different nature, the reconfiguration microcode and execution microcode are considered separately.

3.3.1 Reconfiguration microcode

As mentioned in Section 2.2, the original MOLEN design [12] proposes a straightforward method of generating the reconfiguration microcode: the bitstream of the reconfigurable partition is split into equal blocks, to which the *set* opcode is then appended to create the micro-instructions that perform the reconfiguration. Such micro-instructions are easily generated from the bitstream.

An underlying assumption in the original work is that (partial) reconfigurable designs can be placed into different areas of the reconfigurable fabric as desired. Multiple reconfigurable designs of various sizes can then be loaded and connected to each other using microcode. However, the way that partial reconfiguration works on the Xilinx FPGA family, is that a partial bitstream is loaded into a matching reconfigurable partition, which is a predefined physical area on the FPGA. It is important to note that a partial bitstream can only be loaded into the reconfigurable partition for which it was generated, and not into any other reconfigurable partition. Therefore, in a design with multiple reconfigurable partitions, a separate partial bitstream has to be generated for each partition that the reconfigurable module is used in. Having the exact same physical dimensions for the reconfigurable partitions does not change this fact. Work has been done to change this and allow for sharing of bitstreams between partitions [30], but such tools are not publicly available yet.

This limitation on the bitstreams and reconfigurable partitions has major implications on the implementation of the MOLEN design: having smaller reconfigurable building blocks (such as full adders) that can be joined into a larger design would require many small reconfigurable partitions. For each of those partitions, the bitstream of each small design would need to be separately generated. At the same time, having multiple small partitions makes it impossible to design larger, more complex building blocks, because they would not fit into a single (small) partition. Furthermore, it was already decided in Chapter 1 to use a single reconfigurable partition for the current design.

An alternative method for the reconfiguration microcode is also mentioned in the original MOLEN design [12]: instead of directly loading the bitstreams, a higher level microcode could be achieved by comparing synthesis results or partial bitstreams and identifying the parts responsible for the reconfiguration of specific areas of the reconfigurable partition. Those areas could then be individually reconfigured on a very small scale. Though theoretically possible, this requires reverse-engineering the bitstreams, which are specific to each separate FPGA family, instead of being generalizable. Because of the complexity of such an endeavor, this approach is not considered feasible for the current work.

A final method for the reconfiguration microcode is to take a more coarse-grain approach: instead of loading a bitstream into a reconfigurable partition, a static set of PEs could take the place of the reconfigurable fabric. Those PEs could then be dynamically configured and chained to achieve the desired functionality. This method circumvents the mentioned problems with the partial bitstreams and reconfigurable partitions. However, it poses a limit on the possible functionalities that can be implemented. Although this method could be viable for implementing specific functionality types (such as matrix multiplication), it is less suitable for the implementation of arbitrary functions. The careful PE design that would be needed to ensure a wider range of functions is outside the scope of this work.

Following the previous observations, it is decided to apply the original proposal of wrapping partial bitstreams into micro-instructions for this project. Because such reconfiguration microcode is merely a shell to transport the partial bitstream, it can even be simplified further. The *set* instruction would then point directly to the address of the partial bitstream, instead of the address of the reconfiguration microcode. This will be further expanded on in Section 3.4.

3.3.2 Execution microcode

After designing the reconfiguration microcode in the previous section, we can now consider the execution microcode, which is also affected by the previously mentioned limitations. Having a single reconfigurable partition, means that a single bitstream should contain all functions that can be used simultaneously. In other words, the reconfiguration flow that is envisioned in [12], where the bitstreams of the required building blocks are reconfigured separately, is not feasible when only a single reconfigurable partition is used.

By extension, this means that the execution microcode, which was envisioned to connect the building blocks to each other and to their inputs and outputs, is also not viable in the current design. Instead, having a single reconfigurable partition implies that all of the required functions should be fully implemented and connected within the reconfigurable design itself. This also means there is no need for the execution microcode to connect the individual components, inputs and outputs. Instead, its only function is to perform the required operations when needed, like regular instructions.

When deciding how to design such microcode, the RISC-V architecture has to be taken into account. Specifically, the dedicated opcode space that is available for custom instructions as introduced in Section 3.2, is very suitable for the design of the (modified) microcode. Instead of having a complete microprogram that needs to be loaded from a control store or from memory, a **nested microcode** can be designed using the available opcode space.

Revisiting the RISC-V instruction formats from Section 3.1, it can be observed that the R-type instruction has a total of 10 bits available in the *funct7* and *funct3* fields that specify the operation to perform. With the opcode being the *custom-0* opcode as decided in Section 3.2, the combined *funct7* and *funct3* fields can be designated as the reconfigurable micro-opcode ($\rho\mu-$opcode), allowing 1024 unique micro-instructions to be defined for each reconfigurable design. If that is not enough, the other custom opcode spaces can be incorporated as well to increase the available space. Contrary to the MOLEN proposal, the custom instructions do take up instruction space in this case. However, the nested micro-instructions are not part of the ISA, because decoding them takes place inside the reconfigurable module, and their meaning is not static or known in advance. Furthermore, the available custom opcode spaces mitigate the problem of not having a well-defined or large enough opcode space available for the custom instructions.

Using the R-type instruction for the nested microcode also means that the source and destination register fields can be reused for the custom instructions. Although a separate exchange register file could be added in the future, the current design will use the main register file due to time constraints.

3.4 Proposed ISA extension

	set	*execute*	*status*
opcode	*custom-0* (0001011)		
rd	reconfiguration result	execution result	reconfiguration status
rs1	bitstream address	input operand 1	-
rs2	bitstream length	input operand 2	-
funct10	1023	$\rho\mu-$opcode	1022

Table 3.3: The proposed RISC-V MOLEN extension, consisting of three R-type instructions. The *funct10* field refers to the combined *funct7* and *funct3* fields.

Combining the observations from Section 3.3, we can now specify the actual ISA extension, as depicted in Table 3.3.

As mentioned in Section 3.3.1, the *set* instruction directly points to the bitstream instead of pointing to (reconfiguration) microcode. This implies that the length of the bitstream must also be passed, in order to determine when the entire bitstream has been loaded. Therefore, two input registers are required, making the R-type instruction the only suitable choice. The (integer) value of 1023 is chosen for the implicit *funct10* field.

Following the reasoning of Section 3.3.2, the *execute* instruction is also an R-type instruction, taking two input operands and the $\rho\mu-$opcode. Its return value is the result of the $\rho\mu-$operation.

In addition to the *set* and *execute* instructions, a third instruction was introduced during the experiments described in Chapter 5: the *status* instruction returns the current reconfiguration status. Having this instruction allows the *set* instruction to run in the background, after which its completion can be monitored using the *status* instruction. In this way, the CPU can continue executing (other) instructions while reconfigurations are being performed in the background. The *break* instruction of the complete MOLEN extension, as described in Section 2.2.1, achieves a similar goal. However, the *break* instruction does not return a status, but instead waits for the preceding (parallel) CPU instructions and reconfigurable instructions to finish. Furthermore, the *status* instruction is only applicable to the reconfiguration (*set* instructions), whereas the *break* instruction is applicable to both reconfiguration and execution microcode (*set* and *execute* instructions).

Although the *status* instruction does not take input operands, it is also chosen to be an R-type instruction, following the *set* and *execute* instructions. If any other type would be used, the proposed instructions would have to be distinguished from each other in the decoder by evaluating the *funct3* field on its own. This would significantly reduce the available $\rho\mu-$opcode space. Therefore, the R-type is used for the *status* instruction. The (integer) value of 1022 is chosen for the implicit *funct10* field, leaving a total of 1022 possible $\rho\mu-$opcodes.

3.5 Conclusions

In this chapter, the design of the MOLEN paradigm was adapted to the RISC-V architecture. First, the RISC-V instruction formats were introduced. These are the R-type and R4-type (register-register format), I-type (immediate format), S-type (store format), B-type (branch format), U-type (upper immediate format) and J-type (jump format). Together with the opcode, the *funct* fields (present in most instruction formats) specify the operation to perform.

After introducing the instruction formats, the RISC-V opcode space was explored. The *custom-0*, *custom-1*, *custom-2* and *custom-3* opcode spaces were identified for implementing custom instructions. From these options, the *custom-0* opcode space was chosen for the implementation of the current ISA extension.

Having established the opcode to use for the ISA extension, the design of the MOLEN $\rho\mu$-code could be adapted to the RISC-V architecture. Due to limitations of the FPGA hardware, the reconfiguration microcode was chosen as a wrapper around the partial bitstream bytes. The execution microcode was adapted into a nested microcode due to the same limitations. The R-type instruction was chosen for the implementation of this microcode, with the *funct7* and *funct3* fields forming the (nested) micro-instruction to perform.

Following the design of the reconfiguration and execution microcode, the specific MOLEN RISC-V ISA extension could then be formulated. The *set* instruction reconfigures the hardware, using the bitstream at the specified address. The *execute* instruction performs operations on the reconfigurable hardware by employing the implicit *funct10* field as (nested) micro-instruction. Furthermore, a *status* instruction was introduced to allow the reconfiguration to be performed in the background. This instruction returns the current reconfiguration status.

4 Implementation

After the MOLEN paradigm was adapted to the RISC-V architecture in Chapter 3, the proposed design can now be implemented. In this chapter, the key hardware components from the MOLEN paradigm are identified and adapted to the CVA6 implementation.

In Section 4.1, an overview of the CVA6 implementation is presented, and the components that need to be added or modified are identified. Following that, the memory interface is discussed in Section 4.2, after which the changes to the instruction decoder are outlined in Section 4.3. Subsequently, the implementation of the MOLEN functional unit is discussed in Section 4.4. The reconfigurable design flow is explained in Section 4.5, after which the software design relating to the reconfigurability is introduced in Section 4.6. Finally, this chapter is concluded in Section 4.7.

4.1 Implementation overview

Figure 4.1 depicts the CVA6 architecture. The following six pipeline stages are identified:

1. **Program Counter Generation (PCGEN)**: In this stage, the program counter is updated based on its current value, the predicted branches and other signals such as exceptions.
2. **Instruction Fetch (IF)**: In this stage, the instruction pointed to by the program counter is fetched from the cache. Compressed (16-bit) instructions are re-aligned. Branch prediction is also performed by this stage.
3. **Instruction Decode (ID)**: This stage decodes the instruction into a scoreboard entry that is used by the consecutive stages. The scoreboard entry keeps track of the input and output registers that are used by the instruction, as well as which functional unit it is executed on. The immediate values are also passed using this structure.
4. **Issue**: In this stage, the scoreboard entries are passed to their respective functional units. The (decoded) instructions are issued in-order, but they can complete out-of-order. Therefore, the scoreboard entries from the previous stage are tracked in a scoreboard to keep track of the available registers and functional units. Handshaking is performed with each functional unit using *ready* and *valid* signals.
5. **Execute**: This stage contains all of the functional units, such as the Arithmetic Logic Unit (ALU) and Load Store Unit (LSU). Each functional unit operates independently using its own *ready*/*valid* handshaking signals. The *ready* signal is set to indicate that the module is available and monitoring its inputs. When the operation is completed, the result is set and the next stage is signaled using a *valid*

signal. Another output, *opcode_supported*, is present to signal the next stage about unsupported or illegal instructions.

6. **Commit**: In this stage, the results from the functional units are written back from the scoreboard into the register file. Exceptions such as illegal instructions are also handled here.

From this architectural overview, the following modifications and additions can be specified:

1. **Decoder**: The decoder must be modified to recognize the MOLEN instructions (*set*/*execute*) and send them to the right functional unit. The required changes to the decoder are described in Section 4.3.
2. **MOLEN unit**: A MOLEN functional unit must be introduced to execute the MOLEN instructions. This means it must be able to perform partial reconfiguration, as well as the execution of nested microcode on a reconfigurable design. The implementation of the MOLEN unit is described in Section 4.4.

Before these modules can be modified or implemented, another area that must first be considered is the memory interface (Section 4.2), because it dictates how the MOLEN unit can be implemented.

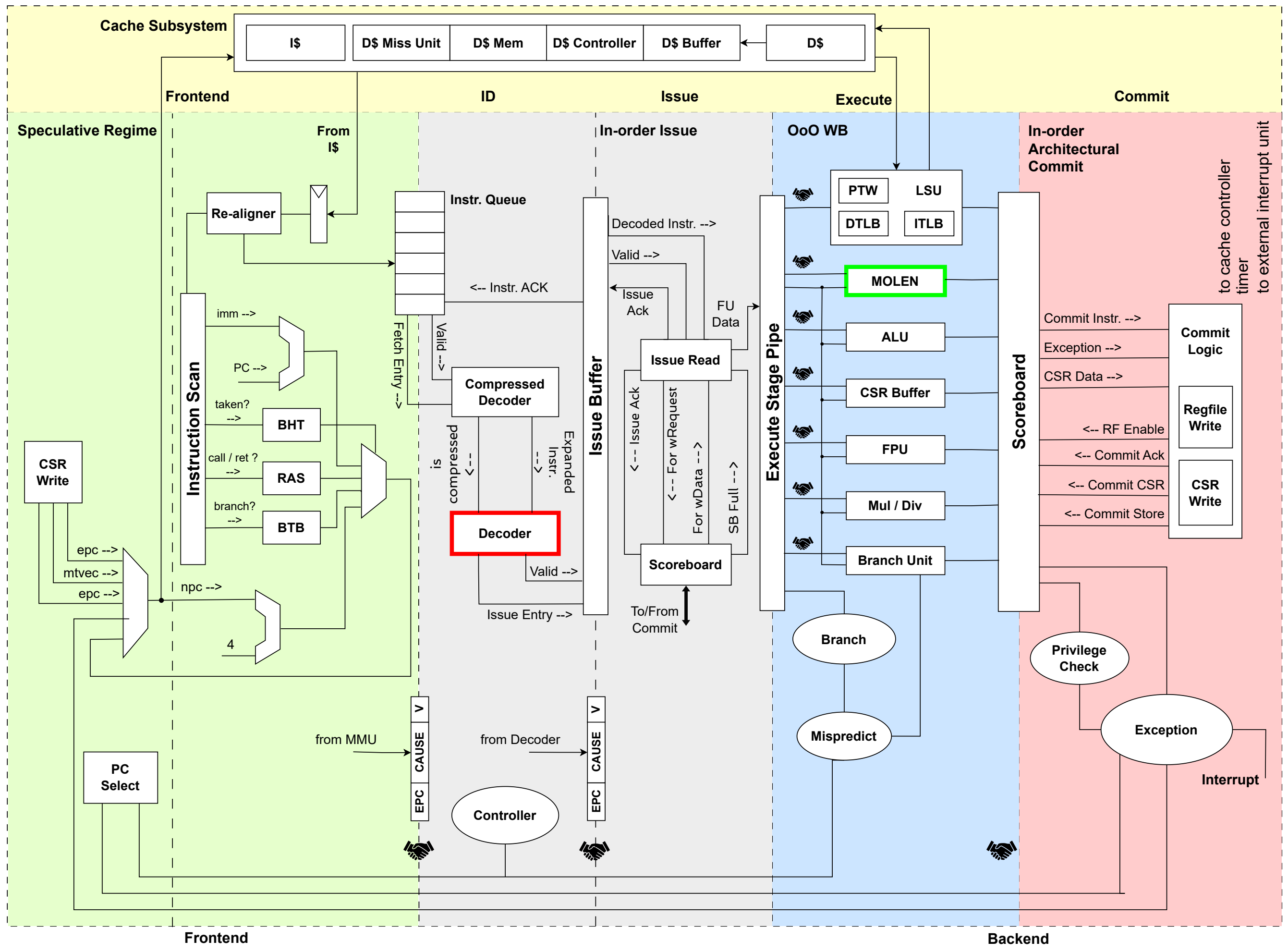

Figure 4.1: The CVA6 architecture with the modified and added components. Image adapted from [2].

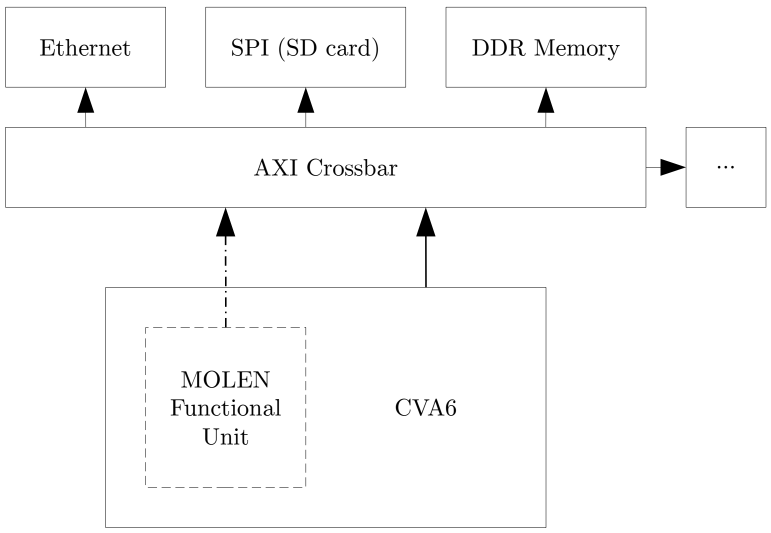

Figure 4.2: The CVA6 interface diagram highlighting the AXI interfaces. The arrows point from the AXI master ports towards the AXI slave ports.

4.2 Memory interface

Figure 4.2 depicts the interface of the CVA6 core with the main memory and the memory-mapped peripherals. The Advanced eXtensible Interface (AXI) protocol is used as communication bus throughout the design. An AXI crossbar forms the backbone of the interconnect, to which all of the (AXI) masters and slaves are connected. In the original design, the CVA6 core is the only master on the crossbar, with the crossbar itself acting as master to all of the slaves (as indicated by the direction of the arrows in Figure 4.2).

In order to extend this design to allow memory access from the MOLEN unit, another AXI master can be added to the crossbar. Alternatively, an additional interconnect could be placed in front of the DDR memory, to allow both the crossbar and MOLEN unit access to the memory. The MOLEN unit would not have access to the peripherals in that case. However, it could be advantageous to access those peripherals from a reconfigurable design, and doing so would increase the available use cases of the reconfigurable module. **As such, the MOLEN unit is added as master on the crossbar**.

The resulting AXI interface can be shared between the partial reconfigurator and the functional unit inside the reconfigurable module, because the memory access from both sources is mutually exclusive. The reconfigurator only needs to access the memory when a reconfiguration is performed, during which time the reconfigurable module cannot access the memory. The details of how the AXI interface is shared are discussed in Section 4.4.3.

In the CVA6 design, some data structures are available to simplify and clarify the implementation of the AXI connections. Specifically, the *ariane_axi::req_t* and *ariane_axi::resp_t* are used to group the signals that are used for a read/write request, and those that are used for a read/write response, respectively. These two data structures form the only access point to the AXI bus from within the masters and slaves. This simplifies the design, because the MOLEN unit only needs these two structures as input/output to provide access to the main memory and peripherals.

4.3 Instruction decoder

The instruction decoder (*decoder.sv*), which is depicted in Figure 4.1, is responsible for transforming the 32-bit instructions into scoreboard entries. This transformation consists of assigning the functional unit that the instruction should be executed on, as well as the specific operation that should be performed. The source registers, destination registers and immediate value are also passed in this process (if present for the instruction type).

In order to extend the decoder with the MOLEN instructions, the opcode of the current instruction is compared to the *custom-0* opcode. In case of a match, the MOLEN functional unit is assigned to the scoreboard entry of the instruction. The *funct7* and *funct3* fields are evaluated to determine the operation to perform. If all bits of this implicit *funct10* field are set (integer value 1023), the *MOLS* (MOLEN *set*) operation is passed into the scoreboard entry. If the *funct10* field has integer value 1022, the *MOLST* (MOLEN *status*) operation is passed. In all other cases, the *MOLE* (MOLEN *execute*) operation is passed. Additionally, the *funct10* value is passed into the immediate field of the scoreboard entry.

4.4 MOLEN functional unit

The MOLEN functional unit (*molen.sv*), depicted in Figure 4.3, takes the decoded MOLEN instructions from the issue stage as scoreboard entries. Having two main types of MOLEN instructions leads to a logical division of the functional unit into two modules:

- **The partial reconfigurator** (*icap_reconfigurator.sv*). This module is responsible for performing the partial reconfiguration when a MOLEN *set* instruction is issued. The implementation of the partial reconfigurator is discussed in Section 4.4.1.
- **The MOLEN reconfigurable module** (*molen_rm*). This module can be loaded with a custom (partial) design. That means it is empty by default and only provides the interface between the static and reconfigurable design. It is assigned a specific reconfigurable partition in the design flow. This module performs the MOLEN *execute* instructions. Its implementation is discussed in Section 4.4.2.

These two modules do not account for the *status* instruction, which is handled by a (local) STATUS Finite State Machine (FSM) instead. This FSM is responsible for returning the reconfiguration result from the partial reconfigurator. This result cannot be returned by the reconfigurator itself, because the *set* instruction returns before the reconfiguration is finished. Additionally, the STATUS FSM is responsible for returning error values if a *set* or *execute* instruction is issued while a reconfiguration is taking place.

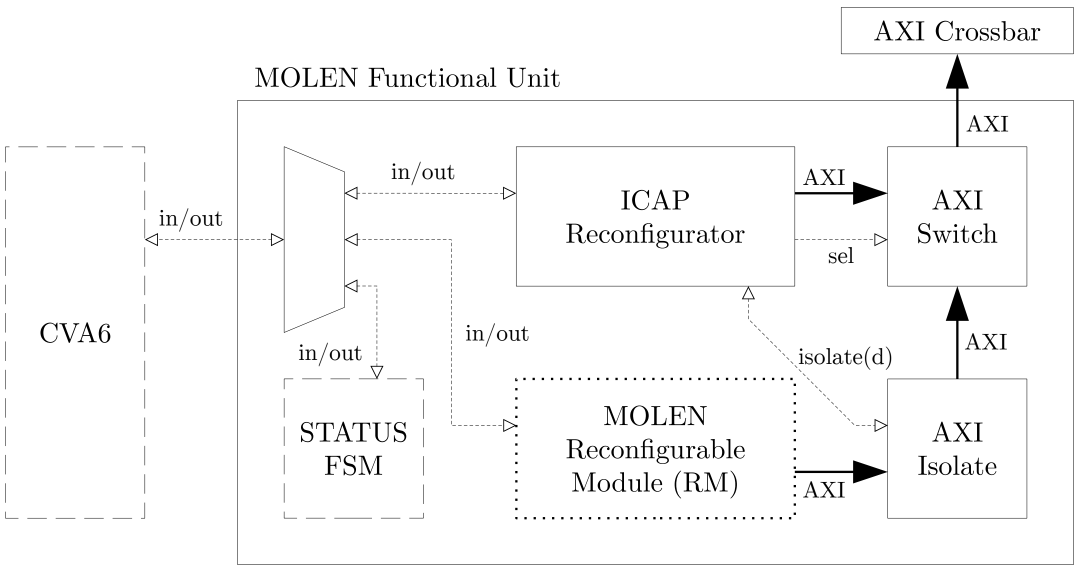

Figure 4.3: An overview of the implementation of the MOLEN functional unit.

Because of the clear separation into modules, the only other functionality inside the MOLEN unit itself is to handle the control signals to and from those modules. Specifically, the handshaking that is performed with the issue stage needs to be converted into a separate handshaking with the two submodules and the FSM. The same *ready/valid* handshaking as for the MOLEN unit is used for this purpose, resulting in the following signals:

- *reconf_valid_op*, *rm_valid_op* and *status_valid_op* are used to signal from the MOLEN unit to the reconfigurator, reconfigurable module or STATUS FSM that the inputs to that respective module (or FSM) are valid. To achieve this, the *operator* field of the scoreboard entry is evaluated. If it is set to the *MOLS* (MOLEN *set*) operation, the *valid_i* signal into the MOLEN unit is passed to the *reconf_valid_op* signal. Likewise, it is passed to *rm_valid_op* when the *operator* is set to the *MOLE* (MOLEN *execute*) operation. Finally, it is passed to *status_valid_op* when the *operator* is set to the *MOLST* (MOLEN *status*) operation, or when a reconfiguration is currently being performed.

- *reconf_valid*, *rm_valid* and *status_valid* are used to signal from the respective module or FSM to the MOLEN unit that its outputs are valid. If one of these signals is asserted, the *molen_valid_o* signal is asserted in turn.

- *reconf_ready*, *rm_ready* and *status_ready* are used to signal from the respective module or FSM to the MOLEN unit that it is ready to accept inputs. If *status_ready* is asserted, or if both *reconf_ready* and *rm_ready* are asserted, the *molen_ready_o* signal is asserted in turn.

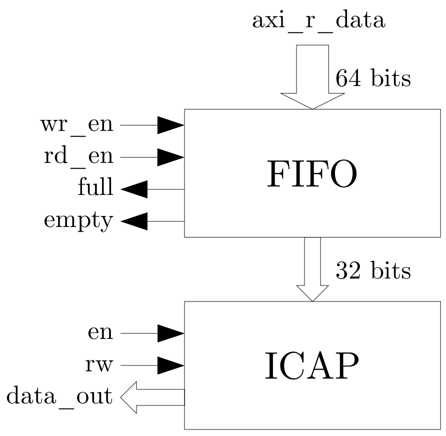

Figure 4.4: An overview of the implementation of the partial reconfigurator.

In Figure 4.3, the switching of these signals is simplified as a multiplexer that connects to the input and output signals.

In this way, a logical separation of the MOLEN unit into submodules is achieved. The implementation of the partial reconfigurator and MOLEN reconfigurable module can now be discussed in the following sections.

4.4.1 Partial reconfigurator

The partial reconfigurator (*icap_reconfigurator.sv*), depicted in Figure 4.4, is responsible for transferring the partial bitstreams into the reconfigurable partition. In order to achieve this, the Internal Configuration Access Port (ICAP) is used. This port allows partial bitstreams to be loaded from inside a running design on the FPGA, as opposed to an external source. The partial bitfile contains all of the required information about the reconfigurable partition and reconfigurable design, which means it can be directly fed into the ICAP. Therefore, the main function of the reconfigurator is to retrieve the partial bitstream from the memory and feed it into the ICAP.

In order to access the ICAP from the hardware design, the *ICAPE2* primitive is used. This primitive defines a 32-bit input port, a 32-bit output port, an enable input pin and a read/write select input pin. Using the input port and enable pin, the bitstream can be written to the ICAP one (32-bit) word at a time. The output port is used to track the current state of the reconfiguration and detect potential reconfiguration errors. The read/write select pin is kept low, because the design only writes to the ICAP.

Because it would be very inefficient to retrieve each 32-bit (half-)word separately from the memory and feed it into the ICAP, a data buffer should be used. A First-In-First-Out (FIFO)-queue is used for this purpose, because it buffers the data while preserving the order. The input and output port widths can be set individually, allowing full 64-bit words being written into the FIFO, while writing 32-bit half-words out of it. Feeding data into the ICAP is then simplified to connecting the output of the FIFO with

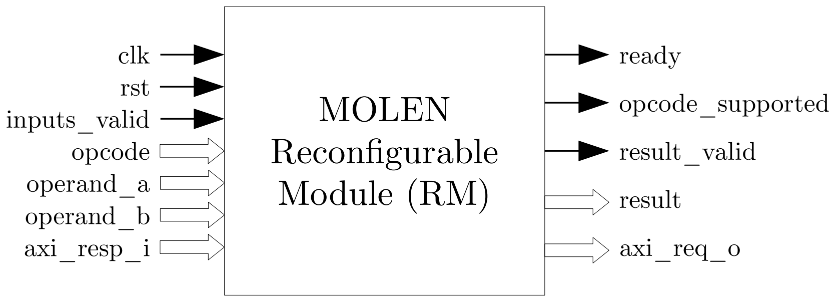

Figure 4.5: The MOLEN reconfigurable module, highlighting the inputs and outputs.

the input of the ICAP and using the *fifo_empty* signal to enable or disable the ICAP.

With the FIFO in place, all that needs to be done is to fill it with the partial bitstream. To achieve this, the AXI bus is used to fetch the data from the address range specified in the *set* instruction. To reduce as much overhead as possible, burst reading is used to request 256 (64-bit) words at once. Once the first word arrives, the next ones are available in each consecutive clock cycle. The FIFO size is chosen to accommodate two full burst reads, or 4 KiB. In the current design, this should ensure that the ICAP is never stalled to wait for new data.

The ICAP of the Xilinx Virtex-7 FPGA family supports a maximum clock speed of 100 MHz [31]. By using separate input and output clocks for the FIFO, the ICAP can be used at its maximum frequency, while the remainder of the design uses the 50 MHz clock. Because the data width of the ICAP is half as large as that of the FIFO input, it should still be able to receive data at a fast enough rate for the 100 MHz clock to be useful.

4.4.2 MOLEN reconfigurable module

The reconfigurable module (*molen_rm*), depicted in Figure 4.3 and Figure 4.5, is logically split into two parts:

- **The interface definition**. This definition specifies the (physical) input and output connections between the reconfigurable module and the CVA6 design. Each design intended for the MOLEN reconfigurable module must adhere to the specified interface. The CVA6 design must also adhere to this interface.

- **The stand-alone implementation**. Each reconfigurable design provides an implementation of the (previous) interface, of which the corresponding partial bitstream can then be generated. When the partial bitstream is loaded into the FPGA, the implemented module is placed in the corresponding reconfigurable partition. The reconfigurable design flow that is responsible for generating the full and partial bitstreams is discussed in Section 4.5.

The interface definition specifies the following inputs, as depicted on the left-hand side of Figure 4.5:

- *clk*: A (50 MHz) clock.
- *rst*: An active-high reset.
- *inputs_valid*: A handshaking signal to specify that the inputs are valid.
- *opcode*: The 10-bit micro-opcode (*funct10*) specified in Section 3.4.
- *operand_a*: The first 64-bit input operand.
- *operand_b*: The second 64-bit input operand
- *axi_resp_i*: The AXI response as introduced in Section 4.2.

The interface definition specifies the following outputs, as depicted on the right-hand side of Figure 4.5:

- *ready*: A handshaking signal to specify that the module is ready to accept inputs.
- *opcode_supported*: A status signal that specifies whether or not the micro-opcode is supported by the reconfigurable module.
- *result*: The 64-bit result of the operation.
- *result_valid*: A handshaking signal that specifies that the result is valid.
- *axi_req_o*: The AXI request as introduced in Section 4.2.

4.4.3 AXI interface

The MOLEN unit utilizes a single AXI interface that is shared between the partial reconfigurator and reconfigurable module, as discussed in Section 4.2. Figure 4.3 depicts the implementation of this design, which consists of two parts:

- **The AXI Switch module**: A simple 2-to-1 multiplexer that selects either the AXI signals from the reconfigurator or reconfigurable module, based on the *sel* input.
- **The AXI Isolate module**: Upon receiving the *isolate* signal, this module blocks new AXI transactions from the reconfigurable module and waits for any ongoing transaction to complete. Following that, the *isolated* signal is raised to indicate that the transactions on the bus have been safely terminated. Terminating the bus in this way before switching is vital to the correct functioning of the design. Switching during an ongoing transaction could stall the bus and prevent the (newly loaded) reconfigurable module from accessing the memory. It should be noted that termination of the ongoing transactions is always needed when loading a new partial design, even if the reconfigurable module would have its own AXI bus.

Both of these modules are already available in the CVA6 design and were not re-implemented. Reusing the existing code in this way simplifies the design and reduces the required maintenance in the future.

Because the reconfiguration is performed by the partial reconfigurator, the AXI Switch and Isolate modules are managed from there. When performing a reconfiguration, the *isolate* signal is raised, after which the partial reconfigurator waits for the *isolated* signal to go high. The *sel* signal is then used to switch the outgoing AXI interface to that of the reconfigurator. The reconfiguration can then be performed, after which the *sel* signal is used to select the AXI interface of the (new) reconfigurable module. A second Isolate module for the reconfigurator is not required, because it is known inside the reconfigurator when the AXI transactions originating there are completed.

4.5 Reconfigurable design flow

Having discussed the implementation of the CVA6 MOLEN design in the previous sections, the only remaining hardware consideration is the design flow that allows the full and partial bitstreams to be generated. Although this flow is specific to the Xilinx family of FPGAs and is not a main contribution of this work, an overview of this process is included for completeness. A detailed discussion of this process can be found in the Partial Reconfiguration Guide from Xilinx [32].

The design flow is based on the original TCL-scripted flow of the CVA6 design [33] and is modified to allow partial bitstreams to be generated. Globally speaking, there are three basic steps involved in both of these design flows:

- **Synthesis**: During this step, the Register-Transfer Level (RTL) abstraction of the (System)Verilog design files is converted into a netlist containing the basic building blocks that are present on the FPGA, and how they are connected.
- **Implementation**: During this step, the components from the netlist of the synthesized design are placed and routed. The result describes the physical implementation on the FPGA.
- **Bitstream generation**: In this step, the implemented design is converted into a bitstream file that can be loaded onto the FPGA. A full bitstream describes an entire design (including the reconfigurable module), whereas a partial bitstream describes only the reconfigurable part of the design.

Two separate flows, depicted in Figure 4.6, are distinguished to avoid the need to regenerate the entire design for each reconfigurable module:

- **The full design**: in this flow, which is executed only once, the full bitstream is generated. An initial reconfigurable module is required to allow the implementation to succeed. Although the specific function of this initial reconfigurable module is irrelevant, a simple multiplier is used for this purpose in this work. The resulting full bitstream is loaded onto the FPGA, whereas the resulting partial bitstream can be discarded. A separate, **static** implemented design is derived from the initial implemented design by removing the reconfigurable module. The blank area that is

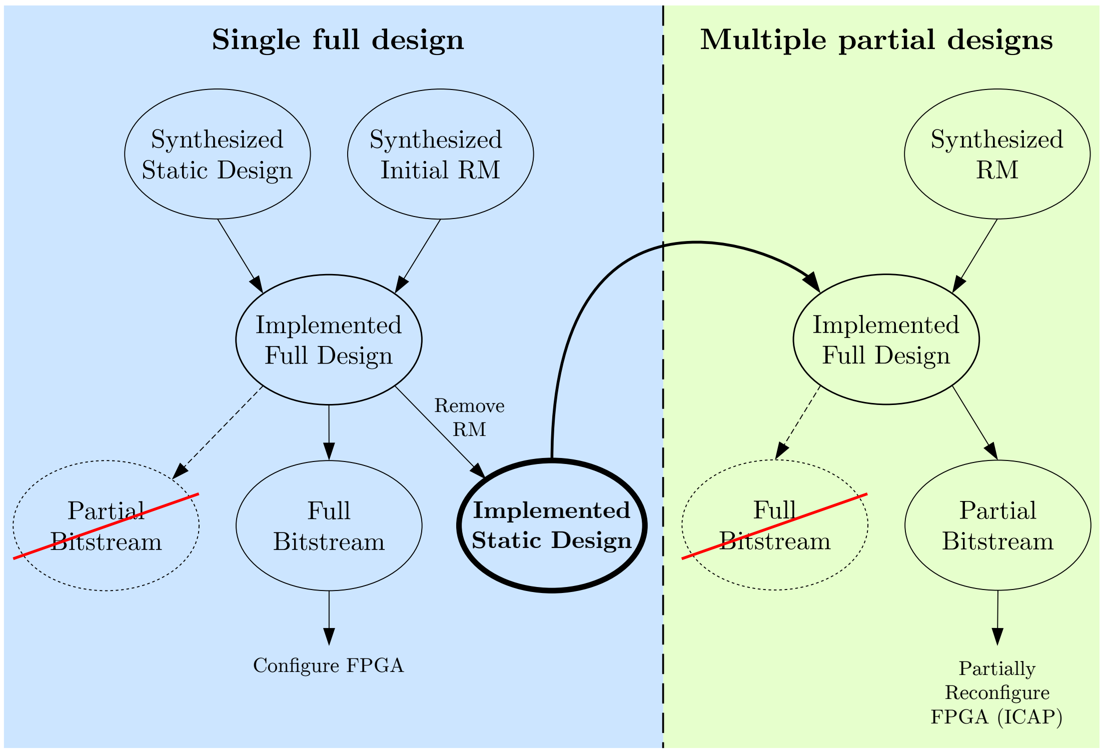

Figure 4.6: A simplified overview of the two design flows for a partially reconfigurable design.

left in place of the reconfigurable module, in which none of the inputs and outputs are connected, is referred to as a black box [32].

- **The partial design**: in this flow, the implemented **static** design from the full design flow is combined with the synthesized design of the current reconfigurable module, resulting in a full implementation. The black box of the static design is filled with the implementation of the current reconfigurable module. From this implementation, the full and partial bitstreams are then generated. The full bitstream from this step is not typically used, because the initial full design should already be loaded onto the FPGA during this phase. The partial design flow is executed for each reconfigurable module that is designed.

For the current design, both of these flows are implemented using a TCL-script, where the full design flow is modified from the original script of the CVA6 project. The script for the partial design flow takes the name of the reconfigurable module as command-line argument, such that any reconfigurable module can be designed and generated, while only requiring the full design to be implemented once.

4.6 Software design

Now that the hardware design considerations have been discussed in the previous sections, the software design can be considered. The basic functionality that is performed by the software consists of two parts:

- Sharing data, such as the partial bitstream, with the MOLEN unit.
- Executing the custom *set/execute/status* instructions that are not part of the RISC-V ISA.

Because the MOLEN unit has access to the AXI crossbar, sharing data becomes a matter of copying it to a memory location that the MOLEN unit can access. To avoid overwriting any of the (virtual) memory of the OS, a separate DMA region is defined at the end of the Random Access Memory (RAM). Caching is disabled for this region, such that output data that is written back from the MOLEN unit can be read correctly. The OS is informed of the DMA region by reducing the memory size in the Linux device tree (*fpga/src/bootrom/ariane.dts*). In this work, a DMA size of 64 MiB has been chosen, but it can easily be expanded if needed. Having defined the DMA region, sharing data becomes a matter of copying it to this region. This can be achieved in the C programming language by using the *open()*, *mmap()* and *memcpy()* functions.

In order to prevent the need for modifications to the RISC-V toolchain when adding custom instructions, the RISC-V assembler supports the *.insn* directive. Using this directive, any (known or unknown) instruction can be specified by supplying the individual fields of the instruction. Each of the instruction formats introduced in Section 3.1 can be specified in this way. However, the *set*, *execute* and *status* instructions are all R-type instructions, as decided in Section 3.4. These instructions can be formatted using the *.insn* directive in the following way:

```
.insn r opcode, func3, func7, rd, rs1, rs2
```

The *opcode* field, which equals the *custom-0* opcode for both instructions, can be specified by name (*CUSTOM_0*), instead of having to specify its value directly. When using the C programming language, the *asm()* function can be used to incorporate the *.insn* directive and set its fields according to variables in the C program. This is achieved in the following way for the *execute* instruction:

```
#define molen_execute(addr, len, uop) ({ \
    int retval = 0, _a = (addr), _b = (len); \
    asm volatile (".insn r CUSTOM_0, " STR(GET_FUNCT3(uop)) \
    ", " STR(GET_FUNCT7(uop)) ", %0, %1, %2\n" \
        : "=r"(retval) \
        : "r"(_a), "r"(_b) \
    ); \
    retval; \
})
```

Here, *uop* is the (10-bit) $\rho\mu$-opcode that specifies the operation to perform. *GET_FUNCT3* and *GET_FUNCT7* are defined as simple bitwise operations to extract the *funct3* and *funct7* fields from the $\rho\mu$-opcode, respectively:

```
#define GET_FUNCT3(uop) ((uop) & 0x7)
#define GET_FUNCT7(uop) ((uop) >> 3)
```

The *STR* function is part of a two-level macro to convert a non-string constant to a string constant, following the GCC C Preprocessor documentation [34]:

```
#define STR_(X) #X
#define STR(X) STR_(X)
```

In this way, the *execute* instruction can be made available for use in C programs, taking regular function arguments for the two inputs and making the output available as return value. The *set* and *status* instructions can then be simplified by a call to the *execute* function with the $\rho\mu$-opcode set to its corresponding value from Section 3.4:

```
inline int molen_set(int address, int length) {
    return molen_execute(address, length, UOP_MOLEN_SET);
}

inline int molen_status() {
    return molen_execute(0, 0, UOP_MOLEN_STATUS);
}
```

4.7 Conclusions

In this chapter, the MOLEN design from Chapter 3 was implemented on the CVA6 processor. After giving an overview of the CVA6 architecture, the following required modifications were identified to implement the MOLEN paradigm: the decoder had to be modified to handle the MOLEN instructions correctly, and a MOLEN functional unit had to be introduced to execute the new instructions. Additionally, it was decided that the MOLEN unit should be added as master on the AXI crossbar, after outlining the CVA6 memory organization.

The decoder was adapted in the following way: first, the opcode was compared to the *custom-0* opcode, after which the *funct7* and *funct3* fields were evaluated for matching the *set* or *execute* operations. In case of an *execute* instruction, the *funct7* and *funct3* fields were passed to the next pipeline stage as immediate value.

The MOLEN functional unit consists of a partial reconfigurator and a MOLEN reconfigurable module. The partial reconfigurator controls the ICAP to reconfigure the FPGA with a partial bitstream that it fetches from the specified memory address. A FIFO-queue is used as buffer to streamline this process. The MOLEN reconfigurable module consists of an interface definition that specifies the inputs and outputs, and an implementation of that interface that executes the actual custom micro-instructions. The AXI interface of the MOLEN functional unit is shared between the partial reconfigurator and MOLEN reconfigurable module, because simultaneous memory access from both sources cannot occur. An FSM inside the MOLEN functional unit is responsible for handling the *status* instructions, as well as the other two instructions when a reconfiguration is being performed.

In order to accommodate the partial reconfiguration inside the CVA6 FPGA design, two separate design flows were implemented. The **full design flow** is performed once and results in a full bitstream containing the CVA6 processor and a single reconfigurable module. The **partial design flow** is performed for each reconfigurable module and results in the partial bitstream that is programmed into the FPGA from within the MOLEN functional unit (when a *set* instruction is issued).

As a final part of the implementation, the software interface with the MOLEN extension must be considered. In order to share data, such as partial bitstreams, with the MOLEN functional unit, a DMA region is defined at the end of the main memory. Executing the MOLEN instructions is simplified by the presence of the *.insn* RISC-V assembler directive, which allows for the execution of custom instructions. In the C programming language, a *molen_execute* wrapper function can be defined using this directive. The *set* and *status* instructions can then be executed by specifying the corresponding $\rho\mu-$opcode for the*molen_execute* function call.

Results 5

Having designed and implemented the proposed design in Chapter 3 and Chapter 4, the implemented design can now be tested for functionality and evaluated in terms of its performance and resource utilization.

After introducing the test setup that is used for the experiments of this chapter in Section 5.1, the following tests and benchmarks are performed:

- **Functional tests** are performed in Section 5.2 to verify the correct behavior of the implementation in a range of circumstances.
- **FPGA synthesis results** are discussed in Section 5.3 as an initial comparison of the MOLEN CVA6 implementation to the base CVA6 implementation.
- The **reconfiguration performance benchmark** of Section 5.4 evaluates the reconfiguration latency that is achieved by the proposed implementation.
- A **matrix multiplication benchmark** is performed in Section 5.5 to evaluate the ability of the MOLEN CVA6 implementation to hide the reconfiguration delay.

Following these experiments, the results are discussed in Section 5.6, after which this chapter is concluded in Section 5.7.

5.1 Test setup

In order to test the functionality of the design and perform experiments, the following test setup is used:

- The hardware platform that the design is tested on is the Xilinx VC707 evaluation kit [17]. It is built around the Virtex-7 XC7VX485T-2FFG1761C FPGA. Although this board is not officially supported by the CVA6 project, experimental support is provided.
- The software suite that is used for the synthesis of the design is Vivado 2018.2 [35]. This version was chosen because it was tested by the CVA6 project [36]. It is also used for simulation of the design.
- The CVA6 version that the design of this work is based on is version 5.0.1, released on March 25, 2024 [33].
- To compile the C code that is executed by the design, version 10.2.0 of the GNU RISC-V toolchain is used [37], containing the GCC compiler and other tools.

- A pre-built Linux image, provided by the CVA6 project, is used as OS in which the tests and benchmarks are performed. Using an OS simplifies the development and execution of these tests, because of its serial terminal interface and its support for accessing the SD card on the board. Furthermore, the overhead that this specific OS causes is limited due to its minimal design.
- Measurements that are generated by the execution of software programs, are repeated 20 times and averaged, in order to mitigate the potential interference of the execution of the OS itself.
- Time is measured in the software benchmarks by counting the clock ticks (under a known clock frequency). Ticks are measured using the RISC-V *cycle* performance counter, accessible through Control and Status Registers (CSRs) [3].

5.2 Functional tests

Test name	**Purpose**	**Outcome**
MOLEN unit test (simulation)	Verify basic *set*, *status* and *execute* behavior	✓
FPGA reconfiguration test		✓
Invalid SYNC test	Verify robustness against invalid bitstream files	✓
Invalid checksum test		✓
set → *set* test	Verify robustness against user error	✓
set → *execute* test		✓

Table 5.1: Overview of the functional tests that were performed on the implemented design.

Before the performance of the design is analyzed in the following sections, the implementation was tested in terms of functionality first. An overview of the functional tests that were performed is presented in Table 5.1.

Instead of testing the complete implementation at once, the MOLEN functional unit (discussed in Section 4.4.2) was first tested on its own. In order to do so, a test bench was developed around the MOLEN unit and a ROM containing a bitstream file. The test bench could then be simulated using tools from the Vivado suite, resulting in waveforms that were inspected for correctness.

No simulation model of the ICAP was available, which meant the actual partial reconfiguration could only be tested on the FPGA. To achieve this, the MOLEN unit was first tested on the FPGA without connecting the ICAP port, using the Xilinx Integrated Logic Analyzer (ILA) to inspect the waveforms of the relevant signals. Once those signals demonstrated the correct behavior, the ICAP port was connected and the partial reconfiguration tested. Multiple simple implementations were created for the reconfigurable partition, such as a multiplier and a divider. Matching software was developed to execute the new MOLEN instructions: a *load-bitstream* application was created to retrieve a user-specified bitstream file from the SD card, place it into the main memory and trigger a reconfiguration using the MOLEN *set* instruction. Each separate design of the reconfigurable partition was given its own application performing the corresponding MOLEN *execute* instruction on the user-specified inputs. This way, all basic functions of the design could be tested.

To increase the robustness of the design, the reconfiguration was also tested using bitstream files that were corrupted or malformed in the following ways:

- **Invalid *SYNC* keyword**: reconfiguration starts when a special synchronization keyword is detected. By modifying the value of this keyword in the bitstream file, the (correct) *SYNC* keyword is never encountered and no reconfiguration is performed. The MOLEN *status* instruction should return an error value that reflects this.

- **Invalid checksum**: Near the end of the partial reconfiguration, the CRC checksum of the written configuration data frames is calculated and compared with a CRC value in the bitstream file. In case of a mismatch, the reconfiguration is considered to have failed, and the reconfigurable partition is then forcibly blanked by the FPGA. Attempting to perform $\rho\mu-$instructions after this event would stall the CPU, requiring a hard reset. The MOLEN *status* instruction should return an error value that reflects this, and also prevent any $\rho\mu-$instructions from being passed to the reconfigurable module.

Other than invalid bitstream files, erroneous usage of the MOLEN instructions was also tested. Specifically, the reconfiguration would be triggered, after which another MOLEN (*set* or *execute*) instruction would immediately be executed, instead of waiting for the reconfiguration to complete using the *status* instruction. In the case of a consecutive *set* instruction, the instruction should be ignored and an error value should be returned. In the case of a consecutive *execute* instruction, an exception should be raised, signaling an illegal instruction.

After testing all of these edge cases and updating the design to handle them correctly, the implementation was considered to be fully functional.

5.3 FPGA synthesis results

Before proceeding to the performance benchmarks in the following sections, the performance and efficiency of the implementation are first evaluated from the perspective of the FPGA synthesis. Specifically, the timing results and resource utilization of the MOLEN CVA6 implementation are compared to those of the base CVA6 implementation.

5.3.1 Timing results

An overview of the timing results is given in Table 5.2. The base CVA6 design without modifications could be run at 50 and 60 MHz without violating any timing constraints. When increasing the clock speed to 70 MHz, the timing constraints are no longer met, with a Worst Negative Slack (WNS) of 0.722 ns. The main critical paths responsible for failing the timing constraints were all originating in the CSR register file and going to the instruction cache. Even though the Floating Point Unit (FPU) was not in the

Configuration	**Clock Speed (MHz)**				
	50	55	60	70	71
CVA6	✓	✓	✓	-	-
CVA6-molen	✓	✓	-	-	-
CVA6 w/o FPU	✓	✓	✓	✓	-
CVA6-molen w/o FPU	✓	✓	✓	✓	-

Table 5.2: Timing results for the base CVA6 and MOLEN CVA6 implementations. A checkmark indicates that all timing constraints are met.

critical path, disabling it resulted in the timing constraints being met once again. This may be explained by the reduction of the area allowing the source and destination of the critical paths to be physically closer together.

Further pushing the clock speed to 71 MHz (without FPU) once again caused the timing constraints to fail, this time due to critical paths from the CSR register file to the LSU.

Attempting to run CVA6 design with the proposed MOLEN extension (with FPU) resulted in the timing constraints being met up to 55 MHz, as opposed to the 60 MHz of the base CVA6 design. This was caused by the same critical paths from the CSR register file to the LSU. However, by disabling the FPU again and reducing the area of the MOLEN reconfigurable partition, the timing constraints could be met for frequencies up to 70 MHz, matching the maximum frequency of the base CVA6 design. If the reconfigurable partition would be implemented on a separate FPGA in future work, this restriction on the reconfigurable partition can be lifted.

From these results, it can be concluded that the proposed MOLEN extension does not impact the critical path, other than by the area that the reconfigurable partition consumes.

5.3.2 Resource utilization

	CVA6	CVA6-molen	MOLEN Unit
Slice LUTs	75682	86314	2111
LUT as Logic	73055	83679	2108
LUT as Memory	2627	2635	3
Slice Registers	49485	55937	431
F7 Muxes	2922	3659	0
F8 Muxes	629	778	0

Table 5.3: Resource utilization of the base CVA6 design and the modified design.

In order to obtain FPGA resource consumption results, the utilization reports from the synthesis tool are used. In the case of the design with the MOLEN extension, the area of the implemented static design (Figure 4.6) is used to exclude the resources inside the reconfigurable partition. Other than the two complete designs, the resource utilization of only the MOLEN unit, *i_molen*, inside the full design is also included. The results are listed in Table 5.3.

Interestingly, the MOLEN design uses significantly more Lookup Tables (LUTs) (14%) and registers (13%) than the base CVA6 design. However, the MOLEN unit does not account for this difference, with its 2111 LUTs and 431 registers. It is also unlikely that the small modification to the decoder unit causes this difference. Instead, it is expected that the area that is reserved for the reconfigurable partition causes the placement and routing becomes more complex, requiring more resources to meet the timing constraints.

5.4 Reconfiguration performance benchmark

After establishing the functional correctness of the design and measuring the FPGA synthesis results in the previous sections, we can now perform the first benchmark. First, it should be noted that the main performance metric of the proposed design is not the speedup of any specific functionality that can be implemented using the reconfigurable micro-instructions. The performance of such functionalities would be similar to those of other FPGA accelerators that do not employ the MOLEN paradigm. Consequently, the resulting measurements would demonstrate the performance of the implemented accelerator instead of the performance of the MOLEN implementation itself.

Instead, one defining performance metric is the reconfiguration time in relation to its theoretical limit. In the remainder of this section, the specifics of the reconfiguration benchmark are first outlined, after which the results are discussed.

5.4.1 Benchmark description

In order to substantiate any measurements relating to the reconfiguration performance, we must first estimate the lower limit of the reconfiguration latency on the specific FPGA architecture that is used for the implementation of this thesis.

On the Virtex-7 FPGA family that is used for the implementation, the reconfiguration time is directly proportional to the size of the partial bitstream file. Bitstream compression can be used by the synthesis tool to reduce the size of the resulting bitstream file. The FPGA does not need any time to decompress the bitstream, because of the compression method: the multiple frame write feature [38] is used to write identical consecutive configuration frames to the FPGA fabric in parallel, allowing all but the first identical frame to be removed from the bitstream. Additionally, any delays that are required for the internal configuration port (ICAP) to perform its tasks, are incorporated into the bitstream file as no-ops. As such, the reconfiguration time is given by the time it takes for all bytes of the partial bitstream file to be sent to the ICAP.

The theoretical minimal reconfiguration time can now be calculated as a function of the bitstream file size and the clock frequency. Because the input port of the ICAP is 32 bits (4 bytes) wide, the minimal reconfiguration time t_r is given by the following equation:

$$t_r = \frac{s_b}{4 \cdot f_{clk}}$$

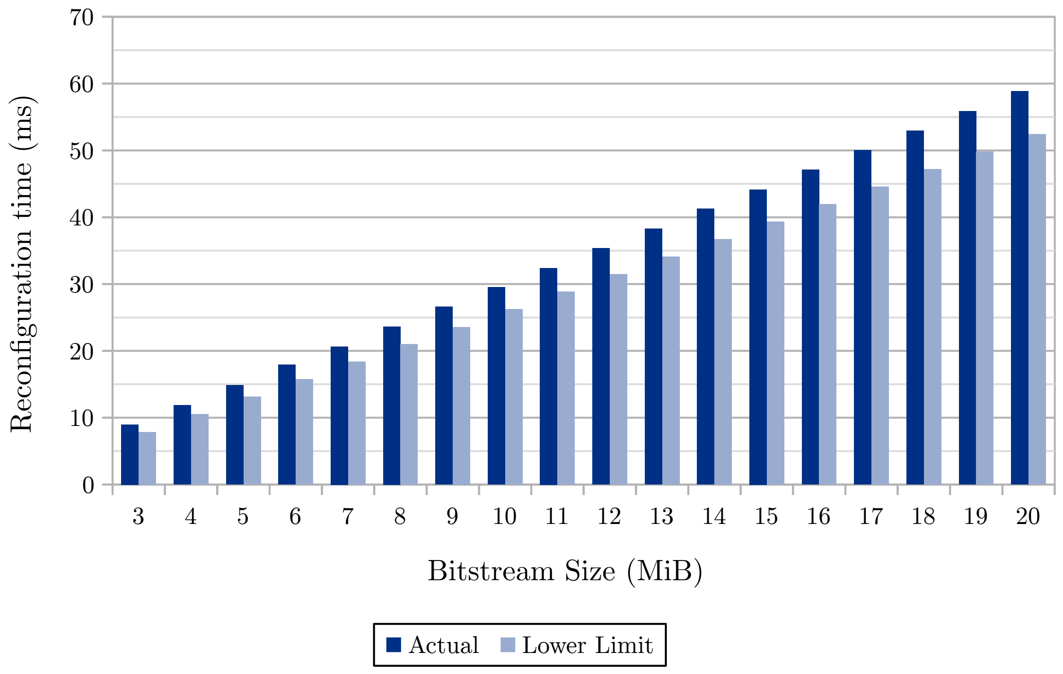

Figure 5.1: Reconfiguration times compared with the theoretical lower limit, running at 50 MHz.

In this equation, s_b is the bitstream size in bytes, whereas f_{clk} is the clock frequency of the ICAP port. Given that the Virtex-7 FPGA has a maximum clock frequency for the ICAP port of 100 MHz, the minimal reconfiguration time becomes:

$$t_r = \frac{s_b}{4 \times 10^8}$$

In order to measure the actual reconfiguration time, a compressed bitstream file of a small multiplier implementation is used. This bitstream file is then zero-padded to achieve bitstream sizes between 3 and 20 MiB. Because the reconfiguration has completed before reaching the zero-padding, the ICAP will discard any data on its input port until the SYNC keyword is encountered again. As such, zero-padding the bitstream file is a safe way of creating arbitrary bitstream sizes while still correctly performing the reconfiguration. The resulting measurements are illustrated in Figure 5.1 for a clock speed of 50 MHz, and in Figure 5.2 for a clock speed of 55 MHz. It should be noted that these measurements only account for the time it takes to load bitstreams from the main memory into the ICAP. The time it takes to copy the bitstream files from storage into memory is not measured. This choice was made due to the relatively slow SD-card interface of the VC707 FPGA board, which does not reflect the speeds of modern storage solutions and would greatly impact the resulting measurements. Furthermore, practical (future) implementations should load often-used bitstreams into memory at boot, or use a dedicated bitstream (control) store.

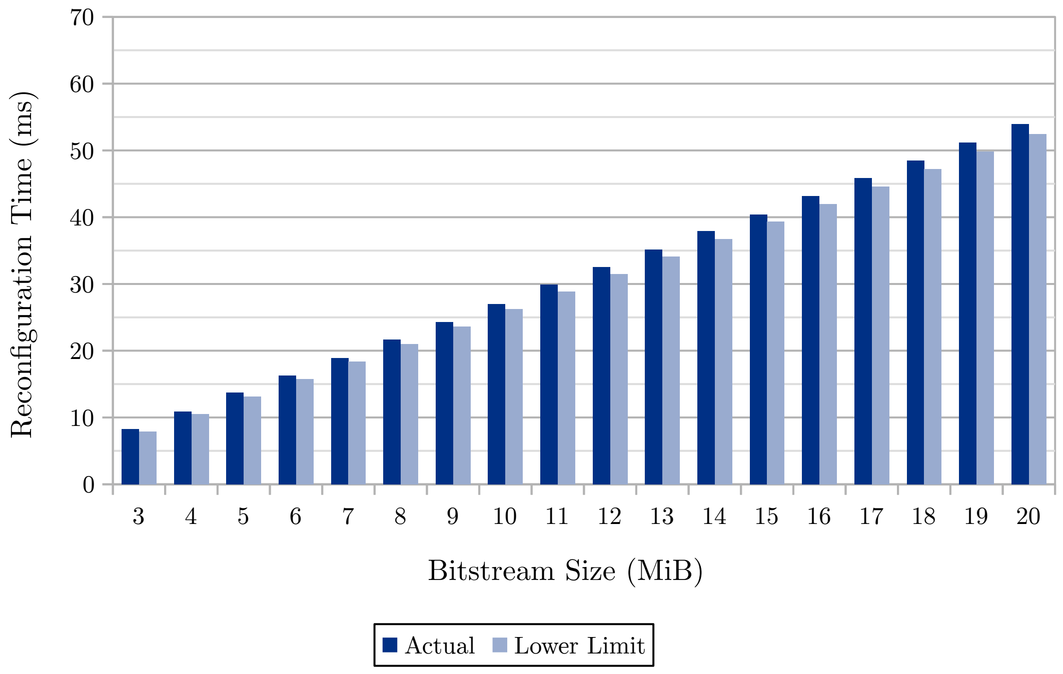

Figure 5.2: Reconfiguration times compared with the theoretical lower limit, running at 55 MHz.

5.4.2 Discussion

In the implementation of the proposed design, the ICAP port is clocked at its maximum frequency of 100 MHz. The CPU is clocked at 50 MHz, but utilizes a 64-bit AXI bus as opposed to the 32-bit input port of the ICAP. Burst reading on the AXI bus ensures maximum throughput as discussed in Section 4.4.1. As such, the design should be able to approach the theoretical limit on the reconfiguration time.

Although the measured reconfiguration times are close to their theoretical limits, an increasing difference between the two can be observed as the bitstream size increases. This difference ranges from 1.12 ms for the 3 MiB bitstream, to 6.44 ms for the 20 MiB bitstream.

To determine the cause of this difference, the signals inside the partial reconfigurator (discussed in Section 4.4.1) are captured using an ILA core. The resulting waveforms are depicted in Figure 5.3. The blue line labeled “T” is the trigger that detects a rising edge in the *molen_valid_i* signal, which signifies that a valid reconfiguration instruction is passed to the reconfigurator. The AXI request and response signals are shown, as well as the *fifo_empty* signal, which is set when the FIFO does not contain any data.

The consecutive burst reads can be identified by the changing *axi_req_o[ar][addr]* and *axi_req_o[ar_valid]* values. It can also be seen that the *fifo_empty* signal rises at the end of each burst read and remains high until new data arrives from the memory (*axi_resp_i[r_valid]*). For the minimal reconfiguration delay, the FIFO should not become empty until the last byte of the bitstream has been written. Measuring the first ten peri-

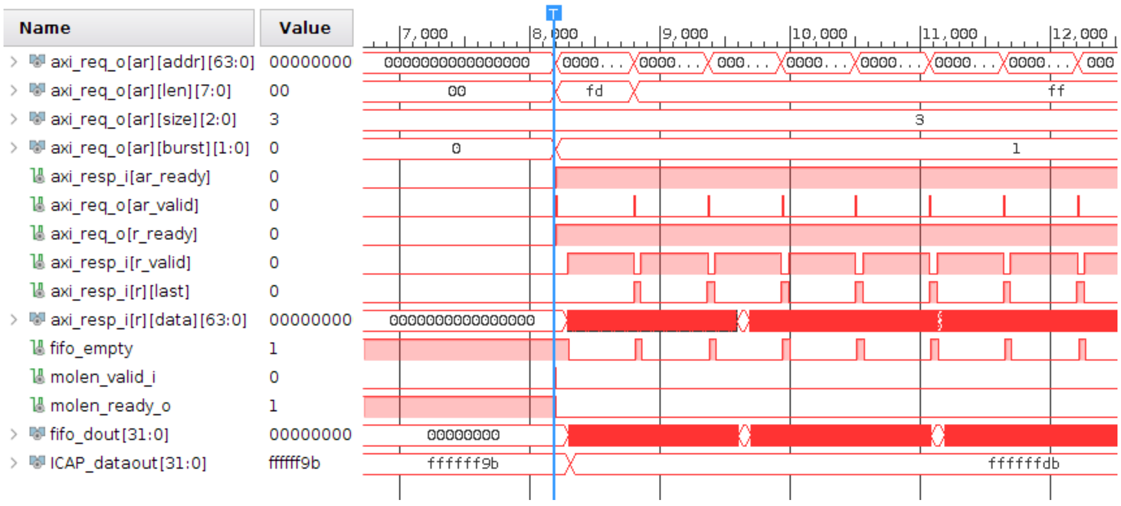

Figure 5.3: Waveform data captured at the start of the reconfiguration.

ods that the FIFO is empty, reveals a delay of 56 (100 MHz) cycles for each occurrence. Because the circumstances are identical, this delay is expected to be equal (or at least similar) for each time that the FIFO becomes empty until the end of the reconfiguration. Since this delay happens after each (fixed-length) burst read, and the total number of burst reads grows linearly with an increase in bitstream size, this explains the increasing gap between the reconfiguration time and its theoretical limit.

The FIFO queue was originally intended as a way to allow the reconfiguration data to be passed to the ICAP port in a single, uninterrupted, flow (after the first bitstream data arrived from the memory). This is based on the assumption that the data arrives faster from the memory than the ICAP can process it. The delay between requesting the next burst and receiving the first data of that burst could then be hidden from the ICAP port by the FIFO. Because the ICAP has a maximum clock frequency of 100 MHz and uses a 32-bit port, this assumption is reasonable for GHz CPU clock speeds. However, a CPU clock speed of 50 MHz is an edge case where data arrives exactly as fast from memory as the ICAP can process. The FIFO will not contain more than a single 64-bit word in that case, because the ICAP immediately consumes the data at the same speed as it is pushed into the FIFO. In this specific case, the FIFO acts only as a clock and data width converter, instead of an actual FIFO.

By increasing the clock speed to 55 MHz (Figure 5.2), it is confirmed that the clock frequency is responsible for the increasing gap with the theoretical limit. The difference now ranges from 0.42 ms for the 3 MiB bitstream, to 1.49 ms for the 20 MiB bitstream. Inspecting the captured waveforms again reveals that the FIFO is still completely emptied on occasion, albeit far less often than before. This can be explained by varying memory response times or memory accesses from the OS that occur in between burst reads. As such, it can be concluded that the small increase in the difference with the theoretical limit can still be explained by the clock speed. When operating the design at GHz speeds, the FIFO should not become empty before the entire bitstream is loaded. The reconfiguration time would then approach the theoretical limit even closer.

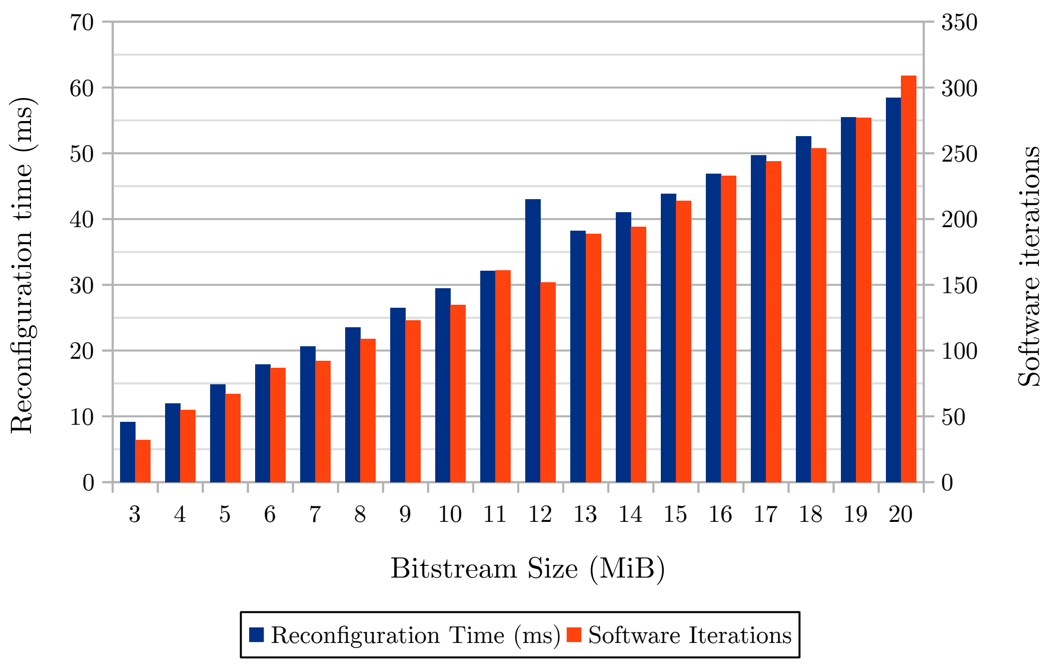

Figure 5.4: Reconfiguration time and number of software iterations for the 8x8 Matrix multiplication.

5.5 Matrix multiplication benchmark

Now that the reconfiguration time has been demonstrated to approach the theoretical minimum in the previous section, the goal of hiding that delay can be evaluated. In order to do so, it must be demonstrated that the CPU can do useful work while a reconfiguration is being performed. The matrix multiplication benchmark described in this section aims to do so. The details of the benchmark are first described, after which the results are discussed.

5.5.1 Benchmark description

This experiment attempts to demonstrate the ability to hide the reconfiguration delay by performing 8x8 matrix multiplications in software, until a reconfigurable design performing the same task in hardware has been loaded, after which the hardware design takes over the multiplication from the software. Before further explaining the details, it should be noted that this benchmark is not a full-fledged real-world application. Instead, it is specifically created for performing measurements. Therefore, the input data is arbitrarily generated.

Specifically, the values of the input matrices are generated as a function of the row and column of each value. The following formula is used to calculate the value at row i and column j of matrix m, as a function of input variables x and y:

$$m[i][j] = x \cdot i + y \cdot j + 1$$

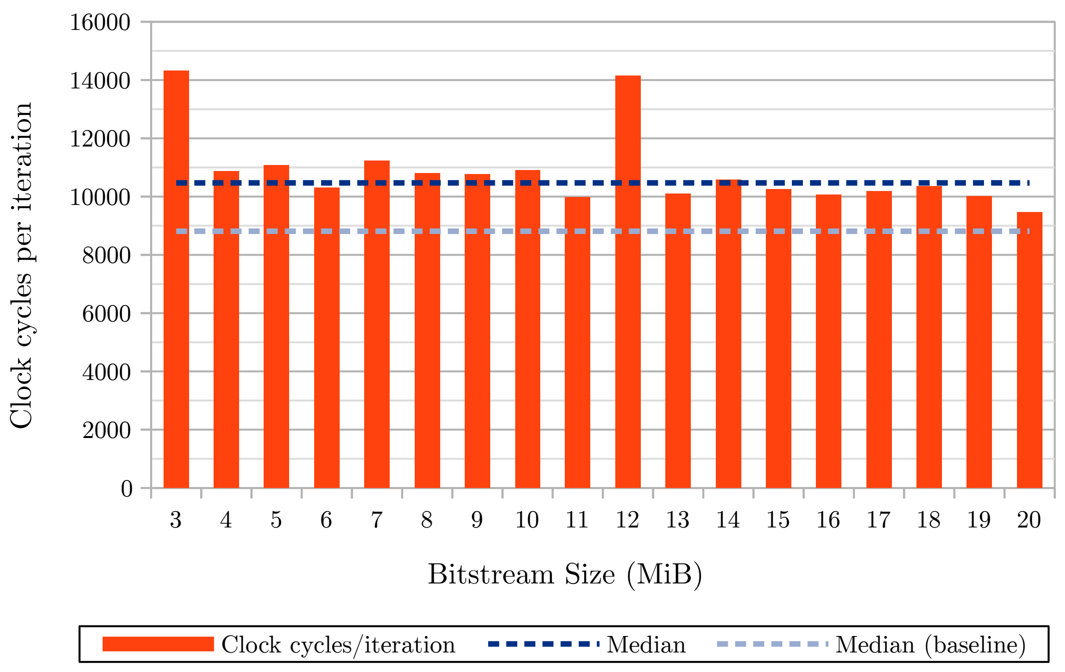

Figure 5.5: Average clock cycles per software iteration of the 8x8 matrix multiplication benchmark.

Using this function, two 8x8 matrices A and B are initialized (using arbitrary values for x and y), after which the matrix product AB is calculated. 8-bit values are used for all matrices. The initialization and multiplication are separate functions that are implemented in both hardware and software. These functions can now be executed in a loop to perform multiple matrix multiplications. Now, the reconfiguration can be triggered first, after which the software starts performing the multiplications. At the end of each loop iteration, the MOLEN *status* is checked to determine if the reconfiguration has completed. If it has, the following iterations use the hardware functions. Otherwise, the software continues the work. By measuring the reconfiguration time and the number of software iterations, the possible influence of the reconfiguration on the performance of the software (or the other way around) can be investigated.

The resulting measurements for bitstream sizes between 3 and 20 MiB are depicted in Figure 5.4. From these measurements, the average number of clock cycles per software iteration can be calculated. These values are depicted in Figure 5.5, along with the median across all bitstream sizes. As a reference, a baseline median is also calculated from a separate set of measurements where no bitstream reconfiguration is being performed while the matrix multiplication is executed.

5.5.2 Discussion

When inspecting Figure 5.4, it can be observed that the reconfiguration delays are mostly similar to those in Section 5.4, ranging from 9.16 ms for a bitstream of 3 MiB (compared to 8.96 ms when only loading a bitstream), to 58.48 ms for a 20 MiB bitstream (compared to 58.34 ms). An outlier is the 12 MiB bitstream, with a reconfiguration delay of 43.01 ms as compared to 35.14 ms when only loading a bitstream. Despite taking more time, the number of software iterations did not increase. Instead, fewer iterations could be executed than for the 11 MiB case. These observations are confirmed by Figure 5.5, which shows that the number of clock cycles per loop significantly exceeds the average for the 12 MiB bitstream. It also shows an outlier at 3 MiB that is less obvious in Figure 5.4.

Inspecting the actual (non-averaged) measurements reveals several outliers that are large enough to affect the averages significantly. For example, one measurement for the 12 MiB bitstream resulted in 60 iterations in 3800273 cycles (76.0 ms), which is approximately 63338 cycles (1.27 ms) per iteration. The average over all 20 measurements is 152 iterations in 2150734 cycles, or 14150 cycles (0.28 ms) per iteration. Interestingly, the mentioned outlier has both a lower than average iteration count and a higher than average clock cycle count. This is also the case for some other outliers, but not for all of them.

A possible explanation for these results may be found in the AXI memory interface that is shared between the CPU and the MOLEN unit. While a burst read of reconfiguration data from the MOLEN unit is being performed, the CPU cannot access the memory. It is up to the AXI interconnect to determine which master is connected to the memory. The software program and its stack may be fully cached, which means that reads can occur without needing access to the main memory. However, the CVA6 CPU uses a write-through cache (although a write-back cache can be configured instead). This means that data that is written is immediately written to the memory instead of only to the cache. Because the matrices in the software are too large to keep in registers, they are placed on the stack. When the matrix multiplication is performed, each value of the resulting matrix gets written (to the memory). If the MOLEN unit is performing a burst read at the same time, the software has to wait for the MOLEN unit to complete. It would then be up to the interconnect to allow the CPU access to the memory instead of keeping the MOLEN unit connected. In general, the results demonstrate that this is working correctly, because both the reconfiguration delays and the number of clock cycles per software iteration approach those of their standalone counterparts. However, there might be a problem in the interconnect that is triggered in specific cases and stalls either or both of the AXI masters (the CPU and the MOLEN unit). Attempting to accurately identify such a problem is outside the scope of this work, especially because the design remains functional and no indefinite stalls occur.

In order to further examine the impact of simultaneous memory accesses from the CPU while the bitstream was being reconfigured, another experiment was attempted. In that experiment, reads and writes were performed at random memory addresses, while reconfiguration was taking place. However, no clear results could be obtained from this experiment, with the measurements seemingly indicating that the CPU did not gain much access to the memory at all until the reconfiguration had completed. However, this behavior is purely due to the design of the interconnect and cannot be resolved with design changes to the MOLEN unit itself.

In order to meaningfully compare the number of clock cycles per iteration, the median across all bitstream sizes is chosen instead of the mean, because it is less affected by outliers. It should be noted that the median could also be used instead of the mean for averaging the 20 independent measurements for each bitstream size. However, doing so would hide the outliers altogether, which would also hide the most interesting information from the results.

When comparing the median values, it can be observed that the median for the case where reconfiguration is performed (10470 cycles) is 1658 cycles larger than the baseline median (8812 cycles). This is a relatively small gap, which demonstrates that the CPU can perform useful work while the reconfiguration is taking place and effectively hide the reconfiguration delay.

5.6 Discussion

When considering all of the results, it can be observed that the MOLEN design performs well overall, but may be hampered in specific cases by its integration into the CVA6 processor. Specifically, the AXI interface that is shared with the CPU limits the ability of the processor to access the memory while the MOLEN unit is performing a reconfiguration. Two possible workarounds could be implemented for this in the current design:

- The interconnect could be modified to prioritize the CPU (or MOLEN) access to the memory as desired.
- The MOLEN unit could be modified to force a delay between consecutive memory reads or to reduce the AXI burst size, which could allow the CPU to access the memory in between burst reads.

However, the matrix multiplication benchmark of Section 5.5 demonstrates that the negative impact of the shared memory interface is limited in typical cases. Only when the implementation is heavily memory bound will this become an issue (such as for the random memory access test that was attempted).

Furthermore, sharing the same AXI interface would no longer be feasible when running the CPU at GHz speeds in potential future implementations. In those cases, a separate memory port or separate memory unit would have to be used. This would automatically resolve the potential issues that might arise with the shared interface that is used in the current design.

Other than illustrating the potential problems with a shared memory interface, some of the results seemed to measure other parts of the design instead of the implementation of the MOLEN unit itself. For example, when comparing the resource utilization of the proposed design with that of the standalone CVA6 processor in Table 5.3, it seems as if there is a considerable increase in area at first glance. However, when looking at the area of only the MOLEN unit, that difference is not accounted for. Instead, the presence of the MOLEN unit affects the area of the remainder of the design, due to how the FPGA design is synthesized.

Another case where another part of the design was measured instead of the MOLEN unit, was the attempted random memory access test. In that case, the interconnect and its switching mechanisms were measured instead of the performance of the MOLEN unit.

Both of these cases underscore the importance of an optimal integration of the MOLEN unit into the (existing) CPU design. Instead of only focusing on the implementation of the MOLEN unit, the proper integration into the CPU is perhaps just as important in order to achieve the desired results.

One final remark about the experiments of the previous sections, is that it is important to note that not all of the benefits of the MOLEN implementation could be captured with measurements. In fact, the main benefit of the MOLEN implementation may not be its performance at all, but rather the way in which it enables developers to incorporate reconfigurability in their designs. Instead of having to manually implement reconfigurability first and then designing the interface to the reconfigurable part of the design, all of those functions are already provided by the proposed design. Developers can immediately start implementing and using their own reconfigurable designs. The integration of the reconfigurable instructions into the instruction set lowers the threshold to get started with reconfigurability and could make it more mainstream.

5.7 Conclusions

In this chapter, the implemented design was subjected to a series of tests to confirm that it functions as expected and to gain insight into its performance and resource consumption. All tests were performed on a Virtex-7 FPGA, using Vivado 2018.2 for the synthesis.

In order to test that the design behaves as desired, a standalone test of the MOLEN unit was first performed in simulation. Subsequently, the design was tested for basic functionality on the FPGA, by testing the reconfiguration capabilities. To conclude the testing, the robustness of the design was verified using corrupted bitstream files and erroneous usage of the MOLEN instructions.

After confirming the functional correctness of the design, the timing results and resource utilization could be examined. Pushing the clock speed of the design from 50 MHz to 70 MHz revealed that the MOLEN implementation does not impact the critical path, except for presence the reconfigurable partition (depending on size). In terms of resource utilization, a significant increase in area was measured compared to the base CVA6 design. However, this could also be explained by the presence of the reconfigurable partition affecting the synthesis.

Once the timing and area results were measured, the reconfiguration performance could be examined. Although the reconfiguration time approached the theoretical minimum, an increasing gap could be observed for increasing bitstream sizes. This gap was explained by the relatively low clock speed limiting the rate at which reconfiguration data could be loaded from the main memory. By increasing the clock speed to 55 MHz, this problem was resolved, with the reconfiguration time closely approximating the theoretical limit.

After having completed the other tests and measurements, it was determined if the reconfiguration time could be effectively hidden. A matrix multiplication benchmark was performed in which an 8x8 matrix multiplication was performed in software and hardware. Once the hardware had finished its reconfiguration, it could take over the multiplication from the software. By counting the number of software iterations, as well as the reconfiguration time, it could be established that the reconfiguration delay could be successfully hidden in most cases. Some outliers in the measurements, as well as an attempted random memory access test, indicated that the CPU might stall in cases that are heavily memory-bound, due to the shared AXI interface and the way the interconnect decides which AXI master is connected.

Although some workarounds may be implemented to try to resolve this issue, sharing the AXI interface in the way that is currently done would no longer be feasible when attempting to pursue real-world (ASIC) implementations of the design, due to the different clock speeds of the CPU and MOLEN unit.

Conclusion 6

6.1 Summary

In Chapter 2, relevant background information was provided on reconfigurable architectures, the MOLEN processor, and the RISC-V architecture.

Two main application domains for reconfigurable architectures were first identified: accelerators for **high-performance computing** and specialized or time-critical tasks in **embedded systems**. Reconfigurable architectures can be distinguished in terms of granularity. Fine-grained systems use the smallest building blocks and can be fully configured. Coarse-grained systems are less flexible and use larger building blocks, allowing for increased clock speeds and a greater energy efficiency.

After establishing a basic understanding of reconfigurable architectures, the MOLEN polymorphic processor could be introduced. In this paradigm, a one-time ISA extension introduces reconfigurable microcode ($\rho\mu$-code), which is used to perform reconfigurations and execute custom operations on the reconfigured hardware. This extension mainly consists of a *set* instruction that loads a reconfigurable design, and an *execute* instruction that performs the custom operations. The von Neumann architecture is extended with a reconfigurable processor that performs the custom instructions, an arbiter that passes the instructions to either the core processor or reconfigurable processor, and a number of exchange registers that are used for storing the input and output operands of the custom instructions.

After introducing the MOLEN paradigm, the RISC-V architecture was introduced. This open-source ISA offers multiple base instruction sets and extension instruction sets, thereby making it suitable for a wide range of applications. Support for custom instructions is also included, enabling its use for this project.

Having introduced the RISC-V architecture and the MOLEN paradigm, existing RISC-V processor implementations were compared, such that the most suitable one could be selected for extension with the MOLEN paradigm. The CVA6 core was chosen for this purpose, because it supports the Linux OS, is actively maintained and has a community with extensive documentation.

Before pursuing the actual implementation of the MOLEN paradigm on the RISC-V architecture, related work had to be examined, in order to substantiate the potential benefits of a novel implementation. Several existing reconfigurable implementations were compared. It was observed that most existing architectures are coarse-grained and application-specific. The proposed design improves upon this by allowing a wide range of applications, while not imposing a coarse-grained or fine-grained design.

In Chapter 3, the design of the MOLEN paradigm was adapted to the RISC-V architecture. First, the RISC-V instruction formats were introduced. These are the R-type and R4-type (register-register format), I-type (immediate format), S-type (store format), B-type (branch format), U-type (upper immediate format) and J-type (jump format). Together with the opcode, the *funct* fields (present in most instruction formats) specify the operation to perform.

After introducing the instruction formats, the RISC-V opcode space was explored. The *custom-0*, *custom-1*, *custom-2* and *custom-3* opcode spaces were identified for implementing custom instructions. From these options, the *custom-0* opcode space was chosen for the implementation of the current ISA extension.

Having established the opcode to use for the ISA extension, the design of the MOLEN $\rho\mu$-code could be adapted to the RISC-V architecture. Due to limitations of the FPGA hardware, the reconfiguration microcode was chosen as a wrapper around the partial bitstream bytes. The execution microcode was adapted into a nested microcode due to the same limitations. The R-type instruction was chosen for the implementation of this microcode, with the *funct7* and *funct3* fields forming the (nested) micro-instruction to perform.

Following the design of the reconfiguration and execution microcode, the specific MOLEN RISC-V ISA extension could then be formulated. The *set* instruction reconfigures the hardware, using the bitstream at the specified address. The *execute* instruction performs operations on the reconfigurable hardware by employing the implicit *funct10* field as (nested) micro-instruction. Furthermore, a *status* instruction was introduced to allow the reconfiguration to be performed in the background. This instruction returns the current reconfiguration status.

In Chapter 4, the MOLEN design from Chapter 3 was implemented on the CVA6 processor. After giving an overview of the CVA6 architecture, the following required modifications were identified to implement the MOLEN paradigm: the decoder had to be modified to handle the MOLEN instructions correctly, and a MOLEN functional unit had to be introduced to execute the new instructions. Additionally, it was decided that the MOLEN unit should be added as master on the AXI crossbar, after outlining the CVA6 memory organization.

The decoder was adapted in the following way: first, the opcode was compared to the *custom-0* opcode, after which the *funct7* and *funct3* fields were evaluated for matching the *set* or *execute* operations. In case of an *execute* instruction, the *funct7* and *funct3* fields were passed to the next pipeline stage as immediate value.

The MOLEN functional unit consists of a partial reconfigurator and a MOLEN reconfigurable module. The partial reconfigurator controls the ICAP to reconfigure the FPGA with a partial bitstream that it fetches from the specified memory address. A FIFO-queue is used as buffer to streamline this process. The MOLEN reconfigurable module consists of an interface definition that specifies the inputs and outputs, and an implementation of that interface that executes the actual custom micro-instructions. The AXI interface of the MOLEN functional unit is shared between the partial reconfigurator and MOLEN reconfigurable module, because simultaneous memory access from both sources cannot occur. An FSM inside the MOLEN functional unit is responsible for handling the *status* instructions, as well as the other two instructions when a reconfiguration is being performed.

In order to accommodate the partial reconfiguration inside the CVA6 FPGA design, two separate design flows were implemented. The **full design flow** is performed once and results in a full bitstream containing the CVA6 processor and a single reconfigurable module. The **partial design flow** is performed for each reconfigurable module and results in the partial bitstream that is programmed into the FPGA from within the MOLEN functional unit (when a *set* instruction is issued).

As a final part of the implementation, the software interface with the MOLEN extension must be considered. In order to share data, such as partial bitstreams, with the MOLEN functional unit, a DMA region is defined at the end of the main memory. Executing the MOLEN instructions is simplified by the presence of the *.insn* RISC-V assembler directive, which allows for the execution of custom instructions. In the C programming language, a *molen_execute* wrapper function can be defined using this directive. The *set* and *status* instructions can then be executed by specifying the corresponding $\rho\mu-$opcode for the*molen_execute* function call.

In Chapter 5, the implemented design was subjected to a series of tests to confirm that it functions as expected and to gain insight into its performance and resource consumption. All tests were performed on a Virtex-7 FPGA, using Vivado 2018.2 for the synthesis.

In order to test that the design behaves as desired, a standalone test of the MOLEN unit was first performed in simulation. Subsequently, the design was tested for basic functionality on the FPGA, by testing the reconfiguration capabilities. To conclude the testing, the robustness of the design was verified using corrupted bitstream files and erroneous usage of the MOLEN instructions.

After confirming the functional correctness of the design, the timing results and resource utilization could be examined. Pushing the clock speed of the design from 50 MHz to 70 MHz revealed that the MOLEN implementation does not impact the critical path, except for presence the reconfigurable partition (depending on size). In terms of resource utilization, a significant increase in area was measured compared to the base CVA6 design. However, this could also be explained by the presence of the reconfigurable partition affecting the synthesis.

Once the timing and area results were measured, the reconfiguration performance could be examined. Although the reconfiguration time approached the theoretical minimum, an increasing gap could be observed for increasing bitstream sizes. This gap was explained by the relatively low clock speed limiting the rate at which reconfiguration data could be loaded from the main memory. By increasing the clock speed to 55 MHz, this problem was resolved, with the reconfiguration time closely approximating the theoretical limit.

After having completed the other tests and measurements, it was determined if the reconfiguration time could be effectively hidden. A matrix multiplication benchmark was performed in which an 8x8 matrix multiplication was performed in software and hardware. Once the hardware had finished its reconfiguration, it could take over the multiplication from the software. By counting the number of software iterations, as well as the reconfiguration time, it could be established that the reconfiguration delay could be successfully hidden in most cases. Some outliers in the measurements, as well as an attempted random memory access test, indicated that the CPU might stall in cases that are heavily memory-bound, due to the shared AXI interface and the way the interconnect decides which AXI master is connected.

Although some workarounds may be implemented to try to resolve this issue, sharing the AXI interface in the way that is currently done would no longer be feasible when attempting to pursue real-world (ASIC) implementations of the design, due to the different clock speeds of the CPU and MOLEN unit.

6.2 Main contributions

Having arrived at the end of this thesis, an answer to the main research question from Section 1.2 can now be formulated. The research question is first reiterated, after which it is answered:

> *Will the application of the MOLEN paradigm to a modern processor allow for hiding reconfiguration latencies?*

By implementing the MOLEN instruction set extension on the CVA6 RISC-V processor (using a Virtex-7 FPGA), the reconfiguration delays could be successfully hidden in a matrix multiplication benchmark, in which the CPU takes over the matrix multiplication until the hardware has finished its reconfiguration, at which point the hardware takes over the calculations from the CPU.

Some outliers in the results of the matrix multiplication benchmark, as well as the initial results of an attempted random memory access benchmark, indicate that the CPU might not be able to do useful work during reconfigurations for heavily memory-bound applications, which means the reconfiguration delay cannot be successfully hidden in such cases. This is due to the (AXI) memory interface that is shared between the CPU and MOLEN unit, and the switching characteristics of the specific interconnect that is used to accommodate both AXI masters. This problem could be resolved by modifying the interconnect and its switching priorities, or by giving the MOLEN unit its own memory interface (or separate memory unit). Doing so would reduce the contention of the CPU and MOLEN unit, which in turn would allow the CPU to perform useful work even for completely memory-bound functions. Having a separate memory for the MOLEN unit would remove this contention altogether.

Other than achieving the central goal of this project, the following main contributions can be identified:

- **A fully functional 64-bit MOLEN RISC-V CPU** has been developed as an extension of the CVA6 processor, a Linux-capable and well-supported open source CPU. The design is implemented on a Virtex-7 FPGA. The functionality and robustness of the reconfigurability are verified using several functional tests.

- **An implementation of the MOLEN extension** was made using a *set* instruction that triggers a reconfiguration and a *execute* instruction that performs functions on the reconfigured hardware. Additionally, a novel *status* instruction was introduced to allow the CPU to evaluate the progress and result of the *set* instruction (the reconfiguration status), such that the reconfiguration can be performed in the background.

- **The novel concept of nested reconfigurable microcode was developed** as practical alternative to the more traditional microcode that was proposed in the original work ([12]). Instead of pointing the MOLEN unit to the address of a microprogram, each separate micro-instruction is encapsulated in a regular RISC-V instruction, such that it can be passed to the MOLEN unit. In this way, the MOLEN paradigm was made feasible for implementation on the available FPGA hardware.
- **Near-optimal reconfiguration performance was achieved** by using AXI burst reading to fill a hardware FIFO that in turn writes its contents to the ICAP port of the FPGA. At the default clock speed of 50 MHz (and the maximum ICAP speed of 100 MHz), the FIFO cannot be filled fast enough to obtain optimal performance. However, at a clock speed of 55 MHz, the performance already approaches the theoretical limit. At higher speeds, the reconfiguration speed will be even closer to its theoretical limit.

6.3 Future work

Although a fully functional implementation was created for this thesis, it should be considered a proof-of-concept rather than a full-fledged design. Some simplifications and changes had to be made to the original MOLEN paradigm, in correspondence with the objectives formulated in Section 1.2. The available FPGA platform and its internal organization also caused some practical limitations on how the MOLEN design could be implemented. This means there is still room for improvements and further research into how to adapt the MOLEN paradigm to a modern processor. Some potential topics to consider are the following ones:

- More extensive performance evaluation.
- Developing an ASIC implementation, as described in Section 6.3.1.
- Redesigning the microcode to be closer to the proposal of the original work, as described in Section 6.3.2.

The more extensive performance evaluation would entail testing of the (reconfiguration) performance in real-world applications, and comparing the results to those of other solutions. The other topics are more involved, and are further explained in the following sections. The individual ideas presented within those sections could also be considered separately or in a different setting.

6.3.1 ASIC implementation

A potential improvement avenue is to pursue an ASIC implementation of the current design. Although a fully functional ASIC implementation of the CVA6 core has already been created, the proposed ISA extension will significantly influence future ASIC implementations. Specifically, the need for reconfigurable fabric in the final design could lead to two separate implementation avenues:

- A separate FPGA that is connected to the ASIC. This option would be the most feasible, because of the wide range of readily available FPGA.
- An integrated SoC, comprising the CVA6 MOLEN core and the reconfigurable fabric. This option is significantly more involved than the previous one and should probably not be considered for an initial implementation.

Both options would require a higher degree of separation between the CVA6 core and the MOLEN unit than the current design provides, due to the physically separated hardware and separate clock speeds for the CPU and reconfigurable fabric. The execution of the reconfigurable micro-instructions could no longer be part of the instruction pipeline of the CPU, as is the case in the current design. Although the MOLEN unit is already separated into an ICAP reconfigurator and MOLEN reconfigurable module (as illustrated in Figure 4.3), both of those modules are still part of the pipeline. In order to achieve the higher degree of separation, the following points should be considered:

- The reconfigurator must be updated due to the change towards a separate reconfigurable fabric. Currently, the reconfiguration it performs is always a partial reconfiguration, because the CPU is located on the same FPGA fabric. However, a separate FPGA or reconfigurable fabric could be either partially or fully configured.
- Performing the reconfigurable micro-instructions outside of the CPU pipeline would require a new way to pass them to the reconfigurable fabric, such as an instruction buffer. This in turn means it must be considered how to return status and output variables back to the CPU. Interrupts could be used for this, or separate polling-instructions (such as the current MOLEN *status* instruction). Alternatively, the *break* instruction from the complete πISA of the original work could be implemented.
- The MOLEN unit could get its own memory unit, or keep its access to the main memory unit of the CPU. Either method has its own implications. If it does get its own unit, the CPU must be given a way of accessing that memory. Conversely, if it is not given its own unit, it must be able to access the main memory unit, which is more involved than it currently is, due to the higher separation.

If an ASIC implementation is considered to be too involved, an alternative option would be to implement a double FPGA version first. In that case, one FPGA would contain the CVA6 core with a (fixed) MOLEN unit, whereas the other one would contain the reconfigurable fabric that is controlled by the MOLEN unit. Such an implementation would achieve the higher separation required for an ASIC implementation, without also introducing the unique hardware design challenges of an actual ASIC implementation.

6.3.2 Microcode redesign

Practical considerations caused the current design to have moved away from the type of microcode proposed by the original work to the nested reconfigurable microcode presented in this thesis. Potential future efforts could involve attempting to match the original microcode more closely. Doing so would achieve a coarser granularity for the reconfigurability, in which individual hardware units can be designed separately. Those units can then be placed and connected as needed in the reconfigurable fabric using microcode. The original work assumes patterns can be discovered in the reconfigurable bitstreams as a result of the low-level building blocks that an FPGA is comprised of. To our knowledge, such reverse-engineering would not be feasible for the specific FPGA platform that was used for the implementation of this work.

Instead, the advent of relocatable partial bitstreams, such as those proposed in [30], may enable the use of the original microcode idea. Being able to generate relocatable bitstreams allows each standalone design to be placed in each available reconfigurable partition. Because those different designs also need to be connected to each other, the need for a shared interface arises. Such an interface could also limit the flexibility of the possible designs. Having multiple reconfigurable partitions also requires careful consideration on the partition sizes. Additionally, being able to load separate reconfigurable designs into separate reconfigurable partitions does not automatically grant the ability to use microcode to connect them. Attempting to use microcode for this purpose may not be a trivial task.

Before evaluating the feasibility of the original microcode, it must also be carefully considered what the possible advantages and disadvantages are when compared to the current design. Currently, hardware modules can still be reused, except that reuse takes place inside the hardware design itself, rather than between separate hardware designs. For example, individual full adders can still be combined into an (n-bit) adder implementation, just not using microcode. If the reconfiguration capabilities are targeted at hardware developers, intricate knowledge on hardware development could already be assumed. In such a case, it may not be beneficial to let them connect individual hardware units using microcode. On the other hand, providing a wide library of hardware modules along with the microcoded reconfiguration could allow other users than hardware developers (such as software developers) to apply reconfiguration in their applications.

Bibliography

[1] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The molen polymorphic processor,” *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, Nov 2004.

[2] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[3] A. Waterman and K. Asanović, “The RISC-V instruction set manual, volume I: User-level ISA, document version 20191213,” Dec. 2019.

[4] G. E. Moore, “Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan 1998.

[5] ——, “Progress in digital integrated electronics [technical literaiture, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.],” *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 36–37, Sep. 2006.

[6] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[7] M. Bohr, “A 30 year retrospective on dennard’s mosfet scaling paper,” *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[8] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[9] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 365–376.

[10] W. Nawrocki, “Physical limits for scaling of integrated circuits,” *Journal of Physics: Conference Series*, vol. 248, p. 012059, nov 2010. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F248%2F1%2F012059

[11] C. Kachris and D. Soudris, “A survey on reconfigurable accelerators for cloud computing,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–10.

[12] J. S. S. M. Wong, “Microcoded reconfigurable embedded processors,” Ph.D. dissertation, Delft University of Technology, Dec. 2002.

[13] "RISC-V origin," https://riscv.org/risc-v-history, accessed: 2020-03-22.

[14] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: Base user-level ISA," University of California at Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011.

[15] T. Chen and D. A. Patterson, "RISC-V geneology," University of California at Berkeley, Tech. Rep. UCB/EECS-2016-6, Jan. 2016.

[16] "About the RISC-V Foundation," https://riscv.org/risc-v-foundation, accessed: 2020-03-22.

[17] "Xilinx virtex-7 FPGA vc707 evaluation kit," https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html, accessed: 2020-06-25.

[18] "RISC-V cores and SoC overview," https://riscv.org/risc-v-cores/, accessed: 2020-08-04.

[19] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 9–16.

[20] N. Clark, M. Kudlur, Hyunchul Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, Dec 2004, pp. 30–40.

[21] M. Wijtvliet, J. Huisken, L. Waeijen, and H. Corporaal, "Blocks: Redesigning coarse grained reconfigurable architectures for energy efficiency," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 17–23.

[22] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep. 2012.

[23] A. Danilin, M. Bennebroek, and S. Sawitzki, "Astra: An advanced space-time reconfigurable architecture," in *2006 International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–4.

[24] S. Liang, S. Yin, L. Liu, Y. Guo, and S. Wei, "A coarse-grained reconfigurable architecture for compute-intensive mapreduce acceleration," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 69–72, July 2016.

[25] Cao Liang and Xinming Huang, "Smartcell: A power-efficient reconfigurable architecture for data streaming applications," in *2008 IEEE Workshop on Signal Processing Systems*, Oct 2008, pp. 257–262.

[26] H. Park, Y. Park, and S. Mahlke, “Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications,” in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 370–380.

[27] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, “A reconfigurable heterogeneous multicore with a homogeneous isa,” in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1598–1603.

[28] Xitian Fan, Huimin Li, Wei Cao, and Lingli Wang, “Dt-cgra: Dual-track coarse-grained reconfigurable architecture for stream applications,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[29] T. Lin, W. Zhang, and N. K. Jha, “A fine-grain dynamically reconfigurable architecture aimed at reducing the fpga-asic gaps,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2607–2620, 2014.

[30] V. M. G. Martins, J. a. G. Reis, H. C. C. Neto, and E. A. Bezerra, “Designing partial bitstreams for multiple xilinx fpga partitions,” in *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM ’15. USA: IEEE Computer Society, 2015, p. 256–259. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1109/FCCM.2015.10

[31] *Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics*, Xilinx, 3 2021, v1.29. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds183_Virtex_7_Data_Sheet

[32] *Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, 6 2018, v2018.2. [Online]. Available: https://docs.xilinx.com/v/u/2018.2-English/ug909-vivado-partial-reconfiguration

[33] “Cva6 5.0.1,” https://github.com/openhwgroup/cva6/releases/tag/v5.0.1, accessed: 2024-06-23.

[34] *The C Preprocessor*, Free Software Foundation, 2020, v10.2.0. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-10.2.0/cpp.pdf

[35] “Vivado design suite - hlx editions: Update 2 - 2018.2,” https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html, accessed: 2024-06-23.

[36] “Corev-apu fpga emulation,” https://github.com/openhwgroup/cva6/blob/v5.0.1/README.md#corev-apu-fpga-emulation, accessed: 2024-06-23.

[37] “Risc-v gnu compiler toolchain,” https://github.com/riscv-collab/riscv-gnu-toolchain, accessed: 2024-06-23.

[38] *Vivado Design Suite User Guide: Programming and Debugging*, Xilinx, 6 2018, v2018.2. [Online]. Available: https://docs.amd.com/v/u/2018.2-English/ug908-vivado-programming-debugging