Low-power Acceleration of Convolutional Neural Networks using Near Memory Computing on a RISC-V SoC

KRISTOFFER WESTRING & LINUS SVENSSON
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Low-power Acceleration of Convolutional Neural Networks using Near Memory Computing on a RISC-V SoC

Kristoffer Westring, Linus Svensson
kristoffer.westring@eit.lth.se, li8620sv-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Joachim Rodrigues

Examiner: Pietro Andreani

October 19, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

Acknowledgement

We would like to express our deepest gratitude to our supervisor, Joachim Rodrigues, for his guidance and knowledge. His door was always open whenever we needed advice on our research. Beyond the academic guidance, Joachim has been an outstanding supervisor, often providing much-needed and meticulously brewed espressos — a testament to the care he fosters for his students. These gestures made this thesis enjoyable. His passion for both academia and coffee has left a lasting impression on us.

This endeavour would not have been possible without the collaboration with Codasip. Our sincere appreciation goes to Keith Graham and Tadej Murovič. Thank you for granting us access to essential resources and for providing valuable feedback during our research.

We are also thankful for our co-supervisors Masoud Nouripayam and Arturo Prieto and the other colleagues at the department who provided us with insightful discussions. Special thanks to Sergio Castillo Mohedano, whose prior work laid the foundation for our research.

We cannot thank Per Andersson enough for his support and expertise. Per was a beacon of knowledge, and his problem-solving skills were crucial in overcoming some of the biggest challenges we faced.

Lastly, we would like to mention our families and friends, whose encouragement and understanding helped us achieve this journey.

Abstract

The recent peak in interest for artificial intelligence, partly fueled by language models such as ChatGPT, is pushing the demand for machine learning and data processing in everyday applications, such as self-driving cars, where low latency is crucial and typically achieved through edge computing. The vast amount of data processing required intensifies the existing performance bottleneck of the data movement. As a result, reducing data movement and allowing for better data reuse can significantly improve the efficiency.

Processing the data as closely to the memory as possible, commonly known as near-memory computing, increases the power efficiency and can significantly reduce the bottleneck in the data movement. However, maintaining a low power consumption while at the same time being able to process large amounts of data is a challenge. The RISC-V Instruction Set Architecture (ISA) was designed for efficient and dense instruction encoding, enabling lower power consumption and quicker execution time [1]. Extending the simple RISC-V ISA with specific instructions for applications like image recognition can make a processor energy-efficient but less versatile than a conventional CISC processor [2]. Codasip, a company specializing in RISC-V processors, offers a toolset for exploring and customizing processor architectures, through their proprietary C-based hardware description language, CodAl, which is used to generate SDK, HDL, and UVM within the Codasip Studio Environment. Codasip provides a selection fully configurable RISC-V cores, tailored for either low-power, and high-performance application.

In this thesis we use a combination of high-level synthesis tools and EDA software to simplify design space exploration of accelerators, allowing for the accelerators to be integrated as Near Memory Computing (NMC) accelerators on a customized RISC-V System on chip (SoC), for both Application Specific Integrated Circuits (ASIC) and Field Programmable Gate Arrays (FPGA). The flow contains implementation of custom instructions as well as a generic flow from Register Transfer Level (RTL) to GDSII for reuse in future works.

The findings from this research shows that tightly integrated NMC accelerators in combination with specialized RISC-V instructions can offer a 10× system performance improvement compared to baseline scenario with minor area increase, of only 38.5%. In this work the performance density was improved to 8.4× that of the baseline case, for certain applications.

Popular Science Summary

In recent years, development and adaption of Artificial intelligence (AI) have increased rapidly. As the usage and accuracy of these models increases, the computation needed for inference increases. This opens up the need for architectural innovation, to reduce both power consumption and latency.

As machine learning becomes an increasingly integral part of our daily lives, the volume of data being processed in our devices is skyrocketing. Through the use of cloud computing, our devices are able to transmit and offload the data that requires heavy computation to servers. However, this method is not without its drawbacks.

In many instances, transferring data over the internet for computation at a remote server induces a latency. For real-time applications such as self-driving cars this delay could be problematic. Being able to perform the computation locally, what is commonly known as *edge computing*, is essential to many critical applications but introduces a new set of hurdles to overcome, namely performance and power efficiency. Computing large volumes of data in a battery powered units poses immense challenges in comparison to a server hall with a virtually limitless supply of power. Recent AI-models, especially large language models, like ChatGPT, consist of billions of parameters that should be moved between memory and processing units. Moving all these data imposes a bottleneck for efficient AI inference.

By shifting towards a more data-centric architecture like NMC, which involves moving the data-intensive calculations closer to the origin of the data, latency and power consumption can be reduced. Through strategic optimizations like these, NMC promises a more efficient and responsive computing architecture for data-intensive applications. Improving energy efficiency and processing power are crucial obstacles that must be overcome for the continuous evolution of technology. Near memory computation is one of the solutions that has been showing promising results in recent studies. [2, 3, 4]

Table of Contents

List of Abbreviations

AHB Advanced High-performance Bus

AI Artificial intelligence

ASIC Application Specific Integrated Circuits

BRAM Block Random Access Memory

CIM Computation-in-Memory

CNN Convolutional Neural Network

CPU Central Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

FPGA Field Programmable Gate Arrays

FU Functional Unit

GPR General Purpose Register

HDL Hardware Description Language

HMC Hybrid Memory Cubes

IFM Input Feature Map

IP Intellectual property

ISR Interrupt service routine

LLC Last-Level Cache

MAC Multiply And Accumulate

MCU Micro Control Unit

NDP Near-Data Processing

NMC Near Memory Computing

OFM Output Feature Map

PC Program Counter

PIM Processing-in Memory

PnR Place and route

PPA performance, power, and area

RAM Random Access Memory

SoC System on chip

SRAM Static Random Access Memory

EDA Electronic Design Automation

ALU Arithmetic Logic Unit

LSB Least Significant Bit

ISA Instruction Set Architecture

RTL Register Transfer Level

List of Figures

List of Tables

Chapter 1

Introduction

The rise of data-intensive applications in the modern world of computing has sparked significant advancements and innovations in information technology. Applications involving real-time sensor data processing and machine learning demand substantial computational capabilities and memory capacity. Running these applications poses a challenge due to the performance and energy requirements.

The primary challenge faced by data-intensive applications is the data movement problem, also known as the von Neumann bottleneck. This issue stems from the separation of the processing unit and the memory in the traditional von Neumann architecture, resulting in a data movement between the memory and the processing unit. The latency and energy cost associated with this data transfer creates a bottleneck in the performance and power efficiency, which can be relieved by minimizing data movement and increasing data reuse. [2]

This is not a recently discovered problem and various solutions have been proposed, for about as long as the architecture has existed. Stone H. contributed to the foundation of NMC by proposing a logic-in-memory computer, organized around a cache with the ability to perform arithmetic and logic operations on blocks of data. [5]

However, it was not until more recently that NMC has emerged as a viable solution to this problem. NMC, also known as in-memory computing or processing-in-memory, is differs from the traditional approach to computing. Rather than constantly transferring data between the processor and the memory, NMC integrates computation inside or near the memory itself.

By moving data processing closer to the storage locations, the requirement of data transfer is significantly reduced, minimizing the energy and latency costs associated with it. NMC architecture allows for more efficient data access and manipulation, making it a promising solution for data-intensive applications in edge computing.

In the following chapters, we will explore the foundation of NMC, and look into its implementation. We will examine how the architecture can be modified to work in tandem with the accelerator, and evaluate its potential in overcoming the data movement problem , enabling the next generation of high-performance computing.

1.1 Related work

There are numerous implementations and interpretations of NMC, ranging from modifying the actual memory cells to being able to perform computation, to placing additional computational cores in close proximity of the storage. Developments in three dimensional memory stacking has sparked new interest in NMC and is seen as groundbreaking as logic can be inserted between the stacked memory layers. [2]

Nouripayam et al. proposed an NMC architecture based on standard SRAM memories, as to utilize the area efficiency of existing SRAM, with a Micro Control Unit (MCU) which functions as a co-processor integrated into the L2 memory, see Fig. 1.1. The main idea behind the use of a co-processor is to reduce workload of the main processor which instead is free to perform general purpose tasks, effectively increasing the bandwidth while retaining flexibility. The NMC unit used in the proposed architecture is an 8-bit fixed point CNN accelerator. [3]

The synthesized result with the specific NMC unit, showed an increase in area of less than 1% with a 38× increase in performance and 28× increase energy efficiency when compared to the baseline processor, PULPissimo, during full inference of the MNIST dataset. [6]

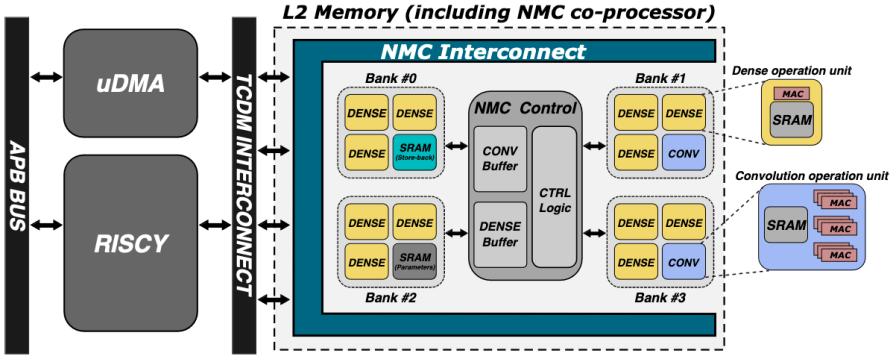


Figure 1.1: Proposed architecture design based around NMC, by Nouripayam et al.

The proposed architecture provides a scalable system in which multiple co-processors can be attached to the system, moving the processing units closer to the memory as a solution to the memory wall problem. [6]

1.2 Thesis outline

The overarching goal of this thesis is to develop a flow for FPGAs and ASIC, in which an existing accelerator can be attached as a NMC unit to a RISC-V core. This integration is designed to enhance data transfer efficiency and enable parallelization between the core and the accelerator, by customizing the core to the needs of the accelerator.

The thesis is organized as follows:

Chapter 1: Introduction. Introduces the problem and provides context for the thesis. It also discusses the motivation behind addressing this problem and the overall structure of the thesis

Chapter 2: Background. Introduces the reader to components and concept that will be used in the final design.

Chapter 3: System Design and Implementation. Describes the design methodology, constraints and design decisions.

Chapter 4: Testing and Verification. This chapter provide the reader with the baseline CNN, which is used in functional tests and performance metrics. Limitations to the verification are presented along with the algorithms used.

Chapter 5: Results. The final metrics for the design are shown and a comprehensive comparison between the base model and the design from this work are presented.

Chapter 6: Discussion & Future Work. Conclusions and future work are discussed.

Chapter 2

Background

Architectures supporting NMC have become a popular data-centric technique to address the growing speed gap between the processing unit and its separated memory [7]. This method decreases the physical distance between memory and computation, reducing expensive data movement thus increasing energy-efficiency while simultaneously reducing latency.

The main focus in recent times has been on- and off-chip implementations using Hybrid Memory Cubes (HMC) with a focus on high memory entropy applications [2]. Regarding CNN the memory entropy is quite low, which means that the cache hit rate is quite large. This suggests that due to the high hit rate, implementing NMC near the cache can further reduce the data transactions. Even though the compute module will be implemented on chip, some of the problems with NMC still persists.

Nouripayam et al. implemented an architecture for NMC at the cache hierarchy on the open-source RISC-V platform PULP, intending for a flexible main CPU handling general purpose applications while the NMC unit would perform CNN operations. The suggested architecture employs standard SRAM, thereby preserving the SRAM's area efficiency. The NMC architecture showed promising results with an energy efficiency as well as a performance increase of more than 28× respectively 34×, compared to the baseline of the same MCU, with an area overhead of 1%. [3]

2.1 RISC-V

RISC-V is an open standard ISA based on Reduced Instruction Set Computer (RISC) principles. The RISC-V ISA is provided under open-source licenses, making it appealing for a wide range of computing devices. The ISAs scalability and modularity provides opportunities for designers to tailor it to the specific needs

of their device, selecting the necessary instruction sets for an efficient design, thus being suitable for everything from embedded systems to high-performance servers.

The scalability of RISC-V is partly due to the availability of multiple baseline ISAs, offering 32-, 64-, or 128-bit integer bases, as well as a variety of optional extensions. This flexibility makes RISC-V easier to implement than many alternatives, leading to its growing popularity in the semiconductor industry. This allows for a clean core design with significant possibilities to expand and adapt as per the requirements.

2.1.1 Codasip

The specific core provided by Codasip for this thesis is the **L31**, a 32-bit, low-powered RISC-V processor with a 3-stage in-order pipeline. This core includes a hardware multiplier and divider for efficient computation, and offer an optional floating point unit to further processing capabilities. The L31 core's built-in support for AMBA AHB and AXI interfaces provides a straightforward way to integrate with existing systems such as caches and tightly coupled memories. The core offers a solid starting point for adapting and optimizing through the use of the architecture description language, *CodAl*.

Codasip Studio, a development environment used for designing, optimizing, and verifying the customized processor core, utilizes the high-level language CodAL to define both the ISA and the micro-architecture. The CodAL description consists of an instruction-accurate (IA) model, containing the basic instruction set used for the compiler and IA simulations. In addition to the IA-model, a cycle-accurate (CA) model provides actual the micro-architectural implementation, from which the RTL code is generated. The IA- and CA-model, with their abilities to be independently debugged, provides an environment for simulation: the IA-model enables instruction-level simulation, while the CA-model offers a more realistic clock-cycle level simulation.

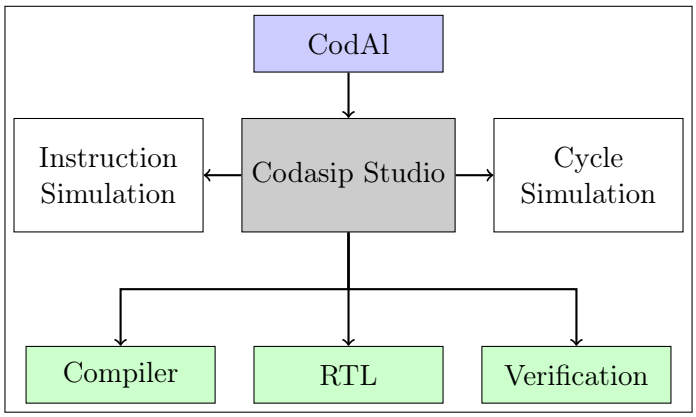


Figure 2.1: General work flow and tools in Codasip Studio.

Beyond simulation and debugging, Codasip Studio is capable of generating a compiler, a software toolchain, and verification components directly from the CodAL description. This capability allows for custom instructions to be interpreted directly by the compiler from the C-code, integrating smoothly into the development workflow.

2.1.2 Instruction Set Architecture

The RV32IMC ISA in the L31 core represents different types of instructions: RV32I is the base integer instruction set, with common instructions such as add, and sub. The 'M'-extension contains integer multiplication and division, and 'C' refers to the compressed instruction set, see Tab. 2.1. The compressed instruction set reduces the code by introducing shortened, 16-bit instructions. Around $50\% - 60\%$ of the instructions in a RISC-V program can be can be exchanged for compressed instruction, which corresponds to a reduction of $25\% - 30\%$ of the text size in a program. However this is an approximation and the actual reduction heavily depends on the program. The L31 core's RV32IMC ISA is designed for efficient and compact instruction encoding while still maintaining strong performance in integer operations. [8]

Table 2.1: ISA of the possible extensions of the baseline L31 core, with the highlights the extensions used in this thesis.

ISA	**Instructions**	**Description**
RV32I	47	32-bit address space and integer instructions
Extension	**Instructions**	**Description**
M	8	Integer multiply and divide
A	11	Atomic memory operations, load-reserve/store conditional
F	26	Single-precision (32 bit) floating point
D	26	Double-precision (64 bit) floating point; requires F extension
Q	26	Quad-precision (128 bit) floating point; requires F and D extensions
C	46	Compressed integer instructions; orangeuces size to 16 bits

The instructions in RISC-V are encoded in several formats. These include R-type, I-type, S-type, B-type, U-type, and J-type. Each of these types represents different operations and are used for different functionalities in the system, corresponding to certain groups of instructions. Each instruction type contains a combination of the fields register sources (rs1, rs2), the register destination (rd), the opcode, the functionality (funct7, funct3) and the immediate value (imm). Due to the flexible of the architecture, besides creating new instructions it is also possible to add new of instructions types.

Table 2.2: RISC-V 32-bit instruction types format.

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

For custom instructions, choosing the most suitable instruction type can have a significant impact on efficiency. For example, for an instruction where no data will be stored in a register, it is preferable to use an instruction type that allows for the transmission of as much data as possible. The S-type instruction does not have a field for register destination, but instead has an extended field for immediate values, as shown in Tab. 2.2. If needed, the extra immediate bits that were gained could be used to handle more data, effectively increasing the efficiency.

2.1.3 Pipeline

The core provided in the thesis is a 3-stage pipelined processor, consisting of the stages Fetch (FE), Instruction Decode (ID), and Execute (EX). In a pipelined processor, these stages run in parallel, thus processing different instructions. During fetching of one instruction, a second instruction is being decoded, while a third is being executed. The goal of pipelining a processor is to improve the throughput of the processor by keeping all parts of the processor busy. The concept of pipelining works, as a single instruction usually requires more than one clock cycle fully execute.

Fetch Stage

The first part in the processing of an instruction is to fetch the instruction from the memory. The Program Counter (PC) keeps track of the following instruction to be executed, and the value from the PC is used to determine the memory location from which to fetch. The fetched instruction is placed into a instruction fetch buffer, to be read from the following stage. The fetch stage in the provided core also contains a branch predictor, which is a technique that tries to predict whether a branch will be taken, prior to actually knowing the answer. This improves the system performance as the instruction following the branch can be predicted instead of waiting until the branch instruction has been fully executed.

Instruction Decode Stage

From the instruction fetch buffer, the instruction is pushed to the ID stage. The instruction decoder breaks down the bits of the instruction, based on its instruction type, see Tab. 2.2, and generates the necessary signals for the upcoming execution stage. The ID stage also contains the hazard detection, which flag for hazards.

Execute Stage

The execution stage performs the actual operation implied by the performed instruction. This stage contains the Arithmetic Logic Unit (ALU) and the Functional Unit (FU), to perform arithmetic, logical, memory operations, etc. In this stage, read and write operations are performed on the general purpose register, based on the instruction type. As some operations such as division and memory operations can take multiple clock cycle to execute, this stage has to ability to perform stalls, which freezes the stages until the operation has been performed.

Pipelining a processor introduces architectural complexities such as hazards that can occur when instructions depend on the result of other instructions. For a lot

of instruction combinations, hazards are not an issue. However for instructions that require multiple clock cycles in the execution stage e.g., division and load word, a hazard can occur when the multi-cycle instruction is writing to a General Purpose Register (GPR) that the following instruction is reading from.

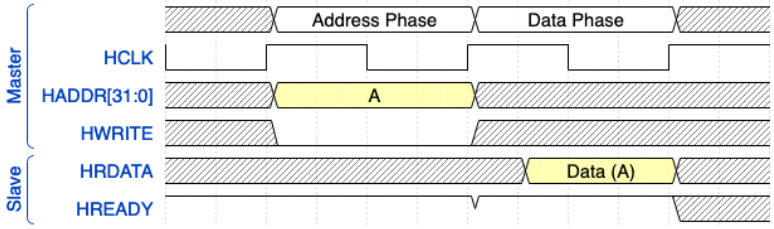
Stalling a stage is the act of pausing one or more stages from continuous operation. A control unit in the ID stage keeps track of any potential hazard, in which case the ID stage will stall. This is essentially the same as introducing a NOP instruction, commonly known as a NOP bubble. The control unit keeps track of the hazard until the result from the EX stage has been completed, after which the stalls are lifted and the instructions continues as expected.

2.2 Advanced High-performance Bus

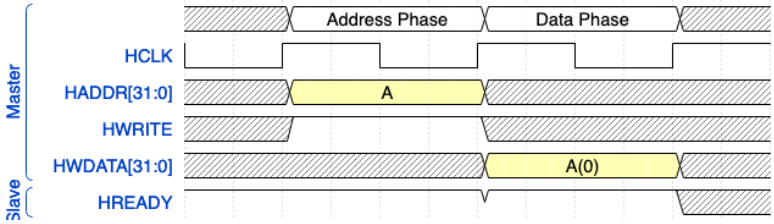
For a SoC, effective communication between cores and peripheral units are of high importance for the overall system performance. Currently there are multiple communication protocols that realizes effective on-chip communication. One of these is the AHB, which is available in multiple versions. In this thesis, the term AHB refers exclusibely to the AHB3:Lite protocol which is a streamlined and lighter version of the larger AHB3 protocol.
The AHB3:Lite protocol offers a subset of the signals of its parent protocol, in order to simplify the design and reduce complexity. Depending on the implementation of the protocol, the signals may vary slightly. However the main signals used for the basic transactions are as shown in Fig. 2.2. The bus is pipelined, meaning that the following address phase can begin at the same time as the data phase from the previous transaction starts. It is also possible, and often the case, that the transactions require wait states as either the master or the slave may not be ready to enter the data phase. [9]

2.2.1 Interconnect

An *interconnect* enables multiple slaves to be connected to each master, as is shown in Fig. 2.3. Through the use of a decoder, based on the address mapping of the slaves, a unique select signal, **HSEL**, is routed to each of the slaves. This indicates whether the slave is active or inactive. The interconnect in this thesis was provided by Codasip, and is configurable with multiple masters, slaves, and decoders. However, it is important to note that the use of an interconnect alone does not allow single slaves to have multiple masters. Thus, if multiple masters are present in the interconnect, they must be completely separated.



(a) Basic AHB read transaction.



(b) Basic AHB write transaction.

Figure 2.2: Waveforms for AHB transactions without any wait states, i.e., the transactions are completed in two clock cycles.

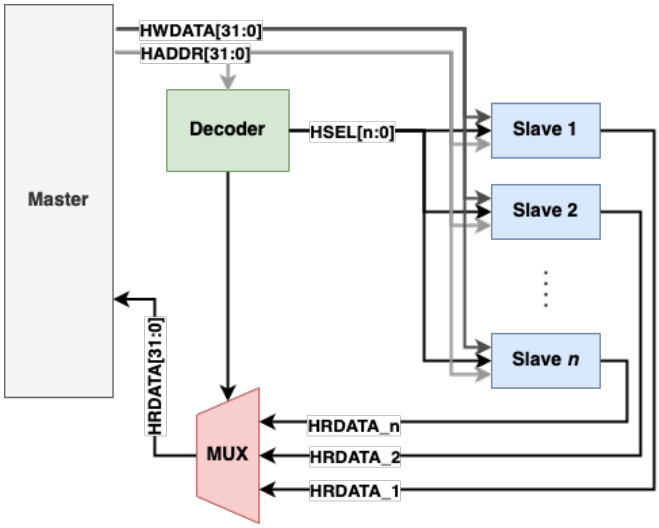


Figure 2.3: AHB block diagram to visualize the address and data bus routing with the use of an *interconnect*.

2.2.2 Arbiter

As previously discussed in section 2.2, the AHB can support multiple slaves for each master. However, the bus does not inherently support multiple masters for a single slave. This limitation can cause conflicts in systems where multiple master

units requests access to a single slave unit at the same time.

An arbiter is a digital circuit that helps resolve these conflicts by managing access to the shared slave. Effectively, it combines multiple master interfaces into a single master interface. The arbiter implements a policy to determine which master's request should be given priority during concurrent requests. Depending on the system's requirements, this policy could follow various protocols, such as *first-come, first-served*, or be based purely on predefined priorities.

The arbiters provided by Codasip for this thesis use a strictly priority-based policy. This means that during the configuration, the user sets a priority list, which determines the priority of the master interfaces. If multiple master units make requests, the arbiter prioritizes the request from the highest priority master and transmits it to the slave within the same cycle. Meanwhile, it buffers the remaining requests, forwards them to the slave in the following clock cycles.

2.3 Near Memory Compute

The traditional Von Neumann architecture contains a Central Processing Unit (CPU) and memory, where the data and fetch bus are shared. Accordingly, instruction and data memory access can not happen in parallel, leading to the von Neumann bottleneck resulting in wait times on the bus. This problem only gets worse as more data processing is required, which is a problem especially in data-intensive applications as AI.

Further increasing strain on the issue is the differing exponential growth rates of CPU and memory speeds for the past decades. Though both units have had an exponential growth, the CPU's speed increase outpaces that of the memory, leading to a bottleneck where the CPU spends more time waiting for data from memory, rather than performing computations. As a consequence, no matter the speed of the CPU, the system will be limited by the transfer rate of the system bus, as increasing the CPU speed will only lead to the processor sitting idle for greater amounts of time. [6, 7]

NMC offers to alleviate the strain on the bus by bringing the processing unit closer to, or even within the memory itself. By doing so, the need to transfer large amounts of data across the memory bus to the CPU is significantly reduced, thereby improving overall system performance. Adding the accelerator in close proximity to the memory fundamentally alters the data transmissions, in comparison to the classic Von Neumann model. This approach creates a more data-centric architecture for more efficient handling of data-intensive tasks in modern computing systems.

Since the first proposal of NMC as a method to bridge the gap between computation and memory performance, a wide range of NMC architectures have been

introduced. NMC, often referred to as Processing-in Memory (PIM) or Near-Data Processing (NDP), is an umbrella term for all types of computations performed in close proximity to a storage unit in the memory hierarchy. Recent advancements in 2.5D and 3D stacked memories have lead to improvements in Computation-in-Memory (CIM) architectures in which processing elements are directly incorporated into the memory system, further reducing data movement since the memories themselves possess computational capabilities. [2]

This architecture requires memories specifically designed for this purpose and integrating an already existing accelerator requires heavy modification of the memory architecture. Implementation of Multiply And Accumulate (MAC) operations as CIM can be achieved by utilizing the word-line as a multiplier, subsequently accumulating the resultant products. However, enabling more sophisticated algorithms with CIM tends to be more complex compared to traditional digital design methodologies. [10, 3]

In contrast to CIM, the less intrusive NMC architecture can use standard memories and integrate existing accelerators in a less disruptive manner. This type of NMC enables for programmable processing units as well as fixed-functional ones. Though possible to implement NMC on all types of memory systems, the most researched interpretation is processing near main memory. [3, 4]

2.3.1 Direct Memory Access

Efficient use of processing power in a system is crucial for an optimal performance. DMA is a technique which improves the system efficiency by allowing subsystems such as peripheral units to access the system memory independently of the CPU. In traditional systems any data passed between the memory and any component, or external interface, has to be passed through the CPU causing the core to spend a significant portion of time managing data transfers.

In contrast, DMA bypasses the CPU by allowing subsystems to access the system memory directly. This relieves the CPU from handling low-level data transfers, freeing it up to perform critical tasks. A DMA is particularly beneficial in systems dealing with large amount of data and data-intensive tasks as this lead to a significant improvement in the overall system performance.

2.3.2 Accelerator

The accelerator used in this work is a CNN accelerator developed with NMC in mind. It was designed to improve energy efficiency by reducing data transmissions through data reuse. The accelerator performs 2D-convolution, linear rectifying and pooling. The accelerator was chosen due to its simplicity in combination with being suitable for the purpose of the thesis [11].

The accelerator was validated with the CIFAR-10 dataset, and is able to perform 2D-convolution with channel dimensions of 8×8, 16×16, 32×32 and a 3×3 filter size. The accelerator inherently supports up to 256 filters and equally many channels, however limitations are imposed on the accelerator as the original memory size of 176 kB have been strictly reduced to work with the system of the thesis. [11] contains three memories, one single-port 2 kB memory storing one 8×8 16-bit pixels for the Input Feature Map (IFM). Another separate 2 kB dual-port memory stores the pixels of the Output Feature Map (OFM). The third memory is used to store weights and biases, as well as for setting configurations for the accelerator.

Writing and reading data to the accelerator is performed directly to the Block Random Access Memory (BRAM)/SRAM native interface, with some additional control signals to determine which of the three memories should be accessed, see Fig. 2.4. The native Random Access Memory (RAM) ports consists of an address, a data input, and a data output bus as well as the controls signals chip enable and write enable.

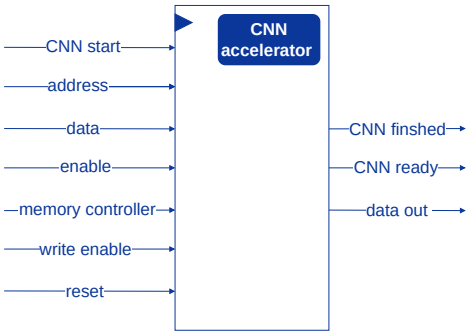
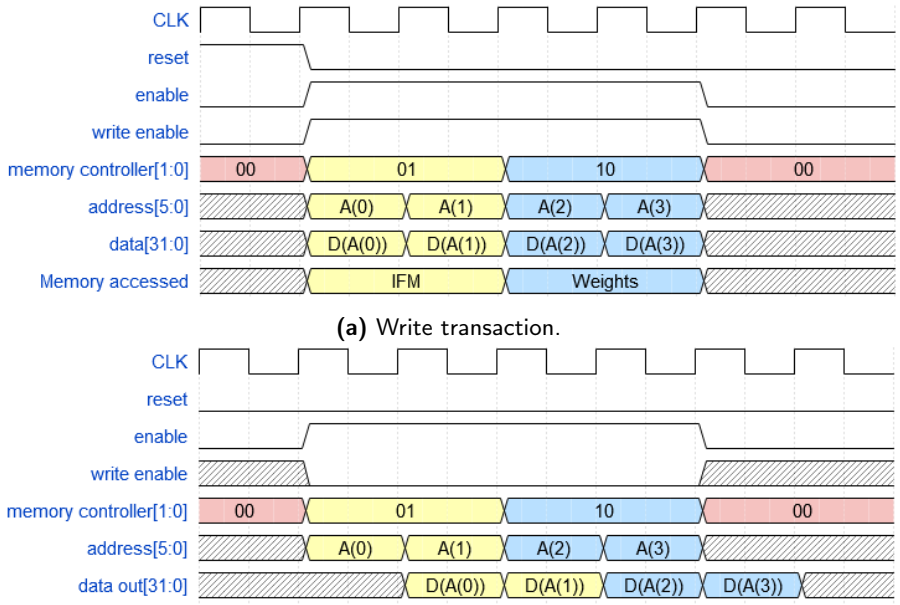


Figure 2.4: Interface of the CNN accelerator, showing the input ports on the leftmost side and the output ports on the right side.

The values on the input port **memory controller** determines which memory is accessed. The write transaction is done in a single clock cycle with input data, **data**, and **address**, sent in parallel. See Fig. 2.5a. In contrast, the read transactions require two clock cycles with the input address preceding and output data sampled at the next rising clock edge, see Fig. 2.5b.

2.4 Memories

Understanding the unique characteristics and constraints of memory types is crucial in system design, as the choices here significantly impact not only overall performance and efficiency but also the system's area and power consumption.



(a) Write transaction.

(b) Read transaction.

Figure 2.5: Waveforms of data transactions of the accelerator, with the all signals but *memory controller* makes up the native RAM interface. The color coding indicates which segments belong to which memory, depending on the state of memory controller.

This section focuses on two key memory types used in this project: Cache memories and SRAM. The discussion will delve into their specific roles within a system, their features, and how they influence the system's design considerations.

2.4.1 Caches

Caches are a type of temporary storage placed between the memory and the CPU, which can help mitigate the von Neumann bottleneck. Caches can improve overall efficiency as they generally have faster access times compared to RAM. This speed advantage is partly due to the fact that caches are typically implemented using SRAM, whereas RAM is usually implemented using Dynamic Random Access Memory (DRAM). The proximity of the cache to the CPU compared to the RAM also contributes to the shorter access time.

Cache configurations vary, often divided into levels: L1, L2, and L3, with L1 closest to the CPU. Cache levels and sizes depend on system requirements, but conventionally, L1 is the smallest and often on the processor chip. L1 caches are sometimes split into separate data (L1-D) and instruction (L1-I) caches. L2

caches, larger but slower than L1, can be on the processor chip or a separate chip. In multi-core systems, there may exist shared caches, see Fig. 2.6.

Caches are smaller than main memory and can only hold a subset of memory locations. They are transparent to cores, meaning cores are unaware of the caches and address only the main memory locations. During a memory access, the L1 cache controller checks if the required memory location is in the cache. If present, the core reads or writes directly to the cache, resulting in a reduced access time. This is known as a cache hit. If the data is not in the cache, a cache miss occurs, and the data is fetched from higher-level caches or main memory. Caches are organized into cache lines, which are blocks of multiple words. During a cache miss, an entire cache line is fetched and stored in the cache.

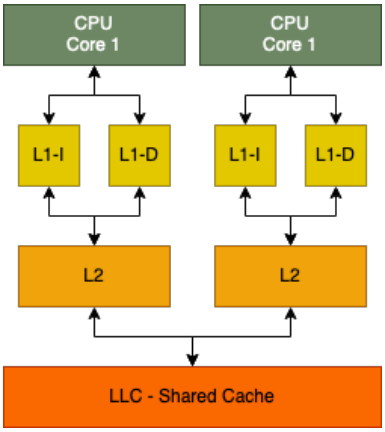


Figure 2.6: Example of possible cache configuration in a multi-core system in which the cores have separate L1 and L2 caches and a shared L3 cache (LLC).

Cache coherence is critical when using cache memories, especially in multi-core systems with shared data resources. When multiple caches store copies of the same memory location, updates to one copy can lead to inconsistent data across caches. Various protocols have been developed to address this issue and maintain cache coherence. These protocols impact the system performance in different ways and are tailored for specific system configurations.

The cache memories provided to this thesis is a set-associative cache memory. A single bit is used as a status bit, *dirty*, to indicate that a cache line has been modified. This may not be sufficient for all system configurations but additional measures can be taken in software. However this puts a burden on the programmer and might not be as effective. Cache coherency can be maintained by using the supported D-cache operations "invalidate line" and "flush line" in conjunction with handshakes between the multiple hosts, such as a core and an accelerator. Invalidating a cache line removes the cache line from the cache, forcing the data to be read again from main memory, thus the most recent version of the data can

Table 2.3: Minimum and maximum configuration of SRAM provided for the ASIC design.[1]

(a) Single-port SRAM.

Feature	Range	
	High-Performance	High-Density
Size	2 kb - 1 Mb	32 kb - 2 Mb
Data width	8 - 256 bits	8 - 256 bits
Word Depth	64 - 16384	1088 - 32768

(b) Dual-port SRAM.

Feature	Range	
	High-Performance	High-Density
Size	1 kb - 160 kb	8448 b - 320 Mb
Data width	4 - 80 bits	4 - 80 bits
Word Depth	64 - 8192	528 - 16384

be obtained. Flushing a cache line writes the data to memory regardless of the replacement policy.

2.4.2 Static Random Access Memory

The SRAM memories used for the ASIC design in the thesis, were provided by Invecas for GlobalFoundries®(GF) 22FDX™22 *nm* technology. Numerous memories are provided, with both single and dual port. Both memory types are available in multiple families, with their own respective frequency performance range and power/area.

The memories can be compiled, for different word depths and data widths, with some limitations as shown in Tab. 2.3. Each memory type can be configured into either a high-density or high-performance mode. Though the high-performance is limited to significantly smaller memory sizes.

Depending on the word depth and data width, some memories may be compiled into physically inconvenient layouts which can significantly affect the layout process. Hence, structuring memories through a process known as banking can effectively streamline the place and route procedure, minimizing potential layout issues.

Restrictions in the block placement of the memories can pose additional challanges

[1] Please note that the table presents the absolute minimum and maximum ranges. However, not all of these configurations may be compatible with each other.

during layout, as no rotations are allowed. However it is possible to mirror the block in y-axis.

A native RAM interface is used to read and write transactions. meaning that write requests are executed in a single clock cycle while read requests require two clock cycles.

2.5 ASIC Design

In ASIC design, the three major design goals are performance, power, and area (PPA). Each of these design goals may negatively impact the other two, so a balance between them must be carefully considered at each design step, and the PPA should be optimized based on the application. During this thesis, the target application is edge inference, therefore the main emphasis is on power efficiency and performance, but the area should still be kept as small as possible to increase yield and lower cost. Trade-offs with regards to PPA will be discussed in the System Design and Implementation section, see section 3.

2.5.1 Low-power design

In low-power design it is common to use multiple power domains to reduce the overall power consumption. This can be achieved by enabling logic to turn of specific power domains or to use physical power domains, i.e., domains that are physically separated and can use different voltage supplies. When creating physical power domains there are two major aspects that must be considered: Physically separate power and core rings, and power routing between different power domains on the die area. Additionally, partition the pad frame into power domain-specific parts using breaker pads.

To physically separate the power routing, it is important to leave a sufficient gap between the power-rings of different power domains and carefully define the power domains properly in the Electronic Design Automation (EDA)-tool, the distance between different power lines is a process dependent variable and should be accessible among the process' design rule.

To partition the pad frame for the different power domains, breaker pads are used to break the supply rails in the pad frame, though the ground rail may be shorted if desired. For digital pads it is common practice to drive the pads with a higher voltage. An example of a digital pad-frame with two physical power domains can be seen in figure 2.7. Where the powerlines of different power domains are cut by the blue breakers.

After power routing and placing of standard cells and hard macros, the IR-drop

(voltage drop) for all power domains must be verified. If the IR-drop is significantly high somewhere on the die, the power planning needs to be revisited to avoid any performance loss.

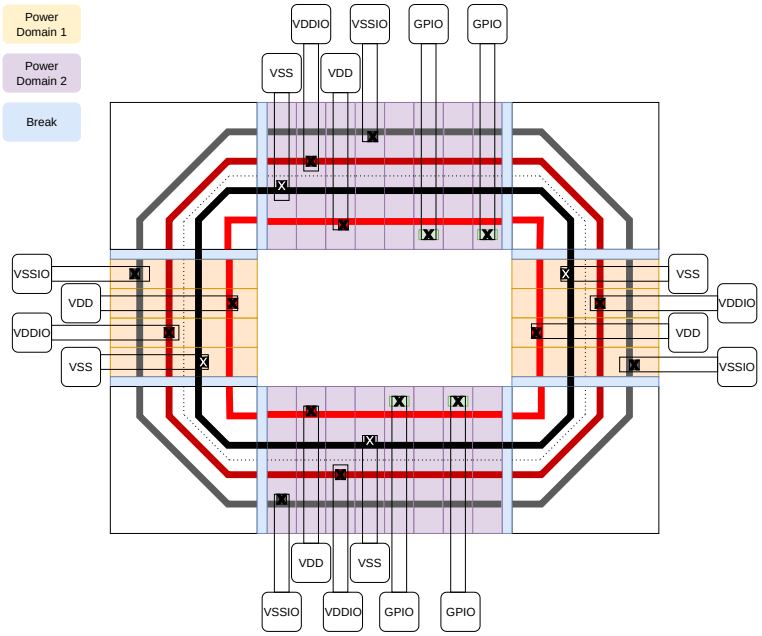


Figure 2.7: A digital pad frame with two power domains. The IO power rail are driven by the VSSIO and VDDIO, while the core power rail is driven by VDD and VSS, which supplies the two different power domains.

Chapter 3

System Design and Implementation

This chapter serves to provide a deeper understanding of the system design and how it was implemented. The key components that form the architecture of the system will be explained as well as challenges encountered during the implementation process, and how they were addressed. This comprehensive presentation of the design and implementation process should provide a clear picture of how the system was built and how it operates.

3.1 Overall Design and Implementation

The original design was a single RISC-V core system with one level of cache memories and one larger on-chip memory connected via an interconnect. In order to connect the AHB from the I- & D-cache, to the interconnect, arbitration is required. Besides the NMC accelerator, the system also contained a DMA attached to the core via instruction level control. The DMA needs to be able to read and store data from the memories, thus requiring an AHB master interface connected to the arbiter, as well as another master interface connected to the accelerator. As an option to the instruction level control, the DMA may also be controlled via a slave interface attached on the interconnect. See Fig. 3.1.

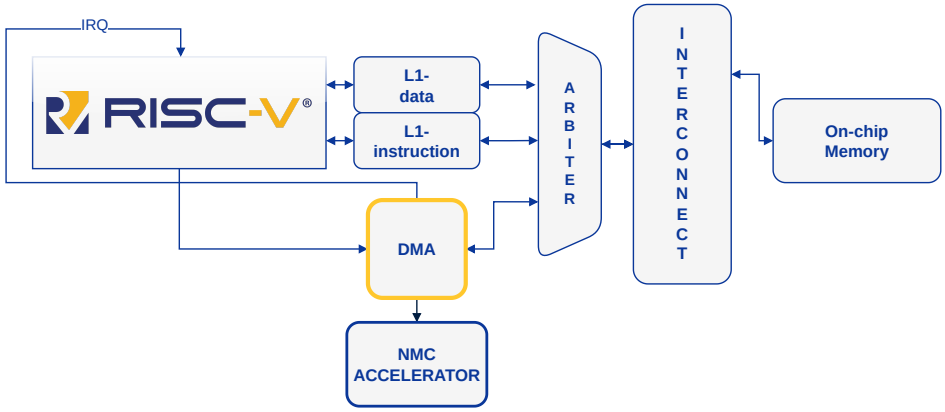


Figure 3.1: Initially proposed design of the system.

As explained in section 2.4.1, the caches provided in the project does not support a protocol that can handle the intended system configuration. To handle this, a the built in functions *flush* and *invalidate* which can be called from the RISC-V core can resolve the coherency. Prior to the core providing an address to DMA, from which data will be obtained, the core has to trigger a cache flush. Equally, when the DMA has written to the main memory and transmits an IRQ to the core, the core will respond by triggering a cache invalidation, before reading that recently updated data.

3.1.1 System Redesign

During the late development phase of the thesis, it became apparent that the proposed system was incompatible with the cache system provided for the project. This constraint hindered the thesis and as a consequence it was deemed necessary to revisit the system design and consider alternative approaches while retaining as much of the original idea as possible.

The alternative system design, can be seen in Fig. 3.2, which replaces the cache memories in favor of an additional SRAM. With this setup it is still possible to maintain parallelism between the calculations in the NMC the RISC-V core. Both memories are accessible to both hosts, though the larger of the two memories (Memory 1, in the figure) is the primary memory for the RISC-V core while the other memory is primarily intended to the NMC unit. The bus from both the DMA and the RISC-V core is attached to the interconnect in which two decoders are used to connect them to the appropriate memories via another arbiter.

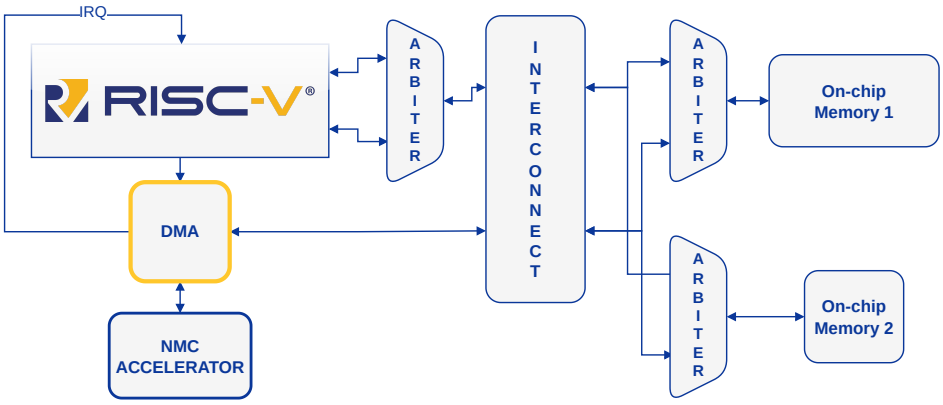


Figure 3.2: Block diagram for the final system design, with the caches replaced by a secondary SRAM memory.

3.2 RISC-V modifications

This section is dedicated to the modifications of the RISC-V core, required for the proposed system design to work as intended.

3.2.1 NMC specific instructions

Communicating with the accelerator, in particular the DMA, through use of customized instruction requires the instruction to be able to transmit data in a single clock cycle containing information about memory locations, DMA specific configuration, as well as as information about which DMA is to be addressed. In this system configuration, there is only one DMA present, but for scalability it is possible to add more DMAs without making any changes to the ISA.

The S-type instruction was a suitable instruction type as this can provide two register sources in combination with 12 immediate bits, see Tab. 3.1. As only one DMA was attached in the current configuration, the first seven bits are redundant, but are otherwise used to address a specific DMA. The five bits composed of *Opt1* and *Opt2* were added for the same reason, to introduce later customization to improve and maximize the total number of bits possible to transmit in a single clock cycle.

Instructions of a mathematically simple nature, can easily be described in the CodAl semantics in such a way that the compiler automatically can identify when the C-code should be compiled to that specific instruction. However, an instruction of this nature which is more abstract can not be easily identified by a compiler as it is not an operation like addition, multiplication, bit shift, etc.

Table 3.1: Customized DMA instruction.

Bit	31 .. 25	24 .. 20	19 .. 15	11.. 9	8 .. 7
S	imm[11:5]	rs2	rs1	imm[4:0]	
Desc	DMA	Mem	Control	Opt1	Opt2

For this reason the compiler is not able to automatically identify which parts of the C-code should be compiled to this custom instruction. The use of the specific instruction (hence referred to as **SWC**) requires inline assembly, as follows. This poses some restrictions on the software programmer as the immediate bits for DMA address (*dma_ addr*) and the optional (*opt1*, *opt2*) has to be determined at compile time and can't be changed during runtime.

Listing 3.1 Definition for inline assembly use of the custom DMA instruction, SWC

```
1: #define SWC(mem_addr, ctrl, dma_addr, opt1, opt2) asm
       volatile ("swc %0, %1, %2, %3, %4" : : "r" (
      mem_addr), "r" (ctrl), "i" (dma_addr), "i" (opt1),
       "i" (opt2) )
```

In order for the DMA units to determine when to sample the data, an additional signal (*Write Enable*) is used to trigger the data on the input buses on the DMA, see Fig. 3.3.

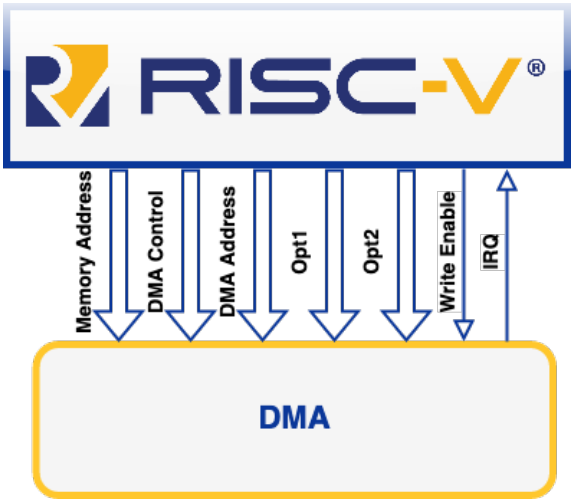


Figure 3.3: Buses and signals between the RISC-V core and DMA used for instruction level control.

The instruction was completely implemented through the use of CodAl as both an instruction and cycle accurate model. Handling the hazards that can arise in

cycle accurate model, is crucial for a functioning instruction as there are numerous multi-cycle instructions that are likely to be used prior to the use of the SWC, e.g., **LW**, see section 2.1.3.

To resolve this issue, in the ID stage the core keeps track of the FU being decoded at the current time. If the decoded FU is the one used to handle the SWC instruction, there is a potential for a instruction hazard. However, it is likely that no hazard will occur, hence the core needs to compare the destination register in the EX stage to source register in the ID stage. If there's a match, the ID stage will stall, inducing a stall on the FE stage as well.

3.2.2 Multiply and Accumulate

The MAC instruction is a simple instructions often used used in convolution, which calculates the product of two integers and accumulating the result with a third, in a single clock cycle. The instructions takes three parameters of which one is both input and output while the others are inputs. The R-type is a suitable instruction type for the MAC, see Tab. 2.2. Due to the mathematical simplicity of the MAC instruction, this can be explained through the CodAl Semantics language in such a way that the compiler easily detects C-code that should be compiled into MAC, as shown below.

Listing 3.2 MAC instruction

```
1: int a = 5;
2: int b = 10;
3: int c = 20;
4: c+=a*b; // Compiled to mac instruction
5: c = c+(a*b) // Compiled to mac instruction
```

The MAC instruction was realized through a combination of CodAl and VHDL as to provide the flow with the possibility to implement an instruction fully, or partly, using Hardware Description Language (HDL), which in some instances may be more appropriate than using CodAl. The actual calculations were implemented using VHDL, while CodAl was utilized to handle hazards and incorporate necessary architectural modifications in the design.

As the baseline processor only supports two values to be read from the GPR at single time, a third reading port was implemented. Hazards were handled in a similar way as explained in section 2.1.3 and 3.2.1.

3.3 Near Memory Computing

In this thesis, the NMC design involves the realization of a DMA and an accelerator. The following sections provide a in-depth explanation of the design exploration choices and the implementations of the respective components.

3.3.1 Accelerator

The accelerator used in this design is a smaller implementation of the design proposed and implemented by S. Castello in his previous work [11]. To be able to tightly couple the accelerator with the RISC-V core, new memories in $22nm$ technology was compiled and integrated. Additionally an AHB interface between the accelerator and the interconnect was designed. Since AHB is a multi-clock cycle protocol, as seen in Fig. 2.2, the interface from AHB needed to be designed to align address and the data. This can be seen in Fig. 3.4, where delays are introduced to address this alignment issue.

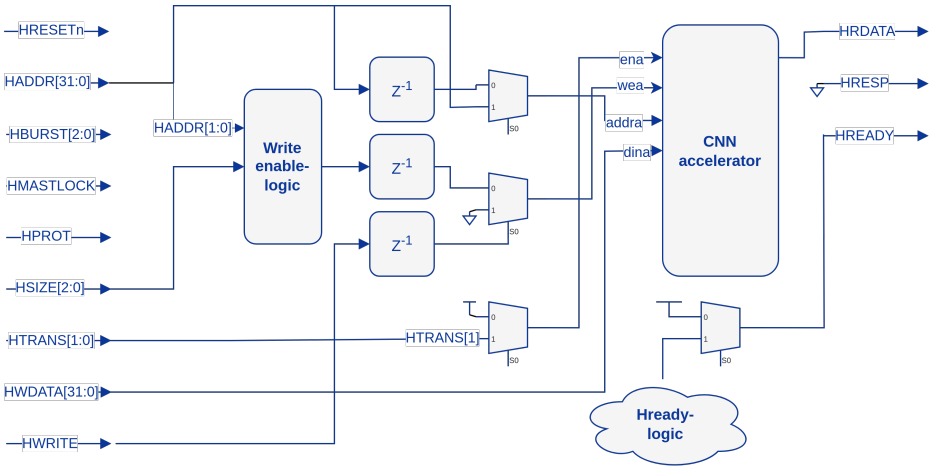


Figure 3.4: Block diagram of the AHB to accelerator bridge.

3.3.2 Direct Memory Access

In applications that demands high data movement, it is important that the processor offloads data transfers to improve system performance. A common methodology to achieve this is to use a DMA unit. A DMA is configured with a start address, destination address and number of transfers and issues read and write request until all the data are transferred between the two memory locations.

DMAs can be optimized for different types of applications, in this work a low area, high throughput and modular DMA to move data from memory into different accelerators. The design consist of a front-end, mid-end and a back-end as seen in Fig. 3.5, the modularity of the designs enables reusability and the option for improvement or extensions to the design. The front-end offers a programming interface between the system and the DMA, the mid-end consist of the DMA-controller and the back-end act as a converter between the interface of the DMA to the specific on-chip protocol.

By decoupling the front and back-end from the control-logic of the DMA, different on-chip protocols can be used to configure the DMA and for the DMA to interface with the memory system.

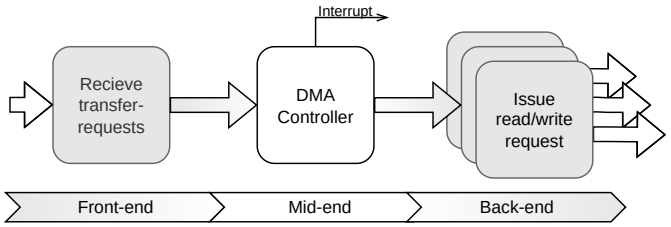


Figure 3.5: Block level functionality of each module in the DMA.

Architecture

The front-end is implemented to either be configured by custom RISC-V instructions or a register-based configuration. Both methods configures the same registers in the DMA, but the instruction-based method is able to configure a transfer request with only one instruction. When a transfer request have reached the front-end it gets polled by the mid-end controller to be handled further. The control, and address FIFO in the front-end module, as seen in Fig. 3.6 act as a scheduler/buffer when the controller are occupied with a transfer. This enables the user to schedule multiple transfers on the DMA, and while utilizing the RISC-V core for other task. When the DMA have transferred data back to the memory, it interrupts the RISC-V core to notify that the data have been transferred. In Fig. 3.6 one AHB master interface is connected directly to the accelerator, multiple accelerators can be connected and targeted by changing *target accelerator* in the control data register. The controller is made to be able to issue read request and write request to the back-end every clock cycle. The DMA is currently limited by the AHB master back-end which only can read data every third clock cycle and write every second. But due to the modularity of the design this could be improved later on.

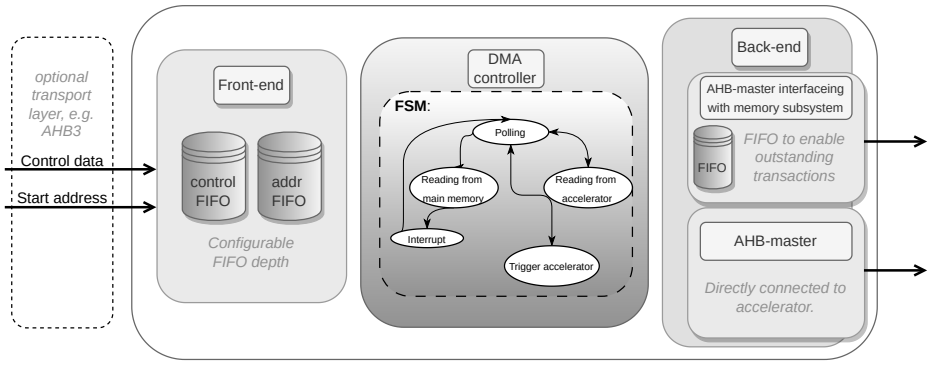


Figure 3.6: In-depth description of the hardware and logic used to implement the DMA.

Programmable configuration

To schedule a transfer request the **start address** and the **control data** must be set correctly. The start address is a pointer to the memory address that from which the DMA read. The control data register contains the *number of transactions*, *bit shift*, *accelerator address*, *target accelerator* and *mode*. These parameters configures how the data shall be moved between accelerator and memory and the bit assignment for the control data register is shown in Tab. 3.2. The current version supports up to 4 accelerators per DMA, where each accelerator have a private memory space.

Hardware configuration for the interrupt

To connect the interrupt to the RISC-V core the interrupt-signal needed to be coupled to the right register in the RISC-V core. The DMA-interrupt was connected to the Least Significant Bit (LSB) of p_int which is the interrupt vector that internally drives the machine interrupt pending (`MIP`) register and especially the `MIP.MEIP` where `E` stands for external. It also trigger the `MPICFLAG` register, which will store the interrupt. During run time the `MIP` triggers the core to start executing the trap handler. From the `MPICFLAG` register the source of the interrupt can be resolved and linked to the accelerator. [12]

3.4 Memories

In order to minimize the die area, the memories should preferably be compiled in the high-density mode. However due to limitations in the provided memory

Table 3.2: Programming guide for the DMA, to use with the SWC instruction.

31 29	28 27	26 15	14 12	11 1	0
MODE	ACC	ACC ADDR	BITSH	NRTX	START

(a) DMA control data register bit assignment.

Abbreviation	Meaning	Clarification
MODE	Mode	000: Read from memory, 010: Read from accelerator, 001: Trigger accelerator.
ACC	Accelerator	Accelerator selection.
ACC ADDR	Accelerator Address	Start address for accelerator
BITSH	Bit Shift	3 bits represent an unsigned int, which will cause a stride when reading/ writing from memory by $2^{\texttt{BITSH}}$.
NRTX	Number of Transactions.	
START	Start Bit	Must be set to 1.

(b) Abbreviations and clarifications.

compiler, the combination of data widths and word depths are only possible in the high-performance mode, see section 2.4.2.

The accelerator requires two 2 kB single-port memories and one 2 kB dual-port memory, as explained in section 2.3.2. The dual-port memory, with its more strict limitations (see Tab. 2.3b), required the use of a high-performance memory. The two single-port memories were also compiled in the preferred high-density mode.

Prior to finding out the limitations of the provided caches, as explained in section 3.1.1, the a combination of the limitations of the cache sizes and memory sizes caused issues. The minimum data width of the cache memories for the data banks, as generated by Codasip, was 256 bits. The total data bank size of the caches were 2 kB, i.e., word depth 64. Due to the phenomenally large ratio between the width and depth, this results in an elongated shaped memory which significantly limits the layout process. However, due to system redesign this issue was naturally avoided.

Two separate memories of different sizes, was compiled to fit the the RAM was split into two memories, See Fig. 3.2. The two memories of 96 respectively 32 kB were both compiled in high-density mode, both of which use 32 bits data width.

3.5 Design flow and methodology

When designing an ASIC it is important to have a functional methodology for the project to progress as expected. During this thesis, that consisted of integrating accelerator, designing hardware, and extending the existing architecture of a RISC-V, the use of Codasip studio as the main IDE for both hardware and software, acted as a foundation for the FPGA and ASIC flow. In figure 3.7 each of the major step in both flows are shown, where Codasip Studio act as a starting point for both flows.

For the FPGA-flow it was possible to first generate the VHDL with Xilinx-primitives and simulate the whole system running code using a RTL-simulator. When bugs were resolved, the synthesis, implementation, bit-stream generation, and programming the FPGA (loading bitstream) was performed from Vivado. The same code used during RTL-simulation was then executed on the FPGA, by comparing the behaviour post implementation with the simulation-results, it was possible to early detect bugs and avoid any bad RTL propagating into the ASIC-flow.

During the ASIC-flow, the RTL-generation from Codasip was changed to target a generic ASIC-flow, the memories were changed from FPGA to the pre-compiled memories mentioned in section 2.4.2. Then the same procedure of first running the behavioral simulation followed by a post synthesis simulation was done. Note that all simulations were started from inside the Codasip studio, with different script supporting each simulation. When the design have passed all these simulations, the post Place and route (PnR) simulation was initiated.

Some valuable lessons for all ASIC-simulations except for the behavioral simulation, was to initialize memories and register without a reset, with a random values. This makes the simulation more similar to a fabricated chip and avoids incorrect X propagation during the simulation.

3.6 Synthesis and constraints

The technology utilized in this thesis is the GlobalFoundries®(GF) 22FDX™22 nm Fully-Depleted Silicon-on-Insulator process, with standard cells, memories, and pads sourced from a third-party vendor. Due to the absence of an on-chip clock generator and the input frequency limitation of 300 MHz imposed by the pads, the design is constrained to operate at a maximum of 300 MHz. Additionally, the pads have a depth of 150 μm, which is notably large for small designs. Both high-performance and area-optimized standard cells and memories were available for use. However, since the area-optimized standard cells and memories were capable of handling frequencies above 300 MHz, the area-optimized Intellectual property (IP)s were chosen for this project."

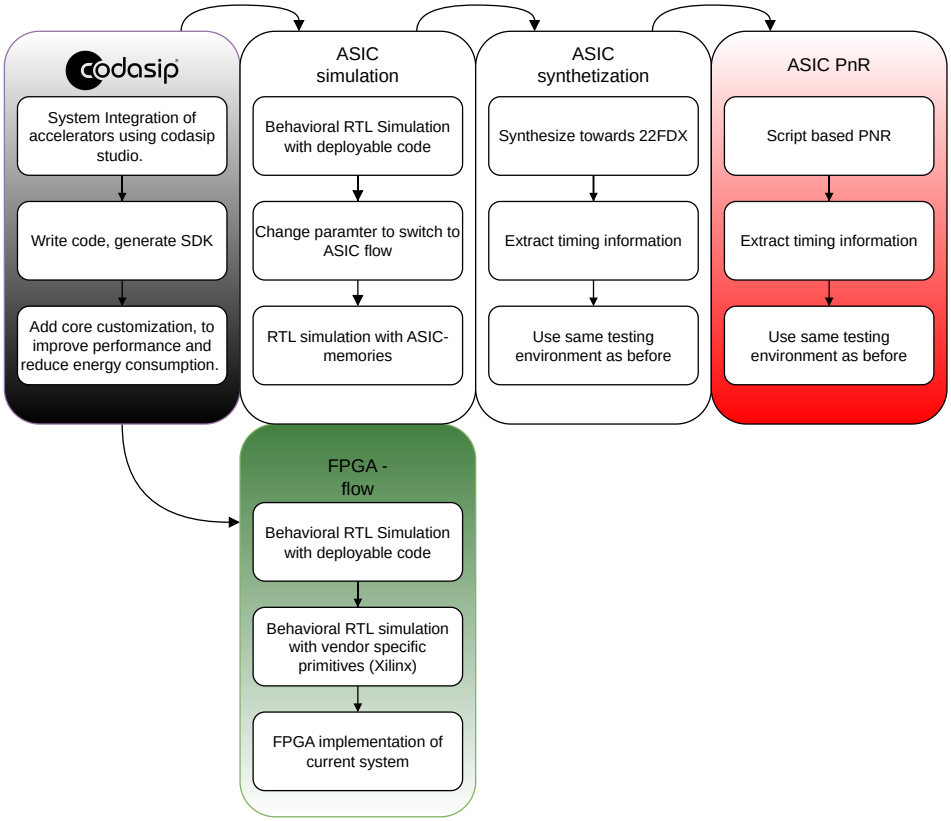


Figure 3.7: Flowchart illustrating the sequential steps taken in both the ASIC and FPGA flow, as well as the preliminary step in Codasip.

3.7 Place and Route

Place and route is one of the more time consuming steps during ASIC-design, therefore automation is key to create a manageable workflow, where it is easy to go back some design steps to test new implementations without needing to redo all the work.

3.7.1 Pad frame

At the beginning of the layout process, a pad frame must be designed. This frame should include all power pads and IO pads, and they must be properly spaced with the correct pitch – the distance between the centers of two pads intended for bonding after fabrication. As various area sizes and pad counts were explored in this project, a script was created to automatically generate the correct pad spacing

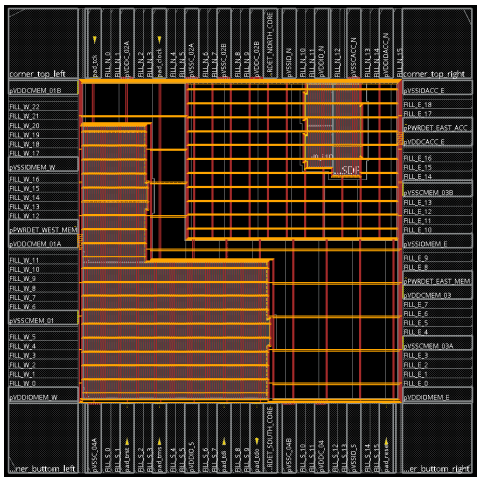
based on the die's geometry, the number of pads on each side, and the specified pad geometry. This script was then used to create the pad frame in the layout suite. Designers only need to specify the order of the pads and the breaks between each iteration. The script automatically rotates and places each cell correctly, and fills in with filler cells as needed. The resulting pad frame is shown in Figure 3.8a.

3.7.2 Power-planning and placement

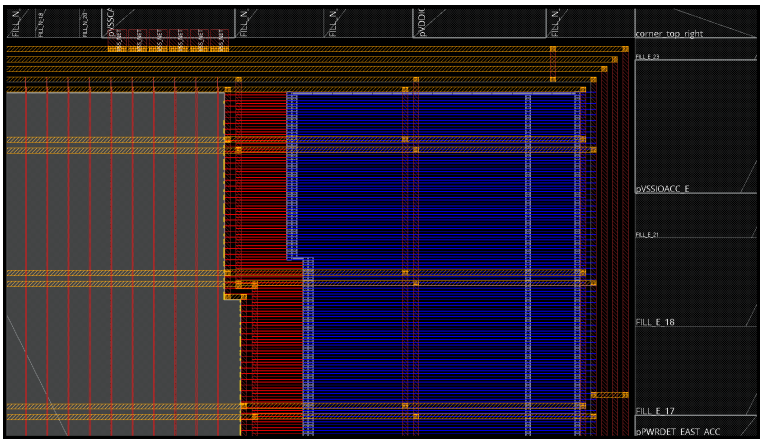
As mentioned in section 2.5.1, using multiple physical power domains is a common strategy for low-power design. In the final implementation, three physical power domains were employed. Although all three had the same supply voltage of 0.8V, the purpose was to enable measurement of the power consumption of different sections of the chip after fabrication. The three power domains were grouped into: the main memories (referred to as PD-MEM), the accelerator and its memories (referred to as PD-ACC), and the rest, which included the RISC-V, DMA, and interconnect (referred to as PD-CORE). In the final design, only PD-MEM and PD-CORE had their own core rings, while PD-ACC only had a dedicated power ring around the area allocated for the accelerator and its memories. The partition of the pad frame for PD-ACC was located only in the top right corner, so it was not necessary to create a core ring for that domain. The core ring for PD-CORE excluded the main memories to avoid unnecessary routing but still included PD-ACC to simplify layout.

During placement, the main memories were positioned along the core edges in accordance with good placement practice. Unfortunately, the memories did not permit rotations or mirroring across the x-axis, so placing them to the left of the core seemed the most optimal solution. The accelerator memories were placed inside the PD-ACC domain. As rotation around the y-axis was acceptable, the memories were placed back to back to simplify power routing during power striping.

For all memories, a specific width between vertical power stripes, as well as the width of each stripe, was specified in the documentation and had to be considered when creating the stripes. End caps were placed at the end of each power domain, and well-taps were positioned closely enough to satisfy the distance rule between well taps. The power rings, power stripes, and end caps are shown in figure 3.8b.



(a) Generated pad frame with macro placement and power routing.



(b) Detailed view of power routing including power rails of the upper right corner.

Figure 3.8: Overview of the power planning, with pad frame.

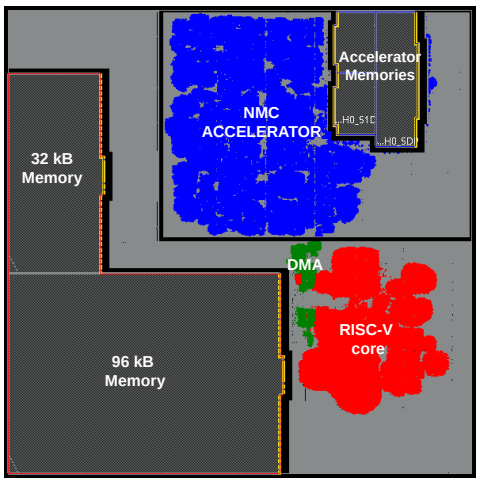


Figure 3.9: Overview of the layout, with each major module highlighted and colored.

3.7.3 IR-drop analysis

After routing, it is essential to conduct an early voltage drop analysis to identify potential voltage drops within the design. As seen in Figure 3.10, the initial voltage drop was notably significant. To mitigate this, the width of the core and power rings was increased. To expedite the development flow, an iterative approach can be adopted to assess the voltage drop after each routing iteration, enabling the entire design flow to be executed only after achieving the desired voltage drop target. The more accurate IR-drop analysis performed after sign-off is termed "late-stage" analysis.

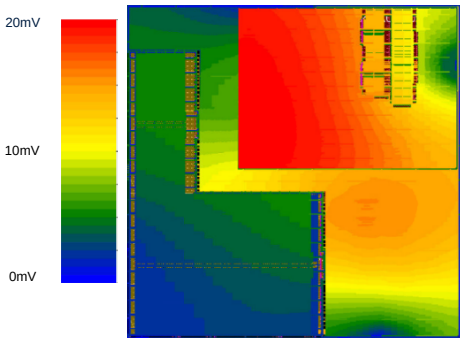


Figure 3.10: Initial IR-drop (voltage) drop on the chip, with a maximum drop of 20 mV.

Chapter 4

Testing and Verification

To test the system, a simple convolutional neural network (CNN) was used, which was trained on the MNIST dataset. The original MNIST dataset consists of monochromatic images of classified handwritten digits with a pixel resolution of 32x32. For this baseline example, the images used for classification were pre-processed to a size of 8x8, resulting in a smaller CNN that uses only one convolutional layer. The architecture of the CNN is presented in Tab. 4.1. To evaluate the system's performance, two different baseline scenarios were tested. In the first scenario, referred to as the **baseline system**, inference was performed on the RISC-V core without any modification. In the second scenario, only the accelerator was added to the interconnect, and this configuration is referred to as the **baseline accelerator**. These baseline scenarios were then compared to the full system configuration, referred to as *this work*, which is depicted in Fig. 3.2, incorporating the MAC and SWC custom instructions.

For a fair and transparent comparison, it was decided that measurements would be taken for different parts of the program, such as the convolution. The metrics used for comparison are based on the number of clock cycles.

Table 4.1: CNN Architecture of the accelerator.

Layer	**Output Size**
Input (8x8 image)	8x8
Convolutional Layer (8 filters, 3x3)	8x8x8
Maxpool Layer (2x2 stride 2)	4x4x8
Flatten	128
Fully Connected Layer	10
Maximum Activation Classifier	1 (one-hot encoding)

4.0.1 Baseline System

The baseline system utilized the L31 core, provided by Codasip. The only modification made to the system was the introduction of an arbiter and a single 128 kB SRAM memory, which was connected to the arbiter via a 32-bit AHB bus. This baseline system does not include any custom instructions or an accelerator.

For the baseline system, the CNN with the settings presented in Tab. 4.1 was implemented in C for inference. The algorithms used for the realization of the convolution, max-pool, and fully connected layer are shown in Algorithm 1, 2, and 3, respectively. These specified algorithms were employed to replicate the calculations performed by the CNN accelerator, as explained in Section 2.3.2.

Algorithm 1 Convolution with Bias

Require: Input tensor I, Filter tensor F, Bias vector B, Pooling size P, Output tensor O

```
function ConvolutionWithBias(I, F, B, P, O)
    for i ← 0 to O.size_1 do
        for j ← 0 to O.size_2 do
            sum ← 0
            for k ← 0 to F.size_1 do
                for l ← 0 to F.size_2 do
                    sum ← sum + I[i + k][j + l] × F[k][l]
                end for
            end for
            sum ← sum + B
            O[i][j] ← sum
        end for
    end for
end function
```

Measurements of execution time were taken at different parts of the code. One of the segments measured was from the start to the end of the convolution, including the max-pooling step, i.e., the combination of Alg. 1, 2. Additional measurements were taken on the fully connected layer as represented by Alg. 3, as well as for the total runtime of the program. The system performance was evaluated at varying optimization levels.

4.0.2 Baseline core with Accelerator

The baseline core with the accelerator is similar to the design presented in Fig. 3.2, except that the DMA is excluded and the accelerator is connected directly via an

Algorithm 2 Maxpool

Require: Input tensor I, Pooling size P, Output tensor O
  function MAXPOOL(I, P, O)
    for $i \leftarrow 0$ **to** O.size$_1$ **do**
      for $j \leftarrow 0$ **to** O.size$_2$ **do**
        max_val $\leftarrow -\infty$
        for $k \leftarrow 0$ **to** P **do**
          for $l \leftarrow 0$ **to** P **do**
            val $\leftarrow I[i \times P + k][j \times P + l]$
            if val $>$ max_val **then**
              max_val $\leftarrow$ val
            end if
          end for
        end for
        $O[i][j] \leftarrow$ max_val
      end for
    end for
  end function

Algorithm 3 Fully Connected

Require: Input vector I, Weight matrix W, Output vector O
  function FULLYCONNECTED(I, W, O)
    for $i \leftarrow 0$ **to** O.size **do**
      sum $\leftarrow 0$
      for $j \leftarrow 0$ **to** I.size **do**
        sum $\leftarrow$ sum $+ I[j] \times W[j][i]$
      end for
      $O[i] \leftarrow$ sum
    end for
  end function

AHB to the interconnect. In this system, without the DMA, the core is required to perform all the data transfers between the memory and the accelerator, which hinders its ability to perform more meaningful computations.

The accelerator performs the convolution computations, while the core is responsible for the computations of the fully connected layer, using Algorithm 3. During inference, the NMC does not allow data to be read from the unit. As a result, for the fully connected layer to execute in parallel with the accelerator, the output from the NMC must be stored in the main memory. This introduces an overhead, as the output is transferred from the NMC to the core and then from the core to the main memory. From there, it can be accessed as needed for the fully connected

computations. However, to achieve good throughput, it is crucial that the core begins transmitting the next input to the accelerator before initiating the fully connected layer. See Fig. 4.1 for a visual representation.

The measurements for the execution time of the convolution computation are defined as starting when the core first reads the input data from the memory and ending when the last output has been written back to the memory.

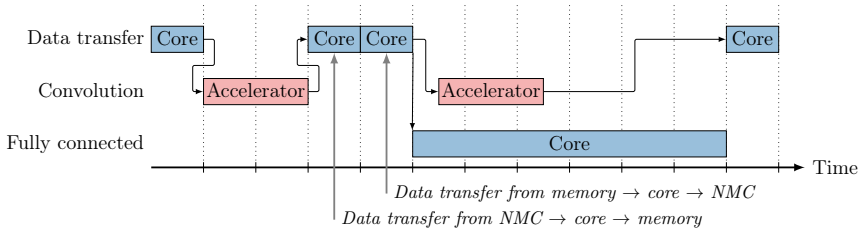


Figure 4.1: Ideal timeline for the operations in the baseline system with the NMC unit connected via the AHB interconnect.

4.0.3 Full System

The system configuration shown in Fig. 3.2, which includes the custom instructions, is referred to as the full system. As in the two base scenarios, the system core performs the operations for the fully connected layer, using Algorithm 3, but in this case, the MAC instruction reduces the number of instructions needed to complete that computation.

When the accelerator has finished the calculations, the DMA will write the result to the memory address previously specified by the core, after which the core is notified through an interrupt transmitted from the DMA. For the Interrupt service routine (ISR) to work, the appropriate registers need to be set in the core. See Appendix A for more details.

The system was measured for both a single convolution in the accelerator and for the fully connected layer. The convolution execution time is defined as starting from the first transaction from the core to the DMA, and ending when the interrupt signal is triggered by the DMA. The setup functions for the IRQ, which are executed once at the startup of the program, are excluded from the measurement. This means that the measurement includes the time it takes for the input data to be transferred to the NMC, as well as the time for the result to be written to memory. The fully connected layer is measured using the same definitions as in the previous setups.

Running the inference and fully connected layer multiple times will yield different results, as both the DMA and the core will be able to access the memories simultaneously, thanks to the dual memory setup. This scenario results in maximum

theoretical performance

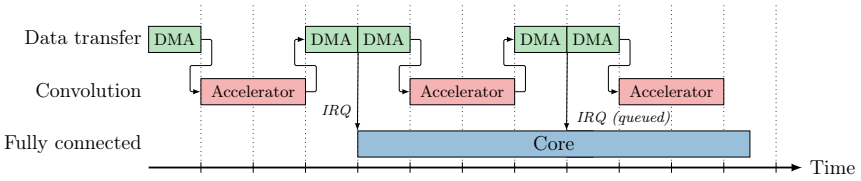


Figure 4.2: Ideal timeline for computations in the full system.

Chapter 5

Results

This chapter presents the results obtained from the various testing scenarios discussed in Chapter 4. The results are based on the previously outlined metrics, which primarily focus on the number of clock cycles required to perform different segments of the program, including computations done in the accelerator, such as convolution and max-pooling, and the fully connected layer. All measurements have been normalized against the baseline system.

5.1 Performance Analysis

As illustrated in Fig. 5.1, the performance of each configuration is measured in terms of execution time relative to the baseline system (*L31*), with lower values indicating better performance. The staple diagram displays the relative execution time of both the convolution and the fully connected layer, the latter of which is always executed in the core.

The results indicate that the addition of the accelerator to the baseline core (*L31 + Accelerator* in Fig. 5.1) considerably reduces the execution time for convolutional computations. As anticipated, the performance of the fully connected layer remains unaffected, as its computations are not handled by the accelerator. This results in an overall performance improvement of 10× for the computations as described in section 4.0.2.

With the additional inclusion of the DMA, and the custom instructions (*This work* in Fig. 5.1), the execution time is further reduced, illustrating the effectiveness of the setup due to parallelism and optimization of computations, as outlined in 4.0.3. In the system equipped with the DMA, a decrease in the execution time for both the NMC and the fully connected layer is observed. This is clearly illustrated in Fig. 5.2, where the performance of the individual computations, in terms of speedup, for the three systems are presented.

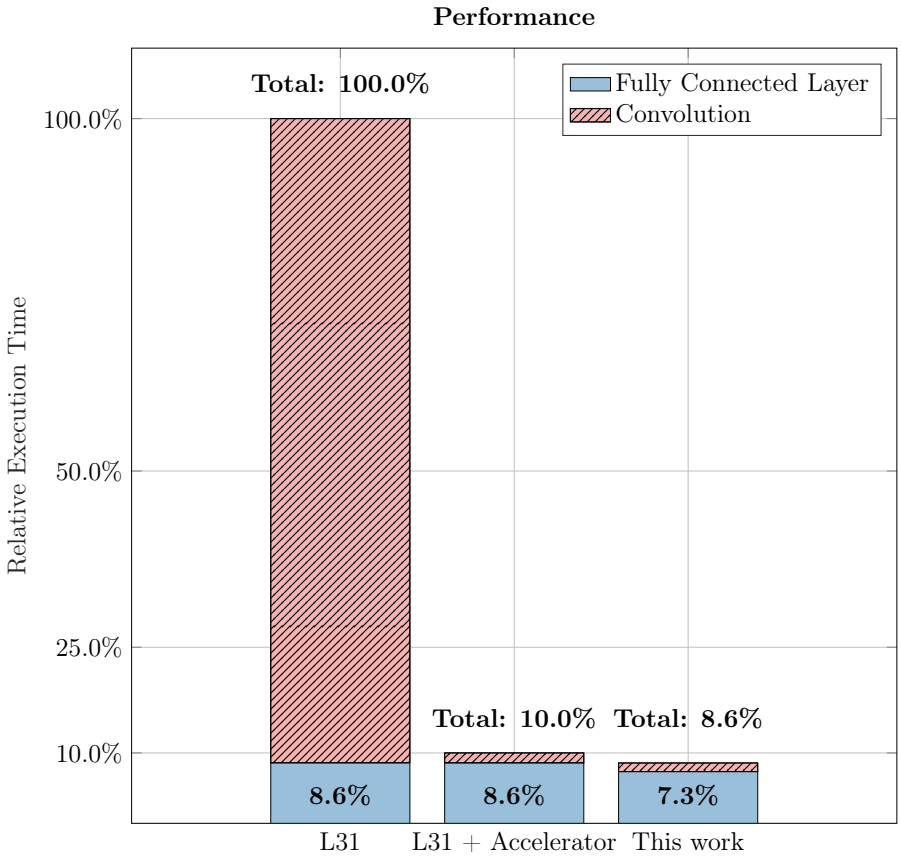
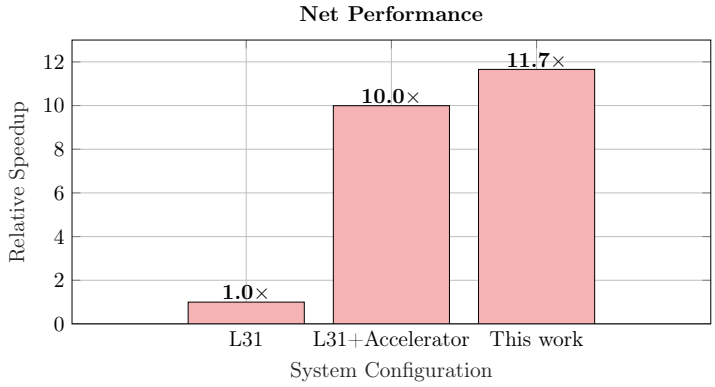


Figure 5.1: Comparative performance analysis, showing the relative execution time for convolutional and fully connected layer, benchmarked against the baseline system, L31.

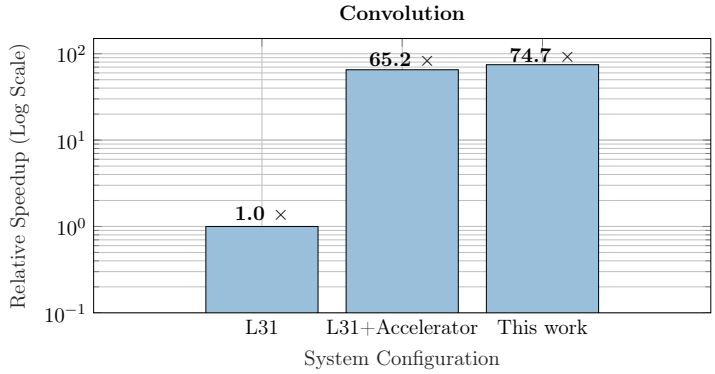
The gain in performance comes at the cost of area, as shown in Fig. 5.3. In comparison to the baseline core, the NMC (including the memories inside the NMC-unit) is 2.94× larger. In comparison to the baseline system (which includes on-chip SRAM), the NMC is 36.7% of the size. In total, the area of the system developed in this thesis is 138.5% that of of the baseline system.

Fig. 5.4 displays the performance relative to the area, normalized to the baseline system. The normalized performance density was determined by dividing the performance of the entire program by the area of the system, and normalizing it against the baseline core. Despite the increase in area, the performance gain was greater, leading to a normalized performance density at 8.4× baseline system.

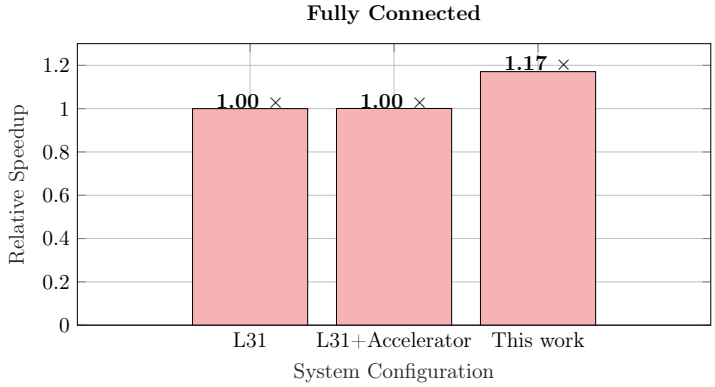
By using the NMC, only a few instructions are required in order to set up the entire inference, the introduction of the DMA reduces the number of instructions further as the data transfers are not handled by the core. The *program size* relative



(a) Total speedup of the program, relative to the baseline processor.



(b) Performance of the convolution computations for the systems, measured as speedup relative to the baseline core.



(c) Performance of the fully connected computations of the systems, in terms of speedup, relative to the baseline core.

Figure 5.2: Speedup comparison between the three systems, for the different computational parts of the program.

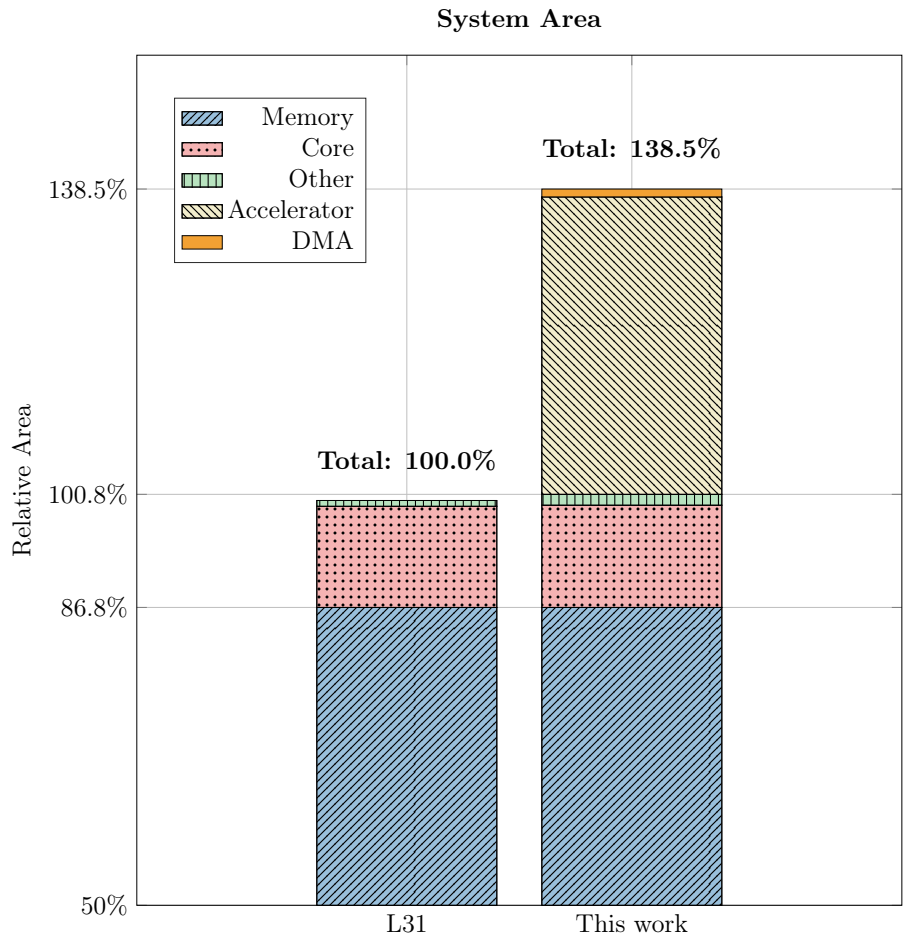


Figure 5.3: Relative area of the system, showing the size of the individual blocks, benchmarked against the baseline core, L31.

to the baseline system is shown in Fig. 5.5. Note that this only accounts for the text of the program (i.e., instructions, not data). As the instruction size for the system with both custom instructions, DMA, and NMC, is only a fraction the size that of the baseline system, it would be possible to reduce the memory on the chip allocated for instructions by approximately 94%.

Due to the large reduction in instruction size, its possible to get a large area reduction by reducing the memory size accordingly. Taking this into account, we get the following relative system size as seen in Fig. 5.6.

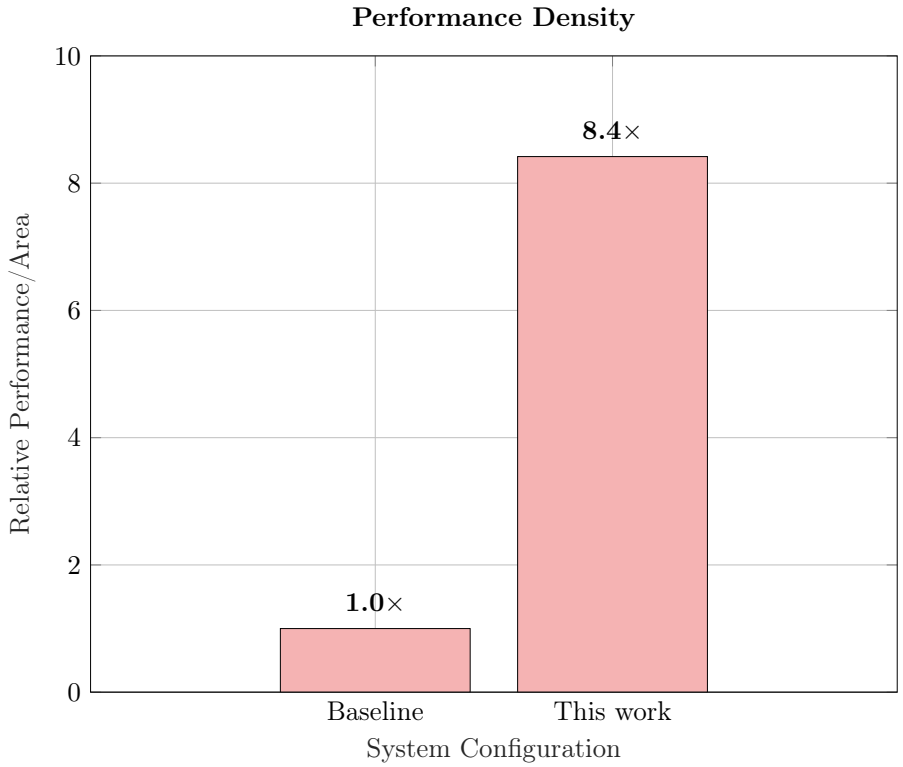


Figure 5.4: Performance over area, normalized to the baseline core.

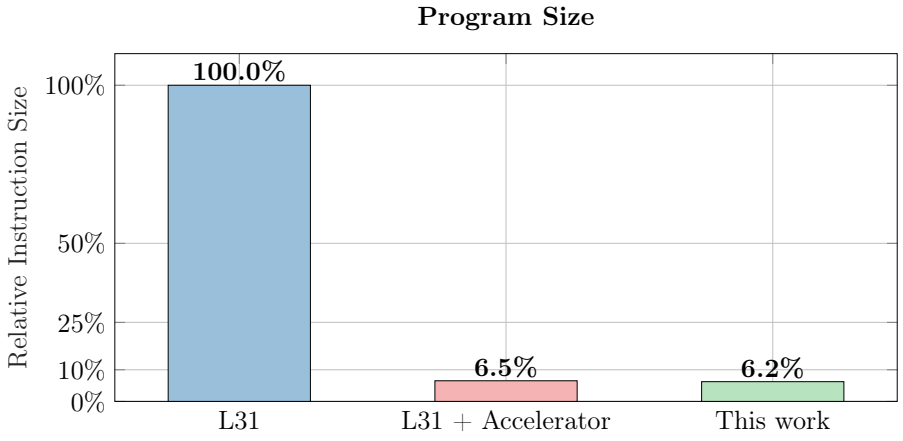


Figure 5.5: Comparative instruction size of the three system configurations, relative to the baseline system.

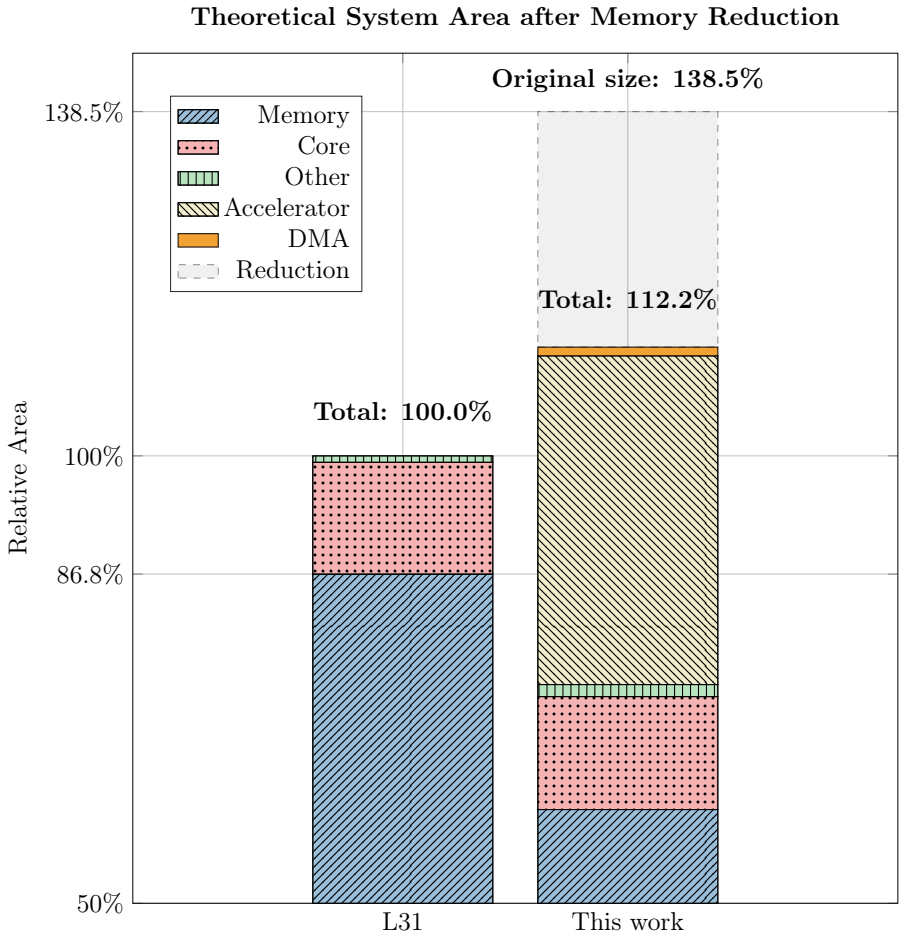


Figure 5.6: Theoretical area after memory reduction as a consequence of the reduction in instruction size.

5.1.1 DMA performance

The DMA designed in this work, was designed to have a small area overhead but still be able accelerate data movement. A short summary of the performance of the DMA is shown in Tab. 5.1. Compared to the scenario without the DMA the read/ writes from the accelerator took 8 and 4 clock cycles, where the writes was faster due to unrolling of loops. It is worth mentioning that unrolling still have an impact on the code size. The DMA also enables more parallelism as discussed in chapter 4.

Table 5.1: Summary of DMA performance.

Read/ Write request (4 bytes)	2/3 clock cycles
Area overhead compared to baseline system	<1%
Front-end (on-chip protocols)	AHB3-Lite or configuration via RISC-V instructions
Back-end (on-chip protocols)	AHB3-Lite

Chapter 6

Discussion and Future Work

This thesis presents a RISC-V based system designed to accelerate convolutional neural networks by offloading convolutions to an NMC-accelerator through the use of a custom-designed DMA. The DMA is precisely controlled by a specialized RISC-V instruction that was added to the core. The fully connected layer was enhanced with the use of a custom MAC instruction. As compared to the baseline scenario, the resulting system exhibits a significant performance increase of 11.7×, with a relatively minor area impact, as discussed in chapter 5.

In hindsight some architectural decisions were less than optimal, particularly regarding the accessibility of the NMC memories and the core. Since the core still needs to fetch the results to compute the fully connected layer, direct access to the memory within the NMC would have been advantageous. This would have reduced the write-back operation performed by the DMA from the NMC to the main memories. Similarly, performance could be enhanced if it were possible to write a new image to the NMC as quickly as the NMC has buffered the previous one. These two improvements would effectively reduce data transfer time to zero when conducting inference for multiple images. This is because, once the NMC-accelerator completes a convolutional layer, it could immediately proceed with the next image, allowing the core to concurrently compute the fully connected layer without waiting for the DMA to transfer data back to memory.

Despite the NMC accelerator achieving a 74.7× performance improvement for the convolutional layer when compared to the baseline scenario, the execution time for the fully connected layer for the final design resulted in a 17% increase as compared to the baseline. Consequently, the overall system performance increase was limited to 11.7×. In order to capitalize on the substantial performance boost provided by the NMC accelerator, it would be worthwhile to explore acceleration options for the fully connected layer. Potential solutions could include modifying the accelerator or incorporating vector multiplication support.

A 10× system performance increase means that the system can be clocked ten

times slower and still perform inference at equal performance as the baseline scenario. This implies that with some dynamic frequency and voltage scaling the system can be optimised for low-power applications. In future work, it could be worthwhile to investigate eventual power consumption reductions from dynamic frequency and power scaling.

With the introduction of the DMA and NMC, the text of the program were drastically reduced to 6.2% of the baseline scenario, as presented in section 5.1. Reducing the instruction size by 90% would yield a significant decrease in the area memory. In total, the full system would require only 60% the size of the memory required by the baseline system. This only holds true for optimized compiling compiling, as the computations result in large amounts of unrolling of loops.

In this work, custom RISC-V instructions were added to simplify the programming model of the NMC and increase performance. To further enhance the programming model, multi-level intermediate representation techniques, as discussed in [13], could be applied to machine learning workloads. This would enable the compilation of high-level model representations in a machine learning framework to executable code. This process would involve the high-level extraction of operations, such as convolution, followed by code generation that configures the DMA and NMC to perform these operations.

In this thesis, considerable emphasis has been placed on automating the FPGA and ASIC flow to enable fast integration of new accelerators and hardware designs. The ability to quickly change architectures and target different platforms facilitates exploration of more design alternatives and their respective performances.

In conclusion, this thesis has demonstrated the possibilities of using attaching NMC units to a customized RISC-V cores. Through the integration of a custom DMA and specialized RISC-V instructions, the system achieved a performance increase of 11.7×, compared to the baseline scenario, with a reasonable impact on area. Additionally, the text size of the program was drastically reduced to 6.2% of the baseline scenario, indicating a potential for reduced memory area requirements. While the system shows impressive gains in performance, architectural decisions relating to the accessibility of NMC memories and the core need to be carefully studied, suggesting possibilities for future work. The work presented in this thesis serves as a foundation for further exploration of hardware acceleration in the field of machine learning, and being able to benchmark accelerators in a real system.

Appendix A

Interrupt handler

Listing A.1 Setup IRQ

```
1: void setup_irq(){
2:     asm volatile (
3:         "li t0, 0x888\n\t"
4:         "csrs mie, t0\n\t"
5:         "csrs mpicmask, 0x1F\n\t"
6:         "la t0, _C_trap_handler \n\t"
7:         "csrw mtvec, t0 \n\t"
8:         // Disable interrupts
9:         "csrci mstatus, 8\n\t"
10:        // Enable interrupts
11:        "csrsi mstatus, 8\n\t"
12:    );
13: }
```

Listing A.2 Trap Handler

```
1: void __attribute__((interrupt("machine")))
       _C_trap_handler() //
2:     {
3:     // Disable interrupts
4:     __asm__ volatile ("csrci mstatus, 0x8");
5:
6:     // get trap source
7:     int code;
8:     __asm__("csrr %0, mcause" : "=r"(code));
9:
10:    // call Interrupt service routine
11:    ISR(((unsigned) code << 1) >> 1));
12:
13:    // Set mstatus.mie bit to enable interrupts
```

```
14:     __asm__ volatile ("csrsi mstatus, 0x8");
15:
16: }
```

Listing A.3 Interrupt Service Routine (ISR)

```
1: #define MCAUSE_MTIP_SHIFTED 0x7
2: #define MCAUSE_MEIP_SHIFTED 0xb
3:
4: void ISR(unsigned int code)
5: {
6:   switch (code)
7:   {
8:     case MCAUSE_MTIP_SHIFTED:
9:       // do something
10:      break;
11:    case MCAUSE_MEIP_SHIFTED:
12:      // Change control signal.
13:      wait_for_interrupt = 0;
14:
15:      // read mpicflag into variable
16:      int mpicflag;
17:      __asm__("csrr %0, mpicflag":"=r"(mpicflag));
18:
19:      // clear LSB to 0
20:      mpicflag &= ~1;
21:
22:      // write modified value back to mpicflag
23:      __asm__("csrw mpicflag, %0"::"r"(mpicflag));
24:      break;
25:    default:
26:      // do something
27:      break;
28:  }
29: }
```

Bibliography

[1] Andrew Waterman et al. "The risc-v instruction set manual, volume i: Base user-level isa". In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[2] Gagandeep Singh et al. "Near-memory computing: Past, present, and future". In: *Microprocessors and Microsystems* 71 (2019), p. 102868. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102868. URL: https://www.sciencedirect.com/science/article/pii/S0141933119300389.

[3] Masoud Nouripayam et al. "An Energy-Efficient Near-Memory Computing Architecture for CNN Inference at Cache Level". In: *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. 2021, pp. 1–4. DOI: 10.1109/ICECS53924.2021.9665530.

[4] Gagandeep Singh et al. "A Review of Near-Memory Computing Architectures: Opportunities and Challenges". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, pp. 608–617. DOI: 10.1109/DSD.2018.00106.

[5] Harold S. Stone. "A Logic-in-Memory Computer". In: *IEEE Transactions on Computers* C-19.1 (1970), pp. 73–78. DOI: 10.1109/TC.1970.5008902.

[6] Sally A. McKee. "Reflections on the Memory Wall". In: *Proceedings of the 1st Conference on Computing Frontiers*. CF '04. Ischia, Italy: Association for Computing Machinery, 2004, p. 162. ISBN: 1581137419. DOI: 10.1145/977091.977115. URL: https://doi.org/10.1145/977091.977115.

[7] C.C. Liu et al. "Bridging the processor-memory performance gap with 3D IC technology". In: *IEEE Design Test of Computers* 22.6 (2005), pp. 556–564. DOI: 10.1109/MDT.2005.134.

[8] May 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[9] ARM Limited. *AMBA® AHB Protocol Specification*. PDF. 2001–2021. URL: https://developer.arm.com/documentation/ihi0033/latest/.

[10] Kea-Tiong Tang et al. "Considerations Of Integrating Computing-In-Memory And Processing-In-Sensor Into Convolutional Neural Network Accelerators For Low-Power Edge Devices". In: *2019 Symposium on VLSI Circuits*. 2019, T166–T167. DOI: 10.23919/VLSIC.2019.8778074.

[11] S. Castillo Mohedano. "Memory Efficient Hardware Accelerator for CNN Inference". MA thesis. Lund University, Department of Electrical and Information Technology, 2023.

[12] Five EmbedDev. *Interrupt Quick Reference*. Blog post. 2023. URL: https://five-embeddev.com/quickref/interrupts.html.

[13] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (Mar. 2021), pp. 708–727. DOI: 10.1109/tpds.2020.3030548. URL: https://doi.org/10.1109%2Ftpds.2020.3030548.

Printed by Tryckeriet i E-huset, Lund 2023

LUND
UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-955
http://www.eit.lth.se