Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Marcus Stensby Young

Improving Memory Scheduling of an Out-of-Order Core

Master's thesis in Embedded Systems
Supervisor: Magnus Själander
Co-supervisor: Amund Bergland Kvalsvik
June 2023

NTNU
Norwegian University of
Science and Technology

Marcus Stensby Young

Improving Memory Scheduling of an Out-of-Order Core

Master's thesis in Embedded Systems
Supervisor: Magnus Själander
Co-supervisor: Amund Bergland Kvalsvik
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

This project aims to optimize the memory system of the RISC-V Berkeley Out-of-Order Machine (BOOM) processor core by improving the handling of L1 cache misses in the Load-Store Unit (LSU). Prior to working on the BOOM core, the handling of L1 cache misses when no Miss Status Holding Registers (MSHRs) were available was inefficient, and is optimized through the implementation of wakeup logic. This logic selectively re-issues relevant load instructions only when an MSHR becomes available, reducing unnecessary activity between the LSU and the data cache. The wakeup logic has decreased the number of not acknowledged signals (nacks) issued by the data cache to the LSU. The implemented changes resulted in a significant 30% reduction in targeted nacks compared to the baseline implementation. Furthermore, benchmarking the design on an FPGA using the SPEC CPU® 2017 benchmark suite demonstrated comparable performance to the baseline implementation, with an optimal load instruction count ranging from 20% to 28% of the total instruction count. While the implementation shows promising results, future work should focus on extending the wakeup logic to address the remaining four identified nack types in the data cache. By doing so, further enhancements in the memory system's efficiency and performance can be achieved.

Sammendrag

Dette prosjektet har som mål å optimalisere minnesystemet til RISC-V Berkeley Out-of-Order Machine (BOOM) prosessorkjernen ved å forbedre håndteringen av instruksjoner som bommer i L1 cachen i Load-Store enheten (LSU). Før arbeidet på BOOM kjernen var håndteringer av bom i L1 cachen når det var ingen ledige Miss Status Holding Registers (MSHRs) lite effektiv, og har blitt optimalisert ved å implementere wakeup-logikk. Denne logikken fyrer av relevante load instruksjoner kun når en MSHR blir ledig, noe som reduserer unødvendig aktivitet mellom LSU og cache. Wakeup-logikken har redusert antall forespørsler som blir markert som "ikke godkjent" (nack) av cachen med omtrent 30%. Videre har ytelsen til det nye designet blitt testet med SPEC CPU® 2017 på en FPGA, noe som har vist at wakeup-implementasjonen har en sammenlignbar ytelse med basis-implementasjonen. Wakeup-implementasjonen har optimal ytelse når antallet load instruksjoner utgjør 20% til 28% av det totale antall instruksjoner. Implementasjonen er lovende, og fremtidig arbeid burde fokusere på å implementere wakeup-logikk for de resterende fire nack-signalene som er tilgjengelige. Et slikt arbeid kan videre øke effektiviteten og ytelsen til minnesystemet.

Preface

This master's thesis is the culmination of a two-year master's degree within Embedded Systems at NTNU. The thesis is written on behalf of the Department of Electronic Systems in collaboration with the Department of Computer Science. The aim of this thesis is to improve the memory scheduling of the Berkeley Out-of-Order Machine through optimizing the memory subsystem.

I want to thank my supervisors Magnus Själander and Amund Bergland Kvalsvik for supporting me and providing keen insight and knowledge within the field of computer architecture. I also want to thank the members of the Computer Architecture Lab (CAL) for providing me with necessary help. Lastly I want to thank all fellow students for contributing to academic discussion.

Marcus Stensby Young
June 2023

Table of Contents

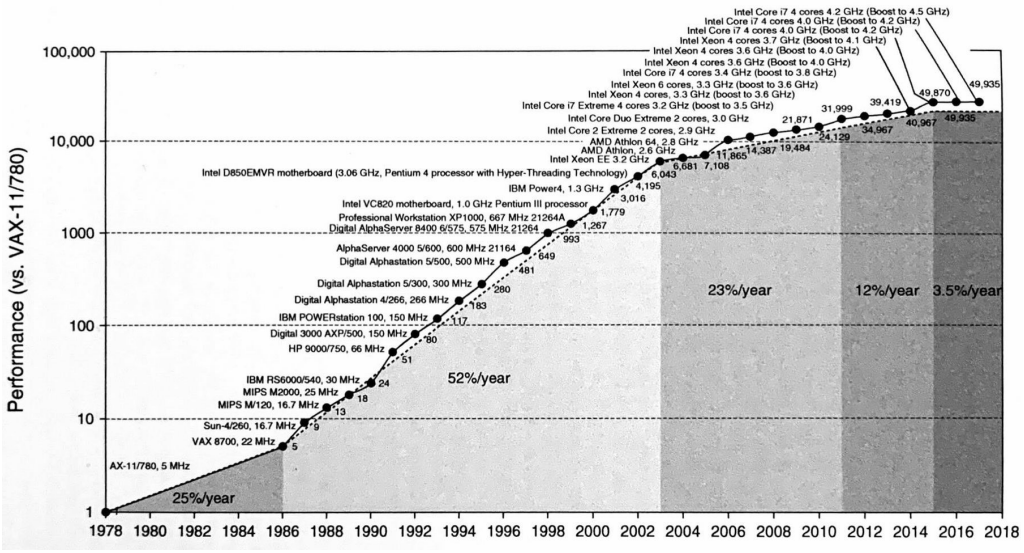
Figures

Tables

Code Listings

1 Introduction

Modern processors are heavily optimized pieces of hardware that perform exceptionally well. However, the performance gain each year is less significant than it used to be. In the 1980s, single-core processor performance doubled at an estimated rate of every one and a half years. At current trends, single-core processor performance doubles at an estimated rate of every 20 years [1]. The three main reasons for this are the end of *Dennard Scaling* [2], the slowing of *Moore's Law* [3], and the emergence of the *Memory Wall* [4]. Figure 1 shows how single-core processor performance gains have slowed down since the mid-2000s.



Source: *Computer Architecture: A Quantitative Approach, page 3 [1]*

Figure 1: Single-core processor performance trend

Over the last 40 years, memory speeds have increased at a slower rate than processor performance. This performance gap introduces a bottleneck, as the memory needs to send enough instructions and data to and from the processor to fully utilize the processor's computational power [5]. One common strategy to reduce this performance gap in modern processors is implementing a hierarchy of differently-sized memory caches that are meant to store the data that is most frequently used at any given time. The cache closest to the processor is usually called the L1 cache and is the smallest but fastest cache. By implementing increasingly sized but slower and cheaper caches, the processor can perceive the memory as large and fast without significantly increasing the unit's price. However, there is still room for improvement in memory design and handling that can be achieved through optimizations, redesigns, or technological improvements. Therefore, optimizing memory handling between or within caches can reduce a system's power usage, hardware usage, memory latency, or any combination of these.

For these reasons, this project aims to optimize the memory system of the *RISC-V Berkeley Out-of-Order Machine (BOOM)* processor core. Specifically, how the *Load-Store Unit (LSU)* in the BOOM core handles L1 cache misses. When a load misses in the L1 cache, the missing instruction is placed in a *Miss Status Holding Register (MSHR)* while retrieving the data from a higher-level cache. However, prior to this project, the BOOM core inefficiently handled L1 cache misses when there were no available MSHRs by opportunistically re-issuing the instruction that missed, even if there were no MSHRs available. This logic has been optimized by implementing *wakeup* logic that only re-issues the relevant instruction when an MSHR is available. This wakeup logic will reduce activity between the LSU and the MSHRs by reducing the amount of *not acknowledged signals (nacks)* that are issued from the data cache to the LSU. Specifically, the implementation targets the nacks that are issued when the LSU tries to place an entry in an MSHR when there are none available.

To verify if the implemented changes decreased the amount of targeted nacks, the design was flashed onto an FPGA and benchmarked with the SPEC2017 benchmark suite. Results from the benchmark show that the implementation has reduced the amount of targeted nacks issued from the data cache unit to the LSU by 30%. This reduction in nacks might result in a more power-efficient unit. However, the wakeup implementation is not as aggressive when re-issuing loads to the MSHRs, which might explain why the core has seen a minuscule reduction in average *instructions per cycle (IPC)*.

This thesis is structured as follows: A review of required underlying theory and an overview of the BOOM core, a brief description of the tools used, a section on the implemented design, a description of how the design was tested, a section on simulation results, a discussion of the achieved results, a section on possible future work, and a conclusion.

2 Background

With the relative stagnation of processor performance improvement, it has become increasingly important to implement smart solutions within computer architecture. A field within computer architecture that has great potential for improvement are memory systems, ranging from entire design philosophies to optimizing minor parts of existing architectures. Given the scope and time frame of this project, it was more feasible to optimize or modify an existing architecture rather than design something from the ground up.

The BOOM core is an open-source processor project built on RISC-V. The core is written in a relatively new open-source HDL that aims to make up for what other HDLs lack. Even so, the core still has room for improvement. With these considerations and the fact that the core is integrated into the Chipyard development framework, the BOOM core is a great candidate for academic research.

2.1 Theoretical Framework

This section covers relevant concepts and theories to better understand the BOOM core's history and the more detailed description of the BOOM core in Section 2.3.

2.1.1 Dennard Scaling

Dennard Scaling states that one could reduce transistor sizes while keeping their power density constant, and by doing this, one could increase transistor count and clock frequency without increasing overall power consumption. This relationship was discovered by and named after Robert H. Dennard in 1974. This theory has held for many years and has provided "free" performance gains. However, in the mid-2000s, transistors became so small that reducing sizes no longer kept the power density constant as leakage current at low threshold voltages became an issue. Consequently, it would no longer be possible to increase clock frequencies year-to-year as drastically. This is known as the end of Dennard Scaling. However, scaling is still happening in modern processors, but the focus has shifted from trying to increase single-core clock frequencies to multi-core processors and parallelism [6].

2.1.2 Moore's Law

Moore's Law is the name of the observation made by Gordon E. Moore in 1965 regarding decreasing transistor sizes. It states that one could double the number of transistors in a given area every two years. The combination of Dennard Scaling and Moore's Law was why processor performance was estimated to double every 1.5 years. Improvements in production transistor and die techniques have upheld this reduction trend for many years, but as transistors have reached sizes on the scale of only a few atoms, this trend is bound to slow down.

2.1.3 The Memory Wall

The Memory Wall, a phenomenon in computer architecture, refers to the growing gap between improving processor speeds and the relatively slower improvement in memory speeds. This gap has critical implications for overall system performance, meaning that the memory will eventually limit the processor's performance.

As processors have undergone significant advancements in performance, primarily driven by Dennard Scaling, Moore's Law, and architectural optimizations, memory speeds have not kept pace with the same rate of improvement. While memory capacities have increased, the speed at which data can be accessed and transferred to and from memory has yet to experience comparable growth. This

situation creates a performance bottleneck where processors often find themselves idle while waiting for data to be fetched from or stored into memory. This idle time can significantly impact the overall efficiency and throughput of the system, limiting the processor's ability to fully utilize its computational power. The memory wall becomes particularly prominent when memory-intensive tasks or applications are executed, such as large-scale data processing, scientific simulations, or complex graphics rendering. These tasks often involve accessing and manipulating substantial amounts of data, leading to frequent memory accesses and exacerbating the performance gap between the processor and memory.

Several techniques and strategies are employed to mitigate the impact of the Memory Wall. One approach is to implement a memory hierarchy consisting of caches. Caches are implemented such that the fastest cache is closest to the processor, while slower caches are placed further up the hierarchy. The caches store frequently used data by exploiting the emergent locality of how each program accesses data. Implementations such as this mitigate the performance gap between processors and memory but will not completely remove the gap.

2.1.4 ISA

An Instruction Set Architecture, *ISA*, is a set of rules and constraints that defines how to interact with the processor hardware. These rules and constraints define what instructions the processor can execute, the instruction layout, the input and output design, memory management, supported data types, and included registers. This information is necessary for programmers to create software that can run on the given processor [7].

2.1.5 RISC

A *Reduced Instruction Set Computer (RISC)* is a family of processor architectures that use a smaller number of highly-optimized and efficient instructions making RISC architectures less complex in terms of hardware design. Compared to a *Complex Instruction Set Computer (CISC)*, RISC instructions do less work but can be completed faster and more efficiently than CISC instructions. The smaller and more rigid RISC instructions make RISC architectures more power efficient but less flexible to the programmer, putting more pressure on programmers and compiler developers to ensure that programs use efficient instructions [8].

2.1.6 Instruction Pipelining

A program that runs on a processor contains several instructions that need to be executed. These instructions have different execution times and need different resources to complete. Furthermore, each instruction needs to be executed in a certain way to function correctly and is commonly known as the different stages of execution. In a non-pipelined processor, all stages of execution are designed as one large stage, meaning that each instruction has to propagate through all stages before a new instruction can begin execution. Consequently, the processor's clock frequency has to be slow enough so that the slowest instruction can propagate through all stages in a single cycle. Consequently, fast instructions will finish long before a clock cycle finishes, resulting in poor performance.

On the other hand, a pipelined processor divides the single large stage into several smaller stages. By doing this, the processor can begin executing a new instruction when the previous instruction completes the first stage. This is the case for all the stages in the pipeline; when a particular stage completes its instruction, it sends the instruction to the next stage and can then take in a new instruction. This allows for a higher clock frequency. The number of stages and their purpose varies and is up to the architects to decide, but the classic RISC pipeline is a five-stage pipeline comprised of the **Instruction fetch**, **Instruction decode**, **Execute**, **Memory access**, and **Writeback** stages. However, pipelining introduces a *hazards* issue. Hazards are classified as either data hazards, control hazards, or structural hazards. Data hazards occur if there is an

overlap in instruction execution in the pipeline that will give a different result compared to if that execution would run sequentially. Data hazards are classified as either Read After Write (RAW), Write After Read (WAR), or Write After Write (WAW). Code Listing 1 shows examples of data hazards. Control hazards might occur when executing a branch instruction. When executing a branch instruction the processor has to choose if the branch should or should not be taken before the result of the branch is resolved. This means that the when the branch is resolved the processor might have progressed down the wrong "execution path", which needs to be rolled back. Structural hazards occur when two or more instructions in the pipeline need access to the same resource. If hazards are not handled correctly the processor will not be able to execute programs correctly.

```
#RAW                    #WAR                    #WAW
iA: add r1, r2, r3      iA: sub r4, r1, r3      iA: sub r1, r4, r3
iB: sub r4, r1, r3      iB: add r1, r2, r3      iB: add r1, r2, r3
```

Code Listing 1: Data hazard examples
(hazards from interactions with r1 in all examples)

2.1.7 Superscalar Processing

Each stage in a pipelined processor performs some task, e.g., fetching an instruction from the instruction queue in the instruction fetch stage or executing an arithmetic operation in the execution stage. In a scalar processor, only one instruction can be executed in each pipeline stage simultaneously, but in a superscalar processor, multiple instructions can be executed in the same pipeline stage. This increases instruction-level parallelism by exploiting that not all instructions are of the same type. A superscalar processor increases the width of pipeline stages that are notoriously slower than other stages. Most noticeable are the issue and execution stages. The execution stage is widened by implementing multiple execution units, each with one or more functional units. An example of a functional unit is an arithmetic logic unit (ALU) that can perform arithmetic operations, e.g. addition, and logic operations, e.g. an OR operation. It is necessary to implement a dispatcher to identify what type of instructions are ready to be executed and to send these instructions to the correct execution unit. Superscalar processing allows for concurrent execution of instructions in the pipeline, but needs additional logic to manage data dependencies and other constraints so that instructions are executed according to the execution order. However, letting instructions use available resources when possible, rather than being blocked from using them given structural limitations, will better utilize the hardware and increase overall throughput and performance.

The effectiveness of superscalar processing depends on factors such as instruction mix, the presence of data dependencies, and the availability of resources. It is particularly beneficial for workloads with a high degree of instruction-level parallelism, where multiple independent instructions can be executed concurrently.

2.1.8 Out-of-order Execution

Traditionally, instructions in a program are executed in sequential order by the processor. Even if the processor is a pipelined superscalar processor, instructions of the same type are executed sequentially. Out-of-order execution allows instruction execution order to be dynamically rearranged when an instruction's resources are available, and its dependencies are met. This means that an independent instruction that would otherwise have to wait for other instructions that are waiting for their dependencies to be met, can be executed if the required resources are available.

Out-of-order execution combined with a superscalar pipeline is a common practice in modern high-performance processor as it can minimize idle time of resources by feeding them with more instructions, which will lead to better performance and increased throughput.

2.1.9 Load-store Architecture

There are four main types of ISAs: stack, accumulator, register-memory, and register-register. In a register-register architecture, also known as a load-store architecture, instructions are divided into memory instructions and arithemetic instructions. These architectures are designed so that data is loaded into registers before the processor performs an arithmetic operation on the loaded registers. This allows arithmetic operations to execute without accessing the memory, which improves the speed of these operations. Furthermore, using registers introduce fewer unnecessary dependencies than stack and accumulator architectures. On the other hand, load-store architectures have a higher instruction count and lower instruction density than other architectures which could negatively impact program size and instruction cache interactions. RISC-V is an example of a load-store architecture.

2.1.10 Power Consumption in Digital Circuits

Power consumption in digital circuits is the total amount of static power consumption plus the amount of total dynamic power consumption [9]. Static power consumption is the power drawn by the inactive parts of the circuit, and is caused by leakage currents in the electrical components. Dynamic power consumption is the power drawn by the active parts of the circuit, the switching power, and the short-circuit power consumption. The switching power consumption can be calculated using Equation 1, where α is the acitivity factor, f is the frequency, C_L is the capacitive load, and V_{DD} is the supply voltage. The total power consumption can be calculated with Equation 2, where t_{sc} is the transition time, I_{peak} is the peak current, and $I_{leakage}$ is the leakage current. Reducing the power consumption is a critical part of designing energy efficient processors and other electrical hardware.

$$P_{switching} = \alpha f C_L V_{DD}^2 \tag{1}$$

$$P_{total} = P_{switching} + t_{sc} V_{DD} I_{peak} + V_{DD} I_{leakage} \tag{2}$$

2.2 RISC-V

RISC-V is an open standard RISC ISA created by the University of California, Berkeley but is now maintained by RISC-V International [10][11][12]. RISC-V was created as an open standard ISA to mainly support research, development, and education, but has in recent years been adopted into commercial and proprietary architectures. With this rise in popularity, RISC-V is set to become a future-proof choice for processor design within the industry as well. RISC-V is a load-store architecture that was designed to support modern standards without favoring a specific microarchitecture style or implementation technology, making it a viable choice for a broader set of designs. Furthermore, RISC-V implements a modular-like extension design that consists of a standard base integer ISA that can be modified with extensions. Examples of such extensions are the **M**-extension that adds support for integer multiplication and division instructions, and the **F**-extension that adds support for single-precision floating point instructions.

2.3 The BOOM Core

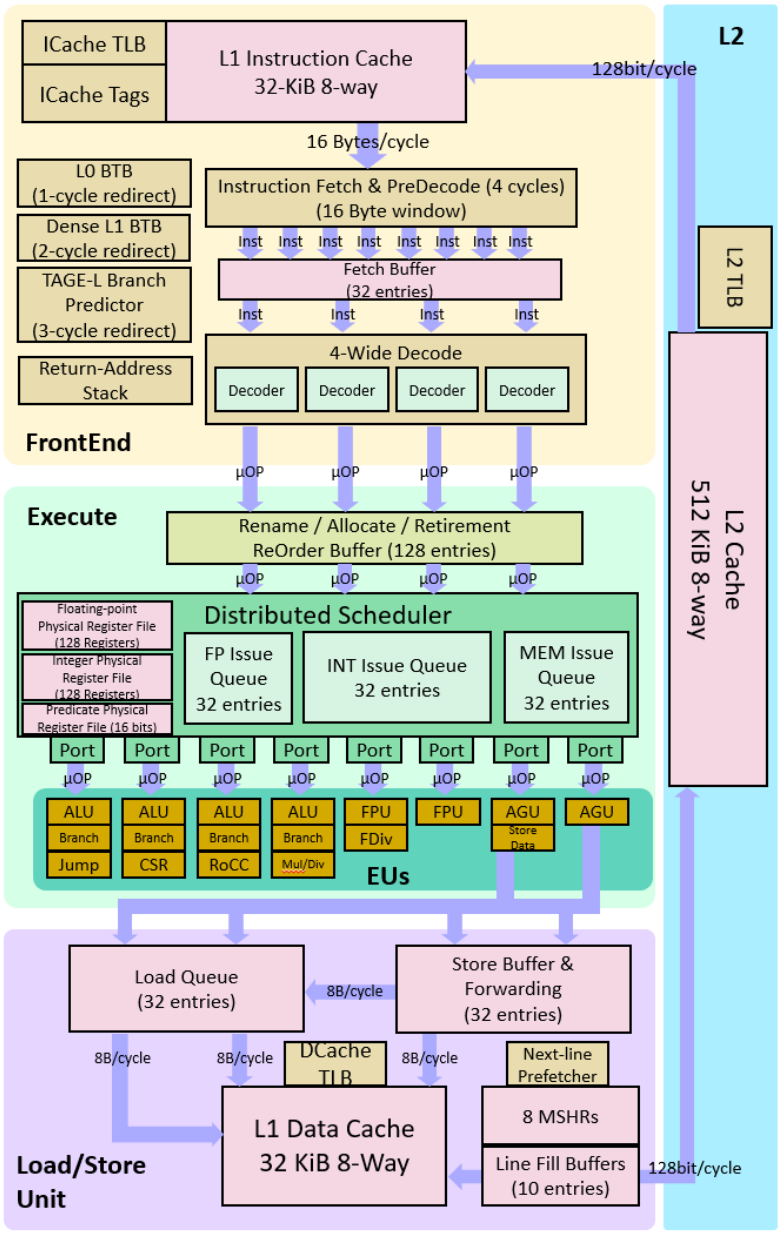
The BOOM core is an open-source superscalar out-of-order RISC-V implementation that is synthesizable and parameterizable [13][14]. The BOOM core is constantly undergoing improvements, and the current version is called *SonicBOOM* or *BOOMv3*. The core implements the RV64GC RISC-V ISA, combining a base ISA and several extensions. The **G** abbreviation is defined as a "general-purpose" ISA and includes a combination of a base ISA and several extensions. Adding the **C**-extension improves performance, code size, and energy efficiency. Other extensions are more

domain-specific and can be added if needed. Table 1 below gives an overview of the included ISA extensions in RV64GC. The implementation of the BOOM core prior to any changes made in the context of this project will be referred to as the *baseline* implementation.

Table 1: Overview of the RV64GC ISA

Abbreviation	**Extension**	**Description**
G	I	The base integer ISA
	M	Adds integer multiplication and division instructions
	A	Adds atomic instructions
	F	Adds single-precision floating-point instructions
	D	Adds double-precision floating-point instructions
	Zicsr	Adds control and status register (CSR) instructions
	Zifencei	Adds the FENCE.I synchronization instruction
C	C	Adds compressed instructions for common operations

The BOOM core pipeline comprises ten conceptual pipeline stages combined into seven distinct stages: **Fetch**, **Decode/Rename**, **Rename/Dispatch**, **Issue/RegisterRead**, **Execute**, **Memory**, and **Writeback**. The tenth conceptual stage is the **Commit** stage which executes asynchronously to the rest of the pipeline and is therefore not counted in the number of actual stages. The BOOM core is designed to generate micro-operations, *micro-OP*, or *UOP*, in the decode stage from the instructions fetched in the instruction fetch stage. This increases the amount of parallel work that can be done for each instruction, hence increasing performance. For example, a store instruction has data it wants to store and an address to store it at. This can be divided into two UOPs; uopSTD uopSTA. These UOPs require different resources and can be issued independently, especially when addresses typically take longer to calculate than it takes to acquire the data. The store is then issued as a single UOP when both the data and the address is ready.



Source: *BOOM GitHub page [15]*

Figure 2: Detailed overview of the SonicBOOM core

Figure 2 above shows a detailed overview of the BOOM core. The block labeled **FrontEnd** comprises the fetch and decode stages as well as the branch prediction unit, the block labeled **Execute** comprises the rename, dispatch, and execute stages, and the block labeled **Load/Store Unit** comprises the memory and writeback stages. The figure shows the exact configuration for the SonicBOOM paper, but most of the units shown in this figure are configurable.

2.3.1 Memory Model

A memory model or memory consistency model, defines a set of rules that specify how memory instructions are ordered in program execution. This is important to ensure that all memory operations execute in an order that respects all dependencies and constraints so that programs are executed correctly. The BOOM core implements a *RISC-V Weak Memory Ordering (RVWMO)* memory model (chapter 14, appendix A, and appendix B in the RISC-V Instruction Set Manual [10]).

RVWMO is defined in the terms of the global memory order, meaning a total ordering of all memory operations generated by load and store instructions produced by each *hart (hardware thread)*. However, memory operations executed on each hart appear to be in the correct order to other operations on that hart, but memory operations on other harts might appear to be executed in a different order. This means that multithreaded code needs additional synchronization to guarantee correct memory ordering of memory opeartions across all harts. The RV64GC RISC-V ISA includes instructions for synchronizing harts.

Even though RVWMO follows global memory ordering it does not respect each harts entire program order. The subset of memory operations in each harts program order that needs to be respected by the global program order is called the *preserved program order*. Three requirements need to be met for a memory operation to precede another memory operation in a hart's preserved program order: Memory operation **X** precedes memory operation **Y** in program order, both memory operations access the main memory, and there is either an overlapping-address ordering, explicit synchronization, syntactic dependencies, or pipeline dependencies.

2.3.2 Load-Store Unit

The main task of the BOOM core's *Load-Store Unit (LSU)* is to organize memory operations. This includes tracking required instruction information, scheduling instructions, keeping track of resource usage, forwarding data, killing data accesses, and returning data to the core. To track the information required to handle all these tasks the LSU employs a *Load Queue (LDQ)* and a *Store Queue (STQ)*. These queues are tables that can hold several entries and track each entry's status bits and necessary data. Entries in these queues are reserved in the decode stage of the pipeline, while addresses and store data are calculated and placed in corresponding entries in the execute stage. Table 2 below shows how each entry is stored in the STQ. The **valid** field indicates if the corresponding store entry is a valid entry. The **addr** field holds the address of the corresponding store, while the **virtual** field indicates if the that entry's address is virtual or not. The **data** field contains the store data, and the **cmt** and **succ** fields indicate if the store entry has committed or succeeded, respectively. Table 3 shows how each entry is stored in the LDQ. The **valid** field indicates if the corresponding load entry is a valid entry. The **addr** field hold the address of the corresponding load, while the **virtual** filed indicates if that entry's address is virtual or not. The **exec** field indicates if the load has been executed, while the **succ** field indicates if the load has succeeded. The **ord** filed indicates if a load order failure has occurred. The **obs** field indicates if the load has been observed.

Table 2: Store Queue (STQ)

STQ					
valid	addr	virtual	data	cmt	succ
⋮	⋮	⋮	⋮	⋮	⋮

Table 3: Load Queue (LDQ)

LDQ									
valid	addr	virtual	exec	succ	ord	obs	st_mask	st_fw	st_idx
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Loads are optimistically fired to memory once the address has been calculated and placed in the

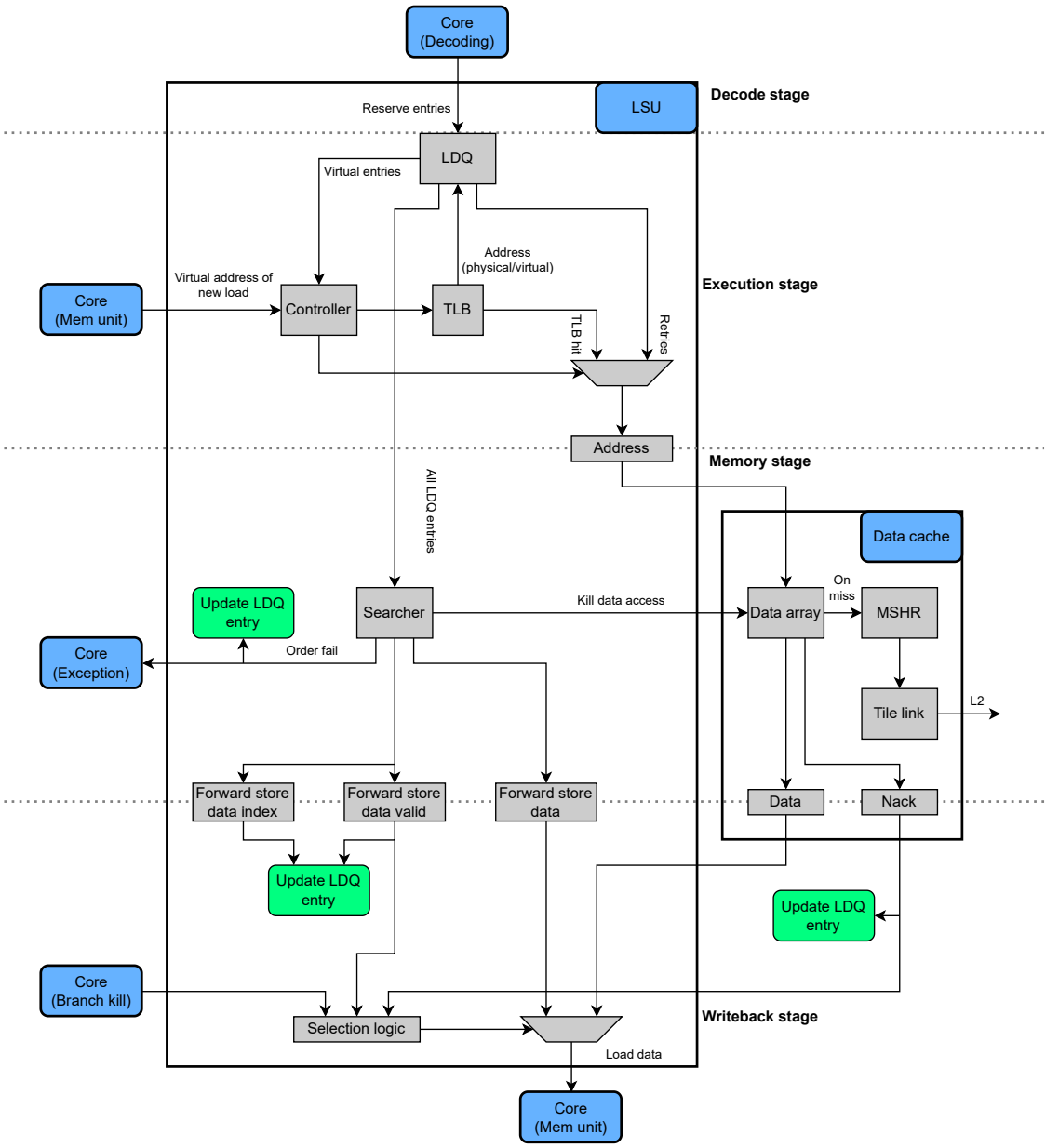
LDQ. At the same time, the load uses the store dependency mask to check if those stores have their store data available in the STQ. If the data is present, it is forwarded to the load instruction, and the memory request is killed. If the data is not present in the STQ, the memory request is killed, and the load is put to sleep. Loads that are put to sleep are woken up and retried later.

Firing time-consuming loads as early as possible increases performance significantly, and is a strong advantage of out-of-order architectures. This is also why the BOOM core fires loads as early as possible. Firing loads early increases performance but also allows for more memory ordering failures to occur. BOOM follows the RVWMO memory consistency model, which requires that loads to the same address to be in order. To address this potential issue, loads search through the LDQ to check if any older loads have a matching address. If an address match occurs and a younger load has executed before an older load, as well as a cache coherence probe event has occurred, the younger load needs to be executed again and the instructions after it needs to be flushed from the pipeline. Otherwise, loads can safely be re-ordered.

In addition to stores and loads, the LSU in the BOOM core is also tasked with handling *atomic* and *fence* operations. Atomic memory operations are operations that both read and write to memory. The BOOM core supports both complex atomic memory operations on single or double words, as well as atomic memory operation primitives. Complex atomic memory operations are supported with the implementation of the **Load-Reserved (LR)** and the **Store-Conditional (SC)** instructions. The **LR** and **SC** instructions create control necessary to check if the memory location that the complex instruction segment wants to access is altered, the operation marks itself as having failed and can try again. This creates allows for creating lock-free complex atomic memory operations. The atomic memory operations primitives are instructions that load data from an address into a register, performs a binary operation between that register and another register, then stores the result in the original address. Both the complex atomic memory instructions and the atomic memory instruction primitives include two bits for altering the ordering constraints for that specific instruction.

Fence operations synchronize the instruction and data streams within a hart. Specifically, a **FENCE.I** instruction synchronizes the instruction and data streams within a hart, guaranteeing that previous data stores are visible to subsequent instruction fetches. However, this does not guarantee that other harts' subsequent instructions fetches can observe the previous data stores of the local hart. To guarantee that all previous data stores are visible to subsequent instruction fetches in all harts, a hart needs to execute a **FENCE** operation followed by issuing a request to all other harts they should execute a **FENCE.I** instruction.

Figure 3 below shows an overview of the LSU excluding hardware that relates to store instructions. The figure shows that only the LDQ entries are fed into the searcher, but the STQ entries are also fed into the searcher.



Source: *LSU documentation page, modified image [16]*

Figure 3: Overview of load-related parts of the LSU

2.3.3 Nacking Loads

The data array in the data cache block in Figure 3 is equivalent to the L1 memory cache. The data cache in the BOOM core has five different internal signals that indicate when a load instruction can not get the data from the data array. These signals are called not acknowledged signals or nacks. These five nack signals are a nack when there is an incoming probe, a nack when something that is being evicted from the cache is hit, a nack when the MSHRs are not ready for a request, a nack from a data bank conflict, and a nack when the data cache is unable to allocate an MSHR for the set that is being written back. In the baseline implementation, the data cache sends these five internal signals back to the LSU as a single nack signal, which then retries the nacking load later. Having a single nack signal eliminates the possibility for the LSU to differentiate between the reasons for a load being nacked.

3 Tools

This chapter will provide a brief description of the tools used to modify and verify changes in the BOOM core, as well as the tools used for running simulations.

3.1 Chisel/FIRRTL

Chisel is a hardware description language (HDL) that is embedded in the Scala high level programming language and was created by developers at the University of California, Berkeley. The motivation behind creating Chisel was that the developers wanted a language that could generate hardware from high-level design parameters and constraints, while also incorporating functionality found in modern high-level languages such as object-oriented programming, type inference, and support for functional programming. The developers opted out of using Verilog and VHDL given that these HDLs were created to serve as hardware simulation languages. Consequently, hardware generated by Verilog and VHDL might not be possible to synthesize, or may result in inefficient hardware.

When using writing Chisel code you are actually writing Scala code using the Chisel library. The library contains different classes, objects, and conventions that allow the Scala program to create hardware. By embedding Chisel within Scala you abstract away some of the complexity of creating hardware with other HDLs, e.g. VHDL, which makes Chisel easier to work with if you are new to HDL.

Flexible Internal Representation for RTL (FIRRTL) is elaborated from Chisel code, and is consumed by FIRRTL compilers to perform a set of transformations on the RTL. The final representation of FIRRTL resembles a netlist and is a result of several transformations on the RTL. This representation is called LoFIRRTL and can easily be used for translating to a different output language such as Verilog [17][18]. An example of a LoFIRRTL representation of a circuit is shown in Code Listing 2 below. In Chipyard two FIRRTL compilers are used: The Scala FIRRTL Compiler (SFC) and the MLIR FIRRTL Compiler (MFC). SFC and MFC both compile the Chisel code into LoFIRRTL, however, MFC is written in C++ which lowers compilation time but also produces a different LoFIRRTL representation [19]. The MLIR (Multi-Level Intermediate Representation) compiler falls under the LLVM compiler infrastructure, which is a project that aims to gather different compiler technologies [20][21].

```
module MyModule :
    input in: {a:UInt<1>, b:UInt<2>[3]}
    input clk: Clock
    output out: UInt
    wire c: UInt
    c <= in.a
    reg r: UInt[3], clk
    r <= in.b
    when c :
        r[1] <= in.a
    out <= r[0]
```

```
module MyModule :
    input in$a: UInt<1>
    input in$b$0: UInt<2>
    input in$b$1: UInt<2>
    input in$b$2: UInt<2>
    input clk: Clock
    output out: UInt<2>
    wire c: UInt<1>
    c <= in$a
    reg r$0: UInt<2>, clk
    reg r$1: UInt<2>, clk
    reg r$2: UInt<2>, clk
    r$0 <= in$b$0
    r$1 <= mux(c, in$a, in$b$1)
    r$2 <= in$b$2
    out <= r$0
```

Source: *Chapter 11 in the FIRRTL specification [18]*

Code Listing 2: Chisel code, left, and equivalent LoFIRRTL representation

3.2 Chipyard

The Chipyard framework was created by the Berkeley Architecture Research Group from the University of California, Berkeley to produce RISC-V SoCs (System-on-Chip) [22]. Chipyard enables developers to create SoCs by integrating open-source and commercial tools in a seamless manner. Currently, Chipyard includes the Rocket core, the BOOM core, and the CVA6 core, as well as accelerators, memory systems, and other peripherals [23]. The Chipyard documentation page includes a chapter on how to configure the framework for your desired use. By following the simple set up guide one can quickly and easily start modifying the Chisel source code for each core.

Simulating smaller benchmarks and simple programs on a core design is a trivial task in Chipyard as it supports both the *Verilator* and the *VCS* simulators, which are open-source and commercial simulators, respectively. Furthermore, the Chipyard framework allows for generation of VCD files and output logs by including a debug flag when running simulations.

Chipyard also supports more advanced functionality. This includes functionality such as using *Hammer*, a modular and reusable physical design flow tool [24], to design the physical layout of a chip, adding custom cores or other hardware, configuring the memory in different ways, and FPGA-accelerated simulation. By integrating all these elements in a single framework, Chipyard aims to gather all the tools a developer needs to create a SoC in a single framework.

3.3 Verilator

Verilator is an open-source Verilog and SystemVerilog simulator [25]. Verilator reads the provided Verilog or SystemVerilog input code and outputs C++ code and header files. These files are then compiled into an executable. Running this executable will perform the wanted simulation. Verilator is integrated into the Chipyard framework, providing easy and fast software simulation of hardware written in Chisel.

3.4 FireSim and FireMarshal

FireSim is an open-source full system FPGA-accelerated hardware simulation platform [26]. Initially, FireSim was made to run on Amazon EC2 F1, a warehouse scale public cloud FPGA cluster hosted by Amazon, but currently also supports running on on-premise FPGAs. *FireMarshal* is a workload generation tool for RISC-V designs, created for the FireSim platform [27]. It can be used to configure details regarding workloads as well as generating workload filesystem images required to run FireSim.

3.5 GTKWave

Debugging through waveform analysis is a powerful tool when designing hardware. It enables the designer to track all signals and their values at any given time, as well as verifying if the timing is correct and if the design is working as intended. *GTKWave* is a fully featured GTK based free waveform viewer for Unix, Win32, and MacOSX. GTK is a free open-source widget toolkit that allows for creating GUIs (Graphical User Interface). GTKWave can read several different file types including VCD (value change dump) files, which is a standard Verilog file type.

Using GTKWave is moderately straightforward, however, there are some limitations running it on a Windows system. VCD files generated by running simulations can be quite large, reaching magnitudes of GBs per file. If a file is too large, GTKWave crashes when loading it. This forces the designer to create smaller test code so that the generated files do not grow too large. The exact cut-off point is not known. It is not known if this issue persists on other operating systems.

3.6 Virtual Machine

Chipyard was developed and tested on Linux-based systems, and it is recommended to opt out of using Windows when using Chipyard. Furthermore, more hardware is recommended given the high computational cost of running lengthy simulations, and large storage requirements. For these reasons, the Department of Computer Science at NTNU has provided select users on their server with Linux-based virtual machines (VM) that include more storage, RAM, and CPU power to leverage the higher hardware requirements of working with Chipyard.

3.7 The Idun Cluster

The Idun Cluster is a computational platform maintained by the High Performance Computing (HPC) group at NTNU. Idun is a joint shareholder project encompassing several faculties and departments at NTNU. The IT Division is at the core of the project as they contribute the infrastructure for the cluster, while the faculties and departments provide computational resources. Any faculty or department at NTNU can join the project by contributing resources to the cluster. Currently the cluster consists of 1936 cores and 92 GPUs in total, where the Department of Computer Science's share is 864 cores and 80 GPUs. Furthermore, the Deparment of Computer Science has four FPGAs available on the Idun cluster. In this project, the cluster was used to build cores and bitstreams, and to run FPGA-accelerated simulations [28].

4 Modifying the Core

The BOOM source code consists of thousands of lines of code, with some inline comments explaining the code. Therefore, fully understanding the functionality of the core through reading the code, comments, and supporting documentation is an extensive task. This project aims to implement wakeup logic for loads that the data cache has nacked. Wakeup logic entails that instead of the LSU re-issuing loads that have been previously nacked opportunistically, the LSU will wait for a signal that indicates that the load will not be nacked for the same reason again. From preliminary testing, it was observed that nacks that occur due to the MSHRs not being ready to handle a request were the most prevalent. Given the project's scope, wakeup logic has only been implemented for this type of nack which will be referred to as an MSHR nack.

4.1 Load Queue

To support the wakeup logic, each entry in the LDQ has to know which nack occured for each specific load. This is implemented by adding a field to the LDQ that is set when a load is nacked by an MSHR nack. Table 4 shows the modified LDQ used in the wakeup implementation. where the modification is the added field furthest to the right. The scheduler in the LSU will arbitrate between which memory request is fired off to memory. In the wakeup implementation, the scheduler prioritizes loads nacked by an MSHR nack over loads nacked from other sources.

Table 4: Modified LDQ

LDQ										
valid	addr	virtual	exec	succ	ord	obs	st_mask	st_fw	st_idx	mshr_nack
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

4.2 Implementing Wakeup Logic

The BOOM source code has a LSU folder located in */chipyard/generators/boom/src/main/scala*, that contains individual scala files for the data cache, the LSU, the MSHRs, the prefetcher, and the translation lookaside buffer (TLB). To implement the wakeup logic, changes have been made to these three files: *mshrs.scala*, *dcache.scala*, and *lsu.scala*.

4.2.1 mshrs.scala

This scala file contains code for the functionality of each individual MSHR, as well as the MSHRs as a unit. Each MSHR has a signal that indicates if that MSHR is ready to receive a request or not. This signal is taken from each MSHR through an OR operation, and routed back to the LSU as a single wire. Figure 4 presents a visual representation of how the signals are routed to the LSU. This implementation allows the LSU to observe if there is at least one MSHR available.

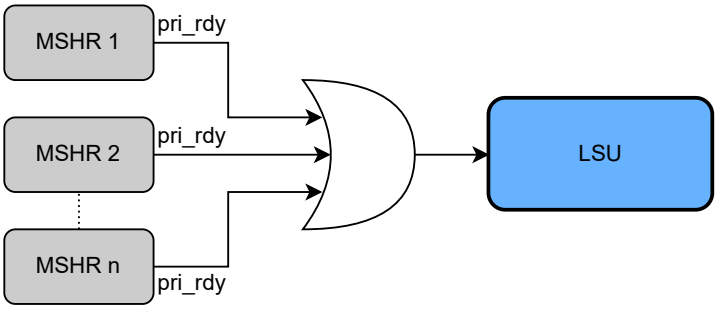


Figure 4: MSHR ready for request signals

4.2.2 dcache.scala

This scala file contains code for all cache related logic that is not already covered in mshrs.scala. Minimal changes have been made to this file to implement wakeup logic. The dcache receives signals from the MSHRs that allow it to differentiate between the nack types, which are only used internally in the dcache. Howver, these signals can be routed out to the LSU, which allows the LSU to track why a load was nacked.

4.2.3 lsu.scala

With the implemented changes made in Section 4.2.1 and Section 4.2.2, the LSU can be modified to implement wakeup logic. The BOOM source code is written so that there is a *can_fire*, a *will_fire*, and a *fired* signal for each type of memory request that can be fulfilled. The scheduler will set the will_fire signal high based on the availability of resources and the value of the corresponding can_fire signal. The fired signal goes high the cycle after the corresponding will_fire signal goes high if request is not killed by a resolved branch.

The wakeup implementation needs to add support for a can_fire-will_fire-fired signal system for the new memory requests that come from a load being nacked by an MSHR nack. The baseline implementation has a can_fire-will_fire-fired signal system for doing this with the different nacks bundled together as a single nack. To ensure that not both of these memory requests fire when a load is nacked by an MSHR nack, the MSHR nack needs to be filtered out of the original nack signal. This results in two can_fire and will_fire signals that handle the MSHR nack and other nacks. Both can_fire-will_fire-fired signal systems trigger the same fired signal when the corresponding load is executed. The new can_fire and will_fire signals pertaining to the MSHR nacks need to be correctly included in logic throughout the code to ensure the total correctness of the LSU.

The bit in the mshr_nack field in the LDQ needs to be set when an MSHR nack occurs. The bit is set when the nack signal's valid bit is set, the issued UOP's control bit that indicates that it is using the LDQ is set, and the MSHR nack signal routed from the dcache is set. This code segment includes an assertion that ensures that a load has not already been executed by checking if the exec bit is set for that entry in the LDQ. A visual representation is presented in Figure 5.

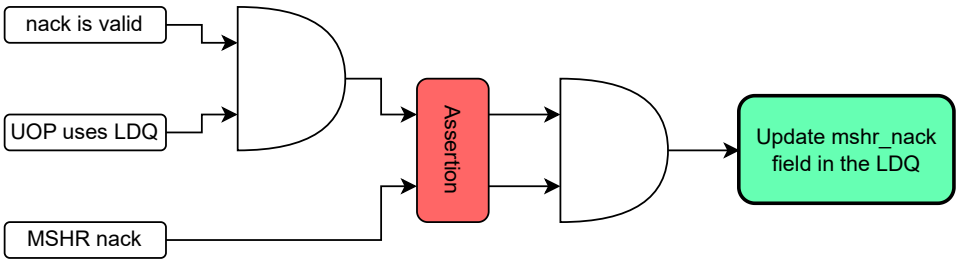


Figure 5: Updating the mshr_nacked field in the LDQ

4.2.4 Debugging and Gathering Results

In addition to the changes made to the design of the BOOM core, hardware counters and other signals have been added to the code for debugging and result gathering purposes.

To leverage initial debugging, signals were established to monitor the activation of specific signals during simple tests. Additionally, hardware counters were integrated to track how many of each nack signal issued from the data cache to the LSU during software simulations. Chisel facilitates the use of C-style print statements, enabling the printing of architectural information during program execution and facilitating debugging for software simulations. However, when running FPGA-accelerated simulations, it is not feasible to print architectural information during execution. To address this challenge, FireSim introduces *AutoCounters*. These performance counters are seamlessly integrated into the simulated design without impacting its behavior. AutoCounters offer configurable sample rates, and users can define start and end trigger instructions using *TracerV* triggers, which are supported by FireSim. By employing AutoCounters with TracerV triggers, users can obtain counter results specifically when the benchmark is running, excluding data from other programs or instructions executed during general system operation. The implementation of AutoCounters utilizes *PerfCounter* objects from the *midas* package. Code Listing 3 presents a code segment demonstrating how the AutoCounters are implemented, with the first argument representing the value by which the counter is incremented each cycle. Setting the first argument to the correct value at the correct time is important to get correct results from the AutoCounter.

```
midas.targetutils.PerfCounter(ld_flag, "ld_flag", "Load instruction")
```

Code Listing 3: AutoCounter in Chisel

5 Simulating the Core

Two cornerstones of scientific research is that the results are empirical and reproducible. Therefore, this chapter will describe how such results were produced in this study. The outline of the chapter is: An overview of the BOOM configurations, running software simulations, and running FPGA-accelerated simulations.

5.1 BOOM Configurations

This section gives an overview of key parameters for the different configurations of the BOOM core. Key parameters for the default small BOOM configuration, which was used for software simulations, are presented in Table 5. Key parameters for the default large BOOM configuration, which was used for both software simulations and FPGA-accelerated simulations, are presented in Table 6. Parameters highlighted in bold were changed individually to test its impact on the baseline and wakeup implementations when running software simulations. FPGA-accelerated simulations were only run with the default large BOOM configuration.

Table 5: Default small BOOM

Parameter	Value
Fetch Width	4
Decode Width	1
Memory Issue Width	1
Memory Dispatch Width	1
Miss Status Holding Registers	2
LDQ Entries	8
Memory Issue Buffer Entries	8
ROB Entries	32
Fetch Buffer Entries	8

Table 6: Default large BOOM

Parameter	Value
Fetch Width	8
Decode Width	3
Memory Issue Width	1
Memory Dispatch Width	3
Miss Status Holding Registers	**4**
LDQ Entries	24
Memory Issue Buffer Entries	16
ROB Entries	96
Fetch Buffer Entries	24

5.2 Software Simulation

Chipyard includes support for running software simulations, including predefined benchmark suites and custom test programs. As described in chapter 7.3 on the Chipyard documentation page, there is support for creating custom baremetal RISC-V programs that can be run in simulation. This is done through creating custom C/C++ programs that are compiled down to RISC-V machine code. These tests can be run by using the commands *make run-binary* or *make run-binary-debug* and providing the location of the test file as the BINARY argument, e.g. *make run-binary BINARY=~/chipyard/tests/CustomTest.riscv*. Furthermore, you can specify which core you want to simulate by passing a CONFIG argument.

Creating custom baremetal programs is useful when you want to elicit a certain type of behaviour in the core, making it is possible to quickly verify if implemented changes have the desired impact. However, testing a design should be done by using proper benchmarks that reflect real-world scenarios.

5.2.1 Chipyard Benchmark Suite

Chipyard includes support for running a simple benchmark suite consisting of 10 benchmarks. These benchmarks are more indicative of the simulated core's performance than running specific custom tests, but might not reflect real-world scenarios to an adequate extent. The source codes for the included benchmarks are located in *~/chipyard/toolchains/riscv-tools/riscv-tests/benchmarks* and are run using *make run-bmark-tests*. A brief description of the benchmarks are presented in Table 7 below.

Table 7: Chipyard Benchmark Suite

Benchmark	Description
dhrystone	Synthetic integer performance benchmark
median	Applies a three element median filter on input data
mm	Unknown functionality
mt-matmul	Multiplies two matrices and stores the result in a third matrix (multi-threaded)
mt-vvadd	Adds two arrays and stores the result in a third array (multi-threaded)
multiply	Simple multiplication between elements in two arrays
qsort	Sorts an array of integers using a quicksort function
spmv	Multiplies a sparse matrix with an array
towers	Towers of Hanoi puzzle problem benchmark
vvadd	Adds two arrays and stores the result in a third array

5.3 FPGA-accelerated Simulation

Benchmarking a core using real-world equivalent benchmark suites requires a substantial amount of computing. This takes time. In turn, running such benchmark suites would not be feasible if the simulation were to be run solely in software. Therefore, accelerating the simulation is necessary, and using FPGAs is a common and efficient method of doing this. The FPGAs available on the Idun Cluster allows for hardware accelerated simulation and enables its users to produce tangible results.

FireSim has been used to run a simple Linux kernel on the FPGA. Running FireSim on the FPGAs on the Idun cluster was moderately straightforward with the aid of the GitHub wiki page maintained by EECS-NTNU [29]. The commands on the wiki page have been modified to fit this project's use case. The command used to flash the design bitstream onto the FPGAs was modified to:

> *fpga-util.py -f af -b /git/chipyard/sims/firesim/sim/generated-src/alveo/FireSim-FireSimLargeBoomConfig-WithAutoCounter_BaseF1Config1Mem_F30MHz/u250/vivado_proj/firesim.bit*

FireSimLargeBoomConfig specifies which BOOM implementation is flashed onto the FPGA and is replaced by *FireSimLargeBoomWakeupConfig* to flash the wakeup implementation. *WithAutoCounter* adds support for AutoCounters. Adding *F30MHz* specifies at which frequency the FPGA will run. After flashing the design bitstream onto the FPGA, a modified FireSim command was run. The modified FireSim commands adds *+trace-select=3*, *+trace-start=ffffffff00008013*, *+trace-end=ffffffff00010013*, and *+autocounter-readrate=100000* to the permissives of the original command found on the wiki page. Adding these permissives enables TracerV triggering with AutoCounters. Running this FireSim command boots up the simple Linux kernel on the FPGA, and benchmarks can be initiated.

5.3.1 SPEC CPU® 2017

"The Standard Performance Evaluation Corporation (SPEC) is a non-profit organization that was formed to establish, maintain, and endorse standardized benchmarks and tools to evaluate performance and energy efficiency for the newest generation of computing systems", from the SPEC home page [30].

SPEC CPU® 2017 is the newest CPU benchmark suite released by SPEC and is a licensed benchmark suite. The benchmark suite will be referred to as SPEC2017 going forward. NTNU has the required license to use SPEC2017, and the Department of Computer Science has precompiled all the benchmarks in the suite for RISC-V architectures using the GCC 10 toolchain. Compiling with this toolchain requires heterogeneous compiler flags across the benchmarks, making the results non-compliant with official SPEC2017 results. However, they provide results that are reliable in terms of how well a design performs.

The official SPEC2017 benchmark suite consists of 43 benchmarks, whereas only 39 are used in this project. The benchmarks are divided into 500-series benchmarks and 600-series benchmarks which are intended for rate and speed tests, respectively. Rate tests are used to measure throughput by running multiple instances of the same benchmark concurrently, while the speed tests are used to measure how fast a processor can execute a benchmark. The 39 benchmarks include benchmarks for both integer and floating point purposes. By running the 39 benchmarks as a single benchmark suite, the performance over a variety of applications can be measured. An overview of the 39 benchmarks used in this project are presented in Table 8 below.

Table 8: SPEC2017 Benchmark Suite

Benchmark		**Type**	**Description**
500-series	**600-series**	**N/A**	**N/A**
500.perlbench_r	600.perlbench_s	Integer	Perl interpreter
502.gcc_r	602.gcc_s	Integer	GNU C compiler
503.bwaves_r	N/A	Floating point	Explosion modeling
505.mcf_r	605.mcf_s	Integer	Route planning
507.cactuBSSN_r	607.cactuBSSN_s	Floating point	Physics: relativity
508.namd_r	N/A	Floating point	Molecular dynamics
510.parest_r	N/A	Floating point	Biomedical imaging: optical tomography with finite elements
511.povray_r	N/A	Floating point	Ray tracing
519.lbm_r	619.lbm_s	Floating point	Fluid dynamics
520.omnetpp_r	620.omnetpp_s	Integer	Discrete event simulation - computer network
521.wrf_r	621.wrf_s	Floating point	Weather forecasting
523.xalancbmk_r	623.xalancbmk_s	Integer	XML to HTML via XSLT
525.x264_r	625.x264_s	Integer	Video compression
527.cam4_r	N/A	Floating point	Atmosphere modeling
N/A	628.pop2_s	Floating point	Wide-scale ocean modeling (climate level)
531.deepsjeng_r	631.deepsjeng_s	Integer	Artificial intelligence: alpha-beta tree search (Chess)
538.imagick_r	638.imagick_s	Floating point	Image manipulation
541.leela_r	641.leela_s	Integer	Artificial intelligence: Monte Carlo tree search (Go)
544.nab_r	644.nab_s	Floating point	Molecular dynamics
548.exchange2_r	648.exchange2_s	Integer	Artificial intelligence: recursive solution generator (Sudoku)
549.fotonik3d_r	649.fotonik3d_s	Floating point	Computational electromagnetics
554.roms_r	N/A	Floating point	Regional ocean modeling
557.xz_r	657.xz_s	Integer	General data compression

Source: *Q13 on the SPEC CPU® 2017 benchmark overview [31]*

6 Results

This chapter will briefly discuss and showcase results gathered from running the included Chipyard benchmark for software simulations, and the SPEC2017 benchmark suite for FPGA-accelerated simulations. Further discussion of these results can be found in Section 7.

6.1 Software Simulation

Results have been produced by running the included Chipyard software benchmark on different configurations and implementations of the BOOM core. These results are mainly used to indicate if the implementations are suitable to be fully tested with the SPEC2017 benchmark using FPGA-accelerated simulations.

6.1.1 Small BOOM simulations

The small BOOM core was simulated in its default configuration for both the baseline implementation and the wakeup implementation. Both implementations were simulated with one and two memory ports. These simulations are quicker to run than the large BOOM core simulations and serve as preliminary results for how well the wakeup implementation performs. The results are shown in Figure 6 below.

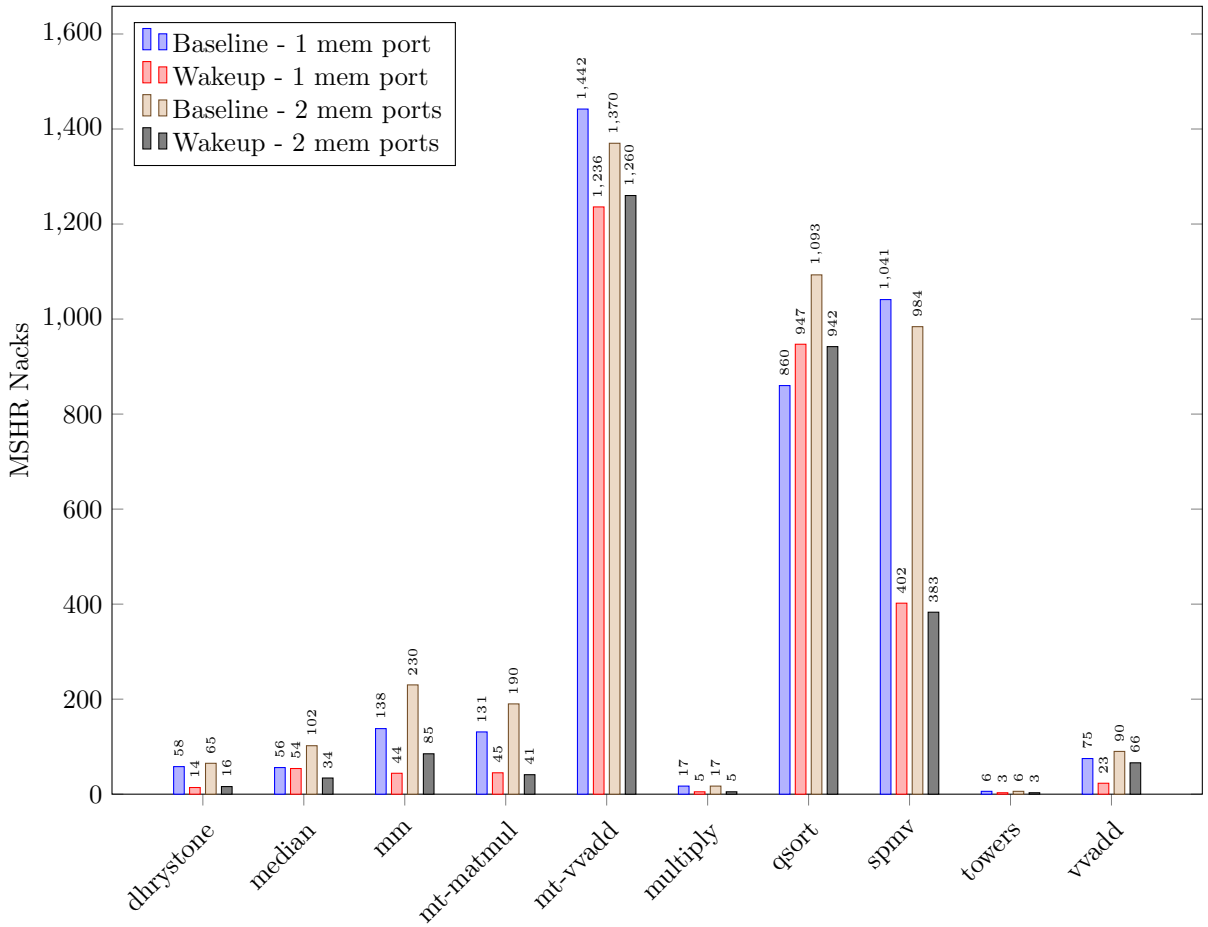


Figure 6: Software simulated small BOOM

6.1.2 Default Large BOOM

The default large BOOM core configuration was used for simulating the baseline and wakeup implementations. Figure 7 shows the results for simulating with one memory port, while Figure 8 shows the results for simulating with two memory ports. These results show that there is some reduction in MSHR nacks for most benchmarks, but a minor increase for the qsort and vvadd benchmarks.

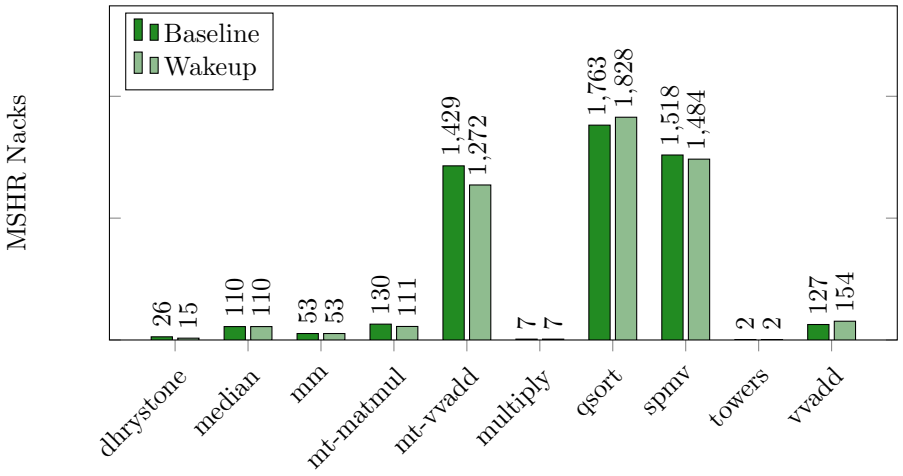


Figure 7: Default large BOOM (1 memory port and 4 MSHRs)

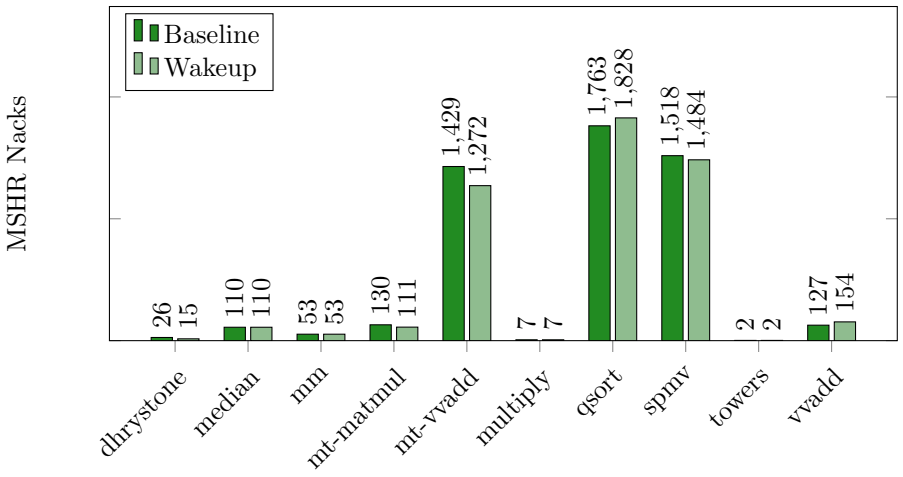


Figure 8: Default large BOOM (2 memory ports and 4 MSHRs)

6.1.3 Customized Large BOOM

Two custom core configurations with two memory ports were simulated for both implementations to investigate how increasing or reducing the number of MSHRs would impact the number of MSHR nacks issued to the core. Figure 9 shows the results from simulating with eight MSHRs and two memory ports, while Figure 10 shows the results from simulating with two MSHRs and two memory ports. The results show that there is no change in the number of MSHRs nacks when the core has eight MSHRs at its disposal. On the other hand, when the core only has two MSHRs at its disposal, there is a significant reduction in the number of MSHR nacks for all benchmarks that experienced more than ten MSHR nacks in its baseline implementation. These results indicate that there is merit to run FPGA-accelerated simulations with the wakeup implementation.

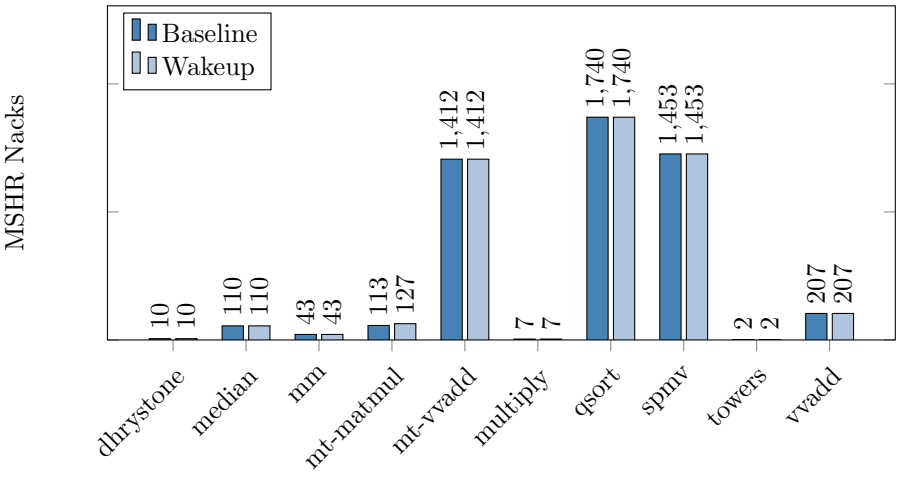


Figure 9: Large BOOM (2 memory ports and 8 MSHRs)

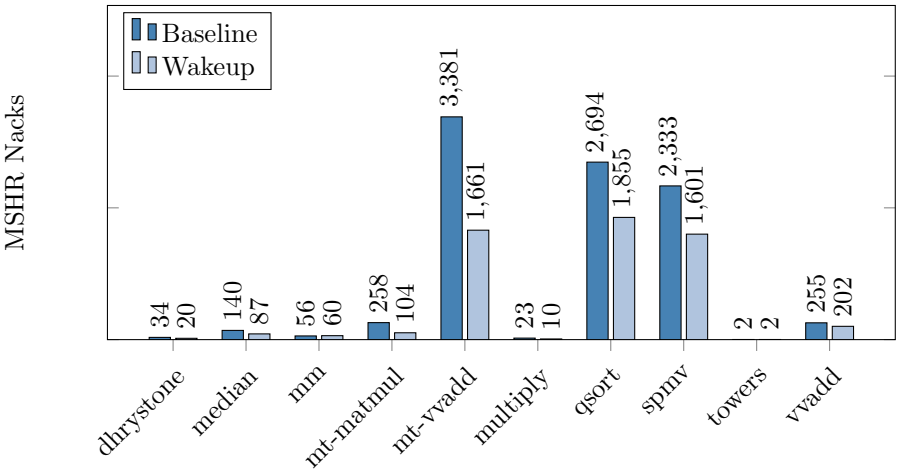


Figure 10: Large BOOM (2 memory ports and 2 MSHRs)

6.2 FPGA-accelerated Simulation

The SPEC2017 benchmark suite was tested on the baseline and wakeup design implementations. The default large BOOM configuration was used for both implementations, and the *test* input was used for the SPEC2017 benchmark suite. Other available inputs are the *ref* input and the *train* input. The different inputs invoke different commands with different arguments, resulting in different execution times. However, the test input is sufficiently large to properly test a design, as the benchmark suite runs for over two trillion cycles. Figure 11 shows the amount of MSHR nacks issued from the data cache to the LSU for the baseline and wakeup implementations when running the SPEC2017 benchmark suite. The bar graphs in the figure are rounded off, but the results show a reduction in MSHR nacks, from the baseline implementation to the wakeup implementation, of approximately 29.38%.

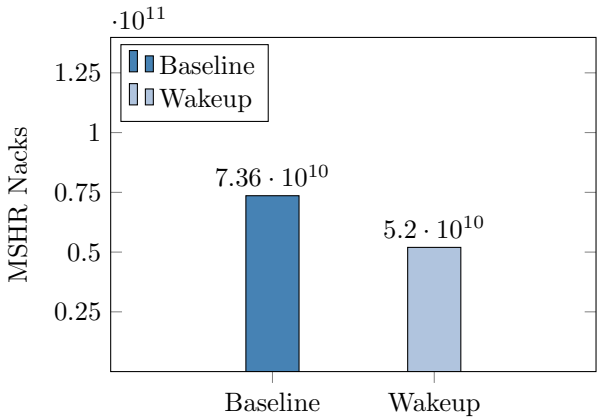


Figure 11: MSHR nacks for the SPEC2017 benchmark suite

To further evaluate the impact of the wakeup implementation, IPC statistics were measured for each individual benchmark in the SPEC2017 benchmark suite for both the baseline and the wakeup implementations. These results are presented in Figure 12 on the next page. The IPCs of 39 benchmarks were measured, where 20 of them have differing IPCs between the two implementations. Section 7 contains a discussion in regards to the potential reasons for the outcome of the results presented in Figure 11 and Figure 12.

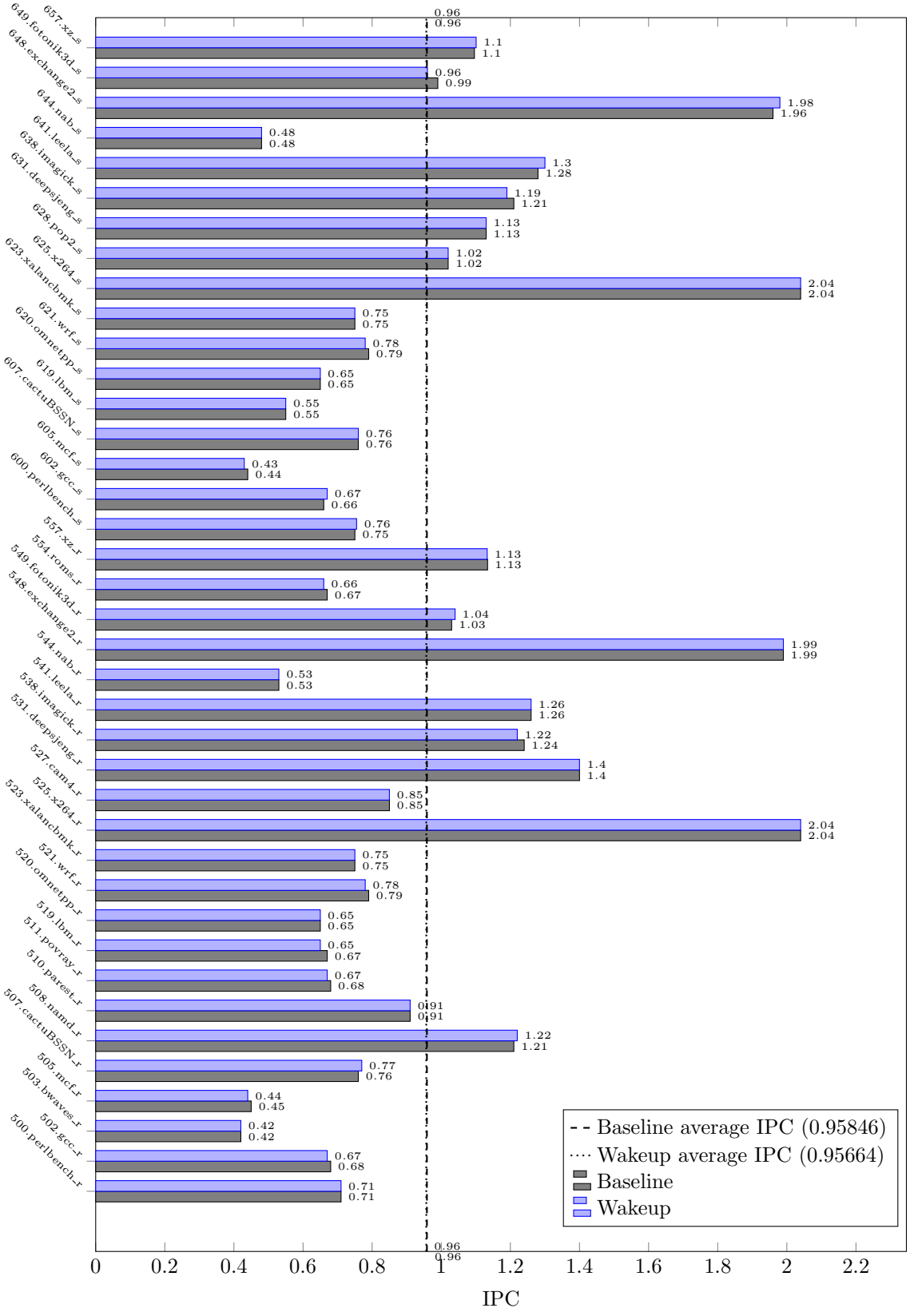


Figure 12: IPC for the entire SPEC2017 benchmark suite

7 Discussion

This section will discuss the performance and power implications of the results presented in Section 6, as well as a holistic discussion.

7.1 Performance

Figure 12 in Section 6.2 presents the IPCs for all benchmarks in the SPEC2017 benchmark suite. For 19 of the 39 benchmarks, the IPC did not change from the baseline to the wakeup implementation, indicating an equal performance between the implementations for those benchmarks. To better evaluate the IPC changes between the implementations for the remaining 20 benchmarks, they are compared with an additional factor; The number of load instructions compared to the total amount of instructions.

Figure 13 presents the 19 benchmarks that experienced a change in IPC between the implementations in ascending order of load instruction percentage. The graph shows that all eight benchmarks where the wakeup implementation saw an increase in IPC has load instruction percentages between 20% and 28%. However, in this 20% to 28% load instruction percentage, five benchmarks saw a decrease in IPC for the wakeup implementation, and ten benchmarks saw no change in IPC between the implementations. Of the eight benchmarks that saw an increase in IPC, five are integer benchmarks and three are floating-point benchmarks. Of the five benchmarks that saw a decrease in IPC, one is a integer benchmark and four are floating-point benchmarks. Of the ten benchmarks that saw no change in IPC, six are integer benchmarks and four are floating-point benchmarks.

These observations imply that the wakeup implementation performs better or is equal to the baseline implementation for workloads with a medium number of load instructions. Additionally, the wakeup implementation performs better with integer-intensive rather than floating-point-intensive workloads.

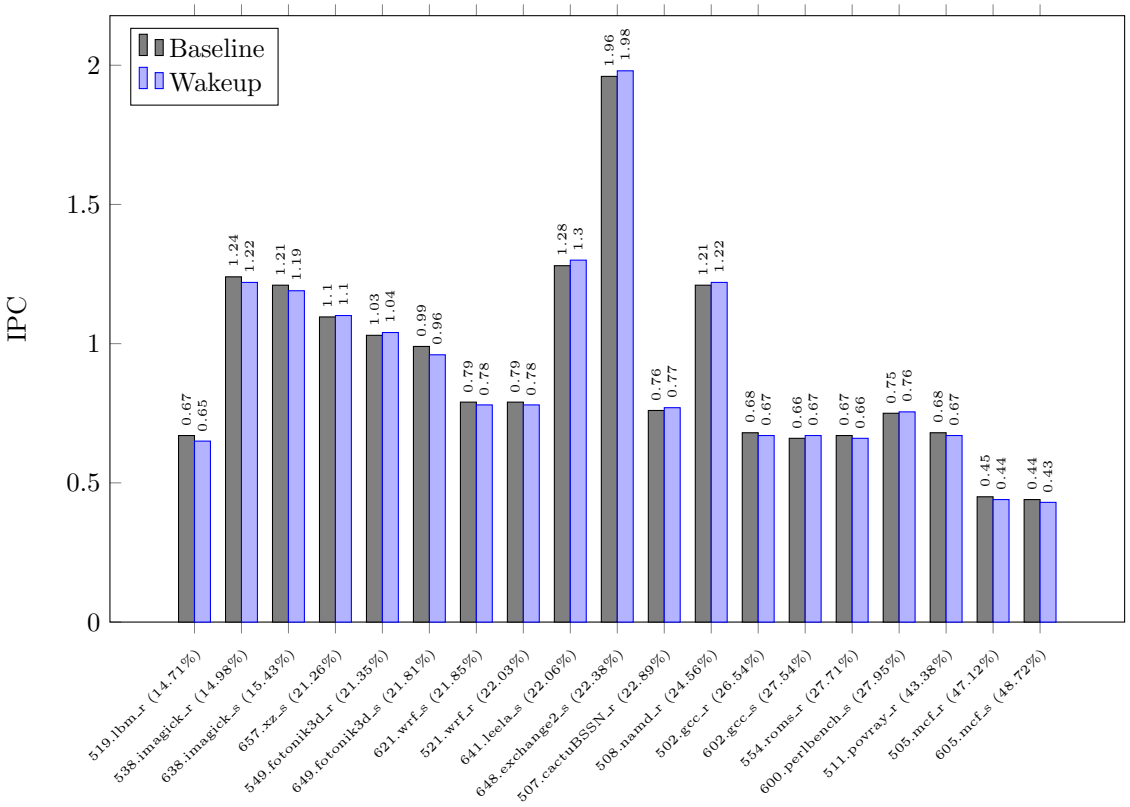


Figure 13: IPC for the SPEC2017 benchmark suite
(benchmarks with non-equal IPC, sorted from low to high based on load percentage)

There might be several possible reaons as to why the performance of the wakeup implementation is worse for certain workloads compared to the baseline implementation. One reason might be the aggressiveness of the re-issuing of nacked loads. The baseline implementation re-issues loads nacked by an MSHR nack even though there are no MSHRs that can handle the request. This results in more MSHR nacks being issued, but might also reduce the latency when re-issuing loads compared to the wakeup implementation. In the wakeup implementation the data cache needs to notify the LSU that there is an available MSHR before it can even try to re-issue an MSHR nacked load. The baseline implementation ignores this and can re-issue whenever the LSU has time. This means that the baseline implementation might save a cycle each time such a scenario occurs. Figure 14 and Figure 15 presents a visual representation of how this scenario would occur.

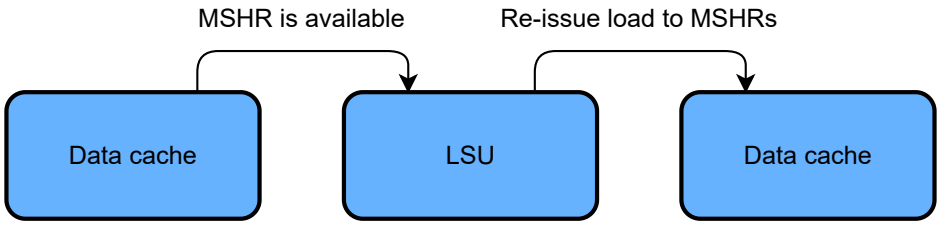


Figure 14: Re-issuing a MSHR nacked load, wakeup implementation

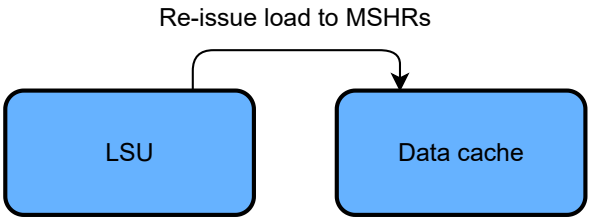


Figure 15: Re-issuing a MSHR nacked load, baseline implementation

7.2 Power Consumption

For this project, reducing the dynamic power consumption is possible through reducing the activity factor from Equation 1. Reducing the number of MSHR nacks issued from the data cache to the LSU might reduce the overall activity factor of the processor, resulting in a more power efficient unit. Although the wakeup implementation has reduced the number of MSHR nacks by approximately 30% compared to the baseline implementation, the total power consumption will not be reduced by 30%. This is because when there is less activity in the wires the leakage current might increase, as well as the fact that the wakeup implementation implements new wires and components to support the new logic which might increase power draw. However, since the reduction in MSHR nacks is so large, coupled with the small power overhead of adding new wires and components, there is reason to believe that the overall power consumption has been reduced.

8 Future Work

This section will list and discuss some possible future investigations and work that can be done to further improve the BOOM wakeup implementation.

8.1 Implementing Wakeup for Other Nacks

Currently, the wakeup implementation is only implemented for MSHR nacks. Expanding this to the other four types of nack signals might increase performance and power efficiency further. The framework for adding wakeup to the other nack types has already been implemented with the wakeup implementation, but the correct signals and timings need to be identified to ensure correctness in the LSU.

8.2 Measuring Power Consumption

Measuring the actual power usage to draw a conclusion to the hypothesis made in Section 7.2 is needed. This might not be possible with the current test setup used for this project.

8.3 Further Benchmarking With SPEC2017

The SPEC2017 benchmark suite was run with the test input, with the large BOOM configuration. Running the benchmark with the ref input might give more accurate results. It would also be beneficial to run the benchmark suite in official SPEC compliant way. Furthermore, running the benchmark suite on a BOOM core configured similarly to the SonicBOOM core might produce results that show how competitive the core is compared to state-of-the-art processor cores.

9 Conclusion

The wakeup implementation optimizes how the BOOM core re-issues loads that have been nacked by the MSHRs in the data cache. The implementation targets logic in the scheduler that allows the LSU to re-issue previously nacked loads. The results show that the implementation reduced the corresponding nack signals by approximately 30% compared to the baseline implementation. Running the SPEC CPU® 2017 benchmark suite shows that the performance is comparable to the baseline implementation. Furthermore, performance results show that running workloads with a load instruction count of 20% to 28% of the total instruction count is optimal for the wakeup implementation. For future work it is suggested to implement wakeup logic for the remaining four nack types the data cache identifies.

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Sixth Edition. Morgan Kaufmann, 2019, pp. 2–5.

[2] Robert H. Dennard et al. 'Design of ion-implanted MOSFET's with very small physical dimensions'. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[3] Gordon E. Moore. 'Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.' In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[4] Wm. A. Wulf and Sally A. McKee. 'Hitting the memory wall: Implications of the obvious'. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[5] Shekhar Borkar and Andrew A. Chien. 'The future of microprocessors'. In: *Communications of the ACM* 54.5 (2011), pp. 67–77.

[6] Jeff Parkhurst, John Darringer and Bill Grundmann. 'From Single Core to Multi-Core: Preparing for a New Exponential'. In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '06. San Jose, California: Association for Computing Machinery, 2006, pp. 67–72. ISBN: 1595933891. DOI: 10.1145/1233501.1233516. URL: https://doi.org/10.1145/1233501.1233516.

[7] Arm Limited. *Instruction Set Architecture (ISA)*. https://www.arm.com/glossary/isa. Accessed: 25.05.2023.

[8] Arm Limited. *RISC*. https://www.arm.com/glossary/risc. Accessed: 25.05.2023.

[9] Inc. Synopsis. *What is Low Power Design?* https://www.synopsys.com/glossary/what-is-low-power-design.html. Accessed: 09.06.2023.

[10] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation. Dec. 2019.

[11] Andrew Waterman, Krste Asanović and John Hauser, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. RISC-V International. Dec. 2021.

[12] RISC-V International. *RISC-V*. https://riscv.org/. Accessed: 13.12.2022.

[13] Jerry Zhao et al. 'Sonicboom: The 3rd generation berkeley out-of-order machine'. In: *Fourth Workshop on Computer Architecture Research with RISC-V*. Vol. 5. May 2020.

[14] The Regents of the University of California. *Welcome to RISCV-BOOM's documentation!* https://docs.boom-core.org/en/latest/. Accessed: 12.12.2022.

[15] Berkeley Architecture Research. *The Berkeley Out-of-Order RISC-V Processor (GitHub repository)*. https://github.com/riscv-boom/riscv-boom. Accessed: 13.12.2022.

[16] The Regents of the University of California. *The Load/Store Unit (LSU)*. https://docs.boom-core.org/en/latest/sections/load-store-unit.html. Accessed: 30.03.2023.

[17] Adam Izraelevitz et al. 'Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations'. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[18] Patrick S. Li, Adam M. Izraelevitz and Jonathan Bachrach. *Specification for the FIRRTL Language*. Tech. rep. UCB/EECS-2016-9. EECS Department, University of California, Berkeley, Feb. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html.

[19] Berkeley Architecture Research. *Adding a FIRRTL Transform*. https://chipyard.readthedocs.io/en/stable/Customization/Firrtl-Transforms.html#firrtl-transforms. Accessed: 01.05.2023.

[20] Chris Lattner et al. 'MLIR: Scaling Compiler Infrastructure for Domain Specific Computation'. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[21] LLVM. *The LLVM Compiler Infrastructure*. https://llvm.org/. Accessed: 03.05.2023.

[22] Berkeley Architecture Research. *Welcome to Chipyard's documentation version ('1.8.1')!* https://chipyard.readthedocs.io/en/stable/. Accessed: 06.04.2023.

[23] Berkeley Architecture Research. *Chipyard Framework (GitHub repository)*. https://github.com/ucb-bar/chipyard. Accessed: 13.12.2022.

[24] Harrison Liew et al. 'Hammer: A Modular and Reusable Physical Design Flow Tool: Invited'. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 1335–1338. ISBN: 9781450391429. DOI: 10.1145/3489517.3530672. URL: https://doi.org/10.1145/3489517.3530672.

[25] Wilson Snyder. *Welcome to Verilator*. https://www.veripool.org/verilator/. Accessed: 05.04.2023.

[26] Sagar Karandikar et al. 'FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud'. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[27] Nathan Pemberton and Alon Amid. 'FireMarshal: Making HW/SW Co-Design Reproducible and Reliable'. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 299–309. DOI: 10.1109/ISPASS51385.2021.00052.

[28] Magnus Sjalander et al. 'EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure'. In: (Feb. 2022). DOI: 10.48550/arXiv.1912.05848. arXiv: 1912.05848. URL: http://arxiv.org/abs/1912.05848.

[29] EECS-NTNU. *$u250_firesim$*. https://github.com/EECS-NTNU/chipyard/wiki/u250_firesim. Accessed: 09.06.2023.

[30] Standard Performance Evaluation Corporation. *Standard Performance Evaluation Corporation*. https://www.spec.org/. Accessed: 08.06.2023.

[31] Standard Performance Evaluation Corporation. *Standard Performance Evaluation Corporation*. https://www.spec.org/cpu2017/Docs/overview.html. Accessed: 08.06.2023.

Appendices

A Master's Agreement

NTNU

Masteravtale/hovedoppgaveavtale

Sist oppdatert 11. november 2020

Fakultet	**Fakultet for informasjonsteknologi og elektroteknikk**
Institutt	**Institutt for elektroniske systemer**
Studieprogram	**MSELSYS**
Emnekode	**TFE4930**

Studenten	
Etternavn, fornavn	**Young, Marcus Stensby**
Fødselsdato	**09.04.1997**
E-postadresse ved NTNU	**marcussy@stud.ntnu.no**

Tilknyttede ressurser	
Veileder	**Magnus Själander**
Eventuelle medveiledere	**Amund Bergland Kvalsvik**
Eventuelle medstudenter	

Oppgaven	
Oppstartsdato	**13.01.2023**
Leveringsfrist	**09.06.2023**
Oppgavens arbeidstittel	**Improving Memory Scheduling on an Out-of-Order Processor**
Problembeskrivelse	**Oppgaven blir å implementere wake-up logikk i Load-Store Unit i BOOM-kjernen, og vurdere hvor mye CPU-ytelse dette vil gi.**

Risikovurdering og datahåndtering	
Skal det gjennomføres risikovurdering? **Dersom «ja», har det blitt gjennomført?**	**Nei** **Nei**
Skal det søkes om godkjenninger? **(REK*, NSD**)**	**Nei**
Skal det skrives en konfidensialitetsavtale i forbindelse med oppgaven? **Hvis «ja», har det blitt gjort?**	**Nei** **Nei**

* Regionale komiteer for medisinsk og helsefaglig forskningsetikk (https://rekportalen.no)

** Norsk senter for forskningsdata (https://nsd.no/)

Eventuelle emner som skal inngå i mastergraden

Retningslinjer - rettigheter og plikter

Formål

Avtale om veiledning av masteroppgaven/hovedoppgaven er en samarbeidsavtale mellom student, veileder og institutt. Avtalen regulerer veiledningsforholdet, omfang, art og ansvarsfordeling.

Studieprogrammet og arbeidet med oppgaven er regulert av Universitets- og høgskoleloven, NTNUs studieforskrift og gjeldende studieplan. Informasjon om emnet, som oppgaven inngår i, finner du i emnebeskrivelsen.

Veiledning

Studenten har ansvar for å

- Avtale veiledningstimer med veileder innenfor rammene master-/hovedoppgaveavtalen gir.
- Utarbeide framdriftsplan for arbeidet i samråd med veileder, inkludert veiledningsplan.
- Holde oversikt over antall brukte veiledningstimer sammen med veileder.
- Gi veileder nødvendig skriftlig materiale i rimelig tid før veiledning.
- Holde instituttet og veileder orientert om eventuelle forsinkelser.
- Inkludere eventuell(e) medstudent(er) i avtalen.

Veileder har ansvar for å

- Avklare forventninger om veiledningsforholdet.
- Sørge for at det søkes om eventuelle nødvendige godkjenninger (etikk, personvernhensyn).
- Gi råd om formulering og avgrensning av tema og problemstilling, slik at arbeidet er gjennomførbart innenfor normert eller avtalt studietid.
- Drøfte og vurdere hypoteser og metoder.
- Gi råd vedrørende faglitteratur, kildemateriale, datagrunnlag, dokumentasjon og eventuelt ressursbehov.
- Drøfte framstillingsform (eksempelvis disposisjon og språklig form).
- Drøfte resultater og tolkninger.
- Holde seg orientert om progresjonen i studentens arbeid i henhold til avtalt tids- og arbeidsplan, og følge opp studenten ved behov.
- Sammen med studenten holde oversikt over antall brukte veiledningstimer.

Instituttet har ansvar for å

- Sørge for at avtalen blir inngått.
- Finne og oppnevne veileder(e).
- Inngå avtale med annet institutt/ fakultet/institusjon dersom det er oppnevnt ekstern medveileder.
- I samarbeid med veileder holde oversikt over studentens framdrift, antall brukte veiledningstimer, og følge opp dersom studenten er forsinket i henhold til avtalen.
- Oppnevne ny veileder og sørge for inngåelse av ny avtale dersom:
    - Veileder blir fraværende på grunn av eksempelvis forskningstermin, sykdom, eller reiser.
    - Student eller veileder ber om å få avslutte avtalen fordi en av partene ikke følger den.
    - Andre forhold gjør at partene finner det hensiktsmessig med ny veileder.
- Gi studenten beskjed når veiledningsforholdet opphører.
- Informere veileder(e) om ansvaret for å ivareta forskningsetiske forhold, personvernhensyn og veiledningsetiske forhold.
- Ønsker student, eller veileder, å bli løst fra avtalen må det søkes til instituttet. Instituttet må i et slikt tilfelle oppnevne ny veileder.

Avtaleskjemaet skal godkjennes når retningslinjene er gjennomgått.

Godkjent av

Marcus Stensby Young
Student

25.01.2023
Digitalt godkjent

Magnus Själander
Veileder

25.01.2023
Digitalt godkjent

Kirsti Klemetsaune
Institutt

26.01.2023
Digitalt godkjent

Master`s Agreement / Main Thesis Agreement

Faculty	**Faculty of Information Technology and Electrical Engineering**
Institute	**Department of Electronic Systems**
Programme Code	**MSELSYS**
Course Code	**TFE4930**

Personal Information

Surname, First Name	**Young, Marcus Stensby**
Date of Birth	**09.04.1997**
Email	**marcussy@stud.ntnu.no**

Supervision and Co-authors

Supervisor	**Magnus Själander**
Co-supervisors (if applicable)	**Amund Bergland Kvalsvik**
Co-authors (if applicable)	

The Master`s thesis

Starting Date	**13.01.2023**
Submission Deadline	**09.06.2023**
Thesis Working Title	**Improving Memory Scheduling on an Out-of-Order Processor**
Problem Description	**Oppgaven blir å implementere wake-up logikk i Load-Store Unit i BOOM-kjernen, og vurdere hvor mye CPU-ytelse dette vil gi.**

Risk Assessment and Data Management	
Will you conduct a Risk Assessment? **If "Yes", Is the Risk Assessment Conducted?**	**No** **No**
Will you Apply for Data Management? **(REK*, NSD**)**	**No**
Will You Write a Confidentiality Agreement? **If "Yes", Is the Confidentiality Agreement Conducted?**	**No** **No**

* REK -- https://rekportalen.no/

** Norwegian Centre for Research Data (https://nsd.no/nsd/english/index.html)

Topics to be included in the Master`s Degree (if applicable)

Guidelines – Rights and Obligations

Purpose

The Master's Agreement/ Main Thesis Agreement is an agreement between the student, supervisor, and department. The agreement regulates supervision conditions, scope, nature, and responsibilities concerning the thesis.

The study programme and the thesis are regulated by the Universities and University Colleges Act, NTNU's study regulations, and the current curriculum for the study programme.

Supervision

The student is responsible for

- Arranging the supervision within the framework provided by the agreement.
- Preparing a plan of progress in cooperation with the supervisor, including a supervision schedule.
- Keeping track of the counselling hours.
- Providing the supervisor with the necessary written material in a timely manner before the supervision.
- Keeping the institute and supervisor informed of any delays.
- Adding fellow student(s) to the agreement, if the thesis has more than one author.

The supervisor is responsible for

- Clarifying expectations and how the supervision should take place.
- Ensuring that any necessary approvals are acquired (REC, ethics, privacy).
- Advising on the demarcation of the topic and the thesis statement to ensure that the work is feasible within agreed upon time frame.
- Discussing and evaluating hypotheses and methods.
- Advising on literature, source material, data, documentation, and resource requirements.
- Discussing the layout of the thesis with the student (disposition, linguistic form, etcetera).
- Discussing the results and the interpretation of them.
- Staying informed about the work progress and assist the student if necessary.
- Together with the student, keeping track of supervision hours spent.

The institute is responsible for

- Ensuring that the agreement is entered into.
- Find and appoint supervisor(s).
- Enter into an agreement with another department / faculty / institution if there is an external co-supervisor.
- In cooperation with the supervisor, keep an overview of the student's progress, the number of supervision hours. spent, and assist if the student is delayed by appointment.
- Appoint a new supervisor and arrange for a new agreement if:
  - The supervisor will be absent due to research term, illness, travel, etcetera.
  - The student or supervisor requests to terminate the agreement due to lack of adherence from either party.
  - Other circumstances where it is appropriate with a new supervisor.
- Notify the student when the agreement terminates.
- Inform supervisors about the responsibility for safeguarding ethical issues, privacy and guidance ethics
- Should the cooperation between student and supervisor become problematic, either party may apply to the department to be freed from the agreement. In such occurrence, the department must appoint a new supervisor

This Master`s agreement must be signed when the guidelines have been reviewed.

Signatures

Marcus Stensby Young
Student

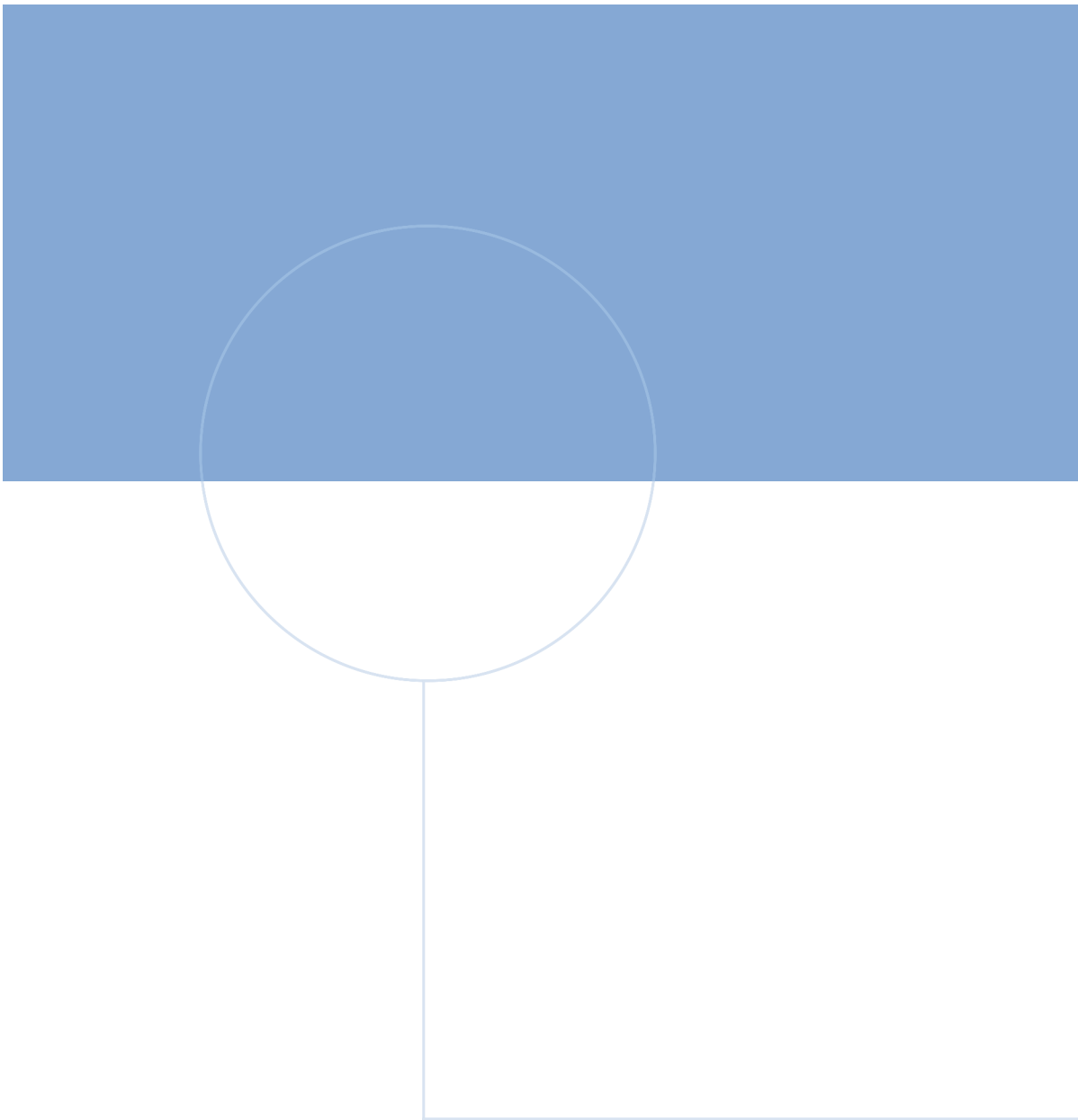
25.01.2023
Digitally approved

Magnus Själander
Supervisor

25.01.2023
Digitally approved

Kirsti Klemetsaune
Department

26.01.2023
Digitally approved



NTNU

Norwegian University of
Science and Technology