Master's thesis

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

Marvin Ademi

Enhancing Performance through Parallel Memory and Arithmetic Operations in the CV32E40X Core

Master's thesis in Electronic Systems Design
Supervisor: Thomas Tybell
Co-supervisor: Øystein Knauserud
June 2023

NTNU
Norwegian University of
Science and Technology

Marvin Ademi

Enhancing Performance through Parallel Memory and Arithmetic Operations in the CV32E40X Core

Master's thesis in Electronic Systems Design
Supervisor: Thomas Tybell
Co-supervisor: Øystein Knauserud
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

Project Description

Embedded devices are vital in various applications, from consumer electronics to industrial control systems. With the increasing demand for more sophisticated and feature-rich devices, the need for high-performance, low-power processors in embedded systems has become crucial. The CV32E40X core is part of a set of open-source CPU cores called the CV32E40X-family, spawned initially from the RI5CY project. The CV32E40X is a low-power embedded processor designed for compute-intensive applications.

This project aims to enhance the performance of the CV32E40X core by enabling parallel execution of memory operations with other operations while considering the trade-offs in area and power consumption. The project builds upon the understanding that optimizing performance and area is challenging in embedded processor design.

Abstract

The study is written in collaboration with Silicon Labs and focuses on enhancing the performance of the CV32E40X processor core by enabling parallel execution of memory operations with other operations. The introduction highlights the increasing demands of modern applications and the importance of improving processor cores. The CV32E40X core, based on the RISC-V architecture, is aimed at compute-intensive applications. This research aims to increase the core's performance without exceeding a 20% increase in area.

Several modifications were made to the CV32E40X core to achieve the objective. A skid buffer introduced variable delay for the LSU and imposed backpressure on the OBI interface. The ID, EX, and WB stages were modified to enable parallel execution. An instruction ID tracker was also implemented to maintain program order during instruction commitment.

The benchmark tests, including coremark, hello-world, ricv_arithmetic_basic_test_1, and csr_instr_asm, were conducted on both the baseline and modified designs. The results indicate that the baseline design outperforms the modified version regarding IPC. The modified design cannot fully execute coremark and hello-world, suggesting potential performance limitations due to unresolved bugs. The results from running csr_instr_asm and ricv_arithmetic_basic_test_1 showed similar performance between the baseline and modified CV32E40X. In contrast, for coremark and hello-world, the baseline is 12.5% and 3.1% better in terms of IPC, respectively.

Bug analysis reveals potential culprits for the observed bugs, such as issues with ready signals in the WB stage, mishandling of split memory operations, and overflow in outstanding transactions. The instruction log also exhibits inconsistencies in tracking instructions, which affects the verification phase and test result interpretation.

Furthermore, power, area, and timing analysis show a slight increase in power consumption and area utilization for the modified design compared to the baseline. However, considering the unresolved bugs and the potential for future improvements, the small overhead of 2.6% and 3.3% in power and area leaves room for optimization.

The conclusion highlights the need for bug resolution for future work. Further performance optimizations, thorough testing with diverse workloads, and exploration of advanced features were also suggested. Resolving the identified bugs, such as those affecting the LSU, is crucial to enhancing functionality and accuracy. Through these efforts, the modified CV32E40X core has the potential to achieve improved performance, reliability, and compatibility for a wide range of embedded applications.

Sammendrag

Studiet er skrevet i samarbeid med Silicon Labs og fokuserer på å forbedre ytelsen til CV32E40X-prosessorkjernen ved å muliggjøre parallell utførelse av minneoperasjoner med andre operasjoner. Introduksjonen fremhever de økende kravene til moderne applikasjoner og viktigheten av å forbedre prosessorkjerner. CV32E40X-kjernen, basert på RISC-V-arkitekturen, er rettet mot dataintensive applikasjoner. Denne forskningen tar sikte på å øke kjernens ytelse uten å overstige en 20% økning i areal.

Det ble gjort flere modifikasjoner på CV32E40X-kjernen for å nå målet. En skid buffer introduserte variabel forsinkelse for LSU-en og påla mottrykk på OBI-grensesnittet. ID-, EX- og WB-stadiene ble modifisert for å muliggjøre parallell utførelse. En instruksjons-ID-sporer ble også implementert for å opprettholde programrekkefølgen.

Test programmene, inkludert coremark, hello-world, ricv_aritmetic_basic_test_1, og csr_instr_asm, ble utført på både grunnlinjen og modifisert design. Resultatene indikerer at grunnlinjedesignet overgår den modifiserte versjonen angående IPC. Den modifiserte designen kan ikke utføre coremark og hello-world fullt ut, noe som antyder potensielle ytelsesbegrensninger på grunn av uløste feil i funksjonaliteten av kjernen. Resultatene fra å kjøre csr_instr_asm og ricv_aritmetic_basic_test_1 viste lignende ytelse mellom baseline og modifisert cv32e40x. I motsetning til dette, for coremark og hello-world, er grunnlinjen henholdsvis 12,5% og 3,1% bedre når det gjelder IPC.

Feilanalyse avslører potensielle årsaker for de observerte feilene, for eksempel problemer med *ready* signaler i WB-stadiet, feilhåndtering av delt minneoperasjoner og overløp i utestående minnetransaksjoner. Instruksjonsloggen viser også inkonsekvenser i sporingsinstruksjonene, noe som påvirker verifikasjonsfasen og tolkningen av testresultater.

Videre viser energi-, areal- og tidsanalyse en liten økning i energiforbruk og arealutnyttelse for den modifiserte designen sammenlignet med grunnlinjen. Men med tanke på de uløste feilene og potensialet for fremtidige forbedringer, gir den lille overheaden på 2,6% og 3,3% i energi og areal rom for optimalisering.

Konklusjonen fremhever behovet for feilløsning for fremtidig arbeid. Ytterligere ytelsesoptimaliseringer, grundig testing med ulike arbeidsbelastninger og utforskning av avanserte funksjoner ble også foreslått. Å løse de identifiserte feilene, for eksempel de som påvirker LSU, er avgjørende for å forbedre funksjonalitet og nøyaktighet. Gjennom denne innsatsen har den modifiserte CV32E40X-kjernen potensial til å oppnå forbedret ytelse.

Acknowledgement

I want to thank my supervisors, Thomas Tybell from NTNU and Øystein Knauserud from Silicon Labs, for their helpful advice and support. We always had informative conversations during our sessions, and I always anticipated your feedback. Your in-depth comments have significantly contributed to this work's development. I want to thank Øystein for his technical knowledge, which was helpful whenever I ran into problems and needed to weigh my options. The same goes for Thomas; I appreciate his guidance in ensuring I continued developing my thesis and remaining on the right track. I also want to thank Giorgi Solomnishvili for some insightful discussions and inputs on the technical aspects of the thesis.

Lastly, I want to thank my family and friends for their everlasting support.

Table of Contents

List of Tables

List of Figures

Listings

Abbreviations

ISA	Instruction Set Architecture	2
RISC	Reduced Instruction Set Computer	2
ILP	Instruction Level Parallelism	6
OoO	Out-of-Order	5
IPC	Instructions Per Cycle	2
CPI	Cycles Per Instruction	47
sOoO	slice-Out-of-Order	10
MLP	Memory Level Parallelism	10
LSC	Load Slice Core	3
ISEs	Instruction Set Extensions	16
IoT	Internet-of-Things	17
HDL	Hardware Description Language	3
LSU	Load-Store Unit	7
ALU	Arithmatic Logic Unit	19
MUL/DIV	Multiplication/Division	19

AGIs	address-generating instructions	10
IST	Instruction Slice Table	11
RDT	register dependency table	11
FSC	Forward Slice Core	10
FIFO	first in, first out	12
ML	Main Lane	12
DLL	Dependent Load Lane	12
DEL	Dependent Execute Lane	12
HL	Holding Lane	12
IC	Integrated Circuit	9
PC	Program Counter	20
HPC	Hardware Performance Counters	20
OBI	Open Bus Interface	23
IBDA	Iterative Backwards Dependency Analysis	27
mux	multiplexer	37
HDL	Hardware Description Language	3
FPGAs	Field Programmable Gate Arrays	28

CRC	cyclic redundancy check	50
MCUs	microcontrollers	48
UVM	Universal Verification Methodology	xi
CPUs	central processing units	48
RVFI	RISC-V Formal Interface	48
UVM	Universal Verification Methodology	xi

Chapter 1

Introduction

1.1 Motivation

Embedded devices play a crucial role in modern society, as they are used in various applications such as consumer electronics, medical equipment, automobiles, and industrial control systems [1]. Embedded applications have evolved significantly in recent years, with a growing demand for more sophisticated and feature-rich devices [2]. With advancements in microprocessor technology, embedded devices have become more powerful and versatile, allowing for more complex functionality [3; 4]. Integration of wireless and wired communication technologies has enabled embedded devices to connect and interact with other devices [5], adding complexity to application devices. As a result, embedded devices are now designed to perform multiple functions, such as data processing, communication, and storage.

Developing high-performance, low-power processors has been a key factor in enabling the continued growth of the embedded device market [6; 7]. For embedded processors, performance and area are two important factors [8; 9]. For performance, the importance lies in the ability of the processor to execute tasks quickly and efficiently. Resources are often limited for embedded systems. Therefore, the processor's performance directly impacts the system's responsiveness and capability to handle real-time operations. An embedded processor with high performance can ensure smooth multitasking, rapid data processing, and swift response time, resulting in a seamless user experience.

Equally significant is the consideration of area [9]. Area, when discussing processors, refers to the physical size or footprint of the processor. Optimization of area in embedded processors is paramount, as area constraints are common for embedded systems. There are several reasons for tight area constraints; amongst other reasons, embedded systems are often used in devices with limited physical space. An example could be a smartwatch. Power consumption is another reason why area is important—using the smartwatch as an example again—batteries power smartwatches. Therefore, minimizing the processor area contributes to reducing power consumption by reducing active circuitry. As a result, increasing the duration it can operate before needing to be charged.

Balancing performance and area in embedded processor design is a constant challenge. Achieving high performance often comes at the cost of increased complexity, which means increased chip size and power consumption. In the end, the direct impact on the overall system's efficiency, functionality, and form factor gives such importance to performance and area. By emphasizing optimizing for both performance and area, designers will be able to create embedded processors that can deliver great computational power while occupying minimal space and, as a result, enable the development of innovative and resource-efficient embedded systems in a wide range of embedded applications.

RISC-V is a recent addition to the microprocessor landscape, offering a viable alternative to traditional technologies such as Intels x86 and ARMs ARM architecture [10]. It was developed to provide a flexible and efficient Instruction Set Architecture (ISA) that can be used for various applications, from low-power embedded systems to high-performance computing. Its architecture follows the Reduced Instruction Set Computer (RISC) principles. Unlike its competitors, RISC-V boasts an open-source ISA [10], allowing for unlimited contributions from the community without licensing fees. RISC-V's open-source nature also enables customization of the ISA for specific applications. Open-source means that users can optimize the architecture for their specific needs, leading to improved performance and reduced power consumption. The popularity of RISC-V has grown significantly since the establishment of the RISC-V foundation, with a rising number of companies and developers contributing to its development. One such group, OpenHW Group, focuses on creating accessible, high-quality open-source technology [11]. They are driven by their members and individual contributors, collaborating in hardware and software design to produce RISC-V-based processor cores. One of their current projects, CVE4, has received significant contributions from Silicon Labs and includes a family of four 32-bit, 4-stage RISC-V processor cores intended for various embedded applications [12]. The CV32E40X core in the CVE4 family is designed for compute-intensive applications. Performance is, therefore, important factors in the development of the core.

1.2 Objectives and Limitations of Study

This thesis aims to increase the performance of the RISC-V CV32E40X processor core without increasing the area of the core by more than 20%. This thesis's challenge is maintaining correct functionality and increasing performance through the proposed new design.

More specifically, this work aims to implement a scheme that allows the CV32E40X core to execute load/store instructions alongside other workloads. The idea is that allowing other workloads to execute alongside load/store instructions can mitigate stalls due to long memory latency, increasing the performance in return. Given that CV32E40X is a low-power embedded processor, the goal has been to try and implement a power-efficient implementation, despite power consumption not being the main focus of the thesis.

Instructions Per Cycle (IPC) will be used to evaluate performance. IPC is a performance metric used in microprocessor design to quantify the number of instructions executed in a single clock cycle. It measures the efficiency of a processor in executing instructions. It is calculated as the ratio of the number of instructions executed to the number of clock cycles required to execute those instructions [13]. The higher the IPC, the more efficiently the processor can execute instructions and the faster the system's overall performance. By adopting a similar architecture to the Load Slice Core (LSC) architecture, this thesis aims to not only improve the IPC on the CV32E40X core but also leverage the strengths of the LSC design for improved performance and efficiency.

Evaluating the performance of a processor core solely based on IPC may be a limitation of the study. While IPC is an important metric for evaluating the performance of a processor core, it may provide a partial picture of its overall performance. For instance, a processor core that achieves a high IPC may still have a high latency, negatively impacting its overall performance. IPC measures the number of instructions a processor can execute in a single cycle. At the same time, latency refers to the time for a single instruction to be executed from start to finish. A processor with high IPC may be able to execute multiple instructions in a single cycle. However, if the latency for each instruction is high, the processor's overall performance may still be impacted.

Hardware Description Language (HDL) design simulation can provide more accurate test results than software simulation since it models the hardware at a lower level of abstraction. The HDL design simulation will simulate the behavior of the processor's gates and registers. In contrast, the software simulation will simulate the processor's behavior at a higher level of abstraction, leading to differences in timing behavior that may be absent in the software.

The optimized CV32E40X core will be evaluated using HDL design simulation exclusively. By simulating the processor's behavior at a lower level of abstraction, a more precise understanding of its performance characteristics can be gained and identify any discrepancies that may arise.

1.3 Structure of the Thesis

The thesis is structured as follows: Chapters 2, 3 and 4 gives comprehensive background with what is needed to understand the design, discussion and motivation to the work done in the thesis. This includes past trends in processor architectures, some description of state-of-the-art architectures, a deep dive into RISC-V and the structure of the CV32E40X core, and lastly a high-level overview of the proposed architecture. In Chapter 5 the design and implementation are presented. The explanation of the complete design is split into four stages: ID stage, EX stage, LSU stage, and WB stage. Afterwards, an overview of the experimental setup and how the evaluation is done is given in Chapter 6. What test programs are run, what kind of tools are utilized and general information on the simulation is presented in this chapter. Lastly, in Chapter 7, the modified CV32E40X is compared and evaluated against the baseline CV32E40X.

Chapter 2

Trends in Processor Architecture

Throughout their history, processors have undergone tremendous evolution, with the introduction of the microprocessor being a key milestone. Intel's first microprocessor, the Intel 4004, introduced in 1971, contained about 2300 transistors, had a clock speed of 740kHz, and could deliver 92,000 instructions per second while dissipating approximately 0.5 W [13; 14]. Since then, a new microprocessor has been launched practically every year, each delivering significant performance improvements over its predecessors. Studies have estimated that this growth has been exponential, around 50% per year [15], resulting in a cumulative growth of over three orders of magnitude in just two decades. These improvements have been driven by manufacturing process advancements and processor architecture innovations [16; 17].

Some of the work in this section is based on the work done in the specialization project. More specifically the figure of the Out-of-Order (OoO) core and some of the research done on the state-of-the-art architectures.

2.1 Scaling of Transistor Performance

In 1974, Robert Dennard, a researcher at IBM, made a significant observation that reducing the size of a transistor could decrease its switching voltage, increase its switching frequency, and have a constant power consumption per square millimeter [18]. This breakthrough became known as Dennard Scaling. With the scaling of transistors, it was possible to lower their operating voltage, raise their operating frequency, and maintain the same power consumption.

More transistors in a chip can be used to increase the processor throughput. In theory, if the number of transistors in a chip is doubled, so would the capability of performing the number of functions simultaneously. Storage would also increase by a factor of two. However, these performance gains are significantly lower in practice due to mainly two reasons. Firstly, the internal micro-architecture of a processor. The performance of key components such as cache memory and issue logic scales non-linearly with area [19]—secondly, wire delay and the general impact of wires. Scaling down transistors by, for example, a factor of two does not result in twice the number of transistors per unit area. The reasoning is that more logic often requires an increase in the number of wires, increasing the area that must be devoted to them. Also, wire delay scales poorly compared to transistor delay. Therefore, as transistors continue to scale down, the impact of wire delays becomes more noticeable and severe. As a result, a large part of the activity performed by current microprocessors is spent on moving data around [14].

Bringing everything into coherence, the increase in transistors density opened up opportunities for architects to add more functionality to processors and set the stage for new micro-architectural innovations. The rapid development in processor technology has enabled multiple design innovations in computer architecture. Computer architects could optimize for Instruction Level Parallelism (ILP) without being overly concerned about area and power [20]. This development resulted in a total performance improvement rate of 52% per year from 1986-2003 [15], despite some of the challenges outlined above. Some noteworthy innovations that followed include deeper pipelines, larger instruction windows, OoO execution, new ISA features, more accurate branch predictors, and superscalar architectures.

2.1.1 Branch Prediction

Branch prediction has been an important microarchitecture technique since the early processors. Any time branch instructions are fetched from memory; the preceding instructions depend on the branch outcome. The branch outcome is unavailable until the branch is executed, which takes a couple of cycles. Therefore, stalling the processor from fetching more instructions until the outcome of the branch is known will cause a significant penalty, specifically because branch instructions are very common in many applications. Using a branch predictor to predict the outcome of the branch and speculatively executing instructions of the predicted path can reduce the penalty given by stalling the fetch unit, given that the predictor is highly accurate. The use of branch prediction goes back as early as the late 1950s in the IBM Stretch computer [21].

In addition to better performance, branch predictors can help reduce power consumption. Reducing power consumption with branch prediction is achieved by reducing the amount of instruction fetches from memory. The amount of data fetched from memory is reduced by speculatively fetching instructions before the branch outcome is known. Secondly, as mentioned earlier, accurately predicting branch outcomes can prevent the processors from needing to stall, which can waste energy as the processor is idle during the stall.

Despite these advantages, branch predictors also have some disadvantages. Including the added complexity of adding a branch predictor, they may also be less effective for certain types of code. Code with irregular branch patterns can be challenging to predict, resulting in an overhead that outweighs the benefits. However, specific branch predictor techniques are designed to handle irregular branch patterns more effectively, emphasizing that application is important when choosing which predictor to implement.

2.1.2 Simplified Instruction Set Architecture

The ISA is an important aspect of processor architecture that has undergone significant changes over time. New instructions have been added over the years to support better common code structures found in mainstream applications, which builds on one of the eight great ideas, as Dr. David A. Patterson stated, ”make the common case fast.” However, a significant shift in ISA design occurred in the early 1980s as a result of independent projects at UC Berkeley [22], IBM [23], and Stanford University [24]. This movement was referred to at RISC and was coined by the UC Berkeley team, a term that others adopted. Before this change, the prevailing ISA design trend was to increase complexity. It was thought that computers could be made more efficient by reducing the semantic gap between high-level programming languages and machine languages. However, these projects demonstrated that a simpler ISA also offered significant benefits, such as a lower design cost and processor verification. This success was partly due to the maturity of compiler technology and the advantages of simpler circuitry in terms of delay, cost, and energy consumption.

2.1.3 In-order and Out-of-Order Execution

The initial microprocessors, such as the Intel 4004 mentioned earlier, were designed as in-order processors [25]. An in-order core microarchitecture typically comprises an instruction fetch unit that retrieves instructions from an instruction cache and stores them in an instruction queue. An instruction decode unit that decodes the instructions and prepares them for execution. The execution of instructions takes place in the back-end of the microarchitecture, which usually includes execution units responsible for carrying out the operations specified by the instructions. Once executed, the instructions are committed, updating the architectural state. Figure 2.1 illustrates a block diagram depicting an in-order core. The instruction queue and issue stage are often discussed as part of the decode stage, depending on the specific type of in-order core. The issue stage represents the point the decision is made regarding the next instruction sent for execution. Additionally, the next part of figure 2.1 is the location of functional units, which includes **ALUs!** (**ALUs!**) and a Load-Store Unit (LSU), commonly referred to as the EX stage in some diagrams.

In-order cores offer advantages in terms of efficiency since they do not require complex instruction reordering or other optimization techniques employed by OoO cores. In-order cores also possess a simpler microarchitecture, leading to lower power consumption and reduced manufacturing costs. However, their execution model based on the program’s order can significantly limit performance, particularly in complex programs involving dependencies and data hazards.

The technique of OoO execution, also known as dynamic scheduling, gained widespread usage during the 1990s [14]. This technique enables the hardware to execute instructions in a different order than their appearance in the binary code while preserving the program’s semantics. Its primary objective is enhancing ILP through instruction reordering. For instance, OoO processors can execute younger instructions independent of the missing load when encountering a cache miss instead of stalling all subsequent instructions as in-order processors do. Figure 2.2 demonstrates that in-order is maintained until the EX stage. Execution happens OoO before retiring in-order again.

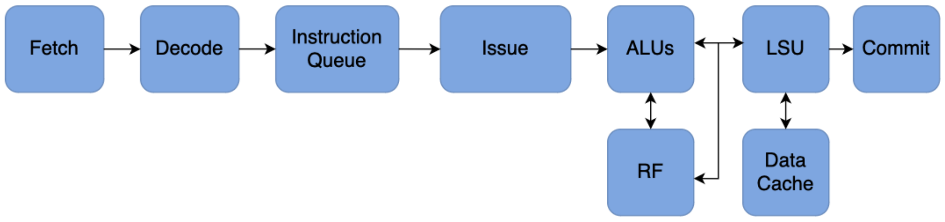


Figure 2.1: A block diagram of how an in-order core is structured

While OoO execution provides significant performance benefits, it comes at a cost. The issue logic, including that of memory instructions, becomes much more complex. Memory dependencies are more challenging to check than register dependencies because register dependencies can be identified at decode time in program order. In contrast, memory dependencies require identification later in the pipeline's back end once the effective addresses of loads and stores are computed out of program order. Despite this cost, out-of-order execution has become popular and is used by most current microprocessors.

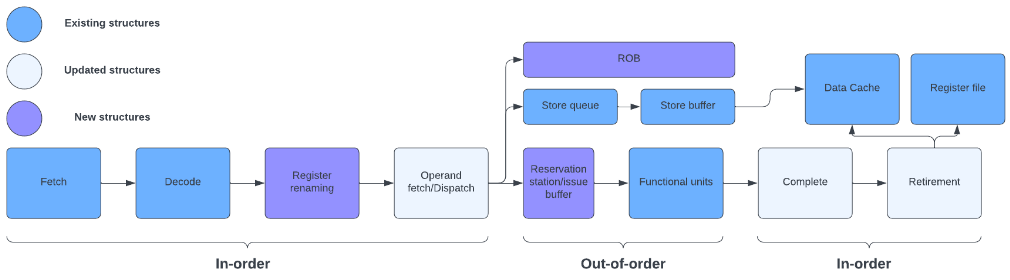


Figure 2.2: Block diagram depicting how an Out-of-Order core is structured

2.1.4 Superscalar

Another aforementioned key microarchitecture technique that became common use during the 1990s was superscalar. It was a new microarchitecture organization to exploit further ILP. A superscalar processor is a type of processor that can process multiple instructions simultaneously in each of its pipeline stages. It can perform various operations like fetching, decoding, renaming, issuing, executing, and committing multiple instructions simultaneously. In figure 2.3 a simplified processor with two execution units is shown. If there are two instructions in the instruction queue with no hazards and not requiring the same execution unit, the processor can dispatch both instructions in the same clock cycle. Figure 2.3 visually represents two instructions retrieved from the instruction queue and executed in parallel. The number of instructions that can be processed in parallel at each pipeline stage is known as the width of the superscalar processor. However, in practical terms, not all pipeline stages may have the same width, so the width of the processor is determined by the minimum width of its pipeline stages. A superscalar processor that has a width of N has the potential to process N instructions per cycle, thereby achieving a performance boost [26]. A superscalar processor requires replicating hardware resources to enable such parallel processing, usually by the same factor as its width. However, the hardware cost may grow non-linearly with the width of some processor parts. For example, the bypass logic incurs a superlinear cost, which grows quadratically with the number of functional units [14].

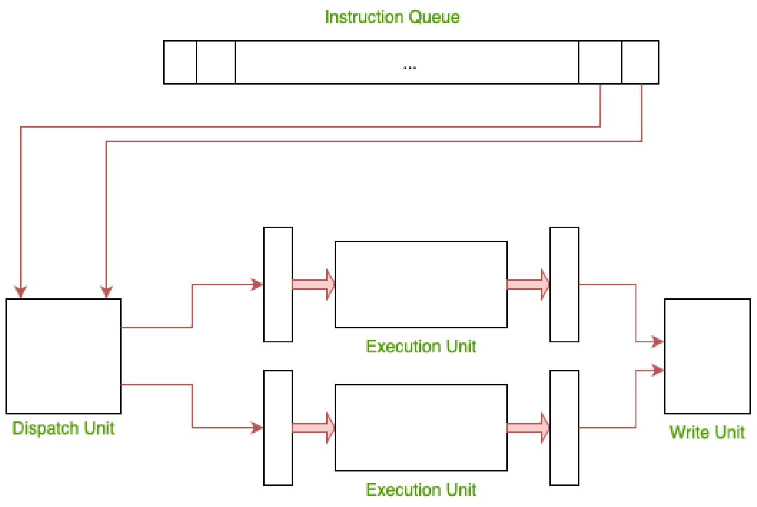


Figure 2.3: Processor with two execution units

2.2 State-of-the-Art

As transistor sizes have continued to shrink, the effectiveness of Dennard scaling has diminished. This is because as transistors approach the atomic scale, they begin to exhibit quantum mechanical effects [27] that make it difficult to predict their behavior accurately. As a result, it has become increasingly challenging to maintain the constant power density required by Dennard scaling. Furthermore, as the power density increases, so does the amount of heat generated by the Integrated Circuit (IC), which can cause reliability issues and limit performance. This has led to a phenomenon known as the ”power wall” [28], where the power consumption of ICs has reached a point where it is no longer feasible to increase clock speeds or add more transistors without excessive power consumption and heat dissipation. In the early 2000s, the power growth in microprocessors resulted unsustainable, and the trend changed towards flat or decreasing power budgets. The end of Dennard scaling has led to a shift in micro-architecture design trends from simple transistor scaling to a more holistic approach that takes into account the entire system and the specific requirements of different workloads [29; 14]. Finding an implementation that considers both power and area alongside performance would therefor be the ideal solution for the CV32E40X core, as it is intended for embedded applications.

A relative new micro-architecture, slice-Out-of-Order (sOoO), represents a novel approach that builds upon the energy-efficient in-order stall-on-use cores, as opposed to the more conventional out-of-order (OoO) cores [30]. The sOoO design aims to overcome the limitations of in-order cores by enabling load and store instructions, along with their corresponding backward slices, to bypass arithmetic instructions in the instruction stream. This, in turn, leads to an increase in both Memory Level Parallelism (MLP) and ILP (instruction-level parallelism). The slices of the load and store instructions refer to the address-generating instructions that precede a memory operation, commonly known as address-generating instructions (AGIs). In essence, sOoO cores can be thought of as OoO machines with certain restrictions. Notable works that have proposed sOoO cores include LSC [30], Freeway [31], and Forward Slice Core (FSC) [32].

As transistor sizes have continued to shrink, the effectiveness of Dennard scaling has diminished. The reasoning is that as transistors approach the atomic scale, they exhibit quantum mechanical effects [27], making it difficult to predict their behavior accurately. As a result, it has become increasingly challenging to maintain the constant power density required by Dennard scaling. Furthermore, as the power density increases, so does the amount of heat generated by the IC, which can cause reliability issues and limit performance. The phenomenon is known as the ”power wall” [28], where the power consumption of ICs has reached a point where it is no longer feasible to increase clock speeds or add more transistors without excessive power consumption and heat dissipation. In the early 2000s, the power growth in microprocessors resulted un- sustainable, and the trend changed towards flat or decreasing power budgets. The end of Dennard scaling has led to a shift in micro-architecture design trends from simple transistor scaling to a more holistic approach that considers the entire system and the specific requirements of different workloads [29; 14]. Finding an implementation that considers both power and area alongside performance would be the ideal solution for the CV32E40X core, as it is intended for embedded applications.

A relatively new micro-architecture, sOoO, represents a novel approach that builds upon the energy-efficient in-order stall-on-use cores, as opposed to the more conventional OoO cores [30]. The sOoO design aims to overcome the limitations of in-order cores by enabling load and store instructions, along with their corresponding backward slices, to bypass arithmetic instructions in the instruction stream. The bypass scheme, in turn, leads to an increase in both MLP and ILP. The slices of the load and store instructions refer to the address-generating instructions that precede a memory operation, commonly known as AGIs. In essence, sOoO cores can be considered OoO machines with certain restrictions. Notable works that have proposed sOoO cores include LSC [30], Freeway [31], and Forward Slice Core (FSC) [32].

DELEN UNDER ER FRA FORDYPNINGSPROSJEKTET UTEN OM SISTE DELKAPITTEL

2.2.1 Load Slice Core

The LSC architecture incorporates two in-order instruction queues to facilitate its operation [30]. These queues include the bypass queue (B-IQ), specifically designed for memory instructions and AGIs, and the main queue (A-IQ), responsible for handling the primary instruction flow. Figure 2.4 provides a visual representation of the LSC microarchitecture, elucidating how it extends the functionalities of the underlying in-order core. Three circles in the figure's upper-left corner correspond to a particular structure's status: new, pre-existing, or updated from the in-order core. The bypass queue plays a crucial role by enabling memory instructions and AGIs to be dispatched for early execution. Instructions in the bypass queue can execute ahead of those in the main queue, thus leveraging the benefits of OoO execution. LSC employs the register dependency table (RDT) and the Instruction Slice Table (IST) to identify AGIs. RDT and IST allow these instructions to proceed with OoO execution alongside the remaining instructions by utilizing the bypass queue. LSC restricts the extent of OoO execution by selectively choosing between the heads of the two in-order instruction queues. This approach minimizes energy requirements while enhancing performance by exploiting MLP. Experimental results conducted by Carlson et al. indicate that LSC achieves a 53% performance improvement over a baseline in-order processor.

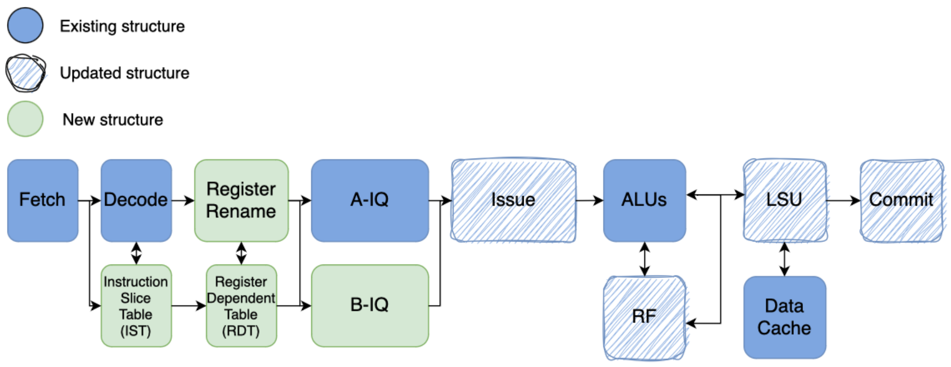


Figure 2.4: Block diagram showcasing the structure of Load Slice Core and how it builds on the in-order core

2.2.2 Freeway Core

LSC enhances MLP by separating load-dependent instructions into a separate queue. However, this approach still serializes the execution of load instructions within that queue, particularly in cases where a load depends on another load. Consequently, when the B-queue experiences a stall due to a load dependency, younger independent loads behind the dependent load also experience the same stall. This limitation hampers the ability to exploit MLP fully. To address this issue, Kumar et al. proposed Freeway [31], which mitigates LSC's MLP constraints caused by load-dependent loads.

Figure 2.5 provides an overview of Freeway's core architecture, wherein the orange-colored blocks represent new structures, and the striped blocks denote updated structures compared to LSC. Freeway resolves the problem by differentiating between two slices within a splice containing loads. These two slices are called producer slices and dependent slices. Freeway separates these two types and directs the dependent slices to a new yielding or Y-queue queue. The Y-queue is positioned beneath the A-queue and B-queue, as illustrated in Figure 2.5. By moving the dependent slices to the Y-queue, the independent slices in the B-queue can be issued for execution, thereby enabling a greater degree of MLP than LSC.

Evaluation conducted by Kumar et al. demonstrates that Freeway surpasses LSC's performance by 12% and achieves MLP levels within 7% of the maximum limits offered by complete OoO execution.

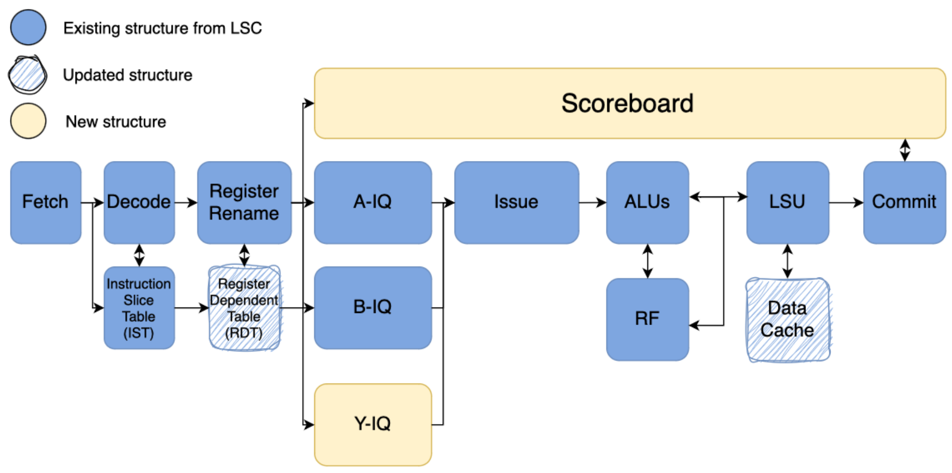


Figure 2.5: Microarchitecture of the Freeway core

2.2.3 Forward Slice Core

FSC is a recent proposal in sOoO cores. The research paper by Lakshminarasimhan et al. [32] introduces FSC. It identifies four limitations in the previous sOoO techniques mentioned earlier, namely limited ILP, limited MLP, IST misses, and Hardware Complexity. FSC addresses these shortcomings by offering distinct features.

There are three key areas where FSC diverges from LSC and Freeway. Unlike LSC and Freeway, which focus on backward slices encompassing both load and store instructions, FSC specifically targets forward slices associated with load instructions. Forward slices consist of instructions dependent on a load that has not finished executing. FSC employs four separate in-order First-In, first in, first out (FIFO) instruction queues: Main Lane (ML), Dependent Load Lane (DLL), Dependent Execute Lane (DEL), and Holding Lane (HL). Figure 2.6 provides a comprehensive overview of the architecture, highlighting the locations of these queues.

Instructions not belonging to a forward slice, including load instructions not part of the forward slice, are transferred to the ML queue. Forward slices are divided into two separate queues: loads are directed to the DLL queue, while non-load instructions are sent to the DEL queue. Instructions that encounter a wait exceeding a predetermined number of cycles at the head of the DEL queue are subsequently moved to the HL queue. This design allows independent instructions to be issued for execution when a stalled instruction, such as one involving prolonged memory access, occurs. The presence of the HL queue enables the extraction of ILP and MLP opportunities that were not feasible with the aforementioned sOoO techniques.

In comparison, using Freeway as a more specific example, dependent slices serialize in the Y-queue, resulting in the potential hindrance of younger independent slices from executing when a stalled dependent slice resides in the Y-queue. Experimental results presented by Lakshminarasimhan et al. illustrate that FSC achieves an average performance improvement of 9.7% compared to Freeway while maintaining a similar power budget.

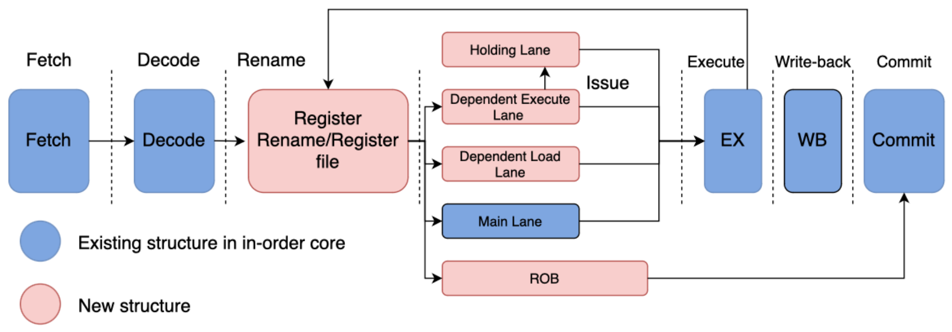


Figure 2.6: Forward Slice Core architecture

2.2.4 Comparing Architectures in Processor Metrics

Table 2.2 overviews how the architectural cores discussed in this chapter perform overall in processor metrics to a baseline in-order stall-on-use core. The data is retrieved from the papers written by Carlson et al. [30], Kumar et al. [31], and Lakshminarasimhan et al. [32]. The measurements are done through the SPEC CPU2006 and SPEC CPU2017 benchmark suites. SPEC CPU2006 benchmark is used on the LSC and freeway core, while the SPEC CPU2017 benchmark is used on the FSC. There is some deviation in data presented between the three papers. Different use of benchmark suites could be one explanation. Another explanation could be that different OoO processors were used for comparison, which is seen by the drastically different area and power overhead compared to the in-order processor. Since the same in-order processor is used between the three papers, this would make sense and explain the deviation in data. Table 2.1 shows the data presented in the paper by Lakshminarasimhan et al. Despite some deviation in the data, it still gives a good overview of how the different cores perform compared to each other. Based on this data, the core strengths of sOoO cores are brought forth, achieving excellent performance gains compared to the in-order core without the enormous area and power overhead that comes with OoO cores. It highlights why it is important to continue studying such implementations and experiment with these architectures on processors used in real applications, as they strike a good balance between area, power, and performance. Looking at more simplified versions of these architectures would be interesting for low-power embedded processors. Also, these results presented from the three papers are achieved from software simulations. Investigating if comparable results are achievable by implementing the architectures on real processor cores is interesting.

Table 2.1: How different architectures perform overall in processor metrics in respect to an baseline in-order stall-on-use core

Architecture	**Performance**	**Area Overhead**	**Power Overhead**
OoO	93%	154%	1250%
LSC	53%	15%	22%
Freeway	60%	16.5%	24%
FSC	64%	1.01%	1%

Table 2.2: Data on performance metrics presented in the paper by Lakshminarasimhan et al.

Architecture	**Performance**	**Area Overhead**	**Power Overhead**
OoO	75%	38%	132%
LSC	44%	15%	22%
Freeway	50%	16.5%	24%
FSC	64%	1.01%	1%

Chapter 3

RISC-V

RISC-V is an open-source ISA that has been in development since 2010 [33]. The establishment of the RISC-V foundation has led to a surge in popularity, with a growing number of developers and companies adopting the technology and actively contributing to its development. This collaborative effort has been instrumental in enhancing the maturity of the RISC-V ecosystem. This section will explore the RISC-V architecture, its features, and the core-v family. Finally, the section will examine the inner workings of the CV32E40X core and explore some of its potential performance limitations.

3.1 RISC-V Architecture

RISC-V is a RISC architecture [34]. RISC architecture has a small number of simple and general-purpose instructions that are easy to execute quickly [35]. In addition, the base instruction set includes only 47 instructions, which is significantly fewer than many other popular architectures, such as ARM or x86 [34; 36; 37; 38]. This simplicity can make the architecture easier to understand and implement and reduce the overhead associated with instruction decoding and execution. More specifically, RISC-V is a load/store architecture, meaning all data must be loaded from memory into registers before it can be operated. The result of the operation must then be stored back in memory [39].

RISC-V's modularity is one of its primary strengths, as it allows for developing specialized cores without an excess of unnecessary instructions. To create a RISC-V core, one can start with one of the five base instruction sets that contain the minimum number of instructions needed for a functioning core. From there, custom instructions can be added, or one can use one of the 21 official RISC-V Instruction Set Extensions (ISEs) [34]. Some of these ISEs are still in development, and Table 3.1 lists several standard base and extension instruction sets.

In addition to its modularity and simplicity, the RISC-V architecture is designed to be highly extensible [40]. The ability for an extensible ISA means that it can be customized and extended to meet the specific needs of different applications and systems. For example,

Table 3.1: Examples of some of the standard RISC-V base and extension instruction sets

Name	**Description**
Base ISA	
RV32I	32-bit instruction set
RV32E	32-bit with 16 registers instruction set
RV64I	64-bit instruction set
RV128I	128-bit instruction set
Standard Extension	
P	Packed-SIMD Instructions
V	Vector Operations
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
Q	Quad-Precision Floating-Point

specialized extensions can be added to support real-time processing, machine learning, or other applications.

Overall, the RISC-V architecture's modularity, simplicity, extensibility, and openness make it a powerful and versatile platform for various applications, from embedded systems to high-performance computing.

3.2 Core-V Family

As of writing this, there are 111 open-source cores available online on the official RISC-V website [41], with more under development. Among these cores are those belonging to the Core-V family [11], which the OpenHW group has developed. The Core-V family comprises seven cores designed for various applications, including Internet-of-Things (IoT) and Unix Operating systems.

One significant contributor to the RISC-V project is Silicon Labs, particularly in the CVE4 subgroup[12]. This subgroup consists of small and efficient cores specifically designed for IoT applications. The CVE4 cores began as a continuation of the RI5CY core from PULP but have since expanded with two additional cores: CV32E40S [42] for more secure applications and CV32E40X [43] for more compute-intensive applications.

3.2.1 CV32E40X Core Overview

The CV32E40X is a 32-bit, in-order RISC-V core with a 4-stage pipeline, and it is open-source. The pipeline stages include Instruction Fetch IF, Instruction Decode (ID), Execute (EX), and Write-Back (WB) [43], as shown in Figure 3.1.

In the IF stage, instructions are fetched from memory, and a prefetcher is implemented to continue fetching instructions even when the preceding stages are halted. Figure 3.2

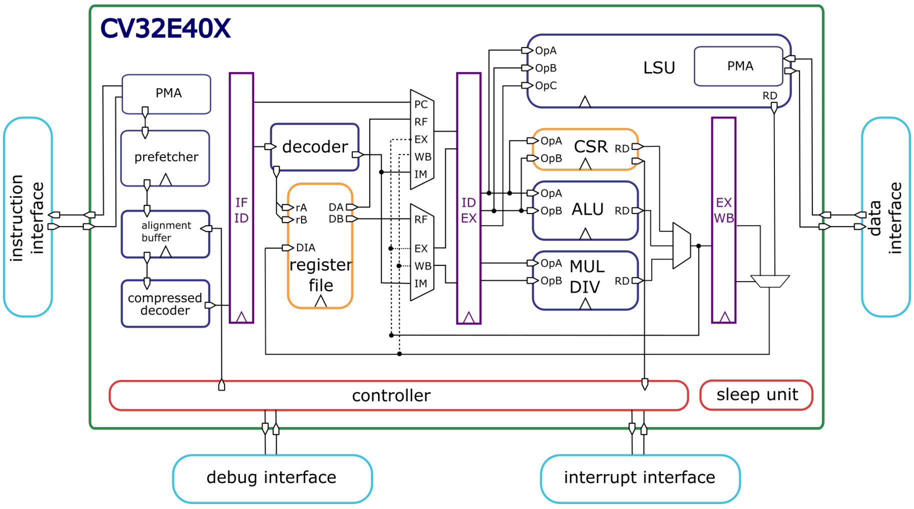


Figure 3.1: Block diagram of the CV32E40X core

gives an overview of the IF stage. The figure showcases the instruction interface where the instruction arrives from memory. Afterward, the instructions are processed and sent to the first pipeline register, where they eventually move to the second pipeline stage of the processor. The IF stage can supply one instruction per cycle to the ID stage, as the aligning buffer can only fetch one instruction per cycle.

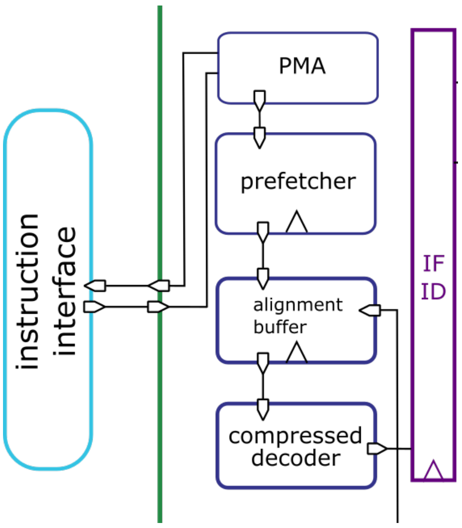


Figure 3.2: Block diagram of the IF stage in the CV32E40X core

The ID stage decodes the fetched instructions, sets the control signals, and acquires data from the instruction and register file to determine the instructions. As is visualized in Figure 3.3, the instructions are sent first sent to the decoder. The decoder interprets the instructions and sets the correct control signals. When this is done, the data in the registers, specified by the instruction, in the register file is read. In Figure 3.3, one can see two inputs for reading registers, labeled **rA** and **rB**. There are also two outputs for the data. The control signals ensure the correct data is outputted from the two MUXes. The data is then sent to the next pipeline register for the specified operation in the EX stage. The RISC-V instruction set manual provides details on how each instruction is structured.

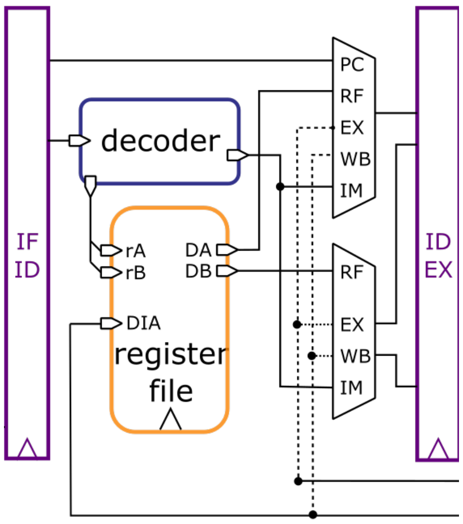


Figure 3.3: Block diagram of the ID stage in the CV32E40X core

The EX stage contains three functional units: a LSU, an Arithmatic Logic Unit (ALU), and a Multiplication/Division (MUL/DIV) unit. Figure 3.4 gives a visualization of the functional units. Dependencies are handled by forwarding the result from EX and WB directly to the next instruction. This work mainly focuses on the transition from the decode to execution stage, which involves passing through the second pipeline register. If the condition are met, branches will be taken from the EX stage. Lastly, for multi-cycle instructions the EX stage will be stalled until they are complete.

The EX stage contains three functional units: an LSU, an ALU, and a MUL/DIV unit. Figure 3.4 gives a visualization of the functional units. Dependencies are handled by forwarding the result from EX and WB directly to the next instruction. This work mainly focuses on transitioning from the decode to the execution stage, which involves passing through the second pipeline register. If the condition is met, branches will be taken from the EX stage. Lastly, the EX stage will be stalled for multi-cycle instructions until they are complete.

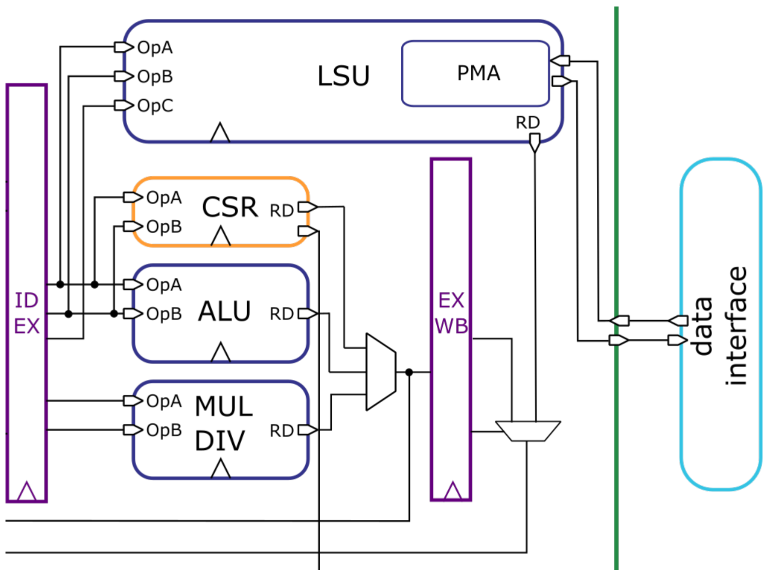


Figure 3.4: Block diagram of the EX and WB stage in the CV32E40X core

Finally, the WB stage. In this stage, the result of ALU, Multiplier, Divider, or Load instructions is written back to the register file.

From Figure 3.1, a controller can be seen beneath the core. Each of the pipeline stages is connected to a controller block. The controller is responsible for handling in- interrupts and exceptions. It also sets the right Program Counter (PC) value to be fetched when it is known if a branch is taken in the EX stage. Another responsibility that the controller has is data handling for performance counters, also called Hardware Performance Counters (HPC). These are used to conduct low-level performance analysis or tuning.

3.2.2 Pipeline Details

Looking at Figure 3.5 gives another detailed overview over the pipeline of the CV32E40X core. It shows how the instruction flow works in the CV32E40X core. Each stage contains a *ready* and *valid* signal. The *valid* signal indicates that the stage has valid data for next stage. The *ready* signal on the other hand, signalizes that the stage is ready for new data. When the *ready* signal is high from the stage is ahead and the *valid* signal is high in the current stage, the data is moved through the pipeline register and onto the next pipeline stage. Looking at the AND gates in Figure 3.5 gives information on the conditions that needs to be met for the *ready* and *valid* signals to go high.

Looking at Figure 3.5 gives another detailed overview of the pipeline of the CV32E40X core. It shows how the instruction flow works in the CV32E40X core. Each stage contains a *ready* and *valid* signal. The *valid* signal indicates that the stage has *valid* data for the next stage. On the other hand, the *ready* signal signalizes that the stage is ready for new data. When the *ready* signal is high from the stage ahead, and the *valid* signal is high in the current stage, the data is moved through the pipeline register and onto the next pipeline stage. Looking at the AND gates in Figure 3.5 gives information on the conditions that must be met for the ready and valid signals to go high.

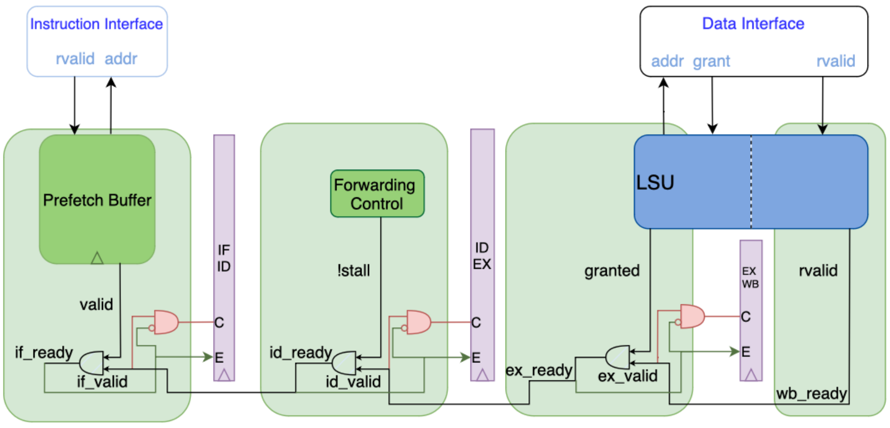


Figure 3.5: CV32E40X Pipeline

Multi- and Single-Cycle Instructions

Table 3.2 gives an overview over the cycle count per instruction type. Not every instruction has a fixed cycle count. Some of the instructions have a variable time, indicated as a range, e.g. 3-35 means that the instruction takes a minimum of 3 cycles and a maximum of 35 cycles. The cycle count of these instructions depends on certain types of conditions. For example, load instructions have normally an cycle count of 1. however, for non-word aligned word transfer the cycle count is 2. The number shown in 3.2 for the different instruction types are based on the assumption that there is zero stall on the instruction-side interface and zero stall on the data-side memory interface.

Table 3.2 gives an overview of the cycle count per instruction type. Not every instruction has a fixed cycle count. Some instructions have a variable time, indicated as a range, e.g., 3-35 means that the instruction takes a minimum of 3 cycles and a maximum of 35 cycles. The cycle count of these instructions depends on certain types of conditions. For example, load instructions usually have a cycle count of 1. However, for non-word-aligned word transfer, the cycle count is 2. The cycle count can also vary depending on memory access time. The number shown in Table 3.2 for the different instruction types is based on the assumption that there is zero stalls on the instruction-side interface and zero stalls on the data-side memory interface.

Hazards can also affect the cycle count. The CV32E40X core experiences two cycle penalties due to hazards [44]. For load and jump register data hazards, there is a 1-cycle penalty. Load data hazards are when an instruction immediately following a load uses the result of the load. In that case, the core must wait for the data to be retrieved from memory in the WB stage. Jump register data hazard occurs when a *jalr* instruction depends on the result of an immediately preceding non-load instruction. The other penalty is two cycles.

Table 3.2: Cycles per instruction type

Instruction Type	**Cycles**
Integer Computational	1
CSR Access	1 or 4 depending on CSR
Load/Store	1 2(non-word aligned word transfer and half-word transfer crossing word boundary)

Multiplication	1(mul)
	4(mulh, mulhsu, mulhu)

Division Remainder	3-35
Jump	2 3(target is a non-word aligned non-RVC instruction)
mret	2 3(target is a non-word aligned non-RVC instruction)
Branch(Not-Taken)	1
Branch(Taken)	3 4(target is a non-word aligned non-RVC instruction)
Instruction Fence	5 6(target is a non-word aligned non-RVC instruction)

The two-cycle penalty can also occur for jump register data hazards. However, only when a *jalr* depends on the result of an immediately preceding load instruction.

3.2.3 Load Store Unit

As the name may suggest, the LSU of the core handles access to the data memory. Table 3.3 lists all the signals used in the interface between the LSU and memory and describes their use case. Look at the Core-V CV32E40X user manual [45].

The data bus interface is Open Bus Interface (OBI) protocol compliant. It is how the LSU communicates with memory. The LSU, as depicted in Figure 3.1 and Figure 3.5, works over both the EX and WB stages. In the EX stage, the LSU provides a valid address contained within the *data_addr_o[31:0]*. Alongside the address, the control information on *data_be_o[3:0]* and *data_we_o*, including data on *data_wdata_o[31:0]*, is also provided. After this is provided, *data_req_o* is set high. *data_gnt_i* is then set high as soon as the memory is ready to serve the request, where the execution in the LSU transitions from the EX stage to the WB stage. Following a granted request, the memory answers with *data_rvalid_i* set high if *data_rdata_i[31:0]* contains valid data. *data_rvalid_i* is also set high for store instructions. However, the data placed in *data_rdata_i[31:0]* has no meaning. The data placed in *data_rdata_i[31:0]* is finally written back to the register file from the WB stage for load instructions.

Table 3.3: LSU interface signals

Signal	Direction	Description
data_req_o	output	Request to memory is valid
data_gnt_i	input	Memory request accepted
data_addr_o[31:0]	output	Requested address from memory
data_atop_o[5:0]	output	Atomic attributes
data_be_o[3:0]	output	Byte Enable. Set for which bytes to read/write
data_memtype_o[1:0]	output	Memory Type attributes
data_data_prot_o	output	Protection attributes
data_dbg_o	output	Debug mode access
data_wdata_o[31:0]	output	Data to be written to memory
data_we_o	output	Write Enable
data_rvalid_i	input	Signals end of memory transaction
data_rdata_i[31:0]	input	Data read from memory
data_err_i	input	An error occurred
data_exokay_i	input	Exclusive transaction status

Figure 3.6 shows a timing diagram of a primary memory transaction done by the LSU in the CV32E40X core. Similar to what was explained in the previous paragraph, the timing diagram shows that the transaction starts with *data_req_o* set to high, with additional information like the address to be written to or read from and whether it is a load or a store instruction. At the end of the transaction, *data_rvalid_i* is set to high alongside the data read if the instruction was a load.

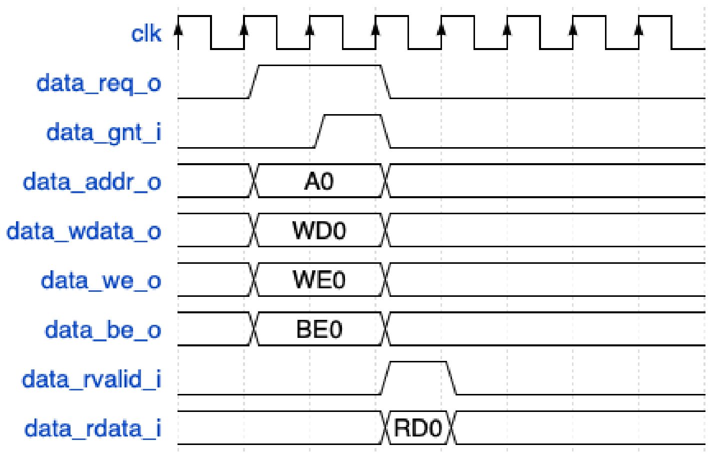


Figure 3.6: Wave diagram of a basic memory transaction in CV32E40X

3.2.4 Performance Limitations

While the CV32E40X core is designed to handle compute-intensive applications, like any processor core, it has some potential performance limitations. One of these potential performance limitations could be due to CV32E40X being an in-order stall-on-use processor core. A stall-on-use core is a type of processor core design where the processor waits for an instruction's result before executing the next instruction. This type of processor core is also known as a load/store architecture.

An in-order stall-on-use processor core can experience significant performance limitations due to data dependencies, which may result in stalls. Stalls occur when the core waits for the necessary data to become available before proceeding with the next instruction. This waiting time can significantly reduce performance, particularly for workloads with many data dependencies. The CV32E40X processor core exemplifies this limitation because it lacks an instruction queue. An *instruction queue* is a buffer that temporarily stores instructions awaiting execution. Without an instruction queue, a stall in the back-end of the pipeline will cause the entire processor core to stall. In contrast, with an instruction queue, the processor core can continue to fetch and decode instructions as long as the queue has sufficient space, despite a stall in the back-end of the pipeline.

Another limitation of an in-order stall-on-use processor core is that it may not fully utilize the available processing resources. In the case of the CV32E40X core, there are three functional units in the EX stage; an LSU, an ALU, and a MUL/DIV unit. If two instructions in the CV32E40X core are independent and can be executed in parallel, one would have to wait until the other completes before it can begin execution. This stall can lead to underutilization of the processor resources and reduced overall performance. Underutilization of a processor can hurt the overall system performance, particularly in systems where the processor is a bottleneck. In such cases, the processor may not be able to fully utilize its resources to complete tasks, leading to slower performance and increased latency. Moreover, underutilization can also lead to inefficient power consumption. In theory, the CV32E40X could execute up to three in-instructions in parallel, given that hardware supporting parallel execution is implemented. Memory accessing can be an excellent example of how underutilization of a processor core can limit performance. The processor can process several orders of magnitude more data than memory, so depending on where in memory the wanted data is located, it can take multiple cycles to retrieve, which would mean a stall for the processor, despite preceding instructions potentially not having any dependencies to the executing load instruction.

In addition, an in-order stall-on-use processor core may struggle with code with a high degree of branching or conditional execution. When the processor encounters a branch instruction, it may have to wait until the condition is evaluated before determining the correct path. This waiting time can be significant, primarily if the branch condition depends on data that is not available, leading to additional stalls. However, due to CV32E40X only having four pipeline stages, the latency related to branches is slightly mitigated. The reasoning behind this is that when a branch instruction is encountered, the processor does not know which instruction to fetch next until the branch condition is evaluated, which can cause a stall in the pipeline. In a deeper pipeline, the delay caused by this stall is more significant because more stages need to be flushed when a branch misprediction occurs.

Chapter 4

Proposed Architecture

This section will delve into the proposed architecture, representing a higher-level abstraction of the envisioned system design. Building upon the understanding of the system requirements and the challenges identified in the previous sections, this proposed architecture outlines a novel and innovative approach to address these challenges and achieve the desired objectives. Some theories used in the implementation will also be presented.

4.1 Key Insights

There are three key insights when implementing the LSC microarchitecture. The first insight is allowing load instructions and AGIs to bypass older instructions blocking the pipeline, waiting for memory. In return, exposing additional MLP leads potentially to performance levels close to full OoO scheduling [30]. The second insight bases itself on achieving simple instruction scheduling. Load instructions and AGIs execute OoO to the main instruction flow. However, to each other, the execution is happening in order. Therefore, the instruction scheduling can be implemented using two FIFO queues, simplifying the complex wake-up and selection logic in fully OoO core designs. The third and final insight is on the detection of AGIs. AGIs can be detected iteratively, going one backward step at a time, using the loop behavior in different applications [30].

4.2 Program Slices

A slice architecture, such as LSC, builds on the concept of program slices. Program slices are a chain of dependent instructions leading to or away from an event. This event is typically performance degrading, which can occur at some point in a program if certain conditions are fulfilled. An example of such an occurrence can be a branch misprediction.

The categorization by Zilles et al. [46] recognizes four sub-slices. Value sub-slice includes instructions that manipulate the operands used by the event instruction. The address sub-slice includes instructions for calculating memory addresses for the value sub-slice—existence sub-slice, which includes branch instructions that determine whether the event instruction will be executed. Lastly, the control flow sub-slice includes branch instructions that determine the path which leads to the event instruction.

AGIs are an important aspect of an LSC. These instructions can be defined as any instruction yet to execute for which a dependency chain exists from that instruction to the load address. In other words, these instructions produce the data that a future load or store instruction will require for address calculation. AGIs are a part of the address sub-slice.

4.3 Iterative Backwards Dependency Analysis

Several techniques for detecting program slices have been developed. One of these techniques is called Iterative Backwards Dependency Analysis (IBDA), a hardware technique for identifying data-dependent instructions. IBDA is implemented in the front-end of the processor, where it can inspect instructions as they are being dispatched to the back-end. It requires two hardware structures, which were shortly mentioned in section 2.2.1. The RDT contains the PC of the instruction that last wrote to a physical register for all physical registers. In other words, the RDT contains an entry of each physical register and maps it to the last instruction that wrote to this register. When an instruction identified as a part of a program slice enters the RDT, the PC of all the instructions that produced its source operands can be looked up. These instructions all belong to a backward slice. Source operands refer to the input values or data an instruction operates on. The previously mentioned PC can then be inserted into the IST, the second component of the IBDA[30].

The IST contains the addresses (PC) of the instructions identified as belonging to the backward slice by the RDT. Figure 2.4 in section 2.2.1 shows that the instructions are looked up in the decode stage, while the RDT is updated during the rename stage. The instructions in the IST are marked for bypass, which helps the LSC determine which instruction should go to which instruction queue. Instructions presented in the IST are inserted into the bypass queue, while the other instructions use the main queue. When the marked-for-bypass instructions enter the RDT, its source registers are looked up, and the producers of these source registers are recorded in the IST [30].

Consider the program in Listing 4.1 as an example. The goal is to track load slices. Instructions in the loop are labeled with a number, which will be used as a reference when explaining. The program consists of a loop centered around instruction (1), which reads data from memory. Instructions (2)-(6) are working on the data, writing it back to memory, and calculating the following store address. Instructions (7)-(8) calculate the address for the next load instruction and are, therefore, part of the load slice. At the start of the program, the IST is empty. In the first iteration of the loop, the IST undergoes an update by incorporating the *li t3*, *0x800500c* instruction, which serves as the producer of the initial load address. However, upon reaching the second iteration of the loop, specifically instruction (1), the RDT associates instruction (8) as the producer of the second load address. Consequently, instruction (8) is appended in the IST. Next, when instruction (8) is encountered within the same iteration, it already exists in the IST. Thus, upon inserting instruction (8) into the RDT, it is designated as a constituent of a program slice, prompting the retrieval of its sole dependency, a2, which is instruction (7). Instruction (7) is therefore included in the IST. In the following iteration, upon reaching instruction (7), its dependency, a1, is looked up alongside its producer, instruction (1). Thus, the entire program slice is obtained, storing instruction (1) in the IST.

```
init:
    li ft0, 50 ; Loop iterator
    li t2, 0x8000301c ; Intial load address
    li t3, 0x8005000c ; Initial store address
    j loop
loop:
(1) lw t1, 0(t2)     # Read data from memory
(2) mul t4, t1, t5   # Arithmetic operation on data
(3) add t6, t4, t1   # Perform addition
(4) div t7, t4, t6   # Perform division
(5) sw t6, 0(t3)     # Write result in memory
(6) addi t3, t3, 32 # Calculate next store address
(7) mul t2, t1, 99   # Calculate next load address
(8) addi t2, t2, 32 # Calculate next load address
(9) addi t0, t0, -1 # Decrement loop counter
(10)bnez t0, loop
```

Listing 4.1: Example RISC-V loop in assembly to showcase IBDA

4.4 Strategies for Pipelining Logic

Sequencing operations can be significant for timing and synchronization. Digital circuits operate based on timing constraints, where signals must arrive at specific times for correct operation. By sequencing operations, it is possible to ensure that different parts of the circuit execute in the correct order and that signals are synchronized appropriately. Sequencing operations can mitigate data corruption, race conditions, and timing violations.

A state machine is one solution to sequencing operations. However, it can be highly inefficient, as similar to Field Programmable Gate Arrays (FPGAs), logic can be created for every state at once also when designing processor cores. Only the correct answer will be selected at the end of each clock tick. One efficient solution to sequencing operations is pipelining[47]. However, a challenge with designing a digital logic pipeline is that the pipeline runs and still produces outputs even when the inputs to the pipeline are not valid. This section will discuss some strategies for handling the signaling associated with pipeline logic.

4.4.1 Pipelining with a Global Valid Signal

Using a global *valid* signal is the first strategy to be discussed in pipelining. The data going through the pipeline, as seen in Figure 4.1, is valid when the global *valid* signal is true. The basic rule of this pipelining method is first that there is a global valid line synchronous with the clock. New data is ready every time the *valid* line is set high. The second and final rule is that logic can only transition when the *valid* signal is true. One of the benefits of this strategy is that real pipeline logic is not required per

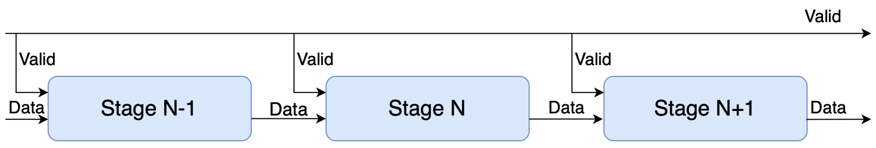


Figure 4.1: Pipelining with a global valid signal

4.4.2 Pipelining with a Propagating Valid Signal

There are two problems with the previously discussed strategy. One of the problems is that there is no way of knowing if an output sample is valid, and the second problem is that the whole pipeline operation depends on a uniform clock to create a *valid* signal. What happens when data is generated in bursts, and there is a need to determine the output's validity and timing? In such a case, another approach is needed.

This second approach propagates *valid* forward through each pipeline stage. Figure 4.2 shows a diagram of how this pipeline would look, with the data propagating alongside the *valid* signal through each stage in the pipeline. The basic rules to this approach go as follows: firstly, data associated with a valid signal that is true must also be *valid*. Secondly, a *valid* signal must be produced with the output data of that stage when it is nearing the end of the same stage. Thirdly, the *valid* signal must first be initialized to zero and set to zero whenever a reset occurs. When a reset occurs, the data should be ignored. Fourthly, only when the *valid* signal is high can something change. As a result, data is only referenced when *valid* is high. The fifth and final rule states that for a new value to be eligible for entry into the pipeline, all the logic components within the pipeline must be ready. In other words, the discussed pipeline architecture does not accommodate for stalls.

This approach proves effective when the pipeline is segmented into stages featuring only a single input *valid* signal and, similarly, when none rely on feedback from future results. In essence, if there is an absence of dependencies requiring waiting, this particular pipelining approach can work.

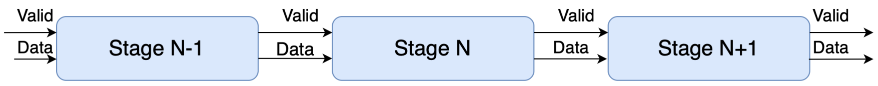


Figure 4.2: Pipelining with a traveling valid signal

4.4.3 Simple Handshake

As discussed with the previous approach, the main problem was that it could not handle any stalls. Consider some communication protocols. Previous discussed approaches to pipelining could fill the transmitter; however, what would the system do if the transmitter was busy? A valid-ready handshake approach can help solve this problem[48].

The handshake approach relies on two signals, one from the current device and one from the next in the pipeline. These two signals are called *Ready* and *Valid*. However, the naming can change depending on different interfaces. Figure 4.3 shows two modules communicating through handshake signaling. The basic rules of this approach go as follows:

- The first rule, a transaction takes place only if both the *Ready* and *Valid* signals are true.
- The second rule, the receiving pipeline must not set the Ready line high if it is not ready to receive data in the next clock cycle.
- The third rule, *Valid* signal should be set to true when data is ready to be sent. It should not wait for the *Ready* signal to be set to true before also being set to true. This helps avoid deadlocks by removing dependency between these two signals.
- Fourth rule, the *Ready* signals idle state should be in the ready condition, thus true.
- The fifth rule, the data that is being sent cannot change if the signal *Valid* is true.
- The sixth rule, data should be ignored in the ”do not care” state if the *Valid* is false.
- Seventh and final rule, *Valid* must be initialized to zero, while *Ready* must be initialized to true, which applies to reset or a clear pipeline operation. Data does not need to be set to anything since the data lines are placed in a ” do not care” condition due to Valid being set to zero.

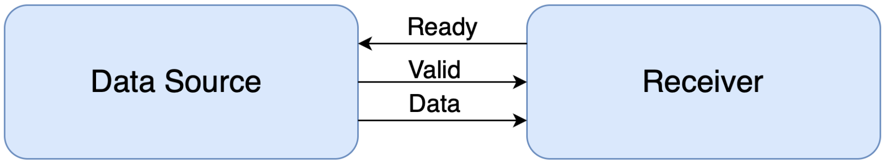


Figure 4.3: Diagram of a simple handshake

Figure 4.4 shows that a transaction gets carried out when both the valid and ready signals are true. Remember that transaction is not an actual signal. It just gives a visual presentation of a transaction with handshake. Similarly to the *Valid* line in the previous propagating *Valid* approach, the *Valid* && *Ready* combination serves as its analog. When the (Transaction) line is in a high state, it indicates the validity of the data (due to the high state of *Valid*), enabling the progression of processing by one additional step.

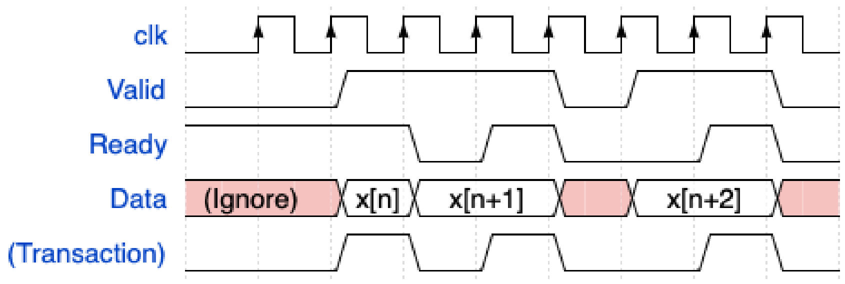


Figure 4.4: Signal flow for a simple handshake interface

4.5 Extending CV32E40X towards a Load Slice Core

In order to discuss the extension of the CV32E40X core into a load slice core, it is essential first to understand the characteristics of the CV32E40X core itself. As discussed in section 3.2.1 the CV32E40X core is an in-order core. In addition, the CV32E40X core does not have an instruction queue nor superscalar capabilities, meaning that a stall in the back-end of the pipeline would stall the entire processor. The LSC builds on superscalar in-order processors. Therefore, the first step in extending the CV32E40X core towards the LSC is to enable superscalar capabilities. This thesis will try a different spin on things and particularly focus on the execution of load/store instructions to the rest of the instructions. In other words, try to implement parallel execution between the LSU and the other functional unit, not between each. One needs to introduce additional functional units and pipeline stages to achieve superscalar capabilities. These new components enable parallel execution of load/store instructions along with other instructions, thereby increasing the overall throughput of the core and reducing the impact of memory latency. However, implementing such capabilities introduces new challenges that need to be addressed.

One of the challenges is maintaining program order. Superscalar processors aim to execute multiple instructions in parallel, which becomes challenging in an in-order core because instructions must be executed in program order. Dependencies between instructions, such as data dependencies, where the output of one instruction is used as an input for another, can create hazards that prevent parallel execution. Resolving these dependencies without violating the in-order execution constraint is a challenge. Another important consideration related to program order and instruction dependencies is the maintenance of precise exceptions. Precise exceptions ensure that exceptions are raised and handled deterministically and predictably. This means that if an exception is raised due to, for example, a page fault, the processor knows which instructions are affected by this error and need to be flushed and which instructions were not affected and are allowed to commit.

Another challenge that needs to be considered is control flow dependencies. Control flow instructions, such as branches and jumps, can create challenges in superscalar implementations. In an in-order core, the outcome of a branch is typically determined late in the pipeline, and subsequent instructions depend on that outcome. If multiple instructions are already in flight and dependent on the branch outcome, incorrect speculation or incorrect program order execution may occur. Handling control flow dependencies while preserving the in-order execution constraint requires careful design.

Figure 4.5 shows a modified version of the CV32E40X core architecture shown in Figure 3.1. It is a rough overview of the instruction flow. Firstly, there are now two pipeline registers between the ID and EX stages, shown in purple in Figure 4.5. Two pipelines will help make the LSU independent from the EX stage. In other words, the EX stage will be divided into two stages: the LSU and EX. All data required by the LSU will be stored in the LSU pipeline register, and all data required by the ALU and MUL/DIV will be stored in the ID/EX pipeline register. There is also a pipeline register implemented from the EX stage to the WB stage for the LSU, which depending if there is an active instruction inside, will determine which instruction will be allowed to commit, which is seen through the mux to the far right of Figure 4.5. In addition, a scoreboard is added for tracking instructions so that the oldest instruction is committed and program order is maintained. The idea of the modifications is to allow load/store instructions to execute in parallel with other instructions. A read memory access can sometimes take multiple cycles to execute, and being able to issue an add instruction in the meantime if there are no dependencies saves meaningful execution time.

By introducing superscalar capabilities for load/store instructions and the rest of the instructions, the foundation is laid for further enhancements and future extension of the CV32E40X core into a more powerful LSC. This iterative approach allows one to build upon existing designs and gradually incorporate new features and optimizations.

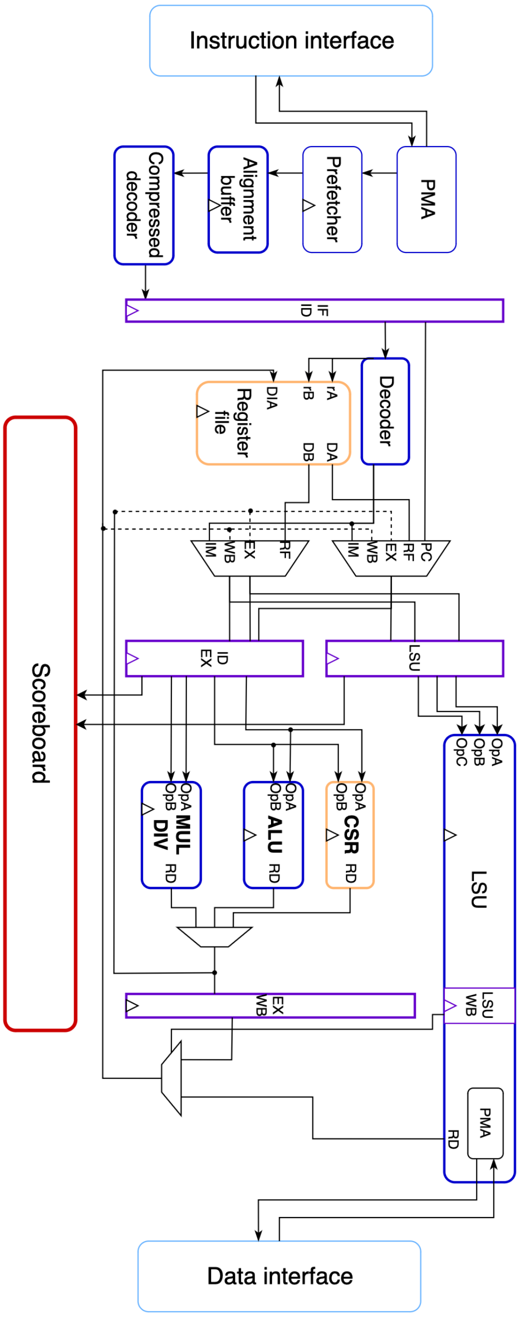


Figure 4.5: Block diagram showing modification to the CV32E40X core

4.6 Significance and Novelty of Proposed Architecture

The proposed architecture addresses performance limitations caused by memory accesses without adding more complexity to the design, minimizing the area and power overhead. There already exist architectures that try and address slow memory accesses. The previously discussed superscalar architecture and OoO execution are examples of this. OoO execution does this by constantly looking in the instruction queue to which instruction is ready to execute and issuing them for execution. So valuable work is done while memory accesses are ongoing. Superscalar architecture also does this by utilizing all of the functional units. However, with superscalar, the instructions are not issued for execution OoO. For example, if there are a lot of back-to-back load instructions, the other functional units cannot be utilized since instructions are issued in order. Superscalar architecture and OoO execution can provide good results in performance output, however, often at the expense of area and power. Therefore, Such implementations are not viable for some low-power embedded systems.

Load instructions tend to have a longer latency than other instructions due to memory being much slower than a processor. With the proposed architecture, higher per- performance is achieved by parallelizing memory operations with the rest of the workload. By doing this, the processor can overlap the execution of memory operations with other computational tasks, effectively reducing the overall execution time. In addition, by overlapping memory operations with other computational operations, the processor can keep more functional units busy, extracting more work from each clock cycle and maximizing system throughput.

A limitation of this proposed architecture is that it is very workload dependent. This architecture's potential performance benefits depend on how long the memory latency is. The longer the latency is to retrieve data from memory, the better the performance potential is. When examining the trade-offs between power, area, and performance, performance potential is important. If memory accesses on specific embedded systems are not a bottleneck, that would mean that the proposed architecture would only add to the area and power without improving the performance much, making the implementation not worth it. The fact that embedded processors are optimized to perform specific workloads speaks in favor of this proposed architecture, as the goal of the architecture is also to optimize a specific workload.

The novelty of the proposed architecture lies in the fact that it seeks to parallelize specific workloads. It can be seen as a restrictive superscalar. With superscalar, the idea is to utilize all available functional units concurrently and have multiple copies of the same functional unit. However, with the proposed architecture, the approach is more toward parallelizing specific workloads. More specifically, allowing memory operations to execute parallel with arithmetic and control operations. It, therefore, requires less logic than a superscalar processor as there is less information to keep track of.

Chapter 5

Design and Implementation

5.1 Introduction

This chapter will describe the practical design and implementation of this work. The task in question is to extend the capabilities of the CV32E40X core [49] in the OpenHW Core-V family [12] for load/store instructions and other instructions to be able to execute in parallel, meaning that the internal HDL of the core will need to be extended and altered. CV32E40X has a comprehensive verification and simulation environment, encompassing the compilation of C or assembly applications, which serve as stimuli for simulation purposes. The modified core design is available on GitHub.

The work consists of a skid buffer module and modifications on the pipeline stages (ID, EX, and WB) and the controller. In addition, an assertion module for the skid buffer has been designed to verify the design. Formal verification is the approach that has been utilized. An overview of an estimate of the complete system is shown in Figure 4.5

5.2 Approach

The design process in this work is structured into three distinct steps, aiming to facilitate the precise and comprehensive development of the complex system, CV32E40X. The first step is logic design using microarchitecture. The second step is HDL implementation in SystemVerilog. The third and last step is verification, also in SystemVerilog, and testing, done through the applications available in the CV32E40X verification and simulation environment.

The microarchitecture serves as a block diagram, visually representing the intricate interconnections and subcomponents within a large and complex design. Managing the numerous components and connections can be challenging, making the microarchitecture an invaluable tool for comprehending the entire system and effectively describing the design.

After getting a better overview of the intended design through microarchitecture, the design is implemented in hardware using SystemVerilog and verified through the verification environment.

5.3 ID Stage

One of the first steps in achieving parallel execution between load/store instructions and other instructions is to remove the dependency between the LSU and EX stage, dividing the EX stage into two pipeline stages. The first step in the design phase was to divide the ID/EX pipeline register from the baseline CV32E40X into two separate pipeline registers, one for the EX stage and one for the LSU. Both pipeline registers have a reset condition and a non-reset condition. The register values are updated in the non-reset condition based on certain conditions. For the LSU pipeline, the register values are assigned if both *id_valid_lsu_o* and *lsu_ready_i* are true. The ID/EX pipeline retains the same conditions when *id_valid_o* and *ex_ready_i* are true. However, the valid signal for the ID stage is slightly modified. Instead of having one valid signal for the ID stage, two valid signals are implemented, one for the LSU and one for the EX stage. The reasoning is to avoid a situation where a pipeline register that should not be updated gets updated. Consider a situation where there is an instruction in the ID stage with both the LSU and EX stage ready for a new instruction. Having one valid signal for the ID stage would then move valid instructions to the LSU and ID/EX pipeline. As a result, there would be a valid instruction in one of the pipeline registers that should not be there. This can cause instructions ahead in the pipeline to get deleted by accident.

Figure 5.1 shows the modification done in the ID stage, as discussed above. Firstly, two, instead of one, ready signals are sent from the EX stage. Since the EX stage and the LSU are not dependent anymore, *ex_ready* is now only dependent on the *wb_ready* signal. *lsu_ready* depends on both the granted signal from memory and *wb_ready* from the WB stage. Load/store instructions transition from the EX stage to the WB stage when these two signals are true, as was discussed in section 3.2.3. This leaves the EX stage of the LSU empty and ready for a new memory instruction. *id_ready* is now true if either or both *lsu_ready* and *ex_ready* are true. This depends on which type of instruction is currently in the ID stage. *lsu_en* signal determines what type of instruction is currently in the ID stage. Looking at Figure 5.1, one can see that it is used as the select signal in the multiplexer (mux). If *lsu_en* is true, a load or a store instruction is currently in the ID stage, meaning that the *id_ready* signal is set by *lsu_ready*. If *lsu_en* is false, then the instruction is not a memory instruction, and *id_ready* is set by *ex_ready*.

Lastly, an additional counter has been inserted in the ID stage. The purpose of the counter is to give each instruction an ID to tell later on in the pipeline which instruction is the oldest. With instruction ID, the aim is to maintain program order when committing instructions and, therefore, have precise exceptions. This counter counts every time there is a valid instruction in the ID stage, and the value of the counter is assigned to the ID of the instruction when it moves in the pipeline register.

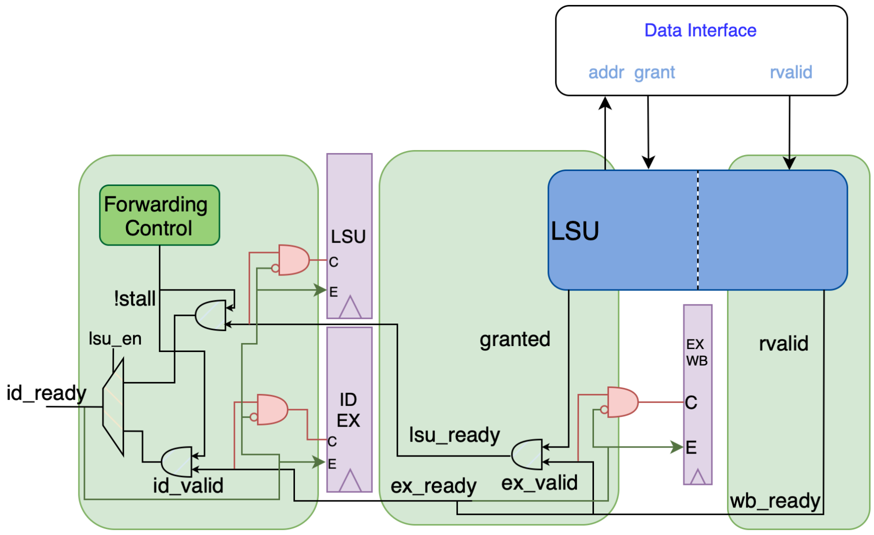


Figure 5.1: Updated ID stage

5.4 EX Stage

The first important thing to be done in the EX stage is removing the LSU interface. Since the goal is to execute instructions in the LSU and EX stage in parallel, removing all dependencies between these two stages is required. This interface is used to share information between the LSU and EX stage. The LSU informs the EX stage when it is ready for new data and when it has been granted its request and is ready to move to the WB stage. The LSU also shares information on when there is a misaligned transaction since, in practice, two instructions would need to be executed for one misaligned transaction. Figure 5.2 shows the interface between the LSU and EX stages. The LSU signals that it is ready through the EX stage by sending it information on the state of the instruction. On the other hand, the EX stage keeps LSU updated on the validity of the instruction and when the WB stage is ready so that the LSU can transition its instruction to the next stage. All the signals used in this interface must be cut out, and the LSU needs to use them independently of the EX stage.

Some logic in the EX stage depends on the signals received from the LSU and therefore needs to be altered. The state of instructions in the LSU affects the valid and ready signals for the EX stage. EX stage is ready for new data when an instruction is killed and otherwise when all functional units are ready unless the stage is being halted. EX stage has valid data when one functional unit has a valid output and is enabled for executing instructions. For example, if the ALU is enabled and has a valid output without triggering an exception, then the EX stage has valid data. Therefore, the ready and valid signals in the EX stage were modified not to consider the state of the LSU, which were the signals *lsu_valid_0* and *lsu_valid_0* seen in Figure 5.2. As mentioned above, the other three signals sent from the LSU to the EX stage are related to a misaligned memory transaction.

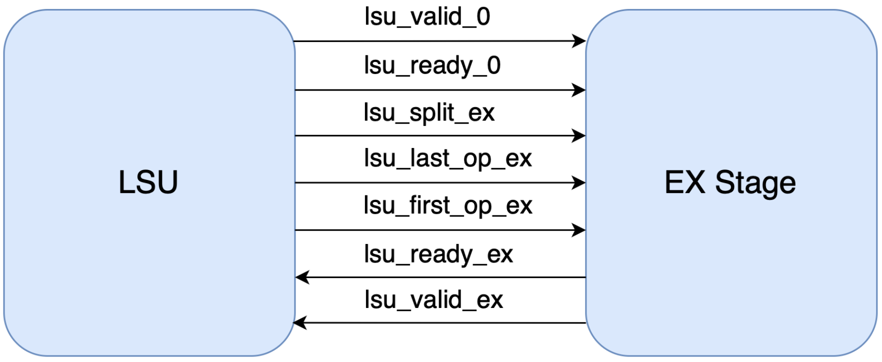


Figure 5.2: Block diagram of the LSU/EX interface

When performing a load or store instruction in Main memory, if the effective address of the data is not naturally aligned to the referenced datatype, certain conditions arise. Precisely, when a word access is attempted, and the address is not aligned on a four-byte boundary, or when a half-word access is attempted, and the address crosses a word boundary, the load or store instruction is executed as two bus transactions. The first transaction corresponds to data transfer from the lowest address involved, and the second transaction handles the remaining data. The signals *lsu_last_op_ex* and *lsu_last_op_ex*, sent from the LSU, are used in the EX stage to determine when the last transaction is finished so that the WB stage does not interpret the two transactions as two instructions but as one. Since all other instructions that are executed in the EX stage are all in one operation, it is quite simple to move the logic used in the EX stage to handle misaligned memory accesses over to the LSU module. This leaves the last two signals sent from the EX stage over to the LSU.

Two signals are sent from the EX stage to the LSU. *lsu_ready_ex* signal is set by the *wb_ready* signal. So instead of having the EX stage send this signal to the LSU, the WB stage sends its ready state directly. *lsu_valid_ex* is set by *lsu_en* and *lsu_en*. These signals are sent from the ID stage through the pipeline registers. Now that there are implemented two pipeline registers between the ID stage and the two stages, the EX stage and the LSU stage, the LSU can retrieve these signals from the LSU pipeline register instead of getting them from the EX stage.

In addition to removing the interface mentioned above between the EX stage and LSU, logic has been added to determine which instruction should be moved to the WB stage to commit. This decision is based on which instruction is the oldest. Knowing the oldest instruction is relevant when executing instructions in the EX stage and the LSU. The signal is called *priority* and is sent to the WB stage, where the final decision is made. The logic is quite simple. It first checks if there are valid instructions in the LSU and EX stage. If there are, the instruction ID is compared between these two instructions. The instruction with the lowest ID is the oldest and gets priority to commit first. If there is just one instruction in the LSU or EX stage, that instruction gets immediate priority.

5.5 LSU

The main changes done on the LSU have been to change which pipeline register the LSU gets its data from, including a pipeline register between the LSU and WB stage. However, there was a problem that was encountered during the design phase. The problem involved the loss of data from memory. In the baseline design, there is no downstream between EX and WB stages for load/store instructions. No downstream means that when a load/store instruction is executed in the EX stage, the WB stage will always remain ready. This is, however, a problem if the goal is parallel execution, as the WB stage can only be ready for the oldest instruction in the pipeline. Since the WB stage is not always ready with this new design, an issue can lead to data loss from memory. The solution chosen to solve this problem is the skid buffer. The skid buffer is the smallest pipeline FIFO buffer and is useful when there is a need to pipeline the path between a sender and a receiver for timing purposes. The following section will discuss the data loss issue and how the skid buffer works. In addition, an explanation of the implementation, including formal verification of the skid buffer, will be presented.

5.6 Skid-buffer

Skid Buffer is an elastic buffer to store the data when the receiver applies backpressure on the sender. The logic behind the skid buffer is quite simple as it incorporates a simple valid-ready handshake. The need for a skid buffer arises when there is a need to generate a stall signal in a registered data-only context. Consider the CV32E40X core, which does not use skid buffers. If there is an instruction the CPU needs to wait for, for example, a memory load, then the decode stage needs to stall, or else an instruction can be lost. This stall propagates backward, meaning that if the decode stage stalls, so will the fetch stage. Due to the CV32E40X core utilizing combinatorial stall signals, the stall signal reaches the fetch stage when little slack is left before the subsequent clock edge. The stall signals could have been registered using skid buffers, breaking the timing accumulation. The problem arises when the stall signal is registered since, in such cases, the preceding stage in the pipeline remains unaware of the stall until it completes processing and registers the values in the subsequent flip-flops.

Consequently, the data needs to go somewhere or get dropped. A similar issue arises with the LSU when trying to change the timing for when it is ready to receive data from memory. From the discussion on the LSU in section 3.2.3, it is understood that the LSU sends a request to memory in the EX stage. Load/store instruction transitions from the EX to WB stage when memory is ready to serve this request. If the LSU is not ready to receive data from memory when a memory request is granted, or in other words, the WB stage is busy with another instruction, the data sent from memory will be lost.

5.6.1 Microarchitecture

It is here where the skid buffer comes in, as shown in Figure 5.3. The objective of the skid buffer is to establish a connection between the combinatorial logic on one side and the registered logic on the other side by facilitating the transmission of the outgoing stall signal, *!o_ready*, which is limited to being a registered signal. Two rules must be followed for the skid buffer to function correctly. The first rule is if the *Valid* signal is set high and the *Ready* signal is set low, as such *Valid & !Ready*, the respective data must remain constant into the next clock cycle. The second rule is that no data may be lost along the way. See section 5.6.3 for the formal expression of these two rules.

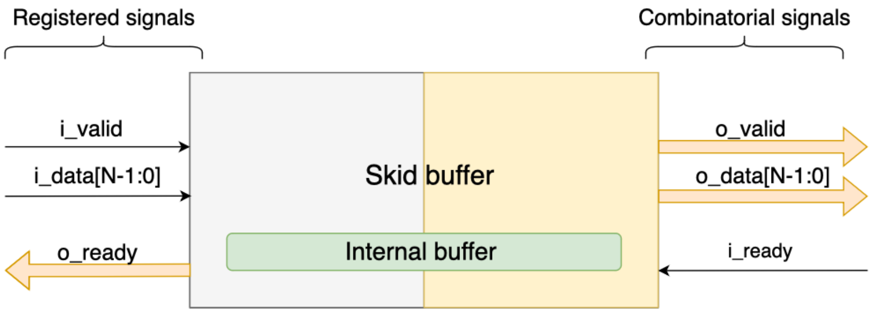


Figure 5.3: A block diagram of a basic skid buffer

When the receiver is ready to receive data, the buffer acts like a pass-through device. Look at Figure 5.4. On the other hand, if it is not ready, everything needs to be copied to an internal buffer to prevent the input data from getting lost in the next cycle. Figure 5.5 gives a visualization of this, where one can see that the incoming **Valid** and data lines are stored in the internal buffer, which gives the buffer its own internal valid signal and data line.

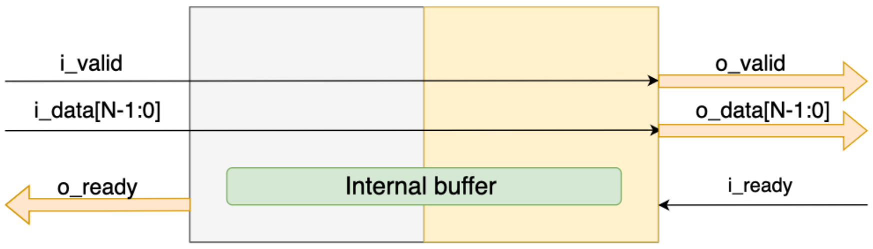


Figure 5.4: When there is no stalls, the buffer will act like a pass through

A quick overview. The skid buffer must operate like a pass-through device when the receiver is ready, as shown in Figure 5.4. The incoming *Valid* and data lines are directly sent to the output. When the receiver is not ready, the incoming *Valid* signal and data lines are stored in the internal buffer. In the next clock cycle, stored data can be sent to the output. Moreover, while the incoming interface may progress to its subsequent data, the discussed module detects that the incoming *o_ready* signal is already set to false and therefore knows that it needs to wait, as shown in Figure 5.6.

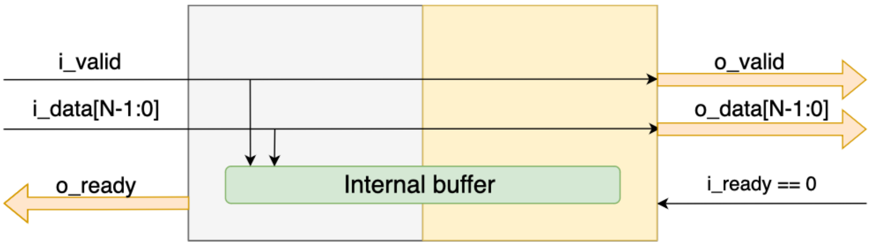


Figure 5.5: Copying the incoming data to an internal buffer

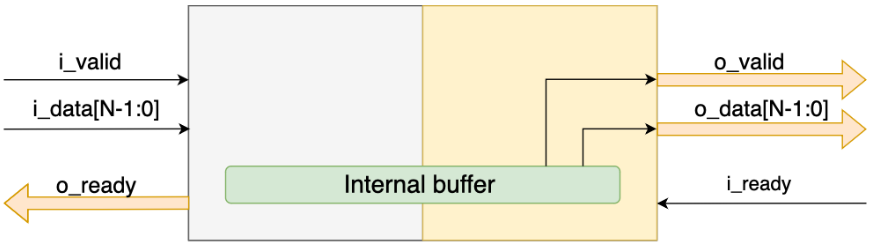


Figure 5.6: The stall signal propagates upstream

5.6.2 Implementation of the skid buffer

The interface of the skid buffer module is set up similarly to what is visualized in figs. 5.3 to 5.6, with three inputs and three outputs. There are three internal signals that the internal buffer is captured by. The first one, *bypass_reg*, indicates which mode the skid buffer runs on. If *bypass_reg* is low, the skid buffer runs on ”skid” mode, storing valid data in the internal buffer. If it is set to high, on the other hand, it will work like a pass-through device. The way the logic of *bypass_reg* works is that every time a valid signal is incoming, but the output is stalled, *bypass_reg* is set to low, or ”skid” mode. The skid buffer operates like a pass-through device as default. Therefore, the internal buffer is cleared, and *bypass_reg* is set high after a reset. Look at Listing 5.1.

```

/*-------------------------------------------
   Synchronous logic
-------------------------------------------*/
always @(posedge clk, negedge rst_n) begin

   // Reset condition
   if (rst_n == 0) begin

      // Internal Registers in the skid buffer
      ready_reg  <= 1'b0 ;
      data_reg   <= '0   ;
      bypass_reg <= 1'b1 ;
```

```

    end

    // Out of reset
    else begin

       // Bypass state
       if (bypass_reg) begin

          ready_reg <= 1'b1;

          if (!i_ready && i_valid && ready_reg) begin
             ready_reg  <= 1'b0   ;
             data_reg   <= i_data ;  // Data skid happened, store to buffer
             bypass_reg <= 1'b0   ;  // To skid mode
          end
}
```

Listing 5.1: Code for the syncronous logic of the skid buffer

Once the receiving end is ready, the skid buffer returns to regular operation (pass-through device). See Listing 5.2

```
         // Skid state
       else begin

          if (i_ready) begin
             ready_rg  <= 1'b1   ;
             bypass_rg <= 1'b1   ;         // Back to bypass mode
          end
```

Listing 5.2: Listing of when the skid buffer is in bypass mode

Listing 5.3 shows the assignments of the outputs on the skid buffer. The logic for the output data is quite simple. It is a mux with *bypass_reg* as the select signal. If *bypass_reg* is high, the data output gets the value from the input. If *bypass_reg* is false, the output data is assigned by the data stored internally in the buffer.

```

/*-------------------------------------------------------------
   Continuous Assignments
-------------------------------------------------------------*/
assign o_ready = ready_reg                                 ;
assign o_data  = bypass_reg ? i_data  : data_reg           ;
assign o_valid = bypass_reg ? (i_valid & ready_reg) : 1'b1 ;
```

Listing 5.3: Listing of the continous assignments of the outputs to the skid buffer

5.6.3 Formal Verification of the Skid Buffer

In order to ensure the correctness and reliability of the skid buffer module, the approach employed is formal verification, more specifically with the use of assertions in SystemVerilog. Formal verification techniques provide an accurate and complete analysis of the design properties and help identify potential bugs or corner cases that may go unnoticed during simulation testing. Simulation of digital circuitry can be very long, so verifying the behavior of the different modules manually by, for example, looking at signal flow can be very time-consuming. However, it is very important to have high assertion coverage, as the assertions cover critical aspects of the design.

Following any reset, all internal signals in the skid buffer need to be cleared. The behavior, as shown in Listing 5.4, can therefore be assumed.

```
property p_reset_clears_reg;
    @(posedge clk)
    !rst_n |=> !ready_rg_i && (data_rg_i == '0) && bypass_rg_i;
endproperty

assert property(p_reset_clears_reg);
```

Listing 5.4: Property checking if reset state is functioning correctly

Another property that needs to be checked is that whenever the receiver is not ready, the valid signal needs to continue into the next clock cycle, and the data needs to remain stable. See Listing 5.5.

```
property p_data_held_when_not_ready;
    @(posedge clk) disable iff (!rst_n)
    i_valid && !o_ready |=> i_valid && $stable(i_data);
endproperty

assert property(p_data_held_when_not_ready);
```

Listing 5.5: Property checking if the behaviour is correct for when the receiver is not ready

There are three rules that the aim is to preserve when designing this skid buffer. The first rule is when there is a stalled outstanding request on the output port. When this happens, the request must remain stable to the next clock. See Listing 5.6.

```
property p_data_stable_after_o_valid;
    @(posedge clk) disable iff (!rst_n)
    o_valid && !i_ready |=> (o_valid && $stable(o_data));
endproperty

assert property(p_data_stable_after_o_valid);
```

Listing 5.6: Property checkingthe first rule. Request must remain stable if there is a stalled outstanding request

The second rule checks for a critical aspect of the skid buffer: the ”no data loss” policy. If there is any incoming data, then it needs to either be sent to the output or stored in the buffer. See Listing 5.7.

```
property p_passthrough_or_store;
    @(posedge clk) disable iff (!rst_n)
    (!i_ready && i_valid && ready_rg_i)
    |=> (data_rg_i == $past(i_data));
```

```
endproperty

assert property(p_passthrough_or_store);
```

Listing 5.7: Property checking that no data is lost

Analyzing the other corner cases for the second rule reveals that if either of these scenarios, *!i_valid* or *i_valid && !o_ready*, occurs, then noting happens in the input that requires attention. Another interesting corner case to consider is when *ready_rg* is low. Specifically, when *ready_reg* is low, and *i_ready* is high, the buffer operates as a straight pass-through, as verified by a brief design examination. Consequently, the remaining case pertains to a scenario for when *ready_reg* is low and *i_ready* is also low, which was discussed previously for Listing 5.7.

Nonetheless, the initial analysis is still incomplete, as it does not discuss how the design returns to an idle state. This aspect is important, as overlooking the return idle can result in hard-to-find bugs. Therefore, ensuring the design's proper return to idle can be crucial. So any time *i_ready* is true on the outgoing interface, all relevant signals should be cleared. Subsequently, on the next clock cycle, *o_valid* should only be true if *i_valid* is also true, which is the third rule. See Listing 5.8.

```
property p_passthrough_or_store;
    @(posedge clk) disable iff (!rst_n)
    (!i_ready && i_valid && ready_rg_i)
    |=> (data_rg_i == $past(i_data));
endproperty

assert property(p_passthrough_or_store);
```

Listing 5.8: Propoerty checking for passthrough or store of the input data

5.7 WB Stage

There have been a few modifications done in the WB stage. Many changes have been primarily due to two pipeline registers instead of one. Looking at Figure 4.5 shows there is now a new pipeline between the LSU and WB stage called LSU/WB. So the necessary changes were to change which signal should be retrieved from which pipeline register. However, a more considerable modification is done involving the *wb_ready* signal. Two individual ready signals are implemented in WB—one for LSU and one for the EX stage. The idea is that by having two ready signals, one can control which instructions can move to WB to commit. The previously mentioned *priority* signal discussed in section 5.4 is retrieved in WB and used to determine which of the ready signals should be set to true. So by knowing what value *priority* has, the processor will know which instruction to let in.

Chapter 6

Experimental Setup and Analysis

The performance analysis of the CV32E40X core, a member of the CORE-V family of RISC-V cores, was conducted utilizing the functional verification project known as CORE-V-VERIF, developed by OpenHW Group [50]. The CORE-V-VERIF project encompasses a comprehensive verification environment based on the Universal UVM. Within this environment, two established benchmarks, namely coremark, and hello-world, along with two custom assembly programs, csr_instr_asm and ricv_arithmetic_basic_test_1, were employed for analysis.

This evaluation aims to compare the modified design of the CV32E40X core against the baseline CV32E40X by examining the test results obtained from running the programs mentioned above. It is important to acknowledge that the findings of this study may be subject to limitations stemming from the relatively small sample size of benchmarks used, which may only partially capture real-world conditions. Consequently, the results should be interpreted with caution.

This section will provide a detailed account of the testing methodology employed in the analysis. It will explain the specific procedures to execute the tests and the subsequent steps to analyze the collected data. This section aims to establish a solid foundation for the subsequent discussion and interpretation of the results by describing the testing process and analytical framework.

6.1 Baseline CV32E40X

CV32E40X is still an ongoing project, so constant modifications are happening to the core. It is, therefore, important to ascertain which version of the CV32E40X is referred to as the baseline version in this work. In Git, a commit hash, or commit ID, refers to a unique identifier for a specific commit within a Git repository. So the commit hash to the version of CV32E40X core that is referred to as baseline and that the work presented in this thesis is built upon is the following: **e982a971ebb79e07cfa528c7dfcf7a35b72da5cd**.

6.2 Toolchain

This section presents the toolchain and methodology employed to improve the performance of the CV32E40X core. The following tools were utilized at various stages of the design process

1. Simulation: QuestaSim and ModelSim, widely-used simulation tools, played a crucial role in validating and verifying the functionality of the enhanced CV32E40X core. Through QuestaSim and ModelSim, extensive simulations were conducted to ensure the correctness and accuracy of the modified design. It facilitated examining the core's behavior under different scenarios and input stimuli. QuestaSim was mainly used for signal flow analysis to verify correct behavior and to identify potential errors.

2. Synthesis: CadenceGenus. The synthesis stage involved transforming the CV32E40X core's RTL (Register Transfer Level) design into a gate-level representation. Cadence Genus, a powerful synthesis tool, was employed to achieve this conversion. Genus optimized the design for area, power, and timing, utilizing advanced algorithms and optimizations to enhance the core's performance characteristics.

3. Automatic Place-and-route++: Cadence Innovus. Cadence Innovus, an automatic place-and-route tool, efficiently mapped the synthesized design onto the target hardware. Innovus utilized sophisticated algorithms and techniques to determine the optimal core components placement and efficiently route the interconnections.

4. STM28 nm Technology Library. The design used the STM (Standard Technology Model) 28nm technology library. This library provided a set of standardized design components and process models specific to the 28nm technology node.

6.3 Evaluation

The upcoming section will provide a comprehensive overview of the statistical measures utilized to analyze the performance of the CV32E40X core. Specifically, the modified version of the CV32E40X core will be evaluated and compared against the baseline version, CV32E40X. The primary focus of the evaluation will be on the IPC metric obtained by executing various test programs. One can assess the impact of the modifications on the core's performance by comparing the IPC values between the modified and baseline versions.

$$\text{IPC} = \frac{Instruction\ count\ of\ the\ program}{Cycle\ count\ of\ the\ program} \tag{6.1}$$

$$CPI = \frac{Cycle\ count\ of\ the\ program}{Instruction\ count\ of\ the\ program} \tag{6.2}$$

IPC is a measure that calculates the average number of instructions executed by a CPU during a specific time frame, divided by the number of clock cycles needed for their execution. IPC is employed for determining speedup, as shown in equation 6.3. Cycles Per

Instruction (CPI), or Cycles Per Instruction, is another commonly used metric in processor performance evaluation. It is simply the reciprocal of IPC and is chosen based on the evaluator's preference. The formula for CPI is provided in equation 6.2.

$$speedup = \frac{IPC_{modified}}{IPC_{baseline}} \tag{6.3}$$

Speedup is a commonly used measure to evaluate the overall performance of processors when running benchmarks. It provides a clear indication of performance improvement. In this context, the speedup is calculated by dividing the achieved IPC of the modified core by the IPC of the original core prior to the modifications. This report examines the potential increase in IPC by allowing instructions to execute concurrently with independent store or load instructions. The theoretical IPC resulting from this approach can then be compared to the IPC measured from the benchmark results, offering insights into the potential speedup achievable with such an implementation. Intuitively, a speedup value greater than 1 signifies a performance improvement.

6.4 Simulation Output and Data Log

UVM report summary is used to see if a simulation was executed successfully. Figure 6.1 shows what the report summary after finishing a simulation looks like.

There is also a message printed out when the program has finished executing. For the hello-world program, look at Figure 6.2 to see how it looks.

6.4.1 RISC-V Formal Interface

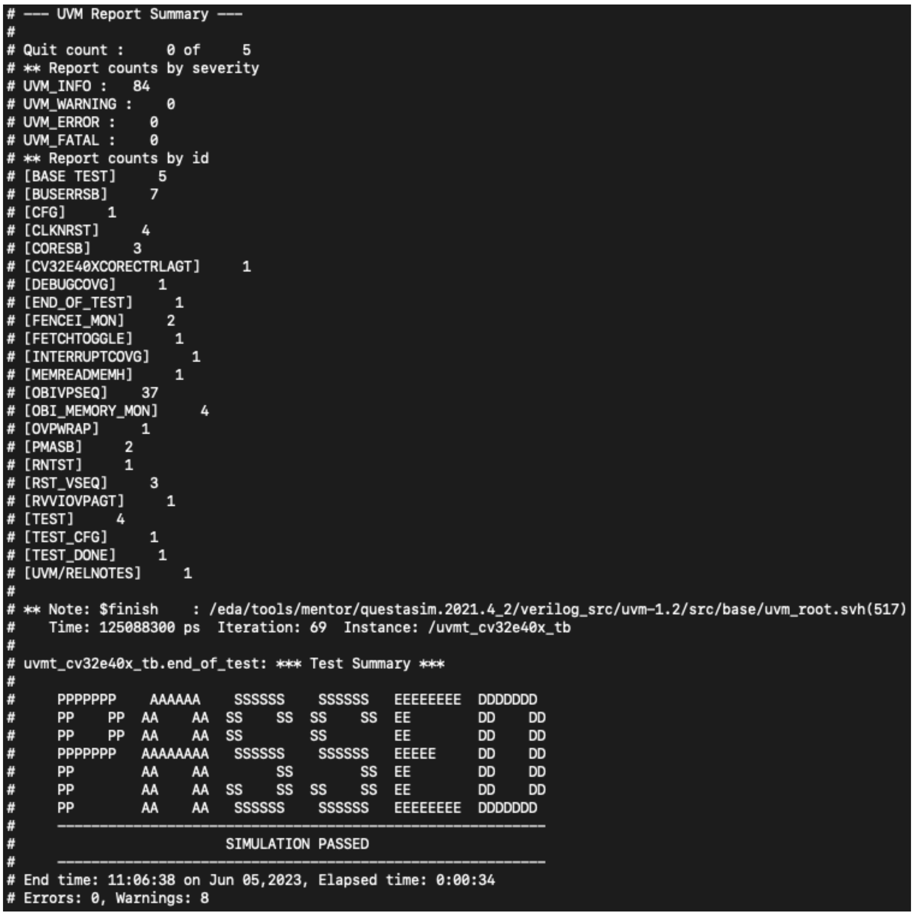
A bindable RISC-V Formal Interface (RVFI) interface is provided with the core. The module is called *CV32E40X_rvfi* and is used to create a log of the executed instructions. This module has three purposes, which goes as follows:

- Formal verification
- Produce an instruction trace
- Used as a monitor for verification reasons

As mentioned, the textitCV32E40X_rvfi module creates a log of executed instructions generated when a test program finishes execution. Figure 6.3 shows what information is included in the execution log. Information such as what type of instructions are executed, what operands were used, and what cycle they finished with its execution is included. The column *ORDER* shows the order in which the instructions finished executing.

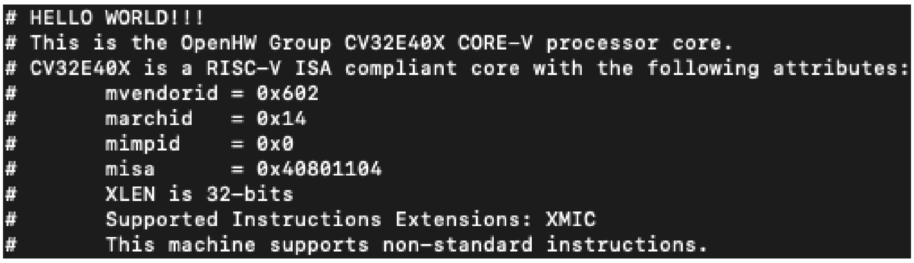
6.5 Experimental framework and Baseline Performance

The coremark benchmark is designed to assess the performance of microcontrollers (MCUs) and central processing units (CPUs) commonly used in embedded systems. It utilizes the



```
# --- UVM Report Summary ---
#
# Quit count :     0 of     5
# ** Report counts by severity
# UVM_INFO :   84
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [BASE TEST]     5
# [BUSERRSB]     7
# [CFG]     1
# [CLKNRST]     4
# [CORESB]     3
# [CV32E40XCORECTRLAGT]     1
# [DEBUGCOVG]     1
# [END_OF_TEST]     1
# [FENCEI_MON]     2
# [FETCHTOGGLE]     1
# [INTERRUPTCOVG]     1
# [MEMREADMEMH]     1
# [OBIVPSEQ]    37
# [OBI_MEMORY_MON]     4
# [OVPWRAP]     1
# [PMASB]     2
# [RNTST]     1
# [RST_VSEQ]     3
# [RVVIOVPAGT]     1
# [TEST]     4
# [TEST_CFG]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
#
# ** Note: $finish    : /eda/tools/mentor/questasim.2021.4_2/verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
#    Time: 125088300 ps  Iteration: 69  Instance: /uvmt_cv32e40x_tb
#
# uvmt_cv32e40x_tb.end_of_test: *** Test Summary ***
#
#     PPPPPPP    AAAAAA    SSSSSS    SSSSSS   EEEEEEEE  DDDDDDD
#     PP    PP  AA    AA  SS    SS  SS    SS  EE        DD    DD
#     PP    PP  AA    AA  SS        SS        EE        DD    DD
#     PPPPPPP   AAAAAAAA   SSSSSS    SSSSSS   EEEEE     DD    DD
#     PP        AA    AA        SS        SS  EE        DD    DD
#     PP        AA    AA  SS    SS  SS    SS  EE        DD    DD
#     PP        AA    AA   SSSSSS    SSSSSS   EEEEEEEE  DDDDDDD
#     ----------------------------------------------------------
#                       SIMULATION PASSED
#     ----------------------------------------------------------
# End time: 11:06:38 on Jun 05,2023, Elapsed time: 0:00:34
# Errors: 0, Warnings: 8
```

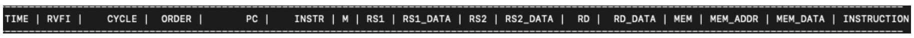
Figure 6.1: UVM report summary of ricv_arithmetic_basic_test_1 program



```
# HELLO WORLD!!!
# This is the OpenHW Group CV32E40X CORE-V processor core.
# CV32E40X is a RISC-V ISA compliant core with the following attributes:
#       mvendorid = 0x602
#       marchid   = 0x14
#       mimpid    = 0x0
#       misa      = 0x40801104
#       XLEN is 32-bits
#       Supported Instructions Extensions: XMIC
#       This machine supports non-standard instructions.
```

Figure 6.2: hello-world message

CoreMark/MHz metric to measure the single-threaded performance per clock frequency. This metric is derived by dividing the single-core CoreMark score by the clock speed employed during the benchmark execution. The benchmark incorporates various algorithms



```
TIME | RVFI |    CYCLE |  ORDER |       PC |    INSTR | M | RS1 | RS1_DATA | RS2 | RS2_DATA |   RD |   RD_DATA | MEM | MEM_ADDR | MEM_DATA | INSTRUCTION
```

Figure 6.3: Overview over what information is stored in the log of executed instructions

and workloads, including list processing (find and sort), matrix manipulation (common matrix operations), state machine (determining valid numbers in an input stream), and cyclic redundancy check (CRC). For additional details on the coremark benchmark, refer to Table 6.1, while Table **??** provides information specifically on the hello-world test.

Table 6.1: Details on coremark benchmark results on the baseline design of CV32E40X core

Baseline CV32E40X	
Benchmark	Coremark
Instruction count	2753145
Cycle count	6400027
IPC	0.43

The hello-world test program is an integral part of the core-v-verif repository and serves as a sanity test designed for the CV32E40X core. This program performs simple tasks such as reading the MISA and MVENDORID CRCs and printing informative messages to the standard output, including the famous ”HELLO WORLD!!!” greeting. As a minimalistic workload, the hello-world program is an initial step in analyzing the core, offering a smaller-scale evaluation compared to the more comprehensive coremark benchmark.

Table 6.2: Details on hello-world benchmark result on the baseline design of CV32E40X core

Baseline CV32E40X	
Benchmark	hello-world
Instruction count	9929
Cycle count	21377
IPC	0.46

In addition to hello-world and coremark, a custom assembly program has been made to test whether load/store instructions can run in parallel with other instructions and also try to see if the design can deal with dependencies. Look at Table 6.3 for program results and details. The program is minimal, only including 19 instructions, a small sample size that is mostly used for testing the design in the design phase. Look at the appendix (Something) for a more detailed overview of the assembly program. The IPC seems higher than expected, given the cycles and instruction counts presented. However, 170 cycles are used for the startup process. The program execution itself takes only 86 cycles to finish. This has not been considered for hello-world and coremark since these are much bigger programs, and the impact from those 170 cycles on IPC is negligible.

ricv_arithmetic_basic_test_1 is an assembly program and a part of the primary verifica-

Table 6.3: Details on csr_instr_asm result on the baseline design of CV32E40X core

Baseline CV32E40X	
Program	csr_instr_asm
Instruction count	49
Cycle count	256
IPC	0.57

tion environment under custom tests. It runs primarily arithmetic operations, as is insinuated by the program name. Table 6.4 shows the test results and details of the program on the baseline CV32E40X. The word ”done” is printed in the terminal output if the program can execute correctly.

Table 6.4: Details on ricv_arithmetic_basic_test_1 result on the baseline design of CV32E40X core

Baseline CV32E40X	
Program	ricv_arithmetic_basic_test_1
Instruction count	11455
Cycle count	37893
IPC	0.3

Chapter 7

Results and Discussion

The main contributions of this work have been the implementations done on the core itself. This chapter will go through the results achieved from running simulations on the modified CV32E40X core and a discussion evaluating the test results alongside if the trade-offs with area and power can be worth it.

7.1 Performance Evaluation

This section will present and evaluate the performance results on hello-world, coremark, csr_instr_asm, and ricv_arithmetic_basic_test_1 for the modified design of the CV32E40X core. The functionality of the modified design will also be discussed. It is important to acknowledge that several bugs were encountered while implementing and evaluating the proposed architecture during the development process. These bugs affected the functionality and overall performance of the processor, thereby limiting the program's completion and compromising the performance evaluation's accuracy.

7.1.1 coremark

Table 7.1 gives an overview of the result retrieved from running the coremark benchmark on the modified version of CV32E40X. Comparing Table 6.1 and Table 7.1 shows that the baseline CV32E40X core performs better regarding IPC. The baseline version has an IPC of 0.43, and the modified version has an IPC of 0.382. That gives a speedup of 0.88. There is also a considerable disparity in instruction count between the two versions of CV32E40X. Baseline CV32E40X can execute 2753145, while the modified CV32E40X can only execute 182898 instructions. Looking at the simulation output shows a " PASSED" message was received. However, coremark also has a self-check that also checks for any errors. The simulation environment was not able to find any errors. The self-check, on the other hand, did report errors. The error report from the self-check is the most plausible reason the program's execution is cut short for the modified CV32E40X.

Errors question the test result's validity since it is hard to say if the design implementation is giving a worse performance or if potential unresolved bugs are hindering the performance. Also, all of the assertions in the verification environment are not triggering, making it harder to pinpoint where the problem is.

Details on coremark benchmark results on the baseline design of CV32E40X core

Baseline CV32E40X	
Program	coremark
Instruction count	2753145
Cycle count	6400027
IPC	0.43

Table 7.1: Details on coremark result on the modified design of CV32E40X core

Modified CV32E40X	
Program	coremark
Instruction count	182898
Cycle count	478818
IPC	0.382

7.1.2 hello-world

A similar conclusion on the coremark results can also be drawn from the test results shown for the hello-world program in Table 7.2. Only 1872 of 9929 instructions were able to execute before the simulation stopped. The simulation tests end when there are five triggers from assertions or when the simulation is complete. This could be from one or multiple assertions as long as they are triggered five times. Like the coremark, hello-world also performs worse on the modified design.

Details on hello-world benchmark result on the baseline design of CV32E40X core

Baseline CV32E40X	
Benchmark	hello-world
Instruction count	9929
Cycle count	21377
IPC	0.46

Table 7.2: Details on hello-world result on the modified design of CV32E40X core

Modified CV32E40X	
Program	hello-world
Instruction count	1872
Cycle count	4196
IPC	0.446

7.1.3 csr_instr_asm

As can be seen from the result retrieved from the baseline design, seen in Table 6.3, and the modified design, seen in Table 7.3, both versions of the CV32E40X core achieved almost the same IPC results. The baseline version finishing with 0.57 IPC and the modified version finishing with 0.55 IPC. Based on these results, the baseline runs faster than the modified version again. However, as discussed in more depth in section 7.2.1, the results are invalid since the execution trace generated for the modified design needs to be corrected. It seems that the incorrectness of the execution log is more aimed at the *ORDER* and *INSTRUCTION* columns. Both designs generated the same output from running the program, printing out ”PASSED” and ”CV32 done” in the same number of clock cycles. It gives reason to believe that the modified CV32E40X could execute the program, csr_instr_asm, in the same amount of time as the baseline CV32E40X.

Details on csr_instr_asm result on the baseline design of CV32E40X core

Baseline CV32E40X	
Program	csr_instr_asm
Instruction count	49
Cycle count	256
IPC	0.57

Table 7.3: Details on csr_instr_asm result on the modified design of CV32E40X core

Modified CV32E40X	
Program	csr_instr_asm
Instruction count	47
Cycle count	256
IPC	0.55

The fact that both versions achieved similar performance is somewhat predictable. The implemented design moves the stalling from the ID to the EX stage. Let us consider some examples. For the baseline design, if a load instruction takes some time to retrieve the data from memory, the ID stage will need to wait until the load instruction is finished executing. For the modified design, the instruction waiting in the ID stage, given that it is not a load/store instruction and there are no data dependencies, will be able to execute while the load instruction is waiting for data from memory. However the instruction will still need to wait for the load instruction to finish and commit before being able to commit. In theory, the stall is moved from the ID stage to the EX stage, and the total length of the stall is still the same.

What is implemented is more of a foundation to increase the performance potential. For example, adding an instruction queue in the WB stage would help avoid stalls in such situations presented in the previous paragraph. Instructions that are finished executing can move to the WB stage independently if they are the oldest instruction in the pipeline. As a result, the functional units are freed much earlier than previously, allowing the following instructions to execute. In the WB stage, the oldest instruction in the instruction queue is then committed.

7.1.4 ricv_arithmetic_basic_test_1

For this program, the CV32E40X core was able to complete it successfully. Looking at Table 7.4 shows that the modified CV32E40X managed to achieve the same performance as the baseline. However, the instruction count is two instructions short compared to the baseline. The missing instructions can be explained by the fact that only ”don” was printed to the terminal when the program finished executing. This means a store instruction could not execute correctly, missing the last character. The results achieved from running the ricv_arithmetic_basic_test_1 program strongly indicate that the bugs observed with the modified design are from the LSU. The primary function of this program runs primarily arithmetic operations, which are executed correctly based on the results.

Details on ricv_arithmetic_basic_test_1 result on the baseline design of CV32E40X core

Baseline CV32E40X	
Program	ricv_arithmetic_basic_test_1
Instruction count	11455
Cycle count	37893
IPC	0.3

Table 7.4: Details on ricv_arithmetic_basic_test_1 result on the modified design of CV32E40X core

Modified CV32E40X	
Program	ricv_arithmetic_basic_test_1
Instruction count	11453
Cycle count	37892
IPC	0.3

7.2 Bug Analysis

There are some potential culprits to what is causing the bugs on the modified design. One of these is related to the ready signals in the WB stage. One of the assertions that triggers when running the hello-world program is related to data forwarding. The properties are part of the assertion module for the top module of the core. It checks that the operands forwarded from the EX or WB stage are written to the WB register file. So if there is a data dependency, the forwarded data from the WB or EX stage must be written to the register file in the current or next clock cycle, respectively. At some point in the execution of hello-world, the data forwarded is not written to the register file. One possible correlation could be the ready signals in WB because there could be a corner case in which the ready signal to either the LSU or EX stage is not set high when it should be. As a result, the instruction cannot move to the WB stage to commit, delaying the instruction indefinitely. Another reason is that an instruction from the ID stage could be moved for execution when the functional unit is still in use, which deletes the executing instruction.

Another potential culprit could be when a split memory operation is performed. As discussed in section 5.4, having the WB stage interpret a split load/store instruction as one instruction, not two, is important. Split memory access is solved by setting the register write-enable signal to false. Nothing is written to the register file for the first transaction. However, the register write-enable signal is true for the last transaction. So the data is written to the register file when the last transaction is committed in the WB stage. Handling a split transaction wrong could trigger the assertions discussed in the previous paragraph. If, for example, instead of only writing the data from memory to the register file in the last transaction, it also happens in the first transaction, which means that the data forwarded and the data written to the register file are not the same. This error can propagate throughout the execution of the program since other instructions could use the wrongly written data to perform their calculations.

Outstanding transaction overflow is another problem occurring during the execution of the hello-world program. CV32E40X is only able to handle two outstanding transactions at a time. The reasoning for the overflow in outstanding transactions is still uncertain. However, what is certain is that the transfer requests are accepted faster than they are handled. So the line of outstanding transactions continues to grow. One possible problem could be that the skid buffer cannot apply back pressure, which is why there are too many transactions on the OBI bus at once.

7.2.1 Instruction log

There is an issue with the instruction log. More specifically, the way instructions are registered in the instruction log. An instruction is registered in the log when it commits in the WB stage. The log is generated through the RISC-V formal interface. *CV32E40X_rvfi* module checks when the *last_op_o* signal and *wb_valid_o* are both true. When this occurs, the instruction is registered in the instruction log. The *CV32E40X_rvfi* module tracks all of the instruction information. Look at Figure 6.3 to see what is printed in the instruction log. This module has not been modified to match the new modifications, so only the instructions in the EX stage are tracked. Therefore when a load instruction is executed, the previous instruction executed in the EX stage is registered in the instruction log, making it seem like the load instruction did not execute. Table 7.5 shows some of the instruction log from executing csr_instr_asm on the baseline version. Table 7.6 shows similar data from the instruction log from executing csr_instr_asm on the modified version for comparison. As can be seen from looking at both of the tables, the data are quite different, especially for instructions. It seems that either the module has been unable to register two instructions during the execution or that a bug has prevented two instructions from executing. Looking at the output from the simulation, nothing suggests that the program was not executed correctly. Both ”PASSED” and ”cv32 done” was printed out, just like the baseline version. Therefore, the execution could be correct for the modified CV32E40X, only that the CV32E40X_rvfi module could not generate the execution trace correctly due to not being modified to accommodate the design changes.

The execution log of instructions is essential during the design phase as it provides details of the sequence of instructions executed during the operation of the processor. It captures the execution flow, including the order and timing of instructions, register values, memory accesses, and other relevant events. If the execution log of instructions generated is useless, the verification phase becomes much more complicated, and the test results will not have any meaning.

7.3 Power, Area and Timing Analysis

The results gathered from running synthesis on both the baseline design and modified design goes as follows: the baseline design exhibited power consumption of 0.321607mW and an area utilization of 24181.809, while the modified design showed a slightly increased power consumption of 0.330064mW and a larger area utilization of 24990.442. Look at Tables 7.8 and 7.7 for a better overview over the results. This makes it an area overhead of 3.3% and power overhead of 2.6%. It is difficult to argue the slight increase in power and area for what benefits the implementation in this work brought, as unresolved bugs prevented the modified core from running application programs successfully. However, it could also argued that not much logic is needed to resolve the bugs based on test results and as was argued in section 7.1.3, adding instruction queues alongside the implementation presented in this work can tap into performance potential. Looking it from this perspective you could say it is a trade off between power, area and future potential. The small overhead in power and area, also leaves room for further optimizations in the future.

The results gathered from running synthesis on both the baseline design and modified design goes as follows: the baseline design exhibited a power consumption of 0.321607mW and an area utilization of 24181.809. In contrast, the modified design showed a slightly increased power consumption of 0.330064mW and a more extensive area utilization of 24990.442. Look at Tables 7.7 and 7.8 for a better overview of the results. In percentage, the area overhead is 3.3%, and the power overhead is 2.6%. It is difficult to argue the slight increase in power and area for what benefits the implementation in this work brought, as unresolved bugs prevented the modified core from running application programs successfully. However, based on test results, not much logic is needed to resolve the bugs. As argued in section 7.1.3, adding instruction queues alongside the implementation presented in this work can tap into performance potential. From this perspective, it is a trade-off between power, area, and future potential. The small overhead in power and area also leaves

Table 7.5: Execution log from csr_instr_asm on the baseline CV32E40X

TIME	CYCLE	ORDER	INSTRUCTION
528.000	170	1	c.j
546.000	176	2	sub
552.000	178	3	addi
558.000	180	4	addi
567.000	183	5	lw
573.000	185	6	addi
579.000	187	7	c.lw
582.000	188	8	addi
594.000	192	9	lw
597.000	193	10	addi
600.000	194	11	c.lw
603.000	195	12	addi
606.000	196	13	c.lw
612.000	198	14	sll
615.000	199	15	slli
621.000	201	16	lw
627.000	203	17	addi
633.000	205	18	lw
639.000	207	19	addi
648.000	210	20	lw
654.000	212	21	lui
657.000	213	22	c.li
660.000	214	23	c.sw
663.000	215	24	addi
669.000	217	25	c.sw
675.000	219	26	addi
681.000	221	27	c.sw
684.000	222	28	addi
690.000	224	29	c.sw
693.000	225	30	addi
699.000	227	31	c.sw
702.000	228	32	addi
708.000	230	33	c.sw
711.000	231	34	addi
717.000	233	35	c.sw
720.000	234	36	addi
726.000	236	37	c.sw
729.000	237	38	addi
735.000	239	39	c.sw
738.000	240	40	addi
747.000	243	41	c.sw
750.000	244	42	c.li
756.000	246	43	c.sw
762.000	248	44	c.sw
765.000	249	45	lui
768.000	250	46	addi
771.000	251	47	lui
777.000	253	48	addi
786.000	256	49	sw

Table 7.6: Execution log from csr_instr_asm on the modified CV32E40X

TIME	CYCLE	ORDER	INSTRUCTION
528.000	170	1	c.j
546.000	176	2	sub
552.000	178	3	addi
558.000	180	4	addi
567.000	183	5	addi
573.000	185	6	addi
579.000	187	7	addi
582.000	188	8	addi
594.000	192	9	addi
600.000	194	10	addi
603.000	195	11	addi
606.000	196	12	addi
612.000	198	13	sll
615.000	199	14	slli
621.000	201	15	slli
627.000	203	16	addi
633.000	205	17	addi
639.000	207	18	addi
648.000	210	19	addi
654.000	212	20	lui
657.000	213	21	c.li
660.000	214	22	c.li
663.000	215	23	addi
669.000	217	24	addi
675.000	219	25	addi
681.000	221	26	addi
684.000	222	27	addi
690.000	224	28	addi
693.000	225	29	addi
699.000	227	30	addi
702.000	228	31	addi
708.000	230	32	addi
711.000	231	33	addi
717.000	233	34	addi
720.000	234	35	addi
726.000	236	36	addi
729.000	237	37	addi
735.000	239	38	addi
738.000	240	39	addi
747.000	243	40	addi
753.000	245	41	c.li
759.000	247	42	c.li
765.000	249	43	lui
768.000	250	44	addi
771.000	251	45	lui
777.000	253	46	addi
786.000	256	47	addi

room for further optimizations in the future.

Table 7.7: Area and power results from synthesis on baseline CV32E40X

Baseline CV32E40X	
Power	0.321607mW
Area	24181.809

Table 7.8: Area and power results from synthesis on modified CV32E40X

Modified CV32E40X	
Power	0.330064mW
Area	24990.442

Notably, the modified design successfully met the timing constraints with a positive slack of 34, leaving room for potential improvements or optimizations in terms of power consumption, area utilization, or performance. The reason for this is that with positive slack, the design has additional time available before the signal needs to stabilize, allowing for potential power-saving measures, or it may be possible to combine or share resources to reduce area footprint, or the designers can use the additional time available to optimize critical paths to increase performance.

Chapter 8

Conclusion

This thesis aimed to enable the CV32E40X core to parallelize memory operations with all other operations and evaluate its performance, considering the trade-offs in area and power consumption. The introduction highlighted the importance of improving processor cores to meet the growing demands of modern applications. This work implements a skid buffer to impose variable delay for the LSU and apply backpressure to the OBI interface. In addition, modifications in the ID stage, EX stage, WB stage, and controller have been done to divide the EX stage into two stages, allowing for parallel execution. An instruction ID tracker has also been implemented to maintain program order when committing instructions.

It was noted in chapter 7 that the modified CV32E40X core design had several issues that hindered its functionality and overall performance. The coremark and hello-world programs exhibited lower IPC and encountered errors during the self-check phase, raising questions about the validity and accuracy of the performance assessment. However, the performance of the ricv_arithmetic_basic_test_1 program was comparable in both the baseline and modified designs, demonstrating that the problems mainly affected the LSU rather than the execution of arithmetic operations.

Potential causes for the observed difficulties were identified through bug analysis, including possible issues with ready signals in the WB stage, improper handling of split memory operations, overflow in pending transactions, and problems with the instruction log. These elements were potentially part of the redesigned CV32E40X reduced functionality and performance. For coremark and hello-world, the baseline is 12.5% and 3.1% better in terms of IPC, respectively.

Power, area, and timing analysis revealed that the redesigned CV32E40X utilized more area than the baseline while consuming more power. A small overhead of 2.6% and 3.3% in power and area However, determining the actual advantages of the changes was difficult because of the remaining bugs. It was argued that fixing the problems and putting instruction queues in place could unlock the performance potential and make the power and space trade-offs acceptable.

Despite the difficulties, the updated design met the timing requirements with positive slack, which opened the door for later power consumption, space usage, and performance enhancements. This extra time can be used to implement resource sharing, optimize essential pathways, or take power-saving measures. It gives designers more room for exciting ideas for future extensions of the presented redesign of CV32E40X.

8.1 Future Work

While this thesis has made some progress in enabling parallel execution between memory operations and the rest of the operations, several avenues for future work and improvements should be explored.

Bug Resolution is the foremost priority for future work. The bugs encountered during the implementation and evaluation of the modified design need to be addressed. Identifying and resolving issues related to ready signals, split memory operations, outstanding transaction overflow, and instruction log tracking would significantly enhance the functionality and accuracy of the core.

Further performance optimizations can be explored once the bugs are resolved. For instance, implementing instruction queues in the WB stage can help mitigate stalls and improve overall performance, as suggested in the discussion. Investigating other potential areas for performance enhancement, such as optimizing critical paths and exploring power-saving measures, would also be valuable.

Conducting thorough testing on a broader range of benchmarks and real-world applications would provide a more comprehensive evaluation of the modified CV32E40X core. Utilizing a broader range of workloads would validate the performance of the modified CV32E40X, identify potential bottlenecks or issues specific to different workloads, and ensure its reliability and compatibility across various use cases.

Exploration of Advanced Features: Future work could explore integrating advanced features and extensions into the modified CV32E40X core, for example, extending the CV32E40X core into a fully LSC. It would be interesting to compare the performance, area, and power results with those presented in the paper by Carlson et al. [30]. and other state-of-the-art sOoO[31; 32]. In addition, analyzing how effective the LSC would be for embedded applications are some ideas proposed in this thesis on the way forward for this work.

Bibliography

[1] G. Karsai, F. Massacci, L. Osterweil, and I. Schieferdecker, "Evolving embedded systems," *Computer*, vol. 43, no. 5, pp. 34–40, 2010.

[2] A. Ometov, O. Chukhno, N. Chukhno, J. Nurmi, and E. S. Lohan, "When wearable technology meets computing in future networks: a road ahead," in *Proceedings of the 18th ACM International Conference on Computing Frontiers*, 2021, pp. 185–190.

[3] S. Furber, "Microprocessors: the engines of the digital age," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 473, no. 2199, p. 20160893, 2017.

[4] F. H. Khan, M. A. Pasha, and S. Masud, "Advancements in microprocessor architecture for ubiquitous ai—an overview on history, evolution, and upcoming challenges in ai implementation," *Micromachines*, vol. 12, no. 6, p. 665, 2021.

[5] Z. Fu, "The integration of communication technology and wireless communication in the internet of things," in *7th International Conference on Education, Management, Information and Mechanical Engineering (EMIM 2017).* Atlantis Press, 2017, pp. 318–322.

[6] D. Dobberpuhl, "The design of a high performance low power microprocessor," in *Proceedings of 1996 International Symposium on Low Power Electronics and Design.* IEEE, 1996, pp. 11–16.

[7] P. Babulal and A. Mankodia, "Embedded system development trends," 12 2006.

[8] P. P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *2001 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC (Cat. No. 01CH37177).* IEEE, 2001, pp. 22–25.

[9] R. Cates, "Processor architecture considerations for embedded controller applications," *IEEE Micro*, vol. 8, no. 3, pp. 28–38, 1988.

[10] "About risc-v," https://riscv.org/about/, accessed: 2022-11-14.

[11] "Openhw group, about us," https://www.openhwgroup.org/about-us/, accessed: 2022-10-29.

[12] "Core-v family of open-source risc-v cores," https://github.com/openhwgroup/core-v-cores, accessed: 2022-10-29.

[13] F. Faggin, "How we made the microprocessor," *Nature Electronics*, vol. 1, no. 1, pp. 88–88, 2018.

[14] A. González, "Trends in processor architecture," *Harnessing Performance Variability in Embedded and High-performance Many/Multi-core Platforms: A Cross-layer Approach*, pp. 23–42, 2019.

[15] J. L. Hennessy and D. A. Patterson, "Computer architecture: a quantitative approach." Elsevier, 2011, p. 52.

[16] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[17] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: recording microprocessor history," *Communications of the ACM*, vol. 55, no. 4, pp. 55–63, 2012.

[18] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[19] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual design automation conference*, 2007, pp. 746–749.

[20] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[21] J. E. Smith, "A study of branch prediction strategies," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 202–215.

[22] D. A. Patterson and C. H. Sequin, "Risc i: A reduced instruction set vlsi computer," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 216–230.

[23] G. Radin, "The 801 minicomputer," in *Proceedings of the first international symposium on Architectural support for programming languages and operating systems*, 1982, pp. 39–47.

[24] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "Mips: A microprocessor architecture," *ACM SIGMICRO Newsletter*, vol. 13, no. 4, pp. 17–22, 1982.

[25] F. Faggin, "The intel 4004 cpu-on-a-chip was developed under pressure on an extremely tight schedule—and it worked." *IEEE SOLID-STATE CIRCUITS MAGAZINE*, vol. 1943, no. 0582/09, p. 9, 2009.

[26] D. M. Harris and S. L. Harris, “7 - microarchitecture: With contributions from matthew watkins,” in *Digital Design and Computer Architecture (Second Edition)*, second edition ed., D. M. Harris and S. L. Harris, Eds. Boston: Morgan Kaufmann, 2013, pp. 370–473. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123944245000070

[27] M. B. Taylor, “Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse,” in *Proceedings of the 49th annual design automation conference*, 2012, pp. 1131–1136.

[28] I. Stamelakos, S. Xydis, G. Palermo, and C. Silvano, “Variation-aware voltage island formation for power efficient near-threshold manycore architectures,” in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014, pp. 304–310.

[29] M. Horowitz, “Scaling, power, and the future of cmos technology,” in *2008 Device Research Conference*. IEEE, 2008, pp. 7–8.

[30] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, “The load slice core microarchitecture,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 272–284.

[31] R. Kumar, M. Alipour, and D. Black-Schaffer, “Freeway: Maximizing mlp for slice-out-of-order execution,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 558–569.

[32] K. Lakshminarasimhan, A. Naithani, J. Feliu, and L. Eeckhout, “The forward slice core microarchitecture,” in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 361–372.

[33] K. A. Andrew Waterman, “The risc-v instruction set manual,” *Volume I: Unprivileged ISA, version: 20191213*, vol. 1, 2019.

[34] “Risc-v specification,” https://riscv.org/technical/specifications, accessed: 2023-01-24.

[35] “What is risc,” https://www.arm.com/glossary/risc, accessed: 2023-01-24.

[36] “Risc-v assembler reference,” https://michaeljclark.github.io/asm.html, accessed: 2023-02-04.

[37] “Arm developer suite assembler guide,” https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference, accessed: 2023-02-10.

[38] intel, “Intel® 64 and ia-32 architectures software developer’s manual.” intel, 2022.

[39] G. GOOSSENS, J. VAN PRAET, D. LANNEER, W. GEURTS, A. KIFLI, C. LIEM, and P. G. PAULIN, “Embedded software in real-time signal processing systems: Design technologies manuscript received february 1, 1996; revised december 2, 1996. publisher item identifier s 0018-9219(97)02051-3.” in *Readings*

in Hardware/Software Co-Design, ser. Systems on Silicon, G. De Micheli, R. Ernst, and W. Wolf, Eds. San Francisco: Morgan Kaufmann, 2002, pp. 433–451. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558607026500399

[40] "27 isa extension naming conventions," https://five-embeddev.com/riscv-isa-manual/latest/naming.html, accessed: 2023-02-19.

[41] "Risc-v exchange," https://riscv.org/exchange, accessed: 2023-02-28.

[42] "Openhw group core-v cv32e40s risc-v ip," https://github.com/openhwgroup/cv32e40s, accessed: 2023-01-21.

[43] "Openhw group core-v cv32e40x risc-v ip," https://github.com/openhwgroup/cv32e40x, accessed: 2023-01-16.

[44] "Pipeline details," https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/pipeline.html, accessed: 2023-03-14.

[45] "Load-store-unit," https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/load_store_unit.html, accessed: 2023-02-10.

[46] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 172–181, 2000.

[47] "Strategies for pipelining logic," https://zipcpu.com/blog/2017/08/14/strategies-for-pipelining.html, accessed: 2023-03-12.

[48] J. Teifel and R. Manohar, "Highly pipelined asynchronous fpgas," in *proceedings of the 2004 ACM/SIGDA 12th International symposium on field programmable gate arrays*, 2004, pp. 133–142.

[49] "Core-v cv32e40x user manual," https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/intro.html, accessed: 2022-10-10.

[50] "core-v-verif," https://github.com/openhwgroup/core-v-verif, accessed: 2022-09-23.