Developing a Hardware Accelerator for Topology Optimization

Kasper Juul Hesse Rasmussen
s183735@student.dtu.dk
Technical University of Denmark

June 18, 2021

Abstract

Topology optimization (TO) is the problem of designing an optimized structure which is able to withstand a set of forces while still being as light as possible. The methods used to solve TO problems are also of great interest to many other engineering disciplines. However, the large linear systems required to solve TO problems makes them computationally expensive.

Many attempts have been made to accelerate TO problems with more efficient software or hardware. Performance gains from software improvements are limited, and while efficient use of hardware resources such as Graphics Processing Units (GPUs) shows definite improvements, properly utilizing these resources is difficult.

Hardware accelerators are integrated circuits that have been designed specifically to solve a given problem faster than software running on a computer's processor. No attempts have so far been made to design a hardware accelerator specifically for topology optimization. The purpose of this project is therefore to design a hardware accelerator to solve TO problems, to give an indication as to whether further research into this area should be conducted.

This thesis presents a hardware accelerator for topology optimization, implemented as an Application-Specific Instruction Set Processor (ASIP). An Instruction Set Architecture (ISA) for the accelerator is defined, and the accelerator has been implemented using the Chisel hardware construction language. The accelerator is found to be 4.2-6.3x faster at computing matrix-vector products than a modern laptop computer, proving that hardware acceleration of topology optimization is a promising research venue.

Contents

1. Introduction

Topology optimization is the problem of designing a structure which is as stiff as possible, making it able to withstand the forces it is subjected to, while minimizing the material usage in the structure. TO problems are formulated as large linear systems with potentially billions of unknowns, making them very time consuming to solve. Finding new ways of accelerating TO problems is therefore of great interest to researchers.

A lot of research has been conducted on ways to increase the speed at which TO problems are solved. Some improvements can be found in writing more efficient software for solving the problems, but the performance gains are not as large as those achieved with better hardware utilization. Properly utilizing hardware resources, on the other hand, is far more difficult than writing better software.

Hardware acceleration is one of the most promising ways of achieving better performance in a vast array of computable problems. Whereas a computer's central processing unit (CPU) is designed to be relatively efficient at all tasks, a hardware accelerator is designed with a specific purpose in mind, allowing it to achieve much higher performance for that exact problem than a CPU. Application-specific Instruction Set Processors (ASIPs) are one kind of hardware accelerator, which aim to strike a balance between the higher performance of a hardware accelerator and the easily programmable interface of an ordinary processor.

This thesis introduces an ASIP for topology optimization, designed to accelerate TO problems using the Solid Isotropic Material with Penalization (SIMP) approach. An instruction set architecture for the accelerator is defined, and an ASIP implementing the ISA is designed using the Chisel hardware construction language, a modern alternative to hardware description languages such as Verilog and VHDL. To the best of my knowledge, this is the first attempt at performing topology optimization using a custom hardware accelerator implemented on an FPGA.

In chapter 2, a review of existing literature on the topic of topology optimization and hardware acceleration is presented, followed by an introduction to the technical background required to read this thesis, presented in chapter 3. Chapter 4 presents the design of the ASIP, and chapter 5 presents the implementation details of the design. In chapter 6 the results of the project are presented. A conclusion is drawn in chapter 7, and future research possibilites are outlined in chapter 8.

No previous knowledge of topology optimization is assumed of the reader. It is assumed that the reader has some knowledge in the area of digital design and computer architecture. If this is not the case, a brief introduction of relevant concepts is given in section 3.4.

2. Related work

Since topology optimization problems may involve billions of design variables [1], considerable effort has been made to reduce the computational cost of solving optimization problems.

In [2], they outline the most important areas where improvements may be found. This includes algorithmic optimizations, where alternative algorithms are used to achieve results which are close to those obtained with other methodologies but take considerably less time to compute. In [3], they are able to cut computation time in half by using a heuristic to reduce the number of design variables on each iteration. Choosing the correct parameters for their heuristic is however a difficult process.

Another approach to software optimization includes using a layered/multigrid approach. This involves first solving the problem at a coarse resolution, and subsequently using that result as the starting point for solving the problem at finer resolutions [2, 4]. Using this approach, design iterations at finer resolution take much less time to complete than the solver was started at this resolution.

When solving the linear system defined for a given problem, several methods have been proposed for accelerating the solving process. [5] shows that a multigrid approach is also suitable when using the Preconditioned Conjugate Gradient (PCG) method to solve the linear system, and [6] shows that the choice of preconditioner can have a significant effect on the time taken to solve a given system.

The speed of convergence when solving the linear system is also affected by the choice of hardware and how this hardware is utilized. Graphics Processing Units are better suited for parallel computations than desktop CPUs, and in [7] they are capable of achieving up to 20x speedup using multiple GPUs on a matrix-free finite-element problem, as opposed to performing topology optimization using an 8-core Xeon processor. The matrix-free implementation requires less memory bandwidth than matrix-based implementations, making it less likely that memory transfers become the bottleneck. Likewise, in [8], they obtain a 3x-8x speedup when using GPUs, but also remark that properly sharing the workload across all processing nodes is a challenging task, which only becomes more complicated as additional nodes are added to the system.

Modern CPUs also support parallel dataflows, and using multiple CPUs at the same time allows for increased performance. In [9] it is shown that by using the OpenMP Application Programming Interface for writing data-parallel programs, a 2-5x speedup can be achieved on design problems with less than 20,000 elements.

A large amount of work has gone into designing accelerators for matrix-matrix and matrix-vector problems. Modern GPUs are one such accelerator, specially designed for the problems that arise in image and graphics processing [10]. Work has also gone into designing efficient matrix processors for FPGAs. In [11], the authors propose a design for a matrix processor optimized for FPGAs which operates on 18-bit fixed point numbers. This design uses a systolic ring structure to pass the intermediate result from one processing stage to the next and is structured in a way that makes transposing matrices trivial.

[12, 13] propose another design for a matrix processor. Instead of computing a full matrix product at once, they utilize a limited number of Processing Elements (PEs) and block matrix multiplication to compute results, storing intermediate results in each processing element. This reduces the need for communication between PEs, which allows for a smaller implementation.

In [14], it is noted that the future of efficient computing is that of customizable processors and application-specific instruction set processors, as power efficiency, and not just raw computing power, is becoming an increasingly important metric when comparing processors. This also makes it easier to target a hardware accelerator than if GPUs are involved, as GPU programming is notoriously difficult [16].

3. Technical background

In this section, an introduction to the technology used in this project is presented. An introduction to the field of topology optimization and the preconditioned conjugate gradient method is given. Next, different systems for storing rational numbers in computers are presented, and finally an introduction to computer architecture, hardware acceleration and FPGAs is given.

3.1 A note on style

Throughout the remainder of this paper, numbers may be written in either binary or decimal, depending on the context.

Binary values are indicated by prepending `0b` to a value. Decimal values will not be explicitly indicated, as all numbers without a prefix should be considered to be decimal values. As such, the following conversion holds

$$0\text{b}100011010010 = 2258$$

As seen above, unless otherwise indicated, all binary values should be interpreted as bitstrings reprenting unsigned numbers. The leftmost bit is always the most significant bit (MSB), and the rightmost bit the least significant digit (LSB).

When writing equations, bold capital letters $\mathbf{M}$ represent matrices, bold lowercase letters $\mathbf{v}$ represent vectors and non-bolded letters are scalars.

3.2 Topology optimization

Topology optimization in the field of mechanics is the problem of optimizing the use of material for a given structure subjected to a set of forces, in order to minimize/maximize a given performance measure. By increasing the amount of material in some sections of the design domain and reducing the amount of material in other sections, an optimized structure can be found. In this case, "optimized" means that the amount of material used in the structure is minimal, while the structure is still able to withstand the forces it is subjected to.

For this project, the TO problem is approached as outlined in [17], where a number of simplifications are used. The design domain is assumed to be a rectangular box, and it is discretized into a number of equally sized cubes. Each of these cubes (and thus, the entire design domain), is assumed to be made of the same material (or no material).

Topology optimizations using the SIMP approach may be formulated according to eq. (3.1). Note that much of the this section is a summary of the method outlined in [17].

$$\begin{cases} \min\limits_{\mathbf{x}} : c(\mathbf{x}) = \mathbf{u}^\mathsf{T}\mathbf{K}\mathbf{u} = \sum\limits_{e=1}^{N}(x_e)^p \mathbf{u}_e^\mathsf{T}\mathbf{K}_e\mathbf{u}_e \\ \text{subject to :} \quad \dfrac{V(\mathbf{x})}{V_0} = f, \\ \qquad\qquad\quad \mathbf{K}\mathbf{u} = \mathbf{f}, \\ \qquad\qquad\quad \mathbf{0} < \mathbf{x}_{min} \leq \mathbf{x} \leq \mathbf{1} \end{cases} \tag{3.1}$$

Where the variables are given by the following:

- $\mathbf{u}$ is the global displacement vector
- $\mathbf{f}$ is the global force vector

- $\mathbf{K}$ is the global stiffness matrix
- $\mathbf{u}_e$, $\mathbf{K}_e$ are the local (per-element) displacement vector and stiffness matrices, respectively
- $\mathbf{x}$ is the vector of design variables / element densities
- N is the number of elements used to discretize the design domain
- p is a penalty factor, commonly set to 3
- $\mathbf{x}_{min}$ is a vector of minimum values for $\mathbf{x}$, all of which are nonzero to avoid singularities
- $V(\mathbf{x})$ is the material volume and V_0 is the volume of the design domain
- f is the desired volume fraction (percentage of the design domain which should be taken up by material once finished)
- $c(\mathbf{x})$ is the compliance function which must be minimized

Minimizing the compliance function $c(\mathbf{x})$ is not done in a single pass, as the system may have billions of unknowns. Instead, an iterative approach is used where the optimized values for $\mathbf{x}$ are slowly converged upon. In each design iteration, the value of $\mathbf{x}$ is updated to approach the optimized value. For small problems ($N < 10000$), 100 to 150 design iterations are necessary. An example of a TO problem being solved can be seen in fig. 3.1.

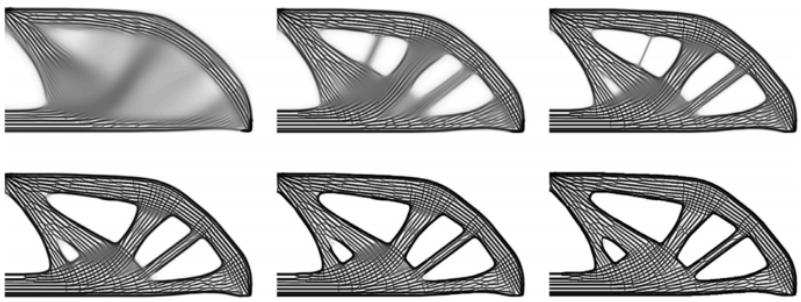

Figure 3.1: The iterative process of topology optimization. The structure starts as a homogenous design domain (not shown), and slowly approaches the optimized structure (bottom right). From [18], CC BY 4.0, `https://creativecommons.org/licenses/by/4.0/`. No modifications have been made to the figure.

On each design iteration, after the values of $\mathbf{x}$ have been updated, a filtering process must be run on the design domain. For this project, a density filter is applied, which sets the element density of all elements in the design domain as a weighted average of the element and all neighbouring element's densities. This ensures that no elements are left without supports in the design domain, as well being a necessary step in solving the optimization problem.

It should be noted that, when using the SIMP approach, the problem of solving $\mathbf{u}^\mathsf{T}\mathbf{K}\mathbf{u}$ from eq. (3.1) reduces to operating only on the 24×24 $\mathbf{K}_e$ matrix. Multiplying the scaled element density and the local stiffness matrix allows us to forgo storing the large, usually sparse, $\mathbf{K}$ matrix. This greatly reduces the memory bandwidth necessary for computation, since all elements in the design-domain have the same stiffness matrix.

For more information about the updating scheme and more information on topology optimization, the reader is referred to [17].

3.2.1 The preconditioned conjugate gradient method

The preconditioned conjugate gradient method is a method for solving linear systems of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ and $\mathbf{b}$ are known, and the vector $\mathbf{x}$ is being solved for. This method is used when the matrix $\mathbf{A}$ is symmetric and positive-definite. [1]

The PCG method is an iterative method which attempts to approximate the solution $\mathbf{x}$ with an initial guess, and for each iteration of the method arrives at a better approximation. Once enough

[1] A matrix is positive-definite if, for any non-zero vector $\mathbf{x}$, the scalar $s = \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}$ is strictly greater than zero.

iterations have been performed, the residual $\epsilon = \frac{|\mathbf{r}_k|}{|\mathbf{b}_k|}$ will be below a chosen threshold (usually on the order 10^{-5} to 10^{-10}), and the system is considered to be solved.

In this project, the preconditioned conjugate gradient method is used to solve the matrix equation $\mathbf{Ku} = \mathbf{f}$. The steps to be taken are as given in [19]:

$$\text{Set } \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0,\ \mathbf{z}_0 = \mathbf{P}^{-1}\mathbf{r}_0,\ \mathbf{p}_0 = \mathbf{z}_0 \tag{3.2}$$

$$\alpha_k = \frac{\mathbf{r}_k^\mathsf{T}\mathbf{z}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k} \tag{3.3}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k \tag{3.4}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k \tag{3.5}$$

$$\mathbf{z}_{k+1} = \mathbf{P}^{-1}\mathbf{r}_{k+1} \tag{3.6}$$

$$\beta_k = \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{z}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{z}_k} \tag{3.7}$$

$$\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k\mathbf{p}_k \tag{3.8}$$

Where $\mathbf{r}$, $\mathbf{z}$, $\mathbf{p}$, α, β are used when performing the PCG, but are not otherwise used in TO. The value k is the current iteration.

$\mathbf{P}$ is the preconditioning matrix which can be chosen in a number of different ways [20]. A preconditioner is used to increase the speed at which the solution converges. In this case, the preconditioning matrix is the Jacobi preconditioner, the diagonal of the matrix $\mathbf{A}$. In addition to increasing the speed of computation, being a diagonal matrix makes it trivial to invert $\mathbf{P}$, requiring less computation. A number of other preconditioners also exist [19], but the Jacobi preconditioner has been chosen due to the ease of computation.

Obtaining the satisfied resolution ϵ may require thousands of iterations of the PCG loop before the algorithm terminates. In [8] it is noted that more than 90% of the time spent solving a particular optimization problem is spent solely on solving the equation $\mathbf{Ax} = \mathbf{b}$.

3.3 Representing rational numbers

As computer memory is not an unlimited resource, it is not possible to represent all numbers with a finite number of binary digits. Instead, an approximation to any real number must be made using rational numbers. Two different approaches to this problem exist: floating-point numbers and fixed-point numbers. A floating-point number is stored in a representation resembling scientific notation. To represent a number, a fractional part (called the significand) is multiplied by the base raised to some power. Fixed-point numbers are represented by applying a scaling factor to all integers, such that a set of rational numbers are also representable.

3.3.1 Floating point numbers

Floating point numbers are a way of expressing a (very large) subset of the real numbers. A floating-point number x is uniquely identified by 4 values, the sign (S), exponent (E), significand (F) and base (b) by the following expression:

$$x = (-1)^S F \cdot b^E \tag{3.9}$$

Most modern computers implement the IEEE 754 floating-point standard [21]. To increase the resolution of floating-point numbers, the significand of all IEEE floating-point numbers is stored in normalized form where the significand is of the type $1.xxxx$. An exception to this, called subnormal numbers, exists, but is not relevant for the further discussion.

Since all significands have a 1 before the decimal separator, this digit can be stored implicitly. Instead of storing the significand, the mantissa (M), which is the fractional part of the significand, is stored. Thus, an IEEE-conformant float is a 32-bit value, consisting of 1 sign bit, 8 exponent bits and 23 mantissa bits. An IEEE-conformant double is a 64-bit value, consisting of 1 sign bit, 11 exponent bits and 52 mantissa bits

To increase performance, the exponents of IEEE floating-point numbers are stored in biased notation, where a constant offset is applied to all values. A binary floating-point number with bias

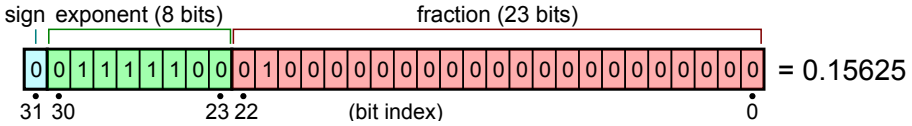

Figure 3.2: Illustration of the binary representation of a 32-bit IEEE floating-point number. From Wikipedia.com / Stannered, CC BY-SA 3.0, `https://creativecommons.org/licenses/by-sa/3.0/`

B is thus expressed as

$$x = (-1)^S(1 + M)2^{(E-B)} \tag{3.10}$$

By using biased notation, it is not necessary to use two's complement to store the exponents, which makes it easier to sort floating-point numbers. The bias used for e.g. doubles is 1023, such that the exponent 1 is stored as $1 + 1023 = 1024 = 0b100000000000$. All positive exponents thus have the MSB of the exponent field set, making it trivial to separate numbers with positive and negative exponents.

Using a bias, the largest exponent which may be represented for a double is $2^{1023} \approx 10^{308}$, and conversely the smallest possible value is $2^{-1022} \approx 10^{-308}$. Using 53 bits for the significand, doubles are accurate to about 16 decimal digits ($52 \log_{10}(2) = 15.65$), vs 7 digits for floats.

Floating-point numbers allow for a very large range of expressible values. This comes at the cost of increased complexity when performing arithmetic operations on floating-point numbers. As they are stored in a sign-magnitude representation and use separate fields for different parts of the number, an ordinary addition circuit is not able to add floating-point numbers. While the process readily lends itself to pipelining, it is more costly in terms of time and area than performing like operations on fixed-point numbers [10].

3.3.2 Fixed point numbers

Fixed-point numbers are another way of expressing rational numbers. While it does not have the same range of expressible values as floating-point numbers, it instead offers arithmetic operations which can be implemented with ordinary arithmetic circuits.

Fixed-point numbers are commonly expressed in the Q number format in the manner Q$n.m$ [22]. The value n indicates how many bits are used to express the integer part of a value, and m indicates how many bits are used for the fractional part. They are stored in the same manner as integers, and the mathematical operations are the same as when working with integers. An unsigned fixed-point number requires $n + m$ bits of storage. A signed fixed-point value will be indicated by the prefix Qs and requires $n + m + 1$ bits.

A signed N-bit binary number B can be expressed as a sum of powers of 2

$$B = \{b_{N-1}, b_{N-2}, \ldots, b_1, b_0\} = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i \tag{3.11}$$

where b_n is the n'th digit of the binary number. The fixed-point number B_f with the same bit representation is obtained by applying a scaling factor s to the summation.

$$B_f = s\left(-b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i\right) \tag{3.12}$$

The scaling factor s and the value m are linked by the relationship $s = 2^{-m}$ [23].

As an example, the Qs2.4 number 0b1100111 should be interpreted as a number with the binary point 4 positions to the left of the LSB, 0b110.0111. The scaling factor S is then $S = 2^{-4} = \frac{1}{16}$, and the number can be interpreted as

$$0b110.0111 = \frac{1}{16}\left(-1 \cdot 2^6 + 2^5 + 2^2 + 2^1 + 2^0\right) \tag{3.13}$$

$$= \frac{1}{16}(-25) = -1.5625 \tag{3.14}$$

A decimal number X can easily be converted to its corresponding fixed-point representation simply by multiplying with 2^m and rounding to the nearest whole number. The bitstring of that integer corresponds to the fixed-point value. Conversely, the decimal value that a fixed-point number represents can be found by interpreting the fixed-point number as an integer and then dividing by 2^m.

For example, 2.345 is expressed as a Qs2.4 number by muliplying with 16 and rounding to the nearest integer.

$$2.345 \cdot 2^4 = 37.52 \approx 38 = 0\text{b}0100110 \tag{3.15}$$

When interpreting this bitstring as a fixed-point number, the decimal value obtained is

$$\frac{1}{16}\left(2^5 + 2^2 + 2^1\right) = \frac{32 + 4 + 2}{16} = \frac{38}{16} = 2.375 \tag{3.16}$$

As it is seen, the resulting fixed-point value is not the exact same as we started with, introducing an error of $|2.345 - 2.375| = 0.03$, a deviation of approximately 1.28% from the original value. Using more bits for the fractional part of the number would obviously yield a better approximation.

For this project, Qs15.38 numbers are used, which grant a precision of $38 \log_{10}(2) \approx 11$ decimal digits. With 15 integer bits and a sign bit, numbers in the range $[-32768.0; 32768.0[$ may be expressed in increments of 2^{-38}, which should be large and precise enough for this project. A large portion of each value has been assigned to the fractional part for increased precision when performing arithmetic operations.

This choice of fixed-point number was also motivated by the fact that a Qs15.38 number requires 54 bits to be expressed. The FPGA used for this project has a number of 27×27 multipliers available in its DSP blocks. By choosing a bitwidth which is a multiple of 27, no multiplier functionality is wasted, and the loss in precision vs. operating with e.g. Qs15.48 (64-bit values) is minimal.

3.4 Computer architecture

All modern computers are built on nanometer-size transistors, which are used to implement logic gates. A logic gate takes binary input signals, which can take on one of two values. They can either be a logic 1 (also known as a true or high signal) or a logic 0 (also known as a false or low signal). An example of a logic gate is the AND gate, the output of which is only high if all inputs to the gate are high.

The circuits that can be created with logic gates are generally separated into two categories. Combinational (asynchronous) circuits take a number of inputs and generate a number of outputs. These circuits do not have any notion of time, and also do not have the ability to store values for later reuse.

The other type of circuits are sequential (synchronous) circuits. In a sequential circuit, a register may be used to store values generated by combinational logic. A global signal called the clock is used to trigger the registers in a circuit. Whenever the clock transitiitions from a low to a high value, the value on the D port of a register is copied onto the output Q port. This output is kept constant until the next tick of the clock, no matter how the input to the register is changed.

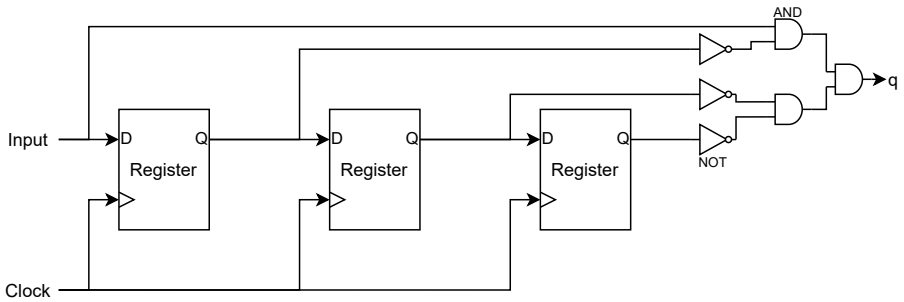

Figure 3.3: An example of a sequential circuit where the output q is only high when the current input is a 1 and the input was 0 on the three previous clock cycles. An AND gate and a NOT gate have been indicated. The NOT gate turns a logic 1 into a logic 0 and vice versa.

An example of a sequential circuit is shown in fig. 3.3. The output q of this circuit is only high if the current input is a 1, and the 3 previous inputs were all 0's.

Since signals take time to propagate, any sequential circuit has a maximum clock frequency at which it can operate. If a circuit is run at too high a frequency, values may not arrive at registers in time before the clock triggers the register, which may cause the system to malfunction. To operate a circuit at a higher clock frequency, pipelining is necessary. By inserting registers at strategic locations, pipeline stages are created, allowing the circuit to operate at a higher frequency than if the circuit were not pipelined.

An example of a pipelined addition circuit is shown in fig. 3.4, where the lower 8 bits of two inputs are added in the first pipeline stage, and the upper 8 bits are added to the result from the previous pipeline stage on the next clock cycle.

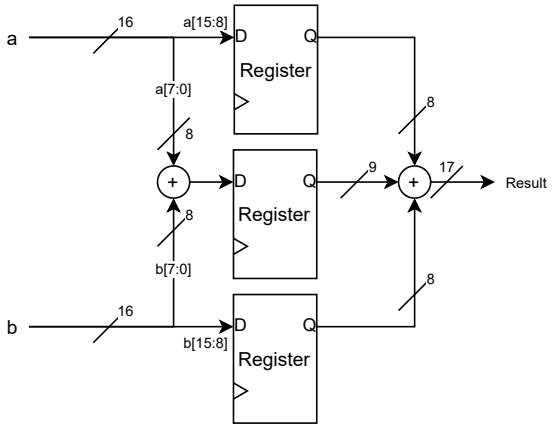

Figure 3.4: A pipelined 16-bit adder, calculating the result over two clock cycles instead of one. The lower 8 bits are added in the first stage, and the upper 8 bits in the second stage. The clock signal has been omitted to make the diagram clearer. The diagonal lines indicate the bitwidth of a signal.

To evaluate a hardware design, it may be implemented on a Field Programmable Gate Array (FPGA). An FPGA consists of thousands of Logic Elements (LEs), which are built from registers and lookup tables (LUTs). Instead of building the circuit from the basic logic gates, all possible input combinations for a given expression are enumerated, and LUTs are programmed to select the correct output given the input signals. The LUTs on the FPGA are connected in such a way that the FPGA performs the same operations as if the circuit had been created with the actual logic gates. Since FPGAs are reconfigurable, they are a very fast and easy way to implement a design without manufacturing a new integrated circuit.

A circuit implementing the logical expression `(a AND b AND NOT c) OR (NOT a AND c))` is seen in fig. 3.5, implemented with logic gates on the left and on the right implemented with LUT.

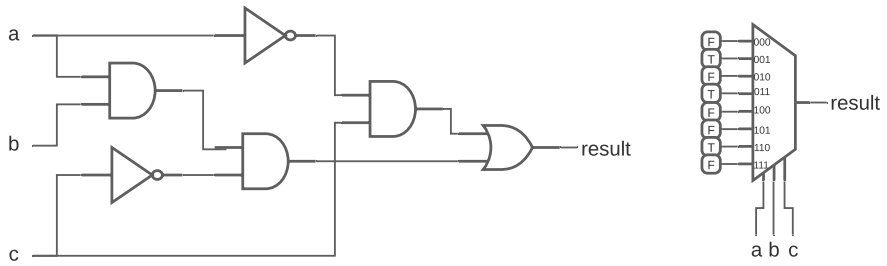

Figure 3.5: Left: A circuit implementing the expression `(a AND b AND NOT c) OR (NOT a AND C)`. The rightmost logic gate is the OR gate, the output of which is true if any one of its inputs are true. Right: A LUT implementing the same expression. Designations on the LUT are enumerated a-b-c from left to right.

In addition to having LUTs, modern FPGAs also incorporate special addition and multiplication circuitry, called hard adders and multipliers. Using these hard blocks is generally much faster

than implementing a circuit with the same functionality by using the LUTs on the FPGA [24].

The CPU in modern computers and smartphones consists of 2-8 cores, each of which is able to execute instructions independently of the other cores. CPUs are able to execute instructions at a very high speed but are not as optimized for heavily data-parallel workloads as other types of processors.

One such optimized processor is the GPU which has thousands of cores, optimized for efficiently computing data-parallel problems across multiple cores. This makes GPUs very efficient at things like matrix-vector multiplication, as each core can compute a subset of the final result, completing the task faster than if a CPU was to solve the same problem. On the other hand, GPUs are not as fast as CPUs at many other tasks, and it is difficult to write software that efficiently utilizes a GPUs resources.

For some computationally expensive applications, a hardware accelerator may be used. This is a hardware circuit specially designed to perform a specific task faster than a program running on a computer. Depending on the purpose of the accelerator, its configurability and complexity may vary. Some accelerators, such as the ones used for computing cryptographic hash functions, need not be highly configurable, as the steps to be taken when computing a hash function are always the same. On the other hand, if more complex problems are to be solved, an application-specific instruction set processor may be preferred. An ASIP is a hardware accelerator which is built on the same principles as ordinary processors, but incorporates circuitry specially designed for the problems at hand. By combining the speedup of a hardware accelerator with the configurability of an ordinary processor, ASIPs make it possible to design hardware accelerators for a large array of problems.

4. Design

This section introduces the design of the hardware accelerator designed for this project. First, the structure of the design domain is explained, followed by an analysis of the topology optimization program currently used at DTU Mechanical Engineering. This analysis is performed to identify critical regions which must be accelerated. Once the analysis of the program is complete, the instruction set architecture for the hardware accelerator is presented.

4.1 Design domain

The TO design domain consists of a number of cubic elements structured in a grid. The number of elements in the x, y and z-directions must be a positive integer, but the values do not have to be the same.

Associated with each element is a value between 0 and 1, representing the density of that specific element (0 is empty, 1 is solid). As pictured in fig. 3.1, elements will either have a density of 0 (white) or one (black) once the problem is solved. Each element in the grid has 8 corners, each of which has 3 degrees of freedom (DOFs) associated with it, for a total of 24 degrees of freedom for each element. In fig. 4.1 is shown an illustration of a $3 \times 3 \times 3$ design domain, with select element indices and DOF indices marked in black (elements) and red (DOFs).

When a corner is shared by two elements, they also share those degrees of freedom. For example, elements 0 and 1 share 12 degrees of freedom, elements 1 and 11 share 6 degrees of freedom, and elements 9 and 22 share 3 degrees of freedom. Elements 19 and 25 do not share any DOFs, as they do not have any corners in common.

Note that the DOF indices shown increment in multiples of 3, since each corner has 3 degrees of freedom associated with it. Only the first index associated with a corner is shown in the figure.

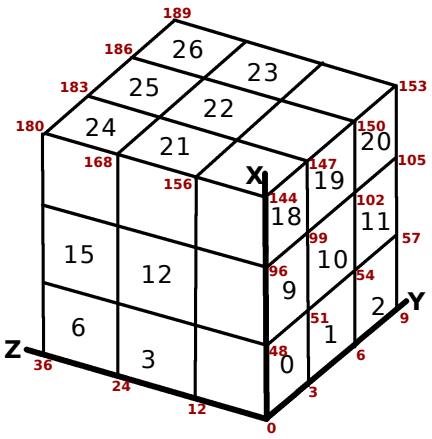

Figure 4.1: Illustration of a 3x3x3 design domain, with select element indices marked with large black numbers, and selected DOF indices marked with smaller red numbers. Only the first DOF index is shown on each corner

Two important values can be defined from the grid dimensions. Given a grid with $NELX$ element in the x-direction, $NELY$ elements in the y-direction and $NELZ$ elements in the z-direction, the total number of elements is given by

$$NELEM = NELX \cdot NELY \cdot NELZ \tag{4.1}$$

Likewise, the total number of DOFs can be calculated as

$$NDOF = 3 \cdot (NELX + 1) \cdot (NELY + 1) \cdot (NELZ + 1) = 3 \cdot NX \cdot NY \cdot NZ \tag{4.2}$$

For the example 3x3x3 grid, $NELEM = 27$ and $NDOF = 192$.

In the subsequent TO problems, all vectors either have length $NELEM$ or length $NDOF$. The vectors holding elements and their DOFs are not always traversed "gridwise". When performing some steps of e.g. the preconditioned conjugate gradient method, the vectors can be traversed without regard for their elements' spatial location in the grid.

4.2 Critical regions for ASIP design

The TopOpt research group at DTU Mechanical Engineering currently uses a MATLAB script for computing topology optimization problems. This Matlab script was ported to C by the TopOpt group before this project was started. The C-code and MATLAB script can be found in this project's GitHub repository (see appendix A.1), alongside the Chisel/Scala code developed for this project.

It was initially evident that a more complex control scheme than what is feasibly obtainable with a finite state machine (FSM) was necessary. To this end, it was decided that the hardware accelerator would be implemented as an ASIP. The initial ASIP design was guided by the methodology given in [25] and [26]. Both resources emphasize that ASIP design should start by analysing the target application, identifying critical regions of the program where the majority of execution time is spent. The critical regions of the program are the ones that will benefit the most from hardware acceleration.

According to the researchers at DTU Mechanical Engineering, the critical region of the C-code is the function `solveStateCG`, which is the function where the linear system $\mathbf{Ax} = \mathbf{b}$ is solved using the PCG. As can be seen from eqs. (3.2) to (3.8), the preconditioned conjugate gradient method relies heavily on vector-vector addition and subtraction, vector dot products and matrix-vector multiplication. These mathematical operations are thus also the operations that the ASIP must be designed to accelerate.

Of the functions called in `solveStateCG`, the function `applyStateOperator` is the main components. This is the function where a matrix-vector product $\mathbf{Kx}$ is calculated. The function `applyStateOperator` loops over all elements in the design domain, performing the matrix-vector product by utilizing the fact that only the $\mathbf{K}_e$ matrix is required when performing topology optimization using the SIMP approach. The code for this function is shown in listing 4.1, where some initial setup function calls have been omitted for brevity.

```
//MATRIXPRECISION is either 'double' or 'float', depending on desired precision
//'in' is the input NDOF long vector, 'out' is the output NDOF long vector. 'x'
    holds element densities
for (unsigned int i = 0; i < ndof; i++)
 *((MATRIXPRECISION *)out + i) = 0.0;

for (int32_t i = 0; i < gc.nelx; i++)
 for (int32_t k = 0; k < gc.nelz; k++)
  for (int32_t j = 0; j < gc.nely; j++) {

   getEdof(edof, i, j, k, ny, nz);

   const uint_fast32_t elementIndex =
    i * gc.nely * gc.nelz + k * gc.nely + j;
   const MATRIXPRECISION elementScale =
    gc.Emin + pow(x[elementIndex], gc.penal) * (gc.E0 - gc.Emin);

   for (int ii = 0; ii < 24; ii++)
   u_local[ii] = in[edof[ii]];

   // matrix-vector product: out_local = elementScale*(ke*u_local)
   cblas_dsymv(CblasRowMajor, CblasUpper, 24, elementScale,
       (MATRIXPRECISION *)ke, 24, u_local, 1, 0.0, out_local, 1);

   for (int ii = 0; ii < 24; ii++)
    out[edof[ii]] += out_local[ii];
  }
```

```
// apply boundaryConditions
struct FixedDofs fd = getFixedDof(gc.nelx, gc.nely, gc.nelz);
for (int i = 0; i < fd.n; i++)
 out[fd.idx[i]] = in[fd.idx[i]];

free(fd.idx);
```

Listing 4.1: Main section of the C function `applyStateOperator` used to calculate a matrix-vector product. From DTU Mechanical Engineering / `top.c`

Of special note in listing 4.1 is the function call `getEdof`. This function calculates the indices of the DOFs associated with a given element, which are later used to index into the array `in`. From this, a very important pattern can be identified: Looping over all elements in the design domain, retrieving the element densities and the 24 DOFs associated with an element, and multiplying a scaled version of the element density onto all 24 DOFs.

In addition to the call to `applyStateOperator`, the program also relies on two filtering functions, `applyDensityFilter` and `applyDensityFilterGradient`. Mathematically, this filtering is equivalent to a 3D convolution with the given filter coefficients. In practice, the filters are applied by looping over the neighbourhood of each element, using the element densities in neighbouring elements to calculate a new value for the central element.

In fig. 4.2 is shown the scaling factors applied to the central element (marked with a dark grey background) and the neighbouring elements when calculating the new value of an element. In the current version of the C-code, only the immediate neighbours affect the new value for the central element. While this is possible to modify in the C-code such that neighbours further away are also taken into account, it was chosen not to implement this piece of functionality in the accelerator.

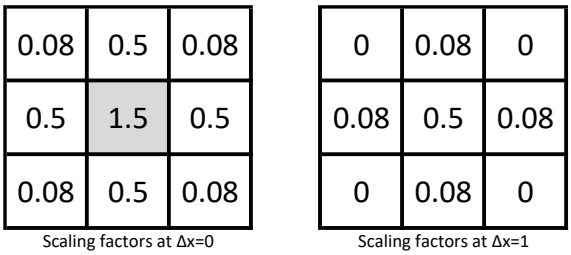

Figure 4.2: Scaling factors applied to elements when applying a density filter. All elements outside the area shown have their scaling factor set to 0.

This defines another important feature of the accelerator: Being able to retrieve the element values of all neighbouring elements and applying a scaling factor to each neighbour, depending on its distance from the central element.

Given this initial analysis of the C-code, it is possible to list some of the operations that the accelerator must support:

- Storing and retrieving elements and their associated DOFs.
- Storing and retrieving an element value as well as the element values of all neighbouring elements
- Storing and retrieving $NELEM$ and $NDOF$ long vectors without viewing the vectors as mapping onto a grid.
- Multiplying a (scaled) element value onto all associated DOF values
- Filtering a central element value with all neighbouring elements
- Performing vector-vector arithmetic operations
- Performing vector dot products
- Performing scalar arithmetic operations

4.3 Grid traversal and mapping indices

As shown in listing 4.1, the final step of the matrix-vector product is adding the calculated values in `out_local` to the values already stored at `out[edof[ii]]`. This means that operating on two neighbouring elements at the same time can cause a collision when storing values back into memory. Whichever element and its DOFs that are stored last will overwrite previous store operations that accessed the same DOF locations, storing incorrect results in memory.

Two ways of solving this problem are by using a nodal traversal of the grid or sectioning the grid by applying an 8-colouring. In a nodal traversal, the grid is traversed node (corner) by node. On each node, the eight neighbouring element's contributions to that node's new value are calculated, and the sum of those contributions is stored in the output vector. In this fashion, once a node has been processed once, the output value is sure to be the final value. This does however require an out-of-bounds check for all nodes and is more difficult to implement than the 8-colouring approach.

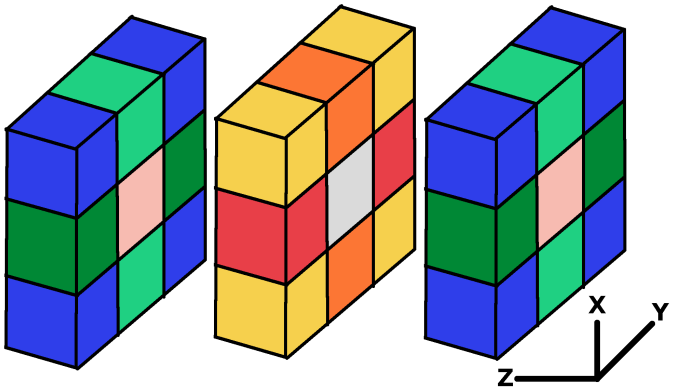

Figure 4.3: Illustrating an 8-colouring, showing three slices of the grid taken at $z = 0$, $z = 1$ and $z = 2$. The grid has been coloured such that no two elements of the same colour share any corners.

Value	Colour
0	
1	
2	
3	
4	
5	
6	
7	

Table 4.1: Mapping colours and their corresponding values

Using an 8-colouring, the grid is separated into 8 separate sections, each of which is applied a "colour" (in practice, a numeric value is given to each colour). The colouring is applied in such a way that no element touches another element with the same colour - an example is shown in fig. 4.3. This means that `applyStateOperator` may be performed on any two elements of the same colour without causing collisions, as they are sure not to share any DOFs.

For this project, it has been chosen to traverse the grid using an 8-colouring. Using an 8-colouring requires that a function like `applyStateOperator` is not implemented with a for-loop where the loop variable increments by 1 on every iteration, but instead increments by two. This ensures that all elements accessed are in the same colouring.

4.4 Instruction set design

Given that the inital requirements for the functionality of the ASIP have been identified, it is now possible to define an instruction set architecture for the processor. Four different instruction types have been defined, all of which are shown in fig. 4.4.

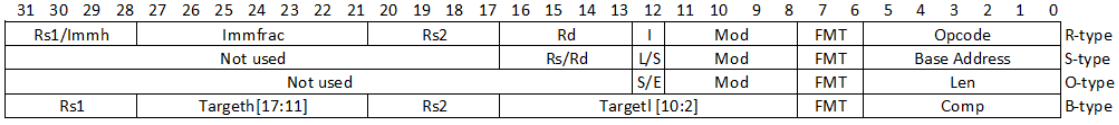

Figure 4.4: Instruction types and their bit mappings defined for the ASIP instruction set.

R-type instructions are executable instructions which access the register files, compute new data and store this data back into register files. They also support immediate values when the

immediate flag is set. S-type instructions are used for memory load/store operations. O-type instructions are used to modify the behaviour of the accelerator, and B-type instructions are used to control execution flow via branches.

The instruction format is inspired by the RISC-V ISA and tries to follow one of the guiding principles of the RISC-V ISA: Namely that all instructions have the same bitwidth, and common fields are placed at the same location in all instructions to make instruction decoding simpler [27].

Each instruction type is uniquely identified by its instruction format, found at bits [7:6] in each instruction. Table 4.2 lists the encodings for each available instruction type.

Format	Encoding
R-type	0b00
S-type	0b01
O-type	0b10
B-type	0b11

Table 4.2: Instruction formats available and their binary encoding

Before explaining how the various instruction formats are designed, it is necessary to explain how the register files for the ASIP must be implemented.

4.4.1 Register files

Since a large amount of the computations required for topology optimization are highly parallelisable, the ISA has been designed to reflect this. It utilizes three different register file types, each of which is used to store different types of data. Two vector register files named the V-register file and X-register file are used, as well as the scalar S-register file. A vast majority of the time spent on topology optimization is spent executing matrix-vector products in `applyStateOperator`. The ISA has been designed with this in mind - namely that it should be as simple as possible to load an element density and its 24 associated DOFs, as well as multiplying the `elementScale` value onto the DOFs.

To this end, the V-register file consists of a number of vector registers, each of which is 24 elements deep. These registers are then grouped into a number of vector register "slots", which are addressible from instructions. Each X-register is as many elements deep as there are elements in each vector register slot. This way, the vector at a V-register slot's position [0] holds the DOFs associated with the element value at an X-register's index [0].

The number of V-registers is denoted NUM_VREG, and the number of addressible slots is NUM_VREG_SLOTS. As it is clear from fig. 4.5, NUM_VREG_SLOTS = NUM_XREG, the number of addressible X-registers. Each V-register slot and X-register holds VREG_SLOT_WIDTH = XREG_DEPTH entries.

Figure 4.5: Illustrating how V-registers and X-registers are mapped. In this example there are 12 V-registers, and the width of each V-register slot is 3. This means that the depth of each X-register must also be 3. The number of V-register slots and X-registers need not be the same.

The scalar S-register file may have up to 16 entries, as 4 bits are used for each register select signal. The value at index 0 is tied to a logical zero to allow for immediate loading and easily clearing register contents.

4.4.2 O-type instructions

O-type instructions are used to pass configuration values into the ASIP, which modify the behaviour of the accelerator when executing upcoming instructions. The fields used in an O-type instruction are as follows:

- Len: Instruction length / increment type. See table 4.5 for possible values.
- Mod: The O-type modifier used to indicate whether the instruction encodes information about an instruction packet or executable section. See table 4.4 for possible values.
- S/E: Start/end flag: Used to indicate the start or end of an instruction packet and the executable section of a packet. See table 4.3 for possible values.

S/E	Encoding	Comments
`end`	0b0	Indicates the end of an instruction packet or executable section
`start`	0b1	Indicates the start of an instruction packet or executable section

Table 4.3: Possible O-type S/E values and their corresponding encodings

Topology optimization relies on performing the same operation on all elements of very long vectors, and the instruction set has been designed with this in mind. If e.g. $NELEM = 1000$ and 100 values could be stored in the vector register file at any one time, the naive approach would be to copy/paste 10 nearly identical instructions to loop through an entire vector. Alternatively, a loop variable could be defined, which is incremented on each iteration and is used to index into a vector.

To eliminate this need for code reuse and loop variables, the ASIP is designed such that all instructions given will act on all entries of a given vector. If one wishes to add two vectors **a** and **b**, it is only necessary to specify the vectors, the vector length, and the operation to be performed. To this end, instructions must be grouped in instruction packets, which are delimited by two O-type instructions. It should be noted that these are not the same as issue packets in Very Long Instruction Word (VLIW) processors, since the instructions in a VLIW issue packet are generally executed at the same time. The instructions in an instruction packet are executed one after the other and are not guaranteed to execute alongside other specific instructions.

Each instruction packet encodes the necessary load/store instructions and operations that should be performed on the values in the register files. Hardware registers which are not addressible from the instruction set are used to generate the correct offsets into memory when performing load/store operations.

Mod	Encoding	Comments
`packet`	0b0001	The packet modifier is used at the start and end of each instruction packet
`exec`	0b0010	The executable modifier is used to delimit the load, executable and store instructions in each packet

Table 4.4: Possible O-type modifier values and their corresponding encodings

Each instruction packet must start with the instruction `pstart`, which has the `S/E` flag `start` and the O-type modifier `packet`. All instruction packets must end with `pend`, which is again a `packet` instruction with the `S/E` flag set to `end`.

In each instruction packet, instructions are separated into three separate sections. Load, execute, and store instructions. The instructions `estart` and `eend` wrap all executable instructions,

with the load instructions placed between `pstart` and `estart`, and store instructions placed between `eend` and `pend`. Listing 4.2 shows the general structure that all instruction packets must follow.

```
pstart <len>
<load instructions>
estart
<executable instructions>
eend
<store instructions>
pend
```

Listing 4.2: The general outline of an instruction packet for the ASIP. A packet may leave out load/execute/store instructions if they are not necessary. The O-type instructions must always be present

The modifier `len` is only used when decoding a `pstart` instruction. This modifier encodes information about the length of the vectors used in the packet, and how these vectors should be traversed. The possible values for `len` can be found in table 4.5.

Length	Encoding	Comments
`ndof`	0b000000	Used when traversing NDOF long vectors directly
`single`	0b000010	Used when a packet should be executed only once. Mostly used to set up values in scalar registers
`nelemvec`	0b000100	Used when traversing NELEM long vectors directly
`nelemdof`	0b000101	Used when traversing NELEM and NDOF long vectors where element values and their corresponding DOFs should be retrieved
`nelemstep`	0b000110	Used when traversing NELEM long vectors one element at a time when density filters are applied

Table 4.5: O-type length values and their binary encodings

The modifiers `ndof` and `nelemvec` are used when element-wise operations are performed, such as vector-vector addition or scalar-vector multiplication. Modifier `ndof` specifies that the vector is NDOF elements long, whereas `nelemvec` specifies that it is NELEM elements long. When using these modifiers, values are transferred to/from the V-register file.

The modifiers `nelemdof` and `nelemstep` are used when performing grid-based operations. `nelemdof` is primarily used when executing `applyStateOperator`, storing element densities in the X-register file and their corresponding DOFs in a V-register slot. `nelemstep` is used for density filtering, requirng multiple X-registers to store the element values necessary to apply the filter.

The modifier `single` is used when the instructions should not be iterated but performed once only. This is primarily useful for loading constant values into an S-register once, such that the instruction does not have to be repeated on every iteration of a subsequent, longer instruction packet.

4.4.3 R-type instructions

R-type instructions are used to encode instructions which operate on data. The following fields are used in an R-type instruction:

- Opcode: Mathematical operation to perform. See table 4.6 for valid opcodes
- Mod: Operation modifer, used to select which register files operands are taken from and the result is placed into. See table 4.7 for valid R-type modifiers.
- Immediate flag (I): When this bit is a logic 1, the instruction is decoded such that an immediate value is loaded
- Rd: Destination register
- Rs1/Immh: First source register / integer bits of immediate

- Rs2: Second source register
- Immfrac: Fractional bits of an immediate value

Opcode	Encoding	Comments
`add`	0b000100	Computes `value[rs1]+value[rs2]`
`sub`	0b000101	Computes `value[rs1]-value[rs2]`
`max`	0b000110	Computes `max(value[rs1],value[rs2])`
`min`	0b000111	Computes `min(value[rs1],value[rs2])`
`abs`	0b001000	Computes `abs(value[rs1])`
`mul`	0b010000	Computes `value[rs1]*value[rs2]`
`mac`	0b010001	Computes `sum(value[rs1]*value[rs2], n=VEC_LENGTH)`
`red`	0b010011	Computes `sum(value[rs1]*value[rs2], n=REG_DEPTH)`
`div`	0b100000	Computes `value[rs1]/value[rs2]`

Table 4.6: Table of possible R-type opcodes and their binary encodings. `value[x]` means whichever value is indicated by `x`, and does not refer to a specific register/vector.

Modifier	Encoding	Comments
`VV`	0b0000	Elementwise operations on values from `Vreg[rs1]` and `Vreg[rs2]`. The result is stored in `Vreg[rd]`
`XV`	0b0100	The first element of `Xreg[rs1]` is applied onto all elements in the first vector of `Vreg[rs2]`, second element of `Xreg[rs1]` with the second vector of `Vreg[rs2]`, etc. The result is stored in `Vreg[rd]`
`SV`	0b1000	The scalar value from `Sreg[rs1]` is applied to all elements of `Vreg[rs2]`. The resulted is stored in `Vreg[rd]`
`XX`	0b0101	Elementwise operations on values from `Xreg[rs1]` and `Xreg[rs2]`. The result is stored in `Xreg[rd]`.
`SX`	0b1001	The scalar value from `Sreg[rs1]` is applied to all elements of `Xreg[rs2]`. The result is stored in `Xreg[rd]`
`SS`	0b1010	A scalar arithmetic operation is performed with `Sreg[rs1]` and `Sreg[rs2]`. The result is stored in `Sreg[rd]`
`KV`	0b1100	The elements of `Vreg[rs1]` are applied onto $\mathbf{K}_e$ for at matrix-vector product. The result is stored in `Vreg[rd]`

Table 4.7: Possible R-type modifiers and their binary encodings

The opcode is used to select the arithmetic operation to be performed, while the modifier field is used to select which of the register files each operand should come from. R-type instructions are written in a manner very similar to the format used in RISC-V assembly language. Some valid R-type instructions are shown in listing 4.3.

```
<op>.<mod> <rd>, <rs1>, <rs2>
add.vv v0, v1, v2 //Add v1 and v2, store result in v0
mul.xv v0, x1, v2 //Multiply element of x1 onto vectors in v2. Store result in v0
abs.xx x0, x1   //Take the abs of all elements in x1, store the result in x0
sub.ix x1, x2, 2.5 //Subtract the immediate 2.5 from all values in x2, store
    result in x1
min.ss s1, s2, s3 //Take the min of the value in s2 and s3, store in s1
red.vv x0, v1, v2 //Elementwise multiply and sum all 24 elements in each vector
    of v1 with v2. Store each result at a position in x0
mac.vv s1, v2, v3 //Compute the vector dot product of v2 and v3, store result in
    s1
mac.kv v2, v3 //Perform a matrix-vector product with v3, store result in v2.
```

Listing 4.3: A number of valid R-type instructions written in assemlby

The modifiers `VV, XX` and `SS` are probably the most intuitive. When these modifiers are used, an elementwise operation is performed on all values in the given source registers, and the result is placed into a result register of the same type. Exceptions to this are the `red` and `mac` opcodes which are explained below.

The modifier `XV` is only used when an X-register should be multiplied onto a V-register, as is done in `applyStateOperator`. Performing this operation will take the first value in `Xreg[rs1]` and multiply onto all elements in the first vector of `Vreg[rs2]`, storing the result in the first vector of `Vreg[rd]`. After this is finished, the second value of `Xreg[rs1]` is multiplied onto the 24 values in the second vector of `Vreg[rs2]`, storing the result in the second vector of `Vreg[rd]`, etc. This is repeated until all elements/vectors in the given X-register and V-register slot have been multiplied.

The modifiers `SX` and `SV` are used when a scalar value should be applied to a V-register or X-register. This is also the underlying modifier used when the immediate flag `I` is set.

The modifier `KV` is only used in conjunction with the opcode `mac`. Since the $\mathbf{K}_e$ matrix is the same for all elements, this matrix is stored directly in hardware. This opcode+modifier combination is used to multiply the elements in `Vreg[rs1]` onto the $\mathbf{K}_e$ matrix, calculating the matrix-vector product.

The opcode `mac` (multiply-accumulate) may also be used with the R-type modifiers `VV` and `SV` to calculate a vector-vector dot product or calculate the (scaled) sum of all values in a single vector. In both cases, the destination register will be an S-register instead of a V-register, as is ordinarily the case when executing instructions with those R-type modifier. When using the `mac` opcode, the operation is performed over *all* elements of an $NDOF/NELEM$ long vector, as specified by the O-type length given in the `pstart` instruction.

The opcode `red` (reduce) may only be used with the modifiers `VV` and `XX`. It works in a manner similar to `mac`, in that it performs a vector-wise multiply-accumulate operation. Whereas `mac` operates over all elements in an entire vector, `red` only operates on the elements in a V-register slot or a single X-register.

If e.g. the instruction is `red.vv x0, v0, v1` the 24 elements at `v0[0]` are elementwise multiplied with the 24 elements of `v1[0]`, and the sum of these 24 multiplications is stored into `x0[0]`. Likewise, the elements at `v0[n]` are multiplied with `v1[n]`, and the sum is stored into `x0[n]`, for $n \in \{0, 1, \ldots, XREG\_DEPTH - 1\}$.

In a similar manner, the instruction `red.xx s1, x0, x1` performs an elementwise multiplication of the XREG_DEPTH elements of x0 with the elements of x1, storing the result into S-register s1.

When the immediate flag `I` is set, the 4 uppermost bits of the instruction are used as the integer part of an immediate value, encoding a Qs3.7 fixed-point number. This value is then sign-extended to a Qs15.38 value. This means that the smallest possible immediate is $2^{-3} = -8$, and the largest possible immediate value is $2^3 - 2^{-7} = 7.9921875$. When larger/smaller immediates are needed, they can relatively easily be constructed by multiplying two immediate values.

Given these R-type instructions, it is now possible to write the contents of the inner loop of `applyStateOperator` in assembly language, as shown in listing 4.4

```
1  //In x0 is the element densities
2  //In v0 is the element DOF of 'in'
3  //In v1 is the element DOF of 'out'
4  //In s1 is 'e0-emin'
5  //In s2 is 'e0'
6  mul.xx x1, x0, x0 //x0 = pow(x,2)
7  mul.xx x1, x1, x0 //x1 = pow(x,3)
8  mul.sx x1, s1, x1 //x1 = pow(x,3)*(e0-emin)
9  add.sx x1, s2, x1 //x1 = emin+pow(x,3)*(e0-emin) = elementScale
10 mul.xv v2, x1, v0 //v2 = u_local*elementScale
11 mac.kv v2, v2 //v2 = ke*(u_local*elementScale)
12 add.vv v1, v1, v2 //out[edof[i]] += out_local[i]
```

Listing 4.4: Assembly code version of the inner loop of `applyStateOperator`.

4.4.4 S-type instructions

S-type instructions are used to perform memory load/store operations. The following fields are used in an S-type instruction:

- Base address: The base address in memory of the vector which should be loaded/stored from/to.
- Mod: Modifier used to change the behaviour of the memory module and how values are loaded/stored. See table 4.9.
- Load/store flag: Used to indicate if a given S-type instruction is a Load or Store instruction
- Rs/Rd: The source register when performing store operations, or the destination register when performing load operations.

To correctly perform topology optimization in a manner similar to the one used in the C-code, it is necessary to address 13 different memory locations. A summary of the memory locations that must be addressible is shown in table 4.8.

Length	Name	Encoding
NELEM	`X`	0
	`XPHYS`	1
	`XNEW`	2
	`DC`	3
	`DV`	4
NDOF	`F`	5
	`U`	6
	`R`	7
	`Z`	8
	`P`	9
	`Q`	10
	`INVD`	11
	`TMP`	12

Table 4.8: List of base addresses available and vector lengths.

Modifier	Encoding	Comments
`elem`	0b0010	Load/store element densities from an NELEM long vector into an X-register
`edn1`	0b0100	Load the element densities of a subset of elements which share an edge with the central element into an X-register
`edn2`	0b0101	Load the element densities of another subset of elements which share an edge with the central element into an X-register.
`fcn`	0b0110	Load the element densities of all elements which share a face with the central element into an X-register.
`sel`	0b0111	Load/store the value of the central element into an X-register.
`dof`	0b1000	Load/store the DOFs associated with an element into a V-register
`fdof`	0b1001	Store the fixed DOFs associated with an element
`vec`	0b1100	Load/store an NELEM or NDOF long vector without taking into account grid coordinates. Stores in a V-register

Table 4.9: All S-type modifiers and their binary encodings

Writing S-type instructions in assembly language is similar to writing R-type instructions, as is shown in listing 4.5

The modifier `vec` is used when vectors should be traversed linearly, instead of traversing the grid using the 8-colouring. This can only be done if the O-type length is `ndof` or `nelemvec` Elements are loaded/stored into/from V-registers.

```
<ls>.<mod> <rsrd>, <baseAddr>
ld.vec v0, u
st.dof v1, r
ld.elem x2, x
ld.fcn x3, xphys
ld.edn1 x0, x
ld.edn2 x1, x
ld.sel x2, xphys
st.fdof v1, r
```

Listing 4.5: A number of valid S-type instructions written in assembly language

The S-type modifiers `dof` and `elem` are used to load/store element densitites and the DOFs of an element. The S-type modifier `fdof` is only used when storing data. As seen at the end of listing 4.1, some of the DOFs are fixed, meaning that the output vector must preserve the original values from the input vector. The modifier `fdof` is used to store the original values from the input vector into the output vector. In hardware, it is then ensured that these writes are only performed to addresses which correspond to a fixed degree of freedom. These three modifiers are mainly used when executing `applyStateOperator`, and are used with the O-type length `nelemdof`.

The four S-type modifiers `fcn, edn1, edn2` and `sel` are used when applying a density filter to the grid. `ld.fcn` will return either 3, 4, 5 or 6 values, depending on whether the element that it was called on is located at a corner, edge, face or internally in the design domain. Likewise, `ld.edn1` and `ld.edn2` may return a varying number of elements, from 0 to 6, depending on the location of the central element. `ld.sel` will always return exactly one value. These modifers are used with the O-type length `nelemstep`.

As these values are stored in X-registers, some of the fields in that X-register may be left unused. When this is the case, the remaining entries in that register are set to zero such that no values left over from a previous instruction remain in the register. Since `ld.fcn, ld.edn1` and `ld.edn2` return up to 6 values each, each X-register must be at least 6 elements deep for the ASIP to function correctly. If the depth of an X-register is 12 or more, it would be possible to only require one modifier `edn` instead of two separate modifiers `edn1` and `edn2`. The current version of the ASIP does not operate with an X-register depth of 12, however, and for this reason, two separate modifiers are necessary.

4.4.5 B-type instructions

B-type instructions are used to control the flow of the program, performing comparisons on two register operands and branching based on the outcome. The fields used in a B-type instruction are:

- Comp: The comparison to perform. See table 4.10 for possible comparisons.
- Rs1: The S-register of the first operand
- Rs2: The S-register of the second operand
- Targeth: The high 7 bits of the branch target
- Targetl: The low 9 bits of the branch target.

All branches utilize PC-relative addressing to calculate branch targets, calculating the new value of the program counter (PC) by adding the target value to the current value of the program counter.

The branch target is a signed 18-bit value with the two least significant bits being implicit 0's. The ASIP is designed with byte-addressable memory in mind, meaning that only 4-byte increments are needed when performing branches. With a signed 18-bit number, each B-type instruction can branch up to $\pm 2^{17}$ bytes or $\pm 2^{15}$ instructions away from the current PC value.

B-type instructions are unique in that they should not be placed inside of an instruction packet. Instead, B-type instructions must be placed outside packets, such that no instructions currently occupy the execution pipeline when a branch is taken.

In assembly, a B-type instruction is written in a manner similar to that from RISC-V, as shown in listing 4.6.

```
<comp> <rs1>, <rs2>, <label>
bne s0, s1, L1 //Branch not equal
beq s1, s2, L2 //Branch equal
blt s2, s3, L3 //Branch less than
bge s3, s4, L4 //Branch greater than or equal
```

Listing 4.6: Examples of valid B-type instructions for the ASIP

Comp	Encoding	Comments
`eq`	0b000000	Branch taken if `Sreg[rs1] == Sreg[rs2]`
`neq`	0b000001	Branch taken if `Sreg[rs1] != Sreg[rs2]`
`lt`	0b000010	Branch taken if `Sreg[rs1] <  Sreg[rs2]`
`geq`	0b000100	Branch taken if `Sreg[rs1] >= Sreg[rs2]`

Table 4.10: List of available B-type comparisons. All comparisons are performed with signed values.

Given this introduction to the ISA designed for the ASIP, it is now possible to write the entirety of `applyStateOperator` from listing 4.1 using assembly language instead. The input vector is **u** and the output vector is **r**.

```
pstart single //Load constant values
estart //No memory load
add.is s1, s0, 0.0078125 //s1 = 0.0078125
mul.ss s1, s1, s1 // s1 = 6.1e-5
add.is s2, s0, 0.015625 //s2 = 0.015625
mul.ss s2, s1, s2 //s2 =9.5e-7 ~= 1e-6 = emin
add.is s1, s0, 1 //s1=1=e0
sub.ss s1, s1, s2 // s1 = e0-emin
mul.sv v0, s0, v0 //Clear v0
eend
pend

//Clear output vector
pstart ndof //operate on ndof long vector
estart //No memory loads
eend //No executable instructions
st.vec v0, R //Store data to clear R vector
pend

pstart nelemdof //Loop over elements and perform applystateoperator
ld.dof v0, U //Load values from U vector into v0
ld.dof v1, R //Load values from R vector into v1
ld.elem x0, XPHYS //Load densities from XPHYS into x0
estart
mul.xx x1, x0, x0 //x0 = pow(x,2)
mul.xx x1, x1, x0 //x1 = pow(x,3)
mul.sx x1, s1, x1 //x1 = pow(x,3)*(e0-emin)
add.sx x1, s2, x1 //x1 = emin+pow(x,3)*(e0-emin) = elementScale
mul.xv v2, x1, v0 //v2 = u_local*elementScale
mac.kv v2, v2 //v2 = ke*(u_local*elementScale)
add.vv v2, v2, v1 //out[edof[i]] += out_local[i]
eend
st.dof v2, R //Store the values in v2 as DOFs into vector R
st.fdof v0, R //Store fixed DOFs with input values from U
pend
```

Listing 4.7: Assembly code version of the inner loop of `applyStateOperator` including O-type instructions and memory load/store instructions

The assembly language version of `applyStateOperator` takes up about the same number of lines of code as the original version of the function written in C. While assembly language programs are generally longer than their counterparts written in a higher-level language, the ISA designed for the ASIP allows one to write a relatively succint version of the same function.

4.4.6 Closing remarks

In this chapter, an introduction to the design domain has been presented. It has been explained how the design domain is structured, and how the design elements and their degrees of freedom are related. The critical region of topology optimization using the SIMP approach has been identified, and an instruction set architecture for the hardware accelerator has been defined.

It has been decided that the accelerator will be implemented as an ASIP, and the ISA has been designed to make matrix-vector products with the $\mathbf{K}_e$ matrix as fast as possible. The ISA is found to be relatively succint, thanks to the use of non-addressable registers to aid in address generation when performing load/store operations. This allows one to write the assembly language version of the function `applyStateOperator` in almost the same number of lines of code as the C version of the function.

5. Implementation

In this section, the implementation of the hardware accelerator for topology optimization is presented. First, an overview of the final design is given, to aid in understanding the following sections. After this, individual pipeline stages of the ASIP are described in detail. The functionality of each stage is described, and decisions made in the design process are presented.

5.1 Design overview

The ASIP that has been designed is based on the generic five-stage pipeline described in [10]. Since topology optimization is a very memory-intensive process, the accelerator has been designed with a focus on performing execution and memory accesses in parallel. For this reason, the memory stage has been placed in parallel to the execute-writeback stage, instead of placing it between the execute and writeback stages of the generic pipeline. Figure 5.1 shows a schematic containing the main pipeline stages and functional units of the accelerator.

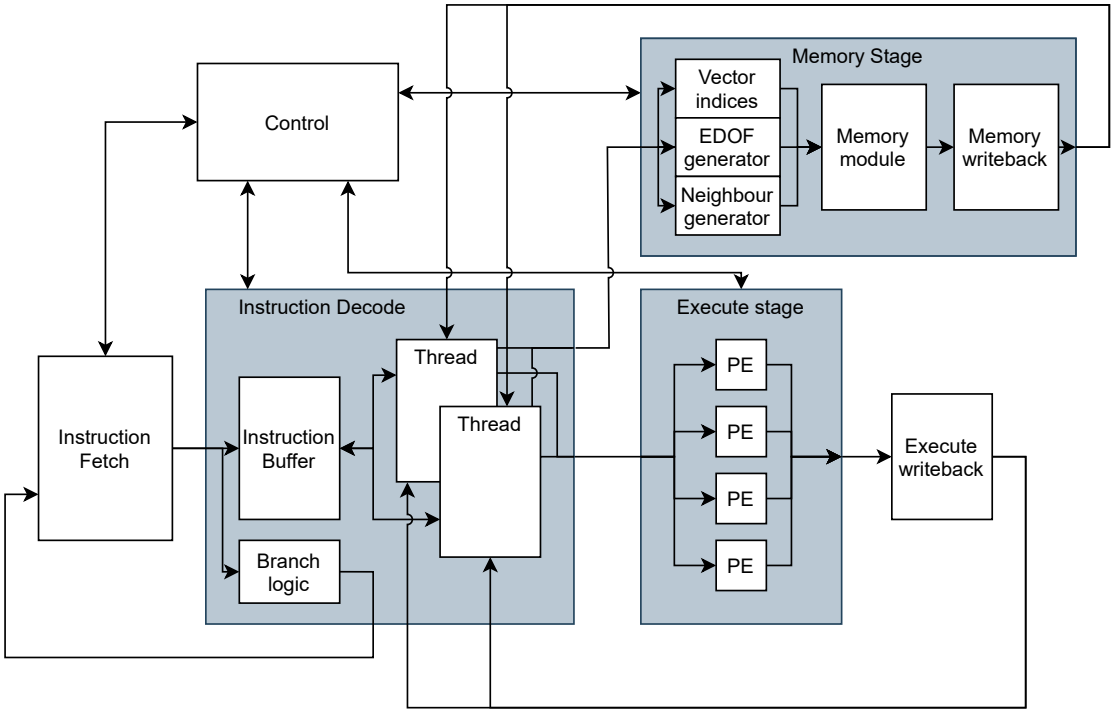

Figure 5.1: An overview of the architecture of the ASIP. Modules containing other modules have been coloured dark grey for better contrast. Note that no switching logic has been included in this drawing to indicate what thread is accessing memory and what thread is accessing the execute stage. Bitwidths have been omitted to keep the drawing simple.

To achieve high utilization of all device resources at any one time, the accelerator has been designed as a pseudo-static dual-issue processor. The processor is not a strictly static multiple issue processor since specific instructions are not required to execute at the same clock cycle across

processing cores. Instead, the decode stage of the processor includes two processing cores, dubbed threads in this project.

At any one time during execution, one thread will be accessing the execute stage, executing R-type instructions. The other thread will be accessing the memory stage, executing S-type instructions to perform memory operations. When both threads are finished accessing their current resource they swap priorities, letting the previously executing thread store the results of its computations and load new data, while the thread that has fetched new data can execute R-type instructions.

The decode stage contains a central instruction buffer, where an entire instruction packet is copied from instruction memory whenever a new packet is initiated. Once a packet has been copied into the buffer, each thread contains its own instruction pointer, used to read instructions from the instruction buffer.

The execute stage contains a Matrix Processing Unit (MPU) consisting of multiple Processing Elements (PEs) on which matrix-vector, vector-vector, scalar-vector and scalar-scalar operations can be performed. A writeback stage accepts results from the execute stage, storing them in a buffer until an entire V-register or X-register has been processed and the new values may be written back into the correct register file.

The memory stage contains a memory module utilizing 8 interleaved banks of memory, allowing one instruction to load 8 words of data on each clock cycle. Ahead of the memory module is a number of different vector index generation units, used to calculate the correct indices to access in a given vector. A memory writeback stage buffers read data from the memory module until a full V-register or X-register has been read, at which point it is written back into the register file.

A central control module receives inputs from all pipeline stages and issues control signals back. These include loading a new instruction packet once the previous packet has been finished, stalling the execute stage if a data hazard is about to occur, or preventing threads from swapping resources before all memory loads have been finished.

In the current implementation, each V-register file is 32 vectors wide, the X-register files are 8 vectors wide, and the S-register file is 16 elements wide. The execute stage uses 8 PEs, meaning that each V-register file is split into 4 V-register slots, each containing 8 vectors. Each X-register is 8 elements deep.

5.2 Execute, writeback and forwarding

First, the implementation of the execute and writeback stages will be explained. The design of the execute and decode stages is closely linked, and understanding how the execute stage is implemented will lead to an easier understanding of the decode stage's implementation.

5.2.1 Arithmetic circuits

Since the chosen FPGA has dedicated addition and multiplication circuits, no effort has been made to design custom addition/multiplication logic, as this likely cannot outperform the hard arithmetic blocks on the FPGA.

Many circuits for computing more complex arithmetic operations, such as division and square roots, rely on iterative approaches, as discussed in [28]. For this project, it has been chosen to implement division via the Newton-Raphson algorithm, and implement square roots via the Babylonian Method, described in [29]. Both are iterative algorithms based on Newton's method, where an initial guess is refined through multiple iterations.

The Newton-Raphson algorithm for computing divisions is implemented in hardware, whereas square roots are calculated in software.

In addition to the basic arithmetic operations, the system must also be able to identify the maximum and minimum of two input values, as well as taking the absolute value of an input. Finally, it should be able to store an intermediate result generated when executing multiply-accumulate instructions with the `mac` and `red` opcodes.

5.2.2 Parallel computation and PEs

With the arithmetic circuits defined, it is possible to combine these into a processing element. The general design for a processing element is inspired by the design proposed in [12].

Each processing element takes two input operands a and b, the opcode and a multiply-accumulate limit as inputs. The operands go into the initial divide/multiply stage, the result of which is stored in a register. The output of this register, alongside the original operands, is then used as the input to a second stage where the remaining operations are implemented in an Arithmetic Logic Unit (ALU). The output of this ALU is presented on the output of the processing element, and it is also stored in another register, for reuse on later iterations when processing `mac` or `red` instructions. The structure of a processing element is shown in fig. 5.2.

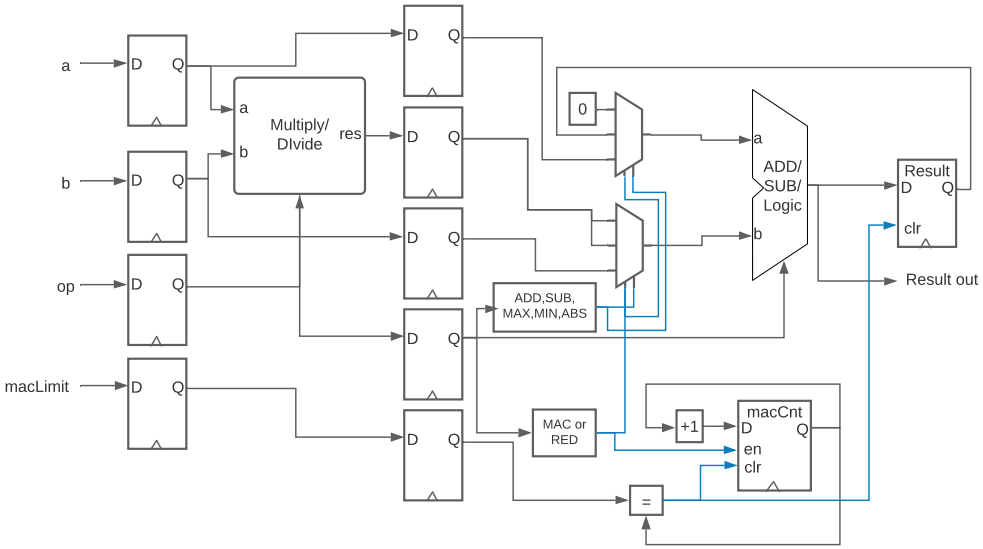

Figure 5.2: Structure of a processing element. Control signals are marked with blue. Some control signals such as valid lines are not shown.

The current opcode is used to select the operands going into the ALU. When a multiply or divide operation is performed, the result is added with 0 and the output is presented on the result port of the PE. If any of the arithmetic operations are performed, the a, b operands are used as inputs to the ALU.

If a `mac` or `red` instruction is performed, an initial multiplication takes place in the first stage of the processing element. The result of that multiplication is added to whichever value is currently stored in the result register and the register `macCnt` is incremented. Once $macCnt = macLimit$, the operation is completed and the output of ALU is the final result for that instruction.

Since e.g. division and multiplication do not take the same number of clock cycles to finish, it is imperative that the input opcode is kept constant throughout an entire operation. Otherwise, if a division is started followed by a stream of e.g.additions, the result of the division is lost. The logic surrounding the MPU is responsible for keeping the opcode constant, as will be explained in section 5.2.4. Inside each PE, valid lines are used to signal that new data can be operated on. All modules are assumed always-ready, hence no ready signalling is used.

5.2.3 The Matrix Processing Unit

Using multiple PEs, a matrix processing unit may be defined. The MPU is capable of performing matrix-vector products, vector-vector, scalar-vector and scalar-scalar operations.

Given that all elements in the design domain have 24 degrees of freedom, and the matrix $\mathbf{K}_e$ is a 24×24 matrix, the MPU must be able to perform 24×24 matrix-vector products. Performing an entire matrix-vector products at once would require 576 processing elements, far more than it is feasible to map onto the FPGA. Instead, a smaller number of processing elements is used, and matrix-vector operations are completed using block matrix multiplication. The only requirement at this point is that the number of processing elements NUM_PROCELEM is a divisor of 24.

Computing matrix-vector products with submatrices allows us to break a large problem into smaller sections, adding the final results in the end. As an example, imagine a 4×4 matrix $\mathbf{A}$ and

a 4×1 vector $\mathbf{b}$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{5.1}$$

These may be broken into 2×2 and 2×1 submatrices and subvectors respectively.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} & \mathbf{A}_{12} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \\ \mathbf{A}_{21} = \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} & \mathbf{A}_{22} = \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \tag{5.2}$$

$$\mathbf{b} = \begin{bmatrix} \mathbf{b}_1 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ \mathbf{b}_2 = \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} \tag{5.3}$$

The original matrix-vector product may then be computed as a matrix-vector product of submatrices and subvectors.

$$\mathbf{Ab} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{b}_1 + \mathbf{A}_{12}\mathbf{b}_2 \\ \mathbf{A}_{21}\mathbf{b}_1 + \mathbf{A}_{22}\mathbf{b}_2 \end{bmatrix} \tag{5.4}$$

Where each individual sub-matrix-vector product can be broken into a scalar-vector sum

$$\mathbf{A}_{11}\mathbf{b}_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} b_1 + \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} b_2 \tag{5.5}$$

The above reduction of a large matrix-vector product into a sum of scalar-vector products allows us to define an MPU layout which is optimized for matrix-vector as well as vector-vector operations, as seen in fig. 5.3.

It has been chosen to use a column of processing elements instead of a grid of PEs as used in [12] since the accelerator will never need to perform matrix-matrix multiplication. By limiting the accelerator to matrix-vector operations and breaking matrix problems into vector-based problems, the MPU is also able to efficiently handle vector-vector operations.

Computing a matrix-vector product with the $\mathbf{K}_e$ matrix now involves taking the first "slice" of the matrix and multiplying this slice with the first element in a 24×1 vector. Performing this multiply-accumulate step 24 times will result in each processing element generating the first NUM_PROCELEM values in the output vector. A total of $24 \frac{24}{NUM\_PROCELEM}$ iterations are needed to calculate all 24 values in an output vector. In the current version of the accelerator, 8 processing elements are used, requiring 72 multiplications in each processing element to compute one matrix-vector product.

5.2.4 Execute stage

In the execute stage, a pipeline register is used to hold input values from the decode stage. Since multiple operations may be processing at once, a queue keeps track of all currently executing instructions. When a valid instruction is passed from the decode stage, the destination register file and register index are stored in the destination queue. When an instruction is finished executing, that destination bundle is dequeued and passed on to the writeback stage with the results of the calculation. A separate queue is used to keep track of `mac` instructions, since the result of a `mac` instruction may be computed after subsequent instructions are passed into the queue. A diagram of the execute stage can be seen in fig. 5.4.

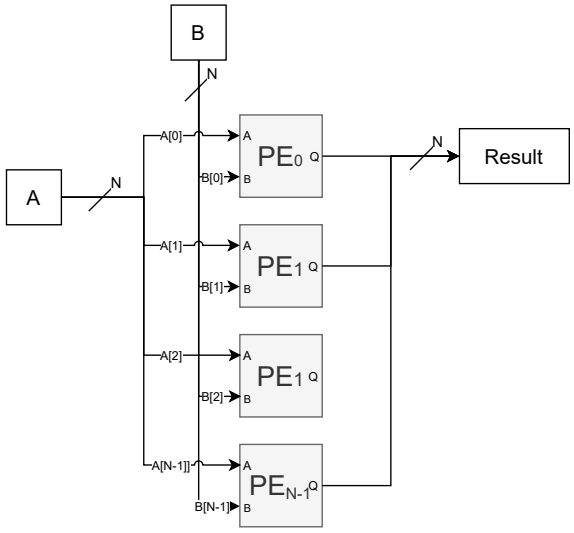

Figure 5.3: Layout of the MPU. A number of processing elements are connected in a column, allowing for matrix-vector or vector-vector operations. The bitwidth N indicates that N separate words come from A and B.

To ensure that no instructions are overwritten, the opcode going into the MPU must be kept constant while an operation is processing. A new opcode is therefore only accepted whenever the destination queue is empty, or when it is known that the upcoming instruction cannot override the current instruction.

If the instruction contains an immediate, the `useImm` signal from the decode stage is pulled high, causing the immediate value to be used instead of the value present on the 'a' input.

The `reduce` output signals to the writeback buffer that all results coming in should be summed together into a single value. This is only activated if a `mac` or `red` instructions has been processed.

At some points during execution, a number of data hazards may occur. These hazard conditions are checked in the control unit (see section 5.6), and the execute stage is stalled if any one of them arises. The execute stage is stalled until the hazard has been resolved, usually by finishing all instructions in the MPUs pipeline.

A forwarding unit has been implemented to avoid pipeline stalls when results have been computed but not yet stored in the destination register file. Data is forwarded from the writeback buffer to the execute stage if the destination register file and index match the source register file and index of any one of the operands.

5.2.5 Execute writeback stage

The execute writeback stage is used as a buffer for storing intermediate results when the result of an operation is calculated over multiple iterations. If e.g the destination register is a V-register, 24 values are required to fill the register, but only NUM_PROCELEM values are generated in the execute stage. A 24-deep writeback buffer is used to build a full result before the output is written to the register file of the attached thread.

If the destination register is an X-register, the behavior of the writeback stage depends on the `reduce` signal. If it is low, an `XX` or `SX` instruction was calculated. Since the number of processing elements matches the depth of an X-register, the result is written to the X-register file on the clock cycle after the result is received. If the `reduce` signal is high, a `red.vv` instruction has been performed. The results from the execute stage are summed together and stored in position 0 of the write buffer. The next results are also summed together and placed into subsequent indices of the write buffer until XREG_DEPTH elements have been received, at which point this is written into the X-register file of the connected thread.

If the destination register is in the scalar register file, the result is always written to the register file on the next clock cycle. If a `mac.vv` or `red.xx` instruction was performed, the results from the execute stage are summed together before writing the output to the register file. Otherwise, an instruction with R-type modifier `SS` has been executed.

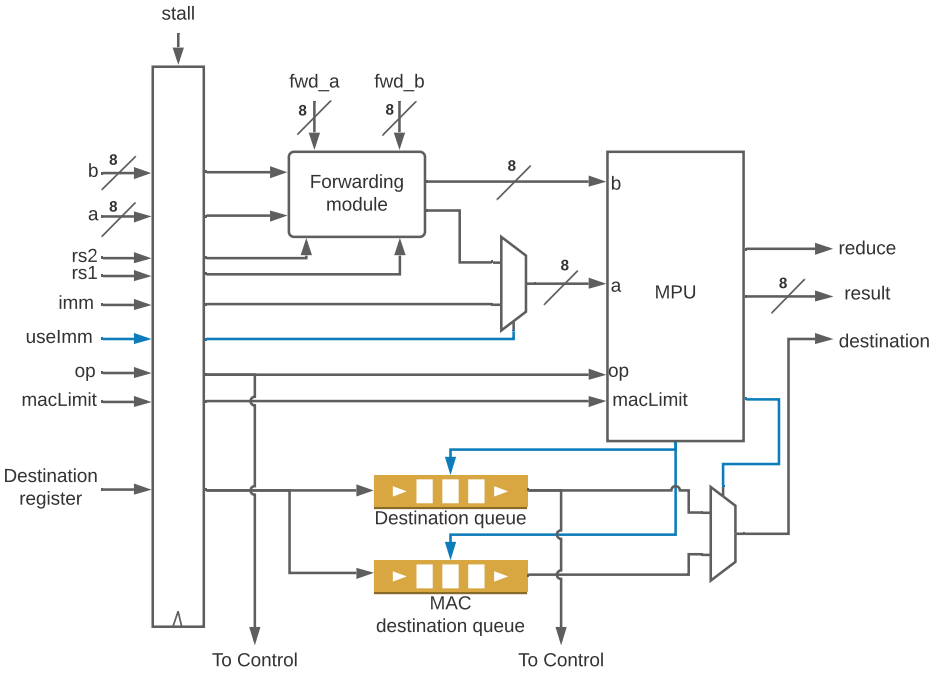

Figure 5.4: The execute stage of the ASIP. The current opcode and head of the regular destination queue are sent to the control module to avoid data hazards. Some control signals, such as valid lines, have been omitted to avoid clutter. The remaining control signals have been coloured blue to separate them from data signals. Widths indicate the number of values bundled together and not the bitwidth of each signal.

5.3 Instruction fetch

The instruction fetch (IF) stage contains the instruction memory of the system. A register holding the program counter is used to index into instruction memory, outputting the instruction stored at that address. An adder is used to calculate the value $PC + 4$ which is the address of the next instruction to be loaded. Figure 5.5 shows a schematic of the instruction fetch stage.

The multiplexer defaults to selecting the current value of the program counter. If the instruction load signal `iload` is asserted in the control module, the value $PC+4$ is instead put onto the register, updating the PC on the next clock cycle. If the branch signal `branch` is asserted from the decode stage, a branch was taken, and the PC takes on the value of the branch target. In case both signals are asserted at once, the branch takes precedence, as is seen in fig. 5.10.

5.4 Instruction Decode and Threads

In the instruction decode stage, an instruction buffer is used to store an entire instruction packet. Since each instruction in a packet is executed multiple times, storing the packet in a buffer makes it easier for each thread to access the correct instruction, instead of loading instructions from the IF stage on every clock cycle.

The decode stage contains the instruction buffer, branch evaluation logic, threads, an FSM which controls the decode stage, and the central S-register file of the system. A central scalar register file is necessary to correctly execute branch instructions, since a branch instruction must uniquely identify the registers to compare. As the ISA does not support addressing registers in the individual threads, the scalar register file must be placed in the decode stage where both the branch logic and executing stage may access it.

A schematic of the decode stage is seen in fig. 5.6.

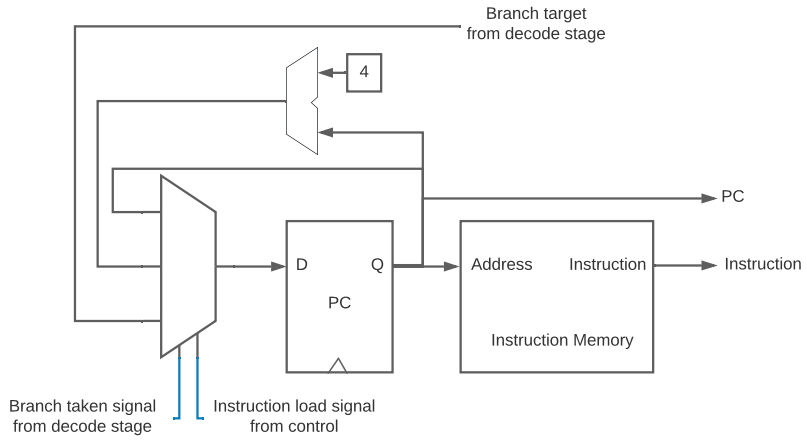

Figure 5.5: Schematic of the instruction fetch stage in the accelerator. Control lines are coloured blue.

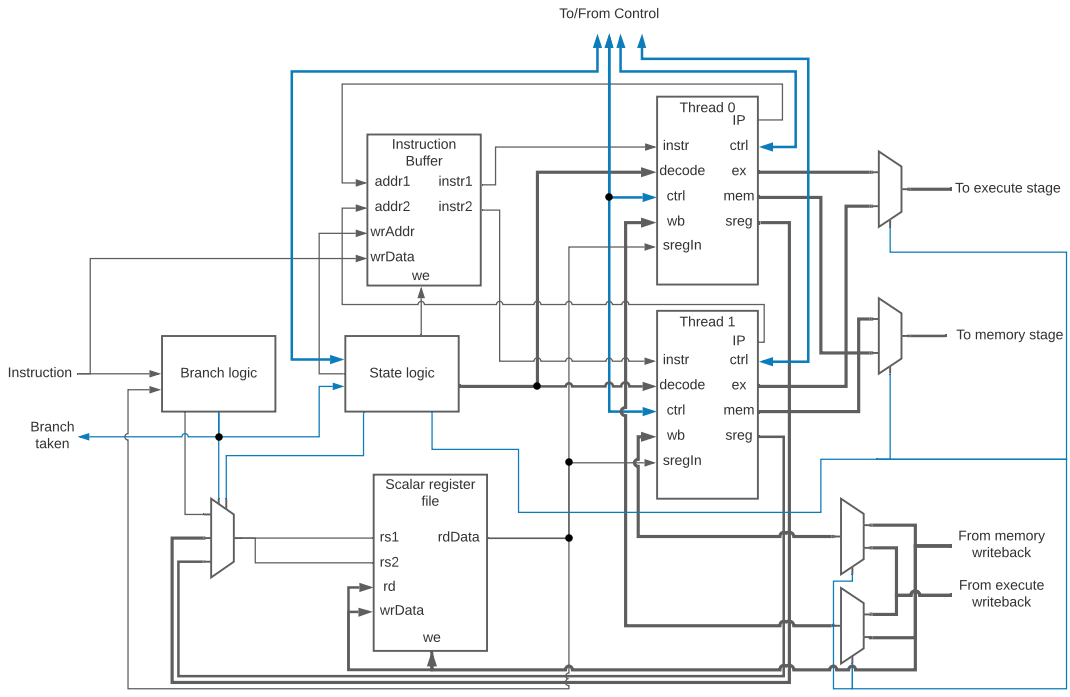

Figure 5.6: Schematic of the instruction decode stage. Control signals are marked with blue. Bundles containing multiple signals are drawn with thicker lines. Bitwidths and some control signals have been omitted to avoid clutter.

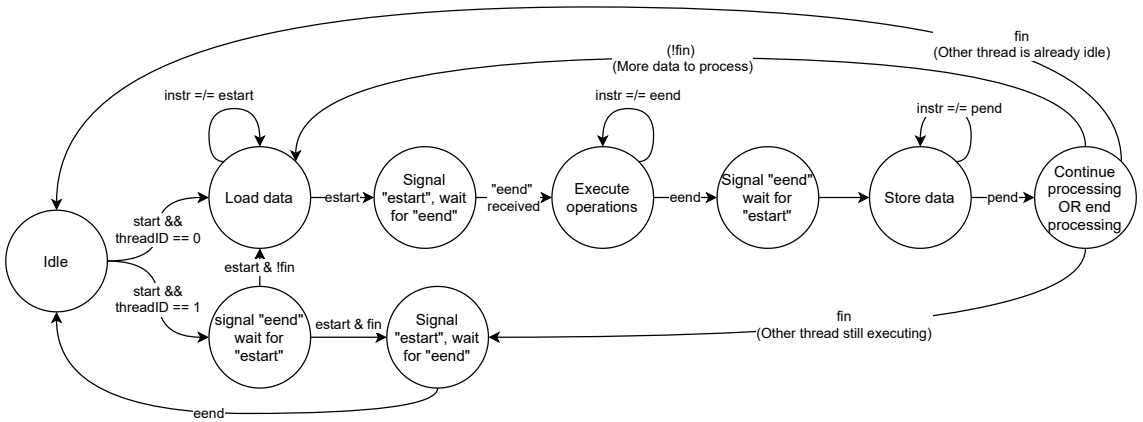

Figure 5.7: State diagram for the FSM controlling each Thread in the decode stage

5.4.1 Threads

Threads perform the majority of the processing in the decode stage. Each thread has its own V-register file, X-register file, immediate generator and instruction decode logic. Threads also contain an FSM that ensures the threads operate in the correct order, only ever attempting to access the execute/memory stage once the other thread has signalled that it is finished operating on that stage. The state diagram for the FSM that is implemented in both threads can be seen in fig. 5.7, and a schematic of a thread can be seen in fig. 5.8.

Two separate datapaths are implemented in each thread. One is used to generate output signals when the thread is performing load/store operations, the other when executing R-type instruction. Threads transfer data to the execute stage on the a and b outputs, alongside the source registers and the immediate value. When writing to memory, the a output is reused as the write data port.

Decoding R-type instructions

Once a thread has entered the execute state, it will start decoding the R-type instructions in the current instruction packet. A number of internal registers are used to select the correct values from the correct register file.

When e.g. executing an instruction with R-type modifier `VV`, the registers `slotSelect` and `X` are used to generate the correct indices into the V-register file. The `slotSelect` register chooses which vector from a given V-register slot should be read, and the register `X` is used to select the correct subvector of that vector. A subvector is NUM_PROCELEM large, such that if the e.g. the number of processing elements is 8, each vector is split into three subvector of length 8.

If e.g. each V-register slot is 4 vectors wide, `slotSelect` will take on four different values. Once all possible combinations of `X` and `slotSelect` have been iterated, all subvectors in the two addressed V-register slots have been read, and the instruction is finished. Table 5.1 contains a summary of which utility registers are used when decoding which R-type modifier.

To perform matrix-vector products as a series of scalar-vector operations, in the manner described in section 5.2.3, the $\mathbf{K}_e$ matrix has been partitioned into a number of submatrices, each of which is NUM_PROCELEM $\times$ NUM_PROCELEM large. Each of these submatrices has then been sectioned into the NUM_PROCELEM columns that make up the submatrix, for a total of $24\,\frac{24}{NUM\_PROCELEM}$ slices. When executing matrix-vector products, the `X`, `Y` registers point to a submatrix, and the `col` register select a column/slice of that submatrix to output on the b output. `X` and `col` together are also used to select one element of the subvector selected by `slotSelect` which is output on all b outputs.

Decoding S-type instructions

Each thread contains a vector index generator and a coordinate generator, dubbed the IJK-generator. The vector index generator generates subsequent indices to traverse a vector directly, and the IJK generator generates coordinate pairs (i, j, k) that map to elements in the grid in such a way that the 8-colouring of the grid is respected. Based on the O-type length encoded in the instruction, either the vector index generator or IJK-generator is used to handle memory acceses.

R-type modifer	Registers used	Comments
`VV`	`X`, `slotSelect`	`slotSelect` is used to select which vector of the vector slot indicated by `rs1`/`rs2` should be accessed. `X` is used to select a subvector from that vector. Each subvector is output on a and b respectively.
`XV`	`text` `slotSelect`	`X` and `slotSelect` index the V-register file as described above, outputting the subvector on the 'b' output. `slotSelect` is used to index into the X-register file, putting the value at `Xreg[rs1][slotSelect]` onto all 'a' outputs
`SV`	`X` `slotSelect`	`X`, `slotSelect` index the V-register file as described above and output on the 'b' port. The value at `Sreg[rs1]` is output onto all 'a' ports
`XX`	None	The values at `Xreg[rs1]` and `Xreg[rs2]` are output directly to the execute stage, each being output on all ports of 'a' and 'b' respectively.
`SX`	None	The value at `Sreg[rs1]` is output onto all 'a' ports. `Xreg[rs2]` is output on 'b'.
`SS`	None	The values at `Sreg[rs1]` and `Sreg[rs2]` are output on all 'a' and 'b' outputs respectively.
`KV`	`X, Y, col`, `slotSelect`	The element at `Vreg[X][col]` is output on all 'a' outputs. The col'th column of the $\mathbf{K}_e$ submatrix at coordinates (X, Y) is output on 'b'.

Table 5.1: Summary of all R-type modifiers and the internal registers which they use to drive the output logic in each thread

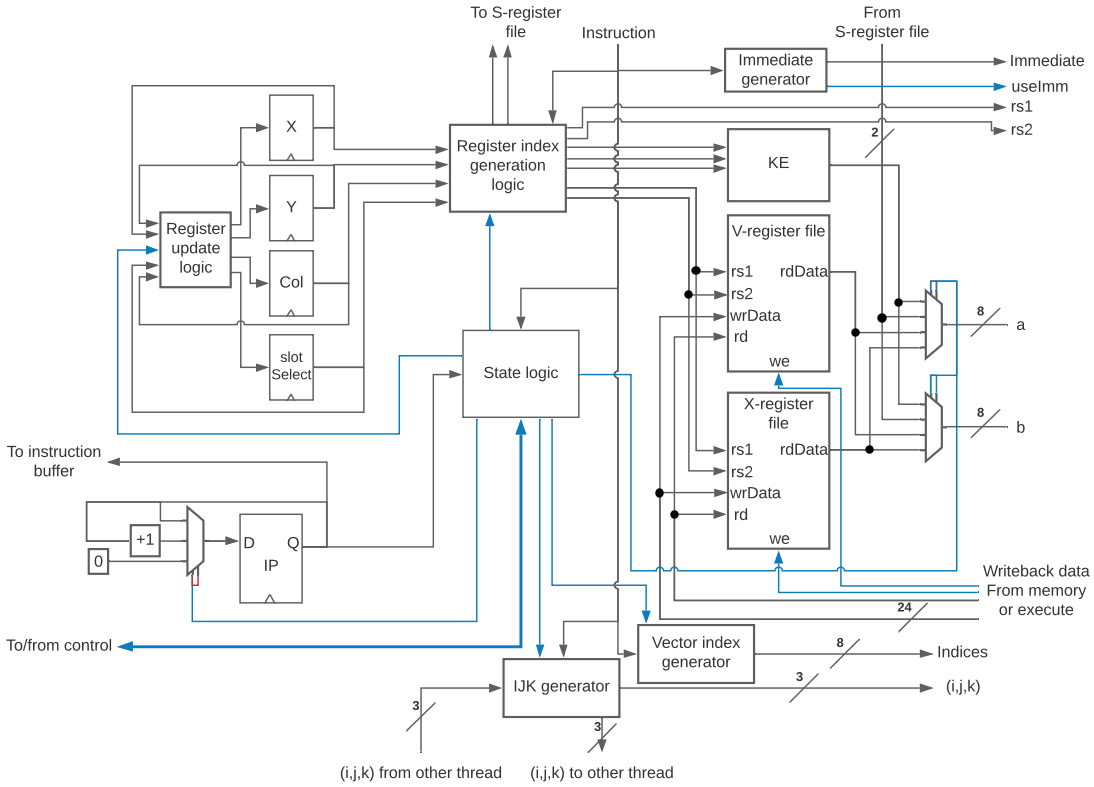

Figure 5.8: Schematic of a thread in the decode stage. All control signals are marked with blue. Widths indicate the number of values bundled together and not the bitwidth of each signal.

The state machine for an IJK generator is shown in fig. 5.9. The state machine for the vector index generator is very similar.

If the O-type length is `nelemdof` or `nelemstep`, the IJK-generator will iterate through coordinates in the same colouring, starting at $(0, 0, 0)$. Once all elements in colouring 0 have been evaluated, colouring 1 is initiated starting at $(0, 1, 0)$, then colouring 2, etc.

When the length is `nelemdof`, the generator will generate XREG_DEPTH coordinate pairs before restarting its generation sequence for the next load operation. This way, all load operations access the same XREG_DEPTH elements in a vector. When the final load operation has been processed, instead of restarting, the generator will increment one step past the final coordinates accessed, transmitting these coordinates to the other thread. The other thread will load these coordinates into its IJK-generator, setting this coordinate pair as the value at which its generation sequence starts. It will then start traversing the grid starting at these coordinates.

The original generator will then restart at $(0, 0, 0)$, performing store operations to the same coordinates as it loaded data from, before receiving a new starting coordinate pair from the other generator. The threads will continue this increment / exchange maneuver until all elements in the design domain have been covered, at which point the instruction packet is finished. The state diagram for the IJK generator can be seen in fig. 5.9.

If the O-type length is `nelemstep`, the IJK-generator proceeds in a manner similar to the one used for `nelemdof`. However, instead of generating XREG_DEPTH coordinates on each iteration, each thread only acceses one coordinate. These instructions thus take longer to iterate through the grid than when using `nelemdof`. This strategy is used when the S-type modifiers `.fcn`, `.edn1`, `.edn2` and `.sel` are used.

When the O-type length is either `ndof` or `nelemvec`, the vector index generator is used. Thread 0 will initially access indices 0-7 of a given vector, and when the memory stage has accepted these indices, it will increment all output indices by 8, now accessing indices 8-15. It will continue to access indices in increments of 8 until a total of ELEMS_PER_VSLOT = 24 VREG_SLOT_WIDTH indices have been accessed, at which point the generator will reset to its previous starting value. This is repeated for every S-type instruction in the load section of the instruction packet.

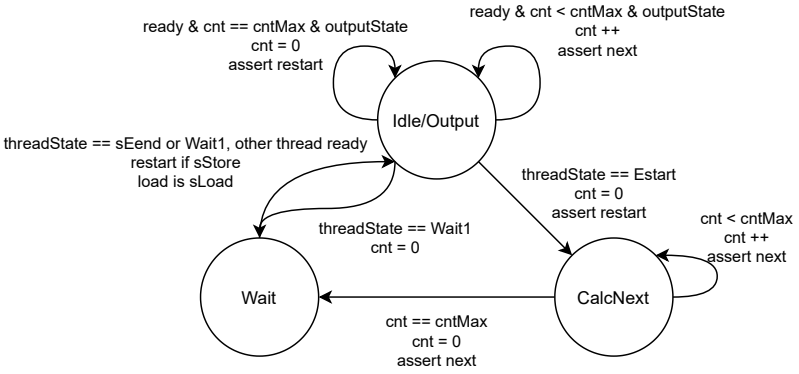

Figure 5.9: State machine for an IJK generator. `next` signals that the next coordinates (i, j, k) should be output. `restart` signals that the generator should restart its output sequence at the saved coordinate pair, and `load` signals that the saved coordinate pair should be set to the value currently output by the other thread.

Once R-type instructions are being executed in thread 0, thread 1 will start performing memory loads at index ELEMS_PER_VSLOT up to and including 2 ELEMS_PER_VSLOT -1. Once thread 0 moves begins to store data, it again iterates through vector indices starting at index 0, for each store instruction in the packet.

Moving to the load section, thread 0 will now start loading elements from address 2 ELEMS_PER_VSLOT. When thread 1 is finished storing its results, it will start loading data from index 3 ELEMS_PER_VSLOT, each thread accessing higher indices until an $NDOF$ or $NELEM$ long vector has been accessed (depending on whether the instruction length was `ndof` or `nelemvec`).

Register file access

When data is written into a register file from the execute stage, it may either be written into the V- and X-register files of the executing thread, or the S-register file. If a value is written into the S-register file, the value at index [0] of the writeback data is written. If data is written into the X-register file, the first XREG_DEPTH values are written into the register file, and if the destination register is a V-register, the entire 24-wide write data vector is written into the destination register.

When data is received from the memory stage, only the V- and X-register files of the connected thread are accessible. It is thus not possible to load or store data directly from the S-register file.

5.4.2 Branch generation

As previously mentioned, branches are decided in the decode stage where the central S-register file is located. When a branch instruction is encountered, the values at `Sreg[rs1]` and `Sreg[rs2]` are compared, using the comparison encoded in the instruction. If the comparison evaluates to true, the signal `branch` is returned to the IF stage alongside the branch target. Figure 5.10 shows a timing diagram where a branch not-taken and branch taken scenario both play out. Note that the `branch` signal takes precedence over the `iload` signal.

5.5 Memory stage

Given the memory-intensive process that topology optimization and matrix-vector product calculation is, it has been a focus of this project to implement a memory stage which allows for multiple parallel memory loads. The memory stage has been implemented with a focus on accelerating memory accesses during `applyStateOperator`, as this is the critical section of the program.

As previously mentioned in section 4.3, it has been chosen to traverse the grid using an 8-coloring. In addition to using an 8-colouring for the grid, it has also been chosen to map element's DOFs using an 8-coloring. Doing this, each element is sure to have exactly 3 DOFs in each colouring. This was not the case with the algorithm previously used by the TopOpt team.[1]

[1] See appendix A.2 for a comparison of DOF mappings.

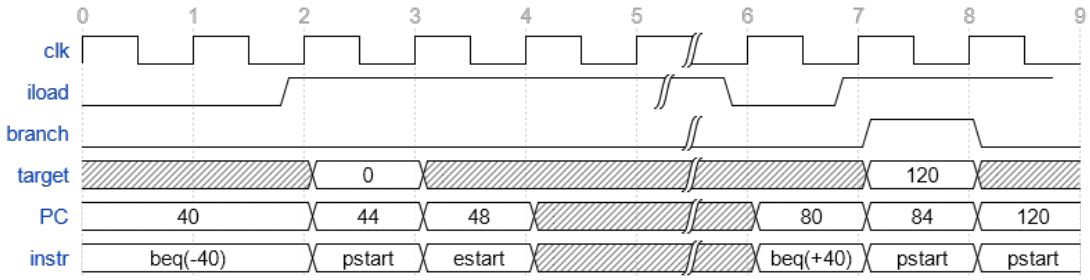

Figure 5.10: A timing diagram showing how branch instructions are evaluated. At $t = 2$ the branch is not taken, but at $t = 7$ the branch is taken. If the `branch` and `iload` signals are asserted at the same time, the branch update takes precedence

Since all element's DOFs are mapped using an 8-coloring, the memory module has been implemented as an 8-way interleaved memory, allowing up to 8 simultaneous read or write operations using 8 memory banks. This means an element's 24 DOFs may be retrieved over 3 clock cycles.

From now on, the "colourings" may also be referred to by the memory bank that they map into. A "0-coloured" element or DOF is thus one where the element / DOF can be found in memory bank 0.

5.5.1 Mapping coordinates to elements

Using an 8-coloring for both elements and DOFs, a scheme must be devised to map an element's coordinates (x, y, z) to the colouring to which that element belongs. Element are assigned to a colouring, starting with the element at $(0, 0, 0)$ which is assigned to colour 0. Iterating through the grid along the y, z and x dimensions, incrementing the coordinate by 2 on every step, each element is assigned to a colouring.

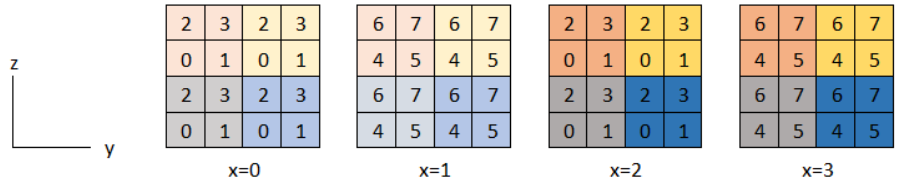

Figure 5.11: Illustration of a $4 \times 4 \times 4$ grid, with all elements marked with the colouring that they belong to. Elements highlighted with the same colour belong to the same element block and are thus present at the same row of memory.

Given an element at coordinates (i, j, k), those coordinates can be mapped to a specific $2 \times 2 \times 2$ block of the grid which contains one element of each colour. These blocks are numbered continously, such that the first block contains the elements which are stored in the first row of memory in banks 0-7, the second block the next 8 elements, etc. Knowing which colour of the grid is currently being traversed, it is then possible to select the correct element from that block. Figure 5.11 shows an example $4 \times 4 \times 4$ grid where the colouring of each element is denoted by a number, and the element block an element belongs to is indicated by the highlight colour. These elements would be mapped into memory in the manner shown in fig. 5.12.

Given a grid where all grid dimensions are even, the index of the element block to which the element (i, j, k) belongs is given by

$$\frac{j}{2} + \frac{k}{2}\left\lceil \frac{NELY}{2} \right\rceil + \frac{i}{2}\left\lceil \frac{NELY}{2} \right\rceil \left\lceil \frac{NELZ}{2} \right\rceil \tag{5.6}$$

Given this value, the exact index into a vector can be obtained by multiplying the index by 8 (in practice a bitshift of three), and then adding the value of the memory bank which sould be accessed.

For this project, it has been chosen to only use even grid dimensions to simplify the lookup process. If odd grid dimensions were used, the lookup process would be more complicated.

Address	+0	+1	+2	+3	+4	+5	+6	+7
0	0	1	2	3	4	5	6	7
8	0	1	2	3	4	5	6	7
16	0	1	2	3	4	5	6	7
24	0	1	2	3	4	5	6	7
32	0	1	2	3	4	5	6	7
40	0	1	2	3	4	5	6	7
48	0	1	2	3	4	5	6	7
56	0	1	2	3	4	5	6	7

Figure 5.12: Illustration of the elements in fig. 5.11 being mapped into the 8 memory banks. Notice that all elements in the same element block are mapped to the same row of memory.

Element's DOFs are numbered in a fashion similar to the one shown in fig. 5.11. Nodes are assigned to a colouring based on their "nodal coordinates". Figure 5.13 shows the nodes in the bottom layer of the $4 \times 4 \times 4$ grid. Only colours 0-3 are present in this layer, whereas colours 4-7 are present in the layer separating elements at $x = 0$ from elements at $x = 1$. Each node is marked in the manner $x.y$ where x indicates which colour / memory bank the node belongs to, and y indicates its position in that colouring. Table 5.2 shows how some of the elements of that grid would be mapped into memory

To obtain the vector index where an element's DOFs are stored, eight separate equations must be used to obtain the correct index of the first DOF for each corner.

$$\text{Bank } 0: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lceil \frac{NY}{2} \right\rceil + \left\lceil \frac{i}{2} \right\rceil \left\lceil \frac{NZ}{2} \right\rceil \left\lceil \frac{NY}{2} \right\rceil \tag{5.7}$$

$$\text{Bank } 1: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lfloor \frac{NY}{2} \right\rfloor + \left\lceil \frac{i}{2} \right\rceil \left\lceil \frac{NZ}{2} \right\rceil \left\lfloor \frac{NY}{2} \right\rfloor \tag{5.8}$$

$$\text{Bank } 2: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lceil \frac{NY}{2} \right\rceil + \left\lceil \frac{i}{2} \right\rceil \left\lfloor \frac{NZ}{2} \right\rfloor \left\lceil \frac{NY}{2} \right\rceil \tag{5.9}$$

$$\text{Bank } 3: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lfloor \frac{NY}{2} \right\rfloor + \left\lceil \frac{i}{2} \right\rceil \left\lfloor \frac{NZ}{2} \right\rfloor \left\lfloor \frac{NY}{2} \right\rfloor \tag{5.10}$$

$$\text{Bank } 4: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lceil \frac{NY}{2} \right\rceil + \left\lceil \frac{i}{2} \right\rceil \left\lceil \frac{NZ}{2} \right\rceil \left\lceil \frac{NY}{2} \right\rceil \tag{5.11}$$

$$\text{Bank } 5: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lfloor \frac{NY}{2} \right\rfloor + \left\lceil \frac{i}{2} \right\rceil \left\lceil \frac{NZ}{2} \right\rceil \left\lfloor \frac{NY}{2} \right\rfloor \tag{5.12}$$

$$\text{Bank } 6: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lceil \frac{NY}{2} \right\rceil + \left\lceil \frac{i}{2} \right\rceil \left\lfloor \frac{NZ}{2} \right\rfloor \left\lceil \frac{NY}{2} \right\rceil \tag{5.13}$$

$$\text{Bank } 7: \quad \left\lfloor \frac{j}{2} \right\rfloor + \left\lceil \frac{k}{2} \right\rceil \left\lfloor \frac{NY}{2} \right\rfloor + \left\lceil \frac{i}{2} \right\rceil \left\lfloor \frac{NZ}{2} \right\rfloor \left\lfloor \frac{NY}{2} \right\rfloor \tag{5.14}$$

To obtain the correct index into a given vector, the value obtained with one of eqs. (5.7) to (5.14) must be multiplied by 24, and the node's colour's value is added. The DOFs are then stored at the given index, as well as at $index + 8$ and $index + 16$ (the next 2 positions in a given memory bank). For example, as table 5.2 shows, the 2-coloured DOFs of the element at $(0, 2, 0)$ are stored at memory address $24 \cdot 1 + 2 = 26$, $26 + 8 = 34$ and $26 + 16 = 42$.

5.5.2 Generating memory indices

Given the various S-type modifiers outlined in table 4.9, and the equations for generating indices presented in the previous section, a number of different modules had to be designed to generate the correct indices into a given vector.

Since vectors may either be traversed linearly using the `.vec` modifier, or elementwise according to the 8-colouring, the memory stage supports a number of different vector index generation schemes. Figure 5.14 shows a drawing of the memory stage. The memory stage uses a ready/valid handshake between all modules expect the memory writeback stage and register file, as the register file is assumed always ready.

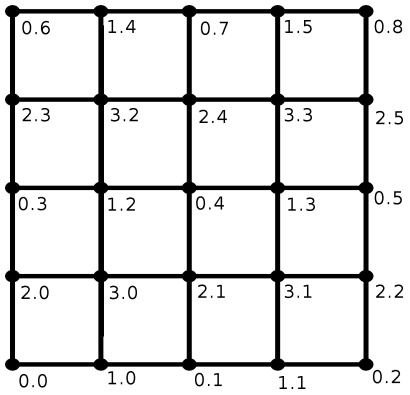

Figure 5.13: Nodes in the bottom layer of a $4 \times 4 \times 4$ grid, marked with their colouring and index in that specific colouring.

Address	+0	+1	+2	+3	+4	+5	+6	+7
0	0.0(x)	1.0(x)	2.0(x)	3.0(x)	4.0(x)	5.0(x)	6.0(x)	7.0(x)
8	0.0(y)	1.0(y)	2.0(y)	3.0(y)	4.0(y)	5.0(y)	6.0(y)	7.0(y)
16	0.0(z)	1.0(z)	2.0(z)	3.0(z)	4.0(z)	5.0(z)	6.0(z)	7.0(z)
24	0.1(x)	1.1(x)	2.1(x)	3.1(x)	4.1(x)	5.1(x)	6.1(x)	7.1(x)
32	0.1(y)	1.1(y)	2.1(y)	3.1(y)	4.1(y)	5.1(y)	6.1(y)	7.1(y)
40	0.1(z)	1.1(z)	2.1(z)	3.1(z)	4.1(z)	5.1(z)	6.1(z)	7.1(z)

Table 5.2: Presentation of how the first 2 nodes of each colouring are mapped into memory. Brackets (x), (y) and (z) indicate that this is the DOF in the x, y or z-direction for the given nodes.

The read queue holds the destination register and S-type modifier of an instruction. When the memory load has been performed, the destination register is dequeued. The write queue holds the 8 write data that should be stored in memory, as well as the S-type modifier of the instruction.

The address generator takes up to 8 vector indices as input, as well as the encoded base address of the operation. Alongside each of the 8 input indices are 8 `validIndex` signals. If this signal is low, read/write operations to the corresponding address are disabled.

The address generator decodes the base address from the instruction and adds all indices to the decoded base address. It then reorders these addresses, such that the address output on line 0 ends in 0b000, the address on line 1 ends in 0b001, etc. This ensures that every memory operation reaches the correct memory bank. Only valid indices are reordered, and in case two indices attempt to access the same memory bank (which should never happen), the index input at a higher position in the input vector is given priority. An example of this reordering is shown in table 5.3

Index	Input index	Valid	Output address	Valid
0	`0b000`	F	`0b1000`	F
1	`0b011`	T	`0b1001`	T
2	`0b011`	F	`0b1011`	F
3	`0b001`	T	`0b1011`	T
4	`0b100`	F	`0b1100`	F
5	`0b111`	T	`0b1111`	F
6	`0b010`	F	`0b1010`	F
7	`0b101`	F	`0b1111`	T

Table 5.3: Index reordering when the input base address decodes to `0b1000`. Valid input indices are reordered such that they are output on the index matching their LSB. Invalid inputs are not reordered. If a valid index does not appear at the correct position, the output uses that index to generate the address, but the output is invalid. F indices that a signal is invalid, T indicates that it is valid.

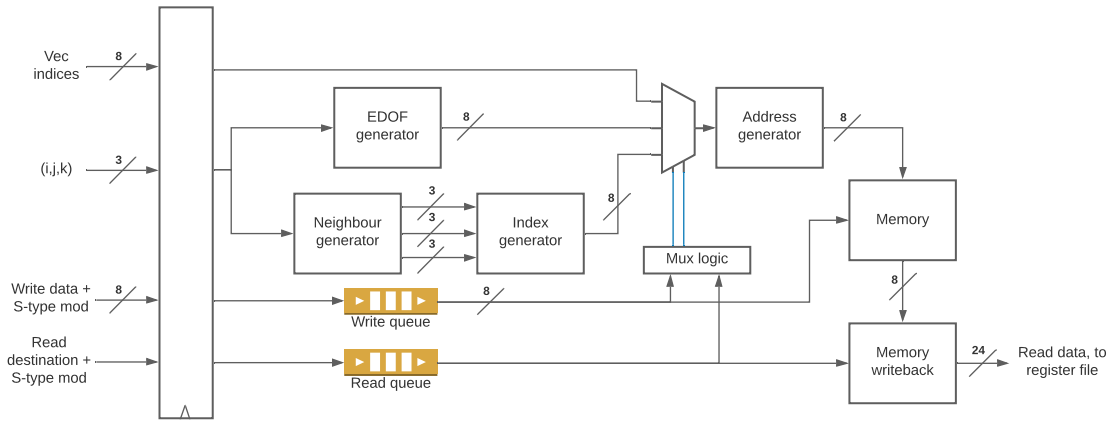

Figure 5.14: Memory stage of the ASIP. Based on the current S-type modifier present at the read/write queue head, the correct index generator is connected to the address generator. Not pictured are ready/valid signals, load/store flag and signals carrying the load/store operations' base address. Widths indicated are not bitwidths, but the number of signals transferred between each module. Blue lines are control signals.

The module connected to the address generator is determined by the S-type modifier present at the head of the read/write queue when performing load/store operations.

When processing instructions with the S-type modifier `.vec`, the 8 indices to access are generated inside of the connected thread. When a read/write is issued, these are transmitted directly to the address generator, alongside their `validIndex` signals, which are also generated in the thread.

If an instruction with S-type modifier `.dof` or `.fdof` is processed, the input goes to the element DOF (EDOF) generator. Given the input coordinates (i, j, k), it uses eqs. (5.7) to (5.14) to calculate the indices of that element's DOFs. Over three clock cycles, it outputs all 24 indices such that all 8 outputs on each clock cycle map to different memory banks.

If instructions with S-type modifiers `.elem, .sel, .fcn, .edn1` or `.edn2` are processed, these go to the neighbour generator. Given coordinates (i, j, k), this module will output the indices of three face or edge neighbours on each clock cycle, taking two clock cycles for each of the `.fcn, .edn1` and `.edn2` modifiers. If the modifier is `.elem` or `.sel`, the input is immediately forwarded on the first output line, and the other two outputs are signalled invalid.

The index generator takes up to three coordinate pairs and maps these to their corresponding indices into a NELEM long vector by using eq. (5.6). Up to three indices are transmitted to the address generator, alongside `validIndex` signals. For all remaining outputs, `validIndex` is tied low.

The memory module takes the addresses received from the address generator and uses these to access each of the 8 memory banks. If a write operation is performed, the data at the head of the write queue is dequeued and written into memory. If a read operation is performed, the read data is forwarded to the memory writeback stage.

The memory writeback stage works in a manner similar to the execute writeback stage. Using the S-type modifier present at the head of the read queue, it collects outputs from the memory module until an entire instruction has been processed. The read data is then written into the V- or X-register file in the connected thread.

5.6 Control module

The central control module is used to ensure that data hazards are avoided and the ASIP functions correctly.

Three different kinds of data hazards have been identified in the system, that must be avoided to ensure correct execution of R-type instructions:

- Calculation hazard: This data hazard arises when an incoming instruction depends on the result of an operation which is currently being computed in the MPU.

- Opcode hazard: If the incoming instruction does not have the same opcode as the currently executing instruction
- Single-cycle instruction hazard: If an instruction with R-type modifiers `SX, XX` or `SS` is started, a one-cycle pipeline stall must be inserted to ensure that the next instruction will raise an opcode hazard if necessary. This is because these instructions only take one clock cycle to decode, and the executing thread must be stalled until an opcode hazard can be raised.

The stall signal going into the execute stage is the same signal as the stall signal going into the executing thread in the decode stage. To avoid calculations hazards, the control module always peeks at the 4 elements at the head of the destination queue. If any destination register in the queue matches the source register of an incoming operand, the stall signal is asserted.

Opcode hazards and single-cycle hazards are avoided by peeking at the incoming opcode/R-type modifier and the opcode/modifer currently present in the execute stage. If these do not match, the stall signal is asserted until the destination queue has been emptied.

When memory operations are processed, the control module ensures that the connected thread does not move on from the load or store states until all load/store operations have been performed (the read/write queue respectively is empty).

When an instruction packet is finished, this module also asserts the `iload` signal to the IF stage, incrementing the program counter until a new instruction packet has been loaded into the instruction buffer in the Decode stage.

5.7 Assembler

As well as the hardware implementation of the ASIP, an assembler has been written for the ISA used by the ASIP. The assembler can either read a text file or an input string holding a program.

The assembler contains a large amount of error checks, aiming to ensure that valid instruction packets are generated. This includes ensuring that load/store/execute instructions are placed at the correct position in a packet, only allowing some combinations of opcodes and modifiers (e.g. disallowing the `KV` R-type modifier for all other opcodes than `mac`) and limiting the S-type modifiers available based on the current O-type length.

All of these limitations have to be placed in the assembler, as the ASIP has no way of dealing with illegal instructions. If an instruction with a bad opcode/modifier combination is given, the behavior of the ASIP is undefined. If the ASIP were to be connected to a computer via the PCI connector on the development board, it would be possible to raise an exception, allowing the operating system / a user program to manage bad instructions.

6. Results and Evaluation

This chapter presents the results of the project. In the first section, hardware synthesis results are presented. Seeing that some modules do not yet meet the timing requirements, some suggestions are made to changes that might improve timing closure. Secondly, benchmarks comparing the ASIPs performance and the performance of a modern laptop are presented. The ASIP and the laptop are compared on three different benchmarks, representing the three most common operations used in solving TO problems with the SIMP approach. Finally, the results and benchmarks are evaluated, and areas for improvement are highlighted.

6.1 Synthesis results

For this project, the Altera Cyclone V GT FPGA Development Kit has been targeted. This development kit incorporates a Cyclone V GT FPGA, model number 5CGTFD9E5F35, with 301K logic elements, 454K registers and 12 MB of on-chip memory. It also incorporates 342 DSP blocks, each of which has a 27×27 hard multiplier. The development board/FPGA has been chosen for its high number of LEs and large amount of on-chip memory, removing the need for an external memory interface in this initial design.

The design has been synthesized with Intel Quartus Prime version 20.1. The synthesis numbers presented are for a design with 8 processing elements, 32 V-registers in each thread, an X-register depth of 8 and 16 scalar registers. Synthesis has been performed with an accelerator made to operate on a $6 \times 6 \times 6$ grid, with an instruction memory size of 1024 instructions. The gridsize only has a meaningful impact on the amount of memory bits used in the memory stage and not on the remaining hardware consumption. This relatively small gridsize was used since synthesis at this size take less time than synthesizing an ASIP operating on larger grids.

Table 6.1 shows the final resource usage numbers for the synthesized accelerator.

Module	LEs	Registers	Memory bits [k]	DSP Blocks	Fmax (slow/fast) [MHz]	
Control	137	12	0	0	650.2	877.2
Decode	9375	10076	178.9	0	106.2	214.6
Per thread	2574	4283	88.8	0	134.4	254.1
Fetch	707	32	0	0	356.2	799.3
Execute	46209	22706	27.4	288	81.2	169.1
Per PE	4222	2780	3.4	36	85.1	189.2
Writeback	679	1368	0	0	103.9	245.9
Memory	5464	3223	725.3		78.0	162.3
Full design	61519	47888	934.4	288	67.1	123.9

Table 6.1: Resource utilization values for the final ASIP, as well as resource utilization for some submodules.

The large number of multipliers / DSP blocks used in the execute stage is mainly due to the division algorithm used. Newton-Raphson division relies heavily on multiplication, meaning that each PE uses 36 multipliers, 32 of which are used for division. This naturally limits the number of PEs that can be instantiated when multiplication uses the on-chip DSP blocks.

The fetch stage does not use any memory bits as the instruction memory has been instantiated as asynchronous-read memory by using LEs instead. Since the synthesis tool will optimize away any empty memory locations, the instruction memory was filled with random numbers for the

purposes of the initial synthesis. Seeing that the number of LE's is less than 1024, I assume that the synthesis tool was able to perform some optimization, bundling multiple values into some LEs.

In general, it is evident that the decode, execute and memory stages make up the majority of the design. Some memory bits are used in the execute stage, since the destination queues use memory in their implementation. The estimated number of registers required was initially higher, but it dropped significantly after synthesizing the design. This is because all DSP blocks and memory blocks have built-in registers, freeing up registers for the remaining design.

6.1.1 Timing closure

To optimize the design, an attempt has been made to achieve timing closure for the design. Timing closure is achieved when a design is able to operate at a given clock frequency.

To determine if the design meets timing closure, the accelerator has been synthesized and a timing report has been generated. A timing report shows paths in the design where either setup or hold-time requirements are not met and gives recommendations on how to fix these issues. Quartus Prime operates with 4 sets of operating conditions (also known as timing corners). By simulating circuit performance in the worst-case and best case scenarios, Quartus is able to tell whether the design will function correctly on a physical FPGA [30]. Since the development board incorporates a 100 MHz clock, it was attempted to reach an operating frequency of 100 MHz in all 4 sets of operating conditions.

For example, synthesizing the decode stage with V- and X-register files mapped to on-chip memory, but the S-register file mapped to physical registers, the critical path in the design went through the S-register file. In this case, the 2 slowest timing corners were failing.

Updating the S-register file to also use on-chip synchronous-read memory removed this critical path, providing timing closure. Table 6.2 shows the maximum operating frequency (Fmax) for each timing corner, before and after optimizing the design.

	Fmax [MHz]
Before optimizing	
Slow 85C model	91.29
Slow 0C model	92.17
Fast 85C model	174.49
Fast 0C model	195.16
After optimizing	
Slow 85C model	106.19
Slow 0C model	110.00
Fast 85C model	185.67
Fast 0C model	214.55

Table 6.2: Timing data for the Decode stage before and after mapping the S-register file to on-chip memory.

As it is evident from table 6.1, timing closure on the full design has not yet been achieved. The worst failing path is currently the path through the address generator in the memory stage, where addresses are reordered such that they access the correct memory bank. Inserting a pipeline stage before the reordering step should help to alleviate this.

In addition to this, another failing path goes from the decode stage, through the control module and into the execute stage's stall signal. Reworking some of the data hazard avoidance logic is probably necessary to meet timing closure on this path.

6.2 Benchmarks

To compare the performance of the accelerator and the C-code used at DTU Mechanical Engineering, three different benchmarks were run to compare the runtime of three functions from the C-code with their corresponding implementations on the ASIP. `applyStateOperator`, the function used to calculate matrix-vector products, `innerProduct` which calculates the inner product $\mathbf{a} \cdot \mathbf{b}$

of two vectors, as well as `applyDensityFilter` which performs density filtering as described in section 4.2.

These benchmarks were chosen since they each focus on a different part of the implementation. Each of the functions utilizes different R-type opcode/modifier combinations, S-type modifiers and O-type lengths. `applyStateOperator` tests matrix-vector products, the main focus of acceleration. Computing the inner product of two vectors is a task that is simple to perform sequentially, meaning that the cache controller in a modern computer should be able to efficiently pre-load data. Density filtering on the other relies on data access patterns that may not be intuitive to the cache controller in a modern computer, meaning that the accelerator may perform better at this benchmark.

All benchmarks of the C functions have been run on a Windows laptop with an Intel i7-8550U 4-core/8-thread processor with a nominal clock speed of 1.8 GHz, capable of achieving up to 4.0 GHz using Intel's "Turbo Boost" technology. The laptop has 16 GB of RAM. All function calls were timed using the function `clock_gettime` from the C standard library's `time.h`. Each function call was measured 50 times, and the average of these runtimes was taken as the final result.

Simulations of the accelerator were performed using the ChiselTest Scala library which generates a cycle-accurate model of the design described in Chisel. The accelerator was simulated with 8 processing elements, 32 V-registers separated into 4 V-register slots, 8 X-registers, each of which is 8 elements deep, and 16 scalar registers. This design size was chosen since it is not possible to map 12 PEs (the next number of PEs which is possible to use) onto the FPGA when each division circuit takes up so many of the on-chip multipliers.

The functions were rewritten in assembly language[1], and a simulation was run for up to 2000 clock cycles. A value-change dump (VCD) file of each simulation was generated, and from the simulation data, the number of clock cycles required to execute the given instruction packets was extrapolated.

As an example, fig. 6.1 shows the waveform obtained when simulating `applyStateOperator` on the accelerator. The marker "Setup" indicates the end of the one-time section where constants are loaded into s-registers and the output vector is cleared. Markers "Load", "Exec" and "Store" indicate the end of the load, execute and store sections of the main loop. No matter the gridsize, it is known that NUM_PROCELEM elements are processed on each iteration, meaning that the total number of clock cycles to execute a matrix-vector product with a given configurations is

$$cc_{total} = cc_{setup} + cc_{load} + cc_{store} + \left\lceil \frac{NELEM}{NUM\_PROCELEM} \right\rceil cc_{exec}$$

Since the time required to execute the R-type instructions in the packet is greater than the time necessary for the other thread to perform its store and load operations, only the first load and final store operations need to be taken into account. Since no other processes run on the accelerator, all execution times should be deterministic.

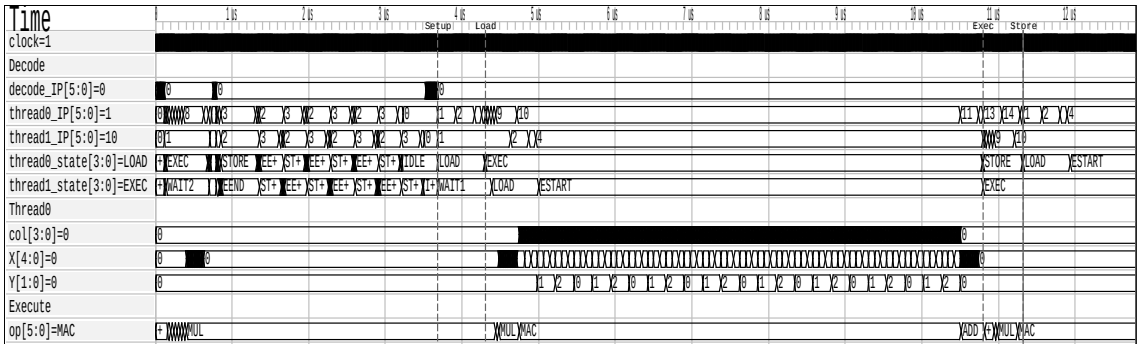

Figure 6.1: Waveform obtained when evaluating `applyStateOperator`. The markers Setup, Load, Exec, Store represent the end of their respective sections, and have been used to extrapolate the time necessary to complete a full matrix-vector product.

The runtime has been calculated with a 100 MHz and 67 MHz clock. The 100 MHz clock has been chosen since this is the operating frequency of the development board's clock, and it should be possible to optimize the design to achieve timing closure at this frequency. The design is currently able to operate at 67 MHz, hence this frequency has also been used for the benchmark.

[1]The assembly-language versions can be found in the `src/resources/programs` directory in the GitHub repository

All tests were run on a number of cubic $n \times n \times n$ grids, where

$$n \in \{10, 15, 20, 25, 30, 40, 50, 75, 100\}$$

No larger values were chosen since, as the result show, the time to compute matrix-vector products at larger grid sizes increases exponentially. There is no limit on the problems that the accelerator should theoretically be able to handle, given enough time and memory.

Runtimes and speedup values for `applyStateOperator` can be found in fig. 6.2 and fig. 6.3 respectively. The average speedup achieved at 100 MHz is 6.3, and at 67 MHz it is 4.2.

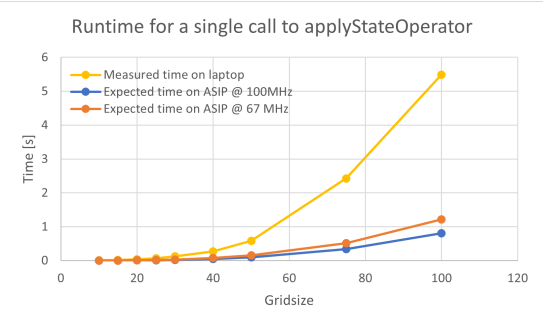

Figure 6.2: Comparison of measured runtime on laptop vs. expected runtime on the accelerator when executing `applyStateOperator`.

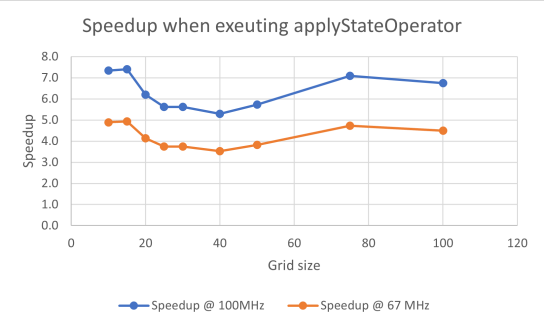

Figure 6.3: Speedup of the accelerator over the laptop when executing `applyStateOperator`. The average speedup at 100 MHz is 6.3, and the average speedup at 67 MHz is 4.2.

Computing the inner product of two $NDOF$ long vectors, the average speedup at 100 MHz is 1.5. Operating at 67 MHz the average speedup is 1.0. Graphs showing runtime and speedup can be found in fig. 6.4 and fig. 6.5 respectively.

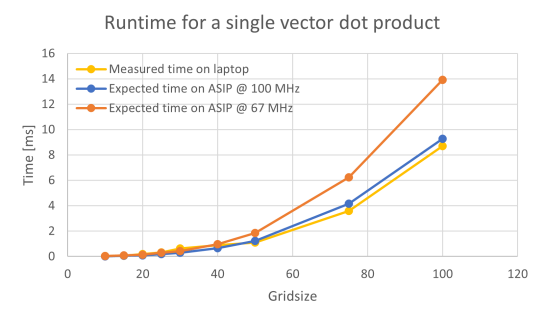

Figure 6.4: Comparison of measured runtime on laptop vs. expected runtime on the accelerator when calculating the inner product of two $NDOF$ long vectors.

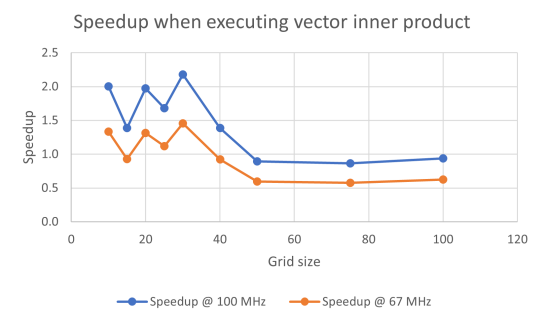

Figure 6.5: Speedup of the accelerator over the laptop when calculating the inner product of two $NDOF$ long vectors. The average speedup at 100 MHz is 1.5, and the average speedup at 67 MHz is 1.0.

When applying a density filter, the accelerator was on average 1.7 times faster than the laptop when operating at 100 MHz. When operating at 67 MHz, the average speedup was 1.1. Graphs of runtime and speedup can be found in fig. 6.6 and fig. 6.7 respectively.

Achieving a 6.3x speedup when computing matrix-vector products at 100 Mhz is very good and is comparable to the speedups with GPUs and CPUs found in [8] and [9]. Operating at 67 MHz the speedup is not as large, but still shows significant improvement over the laptop.

When computing the inner product of two vectors, the laptop is faster than the accelerator once the grid size exceeds 50. Since this is the one operation where vectors are traversed sequentially, it makes sense that an ordinary processor is very efficient at this task. The much higher clock speed combined with an efficient cache controller allows the laptop to perform better than the accelerator on very long vectors. While the performance of the accelerator operating at 100 MHz is comparable, the accelerator operating at 67 MHz is much slower.

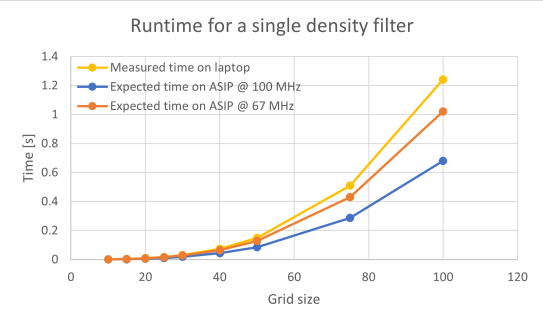

Figure 6.6: Comparison of measured runtime on laptop vs. expected runtime on the accelerator when applying a density filter.

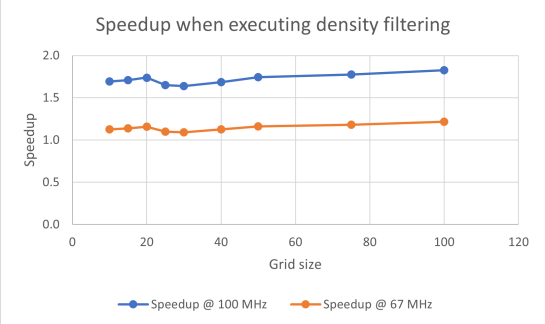

Figure 6.7: Speedup of the accelerator over the laptop when applying a density filter. The average speedup at 100 MHz is 1.7, and the average speedup at 67 MHz is 1.1.

When applying a density filter, the speedup is relatively constant for all gridsizes at a given frequency. The increased frequency of the laptop's processor is most likely offset by the added complexity in efficiently caching the element densities.

6.3 Discussion

During the initial design process, the V- and X-register files had been implemented with registers on the FPGA. Due to the large size of the V-registers, this took up so many registers that it was not possible to map the entire design onto the FPGA. To solve this problem, all register files were instead implemented using on-chip memory. The FPGAs on-chip memory is however synchronous read, whereas hardware registers are asynchronous read, meaning that all read data from registers was delayed by one clock cycle. This introduced some timing issues in the branching, forwarding and data hazard avoidance logic. While an attempt has been made to fix these issues, some edge cases still appear that cause wrong results to be calculated.

To the best of my knowledge, fixing these problems should take approximately 2 weeks. However, while the implementation does not provide mathematically correct results in all cases, it can still be used to benchmark the performance of the accelerator, since none of the remaining issues will affect runtime by more than a 1 or 2 clock cycles per instruction.

The benchmarks that have been run do not prove that the accelerator is actually faster than the laptop at solving TO problems. However, if the benchmarks can be believed, the accelerator is 6.3x faster at computing matrix-vector products if operating at 100 MHz. Assuming that 90% of computation time is spent on matrix-vector products [8], for the accelerator to end up being slower than the computer, it would have to be, on average, 9 times slower than the laptop at the remaining operations. Since this is not what is observed in the other benchmarks, it can assumed the accelerator will be faster than the computer.

The chosen division algorithm relies heavily on multiplication, which means that a vast majority of the FPGAs hard multipliers are tied up in the division pipeline. This is an inefficient use of the multipliers, as only a small fraction of all computations in the project are divisions. While the division algorithm was chosen for its relatively simple implementation, I did not properly envision how many multipliers would be necessary to implement a full division circuit.

Implementing division in software, or with a hardware implementation that requires fewer multipliers, would free up resources such that additional processing elements could be instantiated. With additional PEs, matrix-vector calculations would be much faster. This would require a much larger V-register file, which in turn requires more time for memory loads. It is still believed that this would lead to a major gain in processing speed. Removing hardware division from the accelerator should be one of the first changes to be implemented in the future.

The current implementation of the accelerator uses only on-chip memory. To process larger problems, a memory interface to external RAM must be added to the design. To still operate efficiently on larger problems, this memory interface should incorporate data-prefetching to avoid stalling while waiting on data. Since the memory access patterns used in this project are highly deterministic, implementing such a prefetching system should be feasible.

7. Conclusion

In this project, a hardware accelerator for topology optimization has been designed. The accelerator has been implemented as an application-specific instruction set processor with a custom instruction set, optimized for computing the matrix-vector products used in TO problems. The ASIP is a dual-issue processor, capable of efficiently computing matrix-vector and vector-vector operations.

While it was not possible to implement a working accelerator on an FPGA due to time constraints, a number of simulations have been run which indicate that the ASIP is much faster at computing the matrix-vector products required for topology optimization than a modern laptop computer. The accelerator was able to achieve a 4.2x - 6.3x speedup over the laptop, indicating that hardware acceleration of TO problems is viable.

Using Intel Quartus, timing reports have been generated and the design has been optimized for speed. While timing closure has been achieved for some modules in the design, the full design is not yet capable of operating at 100 MHz. Further work on improving the device pipeline should be able to meet timing closure.

A 6.3x speedup is promising and shows that hardware acceleration of topology optimization on FPGAs is worth further research. Measuring the speedup against a more powerful desktop computer or server should be the first step, to see if the accelerator is still able to outperform better-equipped computers. Secondly, a version of the accelerator which implements division in software should be designed, to see how much better performance is attainable with more PEs.

8. Outlook

The future of hardware acceleration of TO problems is promising. The hardware accelerator described in this thesis is a good starting point and may well be further optimized. One such optimization is to discard the hardware division logic, freeing up hard multipliers. Doing this would allow more processing elements to be instantiated, speeding up matrix-vector product calculations.

To fully realize the potential of the accelerator, it must be connected to more memory. The current implementation is restricted by the 12 MB of on-chip memory present on the FPGA. Creating an interface between the FPGA and the memory of a host computer connected via the PCI-connector on the development board would open up a large array of possibilities, such as much more available memory and multiple FPGAs operating on the same problem. This interface should leverage a data-prefetching module to avoid stalling on memory loads.

In addition to optimizing this accelerator, it would be interesting to use ASIP design tools such as Synopsys ASIP Designer. These tools can generate an ASIP for any computable problem, and they may also generate a compiler targeting the generated ISA, allowing programmers targeting the accelerator to write in C or C++ instead of assembly. If these tools are able to generate an accelerator of equal or better performance, this may be a very promising tool for future development. As an alternative to using an ASIP designer, high-level synthesis tools such as Xilinx Vitis are also able to generate a hardware description from C++ code. Using high-level synthesis to design an accelerator might also lead to performance improvements.

Acknowledgements

I would like to thank my supervisors Martin Schoeberl (DTU Compute), Erik Träff (DTU Mechanical Engineering) and Niels Aage (DTU Mechanical Engineering) for guiding me at our many project meetings over the last couple of months. Your aid has been instrumental in guiding me in the right direction.

9. List of abbreviations

The following abbreviations are used throughout the report:

- ALU: Arithmetic logic unit
- ASIP: Application-specific instruction set processor
- CPU: Central processing unit
- FPGA: Field-programmable gate array
- GPU: Graphics processing unit
- IF: Instruction fetch stage
- ISA: Instruction set architecture
- LE: Logic element
- LUT: Lookup table
- MPU: Matrix processing unit
- PC: Program counter
- PCG: Preconditioned conjugate gradient method
- PE: Processing element
- SIMP: Simple isotropic material with penalization
- TO: Topology optimization
- VCD: Value-change dump

Bibliography

[1] N. Aage, E. Andreassen, B. S. Lazarov, and O. Sigmund, "Giga-voxel computational morphogenesis for structural design," *Nature*, vol. 550, no. 7674, pp. 84–86, 2017.

[2] Z. Liao, Y. Zhang, Y. Wang, and W. Li, "A triple acceleration method for topology optimization," *Structural and Multidisciplinary Optimization*, vol. 60, no. 2, pp. 727–744, 2019.

[3] S. Y. Kim, I. Y. Kim, and C. K. Mechefske, "A new efficient convergence criterion for reducing computational expense in topology optimization: reducible design variable method," *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*, vol. 90, no. 6, pp. 752–783, 2012.

[4] C.-Y. Lin and J.-N. Chou, "A two-stage approach for structural topology optimization," *Advances in Engineering Software*, vol. 30, pp. 261–271, apr 1999.

[5] O. Amir, N. Aage, and B. S. Lazarov, "On multigrid-CG for efficient topology optimization," *Structural and Multidisciplinary Optimization*, vol. 49, no. 5, pp. 815–829, 2014.

[6] C. Augarde, A. Ramage, and J. Staudacher, "An element-based displacement preconditioner for linear elasticity problems," *Computers & Structures*, vol. 84, pp. 2306–2315, dec 2006.

[7] J. Martínez-Frutos and D. Herrero-Pérez, "Efficient matrix-free GPU implementation of Fixed Grid Finite Element Analysis," *Finite Elements in Analysis and Design*, vol. 104, pp. 61–71, 2015.

[8] D. Herrero-Pérez and P. J. Martínez Castejón, "Multi-GPU acceleration of large-scale density-based topology optimization," *Advances in Engineering Software*, vol. 157-158, p. 103006, jul 2021.

[9] J. París, I. Colominas, F. Navarrina, and M. Casteleiro, "Parallel computing in topology optimization of structures with stress constraints," *Computers and Structures*, vol. 125, pp. 62–73, 2013.

[10] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition.* Cambridge MA: Morgan Kaufmann, 2018.

[11] A. Abbaszadeh, T. Iakymchuk, M. Bataller-Mompeán, J. V. Francés-Villora, and A. Rosado-Muñoz, "A scalable matrix computing unit architecture for FPGA, and SCUMO user design interface," *Electronics (Switzerland)*, vol. 8, no. 1, pp. 1–20, 2019.

[12] S. J. Campbell and S. P. Khatri, "Resource and delay efficient matrix multiplication using newer FPGA devices," *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, vol. 2006, pp. 308–311, 2006.

[13] L. Zhang, Y. Peng, A. Huang, and X. Hu, "A Scalable Architecture for Accelerating Multi-Operation and Continuous Floating-Point Matrix Computing on FPGAs," *IEEE Access*, vol. 8, pp. 92469–92478, 2020.

[14] P. Lenne and R. Leupers, *Customizable Embedded Processors.* Morgen Kaufmann, 2007.

[15] Synopsys, "Synopsys ASIP Designer," 2021.

[16] A. Kolesnichenko, C. M. Poskitt, S. Nanz, and B. Meyer, "Contract-based general-purpose GPU programming," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, (New York, NY, USA), pp. 75–84, ACM, oct 2015.

[17] O. Sigmund, "A 99 line topology optimization code written in matlab," *Structural and Multidisciplinary Optimization*, vol. 21, no. 2, pp. 120–127, 2001.

[18] J. Feng, J. Fu, Z. Lin, C. Shang, and B. Li, "A review of the design methods of complex topology structures for 3D printing," *Visual Computing for Industry, Biomedicine, and Art*, vol. 1, no. 1, pp. 1–16, 2018.

[19] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Horst, *Templates for the solution of linear systems*. Philadelphia PA: Society for Industrial and Applied Mathematics, 2 ed., 2008.

[20] A. Quarteroni, R. Sacco, and F. Saleri, "Iterative Methods for Solving Linear Systems," *Numerical Mathematics*, vol. 37, pp. 123–181, 2013.

[21] Microprocessor Standards Committee, *IEEE Standard for Floating-Point Arithmetic*. IEEE, 2019.

[22] ARM Limited, "ARM Developer Suite AXD and armsd Debuggers Guide, version 3.0," 2006.

[23] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design Using VHDL: a systems approach*. Cambridge: Cambridge University Press, 2016.

[24] J. Luu, C. McCullough, S. Wang, S. Huda, B. Yan, C. Chiasson, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "On hard adders and carry chains in FPGAs," *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*, pp. 52–59, 2014.

[25] K. Karuri and R. Leupers, "Application Analysis Tools for ASIP Design," in *Application Analysis Tools for ASIP Design*, pp. 11–34, New York: Springer, 2011.

[26] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: Survey and issues," *Proceedings of the IEEE International Conference on VLSI Design*, pp. 76–81, 2001.

[27] A. Waterman and K. Asanovic, "RISC-V Instruction Set Manual, Volume 1: Unprivileged ISA," 2020.

[28] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers1*, vol. 46, no. 8, pp. 833–854, 1997.

[29] C. Rohwer, "The babylonian method and higher order approximation to square roots," *The Delta*, pp. 183–188, 2010.

[30] M. Mac and C. Wysocki, "Guaranteeing Silicon Performance with FPGA Timing Models," *Intel White Paper*, pp. 1–7, 2010.

A. Appendices

A.1 Source code access

All source code developed in the process of writing this bachelor's thesis is freely available at the following GitHub repository: `https://github.com/KasperHesse/bachelors-project`. Included in the GitHub repository is a README with instructions on how to set up the ASIP and use the assembler.

A.2 Comparison of DOF mapping algorithms

The DOF mapping algorithm used in the TopOpt C-code uses the following 8 equations to generate the 8 base indices at which DOFs are stored. The 24 DOFs are stored at these base indices, and at offsets +1 and +2 from the base indices.

$$\text{DOF } 0: \quad 3(i \cdot NY \cdot NZ + k \cdot NY + (j+1)) \tag{A.1}$$
$$\text{DOF } 1: \quad 3((i+1) \cdot NY \cdot NZ + k \cdot NY + (j+1)) \tag{A.2}$$
$$\text{DOF } 2: \quad 3((i+1) \cdot NY \cdot NZ + k \cdot NY + j) \tag{A.3}$$
$$\text{DOF } 3: \quad 3(i \cdot NY \cdot NZ + k \cdot NY + j) \tag{A.4}$$
$$\text{DOF } 4: \quad 3(i \cdot NY \cdot NZ + (k+1) \cdot NY + (j+1)) \tag{A.5}$$
$$\text{DOF } 5: \quad 3((i+1) \cdot NY \cdot NZ + (k+1) \cdot NY + (j+1)) \tag{A.6}$$
$$\text{DOF } 6: \quad 3((i+1) \cdot NY \cdot NZ + (k+1) \cdot NY + j) \tag{A.7}$$
$$\text{DOF } 7: \quad 3(i \cdot NY \cdot NZ + (k+1) \cdot NY + j) \tag{A.8}$$

If e.g. a $4 \times 4 \times 4$ grid is used, the element at $(0,0,0)$ would have the 24 DOFs seen in table A.1. Similarly, the 24 DOFs obtained with the equations presented in section 5.5.1 can be seen in table A.2. In brackets after each DOF index is written the value modulo 8.

As it is clear from comparing the two mappings, the new mapping ensures that the 8 values generated on each clock cycle each have a different value modulo 8. The original mapping does not ensure this, meaning that it would be less efficient when using the 3 LSB to perform DOF loads.

Address	+0	+1	+2	+3	+4	+5	+6	+7
0	3(3)	78(6)	75(3)	0(0)	18(2)	93(5)	90(2)	15(7)
8	4(4)	79(7)	76(4)	1(1)	19(3)	94(6)	91(3)	16(0)
16	5(5)	80(0)	77(5)	2(2)	20(4)	95(7)	92(4)	17(1)

Table A.1: The 24 DOF indices for the element at $(0,0,0)$ in a $4 \times 4 \times 4$ grid, using the original TopOpt algorithm.

Address	+0	+1	+2	+3	+4	+5	+6	+7
0	0(0)	1(1)	2(2)	3(3)	4(4)	5(5)	6(6)	7(7)
8	8(0)	9(1)	10(2)	11(3)	12(4)	13(5)	14(6)	15(7)
16	16(0)	17(1)	18(2)	19(3)	20(4)	21(5)	22(6)	23(7)

Table A.2: The 24 DOF indices for the element at $(0, 0, 0)$ in a $4 \times 4 \times 4$ grid, using the new DOF mapping algorithm.

A.3 Project plan

Attached below is the project plan handed in on March 2nd. The Gantt chart in fig. A.1 was mostly adhered to. Designing the control logic for the execution stage took longer than expected, causing the time allotted for memory interface implementation to overlap with the time set aside for bug fixing. This is most likely the reason that I was unable to fix all bugs present in the design before the handin date.

Topology optimization is a class of optimization problem where large systems of differential equations are solved, typically using the finite element method (FEM). Problems of this form are ordinarily formulated as matrix equations, with the matrices easily containing thousands to millions of elements.

Mathematical problems involving matrices and vectors naturally lend themselves to parallelisation, as there is little data dependency between calculations. Since ordinary computer processors are generally suited for sequential operations, execution speed may be increased by solving the problems on hardware specially designed for parallel execution.

The purpose of this project is to design a hardware solution to speed up the calculations involved in the optimization problems encountered at the TopOpt group at DTU Mechanical Engineering. To this end, the following learning objectives have been decided upon:

- Describe different number representations and evaluate on the correct type for a given project
- Explain why hardware accelerators may provide higher throughput than what is obtained running software on a desktop PC.
- Implement necessary arithmetic operations in hardware, taking delay and area into account when selecting a design
- Design a pipelined system capable of speeding up matrix, vector and scalar operations.
- Use EDA tools to map a design onto an FPGA, find and remove bottlenecks and perform timing analysis
- Evaluate the performance achieveable with the custom hardware accelerator vs. traditional approaches using desktop PCs.

On the following page in figure A.1 is shown a Gantt chart, outlining the expected workflow for the project.

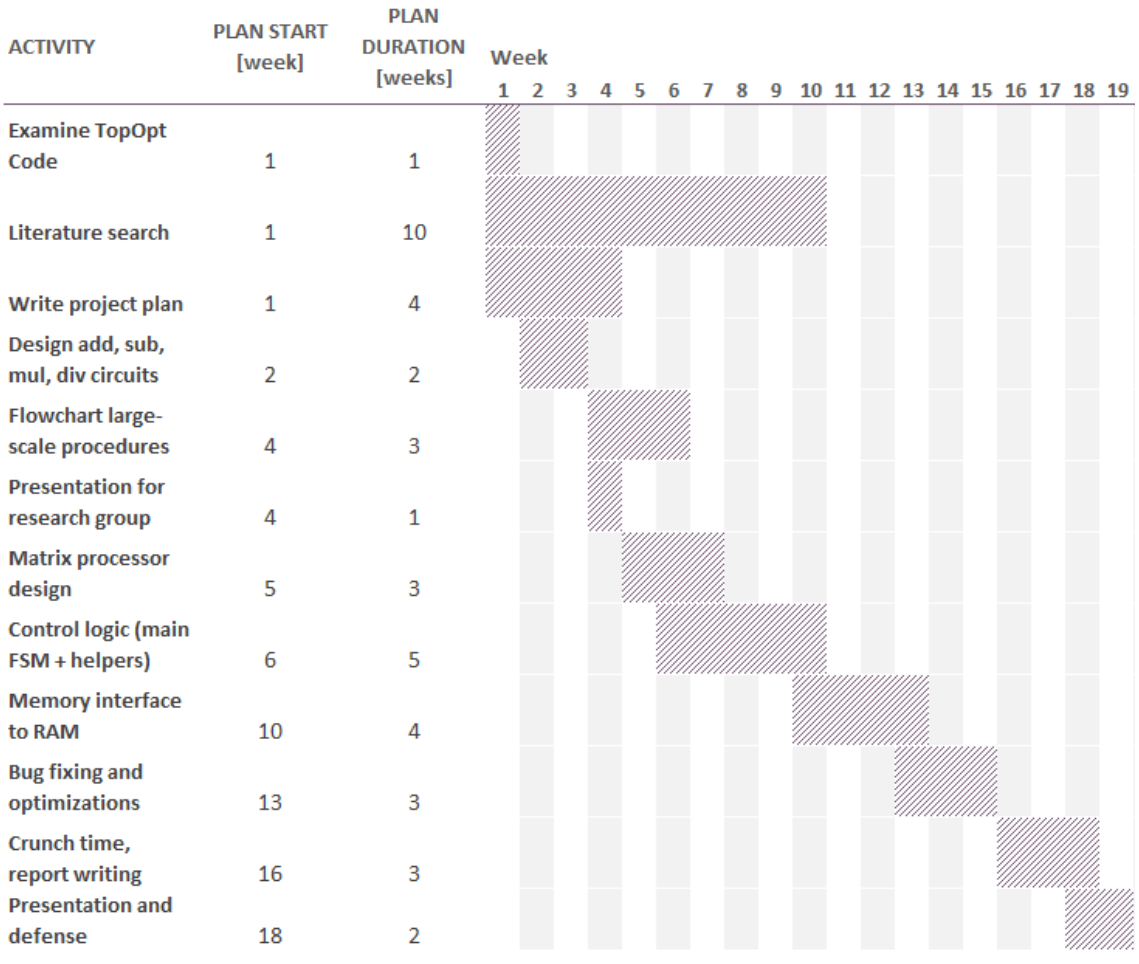

Figure A.1: Gantt chart showing the expected project schedule.