TÉCNICO LISBOA

RISC-V Processing System with streaming support

Eduardo Miguel Ferreira Cabral de Melo

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Nuno Filipe Valentim Roma
Prof. Pedro Filipe Zeferino Tomás

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Pedro Filipe Zeferino Tomás
Member of the Committee: Prof. João Carlos Viegas Martins Bispo

January 2020

Acknowledgments

Quero agradecer aos meus orientadores, Professor Pedro Tomás e Professor Nuno Roma, pelo apoio prestado ao longo do desenvolvimento da minha tese. Quero também agradecer ao Nuno Neves pelo apoio prestado ao desenvolvimento da tese, especialmente em dúvidas relacionadas com o Xilinx Vivado. Por fim quero agradecer à minha família e aos meus amigos, por me apoiarem ao longo do meu percurso de 5 anos e especialmente durante o desenvolvimento deste trabalho.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia under projects UID/CEC/50021/2019 and PTDC/EEI-HAC/30485/2017.

Resumo

Ao longo dos últimos dois anos, a popularidade do conjunto de instruções RISC-V pela comunidade open-source e companhias tem vindo a crescer. O crescimento da popularide do RISC-V subsequentemente incentiva o desenvolvimento de novas extensões opcionais adicionadas ao conjunto de instruções RISC-V que, por sua vez, proporcionam ferramentas para enfrentar diversas tarefas de elevada relevância para a industria, consequentemente promovendo o desenvolvimento de extensões que usam novas técnicas e conceitos ainda nao aplicados. O desenvolvimento deste trabalho foi motivado pela análise de uma nova extensao desenvolvida para o RISC-V, chamada Unlimited Vector Extension, que proporciona suporte ao nível da arquitetura de conjunto de instruções para streaming de dados em conjunto com o suporte para instruções SIMD escaláveis. Este trabalho proporciona uma implementação proof of concept da extensão UVE, partindo de um sistema softcore RISC-V base, adicionando um Stream Engine e um Acelarador Vectorial ao sistema base. O sistema foi implementado com sucesso na placa FPGA Xilinx Virtex UltraScale+ VCU1525 e testado usando um conjunto de benchmarks corretamente vectorizados para uso com a extensão UVE, resultando em tempos de execução até 23 vezes mais elevados e valores de EDP até 195 vezes mais baixos quando comparado com código RISC-V não vectorizado.

Palavras-chave: RISC-V, UVE, Data Streaming, SIMD, Acelarador Vectorial, FPGA

Abstract

Over the last two years, the adoption of the RISC-V ISA by the open-source community and companies has been increasing. This further encourages the development of new custom extensions added on top of the base ISA, some of them making use of novelty techniques and concepts. The development of this work was motivated by analysing a novel isntruction set extension that was recently developed for the RISC-V ISA, namely the Unlimited Vector Extension(UVE). It provides ISA level support for data streaming coupled with scalable SIMD instructions, and, in simulation, provides substantial performance gains regarding the state-of-the-art ARM Scalable Vector Extension (SVE). This work provides a proof of concept implementation of the UVE extension on a base RISC-V Softcore system, by adding a Stream Engine and Vector Accelerator hardware component to the design. The system was successfully implemented on a Xilinx Virtex UltraScale+ VCU1525 FPGA Board and tested by vectorizing a set of benchmarks, which resulted in execution times of up to 23 times higher and EDP values of up to 195 lower when compared with the base RISC-V code.

Keywords: RISC-V, UVE, Data Streaming, SIMD, Vector Accelerator, FPGA

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Topic Overview

The CPU market over the last few years has been dominated by either off-the-shelf third party solutions, such as INTEL and AMD x86 processors, in consumer grade high performance devices, mainly laptop and desktop computers, or by ARM's proprietary instruction set architecture and processor designs in the mobile device and microcontroller domains. Although using third-party proprietary solutions has its benefits, such as not needing a dedicated hardware team to design and implement in-house solutions, it also comes with disadvantages, mainly due to the fact that the customer of the third party solution has little to no control over the processor architecture or over the changes made to the instruction set architecture, including its flexibility. In fact, APPLE's recent decision of moving their Macintosh computers from Intel x86 to ARM validates that having more control over the whole development stack does prove advantageous.

In order to provide the hardware community with a royalty free, flexible open source instruction set architecture, the RISC-V ISA was developed at the University of California, Berkeley and the ISA specification was later published so it could be used both in academia and in commercial products alike. The RISC-V ISA was structured to be as modular as possible. Consequently, and due to being a RISC instruction set architecture, the base integer instruction set, RV32 or RV64, depending on if the system is targeting 32 bits or 64 bits, is minimal. This leads to the base ISA being suitable for microcontrollers and others solutions that have strict power and area constraints. In contrast, if one requires a solution that does not have strict power and area constraints, but on the other hand requires more advanced features such as floating point support, or atomic and memory fencing instruction support, then one can make use of the several extensions that RISC-V provides on top of the base integer instruction set. Of course, due to the open-source nature of the RISC-V ISA, third-party extensions that fulfill custom needs can be developed and integrated into a RISC-V design.

The motivation for this work started from the analysis of a novelty SIMD extension for RISC-V, the Unlimited Vector Extension (UVE). This extension provides ISA level support for data streaming, by describing a set of streaming instructions. Moreover, it provides a set of SIMD instructions that can be used to produce or consume data to/from the streams defined and a set of stream branch instructions for stream-based loop flow control.

Another motivation for this work was the analysis of the Configurable RISC-V Softcore Processing System. This system provides a RISC-V Softcore implementation with support for 32-bit integer instructions, including the multiply and divide extension instructions. Furthermore, it provides support for a level 1 data cache and a set of memory-mapped peripherals. However, the lack of floating point support and of an Instruction Cache limit the flexibility and performance of the system.

This thesis strives to corroborate the flexibility provided by the RISC-V and the viability of ISA level data streaming support by providing a proof of concept implementation of a novel custom RISC-V scalable SIMD extension with ISA level support for data streaming on top of an already existing RISC-V processing system, while also contributing with improvements to the existing system, by the introduction of an Instruction Cache and floating point support.

1.2 Objectives

The objectives set and completed during the development of this work can be summarized in the following points:

- Addition of floating point support to the Scalar Core
- Development and inclusion of an Instruction Cache
- Development of a proof-of-concept implementation of of the UVE extension on top of the base softcore system

1.3 Contributions

The addition of data streaming to a RISC-V system is the main focus of this work, consequently the main contribution of this work is the development of an hardware solution that allows support for data streaming at an ISA level. Throughout the realization of this work, others contributions were made, such as the addition of a dedicated Single Instruction Multiple Data accelerator, the development of an instruction cache, the communication mechanisms required so that the core and accelerator can both interface with the memory hierarchy of the system and peripherals, the introduction of support for different clock domains in the system and finally the analysis of the system results using a set of benchmarks.

1.4 Thesis Outline

This document is divided into 5 chapters, excluding the introduction and conclusions of this work. Chapter 2 provides the necessary background knowledge and concepts, related to computer architectures, needed to follow the work described in the following chapters. Succeeding chapter 2, a brief description of the baseline system used in this work and its components is provided in chapter 3. Consequently, this chapter provides some context to the reader about what part of the system is work prior to this thesis. Chapter 4 then explains in detail the hardware solutions developed in this work, and what is the function of each component when looking at the overall system. To more easily present relevant information, this chapter has been divided into three sections. The first section describes in detail the architecture of the developed SIMD accelerator. The second section focuses solely on the development of the Stream Engine and gives the reader an understanding of how data streaming is handled in the system. The third section then delves into the further changes that complement the addition of data streaming to the baseline system. Finally, chapters 5 and 6 provide an overview of the benchmarks used to evaluate the performance of the system, the software tools needed to compile such benchmarks, the system resource, frequency and power requirements and at last benchmark results and discussion.

Chapter 2

Background and Related Work

2.1 Scalable Single Instruction Multiple Data Extensions

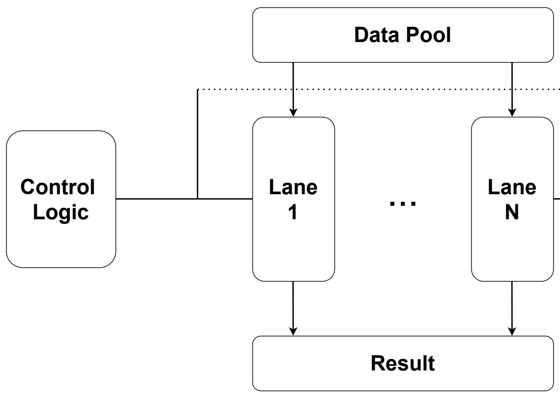


Figure 2.1: Example of Execution Lanes for use in SIMD Instructions.

The inclusion of Single Instruction Multiple Data (SIMD) extensions has become a standard in modern instruction set architectures. These extensions allow the architecture to explore data level parallelism, by processing several data elements, i.e a vector of elements, using a single program instruction, therefore increasing data throughput. The parallel processing of elements is achieved at the architecture level by including several lanes of execution in the core, where each lane processes a given data element and all lanes are arranged in parallel, like the example in figure 2.1. The number of lanes is proportional to the register width of the SIMD extension, which is most commonly 256 or 512 bits. In scalable SIMD extensions, the vector length is scalable at the implementation level. With this mechanism, the flexibility of the extension greatly increases, since it provides the user with the capability to use smaller or larger vector lengths, depending on the application, therefore producing reusable code. This section will present two scalable SIMD extensions of note, the ARM SVE extension and the RISC-V V extension.

2.1.1 ARM Scalable Vector Extension

ARM's solution to provide a scalable SIMD extension to its instruction set is the SVE extension[17]. The SVE extension architectural state is composed of 32 vector registers, named *Z0* to *Z31*, extended on top of the already existing Neon registers to provide scalable containers for the supported width data elements, the element widths being 8, 16, 32 and 64 bits. The SVE extension also includes the addition of 16 predicate registers, *P0* to *P15* that are used to control the execution of the different lanes that operate on the vector elements. From these 16 predicate registers, only the first 8 are allowed in use of instruction predication, and all 16 are allowed for use in predicate only instructions. Since SVE is a scalable SIMD extension, the vector length can be chosen by the user, however the value of the vector length must be a multiple of 128 bits, up to a maximum value of 2048 bits. Consequently, the predicate length is also variable, and is 1/8th of the size of the vector length in use, serving as a byte mask that conditions the execution of given lanes, depending on the values of the bits of the predicate.

To better understand the scalable nature of SVE, and how it is used in practice, an example kernel of the SAXPY (Single Precision A times X plus Y) algorithm implemented using the SVE extension is presented in Listing 2.1. The loop preamble sets the initial vector length using the instruction *whilelt p0.s, x4, x3*, by populating the *p0* predicate register through the values of registers *x4* and *x3*. In detail, the value *x4* represents the iteration variable, commonly refered as *i*, and register *x3* represents the comparison value and the predicate register is populated with ones until the value in *x4* is greater than or equal to the comparison value *x3*. This process is repeated each iteration of the loop and the iterator variable is updated using the *incw x4* instruction. The *incx* family of instructions updates the iterator variable according to the data type being processed, which in the case of the SAXPY algorithm is the word type, corresponding to 32 bit elements. At last, the instruction *b.first .loop* controls the flow of the loop code, by performing a branch to the *.loop* tag if the iterator variable does not meet the finish condition. The condition codes needed for the *b.first* branch instruction to work correctly, i.e the predicate register, are set by the *whilelt* instruction each time it is called, and since the predication is continuous, i.e there are no zeroes interleaved with 1's, the branch instruction needs only to check the first element and check if it is active or inactive.

Listing 2.1: SAXPY kernel implementation using ARM SVE extension.

```
saxpy:
   mov     x4,#0
   whilelt p0.s,x4,x3
   ld1rs   z0.s,p0.z,[x2]
.loop:
   ld1s    z1.s,p0/z,[x8,x4,lsl #2]
   ld1s    z2.s,p0/z,[x9,x4,lsl #2]
   fmla    z2.s,p0/m,z1.s,z0.s
   st1s    z2.s,p0,[x1,x4,lsl #2]
   incw    x4
   whilelt p0.s,x4,x3
```

```
            b.first .loop
```

2.1.2 RISC-V Vector Extension

The SIMD solution offered by the RISC-V Instruction set, RISC-V V extension[15], was introduced to provide the data level parallelism support that is lacking from the base RISC-V ISA. Contrary to the ARM SVE extension, the architectural state added by the RISC-V extension is separate from the base ISA architecture. Consequently, 32 new vector registers, named *v0* to *v31*, are added by the extension, that serve as containers for elements of the types inherited from the base ISA and its optional single precision and double precision floating point extensions, namely the integer and fixed point data types. In constrast with the ARM SVE extension, the RISC-V V extension demands no limitation on the maximum vector length, except it must be a multiple of the highest width element supported, providing a more flexible and easier code development platform. The vector registers can optionally be used as predicate registers to mask the execution of the SIMD instructions provided by the extension. The control of the vector length in use by the extension is managed using the *vsetvli* instruction. This instruction takes as arguments the vector size,in number of elements, and element size requested, calculates the actual vector size in bits needed and compares it with the implemented vector length, using the minimum of both values as the configured size for all the vectors and writes this size to the destination register. This instruction also has a grouping factor option. This grouping factor can be used to group, for example, vectors *v0* to *v3*, such that a instruction that uses *v0* as an operand will use the values of all the vectors grouped to *v0* in the computation. The grouping factor argument can also be used to divide a particular vector register into multiple vectors, therefore processing more that one vector using a single SIMD instruction.

The code presented in Listing 2.2 represents an implementation of the same SAXPY kernel referenced before, however this time using the RISC-V V extension. In this case, the entire kernel is inserted in loop code, where the vector length is controlled by the *vsetvli* and the remaining loop control code uses the base ISA instructions such as the *bnez* branch instruction. The vector configuration uses an element width of 32 is used, set by the use of the argument *e32* and a grouping factor of 8, meaning that registers *v0* to *v7*, *v8* to *15* and so on are grouped together and treated as a single vector register. Contrary to the SVE example, the iterator and address calculation is done almost entirely using base RISC-V instructions, as can be seen by the use of the *slli*, *sub* and *add* instructions used in the loop code, using only the configure size, i.e the number of elements processed, from the *vsetvli* instruction.

Listing 2.2: SAXPY kernel implementation using RISC-V V extension.

```
saxpy:
   vsetvli    a4,a0,e32,m8
   vlw.v      v0,(a1)
   sub        a0,a0,a4
   slli       a4,a4,2
   add        a1,a1,a4
   vlw,v      v8,(a2)
   vfmacc.vf  v8,fa0,v0
   vsw.v      v8,(a2)
   add        a2,a2,a4
   bnez       a0, saxpy
```

2.2 Data Streaming Architectures

Over the last years, with the evolution of the semiconductor process technology, processor clock speeds have been significantly increased. In fact, this evolution of clock speed has completely overshadowed the increases in clock speed of off-chip memory, which in turn has had its improvements focused on providing larger memory capacity. Therefore, modern processors now have to deal with pipeline stall times of hundreds of clock cycles if a cache miss occurs, resulting in a large amount of wasted resources. Most modern systems rely on prefetching techniques such as cache prefetching, to attempt to mask the high latency costs of cache misses by attempting to predict and load data that will later be used by the processor, using a variety of algorithms such as sequential prefetching, strided prefetching or Access Map Patter Matching (AMPM) [2][7]. However, this type of hardware prefetching suffers from often not providing large gains to execution time. This is mainly due to the fact that cache prefetching is more effective on simple memory patterns, and struggles to predict more complex memory access patterns, where the most gains to be had from prefetching are located. To explore such gains, the use of independent components, entirely dedicated to data streaming, that can be programmed to prefetch complex memory access patterns have been proposed and tested, with favourable results. These components can be the base from where the architecture of the system is built around, or can be integrated in an already existing architecture. This chapter will present some examples of Data Streaming Architectures, focusing on multiple and single processing element architectures.

2.2.1 Data Streaming to Multiple Processing Elements

Systems that implement a multi-core architecture strive to explore the benefits of concurrent execution of several instructions, by offloading different sections of the program to different cores of the system. However, with the concurrent execution of several cores in the system comes the inevitable concurrent memory accesses of the different cores that exert a large amount of pressure on the memory hierarchy of shared-memory many-core systems.

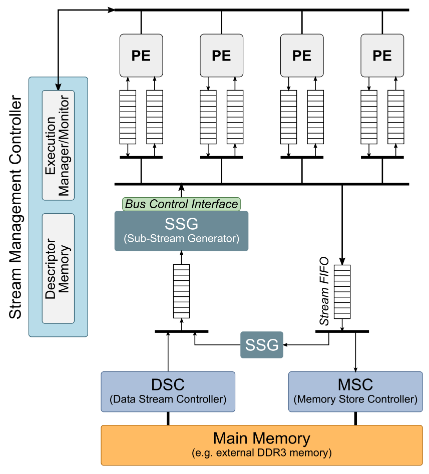


Figure 2.2: Stream Management Engine Overview, proposed by Neves et al. [9]

Consequently, a stream management engine solution [9] was proposed to minimize the number of memory accesses, maximize data throughput to the processing elements and reduce energy consumption on shared-memory multi-core systems, represented in figure 2.2. The stream management engine is comprised of several hardware structures, namely the Data Stream Controller(DSC), the Memory Store Controller(MSC), the Sub-Stream Generator(SSG), the Stream Management Controller(SMC) and a set of stream FIFO memories. The DSC, MSC and SSG all implement a Description Tree Controller that is used to generate addresses based on a stream description using stream descriptors. The Data Stream Controller then uses the calculated addresses by the DTC to issue memory read operations and generates the resulting data streams. The data received from memory is tagged with ordered serial numbers and a unique per stream identifier, so that data ordering and destination processing elements(PE's) can be recognized, and subsequently sent to an output stream FIFO. On the other hand, the Memory Store Controller uses the values calculated by the DTC to issue memory write operations, that consume data produced by the PE's. Additionally, monitoring logic is used to identify the incoming streams that should be written to main memory and streams that should be redirected to other PE's using the Sub-Stream Generator. The last of the components based on the Descriptor Tree Controller, the Sub-Stream Generator, generates sub-streams which ,instead of generating memory addresses, generate serial numbers coupled with stream identifiers. The data incoming to the SSG is the redirected with the new serial numbers and stream identifier to the destination processing elements. The inclusion of the SSG component allows the Stream Management Engine to produce sub-streams from a main stream, and redirect the sub-streams to the appropriate PE's and also to generate an output stream from multiple sub-streams, effectively implementing scatter and gather operations using data from the different PE's. Additionally, the SSG can be used to create streams from output streams from the PE's and redirect the newly created stream back to the processing elements, providing an easy mechanism for data reuse. Lastly, the Stream Management Controller is responsible for maintaining a central descriptor memory, used to describe streams, assigning descriptors to the DSC, MSC and the SSG's and controlling the execution of the several processing elements of the system based on the data-stream flow.

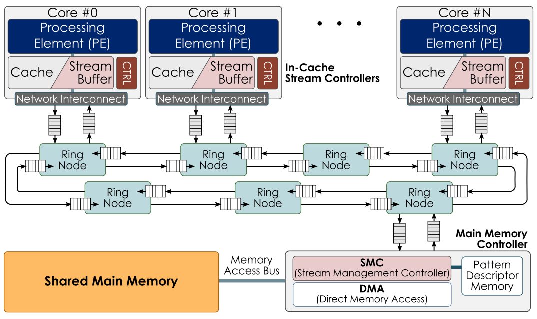
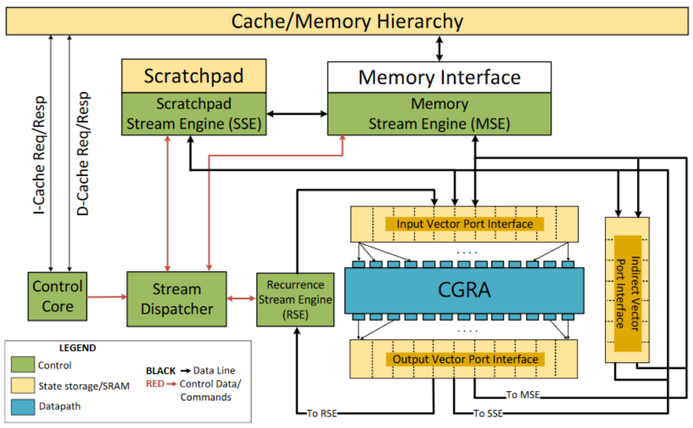


Figure 2.3: In-Cache Streaming Architecture Overview, proposed by Neves et al. [10]

An alternative streaming solution [10], proposes the use of an in-cache stream paradigm to provide a streaming architecture that can merge the benefits of traditional address-based and stream-based communication paradigms and consequently be able to tackle all types of applications, from simple to complex memory patterns. The proposed architecture is represented in figure 2.3. Contrary to the solution described above, each processing element has access to a data cache, following more traditional multi-core architectures, however the data cache implementation deviates from the standard. In detail, the processing element can view its set of n-ways present on the data cache as typical cache ways or as stream buffers, due to the fact that the cache includes two types of controllers, a typical address-based cache controller and a stream controller. Both controllers have access to the same resources, i.e the cache ways available by the implementation, however a specific cache way is either controller by the cache controller or by the stream controller, not both simultaneously. The cache does provide a dynamic way to change the ownership of the cache ways through a register, and this ownership can be controller by the processing element associated with the given cache. When controller by the cache controller, a given cache way works using the normal tag comparison and line selection logic, however when controlled by the stream manager controller, the cache way is interpreted as a circular buffer to where data can be produced or consumed. Using this mechanism, the cache is able to change its behaviour depending on the type of communication needed, typical address-based or stream-based, and can in fact employ both communication mechanisms at the same time by shared the cache ways between the cache controller and stream controller. The data caches associated with the given processing elements are interconnected and connected to the Main Memory Controller through interconnected ring nodes that convey point-to-point and broadcast capabilities, providing the system with support for scatter and gather operations using the different PE's and shared main memory. At last, the Main Memory Controller provides the typical direct memory access support required for usual address-based communication and additionally contains a Stream Management Controller coupled with a Pattern Descriptor memory bank, used to control data-stream flow through the system, analogously to the Stream Management Controller described in the previous solution.

2.2.2 Data Streaming to Dedicated Accelerator

The rise in popularity of machine-learning, computer vision, and other fields of science that require tasks with high computational intensity have started to shift the industry focus from general purpose high performance computing solutions, such as SIMD support or GPGPUs, to application and domain-specific accelerators that offer higher performance gains at the cost of generality. These application and domain-specific accelerators are usually coupled with the main processing core of the system, which provides the necessary control signals and data to the accelerator. Being that the accelerators are designed for a specific domain or application, it follows that the cost of processing should be lower that the cost of memory accesses, for a given application, making most domain-specific accelerator performance gains memory-bound.



Cache/Memory Hierarchy
I-Cache Req/Resp
D-Cache Req/Resp
Scratchpad
Scratchpad Stream Engine (SSE)
Memory Interface
Memory Stream Engine (MSE)
Input Vector Port Interface
CGRA
Output Vector Port Interface
Indirect Vector Port Interface
Control Core
Stream Dispatcher
Recurrence Stream Engine (RSE)
To RSE
To SSE
To MSE
LEGEND
Control
State storage/SRAM
Datapath
BLACK → Data Line
RED → Control Data/ Commands

Figure 2.4: Softbrain Architecture Overview, proposed by Nowatzki et al. [11]

In order to lower the impact of memory accesses in application or domain-specific accelerators, a hardware streaming solution was proposed by Nowatzki et al. and a microarchitecture, illustrated in figure 2.4, was developed to evaluate its performance. The Softbrain microarchitecture is composed of five main elements, the Control Core, Stream Dispatcher, Stream Engines, Vector Ports and the Coarse Grained Reconfigurable Architecture (CGRA).

The Control Core is a low-power single-issue inorder core whose purpose is to generate stream-dataflow commands to the stream dispatcher, i.e it acts as a programmer of the stream dispatcher. The low power and low area requirements are due to the simple function the core has to serve, since it is not a general purpose computing core.

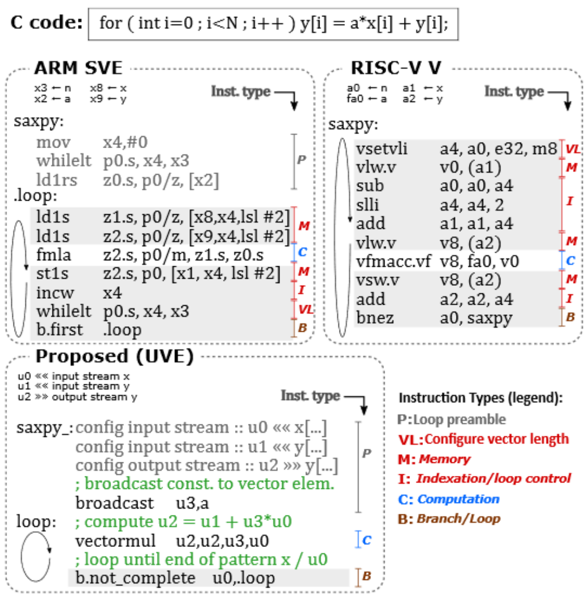
The Stream Dispatcher Unit then processes the commands from the Control Core and controls the concurrent execution of the stream engines of the architecture, tracking resource dependencies and issuing commands when needed. There are a total of three stream engines that need to be controlled by the Stream Dispatcher Unit, the Memory Stream Engine, the Scratchpad Stream Engine and the Recurrence Stream Engine. The Memory Stream Engine processes streams and in turn issues the respective read and write requests to main memory, while also producing and consuming its values to/from the Vector Port Interfaces. The Scratchpad Stream Engine processes streams that access the Scratchpad memory instead of main memory, effectively allowing for reuse of loaded data. This scratchpad memory is populated by the use of store streams that use data from the Output Vector Port Interface. At last, the Recurrence Stream Engine is used for immediate reuse of data without memory storage, i.e the data is forwarded from the Output Vector Port directly to the Input Vector Port after being processed by the Recurrence Stream Engine.

The several stream engines present in the microarchitecture share data between themselves and with the CGRA through the Vector Port Interfaces. The Input Vector Port Interface stores incoming data from streams until the CGRA is ready to consume the data for the computation, while the Output Vector Port Interface stores the result of the computation performed by the CGRA until it is ready to be consumed by an outgoing stream. An additional interface, named Indirect Vector Port Interface, is used to store the streaming addresses of indirect loads or stores.

Lastly, the Coarse Grained Reconfigurable Architecture is a domain-specific accelerator that eanbles pipelined computation of dataflow graphs using a mesh of processing elements, where each processing element contains a set of pipelined functional units.

When comparing the core + acceleration solution with the many-core solutions described previously, it is clear that some of the techniques are reused, regardless of the target architecture. These techniques are the use of a dedicated Stream Controller, the use of buffering memory modules, such as stream FIFOS, to allow prefetching of data, communication buses that allow for scatter and gather operations and hardware support for data reuse to reduce memory accesses.

2.3 Unlimited Vector Extension


C code:
```
for ( int i=0 ; i<N ; i++ ) y[i] = a*x[i] + y[i];
```

ARM SVE
```
x3 ← n   x8 ← x
x2 ← a   x9 ← y
saxpy:
    mov      x4,#0                           P
    whilelt  p0.s, x4, x3
    ld1rs    z0.s, p0/z, [x2]
.loop:
    ld1s     z1.s, p0/z, [x8,x4,lsl #2]      M
    ld1s     z2.s, p0/z, [x9,x4,lsl #2]
    fmla     z2.s, p0/m, z1.s, z0.s          C
    st1s     z2.s, p0, [x1, x4, lsl #2]      M
    incw     x4                              I
    whilelt  p0.s, x4, x3                    VL
    b.first  .loop                           B
```

RISC-V V
```
a0 ← n   a1 ← x
fa0 ← a  a2 ← y
saxpy:
    vsetvli    a4, a0, e32, m8   VL
    vlw.v      v0, (a1)          M
    sub        a0, a0, a4        I
    slli       a4, a4, 2
    add        a1, a1, a4
    vlw.v      v8, (a2)          M
    vfmacc.vf  v8, fa0, v0       C
    vsw.v      v8, (a2)          M
    add        a2, a2, a4        I
    bnez       a0, saxpy         B
```

Proposed (UVE)
```
u0 «« input stream x
u1 «« input stream y
u2 »» output stream y
saxpy_:config input stream :: u0 «« x[...]                 P
       config input stream :: u1 «« y[...]
       config output stream :: u2 »» y[...]
       ; broadcast const. to vector elem.
       broadcast    u3,a
loop:  ; compute u2 = u1 + u3*u0
       vectormul    u2,u2,u3,u0                            C
       ; loop until end of pattern x / u0
       b.not_complete   u0,.loop                           B
```

Inst. type
Instruction Types (legend):
P:Loop preamble
VL:Configure vector length
M: Memory
I: Indexation/loop control
C: Computation
B: Branch/Loop

Figure 2.5: Saxpy example kernel implementations using ARM SVE, RISC-V V and the UVE extension[4]. Instructions with shaded background represent loop overhead.

Section 2.1 of this chapter explained the advantages of a SIMD adopting a scalable aproach, i.e the extension makes no assumptions nor limits the maximum vector length, facilitating the vectorization and providing more flexibility to the produced code. However, both scalable solutions analysed so far in this thesis express a fundamental drawback that limits the theoretical performance gains achievable, the drawback being the large instruction overhead required to set up the processing of the required data elements. The instruction overhead manifests itself in the form of indexation, loop control and memory access instructions, which in fact can, in most cases, represent the majority of the loop code (shaded instructions illustrated in figure 2.5), therefore lowering processing throughput by wasting processor resources.

The Unlimited Vector Extension [4] is a custom scalable single instruction multiple data extension developed for the RISC-V instruction set architecture, created with the main purpose of reducing the number of overhead instructions in loop code in order to extract more performance gains out of data level parallelism. To this end, it uses a set of hierarchically ordered set of descriptors to describe memory access patterns. These descriptors are defined by user code, using the stream instructions of the UVE extension, at the loop preamble, as can be seen in figure 2.5. The decoupling of the memory instructions from the loop code is achieved by allowing a Stream Engine to process the descriptors defined in the prefetch instructions and stream the relevant data to/from the core. This decoupling can be done without losing flexibility in the type of memory accesses possible, since complex memory patterns can still be supported by conjugating several types of descriptors in a hierarchical fashion.

2.3.1 Memory Access Representation

The UVE extension uses a memory access representation based on stream descriptors to describe the sequence of addresses that comprise accesses to array-based variables. This model follows the structure of nested for loops used in regular code, in order to make the transition from non-vectorized code with no prefetching to UVE code easier of both understand and perform, either by hand or by the compiler.

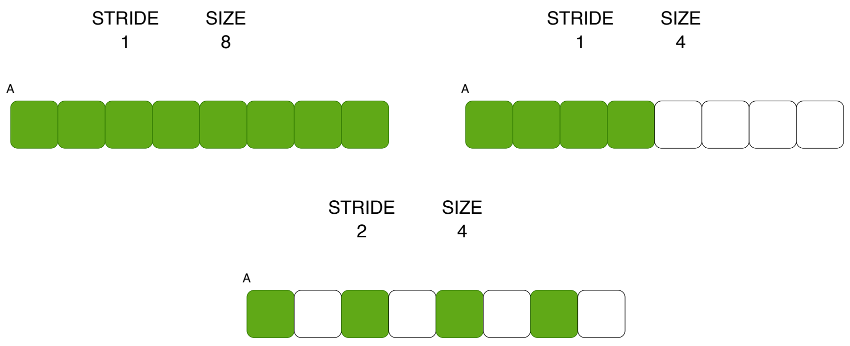


Figure 2.6: One dimensional access examples.

Using regular C code, one could represent an one dimensional access to memory using a single for loop, as illustrated in Listing 2.3. This access makes use of three fundamental variables, the base address of the variable *A*, the stride of the memory access and *N*, the size of the memory access.Consequently, the UVE representation for a one dimensional access is a descriptor, called base stream descriptor, that is represented by a three value tuple. The tuple that represents the base stream descriptor can be used to calculate memory addresses by applying it to a simple affine function, $address_i = base\_address + i * stride, i = 0, .., N - 1$. By using this internal representation, several memory access patterns are possible, by varying the base address, size and stride variables. Some example accesses are represented in figure 2.6.

Listing 2.3: Single For Loop Example.

```
for(i=0; i < N; i += stride)
    A[i] = 2*A[i]; //example computation
```

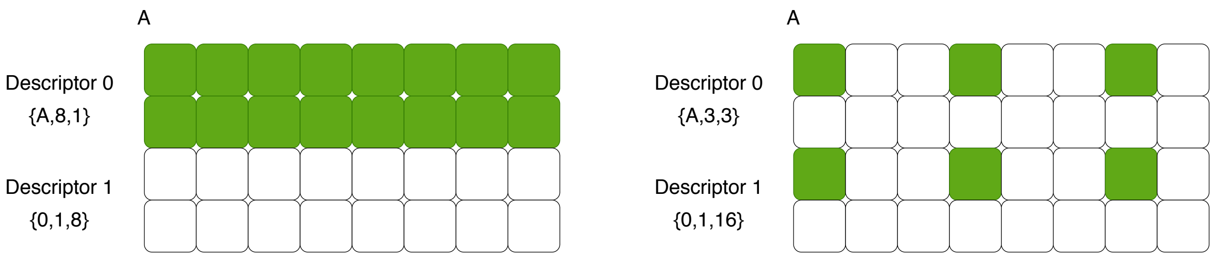


Figure 2.7: Two dimensional access examples.

The use of a single base stream descriptor can be used to represent a multitude of one dimensional accesses, depending on the values of the tuple variables, however it can not be used to represent multi-dimensional memory access patterns. Therefore, to provide support for a n-dimensional memory access pattern, the UVE extension employs a hierarchical organization of several stream descriptors. The stream descriptors are combined using a linear cascade scheme, analogous to the nested for loop method, where a descriptor corresponding to dimension *i* is used to calculate an offset, using a affine function, that is then added to the offset of the descriptor associated with dimension *i-1*. This method is effectively linearly combining the affine functions off all the base descriptors used to describe the memory access. Consequently, the level of flexibility provided is far greater when compared with the use of a single base descriptor, since instead of 3 variables that can be changed to affect the memory pattern, the use of cascaded descriptors provides n*3 variables that can be tuned to create the desired memory access pattern. Some examples of memory patterns using two cascaded base descriptors are illustrated in figure 2.7.

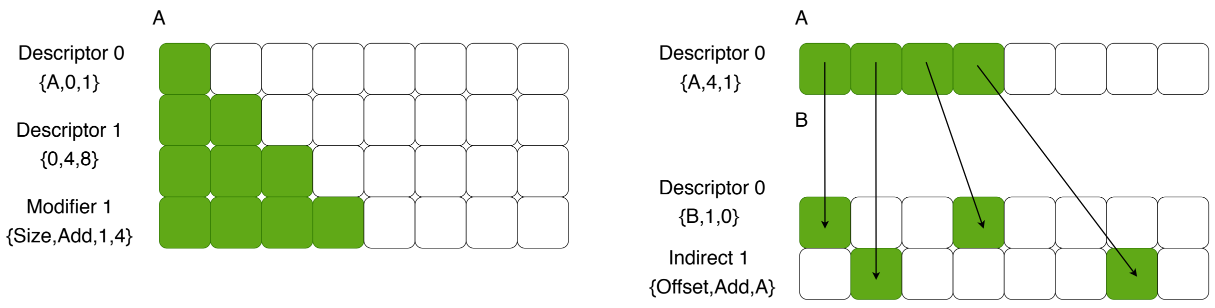


Figure 2.8: Lower triangular and Indirect Access Examples.

Whilst the hierarchical organization described in the previous paragraph does indeed provide greater flexibility that the use of a single descriptor, there are still some types of memory patterns that the descriptor tree cannot represent, as is. Lets take as an example the first for loop represented in Listing 2.4. This loop represents a lower triangular access pattern to matrix A by the use of an inter loop dependency. In this case the size value of the inner loop is modified by each iteration of the outer loop. To deal with these inter loop dependencies, the UVE extension provides a new mechanism called static descriptor modifiers. A static descriptor modifier is represented by a tuple comprised of four elements. The first element is the target, which is as an identifier of the descriptor parameter to modify. The second and third elements are the behavior and displacement of the modifier. The behavior defines the operation to be executed, i.e addition or subtraction, and the displacement is the value used by the operation to update the parameter value. The final value of the tuple is the size of the static descriptor modifier and it specifies the total number of iterations for which the modification should occur.

The second for loop of Listing 2.4 provides yet another example of a memory access pattern that cannot be described only with the use of base descriptors. In this case, the array *B* is not directly indexed by the index variable of the for loop but is instead indexed by the value present in slot *i* of array *A*, i.e array *B* is indirectly indexed using array *A* and index variable *i*. In order to represent the behavior of indirect indexing in the stream descriptor architecture, the UVE extension describes another type of tuple called indirect descriptor modifier. It is composed of 3 values, the first and second values are the target and behaviour of the modifier, just like in the static descriptor modifier, however the third value of the tuple is a pointer to the origin data stream. The indirect descriptor modifier does not need a size value in its tuple since we are associating the origin stream with the target stream and consequently the sizes of the target stream depends on the size of the origin stream. The behavior value of the indirect stream modifier encodes the following operations:

- *add*: the displacement value resulting from the origin stream is added to the target stream parameter each iteration.
- *sub*: the displacement value is subtracted to the target parameter.
- *value*: sets the target value to the value resulting from the origin stream.

A visual representation of the examples present in Listing 2.4 that illustrate the use of the static and indirect descriptor modifiers can be observed in figure 2.8. The memory pattern on the left makes use of the static descriptor modifier to modify the size of the inner base descriptor by incrementing it every iteration, resulting in a lower triangular pattern. The memory patter on the right makes use of the values from vector A, resulting from the origin stream, as the index to access values in vector B, using the indirect descriptor modifier. It should be noted that, to modify a parameter associated with dimension *i*, the corresponding modifier should be associated with dimension *i+i*.

Listing 2.4: Inter loop dependency and indirect indexing examples.

```
/* Lower Triangular Access, Inter Loop Dependency*/
int K = 0;
for(i=0; i<N; i++)
```

```
    {
        K++;
        for(j=0; j<K; j++)
        {
            A[i*N + j];
        }
    }
    /*Indirect Indexing of array B*/
    for(i=0; i<N; i++)
        B[A[i]];
```

2.3.2 Architectural State

Since the UVE extension is at its core a SIMD extension, it defines two new register files, one composed of vector registers and another composed of predicate registers. However, contrary to other SIMD extensions, UVE also defines a stream interface to the vector instructions due to the fact that the ISA extension supports data streaming.

The vector register file comprises 32 vector registers, named *u0* to *u31*, analogous to the scalar register file defined by the RISC-V base ISA. The vector registers are not limited by the extension to any particular maximum size, due to the extension adopting a scalable approach to vector. The only restriction applied to the maximum size of the vector is that it must be a multiple of the largest element width allowed, which in this case is 64 bits. However a minimum size allowed for the vector registers is defined. This is the case because the vectors are comprised of individual elements, and the elements widths supported by the extension are byte(8 bits), half-word(16 bits), word(32 bits) and double word(64 bits). Consequently the minimum size allowed for a vector register is the maximum width of the supported elements, which in the case of the UVE extension is 64 bits. If, for example, one decides to only support up to 32-bit elements then the minimum size requirement can be relaxed to 32 bits since 64-bit wide elements are not supported.

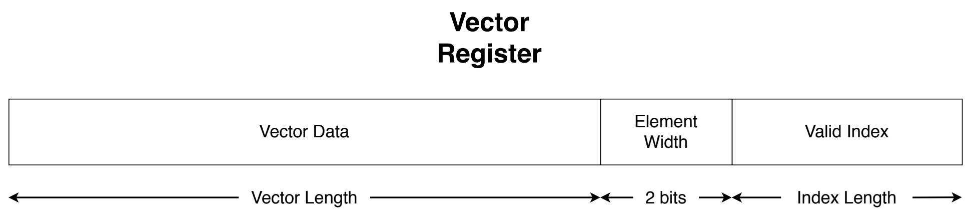


Figure 2.9: Vector Register Example Layout.

Due to the fact that the vectors are scalable in size, some additional meta-information needs to be kept for each register to support the scalable behavior. Consequently, each vector register holds not only the vector data but also the width of the elements in the vector and a valid index value. The element width informs the processor about the widths of the elements in the vector so it can process them correctly in the execute stage. The valid index informs the processor about how many elements in the vector are valid, assuming all valid elements are contiguous in the vector. By using these two values, each vector register has the flexibility of processing elements of different size and also store vectors of different size with the use of the valid index value. A simple illustration of the contents of a vector register is represented in figure 2.9. The element width data is encoded in 2 bits since only 4 possible element widths are available while the index width depends on the vector length of the implementation.

The predicate register file, contrary to the RISC-V scalar register file and the vector register file, features only 16 predicate registers, named *p0* to *p15*, however only the first 8 registers, *p0* to *p7*, are used to predicate arithmetic and regular memory instructions. The remaining registers, *p8* to *p15*, are used in predicate instructions that configure values for the first 8 registers or for context saving if needed. Additionally, predicate register *p0* is hardwired to 1 removing the need for pre-configuration of a predicate register if normal execution is pretended (no predication).

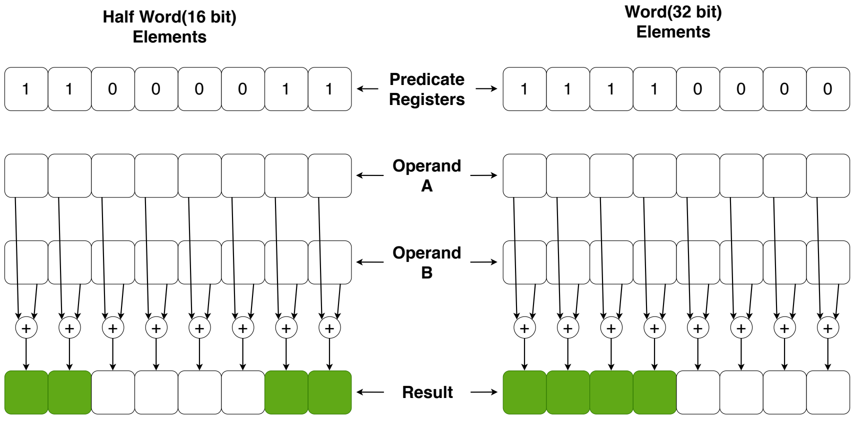


Figure 2.10: Predication Examples.

The width of a single predicate register is dependent of the size of the vector length of the implementation. In fact, the data of the predicate register acts as as a byte mask that selects which lanes should execute and which lanes should not (no operation), so the precise width of a predicate register is $Vector\_Length/8$. A couple examples of predication using different element widths are illustrated in figure 2.10, considering a vector length of 64 bits. The left example shows a predication on half word elements and consequently the bits of the predicate register that are set are always in pairs, since each bit represents a byte of the vector. With this predicate register, only the first and last half words of the vector are changed to the result of the operation (highlighted in green) and the rest of the half words remain unchanged. In the right example, the element width is now 32 bits, and the predicate register is allowing the operation on the first word, but not on the second word of the vector.

Due to the fact that the UVE extension provides ISA level support for data streaming, it defines a streaming interface that must be followed for the correct functioning of the streaming instructions provided. The streaming interface makes use of the already existing vector register file, by associating each stream with a specific vector register. Therefore, when a compute instruction reads or writes to a specific vector register, if that register is associated with a data stream, then the instruction consumes or produces, depending if the stream is a load or store stream, the vector data from or to the stream. This instruction behavior is destructive, meaning that if an instruction reads the contents of, for example, vector register *u1* and this register is associated with a load stream, then the data read is permanently consumed and cannot be accessed again using the stream interface. If the consumed data is to be accessed again, then the processor must save it in an intermediate register or memory address of choice. This behaviour eliminates the need of dedicated step instructions for streams since consuming or producing values from/to a configured stream automatically iterates the associated stream. Associating each active stream with a vector register provides advantages due to the fact that no further decoding bits are necessary in the instructions to read/write to streams and not exception mechanisms are needed when reading or writing to a non-configured stream due to the fact that if the stream is not configured, then the instructions simply reads/writes from/to the normal vector register is question.

2.3.3 Streaming Instructions

Due to the fact that the UVE extension provides support for ISA level data streaming, it provides a set of instructions to control the configuration and flow of streams. These instructions can be divided into several groups, mainly configuration instructions, stream control instructions and loop control instructions and allow the user code full control over the prefetching mechanisms, memory patterns and loop flow.

Configuration Instructions

```
ss.ld.{b | h | w | d}

ss.st.{b | h | w | d}
```

Figure 2.11: Simple 1D stream configuration instructions.

The configuration instructions give the user access to configuration of a variety of memory patterns by making use of the descriptors explained in the previous section. The simplest memory pattern available is a one dimensional memory pattern but, despite being simple, it is one of the most common memory patterns and consequently the UVE extension provides a pair of instructions to define one dimensional streams, illustrated in figure 2.11. The *ss.ld* instruction creates a load stream following a one dimensional pattern defined by the tuple values supplied by the user in the operand registers while the *ss.st* instruction creates a store stream. Both instructions require a prefix that provides information about the element width of the stream to be configured, *b* for byte elements, *h* for half word elements, *w* for word elements and *d* for double word elements.

```
ss.{ld | st}.sta.{b | h | w | d}

ss.app[.mod | .ind]

ss.end[.mod | .ind]
```

Figure 2.12: Multiple dimension stream configuration instructions.

To make use of the full potential of the hierarchical descriptor model representation of memory patterns explained in the previous section, mainly multi-dimensional memory patterns and modifiers, UVE provides access to three types of stream configuration instructions, represented in figure 2.12. The instruction *ss.sta* creates a stream, with direction chosen by the prefix *ld* or *st*, and appends the descriptor associated to the first dimension, much like the *ss.ld* or *ss.st* instructions explained before, however in this case the stream configuration is not terminated and more descriptors can be added. To add more descriptors to the stream in configuration, the user can make use of both the *ss.app* and *ss.end* instructions. This *ss.app* instruction appends a descriptor to a stream, with the option of appending an indirect or static modifier instead if the prefix *.ind* or *.mod* are present, respectively. The *ss.end* instruction behaves just like the append instructions, with the exception that the *ss.end* instruction signals the termination of the configuration phase of the stream in question and consequently no further descriptors or modifiers can be added.

Stream control Instructions

ss.resume
ss.suspend
ss.stop
ss.getvl
ss.setvl
so.cfg.mem*x*

Figure 2.13: Stream control instructions.

When a UVE stream has been successfully configured, it will start prefetching data to the processor immediately, however the need to stop or suspend a stream with a user command might be necessary for the correct program flow, consequently the UVE extension provides a set of stream control instructions that give the user control over the stream state and data path. The instructions *ss.resume*, *ss.suspend* and *ss.stop* control the stream state. If the user requires a stream to be suspended for a period of time, then *ss.suspend* can be used and the stream can then later be resumed with the *ss.resume* instruction. If, however, the user requires an early termination of a stream, i.e the termination of a stream that has yet to complete the full memory pattern described, then the *ss.stop* instruction should be used. The UVE extension also allows the query and manipulation of the vector length used by each stream with the use of the *ss.getvl* and *ss.setvl* instructions, respectively. Lastly, the *so.cfg.memx* instruction configures data path to be used by the stream. In particular, the prefix *x* point to the usage of the level *x* memory level of the system in question. This instruction allows the user to decide if the stream in question benefits from the use of the cache levels of the system, to make use of the latency reduction of accesses, or direct access to main memory, in order to prevent cache pollution.

Loop Control Instructions

Data streaming mechanisms provide the most benefit when used in user code that iterates over memory addresses using loop code. Consequently the UVE extension provides instructions to control the flow of such loops using stream dimension information. Two types of loop control instructions are so.b.[n]c

so.b.[n]dc

Figure 2.14: Stream loop control instructions.

provided, in the form of conditional branches, and each of the types includes a negation variant using the optional *n* prefix. The *so.b.c* instruction performs a branch operation if the stream in question ended, while the *so.b.dc* performs a branch operation if the dimension of the stream indicated by a register operand ended. By using this set of loop control instructions, a fine control over stream data and processing based on stream dimension can be achieved, analogously to nested for loop behavior.

2.4 Summary

This chapter started out by describing some of the state of the art scalable SIMD solutions, namely the ARM SVE extension and the RISC-V V extension, and providing vectorized code examples for each solution. Then, in the second section, an analysis of proposed streaming architectures as done, focusing on architectures streaming to multi-core systems and core-accelerator systems. Finally, the third chapter explained in detail the novelty UVE RISC-V extension, which combines scalable SIMD instructions with data streaming instructions, to provide ISA level data streaming and parallel processing support.

Chapter 3

Base RISC-V Softcore Processor

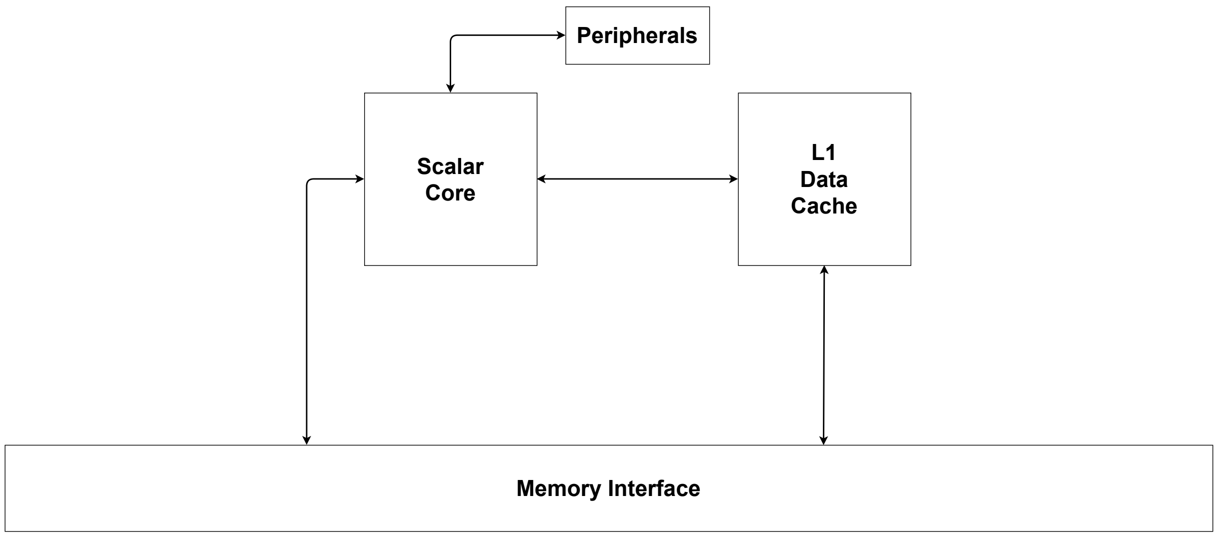


Figure 3.1: Base RISC-V Processing System.

The Configurable RISC-V Softcore Processor [16] was developed at Instituto Superior Técnico, as part of João Rodrigues's Master Thesis and had the objective of providing multi-cycle functional unit and peripheral support to a RISC-V softcore, features that were lacking in others RISC-V Cores at the time, consequently it was used as a baseline for this work. In figure 4.1, the base system is illustrated, being composed of three main elements, the Scalar Core, which implements a subset of the RISC-V ISA, a level 1 data cache with a direct connection to main memory via the memory interface, and a set of memory mapped peripherals, namely a UART interface, a cycle counter and a timer.

3.1 Scalar Core

The integral part of the system used as a starting point is the Scalar Core, which implements the 32 bit base RISC-V instruction set plus the Integer Multiply and Divide extension. In figure 3.2, one can observe the architecture diagram of the Scalar Core. The core implements an in-order 5 stage pipeline where the ID stage features a Dependency Handler Unit, in this case a Scoreboard, to manage hazards

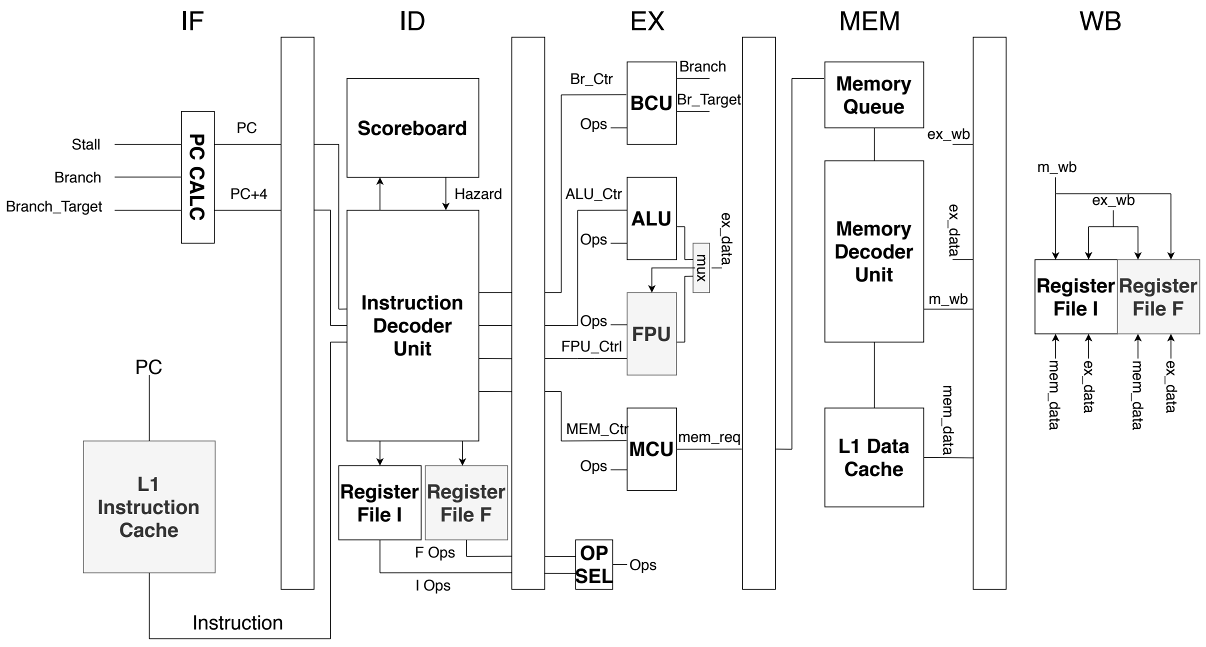


Figure 3.2: RISC-V Scalar Core.

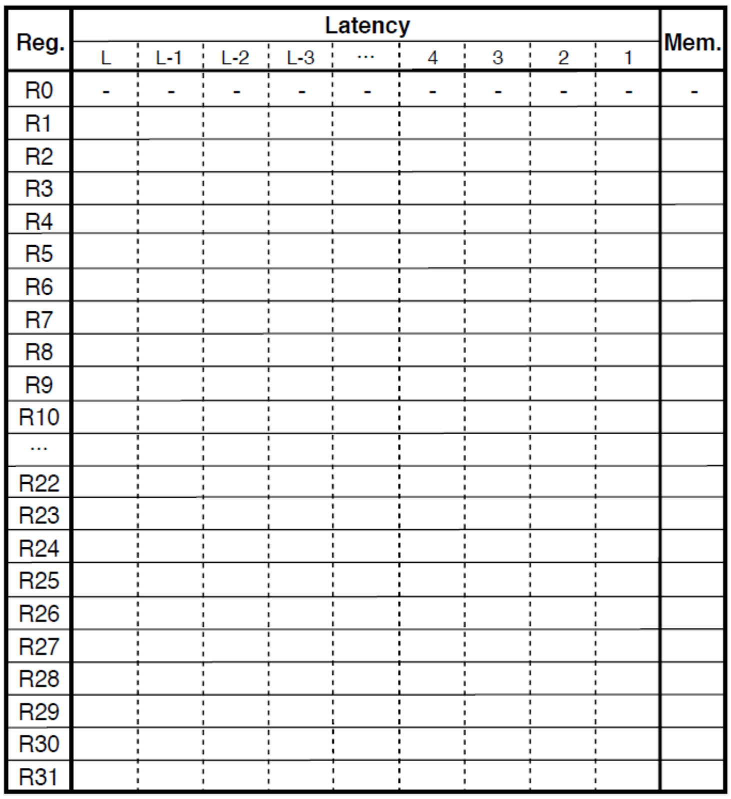
that originate from the use of multi-cycle units, such as the Multiplier and Divider, and from memory accesses.

3.1.1 Instruction Fetch Stage

The Instruction Fetch (IF) stage is responsible for calculating the address of the next instruction using the current Program Counter value or, in case of a branch, the branch target value calculated in the Execute stage. In the original system, the core had a built in BRAM memory that stored the program instructions and as such, the program counter was used to access the BRAM data each clock cycle to provide the next instruction to the Instruction Decode stage.

3.1.2 Instruction Decode Stage

The instruction loaded from the integrated BRAM in the IF stage is forwarded to the Instruction Decode (ID) stage. In this stage a Instruction Decoder Unit decodes the necessary signals to control the Execute and Memory stages in order to perform the instruction requested by the IF stage. In parallel, the Scoreboard unit, represented in figure 3.3, analyses the operands and destination registers of the provided instruction and calculates if a hazard will occur. This hazard flag is then used by the ID stage to stall the pipeline if indeed a hazard is detected. The types of hazards relevant for this in-order pipeline are Read After Write (RAW), Write After Write, due to multi-cycle units, and Structural hazards. Finally the ID stage includes the Integer Register File, which holds the register operands needed by the Execute Stage.



Reg.	Latency									Mem.
	L	L-1	L-2	L-3	···	4	3	2	1	
R0	-	-	-	-	-	-	-	-	-	-
R1										
R2										
R3										
R4										
R5										
R6										
R7										
R8										
R9										
R10										
···										
R22										
R23										
R24										
R25										
R26										
R27										
R28										
R29										
R30										
R31										

Figure 3.3: Configurable Softcore Processor Scoreboard Overview [16]

3.1.3 Execute Stage

After the instruction is successfully decoded and no hazard is detected, the operands are loaded from the Integer Register File and used as inputs to the EX, along with some control signals, mainly the ALU and MEM control Signals. The Execute stage features an ALU capable of computing all the instructions present in the RISCV32IM subset of the RISC-V instruction set, including multi-cycle multiplier and divide units. It also supports a branch control unit, which calculates the branch target address and the branch result, based on the condition specified by the branch instruction. The core uses a static branch not taken prediction mechanism, meaning that the core always assumes the branch is not taken and fetches the next program instruction. If a branch is indeed taken, the pipeline will be flushed, voiding all instructions that were fetched during speculative regime, else the pipeline will continue with normal operation. The EX stage also includes a memory control unit that calculates the memory address based on ID control values and forwards the results to the Memory (MEM) stage. One should note that non-memory instructions skip the Memory stage of the pipeline and are directly forwarded to the Write Back stage.

3.1.4 Memory Stage

If the decoded instruction is a load/store instruction, then the appropriate address is calculated in the EX stage and the relevant control signals are forwarded to the Memory Stage. This stage implements a request buffer, meaning that several memory requests will not stall the pipeline due to the fact that they can be stored in a buffer for later processing. When a request is read from the buffer, the MEM stage identifies if the request is a store or load request and assembles the required signals to form a request to the Data Cache. The MEM stage then waits for the data cache response, i,e the data requested by the core, or the confirmation that the data was stored, and only after the response is acknowledged does the MEM stage read the next request from the buffer. This ensures that the requests are handled in order and that no other handler units need to be developed to reorganize memory requests.

3.1.5 Write Back Stage

After the data has been successfully computed in the EX stage or loaded in the MEM stage, the Write Back Stage (WB) will save the data in the appropriate register, according to the instruction. The Integer Register File was built to have support for 2 separate write channels, the memory write channel and the execute write channel, with the later having priority over the other in case of simultaneous write. However, this mechanism is implemented as a fail-safe as it indicates the presence of a WAW, which should be handled by the Scoreboard, by delaying the hazard generating instruction.

3.2 L1 Data Cache

Since the Scalar Core supports load and store request from memory, the original system contemplated a level 1 Data Cache, with the purpose of drastically reducing load and store latency, since accesses to main memory can be avoided if the data was previously fetched and stored in the data cache.

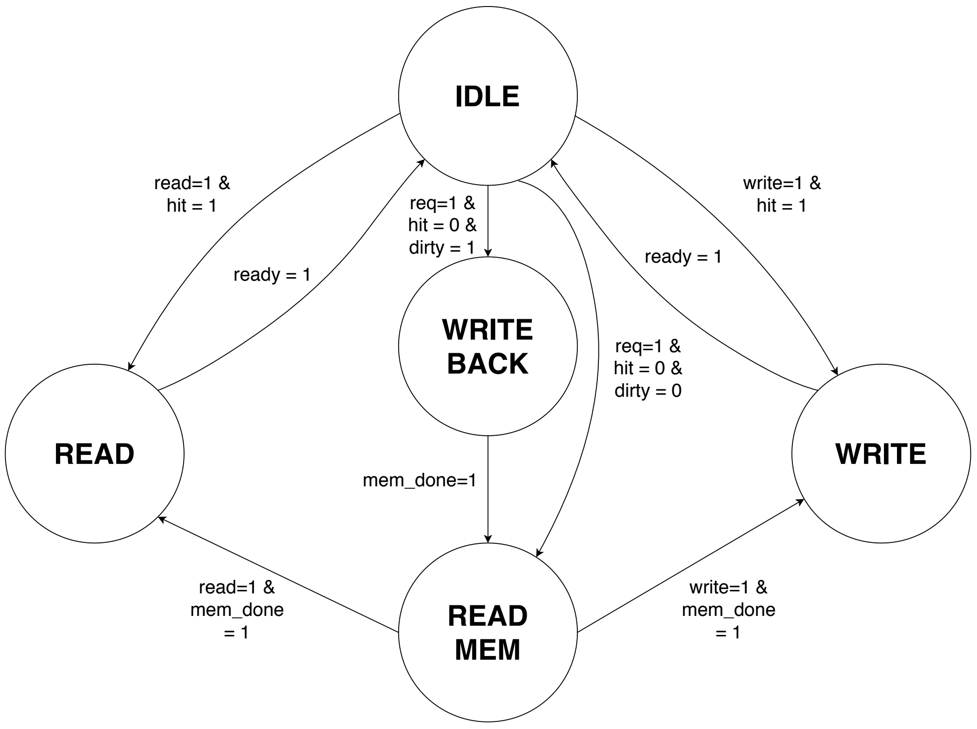


Figure 3.4: Cache WriteBack/WriteAllocate FSM.

The cache can be configured to use two different access mechanisms: direct-mapped or 2-way set associative using the Least Recently Used(LRU) data replacement policy, both with cache line size of 256 bits, number of lines equal to 16 and each line featuring a valid and dirty bit. Both these configurations use the Write-Back/Write Allocate policy, controlled by a cache controller that implements the finite state machine model present in figure 3.4. The cache starts out in the IDLE state and transitions if a read or write request is performed. If the given request results in a cache hit, then in 1 clock cycle the data is read/written depending on the request and the state returns to IDLE. If the request results in a cache miss, then either the cache line in question is dirty, i.e has already been written to by the processor, or it is not dirty. If the cache line is dirty, the state transitions to WRITE BACK and the contents of the cache line are written to main memory. If the cache line is not dirty or after the data has been written to main memory in the WRITE BACK stage, the cache makes a request to read the new line data from memory in order for it to be read or written to, according to the incoming request. Using this state machine, one can reduce the number of requests to main memory, comparing with the write through policy, and reduce memory request latency as a result.

3.3 Peripherals

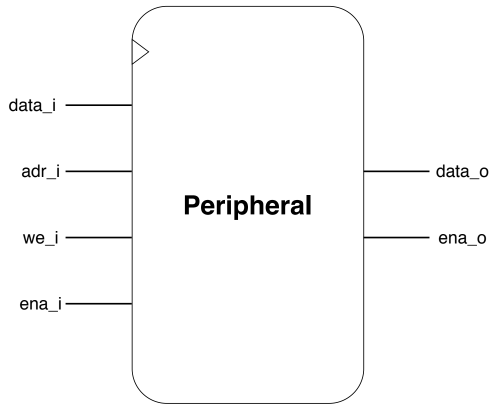


Figure 3.5: Peripheral Interface Diagram.

The original system provided support for several memory mapped peripherals, such as an UART interface, a cycle counter and a timer. On the original architecture, additional peripherals can be added, by implementing the interface illustrated in figure 3.5, where the *data_o* and *ena_o* signals are mandatory only if the peripheral supports read transactions.

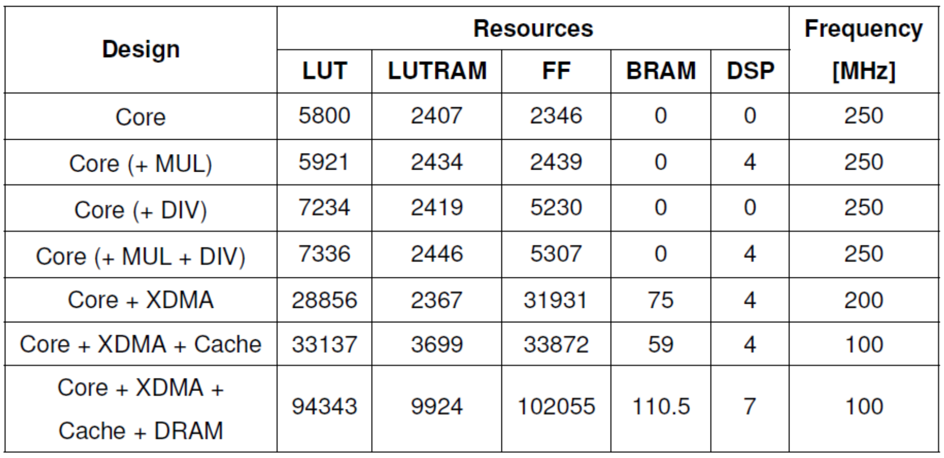
The UART peripheral provides an interface for serial communication using the RX channel for reading and TX channel for writing. This interface can be used to communicate with other peripherals or the host system, if it exists. The cycle counter implemented in the system uses the processor clock to keep track of how many clock cycles have elapsed. The counter can be used to measure cycles of execution for benchmark purposes. The last peripheral, the timer, uses a clock with known frequency to calculate elapsed time. With this peripheral, one can measure execution times by requesting the timer value at the start and end of execution and calculate the difference between the two values. All the peripherals are accessed by normal load or store instructions, meaning that they are mapped to the processor memory space. The start and final address values of each peripheral are represented in table 3.1. Since the peripherals are memory mapped and the entire address range of the processor is not occupied, the system does support inclusion of additional peripherals, if needed. No changes to the core are necessary, only the address decoder must be updated to reflect the new memory mapped peripherals to be added.

Peripheral	Start Address	Final Address
UART	0xFFFFFFF0	0xFFFFFFFC
Cycle Counter	0xFFFFFF00	0xFFFFFF04
Timer	0xFFFFFF80	0xFFFFFF84

Table 3.1: Peripheral Address Ranges.

3.4 Summary

This chapter introduced a system implementation composed of a softcore that supports the RV32IM subset of the RISC-V ISA, represted in the first table of appendix A, a data cache and memory mapped peripherals. The system was implemented in a Virtex UltraScale+ VCU1525 FPGA obtaining the resource usage and operating frequency values represented in figure 3.6.



Design	Resources					Frequency
	LUT	LUTRAM	FF	BRAM	DSP	[MHz]
Core	5800	2407	2346	0	0	250
Core (+ MUL)	5921	2434	2439	0	4	250
Core (+ DIV)	7234	2419	5230	0	0	250
Core (+ MUL + DIV)	7336	2446	5307	0	4	250
Core + XDMA	28856	2367	31931	75	4	200
Core + XDMA + Cache	33137	3699	33872	59	4	100
Core + XDMA + Cache + DRAM	94343	9924	102055	110.5	7	100

Figure 3.6: Resource usage and operating frequency of the different configurations of the Configurable Softcore Processor [16]

Chapter 4

Proposed System Architecture

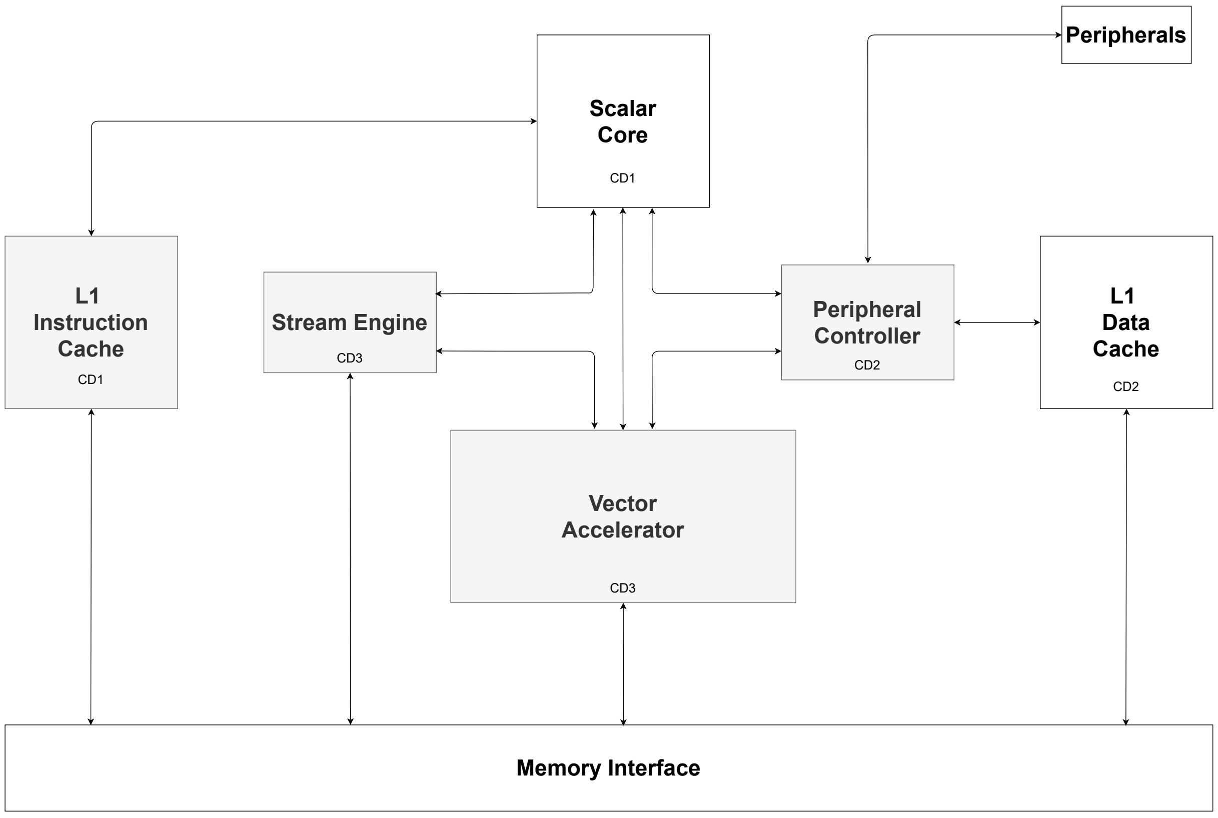


Figure 4.1: RISC-V Streaming Processing System.

This chapter will explain in detail the modifications to the base RISC-V system (figure 3.1) and the architecture of the new hardware blocks introduced by this work, mainly the Streaming Engine, the Vector Accelerator, the Peripheral Controller, and the L1 Instruction Cache, illustrated in figure 4.1. The system provides support for three clock domains, consequently each component has a corresponding clock domain, also represented in figure 4.1. The Scalar Core was modified to have the capabilities of programming and controlling both the Streaming Engine and the Vector Accelerator. The Streaming Engine processes streams configured by the Scalar Core, interfaces with main memory to load or store the data required, and interfaces with the Vector Accelerator to produce/consume data to/from SIMD instructions. The Vector Accelerator provides the necessary hardware to support Single Instruction Multiple Data instructions and a stream memory bank to interface with the Streaming Engine in order to process stream data. At last, the Peripheral Controller performs memory request arbitration between the Scalar Core and the Vector Accelerator, and further decodes the address to redirect the request to either the level 1 Data Cache or to the relevant memory mapped peripheral.

4.1 Vector Accelerator

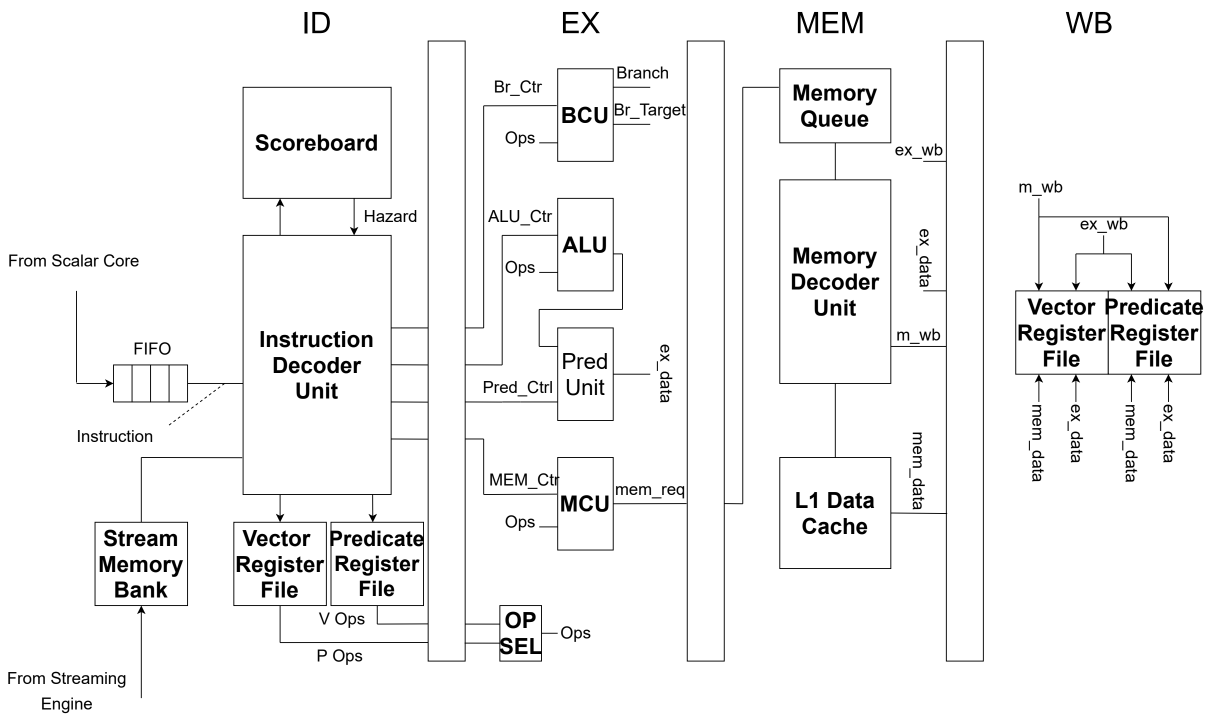


Figure 4.2: Vector Accelerator Pipeline Diagram.

In order to add support for SIMD instructions from the UVE extension, two choices were considered. The first choice considered was to modify the Scalar Core to support the new instructions. Several changes should be made such as the inclusion of a new vector register file and addition of the required functional units and bus width modifications to support vector operations. The second choice was to develop a separate accelerator to support the new instructions. With the latest approach, the new accelerator can run in parallel with the scalar core. Moreover this approach has the added benefit of avoiding SIMD support to affect the critical path of the previously developed Scalar Core, as the accelerator can simply operate with a different clock. The drawback to this method is the need to add inter clock communication between the core and accelerator, which adds to the complexity of the overall design. In this work, the second option was chosen and a Vector Accelerator was designed.

The Vector Accelerator implements a 4 stage in-order pipeline architecture that is controller by the Scalar Core via a asynchronous fifo buffer. It supports a subset of the UVE extension, represented in appendix B, instructions and as such features programmable vector width, as explained in 2.3. The simplified diagram of the accelerator pipeline is illustrated in figure 4.2.

4.1.1 Instruction Decode Stage

The ID stage of the Vector Accelerator receives the instruction from the Scalar Core through a FIFO instead of an instruction cache or integrated memory. As such, the ID stage stalls the pipeline if the FIFO is empty, since there are no instructions to decode. If a valid instruction is ready to be read on the FIFO, then the instruction is decoded and the scoreboard computes if a hazard would be caused by issuing the instruction. If a hazard is indeed detected, the ID stage will stall the pipeline and the instruction will not be read from the FIFO since it needs to be available until the hazard is resolved. The ID stage also controls the Vector Register File and Predication Register File using decoded signals from the incoming instructions and provides the values to the Execute stage.

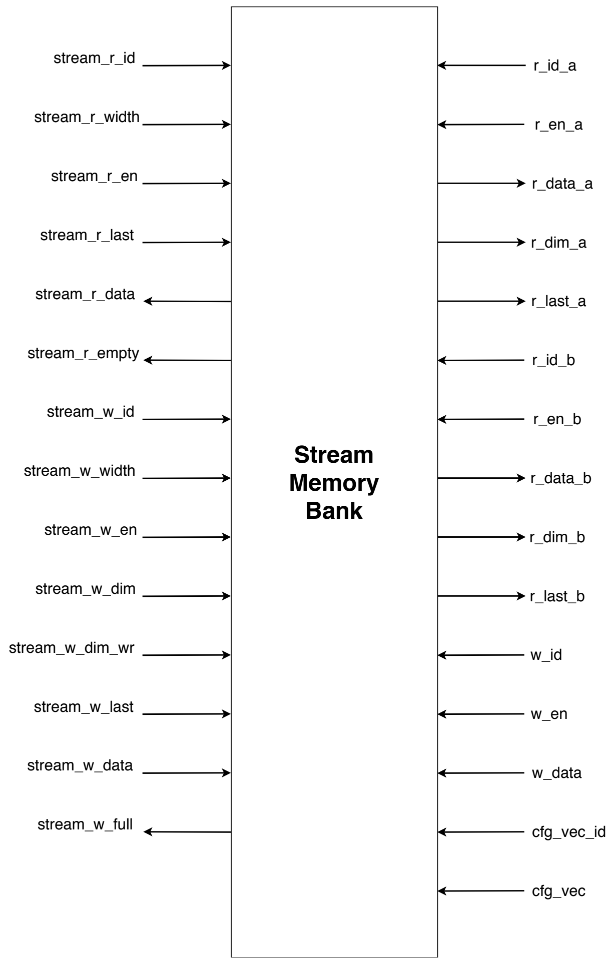


Figure 4.3: Stream Memory Bank I/O.

Considering that the Vector Accelerator supports instructions that operate on streaming data, the Instruction Decode provides a memory bank, separate from the Vector Register File, to store data both incoming and outgoing from/to a stream. As explained in section 4.2.6, to interface with the Streaming Engine, the designed memory bank should implement the necessary signals that comprise the read and write channels. Consequently the Stream Memory Bank, illustrated in figure 4.3 was implemented, were the signals on the left represent channels of communication with the Streaming Engine and signals on the right represent channels of communication with the Vector Accelerator ID stage. The memory bank is composed of 64 first-in first-out buffers, i.e 2 buffers for each register in the Vector Register File. Since streams can be of two types, load and store, two buffers were implemented for each register in order to minimize the control logic needed to operate the buffers. The data incoming from the accelerator is directly written/read from the FIFO, while stream data goes trough a write/read controller that assembles vector sized blocks out of the elements of the stream and writes/reads these blocks when necessary. The signals *cfg_vec_id* and *cfg_vec* are used to make the write controllers sensitive to dimension info, i.e the vector data is written to the FIFO when a dimension finished, even if the valid index, i.e the number of elements valid in the vector, is not equal to the maximum number of valid elements. This configuration is used to force the vector data to mirror the stream topology.

4.1.2 Execute Stage

The Execute stage of the pipeline is composed of four different main blocks that operate on different types of instructions. If a branch instruction was decoded, the branch target address and branch flag are calculated by the Branch Control Unit and then shared to the Scalar Core. The branch and branch values need to be shared to the Scalar Core since it controls the flow of the program with its IF stage, which is not present in the Vector Accelerator.

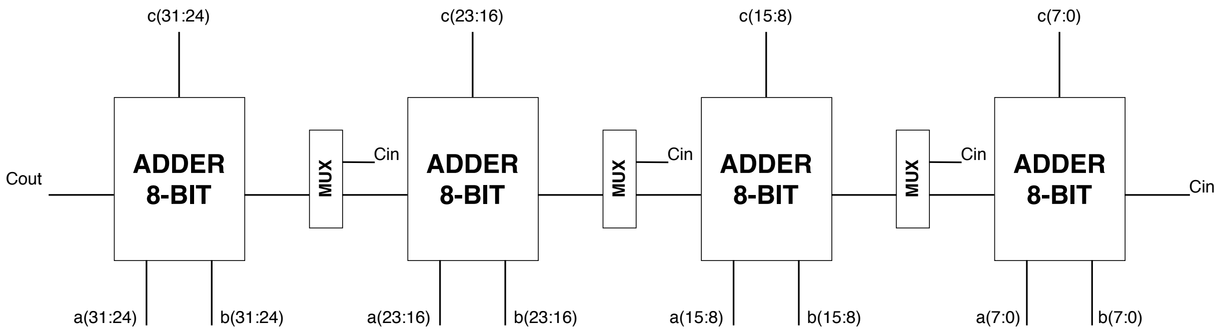


Figure 4.4: 32 bit Variable Width Adder.

If the decoded instruction is an arithmetic or logic instruction, then the result is calculated in the Arithmetic Logic Unit and then predicated using a Predicate Register in the Predication Unit. Since the Vector Accelerator implements UVE extension instructions, it requires support for 8, 16 and 32 bit element width vectors. As such, the ALU supports a variable width adder and variable width multiplier. Since the maximum supported width of an element is 32 bits, the vector adder can be implemented by several 32 bit programmable adders in parallel. The 32 bit adder circuit diagram can be observed in figure 4.4. It is composed of 4 separate 8-bit adders that can work in parallel if all the multiplexers output the carry in value, but can also run cascaded by controlling the select bits of the multiplexers. If the right most multiplexer outputs the carry out of the first adder and the left most multiplexer outputs the carry out of the third adder then both adders 1/2 and 3/4 are paired and the circuit outputs two 16 bit additions. If all multiplexers output the carry out values of the respective adders, then all adders are cascaded and the output is a 32 bit addition.

As with the vector adder, the vector multiplier can be implemented using several 32 bit programmable width multipliers. The 32 bit programmable multiplier generates its partial products using the radix-4 booth encoding algorithm. The first step of the partial product generation is to look at bits of the multiplier 3 at a time, starting with a trailing 0. By using this method, each block of 3 bits can have 8 possible values. Each value of the 3 bit block results in an operation made to bits of the multiplicand, and the possible outcomes are represented in table 4.1. This algorithm is easily extended to variable element width multiplication, one needs only to divide the multiplier into sub-products of the selected element width and apply the algorithm to all of them.

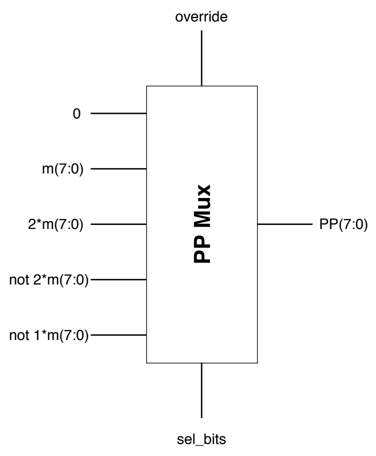


Figure 4.5: Partial Product Operation Select Mux.

To carry out the operations needed on the multiplicand bits the resulting block values are first encoded, since we have 5 possible operations and 8 different block values. After the bits are encoded, they are used as the select bits of a multiplexer which chooses between the desired operations. The multiplexer that operates on the first 8 bits of the multiplicand can be observed in figure 4.5. The multiplication by 2 is done using a simple shift right operation and for the negative multiplication the values are also negated. The output values that correspond to the multiplication by -2 and -1 are not in two-complements after the multiplexer but will later be properly incremented when the partial product is sign extended. One should also note that besides the select signal, the multiplexers that operate on the multiplicand additionally have an override signal. This signal forces the output to be 0, and is used to disable certain partial products depending on the element width value of the operation.

After all the partial products are generated and properly sign extended, they need to be summed together. To this end, the multiplier implements a Wallace Tree, composed of Carry Look Ahead Adders (CLA), that performs a reduction of the partial products. Since the multiplier inputs are 32 bits wide, then 16 partial products are generated and as a result of that, a 6 level Wallace Tree, illustrated in figure 4.6, is needed. The four 8 bit multiplications and two 16 bit multiplications can then be obtained by summing intermediate values of the Wallace Tree, while the 32 bit multiplication result is the output of the tree.

Lastly, if the decoded instruction is a memory instruction, then the Memory Control Unit calculates the base address of the request, the size, in number of elements, of the request and the remaining control signals needed by the Memory Stage to perform the request.

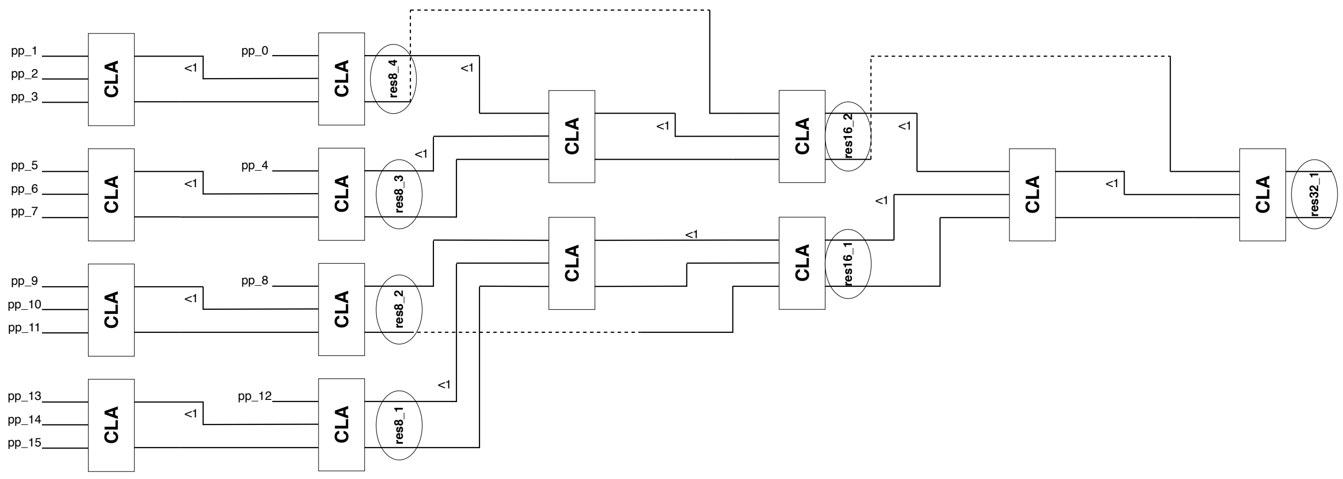


Figure 4.6: 6 Lever Wallace Tree Reduction.

Block	Partial Product Operation
000	0*multiplicand
001	1*multiplicand
010	1*multiplicand
011	2*multiplicand
100	-2*multiplicand
101	-1*multiplicand
110	-1*multiplicand
111	0*multiplicand

Table 4.1: Radix-4 Block Values and correspondent operation.

4.1.3 Memory Stage

The Memory stage of the Vector Accelerator employs the same memory queue technique used in the Scalar Core, i.e the memory requests are first stored in a FIFO to avoid stalling the pipeline due to the high latency of memory accesses. In contrary to the Scalar Core, the memory requests of the Vector Accelerator can have dynamic size, i.e one request can load/store between one and the max number of elements of a vector. For that purpose, the Memory stage provides control bits to the Peripheral Controller that give information about the element width and size of the transfer. Using these values the memory stage also calculates the number of valid elements in the vector, in case of a load instruction, since this meta information is appended to the vector in the register file.

4.1.4 Write Back Stage

The Write Back stage of the accelerator is composed of two Register Files, i.e the Vector Register File and the Predicate Register File. The Vector Register File supports 32 bit registers with programmable size equal to the vector size desired plus additional meta information. This meta information is composed of vector width bits and valid index bits. The Predicate Register File supports 16 registers were the size, in bits, equals the number of bytes of a vector in the implementation. Using this implementation, each predicate register acts as a byte mask for compute operations, i.e if bit i is set then the byte i of the vector register will be written. Although the Predicate Register File supports 16 registers, only the first 8 registers, P0 to P7 are allowed to be used, following UVE extension ISA specifications. Both register files implement a dual write channel interface, one channel for execute data and one for memory data, however the memory write interface of the predicate register is unused in the current implementation.

4.2 Streaming Engine

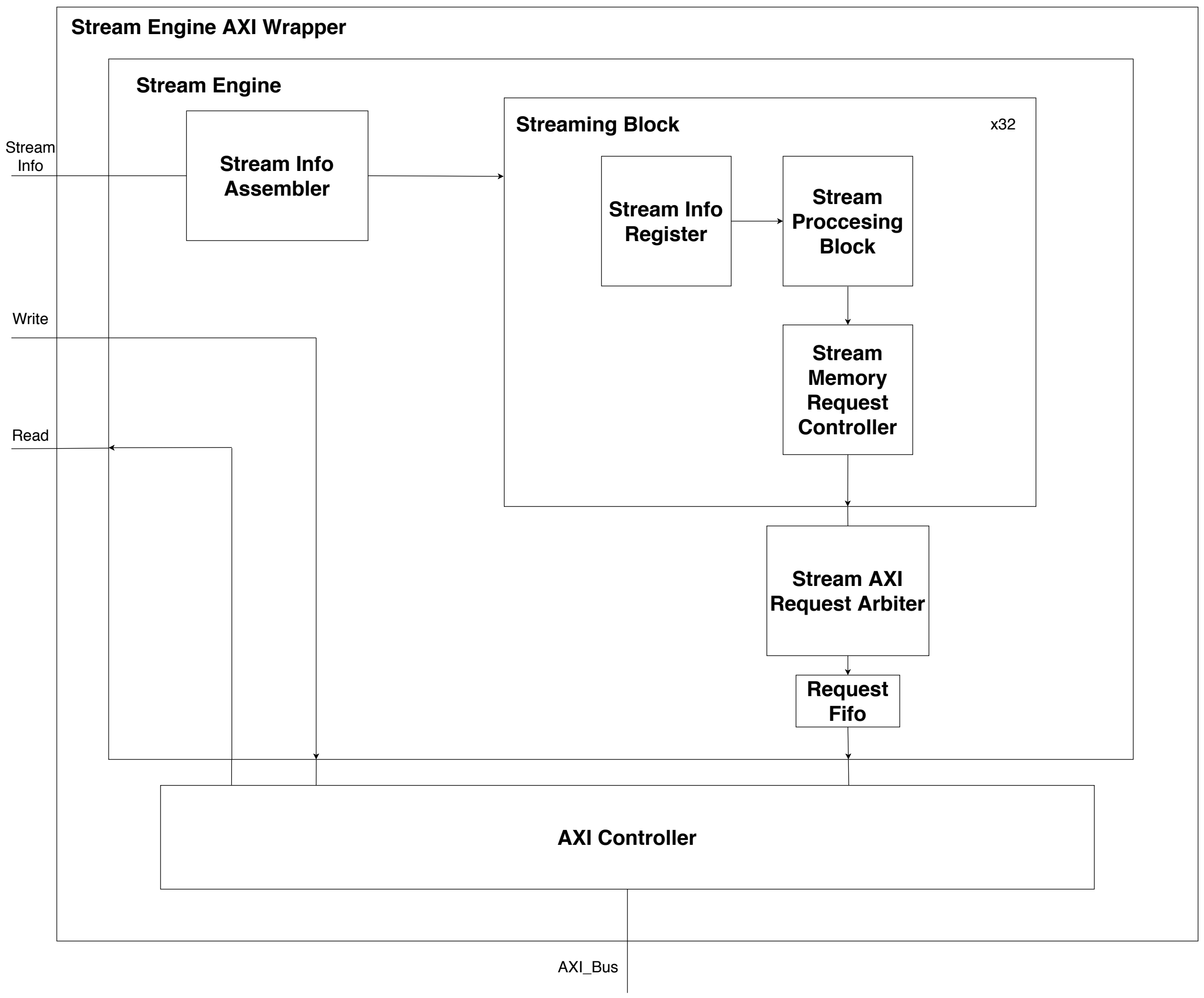


Figure 4.7: Streaming Engine Architecture Overview.

In order to get the full flexibility and performance benefits of the UVE extension, in particular for processing large amounts of data, support for the streaming instructions (appendix B) should be present. With that goal in mind, a hardware streaming solution was designed and coupled with both the Scalar Core and the Vector Accelerator. The Scalar Core provides the decoded streaming control signals from the streaming instructions to the Streaming Engine while the Vector Accelerator implements an interface that allows processing of data from streams by consuming or producing data from/to the Streaming Engine.

As this is a streaming block designed for use with the UVE Extension, it follows the streaming architecture described in detail in section 2.3. As such, the Streaming Engine supports up to 32 concurrent streams. A stream can be a load or store stream, and the data width can be 8, 16, 32 or 64 bits, configurable per stream. Each stream has hardware support of up to 8 associated stream descriptors, however from the three types of descriptors described in section 2.3, only the base stream descriptors are supported.

The Streaming Engine is composed of 5 main types of hardware blocks, represented in figure 4.7. The first block is the Stream Info Assembler, which assembles streaming information and saves it in the corresponding Stream Info Register based on Stream ID. The second and third blocks are the Stream Processing and Stream Memory Request Controller blocks, whose function is to calculate stream addresses each clock cycle and create AXI requests from those addresses, respectively. The fourth block, Stream AXI Request Arbiter, manages the access of the several Memory Request Controllers to the Request Fifo. The last block, AXI Controller, implements a finite state machine that controls load/store requests to/from main memory via the Advanced eXtensible Interface(AXI) Protocol[1].

4.2.1 Stream Info Assembler

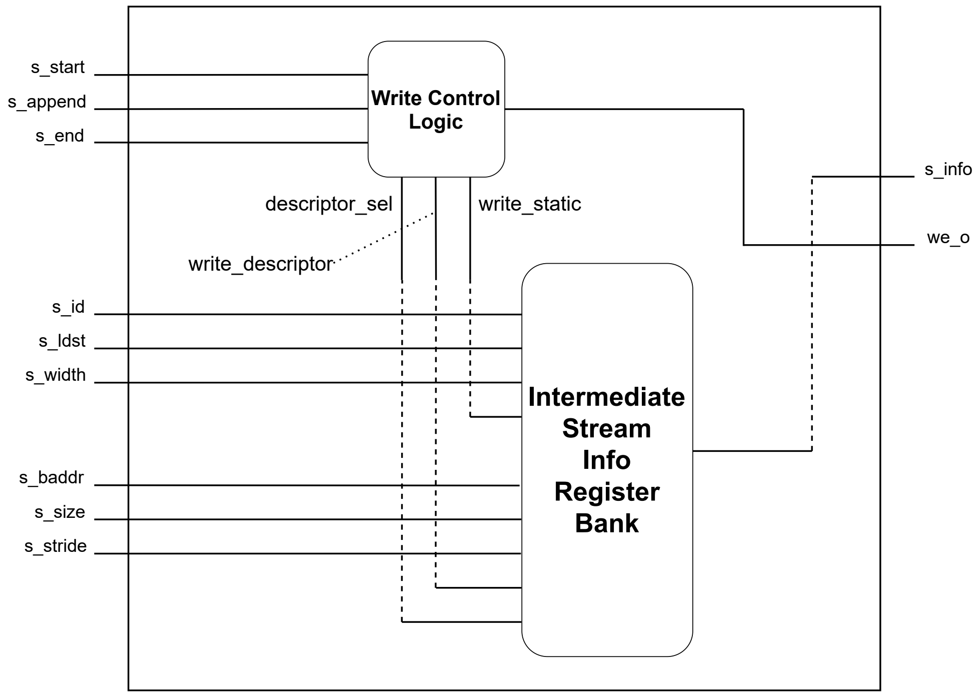


Figure 4.8: Stream Info Assembler Diagram.

Since we can have streams defined by multiple streaming instructions, the control signals for a given stream received each clock cycle need to be assembled into a Stream Info Register for later use in the Streaming Engine. For this reason, the first stage of the Streaming Engine consists of an assembly block that keeps a record of stream info until it is ready to be written to the respective Stream Info Register. To configure a stream, the Stream Info Assembler block uses the control and data signals, properly identified in table 4.2 and illustrated in figure 4.8.

The signal *s_start* informs the Streaming Engine that a new stream must be created and that the first descriptor information, *s_baddr*, *s_size* and *s_stride*, along with static stream information, such as *s_id*, *s_ldst* and *s_width*, should be stored. If *s_append* is set, the new descriptor information will be added to the already created stream with id equal to *s_id*. Finally, if *s_end* is set, then the last descriptor information is stored and the stream with id *s_id* is written to the corresponding Stream Info Register and removed from configuration.

Signal	Type	Width(bits)	Description
s_start	control	1	Start configuration of new stream.
s_append	control	1	Append descriptor to stream
s_end	control	1	End configuration of stream
s_id	data	5	Unique identifier of the stream
s_ldst	data	1	Stream type, load or store.
s_width	data	2	Width of the data elements of the stream
s_baddr	data	32	Base address of the descriptor
s_size	data	32	Size value of the descriptor
s_stride	data	32	Stride value of the descriptor

Table 4.2: Stream Info Assembler Input signal types and widths.

One should note that *s_start*, *s_append* and *s_end* are the only signals that trigger value changes in the Stream Info Assembly registers and that not all combinations of these signals form valid requests. As such, a table with the valid combinations of the control signals and a brief description of what each request represents, as per the *UVE* extension standard, can be observed in 4.3.

s_start	s_append	s_end	Description
1	0	0	Starts configuration of a new stream with id = *s_id*, saves static and first descriptor information
0	1	0	Appends new descriptor information to the stream with id = *s_id*
0	0	1	Appends last descriptor information to the stream with id = *s_id* and removes it from configuration
1	0	1	Single cycle configuration of a simple stream with one descriptor

Table 4.3: Stream Info Assembler valid requests.

4.2.2 Stream Processing Block

After a stream is properly configured in the configuration stage, the resulting data is written to a Stream Info Register and the contents of this register are directly connected to the Stream Processing block, as can be seen in figure 4.9. The stream info register holds the required information to process a stream, information that can be divided into two groups, stream properties and descriptor information.

Stream property information includes the stream id, stream type, i.e load or store, and element width while descriptor information includes the base address, size and stride values of each descriptor of the stream.

Stream Descriptor Update Circuit

The basic information block of a stream is the stream descriptor and is composed of three values:

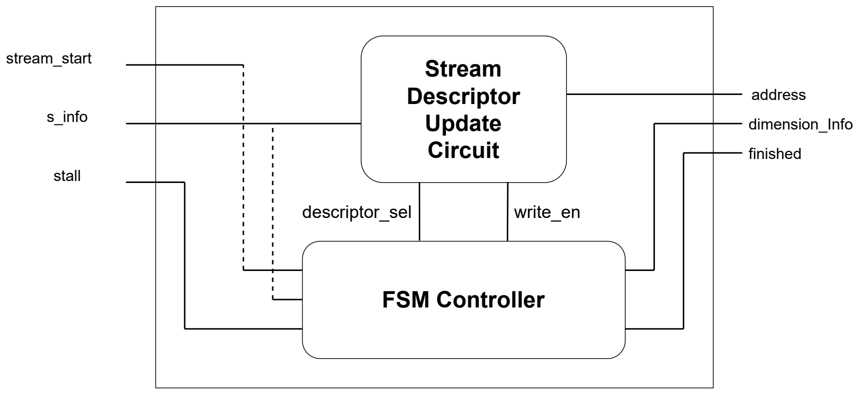


Figure 4.9: Processing Block Diagram.

- base address: address offset
- size: number of elements accessed
- stride: spacing, in number of elements, of two subsequent accesses to memory

These three values fundamentally describe a 1 dimensional access to memory where each address can be calculated by

$$address_i = base\_address + i * stride \quad i = 0, ..., size - 1 \,. \tag{4.1}$$

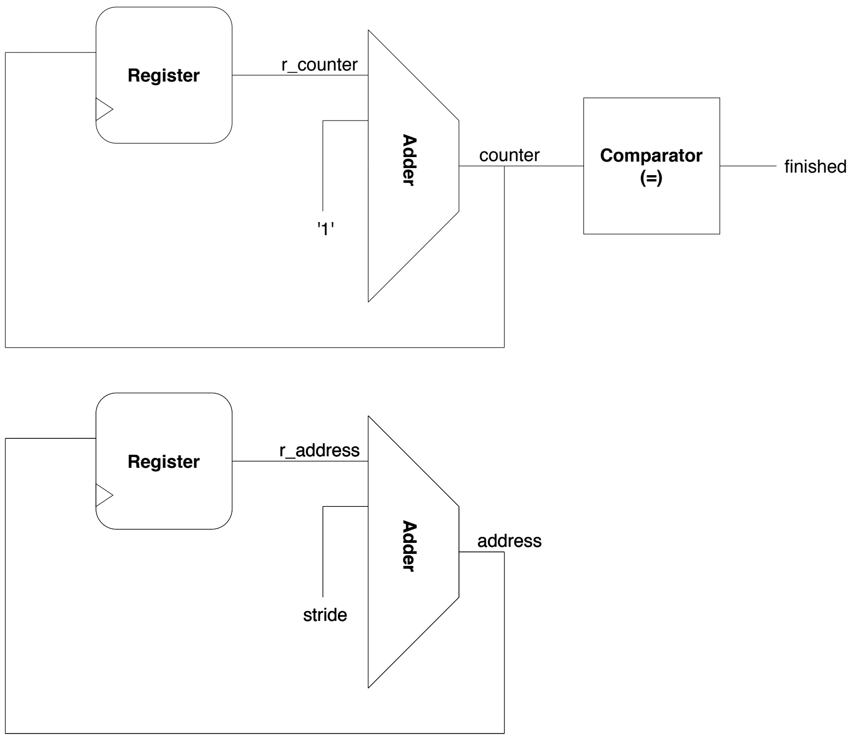


Figure 4.10: Single Descriptor Update Circuit Diagram.

One can easily implement equation 4.1 in hardware, resulting in the circuit illustrated in figure 4.10. Firstly we implement multiplication by doing successive addition, using an adder and an accumulate register for the *address* signal. Then, the finished flag of the descriptor can be easily calculated by keeping track of the iteration number using a register, incrementing its value every clock cycle and comparing it to the descriptor size. Note that using this circuit, the initial value of the accumulate register is the base address value of the descriptor and the stride input to the adder is properly shifted according to the stream element width value. The shifting of the original stride value of the descriptor is needed since it is originally measured in number of elements and needs to be converted to number of bytes to be compatible with a byte addressable memory.

The circuit in figure 4.10 is perfectly capable of handling any stream which requires a 1 dimensional access pattern to main memory, however streams in the UVE extension can have up to 8 descriptors, or 8 dimensions, so a generalization of the circuit in question is necessary.

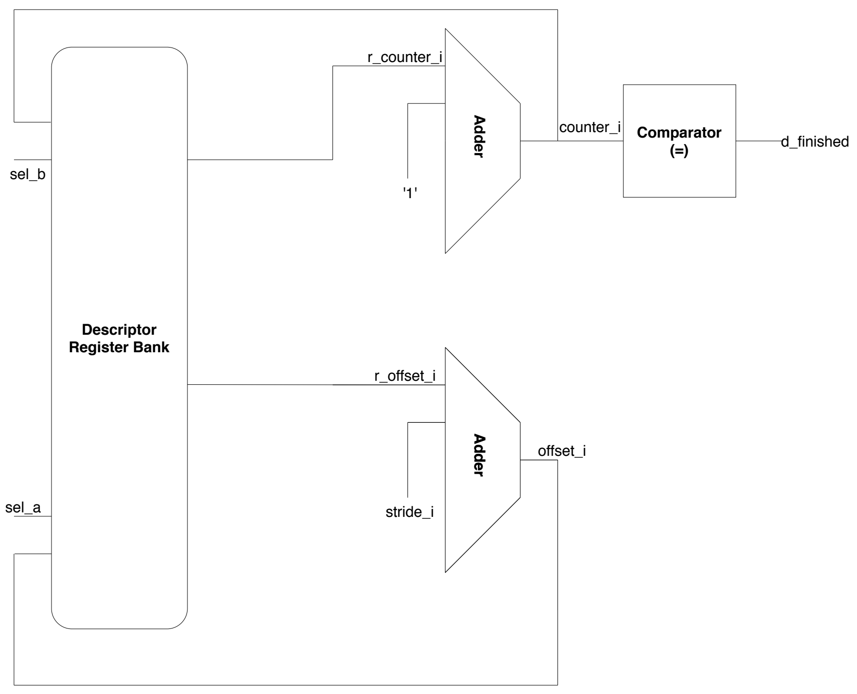


Figure 4.11: Multiple Descriptor Update Circuit Diagram.

Firstly, one should note that the descriptors of a given stream are ordered hierarchically, as explained in section 2.3, and as such, the proposed solution updates a single descriptor is updated during each clock cycle. This allows reusing the circuit in figure 4.10 for all the descriptors of a given stream, meaning only the sequential elements need modifications to generalize the circuit to operate on several descriptors. Consequently the registers present in the original circuit were replaced by a single register bank holding registers for the necessary descriptor offsets and counters. With this approach, the counter and offset values of a given descriptor are selected and updated each clock cycle, as illustrated in figure 4.11. The address can then be calculated each clock cycle by adding all the offsets of the descriptors of a given stream.

In order for the processing block to be complete and compliant with the UVE streaming instructions, a finite state machine was added to provide not only control signals to the descriptor register bank in figure 4.11, but also implement the hierarchical organization of the stream descriptors described in 2.3. The processing block starts in IDLE mode, as can be observed in figure 4.12 and a state change is

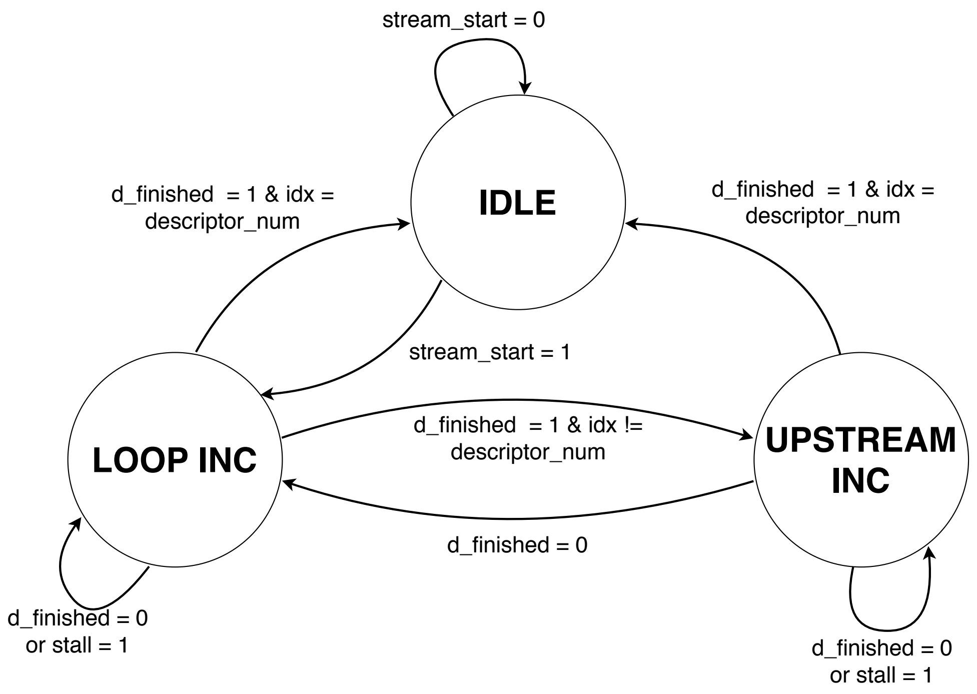

stream_start = 0
IDLE
d_finished = 1 & idx = descriptor_num
d_finished = 1 & idx = descriptor_num
stream_start = 1
LOOP INC
d_finished = 1 & idx != descriptor_num
UPSTREAM INC
d_finished = 0
d_finished = 0 or stall = 1
d_finished = 0 or stall = 1

Figure 4.12: Processing Block Finite State Machine Model.

only triggered if a stream is ready to start, if it has left configuration stage and all the information has been loaded to the Stream Info Register. When a stream starts, the processing block changes state to LOOP_INC and starts updating the offset and counter values corresponding to the first descriptor, present in the descriptor register bank, and outputting a valid address each clock cycle until the dimension finished flag is set. When said flag is set, there are two possible state changes, either the stream has only one descriptor and, as such, the state changes to IDLE to indicate that no more processing is necessary, or the stream has more descriptors upstream that need to be incremented and the state changes to UPSTREAM_INC. The UPSTREAM_INC state has a key difference when compared with the LOOP_INC state, it does not output any address but rather only updates the descriptor values in question. In this state, if the given descriptor is the last descriptor of the stream and it finished, the processing block transitions to IDLE mode and the stream ends, else if the given descriptor finished but is not the last descriptor of the stream, there is no change in state and in the next clock cycle the next descriptor upstream will be updated. Finally if the descriptor in question did not finish, if the finished flag is set to 0, then the state changes back to LOOP_INC to continue address generation. The Finite State Machine also supports a stall flag that ensures not only that the state is not changed, but also that the internal registers of the processing block do not change while the flag is active. One final detail to note is that, when the processing blocks transitions from LOOP_INC to UPSTREAM_INC or IDLE, the dimension-specific finished flags for each descriptor are stored and later combined since they will be necessary to provide the vector core with information about when a given dimension ends.

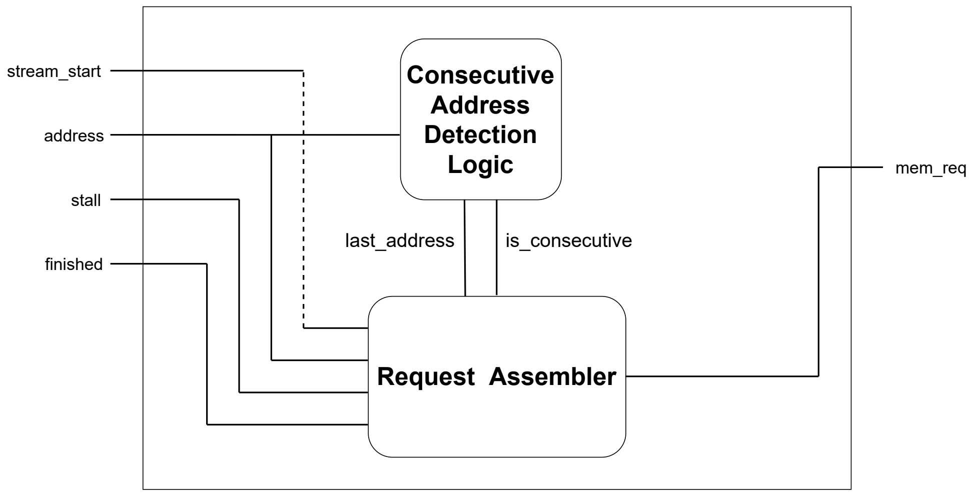


Figure 4.13: Stream Memory Request Controller Diagram.

4.2.3 Stream Memory Request Controller

Section 4.2.2 explained the process of calculating a single address from the stream each clock cycle using stream descriptors. These addresses could be used directly to issue requests to main memory however, in order to use the maximum bandwidth possible from the bus connecting the Streaming Engine to main memory, a Memory Request Controller, illustrated in 4.13, was designed and implemented.

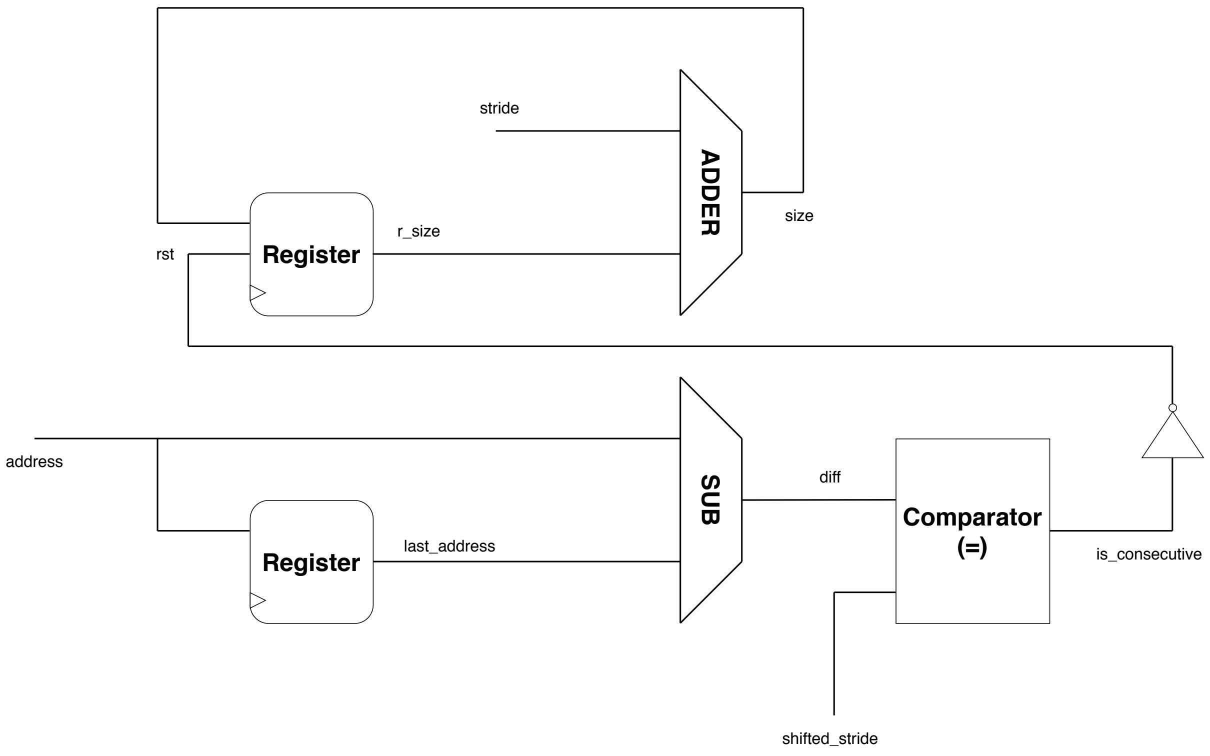


Figure 4.14: Memory Request Controller Detection Circuit.

The proposed implementation of the Memory Request Controller uses the fact that consecutive addresses of a stream are usually correlated, i.e spaced apart by a given stride value. Hence it sends a single request to main memory that loads/stores several elements using a single base address and stride information. Since a stream can have a stride value for each of its descriptors, there is a choice to be made. In this implementation the stride of the first descriptor was chosen because, if we take into account how the descriptors affect the address generation, we can deduce that the most used stride value is the one from the first descriptor, since this is the descriptor that is updated more often. With this in mind, the circuit illustrated in figure 4.14 was designed. The circuit uses the current address, the last calculated address and the first descriptor stride to evaluate if the current address and last address are consecutive addresses in the stream, i.e if they are spaced by the stride value. The stride value used for the comparison with the result from the adder is appropriately shifted to compensate for the fact that the stride is represented in number of elements instead of number of bytes. If the addresses are consecutive, then the Memory Controller will not send a request to memory, but will instead wait for the next clock cycle to evaluate the new address. If, however, the address values are not consecutive, then the Memory Controller will output a memory request with base address, size and stride information. One exception to this is when the first address of a stream is calculated and detected by the Memory Controller. In this case, since there is no last address saved, the controller assumes the addresses are consecutive in order to guarantee the correct behaviour of the system. When a request is sent, i.e two non consecutive addresses were detected, the counter that keeps track of the number of elements in a request is reset, and the current address will be saved as the base address of the next request. If the number of elements in a request equals the total width of the bus that connects the Streaming Engine to main memory, then the Memory Controller will force a request and reset the appropriate variables. There is one further detail to have in consideration, which is when the stride value is greater than the number of elements allowed in a single burst request to main memory. If this is the case, then the Memory Controller will send one request to main memory for each address in the stream. The Memory Controller further supports three control bits, *stream_start*, *stall* and *finished*. The *stream_start* bit signals the Memory Controller that the stream associated has started and that internal state changes are allowed, the *stall* bit stalls the Memory Controller and the *finished* bit signals that the stream has finished and a global reset of the controller can be performed in order to be ready to process a new stream when available.

4.2.4 Stream Request Arbiter

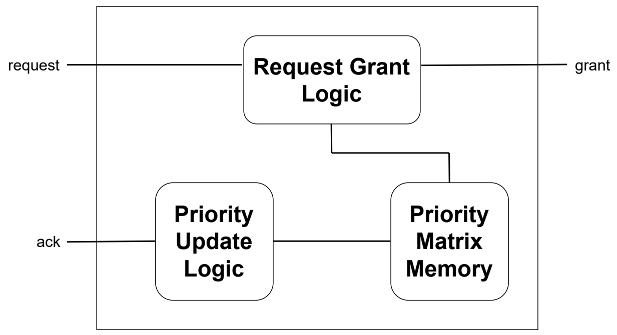
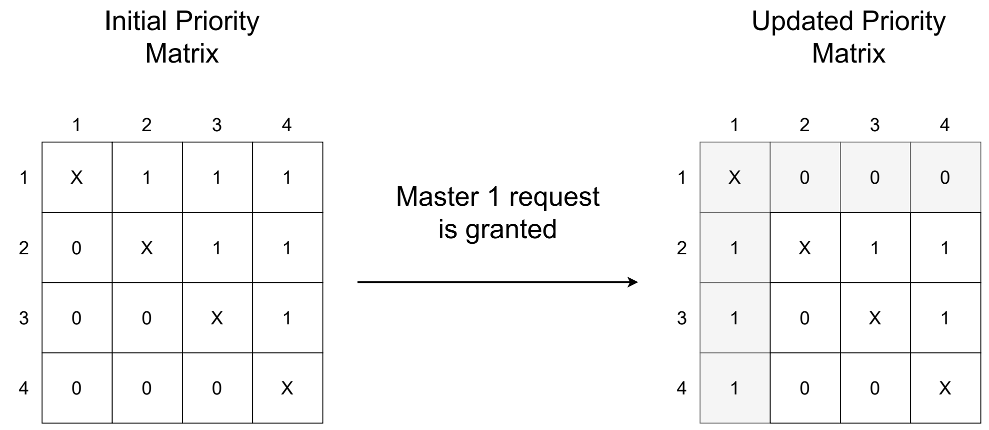


Figure 4.15: Stream Request Arbiter Diagram.

The Processing and Request stages of the Streaming Engine have been explained in detail in sections 4.2.2 and 4.2.3 and could be used to directly connect to memory via AXI interface if only one stream was processed concurrently. However the UVE extension standard requires processing of up to 32 streams at the same time. The issue that arises from concurrent processing of streams is the arbitration of memory requests. Using the AXI Protocol and a single memory interface to main memory, only one request can be made at a time. Since this limitation is present in the presented system, a request arbiter, figure 4.15, was integrated into the design of the Streaming Engine.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & X & 1 & 1 & 1 \\ 2 & 0 & X & 1 & 1 \\ 3 & 0 & 0 & X & 1 \\ 4 & 0 & 0 & 0 & X \end{array} \xrightarrow{\text{Master 1 request is granted}} \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & X & 0 & 0 & 0 \\ 2 & 1 & X & 1 & 1 \\ 3 & 1 & 0 & X & 1 \\ 4 & 1 & 0 & 0 & X \end{array}$$

Figure 4.16: Priority Matrix Update Example.

To choose the best arbitration scheme, one must analyse the behaviour of the stream requests and how different streams can be related. A stream can have two different types, load and store. Lets assume that there are two streams being processed in the streaming engine and that they are connected, i.e the values loaded from the first stream are processed in a external compute unit and then the result is forwarded back to the store stream to be stored in main memory. Let us also assume that, even though the streams in question are related, the stride values of the first descriptor of each stream are such that the load stream produces a valid request every 4 clock cycles and the store stream produces a valid request every 2 clock cycles, as an example. In this case, the first stream will, with each request, load 4 elements from main memory and the second stream will store 2 elements with each request. In this case, both streams have the same throughput per time unit, i.e the first stream requests a load of 4 elements in 4 clock cycles and the store stream requests 2 stores in 2 clock cycles, and produce requests concurrently every 4 clock cycles. However, even with the same throughput per time unit, if the store stream is given the priority in clock cycle 4, the store request will try to request data that has not yet been loaded and processed, resulting in a permanent stall of the Streaming Engine. To prevent this situation we must guarantee that if two streams, one load and one store, are both trying to send requests to main memory, the load stream will have priority. The first mechanism introduced for this purpose is the use of a stall bit. A store stream will always stall if there is no data ready to be stored. This mechanism will also guarantee that if two streams, one load and one store, are related, then the load stream will always make a request first since the store stream will be stalled until the first batch of data requested from the load stream has been computed and made available for the store stream. The second mechanism introduced was the matrix arbitration scheme used by the Stream Request Arbiter.

This dynamic arbitration scheme guarantees that the master who last used the common bus will have the lowest priority, using an internal priority matrix. If the bit at row *i*, column *j* of the priority matrix is set, then master *i* has priority over master *j*. Each time a request is served for master *i*, row *i* will be set to zeros and values in column *i* will be set to 1, ensuring that all masters will have priority over master *i*, as can be seen in the example of figure 4.16. What this means for our test case is that at clock cycle 4, even though both the store and load streams make concurrent requests, the load stream will be served first since the bus was last accessed by the store stream at clock cycle 2. Thus using this arbitration scheme the load and store order is correctly preserved if streams are related and the use of the common bus is evenly distributed if streams are not related, since bus overpopulation by a single master is not allowed using this arbitration scheme, unless no other masters have valid requests to be served.

4.2.5 Stream AXI Controller

The previous sections described how the Streaming Engine configures, processes and manages requests of streams, yet there is still a need to forward the requests produced by the engine to the main memory. There are several protocols that one could use to achieve this, and the choice is dependent of the support of the system in question. In this thesis the AXI Protocol was used, since the original system already supported a connection to main memory via a Xillinx MIG IP, which is configured to use the AXI Protocol.

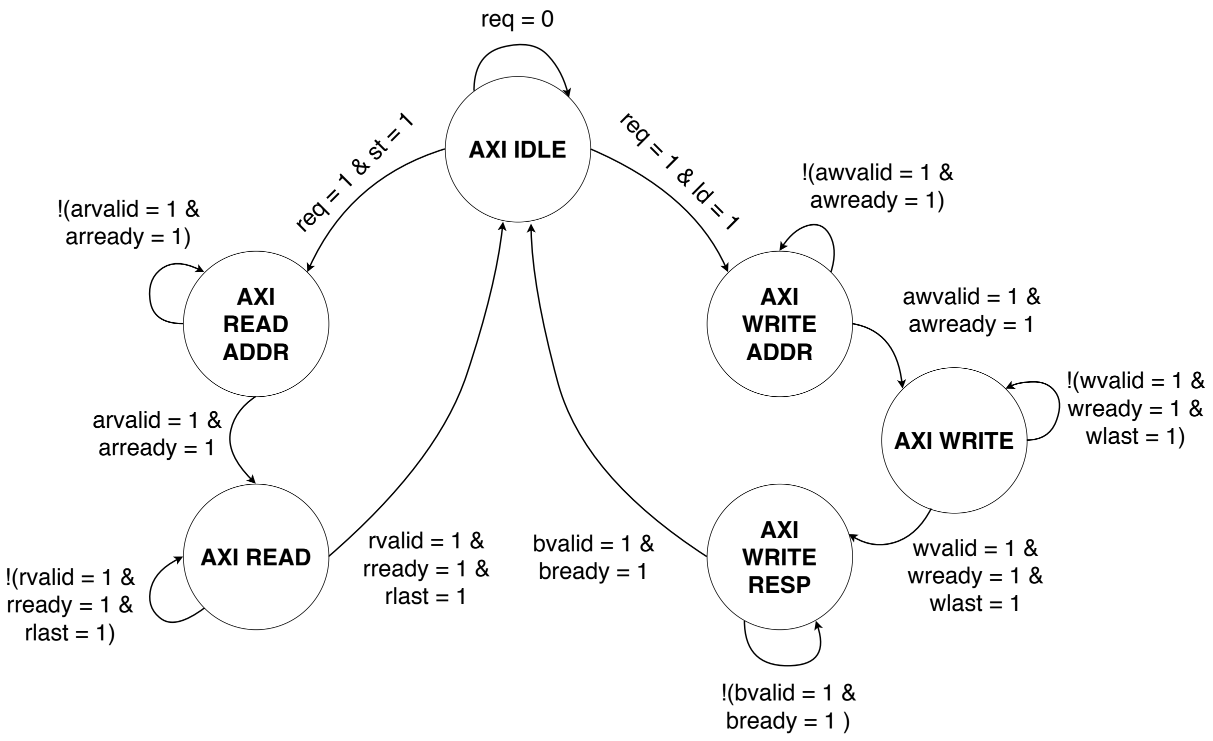

req = 0
AXI IDLE
req = 1 & st = 1
req = 1 & ld = 1
!(arvalid = 1 & arready = 1)
AXI READ ADDR
!(awvalid = 1 & awready = 1)
AXI WRITE ADDR
awvalid = 1 & awready = 1
!(wvalid = 1 & wready = 1 & wlast = 1)
AXI WRITE
arvalid = 1 & arready = 1
AXI READ
!(rvalid = 1 & rready = 1 & rlast = 1)
rvalid = 1 & rready = 1 & rlast = 1
bvalid = 1 & bready = 1
AXI WRITE RESP
wvalid = 1 & wready = 1 & wlast = 1
!(bvalid = 1 & bready = 1)

Figure 4.17: AXI Controller Finite State Machine Diagram.

As such, an AXI Controller block was developed and added to the Streaming Engine structure. In order to control the transfers to and from main memory, the AXI Controller implements the finite state machine, whose diagram can be observed in figure 4.17. The controller starts in an idle state and can change either to AXI_READ_ADDR or AXI_WRITE_ADDR depending on the type of incoming request, load or store. In the AXI_READ_ADDR, the base address, size, length and type of transaction are specified by the master. The base address is the first address of the transaction, the size indicates the number of total elements to load, the length specifies the element width for this transaction and finally the type of the transaction indicates how the slave will calculate the transaction addresses considering the base address. In this implementation the type of the transaction is always increment, meaning that the next address is obtained by incrementing the previous address, for strides bigger than zero, and fixed, meaning that the next address is equal to the previous address, for strides equal to zero. After the handshake on the read address channel occurs, the state changes to AXI_READ. During this state the elements are read from main memory and for each element there is an handshake using the *rvalid* and *rready* signals. The final element is identified by the *rlast* signal and triggers the change of state to AXI_IDLE. Since requests made by the engine can have stride values different than 1, a counter was implemented to keep track of what elements should be ignored and what elements should be stored by the Streaming Engine. If the incoming request is of the store type and the controller is idle, it will transition to the AXI_WRITE_ADDR state, which works analogously to the AXI_READ_ADDR state, apart from using different channels to communicate. The controller then transitions to the AXI_WRITE state where the elements are written to the main memory. Since store requests can have stride values different than one, we use the *wstrobe* signal to indicate which values should be written. The *wstrobe* signal acts as a byte mask for the data sent to main memory, meaning that if a certain byte in the bus is to be written to memory, the corresponding bit in *wstrobe* should be set. After the last element is written, the state transitions to AXI_WRITE_RESP where a final handshake is performed and the transaction is complete.

4.2.6 Stream Data Input/Output Interface

The Streaming Engine implements a general read and write interface that can be used by an external block, the Vector Accelerator in this work, to load and/or store data using streams. The read and write interface signals were assembled into two separate tables, 4.4 and 4.5, and a small description of the function of each signal is provided.

Signal	I/O	Description
r_id	out	Id of the stream that is requesting data
r_width	out	Element width of the data being requested
r_en	out	If 1, then the incoming read request is valid
r_last	out	Bit that signals the current read request is the last request of the stream
r_data	in	Data requested by the stream
r_empty	in	Informs the Streaming Engine if there is data to be read

Table 4.4: Streaming Engine Read Interface.

Any external block that may want to use the Streaming Engine needs to implement both the Read and Write Interfaces. Some output signals from the engine can be ignored if the external block does not need to use them, i.e dimension info, dimension write and last signals, however all other signals must

Signal	I/O	Description
w_id	out	Id of the stream that is requesting data
w_width	out	Element width of the data being requested
w_en	out	If 1, then the incoming write request is valid
w_dim	out	Vector of bits that signal if stream dimensions have finished
w_dim_wr	out	If 1, then dimension info is valid
w_last	out	Bit that signals the current write request is the last request of the stream
w_data	out	Data requested by the external block
w_full	in	Informs the Streaming Engine if data can be written

Table 4.5: Streaming Engine Write Interface.

be correctly implemented.

4.3 System Additions and Modifications

Sections 4.1 and 4.2 described in detail the architecture of the Streaming Engine and Vector Accelerator added to the system. However, further modifications to the system were made to both complement the addition of the the new hardware blocks, namely the Peripheral Controller, and to improve the already existing Scalar Core, namely the Instruction Cache and addition of floating point support.

4.3.1 Peripheral Controller

With the addition of the accelerator described in 4.1 to the system, the direct connection existent between the Scalar Core can and the L1 Data cache can no longer be supported. This is the case because both the core and accelerator need to perform requests to the cache, so the need of a controller that arbitrates requests of these two components arises.

To this end, the Peripheral Controller was implemented and added to the system. The controller is composed of a finite state machine that implements the model illustrated in figure 4.18. When in idle mode, the controller waits for a valid memory request from the Scalar Core or the Vector Accelerator, issued by the respective memory stages of each core. Two transitions are then possible, depending on the valid requests received. If only a scalar request is detected, then the controller transitions into the SCALAR_REQUEST. Similarly, if only a vector request is detected, then state transitions to VECTOR_REQUEST. However, if both a scalar request and a vector request, the controller gives priority to the scalar request. This priority order ensures the minimum impact in the program flow, since the Scalar Core is the block responsible for fetching instructions from memory. Since the system provides support for memory mapped peripherals, the controller includes an address decoder that controls if the Data Cache or a given peripheral are to be interfaced with in the request stages.

The SCALAR_REQUEST state issues the required request to either the data cache or the memory mapped peripheral associated with the address requested and waits for the valid signal to be set. The requests have variable latency, since one can have single cycle response peripherals, multi-cycle response peripherals, cache hits or cache misses and so this state can last for several clock cycles.

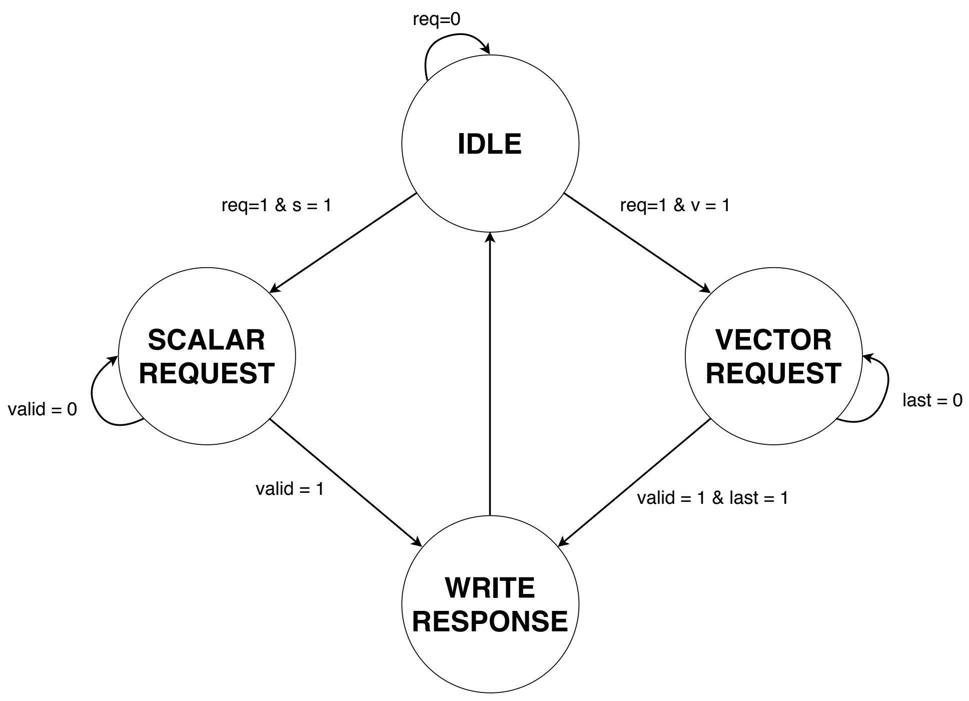


Figure 4.18: Peripheral Controller Finite State Machine Model.

Analogous to the SCALAR_REQUEST state, the VECTOR_REQUEST state also interfaces with either the data cache or a memory mapped peripheral, however a single vector request can be comprised of several scalar requests. Consequently, with each successful request made, i.e when the valid flag is set, the address value is incremented according to the element width value of the transaction until the number of requests made equals the size of the original vector request, in number of elements.

After the request is successfully executed and the resulting data is obtained, the state transitions to the WRITE_RESPONSE state. In this state, the controller uses the Scalar Core or Vector Accelerator response channel, depending on the request fulfilled, to write the result of the request. The result is comprised of a valid bit and data bits, however the data bits are only written to if the request is a read operation.

4.3.2 Scalar Core

With the addition of several new hardware blocks to the system, mainly the Streaming Engine and the Vector Accelerator, it became imperative that some minor changes to the Scalar Core needed to occur, owing to the fact that the Scalar Core acts as programmer of these new hardware blocks. Consequently, instruction detection for UVE instructions was added to the existing decoder, and FIFO buffers were added to provide communication channels between the Scalar Core and the other hardware blocks. All streaming instructions are fully decoded in the Scalar Core and the relevant control and data signals, detailed in 4.2.1 are provided to the Streaming Engine. If an instruction is an arithmetic, logic or memory SIMD instruction, then the instruction word is provided, without being decoded, to the Vector Accelerator for further decoding and execution.

Further changes were made with the objective of making the core more complete. Therefore the existing integrated BRAM was replaced by an Instruction Cache, and full support for the RISCV 32 bit Floating Point Extension was added.

Instruction Cache

While the use of an integrated BRAM is convenient, due to the fact that the processor can access the instruction memory directly without the use of complex communication protocols, it does come with disadvantages. Firstly, it directly associates the amount of valid program memory to the amount of BRAM memory available, which is typically much lower than the amount of RAM available to a system. Secondly, it creates a division between program memory and data memory, since one is stored in a BRAM, accessible only to the core, and the other is stored in system RAM. Therefore, to standardize the system, the integrated BRAM was removed and an instruction cache was added, with the goal of reducing memory access latency when fetching instructions.

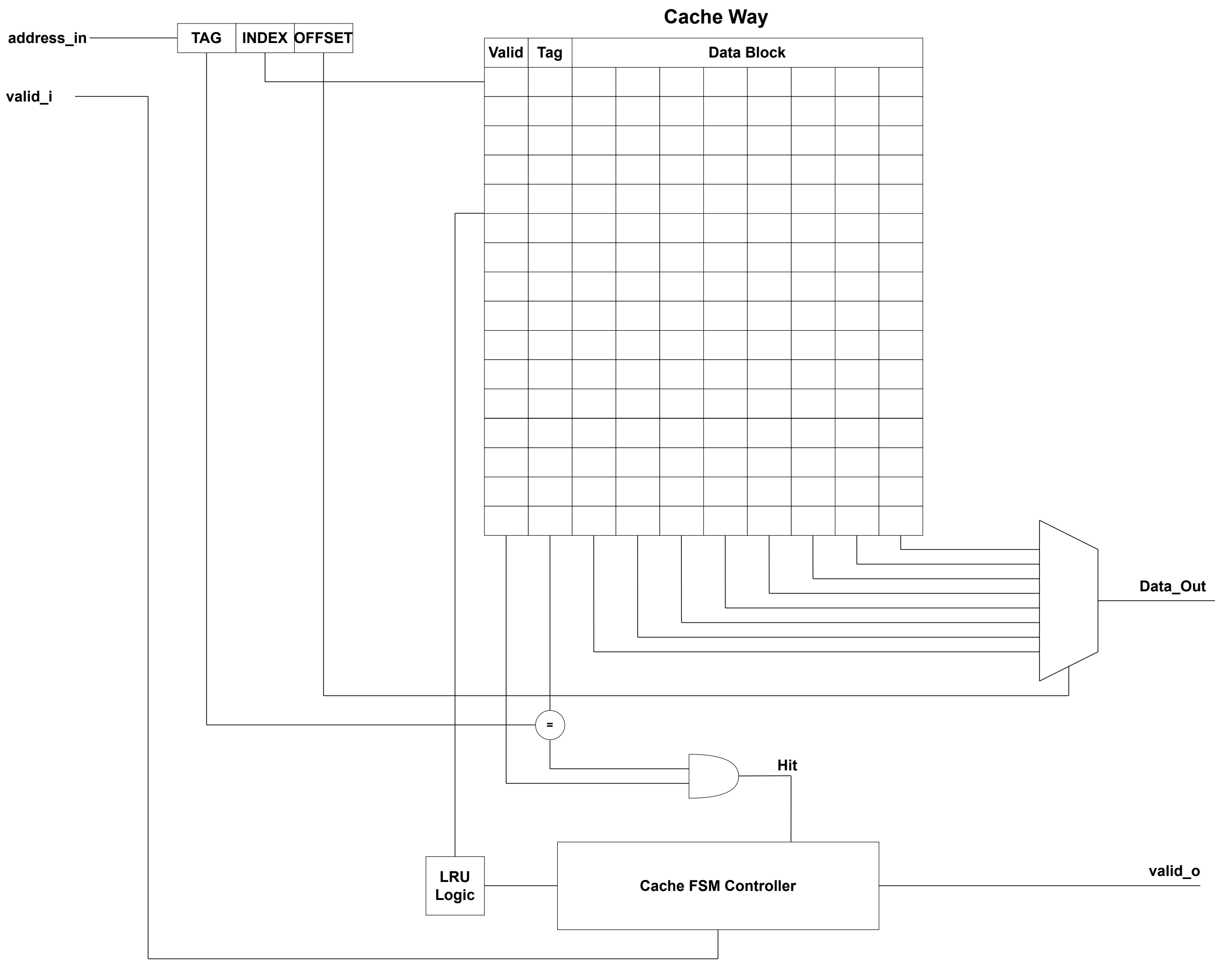


Figure 4.19: Instruction Cache Simplified Diagram.

The Instruction Cache developed and integrated with the system implements a 2-way set associative access mechanism and supports an input address of 16 bits. An example of the simplified cache diagram with 16 cache lines is illustrated in figure 4.19. Being that the cache stores only instructions and no data values from the program, then only a write interface with main memory and a read interface with the

Scalar Core are needed. The read interface of the instruction cache validates both the request and the response with two valid signal bits, one for each direction of communication. Since the supported bus that connects the system to main memory is 256 bits wide, then the chosen block size, i.e data size of a cache line in bits, was 256 bits, analogous to the data cache, detailed in section 3.2. The number of lines is programmable, and consequently the number of index bits depends on the number of lines chosen. To arbitrate between cache lines, the Least Recently Used policy was used. The access to main memory is performed using the AXI protocol, similarly to the data cache, however in this case only read support is necessary since the instruction cache does not write to main memory. The read latency of the instruction cache is 1 clock cycle in case of a cache hit and may vary in case of a cache miss, due to the interface with the main memory and its respective access latency.

Floating Point Support

While this work implemented only the integer subset of instructions from the UVE extension, further work on the proposed system may need to include support for floating point SIMD instructions. However, since the accelerator and the core work in unison with each other, if the accelerator is to support floating point instructions, then the core should also provide support for the same data type. Consequently, support for all the 32 bit RISCV Floating Point Extension instructions was added.

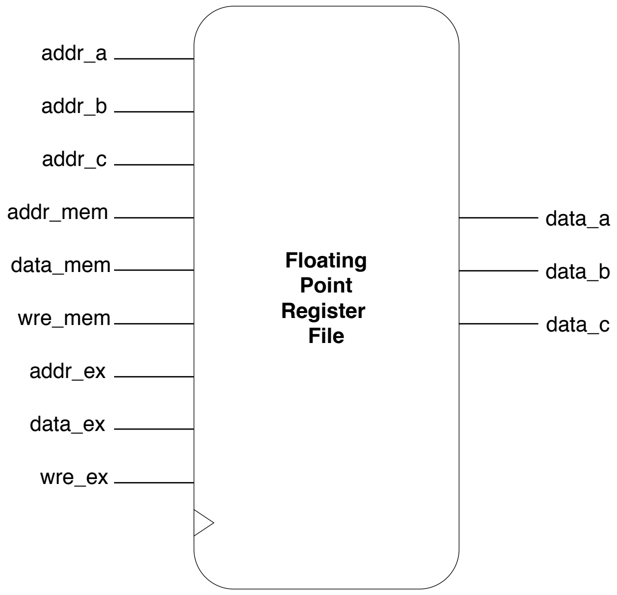


Figure 4.20: Floating Point Register File I/O.

Firstly the decoder unit present in the decode stage was modified to support decoding of the RISCV32F instructions. To support these instructions, the addition of a Floating Point Register File was needed. The number of floating point registers follows the RISC-V standard, i.e there are 32 registers encoded in the instructions using 5 bits. Since the floating point instructions support up to three operands, the register file implements a read interface with three input addresses and three outputs. As with the already existing Integer Register File, the Floating Point Register File implements a write interface with two channels. One channel is used for data incoming from the Execute stage, while the other channel is used for data incoming from the Memory stage. If both channels attempt to write to the same register on the same clock cycle, priority is given to the execute channel, although this should never happen since the core implements a Dependency Handler Unit that solves structural hazards of this type. The I/O of the Floating Point Register File can be observed in figure 4.20. Since a new register file was added, the Dependency Handler Unit, i.e the Scoreboard, was updated to support another set of 32 registers. This was done by extending the table of 3.3 to feature the floating point registers, while maintaining the same hazard calculation logic.

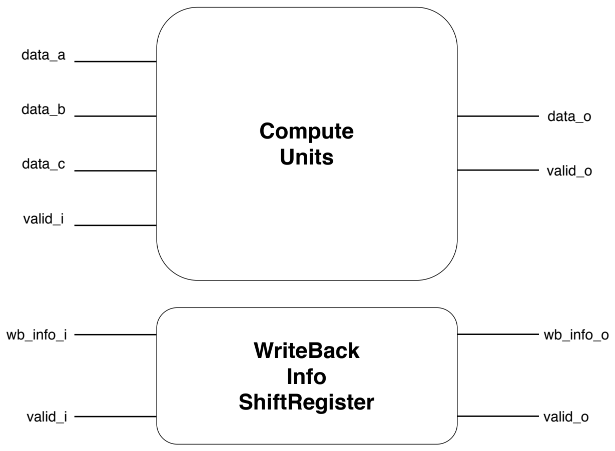


Figure 4.21: Floating Point Unit Schematic.

As can be observed in figure 3.2, the Execute stage of the Scalar Core benefited from the addition of a Floating Point Compute Unit. The FPU is composed of several multi-cycle pipeline compute blocks, generated using the Xilinx Floating Point IP Generator [26]. These compute blocks implement the arithmetic operations needed to support the RV32F extension, ranging from addition/subtraction to conversion and from floating point to signed/unsigned integer. Table 4.6 reflects the operations that are implemented using multi-cycle units and the respective latency of each unit. Since the FPU supports multi-cycle units with different latencies, it needs a mechanism to keep track of write back data, such as destination register and register type, for each ongoing computation in the pipeline of the FPU. Consequently, a shift register as coupled with the Floating Point Unit. Each time an operation is started in the FPU, the write back data is saved in the shift register, according to its latency, i.e if an operation takes 7 cycles to complete, then the write back data will be saved in the seventh register of the shift register, guaranteeing that when the operation finished, the write back data of the operation will be present in the output of the shift register. The simplified schematic of the FPU including the shift register is illustrated in figure 4.21. One should not that the FPU should be used in conjunction with a Dependency Handler Unit to prevent structural hazards, such as two operations finishing at the same time even though they were issued during different clock cycles, due to differences in latency.

Operation	Latency(in clock cycles)
fadd/fsub	11
fmul	8
fdiv	28
fsqrt	28
ftoi_cvt	5
itof_cvt	5
fcmp	2
fmadd/fmsub	16

Table 4.6: Multi-cycle floating point operations.

4.4 Summary

In this chapter, the modifications and additions to the system to allow for data streaming and SIMD support were presented. The first section detailed the architecture of the Vector Accelerator added to the system, that provides SIMD support and includes an interface used to communicate with the Streaming Engine. The second section section explained in detail the components developed in order to provide a functional streaming solution to the system, the Streaming Engine, by presenting the architecture of its several internal blocks. Finally, the third section detailed the further additions and modifications to the system, mainly the addition of the Peripheral Controller, that arbitrates communication to the data cache and peripherals, the Instruction Cache and the modification of the Scalar Core to allow floating point support.

Chapter 5

Development Workflow and Benchmarks

5.1 Software Tools

The software tools comprise the set of necessary software needed to develop and deploy an application to the proposed hardware in this work, mainly code compilation and support for the board used for development, described in the following sections. More specifically, the tools used are a modified version of the RISC-V GCC Compiler, a set of board specific files, i.e the Board Support Package and finally a linker script that reflects the memory mapping of the implemented system.

5.1.1 GCC Compiler

The RISC-V International Organization provides an official GNU toolchain [13] distribution with support for the GCC compiler, the GDB debugger and the newlib and glibc libraries, in order to support both bare metal applications and applications targeting the Linux operating system. The RISCV branch of the GCC compiler supports the official RISCV extensions present in the RISCV ISA Specifications. To provide the compiler with information about the hardware support, one needs to use the compiler options *-march* and *-mabi*. The *-march* option supplies the compiler with information about the base architecture and supported extensions. The *-mabi* option specifies the calling convention for integer and floating point values. For example, if *-march=rv32imf* and *-mabi=ilp32f*, then the target system supports the 32 bit base RISCV ISA plus the Multiply/Divide and Single Precision Floating Point extensions, *'int'*, *'long'* and pointers are 32 bits wide and both integer and 32 bit floating point values can be passed as registers. The calling convention might not always support all the data types that the architecture itself supports, for example the configuration using *-march=rv64ifd* and *-mabi=lp64f* supports single and double precision floating point value extensions, however only single precision floating point values and integer values are allowed to be passed as registers.

5.1.2 Unlimited Vector Extension Support

Due to the fact that the Unlimited Vector Extension is not part of the official RISC-V extensions, further changes to the GNU Toolchain, in particular to the GCC Compiler are needed. The changes focused on adding the new architectural state, i.e the new supported registers, and the new instructions to the compiler, and are explained in detail in João Domingos's Thesis[4]. This works makes use of this modified version of the GCC compiler to develop software applications for the developed system.

Even though the compiler was modified to support the new instructions and architectural state, it does not perform automatic code vectorization. Consequently, to use the new instructions the user should add assembly directives to the source code. A couple examples of how to add UVE instructions to C code are depicted in listing 5.1. The first example adds a stream configuration instruction that configures a simple one dimensional load stream associated with register *u1*, using scalar registers *s2*, *s3* and *s4*. The second example adds a SIMD add instruction that consumes data from the stream associated with register *u1*, adds that value to the value of register *u2* and stores the result in register *u3*. Since the changes to support the UVE extension were mainly extending the support for the new instructions and architectural state needed, the compilation of UVE code is analogous to compilation of RISC-V code since no new compiler flags are needed. An example compilation command of a C file containing UVE code to an ELF file is illustrated in listing 5.2

Listing 5.1: Example UVE Code insertion.

```
//add a simple 1D load stream assigned to register u1
asm volatile("ss.ld.w u1, s2, s3, s4 \t\n");
//add a UVE signed add instruction u1 = u2 + u3 predicated by p0
asm volatile("so.a.add.sg u3, u2, u1, p0 \t\n");
```

Listing 5.2: Example UVE Code compilation.

```
riscv32-unknown-elf-gcc -march=rv32imf -mabi=ilp32 -std=gnu11 -Wall -O0 -mno-relax
    -Iinclude -fno-common -o main.elf main.c -T ls.lds -nostartfiles -static
    -LRISC-V_UVE_COMPILER/lib/gcc/riscv32-unknown-elf/8.2.0 -lgcc
```

5.1.3 Board Support Package and Linker Script

Due to the fact that the code used in this work targets a system without an operating system, a Board Support Package, included in Appendix C, is used to provide some necessary start and end routines that are appended to the developed code. The file responsible for the starting routines is the entry.S file. The file defines a function called *_start*, within the init section of the program. This function is then used to initialize the global pointer, the stack pointer and to clear the bss section so that the main function of the developed program can be called and user part of the code can be executed correctly.

With the inclusion of the *_start* routine, the correct execution of the user code is guaranteed due to the initialization of the relevant pointers used by the architecture, however the code will run endlessly due to the lack of end of execution control. To solve this issue, the Board Support Package provides an end routine, called after the user code has been executed. This routine, called *__end*, appends three NOP instructions and a jump instruction with register *r0* used for all the register operands, and an immediate value of 0. The three NOP instructions are used to provide time for the previous instructions to complete execution, and then the jump instruction to address 0 is used as an end of execution flag. Consequently the Scalar Core provides detection capability of this instruction and sets the end execution flag of the processor a jump instruction is detected with 0 as all the operands. Using this method, the system prevents endless execution of code by having a landmark for the end of execution of user code.

The Board Support Package further includes a linker scripted, also represented in Appendix C, that has the function of providing information about the memory layout of the target system to the linking process that occurs after program compilation. The memory structure of the developed system, represented in table 5.1, is mirrored in the linker script using the commands *rom (x!rw) : ORIGIN = 0x00000000, LENGTH = 16k* and *ram (wxa—ri) : ORIGIN = 0x00004000, LENGTH = 16k*. The first 16k bytes of memory are read only and are used to store the instruction data of the program, while the following 16k bytes are used as random access memory to store and load program data. The random access memory part of the memory structure is further divided into two sections, the heap and the stack. The heap covers the first 12k bytes of the ram while the last 4k bytes are covered by the stack. If changes to the memory layout of the system are needed, then one should also reflect the changes done in the linker script to guarantee memory coherency when running user code.

Memory Section	Start Address	End Address
Instruction	0x0000_0000	0x0000_3FFF
Data	0x0000_4000	0x0000_7FFF

Table 5.1: System memory layout.

5.2 Benchmarks

To evaluate the performance of the developed system detailed in chapter 4, a set of benchmarks were chosen and and implemented in both base RISC-V and RISC-V + UVE code. The benchmarks chosen share the characteristic of using loop code to form memory patterns, providing relevant memory accesses to test the developed streaming engine, while also being commonly used in user code, which provides relevancy to the obtained results. The code was developed using the C Programming Language, with support of the extended ASM[6] mechanism that allows reading and writing of C variables using assembly code.

5.2.1 Memory Copy

The memory copy algorithm is the simplest form of memory access algorithm and is regularly used in user code to copy data from one array to another. The memory copy kernel used as benchmark, implemented in C, can be observed in listing 5.3. It consists of a single for loop, which iterates over a subset of an array *a*, or the whole array depending on the size of the iterator, *sizeN*, and copies its values to array *c*. The number of accesses to main memory is directly proportional to the size of the loop, and due to the loop only copying the values of the array, the execution time of the kernel is memory bound.

Listing 5.3: Memory Copy kernel.

```
for (j=0; j<sizeN; j++)
    c[j] = a[j];
```

To adapt the kernel to use the UVE extension stream and compute instructions, one must first analyse the memory pattern defined by the kernel. In this case, the for loop defines two one dimensional access patterns to main memory, one to load the values of array *a* and another to store the values in array *c*. Both patterns access main memory with stride equal to 1, and size equal to *sizeN*, however they differ in the base address used, since array *a* and *c* have different base addresses. With this knowledge, the base descriptor tuples needed to define both streams can be easily created using the UVE stream instructions, as can be seen in listing 5.4.

Listing 5.4: Memory Copy UVE memory pattern configuration.

```
asm volatile(
            /*offset, size, stride*/
    "ss.ld.w u1, %[a], %[sn], %[one] \t\n" //a[i]
    "ss.st.w u30, %[c], %[sn], %[one] \t\n" //c[i]
    :: [a]"r"(a), [c]"r"(c),
        [sn]"r"(sizeN), [one]"r"(1)
);
```

With the configuration done in the preamble of the loop, the version of the memory copy kernel adapted to use the UVE extension features a loop with only two instructions, represented in listing 5.5. The first instruction, *so.v.mv*, is a SIMD move instruction that moves the values of vector *u1*, which is associated with the load stream, to vector *u30*, associated with the store stream. Consequently the move instruction both consumes and produces data to the necessary streams, with no need of further instructions. The second instruction, *so.b.nc*, performs a branch to the move instruction if the stream associated with register *u1* is yet to finish. The combination of this two instructions guarantees that all data is consumed processed and produced to the relevant streams.

Listing 5.5: Memory Copy UVE loop code.

```
asm volatile(
    "1: \t\n"
      "so.v.mv u30,u1 ,p0 \n\t" // c[] = a[]
    "so.b.nc  u1, 1b \n\t"
);
```

5.2.2 Scaled Vector Addition

The Scaled Vector addition kernel, generally ”z = a*x + y”, where *x*,*y* and z are vectors and *a* is a scalar value, is common among various scientific computation tasks. Particularly, if one sets the destination vector *z* to be the same array as *y*, i.e ”y = a*x + y”, we get the family of algorithms called ”SAXPY” or ”IAXPY”, standing for ”Single Precision A times X plus Y” and ”Integer A times X plus Y”, depending on the type of data used. The implementation of the kernel in C is represented in listing 5.6. The memory pattern defined by the for loop is identical to the memory copy kernel, however processing is done using vectors *x*, *y* and the scalar value before it is copied to the destination. Since this kernel contains multiplication of a scalar with an array, the execution time can be compute bound if the latency of the multiplication instruction is high or memory bound if not.

Listing 5.6: Scaled vector addition kernel.

```
for (j=0; j<sizeN; j++)
    z[j] = y[j]+scalar*x[j];
```

The streaming instructions used to configure the kernel data flow are similar to the memory copy example explained before, since both kernels define the same type of one dimensional memory access pattern, varying only in the base address for each array. Consequently, the configuration of the streams for the scaled vector addition kernel is analogous to the memory copy kernel configuration, except now three streams need to be configured, two load streams associated with arrays *x* and *y*, and a store stream associated with array *z*.

Listing 5.7: Scaled vector addition UVE memory pattern configuration.

```
asm volatile(
    "ss.ld.w u1,  %[y], %[sn], %[one] \t\n" //y[i]
    "ss.ld.w u2,  %[x], %[sn], %[one] \t\n" //x[i]
    "ss.st.w u30, %[z], %[sn], %[one] \t\n" //a[i]
    "so.v.dp.w  u10, %[sc], p0 \t\n"
    :: [x]"r"(x), [z]"r"(z), [y]"r"(y),
        [sc]"r"(scalar),
        [sn]"r"(sizeN), [one]"r"(1)
);
```

The reduction of loop instructions due to the use of the UVE extension contemplated in the memory copy kernel is also present in the implementation of the scaled vector add kernel, illustrated in listing 5.8. In this case, three instructions are needed to perform the desired computation. The data loaded from the load stream associated with vector *u2* is multiplied by the scalar value which has been correctly duplicated to fill the whole vector length of *u10* in the loop preamble, using instructions *so.a.dp.w* and *so.a.mul.sg*. subsequently, the resulting vector *u20* is summed with vector *u1* which is associated with another load stream, with the use of the *so.a.add.sg* instruction. The output value of the sum produces the data needed for the store stream due to the fact that it writes the value to register *u30*, which is associated with the stream. The *sg* prefix in the arithmetic instructions is used since the data being processed is of the *int* type. At last, the loop flow is guaranteed by the use of a branch instruction that performs the branch if the stream associated with *u1* has not finished.

Listing 5.8: Scaled vector addition UVE loop code.

```
asm volatile(
    "1: \t\n"
      "so.a.mul.sg u20,u2 ,u10 ,p0 \n\t" // intermediate = scalar * x[]
      "so.a.add.sg u30,u1 ,u20,p0 \n\t" // z[] = y[] + intermediate
    "so.b.nc  u1, 1b \n\t"
);
```

5.2.3 Matrix Vector Multiplication

So far, the benchmarks chosen defined a simple one dimensional access to main memory, which is useful due to the simplicity of the pattern, however several algorithms in computer science use more complex memory patterns, and offer more opportunities to explore the benefit of the use of a streaming engine. Consequently, a variant of the matrix vector multiplication algorithm was chosen as a benchmark for evaluation of streams with more than one dimension. The kernel used as benchmark implemented in C is represented in listing 5.9. Two for loops are used by the kernel, the inner loop controls access to the matrix elements in each row and the accesses to the vectors, while the outer loop updates the row of the matrix to be accessed each iteration. The row element accessed from matrix *A* each inner loop cycle is multiplied with the respective element of vector *y_1* and accumulated in the *i'th* position of vector *x*, which equates to the dot product result of the *i'th* row vector of *A* with the vector *y_1*. The outer loop repeats this process *sizeN* times resulting in *sizeN* dot product results which are saved in the corresponding elements of vector *x*, which is the result of the multiplication of matrix *A* with vector *y_1*.

Listing 5.9: Matrix Vector Multiplication kernel.

```
for (i = 0; i < sizeN; i++)
  for (j = 0; j < sizeN; j++)
    x1[i] = x1[i] + A[i*sizeN+j] * y_1[j];
```

The adaptation of this benchmark to use UVE extension instructions introduces the use of the more flexible multi-dimensional stream configuration instructions, illustrated in listing 5.10, in addition to the one dimensional counterparts used so far. By analysing the C kernel, one quickly notices that matrix *A* is a square matrix, due to the fact that both for loops have the same size, *sn*. Consequently, two base descriptors need to be defined, one for the row elements of the matrix, and one to simulate the increment of the row pointer. The instruction *ss.sta.ld.w* starts the configuration of the stream that loads matrix *A* and simultaneously creates the first descriptor using the base address of *A*, size equal to *sn* and stride of 1. The configuration then follows with the *ss.cfg.vec* instruction that signals the stream in question that the vectors obtained from the stream must follow the topology of the memory access, which in this case means that each vector corresponds to a row of matrix A, if the vector length is large enough to allow it. The configuration of the stream then ends with the *ss.end* instruction that appends the final descriptor with base address of 0, size equal to *sn* and stride equal to *sn*, effectively updating the stream to point to the beginning of the next row of the matrix.

The stream that loads the vector *y_1*, despite the fact that it is used to load a vector which follows a one dimensional access pattern, benefits of the use of two dimensions. This is the case because *y_1* is reused for every outer loop cycle, consequently two descriptors are defined. The first descriptor is a simple one dimensional access pattern using the base address of *y_1*, size *sn* and stride 1. The second descriptor uses the combination of setting both the base address and stride to 0, resulting in a repeating pattern of the previous descriptor, based on the size of the second descriptor, which in this case is *sn* so the vector *y_1* can be loaded exactly *sn* times.

The vector that holds the result of the multiplication, vector *x*, is used only partially each iteration, i.e only one element of *x* is used for each outer loop of the algorithm. Consequently, the load stream associated with *x* uses the first descriptor to load a single element of *x* *sn* times, by setting the base address to the base address of *x*, the size to *sn* and stride to 0. The second descriptor uses a base address of 0, size equal to *sn* and stride 1 to increment the address pointer so that the next item of *x* is loaded by the first descriptor. Additionally, a store stream associated with vector *x* is also created, using a one dimensional access pattern, to store the results of the matrix vector multiplication.

Listing 5.10: Matrix Vector Multiplication UVE memory pattern configuration.

```
asm volatile(
    // A stream load
    "ss.sta.ld.w          u1, %[src1], %[sn], %[one] \t\n" //row vector
    "ss.cfg.vec           u1 \t\n"
    "ss.end               u1, zero, %[sn], %[sn] \t\n"  //slide vertically stride N
    // y_1 stream load
    "ss.sta.ld.w          u2, %[src3], %[sn], %[one] \t\n" //y vector
    "ss.end               u2, zero, %[sn], zero \t\n"   // repeat: 'sn' times
    // x_1 stream load
    "ss.sta.ld.w          u3, %[src2], %[sn], zero \t\n"   //x[i] element
    "ss.end               u3, zero, %[sn], %[one] \t\n" //slide horizontally by 1
    // x_1 stream store
    "ss.st.w          u4, %[src2], %[sn], %[one] \t\n" //x vector
    :
    : [src1] "r"(A), [src3] "r"(y_1), [src2] "r"(x_1),
    [sn] "r"(sizeN), [one] "r" (1)
);
```

Since the C kernel contemplates two for loops, the equivalent UVE loop features features two loops controlled by two different stream branch instructions, which branch to the loop tags ”fLoop1” and

”jLoop1”, as can be observed in listing 5.11. The inner loop computes the term by term multiplication and addition of the row vector of matrix *A* with vector *y_1* using the ”so.a.mul.sg” and ”so.a.add.sg” instructions, which combine to act as a multiply and accumulate instruction on vector register *u6*. In the outer loop, the result from the inner loop is further reduced to obtain the scalar value result of the dot product and is then added to the *i'th* element of the *x* vector, using instructions ”so.a.adde.sg” and ”so.a.add.sg”. The inner loop repeats until the first dimension of the stream associated with *u1* finishes, guaranteeing that all the row elements of the current row of matrix *A* are processed, while the outer loop repeats until the stream is finished, indicating that all elements of matrix *A* have been consumed.

Listing 5.11: Matrix Vector Multiplication UVE loop code.

```
asm volatile(
    "fLoop1: \t\n"
    "so.v.dp.w u6, zero, p0\n\t"//initialize vector to all zeros

    "jLoop1: \t\n"
      "so.a.mul.sg u5, u1, u2, p0 \n\t" // inner prod A[i][] * y[]
      "so.a.add.sg  u6, u5, u6, p0 \n\t"
      "so.b.ndc.1 u1, jloop1 \n\t"//not dimension 1 complete

    "so.v.mv      u7, u3, p0     \n\t" // load x[i]
    "so.a.adde.sg u6, u6, p0     \n\t" // reduce vector
    "so.a.add.sg  u4, u6, u7, p0 \n\t" // x = red + x
    "so.b.nc  u1, fLoop1 \n\t" //not stream complete
);
```

5.3 Summary

This chapter focused on the presentation of both the tools necessary to develop software for the developed system and the benchmarks used to evaluate its performance. The first section briefly described how certain compilation flags inform the compiler of the extensions supported by the target architecture and a further description of how to introduce UVE code into a C program, using assembly directives supported by GCC, was discussed. The second section focused on the detailed description of the set of benchmarks used to evaluate the performance of the developed system. The base C version of the benchmarks was initially introduced and then a vectorized version using UVE was later discussed, including the description of the assembly instructions used.

Chapter 6

Implementation and Benchmark Results

In order to analyse both the feasibility and the performance of the system proposed in chapter 4, the system was submitted both to an implementation process and tested using the set of benchmarks presented in the last chapter. Consequently, this chapter will provide the reader with the results of the implementation process for the discussed architecture, with a vector length of 256 bits, and the benchmarks, while providing additional discussion and analysis of the benchmark results.

6.1 Implementation Results

The implementation process of the proposed architecture was carried out using the Xilinx Vivado tool, version 2019.1 in a Linux environment. In order to implement a design, a target board must be available, consequently the board used to implement the architecture of this work was the Xilinx Virtex Ultrascale+ VCU1525. This board is mainly targeted for computationally intensive tasks given the amount of resources offered, including available communication interfaces, such as an UART interface via USB, four DDR4 DIMM slots, each with 16GB memories, and both Gen3 x16 and Gen4 x8 PCIe interfaces.

6.1.1 Area Analysis

To analyse the area requirements of the system, several post implementation leaf cells were highlighted to provide a visual representation of the implementation placement result of the system represented in 4.1 and an utilization report was run over the implementation result to obtain resource utilization values for each of the components of the system, namely the Scalar Core (including the Instruction Cache), Peripheral Controller and Cache subsystem, Vector Accelerator and Streaming Engine. The placement result of the implementation is illustrated in figure 6.1. The leaf cells highlighted in purple correspond to the Scalar Core implementation and use the least amount of resources of the four main

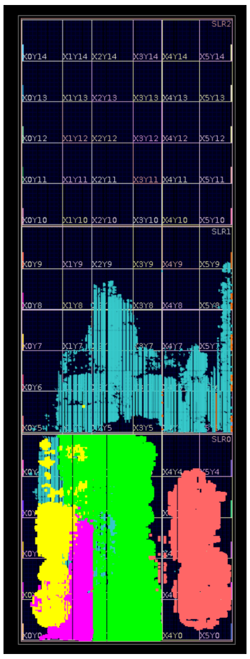


Figure 6.1: Implementation placement result with highlighted components.

blocks of the system. Closely connected to the Scalar Core is the Peripheral Controller and Data cache subsystem (highlighted in yellow). The Vector Accelerator also has leaf cells closely connected with the Peripheral Controller and Data Cache subsystem, and is the component that occupies the largest amount of area. The larger area occupancy of the Vector Accelerator is due to the fact that the data bus of the accelerator in the implementation is 256 bits, consequently the amount of routing resources needed for this component is significantly higher than the other components of the system. The last system component, highlighted in orange, is the Streaming Engine, which is closely connected with the Vector Accelerator due to the shared interface used by the two components to exchange stream data. The remaining leaf cells, which are highlighted in the default blue color, correspond to the XDMA interface that controls data transfers using the PCIe connector of the FPGA, the MIG that acts as a memory controller and bridge to the RAM installed in the DIMM slots of the FPGA, and the several AXI interconnect logic needed to arbitrate AXI connections between the different components of the system. All these extra components that are needed for the implementation of the original system (figure 4.1) in the FPGA use roughly one third of the resources of the total used by the system, mainly due to the fact that the XDMA is a complex device that requires high resource usage. The exact resource values used for each component, along with the percentage resource usage of the system, can be observed in table 6.1.

Component	LUT	LUTRAM	FF	BRAM	DSP
Scalar Core	14275	226	19465	0	10
Peripheral Controller+Data Cache	20693	0	10051	1	0
Vector Accelerator	84506	10000	22285	0	0
Streaming Engine	35262	1152	39933	0	0
XDMA	72843	5837	69603	124	0
MIG	18282	1729	19299	25	1
AXI Interconnect	18743	5922	29004	0	0
Total System Resource Usage	264604	24866	209640	150	11
Total FPGA Resources	1182240	591840	2364480	2160	6840
% of Resources Used	22.38	4.20	8.87	6.94	0.16

Table 6.1: FPGA resource usage.

6.1.2 Timing Analysis

The implemented system architecture is composed of three main clock domains required for the four main components of the system and a few other clocks, mainly the PCIe reference clock and the MIG clock, mandatory by the design due to the use of the XDMA and MIG controllers. The Scalar Core component, which includes the instruction cache of the system, uses one clock domain, that shall be named as clock domain 1. The subsystem that contains both the Peripheral Controller and Data Cache blocks use clock domain 2 and at last both the Vector Accelerator and the Streaming Engine use clock domain 3. Using this approach, the system has three operating frequencies, which increases the performance flexibility of the system since components of a clock domain don't affect the critical path of the other clock domains, however this method comes with the cost of communication latency between blocks of different clock domains due to clock domain crossing logic.

To evaluate the maximum operating frequency of the different clock domains of the system, two different scenarios were evaluated. The first scenario was the evaluation of the operating frequency of the different clock domains isolated, to remove routing and other implementation bottlenecks that may affect the critical path of the components. The second scenario was the evaluation of the operating frequencies of the different clock domains with the whole system implemented, including the external blocks such as the XDMA and MIG. The results from both scenarios are represented in table 6.2. Focusing on the first scenario, the clock domain with the highest operating frequency is clock domain 2, which corresponds to the Peripheral Controller and Data Cache subsystem, followed by clock domain 3, which includes the Vector Accelerator and Streaming Engine components, and lastly clock domain 1, which includes the Scalar Core and the Instruction Cache. The additions made to the Scalar Core to allow the control of the other blocks and support for floating point operations and the fact that both the Vector Accelerator and Streaming Engine are components designed to process data in parallel, at the cost of area but with the benefit of shorter critical paths, explain why, when isolated, the Scalar Core + Instruction Cache suffer from being the slowest clock domain. On the other hand, when the focus is shifted to the whole system operating frequency, the values significantly change. Two main factors are directly responsible for the changes seen between the clock frequencies of the two scenarios, one is the increased use of routing resources when implementing the system as a whole instead of only selected components, and the other is the dependencies between components of different clock domains, mainly communication interfaces, that limit the placement of such components. The Peripheral Controller and Data Cache subsystem interfaces with both the Scalar Core and the Vector Accelerator and consequently these components were placed tightly coupled in the FPGA by the implementation algorithm, as can be seen in figure 6.1, so that the critical path is not largely affected by the interface connections. Another component that has a communication dependency is the Streaming Engine, since it shares an intra-clock interface with the Vector Accelerator and an inter-clock interface with the Scalar Core. However, due to the area occupied by the Streaming Engine and Vector Accelerator, there is no suitable placement spot for the Streaming Engine for which it is both near the Vector Accelerator and the Scalar Core. Consequently, the implementation algorithm placed the Streaming Engine coupled with the Vector Accelerator, reducing the impact of the intra-clock interface, however the routing formed by the inter-clock interface to the Scalar Core is long, affecting the operating frequency of both clock domains.

	Operating Frequency isolated (Mhz)	Operating Frequency in system (Mhz)
Clock Domain 1	170	106
Clock Domain 2	370	115
Clock Domain 3	185	94

Table 6.2: Maximum operating frequency results. Clock domain 1 correspond to the Scalar Core and Instruction Cache, clock domain 2 to the Peripheral Controller and Data Cache and clock domain 3 to Streaming Engine and Vector Accelerator. Large operating frequency drops are due to routing issues.

6.1.3 Power Analysis

The Vivado tool provides the user with a post-implementation routine that is run on the placed design and estimates the power consumption of the system as a whole and its components, using constraint and simulation files. Using this feature, the power consumption of the developed system was analysed and the results are illustrated, per component, in table 6.3. As can be seen, 96% of the power required by the system is focused on the external blocks, mainly the XDMA controller, MIG controller and the AXI Interconnect IP. Of the developed components, the Streaming Engine and the Vector Accelerator require the most power. This is due to the architecture of these hardware blocks, mainly the fact that they explore hardware parallelism at the cost of higher resource usage, with the arithmetic lanes of execution on the Vector Accelerator and the parallel stream processing blocks in the Streaming Engine. In systems with low power requirement, the number of parallel processing blocks used by the Streaming Engine can be reduced to diminish its power consumption, with minimal to no cost on performance, depending on the number of concurrent streams to process. The total dynamic power requirement of the system is obtained by summing the entries of table 6.3 and equates to $10.061W$.

Component	Power(W)	% of total power of the system
Scalar Core	0.084	1
Peripheral Controller+Data Cache	0.074	1
Vector Accelerator	0.208	2
Streaming Engine	0.352	3
XDMA	7.037	70
MIG	1.472	15
AXI Interconnect	0.834	8

Table 6.3: Dynamic power requirement per component.

6.2 Benchmark Results

This section will go over the results obtained by running the benchmarks explained in detail in chapter 5. The results were obtained by simulating the system of figure 4.1 plus the memory controller, in order to obtain accurate results with respect to the latency of memory accesses, in the Vivado simulation environment. The clock frequencies of the different clock domains are the same as obtained in the implementation process, meaning that clock domain 1 runs at 106 Mhz, clock domain 2 at 115 Mhz and clock domain 3 at 94 Mhz, while the vector length chosen for the evaluation was 256 bits, meaning that one vector instruction can process up to eight 32 bit values concurrently. One should not that, when comparing the benchmark results presented below with the respective counterparts in Domingos thesis [4], the values differ in magnitude due to the fact that this work uses an in-order processor, contrasting with the out-of-order simulated model used in [4]. In particular, an out-of-order core is able to partially hide thememory access latency by allowing non-dependent instructions to continue executing in parallel, even if speculatively. Moreover, typical solutions employ complex memory hierarchies which allow issuing multiple load requests, which are served in parallel, and do not necessarely stall the processor when one results in a cache miss. In contrast, the base in-order core follows a more simple execution model, which can only serve one load request at a time, and stalls the processor whenever a memory-dependent instruction enters the pipeline. Furthermore, the out-of-order model supported several cache levels and different prefetch mechanisms, which further increases the performance of the base model without streaming support, contrasting to this work's in-order model with support for only 1 level of data cache and no prefetching.

6.2.1 Memory Copy

The Memory Copy benchmark provides an evaluation of a one dimensional, unitary stride memory access pattern on the developed architecture. Consequently the benchmark was evaluated using four different loop size values, where the first loop size, N = 8, was chosen because it fills exactly one vector when executing the vectorized code, and the last loop size, N = 256, was chosen because it fills the entire data cache of the system. The speedup values, represented in figure 6.2, were obtained by comparing the execution times of the RISC-V kernel (baseline) with the UVE kernel. When executing the code with the lowest value of N, a speedup or around 8 times is obtained when comparing UVE code with the baseline RISC-V code. In this case, since the number of memory accesses is low, the streaming

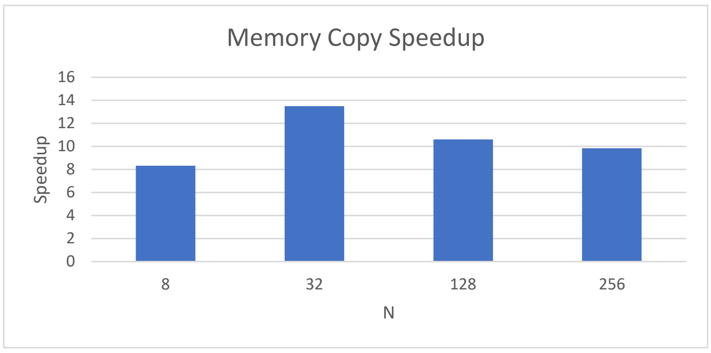


Figure 6.2: Memory Copy kernel speedup.

mechanisms offered by UVE provides a lower impact on the overall speedup obtained. However the UVE code significantly reduces the number of loop instructions of the kernel and, due to the fact that it is a SIMD extension, allows for code vectorization, which are the main factors for the large speedup obtained for a low loop size value. As the value of N is increased, two factors need to be taken into account. Firstly, the number of elements that are being processed increases, therefore the impact of code vectorization in the overall speedup values will increase, resulting in a speedup of around 13 times for a loop size of 32. Secondly, the number of memory accesses will also increase and eventually overshadow the benefits of code vectorization due to the kernel being memory bound. Consequently, the higher values of N show a speedup of around 10 times, which is mainly due to the use of data streaming in the UVE kernel, allowing for a reduction in memory access latency, instead of the vectorization of the code which, despite still affecting the overall result, has a lesser impact when the kernel is memory bound and the number of memory accesses is high.

6.2.2 Scaled Vector Addition

Similarly to the Memory Copy benchmark, the Scaled Vector Addition kernel also make use of the same one dimensional, unitary stride memory access pattern. However, a key difference exists, related to the use of a multiplication instruction followed by an add instruction in the loop code. The use of these instructions, in particular the multiply instruction, makes this benchmark more dependent on the compute performance of the Vector Accelerator, allowing for analysis of the impact of vectorization support on the developed system.

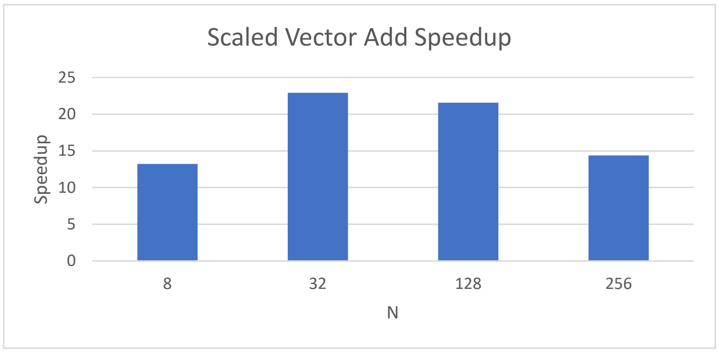


Figure 6.3: Scaled Vector Addition kernel speedup.

The benchmark was evaluated using the same loop size values as the memory copy benchmark, due to the fact that they describe the same memory access pattern and so that the impact of vectorization can be evaluated by comparing the results of both benchmarks. The speedup results of the Scaled Vector Addition kernel are illustrated in figure 6.3. For the lowest loop size value, N = 8, a speedup of around 13 times was achieved. This value is significantly higher than the speedup of 8 times obtained with the memory copy kernel for the same loop size value. This is the case due to the fact that this kernel is more computationally intensive that the memory copy kernel, and due to the difference of latency between the scalar multiply instruction and the vector multiply instruction. In detail, the scalar multiply instruction has a latency of 6 clock cycles, while the corresponding vector multiply instruction has a latency of 1 clock cycle. Once again, when N is increased the impact of vectorization is more noticeable and a speedup of 22 times is obtained for a loop size of 32. If one continues to increase N, then the memory access component of the kernel becomes more relevant due to the increase of memory accesses and a speedup of around 14 times is achieved. By analysing both the memory copy and scaled vector addition speedup graphs at the same time, one notices that they follow the same trend, i.e they increase and decrease the speedup values in the same fashion, given an N value, however the scaled vector addition speedups are overall higher due to the higher computational complexity of the kernel that can be explored by the vector instructions provided by the UVE extension.

6.2.3 Matrix Vector Multiplication

Contrary to the benchmarks so far, the Matrix Vector Multiplication algorithm describes a more complex memory access pattern. In detail, it describes a two dimensional access pattern, used to load a 2D matrix, and one dimensional access patterns to load the vectors used in the algorithm. Consequently, this benchmark provides an evaluation of the impact of data streaming and code vectorization on more complex access patterns.

The benchmark was evaluated using a set of four different *N* values. However in this case one must note that the number of memory accesses in now proportional to the square of N, since the algorithm processes a $N \times N$ matrix. The values of N used and the obtained speedups are represented in figure 6.4. For N = 4, a speedup of 6 times was obtained. This speedup value is lower than the speedup obtained for the lowest N value of the other benchmarks analysed so far. This is the case due to the

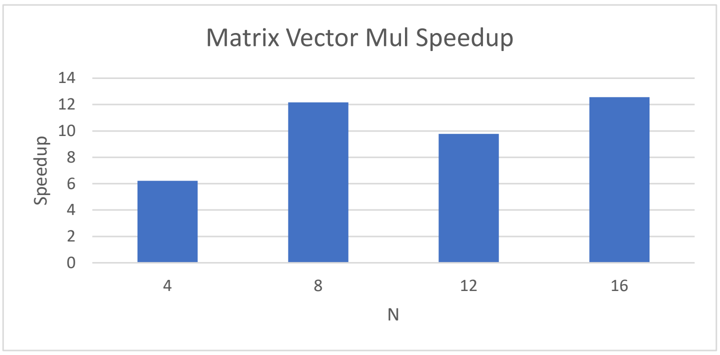


Figure 6.4: Matrix Vector Multiplication kernel speedup.

fact that this benchmarks requires the setup of a larger number of streams, including a two dimensional stream, which incurs in a higher overhead that affects speedup values when the amount of data to be processed is small. However, once N is scaled to higher values the speedup values increase drastically. Specifically, for N = 16 the speedup doubles, resulting in a value of around 12 times. This value is is significantly higher than the speedup obtained by the memory copy kernel for N = 256, despite both kernels being memory bound, and close to the speedup value of the scaled vector addition kernel due to the fact that the data streaming mechanisms provided by the UVE extension offer a higher gain of performance in more complex memory access patterns, since the store and load instructions are dealt with by the Streaming Engine, allowing for data prefetching, and eliminating unnecessary indexation instructions.

6.2.4 Energy Efficiency Analysis

To analyse the impact of executing the analysed benchmarks using the vectorized UVE kernel in relation to the baseline RISC-V kernel, the Energy Delay Product, calculated using the total power requirement value of the system P and the execution time of the benchmarks t, $EDP = P \times t^2$ was used. A lower value of EDP equates to a higher energy efficiency, since the equation is directly proportional to system dynamic power P and the square of the execution time t^2. The values for t used to calculate the values of EDP represented in table 6.4 were measured using the highest value of N for each benchmark to evaluate the most memory and computational intensive variant. From the table, one can observe that across all benchmarks the values of EDP are lower by 2 orders of magnitude when considering the UVE kernel, with the UVE/RV ratio ranging from 97 up to 195. This is to be expected due to the values of speedup obtained by using such kernel in favor of the base RISC-V kernel and the fact that the EDP is proportional to the square of the execution time, consequently benefiting from lower execution times in comparison to lower power requirements. These types of results are to be expected from architectures that employ SIMD support since this support requires an increase in overall power requirement of the system that must be compensated in the decrease of the execution time of applications in order for the system to remain energy efficient.

	Memory Copy	Scaled Vector Addition	Matrix Vector Multiplication
EDP RISC-V (Js)	4.23×10^{-5}	2.04×10^{-4}	1.50×10^{-4}
EDP UVE (Js)	4.62×10^{-7}	1.04×10^{-6}	1.01×10^{-6}

Table 6.4: Energy Delay Product values.

6.3 Summary

In this chapter, an implementation of the proposed system, using vector registers of 256 bits, was discussed in the first section. The area results show that most components require low resource usage, with the highest resource components being the Vector Accelerator, due to its 256 bit data bus and parallel lanes of execution, and the Xilinx XDMA IP. The operating frequency of the several clock domains of the system was then discussed, presenting two distinct results. One set of results was obtained by evaluating each clock domain separately, and the other set was obtained by evaluating the operating frequencies of the clock domains when the whole system was implemented. These results showed that the operating frequencies are lower when considering the whole system, due to routing and placement constraints on the FPGA. Finally, the dynamic power requirement of the system and its components was obtained using a post-implementation routine, revealing that 96% of the dynamic power requirements come from the XDMA, MiG and AXI Interconnect IPs.

The second section of this chapter presented the results of the benchmarks previously explained in chapter 5 and a an analysis of said results was made. The results showed that the use of the UVE extension provided speedups of up to 23 times and EDP values of up to 195, when compared with base RISC-V code, showing that the system is able to perform faster and in a more energy efficient manner when using UVE.

Chapter 7

Conclusions

Over the last couple years the number of RISC-V processors has increased drastically, with some high performance and Linux-capable solutions on the market. This is due to the contribution of both the open-source community and companies that adopted the use of the RISC-V ISA to develop and produce their own designs. This adoption has as a consequence the further development of custom extensions to provide the necessary tools to tackle either domain specific or general purpose tasks.

The development of this work began after the analysis of a novelty custom SIMD extension, the Unlimited Vector Extension. This extension provided support for SIMD instructions that operate on scalable vector length registers, with the added support of instruction set support for data streaming. The inclusion of ISA level data streaming support provides an alternative to the RISC-V V SIMD extension that is more suitable for the computational tasks demanded by the industry, such as machine learning, computer vision and data analysis.

Based on the analysis of the UVE extension requirements, a hardware streaming solution, i.e a Stream Engine, was developed and coupled with an already existing RISC-V system composed of a softcore, data cache and memory mapped peripherals. Subsequently, a Vector Accelerator, controlled by the original softcore, was introduced to the system to provide support for the computation SIMD instructions and support for consumption/production of data from/to the developed Stream Engine. The addition of the Vector Accelerator also required the introduction of an hardware arbitration block capable of controlling accesses from the softcore and the accelerator to both the data cache and memory mapped peripherals.

The complete system was then implemented successfully in a Xilinx Virtex UltraScale+ VCU 1525 FPGA board. The performance gains of the system using streaming and SIMD instructions, when compared to base RISC-V code, was measured with the use of a set of benchmarks, resulting in execution of up to 23 times faster and EDP values of up to 195 times lower when using the UVE extension.

7.1 Future Work

This system can be used in a future work for the development of a multi-core system, as was the original motivation that originated this work, however some further improvements to the system that can be developed are:

- Add support for indirect and descriptor modifiers to the Stream Engine, providing more flexible stream configuration
- Creation of an alternative connection for the Stream Engine, that provides communication with the Data Cache, to provide the user with more flexibility over the stream datapath
- Implement atomic and fencing instructions present in the RISC ISA, approaching operating system support
- Make the number of processing blocks used by the Stream Engine Configurable, so that lower area requirement implementations can be studied and compared
- Test the developed system in other FPGAs and evaluate an eventual ASIC implementation

Bibliography

[1] ARM. AMBA® AXI^{TM} and ACE^{TM} Protocol Specification, 2011.

[2] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[3] Western Digital. SweRV RISC-V Core from Western Digital. Available: `https://github.com/chipsalliance/Cores-SweRV`, 2020. [Online].

[4] João Mário Ribeiro Domingos. Unlimited Vector Extension with data streaming support. Master's thesis, Instituto Superior Técnico, October 2020.

[5] F. Zaruba, L. Benini. CVA6 RISC-V Core. Available: `https://github.com/openhwgroup/cva6#cva6-risc-v-cpu`, 2020. [Online].

[6] GNU. GCC Assembler Instructions with C Expression Operands . Available: `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm`, 2020. [Online].

[7] Y. Ishii, M. Inaba, and K. Hiraki. Access Map Pattern Matching forHigh Performance Data Cache Prefetch. *Journal of Instruction-Level Parallelism*, 2011.

[8] nandland. Register Based FIFO in VHDL . Available: `https://www.nandland.com/vhdl/modules/module-fifo-regs-with-flags.html`, 2020. [Online].

[9] N. Neves, P. Tomás, and N. Roma. Efficient Data-Stream Management for Shared-Memory Many-Core Systems. *International Conference on Field-Programmable Logic and Applications*, 2015.

[10] N. Neves, P. Tomás, and N. Roma. Adaptive In-Cache Streaming for Efficient Data Management. *IEE Transactions on Very large Scale Integration Systems*, 2016.

[11] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-Dataflow Acceleration. *International Symposium on Computer Architecture*, 2017.

[12] RISC-V. RISC-V Assembly Programmer's Manual. Available: `https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md`, 2020. [Online].

[13] RISC-V. Official RISC-V GNU Toolchain, including GCC. Available: `https://github.com/riscv/riscv-gnu-toolchain`, 2020. [Online].

[14] RISC-V. Spike, A RISC-V ISA Simulator. Available: https://github.com/riscv/riscv-isa-sim, 2020. [Online].

[15] RISC-V. RISC-V V vector extension. Available: https://github.com/riscv/riscv-v-spec, 2020. [Online].

[16] João Filipe Monteiro Rodrigues. Configurable RISC-V Softcore Processor for FPGA Implementation. Master's thesis, Instituto Superior Técnico, November 2019.

[17] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gbrielli, M. Hornsell, G. Magklis, A. Martinez, N. Premillieu, A. Rico A. Reid, and P. Walker. The ARM Scalable Vector Extension. Available: https://ieeexplore.ieee.org/document/7924233, 2020. [Online].

[18] A. Waterman and K. Asanovic. The risc-v instruction set manual. Technical report, University of California, Berkeley, 2019.

[19] Xilinx. Ultrascale Architecture-Based FPGAs Memory IP. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf, 2016. [Online].

[20] Xilinx. AXI Reference Guide. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, 2017. [Online].

[21] Xilinx. FIFO Generator v13.1 - LogiCORE IP Product Guide. Available: https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_1/pg057-fifo-generator.pdf, 2017. [Online].

[22] Xilinx. VCU1525 Reconfigurable Acceleration Platform - User Guide. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf, 2019. [Online].

[23] Xilinx. Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit. Available: https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html, 2020. [Online].

[24] Xilinx. XIlinx DMA IP Reference Drivers. Available: https://github.com/Xilinx/dma_ip_drivers, 2020. [Online].

[25] Xilinx. XIlinx DMA/Bridge Subsystem for PCI Express v4.1 - Product Guide. Available: https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf, 2020. [Online].

[26] Xilinx. Floating Point Operator v7.1 - LogiCORE IP Product Guide. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf, 2020. [Online].

Appendix A

RISC-V Supported Instructions

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S
0010000		rs2	rs1	001	rd	1010011	FSGNJN.S
0010000		rs2	rs1	010	rd	1010011	FSGNJX.S
0010100		rs2	rs1	000	rd	1010011	FMIN.S
0010100		rs2	rs1	001	rd	1010011	FMAX.S
1100000		00000	rs1	rm	rd	1010011	FCVT.W.S
1100000		00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000		00000	rs1	000	rd	1010011	FMV.X.W
1010000		rs2	rs1	010	rd	1010011	FEQ.S
1010000		rs2	rs1	001	rd	1010011	FLT.S
1010000		rs2	rs1	000	rd	1010011	FLE.S
1110000		00000	rs1	001	rd	1010011	FCLASS.S
1101000		00000	rs1	rm	rd	1010011	FCVT.S.W
1101000		00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000		00000	rs1	000	rd	1010011	FMV.W.X

Appendix B

UVE Supported Instructions

so.a.add.s vd vs1 vs2 ps3	vector signed vertical add
so.a.sub.s vd vs1 vs2 ps3	vector signed sub
so.a.mul.s vd vs1 vs2 ps3	vector signed mul
so.a.adde.s vd vs1 vs2 ps3	vector signed horizontal add
so.a.nand vd vs1 vs2 ps3	vector bitwise nand
so.a.and vd vs1 vs2 ps3	vector bitwise and
so.a.nor vd vs1 vs2 ps3	vector bitwise nor
so.a.or vd vs1 vs2 ps3	vector bitwise or
so.a.not vd vs1 ps3	vector bitwise not
so.a.xor vd vs1 vs2 ps3	vector bitwise xor
so.a.inc vd vs1 ps3	vector increment
so.a.dec vd vs1 ps3	vector decrement
so.v.ld.s.[width] vd rs1 rs2	vector load
so.v.st.s.[width] vd rs1 rs2	vector store
so.v.dp.[width] vd rs1 rs2	vector duplicate from scalar
so.v.mv.[stream] vd vs1 ps2	vector move from vector register or stream
so.v.mvt.[stream] vd vs1 ps2	transposed vector move variant
so.v.mvvs rd vs1	vector move from vector to scalar
so.v.mvsv.[width] vd rs1	vector move element from scalar to vector
ss.ld.[width] vd rs1 rs2 rs3	simple load stream
ss.st.[width] vd rs1 rs2 rs3	simple store stream
ss.sta.ld.[width] vd rs1 rs2 rs3	start load stream configuration
ss.sta.st.[width] vd rs1 rs2 rs3	start store stream configuration
ss.app vd rs1 rs2 rs3	append descriptor to stream in configuration
ss.end vd rs1 rs2 rs3	end configuration of stream
ss.cfg.vec vd	configure vector to mirror stream pattern topology
so.b.ndc.x vs1	branch if dimension x not finished
so.b.dc.x vs1	branch if dimension x finished
so.b.nc vs1	branch if stream not finished
so.b.c vs1	branch if stream finished

Table B.1: UVE Supported Instructions.

Appendix C

Board Support Package

Listing C.1: entry.S

```
.section .init
.globl _start
.type _start,@function
_start:
# Initialize the global pointer
.option push
.option norelax
la gp, __global_pointer$
.option pop
# Initialize the stack pointer
la sp, _sp
# Clear the bss section
la a0, __bss_start
la a1, _end
bgeu a0, a1, 2f
1:
sw zero, (a0)
addi a0, a0, 4
bltu a0, a1, 1b
2:
auipc ra, 0
addi sp, sp, -16
sw ra, 8(sp)
# Start the program
li a0, 0
li a1, 0
call main
```

Listing C.2: exit.c

```c
void __exit(void) {
  asm volatile ("nop");
  asm volatile ("nop");
  asm volatile ("nop");
  asm volatile ("jalr zero, zero, 0x0");
}
```

Listing C.3: Linker Script

```
OUTPUT_ARCH ( "riscv" )
ENTRY ( _start )
MEMORY
{
rom (x!rw) : ORIGIN = 0x00000000, LENGTH = 16K
ram (wxa!ri) : ORIGIN = 0x00004000, LENGTH = 16K
}
SECTIONS
{
__stack_size = 4K;
.init : {
KEEP (*(SORT_NONE(.init)))
} >rom
.text : {
*(.text .text.*)
} >rom
.fini : {
KEEP (*(SORT_NONE(.fini)))
} >rom
PROVIDE ( __etext = . );
PROVIDE ( _etext = . );
PROVIDE ( etext = . );
.rodata : {
*(.rdata)
*(.rodata .rodata.*)
} >ram
.data : {
*(.data .data.*)
*(.sdata .sdata.*)
PROVIDE( __global_pointer$ = . );
} >ram
PROVIDE ( _fbss = . );
PROVIDE ( __bss_start = . );
.bss : {
*(.sbss)
*(.bss .bss.*)
*(COMMON)
} >ram
PROVIDE ( _end = . );
PROVIDE ( end = . );
.stack ORIGIN(ram) + LENGTH(ram) - __stack_size : {
PROVIDE ( _heap_end = . );
. = __stack_size;
PROVIDE( _sp = . );
} >ram
}
```