RISC-V Processing System with streaming support

Eduardo Miguel Ferreira Cabral de Melo, Instituto Superior Técnico, Universidade de Lisboa

***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 novelty extension developed for the RISC-V ISA, namely the Unlimited Vector Extension(UVE), that provides ISA level support for data streaming coupled with scalable SIMD instructions. 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.**

***Index Terms*—RISC-V, UVE, Data Streaming, SIMD, Vector Accelerator, FPGA**

I. Introduction

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, in the case of INTEL and ARM, or over the changes made to the instruction set architecture, including its flexibility, in the case of ARM. 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 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.

This work 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 [6].

II. Related Work

This work set out as an objective the implementation of a novelty RISC-V SIMD extension, named Unlimited Vector Extension (UVE) [2] on top of a RISC-V Softcore System [6]. Consequently, this section will provide context to the reader about the necessary knowledge of the UVE extension needed to understand the system implementation.

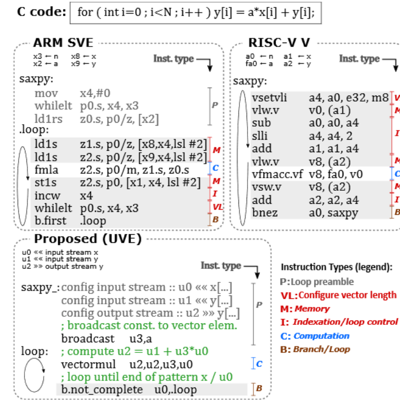

Fig. 1. Saxpy example kernel implementations using ARM SVE, RISC-V V and the UVE extension [2]. Instructions with shaded background represent loop overhead.

The Unlimited Vector Extension [2] 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, when compared with other solutions [7] [5], 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 1. 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.

A. Memory Access Representation

The UVE extension uses a memory access representation based on stream descriptors, inspired by [3], [4], 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.

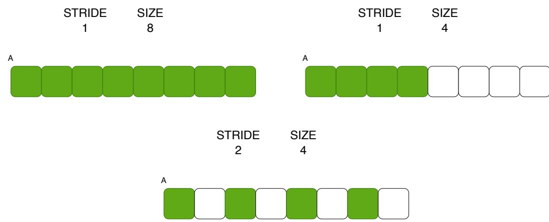

Fig. 2. One dimensional access examples.

Using regular C code, one could represent an one dimensional access to memory using a single for loop. This access makes use of three fundamental variables, the base address of the variable to be addressed, for example *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.

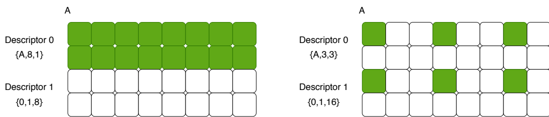

Fig. 3. Two dimensional access examples.

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 3.

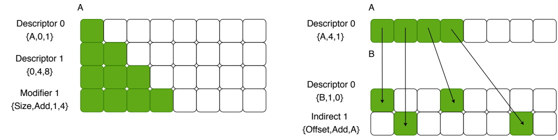

Fig. 4. Lower triangular and Indirect Access Examples.

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, such as indirect accesses and accesses with inter-loop dependencies, i.e outer loop modifies descriptor values of inner loop.

Firstly, 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.

Secondly, 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 some examples that illustrate the use of the static and indirect descriptor modifiers can be observed in figure 4.

B. 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.

1) Vector Register File: 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.

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. 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.

2) Predicate Register File: 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).

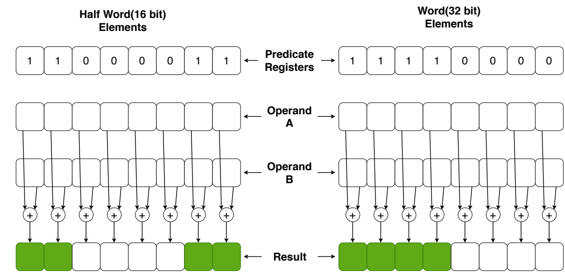

Fig. 5. Predication Examples.

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 5, 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.

3) Stream Interface: 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.

C. 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.

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

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

Fig. 6. Simple 1D stream configuration instructions.

1) 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 6. 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]
```

Fig. 7. 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, the UVE extension provides access to three types of stream configuration instructions, represented in figure 7. 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.

```
so.b.[n]c

so.b.[n]dc
```

Fig. 8. Stream loop control instructions.

2) 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 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.

III. PROPOSED HARDWARE MODIFICATIONS

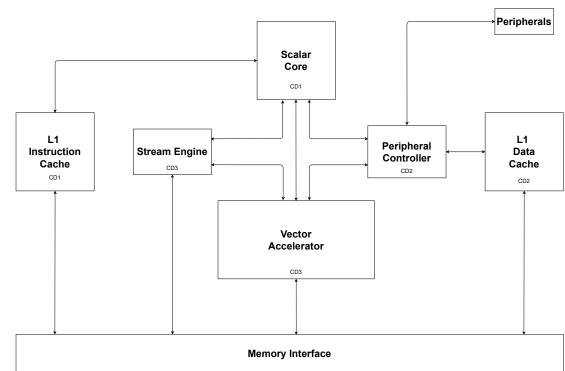

Fig. 9. RISC-V Streaming Processing System.

This section will explain in detail the architecture of the new hardware blocks introduced by this work, mainly the Stream Engine and Vector Accelerator, which are illustrated in figure 9. The Stream 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 Stream Engine in order to process stream data.

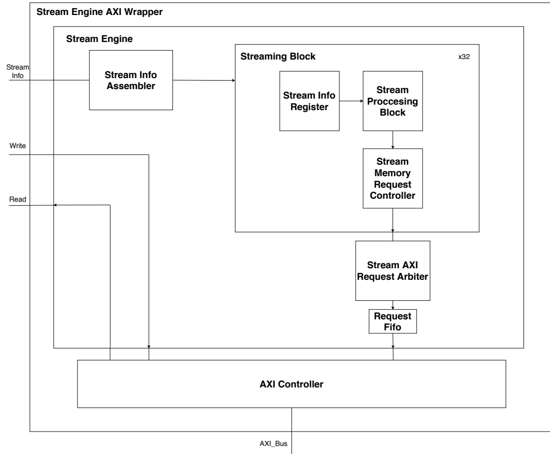

Fig. 10. Stream Engine Component Diagram.

TABLE I
STREAM INFO ASSEMBLER INPUT SIGNAL TYPES AND WIDTHS.

Signal	Type	Width(bits)
s_start	control	1
s_append	control	1
s_end	control	1
s_id	data	5
s_ldst	data	1
s_width	data	2
s_baddr	data	32
s_size	data	32
s_stride	data	32

A. Stream Engine

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(add a reference to table with UVE instructions) should be present. With that goal in mind, a hardware streaming solution was designed and coupled with both the Scalar Core and Vector Accelerator. The Scalar Core provides the decoded streaming control signals from the streaming instructions to the Stream Engine while the Vector Accelerator implements an interface that allows sending/receiving data to/from the Stream Engine.

The Stream Engine is composed of 5 main types of hardware blocks, represented in figure 10.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].

1) Stream Info Assembler: 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 Stream Engine. For this reason, the first stage of the Stream 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 I.

The signal *s_start* informs the stream 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.

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. 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.

3) Stream Descriptor Update Circuit: The basic information block of a stream is the stream descriptor and is composed of three values:

- 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

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

One can easily implement equation 1 in hardware, resulting in the circuit illustrated in figure 11. 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.

The circuit also contemplates a Descriptor Register Bank, which is a Register File containing all descriptor variables and counters needed to update a descriptor from the hierarchy each clock cycle.

4) Stream Descriptor Update FSM: In order for the processing block to be complete and compliant with the UVE

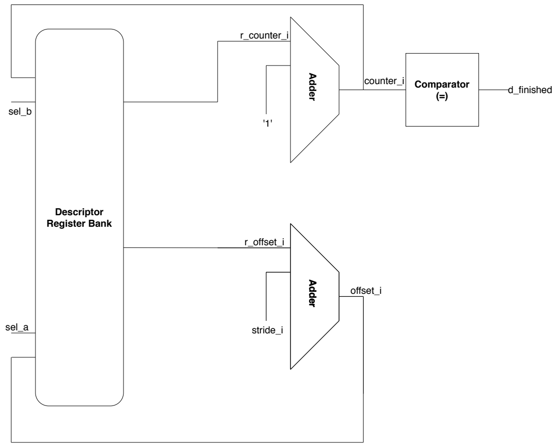

Fig. 11. Multiple Descriptor Update Circuit Diagram.

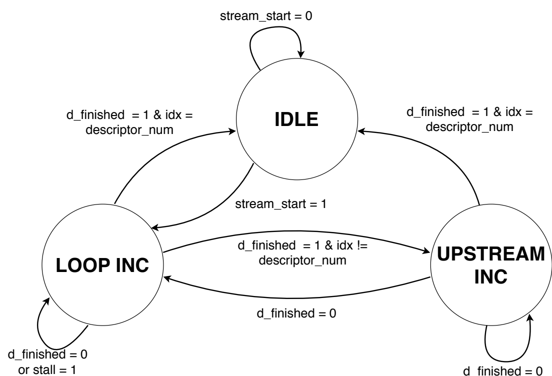

Fig. 12. Processing Block Finite State Machine Model.

streaming instructions, a finite state machine was added to provide not only control signals to the descriptor register bank in figure 11, but also implement the hierarchical organization of the stream descriptors described in II. The FSM implements this behaviour using the following states:

- IDLE: Processing Block is idle and waiting for a stream to process
- LOOP INC: First descriptor values of the stream being processed is updated, and an address is produced, each clock cycle
- UPSTREAM INC: Descriptors higher in the hierarchy are conditionally updated, depending on if the dimension associated with the given descriptor finished, providing new offset values to be used by the first descriptor to calculate new addresses

5) Stream Memory Request Controller: Section III-A2 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 request to main memory, however in order to use the maximum bandwidth possible from the bus connecting the Stream Engine to main memory, a Memory Request Controller was designed and implemented.

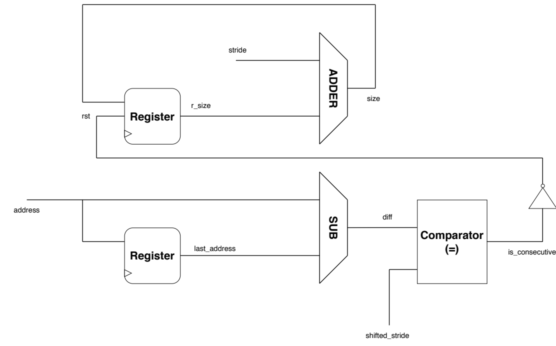

Fig. 13. Memory Request Controller Detection Circuit.

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 and sends a single request to main memory that loads/stores several elements using a single base address and stride information. With this in mind, the circuit illustrated in figure 13 was designed. The circuit uses the current address, 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. 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 Stream 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.

6) Stream Request Arbiter: The Processing and Request stages of the Stream Engine have been explained in detail in sections III-A2 and III-A5 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 was integrated into the design of the Stream Engine.

	1	2	3	4
1	X	1	1	1
2	0	X	1	1
3	0	0	X	1
4	0	0	0	X

Fig. 14. Internal Matrix Representation Example.

The arbitration scheme used by the Stream Engine Request Arbiter is the matrix arbitration scheme. This dynamic arbitration scheme guarantees that the master who last used the common bus will have the lowest priority, using an internal matrix representation of priorities, illustrated in figure 14 for a system with 4 masters. If the value at row i, column j 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 0 and values in column i will be set to 1, ensuring that all masters will have priority over master i. This arbitration scheme guarantees that no single stream can have total control of the request bus, allowing for smooth arbitration between concurrent streams.

7) Stream AXI Controller: The previous sections described how the Stream Engine configures, processes and manages requests of streams, yet there is still a need to forward the requests produced by the engine to 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 implementation 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.

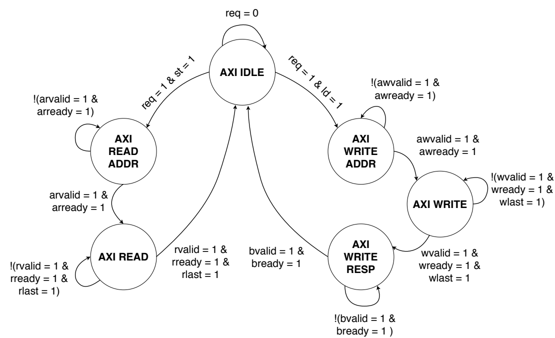

Fig. 15. AXI Controller Finite State Machine Diagram.

As such, an AXI Controller block was developed and added to the Stream Engine structure, which implements the finite state machine model represented in figure 15. This FSM model implements the handshake model of transactions used by the AXI interface, using several states:

- AXI IDLE: Wating for AXI Request
- AXI READ ADDR: Exchange of read address and burst type information
- AXI WRITE ADDR: Exchange of write address and burst type information
- AXI READ: Exchange of data requested by a read request
- AXI WRITE: Exchange of data requested by a write request
- AXI WRITE RESP: Exchange of information that indicates the success or failure of a write request

8) Stream Data Input/Output Interface: The Stream 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, II and III, and a small description of the function of each signal is provided.

TABLE II
STREAM ENGINE READ INTERFACE.

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 stream engine if there is data to be read

TABLE III
STREAM ENGINE WRITE INTERFACE.

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 stream engine if data can be written

B. Vector Accelerator

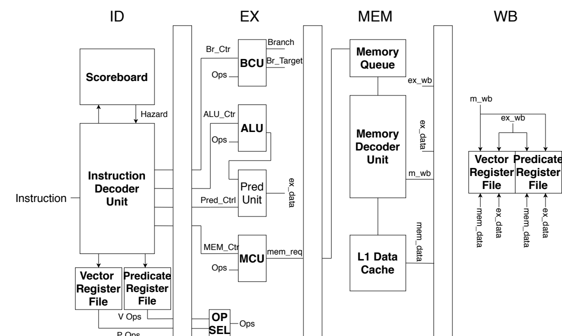

Fig. 16. Vector Accelerator Pipeline Diagram.

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 instructions and as such features programmable vector width.The simplified diagram of the accelerator pipeline is illustrated in figure 16.

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.

The Instruction Decode also provides a memory bank, separate from the Vector Register File, to store data both incoming and outgoing from/to a stream, that implements the read and write interfaces of III-A8. 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, consequently support both for read and write support for each component, stream engine and vector accelerator, is necessary. 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.

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.

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. The variable width adder is implemented using 32-bit programmable adders in paraller, while the variable width multiplier is implemented using 32-bit programmable multipliers that use the radix-4 algorithm for multiplication.

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.

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 first-in first-out queue 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) 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. 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.

IV. Implementation

The implementation process of the proposed architecture was carried out using the Xilinx Vivado tool, version 2019.1 in a Linux environment using the Xilinx Virtex UltraScale+ VCU1525 FPGA [8] and several routines were run to analyse system area, operating frequency and power requirements.

A. Area Analysis

To analyse the area requirements of the system 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 Stream Engine, which can be observed in table IV. The component that demands more resources is the Vector Accelerator due to exploring hardware parallelism by using multiple lanes of execution. All other components, with the exception of the complex XDMA IP, require low amount of resources from the FPGA.

TABLE IV
Component FPGA resource usage.

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
Stream 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

B. 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 Stream Engine use clock domain 3. The maximum operating frequencies of these clock domains were evaluated both isolated and integrated in the whole system, and the results can be observed in table V. When isolated, the operating frequencies are higher due to less resource requirements, more placement possibilities and routing flexibility.

TABLE V
MAXIMUM OPERATING FREQUENCY RESULTS.

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

C. Power Analysis

The Vivado tool provides the user with a post-implementation routine that is run on the placed design and outputs the power consumption of the system as a whole and its components. Using this feature, the power consumption of the developed system was analysed and the results are illustrated, per component, in table VI. 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 Stream 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 Stream Engine. The total dynamic power requirement of the system is obtained by summing the entries of table VI and equates to $10.061W$.

TABLE VI
DYNAMIC POWER REQUIREMENT PER COMPONENT.

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
Stream Engine	0.352	3
XDMA	7.037	70
MIG	1.472	15
AXI Interconnect	0.834	8

V. BENCHMARK RESULTS

This section will go over the results obtained by running a set of benchmarks, namely the Memory Copy, IAXPY and Matrix Vector Multiplication benchmarks, when simulating the implemented system, with implemented vector register length equal to 256 bits and including the memory controllers for accurate memory latency simulation, using the Xilinx Vivado application. Further, an energy efficiency analysis of the system is performed using the Energy Delay Product metric.

A. Memory Copy

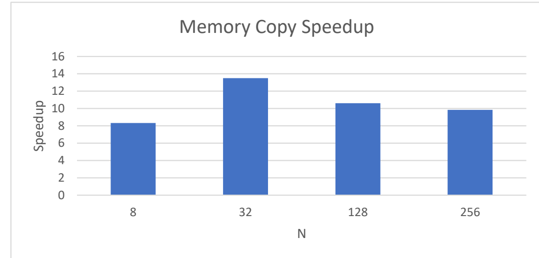

Fig. 17. Memory Copy kernel speedup.

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, corresponds to moving an entire vector, and the last loop size, N = 256, loads the exact amount of data to fill the entire data cache of the system. The speedup values, represented in figure 17, were obtained by comparing the execution times of the RISC-V kernel (baseline) with the UVE kernel. Speedups varying from 8 to 13 times were achieved when using the vectorized UVE code, due to the use of data streaming instructions and SIMD instructions, that both mask the latency of memory operations and increase the data throughput of the system.

B. IAXPY

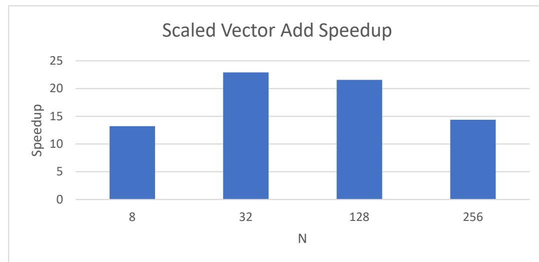

Fig. 18. IAXPY kernel speedup.

Similarly to the Memory Copy benchmark, the IAXPY kernel also make use of the same one dimensional, unitary stride memory access pattern, however a key difference exists, this difference being 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 a more compute bound algorithm than the memory copy benchmark, allowing for analysis of the impact of vectorization support on the developed system. 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. Comparing figure 17 to 18, it can be observed that both graphs follow the same curve, however the IAXPY speedups are higher across the board, ranging from 14 to 22 times. This is due the the more compute-bound nature of the IAXPY kernel when compared with the Memory Copy kernel, and as such, the vectorization of the code provides more benefits in speedup.

C. Matrix Vector Multiplication

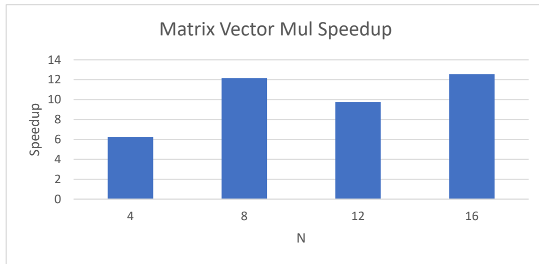

Fig. 19. Matrix Vector Multiplication kernel speedup.

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 values of N used and the obtained speedups, ranging from 6 to 12 times, are represented in figure 19. Contrary to the Memory Copy and IAXPY results, the increase in loop size provides a greater increase in speedup values, even in the highest value of N used. This is the case because the memory access pattern is more complex, consequently the use of data streaming in the UVE kernel provides more relevant gains when the memory accesses increase, compared with the base RISC-V kernel.

D. 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 values were calculated using the dynamic power requirement of the system and the execution times of the different benchmarks, for the highest value of *N*. 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 lower execution times in comparison to lower power requirements. These results also indicate that the use of the UVE extension for computationally or memory intensive algorithms, such as the benchmarks used, results in higher energy efficiency when compared with the use of base RISC-V code.

TABLE VII
ENERGY DELAY PRODUCT VALUES.

	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}

VI. CONCLUSION

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 architecture level support for data streaming.

Based on the analysis of the 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 was introduced to the system to provide support for the SIMD instructions provided by the UVE extension.

The complete system was then implemented successfully in a Xilinx Virtex UltraScale+ VCU 1525 FPGA board and subsequently 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.

REFERENCES

[1] ARM, “AMBA® AXI™ and ACE™ Protocol Specification,” 2011.

[2] J. M. R. Domingos, “Unlimited Vector Extension with data streaming support,” Master’s thesis, Instituto Superior Técnico, October 2020.

[3] 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.

[4] ——, “Adaptive In-Cache Streaming for Efficient Data Management,” *IEE Transactions on Very large Scale Integration Systems*, 2016.

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

[6] J. F. M. Rodrigues, “Configurable RISC-V Softcore Processor for FPGA Implementation,” Master’s thesis, Instituto Superior Técnico, November 2019.

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

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