Hardware Optimization of Vector Operations for the RISC–V Instruction Set Architecture

Alex Short
Computer Engineering Department
Physics Department
California Polytechnic State University
San Luis Obispo, United States of America

Srinivas Sundararamen
Computer Engineering Department
California Polytechnic State University
San Luis Obispo, United States of America

***Abstract*—The need for operations performed on vectors (lists) has become more demanding as processes such as graphics and simulations have become more intensive. The idea to optimize hardware for these types of operations rather than software has led to breakthroughs in devices such as GPUs, and has provided software developers with more freedom in design for cutting edge graphics. In this report we focus on the extensions a designer can make to general processors (CPUs) in order to allow the hardware to perform vector operations. We also explore the potential hardware implementation in the OTTER MCU for the RISC-V architecture.**

***Index Terms*—Instruction Set Architecture, Vector Operation, SIMD, Processor, CPU, Parallelism**

I. Introduction

In many commercial processors, many consumers may notice a 'built in' graphics chip in their choice of processor. Both Intel and AMD processors appear to have a sort of graphics capability, which in turn mean that the processor itself has been optimized for graphics. Central processing units (CPUs) are designed based on an instruction set architecture (ISA), a specification that details which operations a CPU must be able to run. An ISA usually consists of a list of instructions, and the hardware must be designed to process and execute all of those instructions. For example, the RISC-V ISA has 38 base instructions [1].

Hardware can be optimized for different types of processes, and this is specified in the ISA. Different types of instructions are indicators for different applications, and this is especially true for graphics. Graphics, general image processing, and even sound processing require mathematical operations on matrices and vectors (or more generally, tensors), which could be incredibly slow on a normal RISC-V CPU. The main idea behind optimizing operations on tensors is called 'parallelism', running multiple instructions at the same time. In the case of GPU architecture and graphics processing we often need to run the same instruction on every component of a tensor.

Almost every type of ISA has a set of complementary instructions geared towards vectors. A vector instruction (also known as a SIMD instruction) is an instruction that executes on every element of a vector. This is especially useful in GPU architectures, as they often are designed for parallelism and running instructions on multiple values at once. For RISC-V, multiple attempts have been made (and successfully executed) to design a SIMD extension for the R-V ISA, [2], [3], so the goal of this report is to discuss how we might tackle the problem of designing vector instructions for RISC-V and how the hardware implementation would change.

These past works include versions of most integer instructions rebuilt for vectors, and specify some of the hardware changes to optimize vector operations. This includes discussions of the number of functional units, the types of signals needed, and size of vectors, which will all be discussed later in this paper.

The major advantages of vector instructions appear in the speed, compactness, and scalability of the operations. As already stated, a vector instruction executes many operations at the same time, vastly increasing the speed of element-wise operations on large lists, which makes cutting edge simulation and graphics possible even on small devices. Moreover, N instructions can then be encoded into one. When coding in assembly, a vector ISA allows multiple operations to be compacted into one line, making code easier and simpler to read. The ability to execute many instructions at once scales exponentially with the addition of parallel CPUs. If multiple CPUs run vector instructions in parallel, then the amount of operations per cycle increases drastically, a huge advantage for those computers.

One of the most important applications of parallel computing is in machine learning. Machine learning is usually based on long lists of weights and balances, and mathematical operations need to be performed on all of them. A vector ISA and implementation in a CPU can allow the machine learning to progress at a very fast rate. Understanding vector instructions is important for almost every field of computer science and architecture that includes large scale mathematics, as the speed improvements are too high to ignore.

II. Vector Instructions

Normal instructions in the RISC-V instruction set typically have an op-code, a destination register, and one or more input registers. To create the environment necessary to work with vectors we need to introduce a 'vector file' an analog to the register file that instead stores lists of 32-bit values, or vectors. We chose a vector file containing eight vectors, each with eight components.

Vector instructions need to have a similar structure to integer instructions in order to be easily integrated into CPU. In RISC-V, instructions begin with a seven bit op-code, so our vector instructions will as well. After this, different types of instructions begin to have different structures. For our exploration, we designed vector I-type instructions, I', S-type instructions, S', and R-type instructions, R'. These are the most important types for vectors, loading, storing, and mathematical operations.

I-type instructions for integers require a register destination, a 3 bit identifier (called a 'func3') and a two input registers. For vectors, all of this remains, but the destination is instead a vector in the vector file. The op-code is seven bits, as stated before, and the vector destination is technically only needed to be three bits, but we decided to keep it as five to be consistent with the integer instructions. Bits 12-14 were then kept as the func3, and the next fields were the input registers. The first is either from the vector file or the register file depending on whether the instruction is a load or not. Thus it was kept to five bits. The next input is the immediate. In the context of vectors, the immediates were treated as items in the register file, so that operations like 'vaddi' would act on all components of a vector with a value from the register file. After the input registers is a requirement unique to vector instructions, called the 'parameters'. The type of vector (byte, half word, or word) and the length of the vector need to be known when storing and loading with memory. The system can handle vectors of all three types, with vector lengths up to eight components. Thus the parameters field is five bits, three for the length and two for the type. Finally, the instruction is padded with two more zeros to make the full instruction 32 bits.

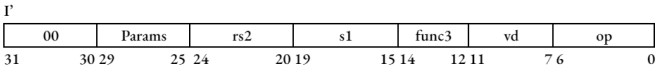

Fig. 1. instruction layout for vector I-type instructions

Similar changes were made for the vector S and R type instructions. Notably for S-type instructions, the vector form needs the immediate value to be three bits less in length. The extra parameters field needs five bits, but the second input register is a vector, and thus only needs three bits.

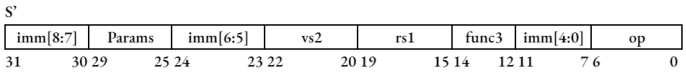

Fig. 2. instruction layout for vector S-type instructions

With the R' type instructions, both input registers are vectors and only need to be three bits, but since no parameters are required and no immediate value is needed to be input, we kept those as five bits to be more consistent with the rest of the ISA.

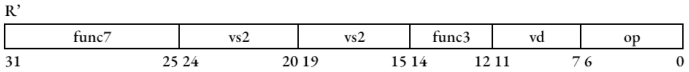

Fig. 3. instruction layout of R' (vector R) type instructions

III. Hardware Implementation

A. The Vector File

Changes and additions are required to implement this ISA in hardware. The most obvious addition is the vector file. This module contains eight rows of eight words, which corresponds to eight vectors. As such, addresses are three bits, and each vector has eight components.

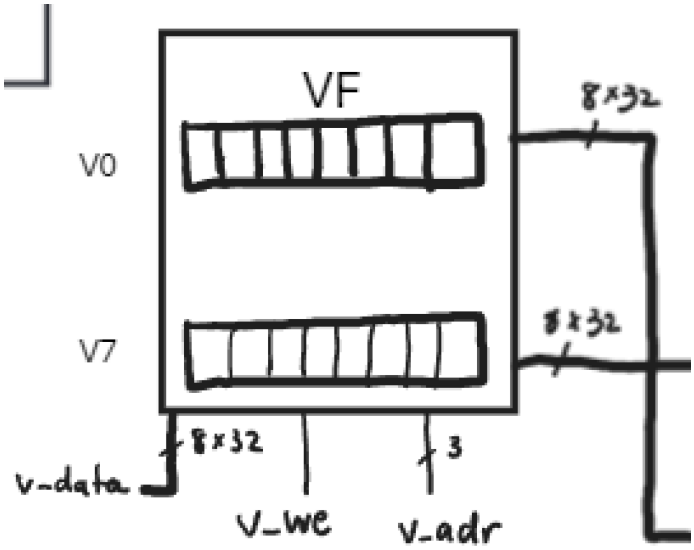

Fig. 4. vector file component of the OTTER MCU used to store temporary vectors for operation

As can be seen in the diagram, there is a 3 bit address input, as well as a single write enable signal. This would be controlled by a MUX at the end of execution, similar to the normal register file. Importantly, the vector at register 000 is held to zero in a similar fashion to the register file. In addition, there are two outputs, and each one is a set of eight 32 bit vectors. These are vectors read from the register file for execution in the next changed module, the ALU.

B. The ALU

The arithmetic logic unit (ALU) is the component in which mathematical operations are applied to inputs, which of course is incredibly important for vector capabilities. We decided to use an 8-lane ALU which is essentially eight ALUs, one for each potential component of an input vector. Doing this takes more resources, but allows the entire vector operation to be completed in one cycle. A designer could use a 2-lane or 4-lane which would both still improve efficiency over the base CPU. The first components of both input vectors are wired into the first lane of the ALU, and so on for all components. The output is then eight components, and is easily fed back into the v_data field of the vector file.

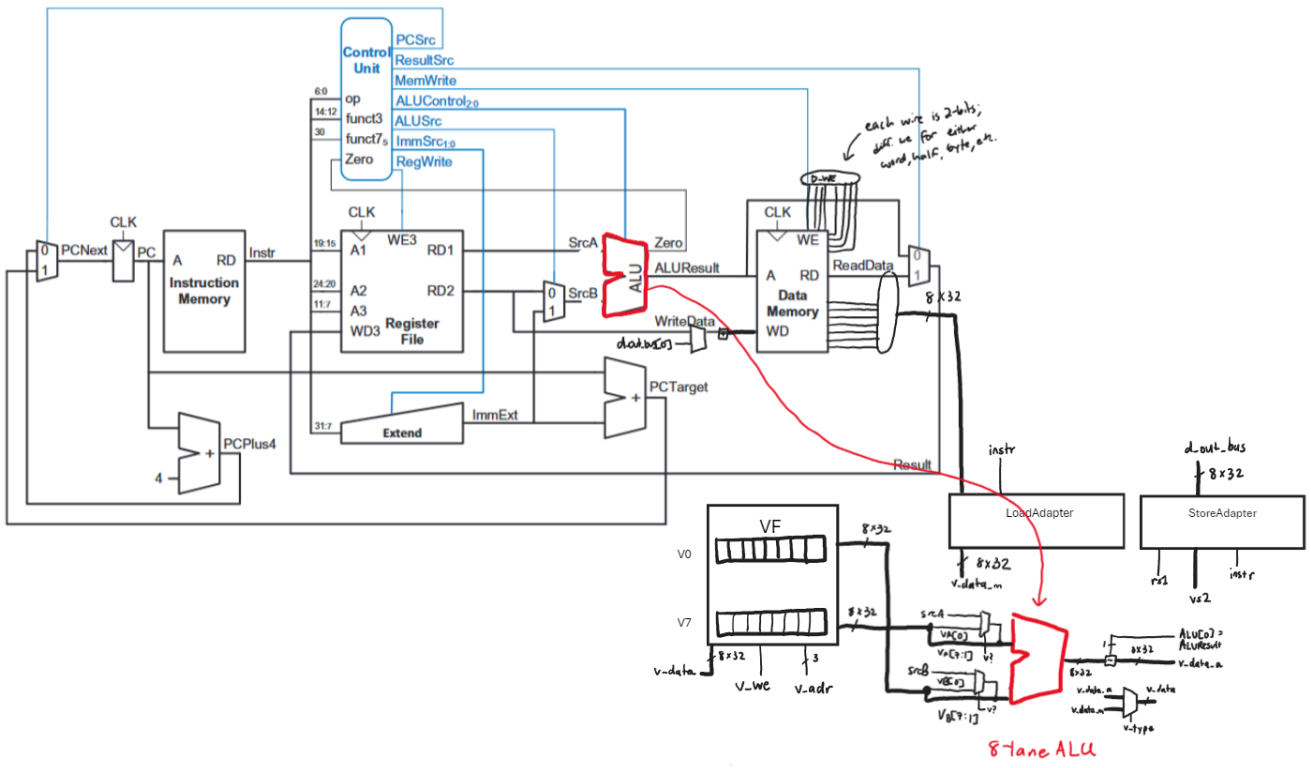

Fig. 5. The full design of the vector capable OTTER MCU including all new modules and signals

C. Loading and Storing

Loading a vector from memory presents an interesting challenge considering the handling of different types of vectors (bytes, half-words, or words). Stored data memory is packed closely together, and so the data memory module was modified to output eight words at all times. The first word would be wired into the register file and the vector file, and the remaining seven would just be wired into the vector file. This is necessary, as the maximum amount of data required to load a vector is eight words. If the vector is made of bytes, not words, technically only two words would need to be read from data, but eight words will always be read.

To handle the different types, we created a 'Load Adaptor' module, which takes as input the eight words from data memory and unpacks them into a usable vector. If the vector is made of bytes, then this adaptor would unpack the first two words into their separate bytes and output eight 32-bit components, each component with one byte. The complexity of this module is compounded with the fact that vectors can be different lengths, but we know that this module can be asynchronous if implemented using assign statements in verilog.

Both the load and store adaptors can be seen in Fig. 5. Storing is even more complex than loading, as the action of repacking a vector is difficult and also requires an adaptor. The adaptor takes in the vector, and outputs eight 32 bit values, containing the compressed vector. If the vector was made of half words, the input would be eight 32 bit values where the first half of each value is valid data. The output, however, would be eight 32 bit values where only the first four registers are filled with valid data. It is for this reason that we have added eight 'write enable' signals to the data memory module, one for each of the outputs from the store adaptor. These signals are two bits, with each value corresponding to an action.

When writing to memory, it is inadvisable to unnecessarily overwrite anything, and these eight write signals will prevent that. In the previous example, we want to fully write the first four input words, and not to write the last four input words, as they do not contain anything important. If the write enable signals are set to 00 for these last words, the data memory will not write them. It is possible that an input word is only filled with one byte (consider a vector of bytes with five components) or only filled with one half word. In these cases the enable signal is set to 01 and 10 respectively, indicating to write a byte or a half-word. In the case of a seven component vector of bytes, a fully compressed output from the adaptor would contain a 32 bit value with the first three bytes filled. To handle this, the adaptor would instead place two bytes in the the second output word and one byte in the third, setting the write enable signals as stated before. It is an efficient system to repack vectors and write them to memory without unnecessary removal of data. Of course, if the write enable signal is set to 11 the entire word will be written.

D. Algorithms and Further Complexity

The algorithms for the load and store adaptors have levels of complexity that can be further integrated into different designs for the RISC-V hardware. As stated before, the load adaptor takes in the eight words read from data memory, and must combine them to create a vector. Based on the Params given in the instruction, the adaptor can MUX different combinations of the eight words and sign-extending halves or bytes to fill each element as needed.

The store adaptor does the inverse of the load adaptor. Given an unpacked vector and a Params from the instruction, it packs the data back into memory by taking the element and outputting a max of 8 valid words. It is a little bit more complicated than the load adaptor because depending on the word boundary, certain data written into the data memory will be either stored as halves or bytes, as to not overwrite data that should not be replaced in data memory. Thus, certain MUXes can determine which part of the word in memory is overwritten, and the data memory is reconfigured to have 8 write-enables for each word that the store adaptor can output at once.

This hardware design is easily integrated with pipelines and caches. For the pipelined architecture, data dependencies can be checked along with registers in the forwarding unit, and stall can still take place for load hazards. Control hazards can also easily flush vector instructions without much more work. For a cached system, there are a few ways to go about implementation. Firstly, a 2-way 8-set L1 cache that is used in the instruction memory can be implemented for data memory, except the cache still needs to output 8 words at a time (unless a burst cachline FSM is used). Another idea would be to have 2 sets of cache hierarchies; one for scalar memory, and one for vector memory (L1I cache, and L1V cache). This would definitely complicate dirty bit algoirthms and would need a lot more connections to wire up, but might be faster depending on the data regions that vectors and registers specifically work on. Finally, in order to maintain the integrity of the base RISC-V32I, the first wire of a vector BUS (input or output from data memory, first lane of ALU, etc) and for the register file.

In essence, the adaptors will be complicated modules, as they are mostly collections of chained MUXes to determine where certain pieces of data need to go. We desired an ISA extension that is integrable with different hardware designs, and these modules have that capability. Data dependencies and control hazards are handled for the vector file in the exact same way as the register file, and the cache can be optimized for better performance due to the new layer of complexity when reading from memory. Thus these new hardware changes can easily be implemented into our OTTER MCU.

IV. An Example

To aid in the understanding of these processes and components, an example is provided in thorough detail about how the vector operations would work.

Consider a situation in which the data memory looks like Fig. 6 below.

0x00001234
0x11112222
0x33334444
0x53530000
0x12340000
0x12123123
0x45645645
0x00000000

Fig. 6. hypothetical data memory

Suppose you read in two vectors of half-words each with eight components. This would require two instructions of the form 'vload v1, 0(t0), 1, 8'. The value 't0' represents a register in the register file containing an address. The numbers 1 and 8 represent the parameters that the vector contains half-words and that it has eight components. To translate this to machine code, see Fig. 1. The 'Params' field is 11101. 'rs2' is the register containing zero (the offset) so it is 0, and the rest are self-explanatory. The data memory would output all of those eight words in Fig. 6. Those would enter the load adaptor, and be unpacked into eight words:

0x00001234	0x00000000
0x00002222	0x00001111
0x00004444	0x00003333
0x00000000	0x00005353

Fig. 7. output from the load adaptor reading from memory

As can be seen the first four words in data memory are split between eight outputs and thus an eight component vector is constructed. The first line in the vector file would then contain the eight words shown in Fig. 7. A similar process occurs for the second vector, although now we read from address 0x10 in data memory, and therefore the adaptor breaks the last four words in data memory apart into eight outputs into the vector file.

Now we wish to add these two vectors, so we input the instruction 'vadd v3, v2, v1'. To translate this to machine code see Fig. 3. The first components of each vector are wired into the first ALU, the second are wired into the second ALU, and so on, each ALU performing the normal 'add' on the 32 bit inputs. The 32-bit inputs are then fed through the rest of the pipeline and then into the vector file.

To finish this example, we store the result of that 'vadd' operation back into data memory at address 0x0. The eight halfwords are wired into the eight inputs to the store adaptor, and then are compressed into four words. The vector in 'v3' would contain the values shown in Fig. 8:

0x00001234	0x00001234
0x00005345	0x00002323
0x00009a89	0x00007897
0x00000000	0x00005353

Fig. 8. input to the store adaptor from the vector file

Each row in this table would be compressed into one word by the store adaptor, and then the final four would simply be zero. Now the store adaptor would send those four words to the data memory, along with four write enable signals of 11. The last four signals would be 00, indicating that the data memory should not write the last four words. Thus, after the data memory writes to address 0x0, the memory will look like this:

0x12341234
0x23235345
0x78979a89
0x53530000
0x12340000
0x12123123
0x45645645
0x00000000

Fig. 9. data memory after the store instruction

V. Conclusion

We have successfully designed a method to extend the RISC-V ISA to add the capability for vector operations. Vector operations are instructions to the CPU that are run on multiple stored values at once. These values are stored in lists known as 'vectors'. Vector instructions operate on these lists. To extend the ISA required the addition of more modules and signals, and more instruction types. For RISC-V vectors, the most important types are the I, R, and S type instructions, as applications mostly involve mathematical operations. We presented a design for new instructions of these types built for vectors, and detailed the new modules needed for these instructions to execute properly.

The ability for a CPU to perform vector operations has universal applications in technology, ranging from graphics to machine learning to cryptocurrency. The need to execute mathematical operations on massive lists of values is prevalent in many fields of science and technology. In this report we detailed a method to design a CPU with this capability by extending the RISC-V ISA to include the loading, storing, and interaction of vectors. If this work were to be continued, a verilog implementation would need to be created, and it would be interesting to attempt to design more complicated instructions such as the scalar or vector products of two vectors.

References

[1] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual*, CS Division, EECS Department, University of California, Berkeley, 2019.

[2] K. Asanovic, Workshop Lecture, Topic: "The RISC-V Vector ISA Tutorial", CS Division, EECS Department, University of California, Berkeley, Berkeley, CA, May., 2018.

[3] P. Papaphilippou, P. H. J. Kelly, and W. Luk, "Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions", *Fifth Workshop on Computer Architecture Research with RISC-V*, June, 2021. Available: https://arxiv.org/abs/2106.07456.