RISC-V Vector Instructions vs ARM and x86 SIMD

Is old Cray-1 style vector machines coming back? What exactly is the difference between vector instructions and modern SIMD instructions?

Erik Engheim Follow

Jan 1 · 22 min read ★

In the 1980s, super-computers looked like what you see in the image below. The semi-circular shape of a Cray was synonymous with super computers in the 80s. That was just what a super computer looked like.

A Cray super computer from the 1980s.

What does this bygone era of supercomputing have to do with RISC-V? You see, Cray computers where what we call vector processing machines. Something that has long been considered a relic of the past.

Yet RISC-V is bringing Cray style vector processing back, even insisting it should replace SIMD (Single Instruction Multiple Data). Heresy?

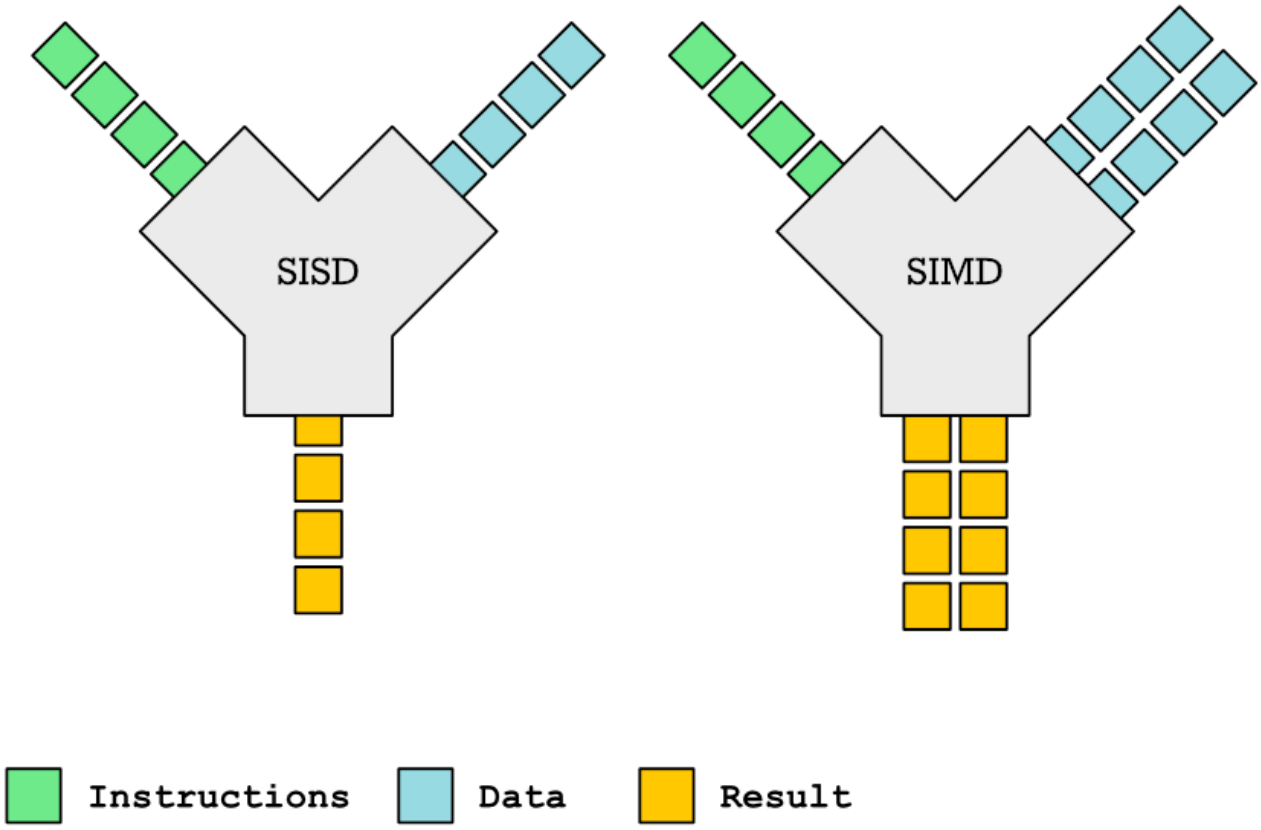
Such a bold and different strategy surely needs some explanation. Why are RISC-V designers taking a completely different route from their competitors x86, ARM, MIPS and others?

As usual we need a little detour to explain what exactly these technologies are and how they are different. Despite SIMD instructions coming last, I believe it is easier to grasp vector processing instructions by beginning with SIMD.

What is SIMD (Single Instruction Multiple Data)?

Most microprocessors whether x86 or ARM based provide what we call SIMD instructions in the microprocessors. You may have heard of MMX, SSE, AVX-2 and AVX-512. ARM has their own called Advanced SIMD and SVE.

What these instructions allow you to do is to apply the same operation to multiple elements. We can contrast this with SISD (Single Instruction Single Data), where an operation is only performed between single elements. The diagram below is a simple illustration of this:



Single Instruction Single Data (SISD) vs Single Instruction Multiple Data (SIMD)

We can write some simple code to illustrate the difference, below is an example of SISD. We can also call it operations on scalars (single values):

```
3 + 4 = 7
4 * 8 = 32
```

SIMD is about operations on vectors (multiple values):

```
[3, 2, 1] + [1, 2, 2] = [4, 4, 3]
[3, 2, 1] - [1, 2, 2] = [2, 0, -1]
```

Let me get more concrete with some pseudo assembly code for doing SISD. In this case we want to add two arrays with two elements each. Each element is a 32 bit integer. One starts at address 14, and the other at address 24:

```
load  r1, 14
load  r2, 24
add   r3, r1, r2   ; r3 ← r1 + r2

load  r1, 18
load  r2, 28
add   r4, r1, r2   ; r4 ← r1 + r2
```

With SIMD we can load multiple values and and perform multiple additions:

```
vload.32 v1, 14
vload.32 v2, 24
vadd.i32 v3, v1, v2   ; v3 ← v1 + v2
```

It is common to prefix vector and SIMD instructions with a `v` to separate them from scalar instructions. Conventions varies but this is inspired by ARM, where the `.32` suffix means we are loading in multiple 32-bit values. Assuming our vector registers `v1` and `v2` are 64-bit, then that means two elements are loaded each time.

The `vadd` instruction has the `.i32` suffix to indicate that we are adding 32-bit signed integer numbers. We could have used `.u32` to indicate unsigned integers.

Of course this is an entirely unrealistic example, as nobody would use SIMD for that few elements. More realistic we would operate on 16 elements.

How Does SIMD Work?

Ok we have a high level description of how SIMD instructions work. But in practical terms how are they handled at the CPU level? What is going on inside a CPU when it performs SIMD instructions?

Knowledge relevant to topic: How Does a Modern Microprocessor Work?.

Don't worry if you don't want to bury yourself in my microprocessor article. We will to a short version here skipping some details. Below you see a very simplified diagram of a RISC microprocessor.

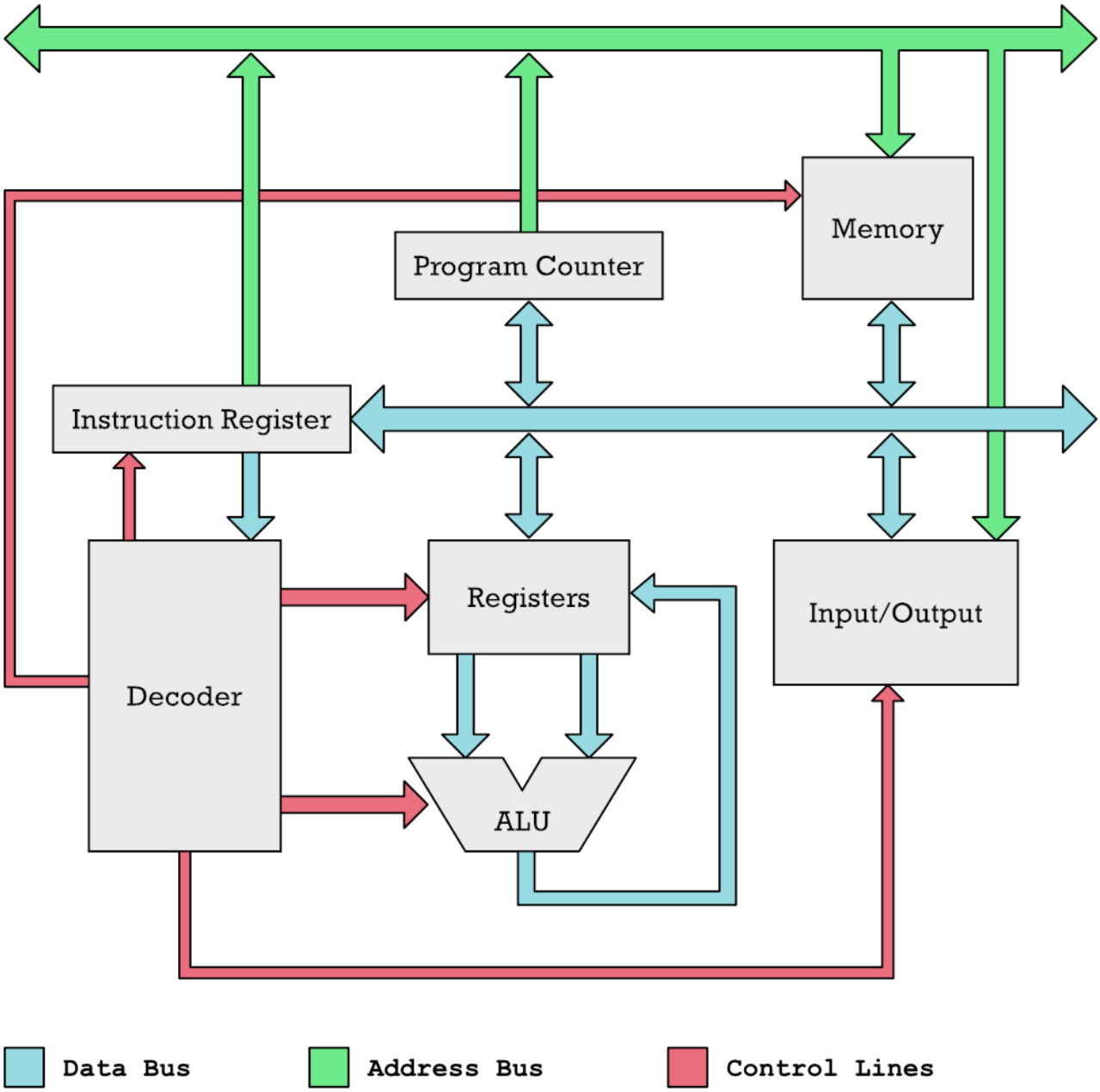
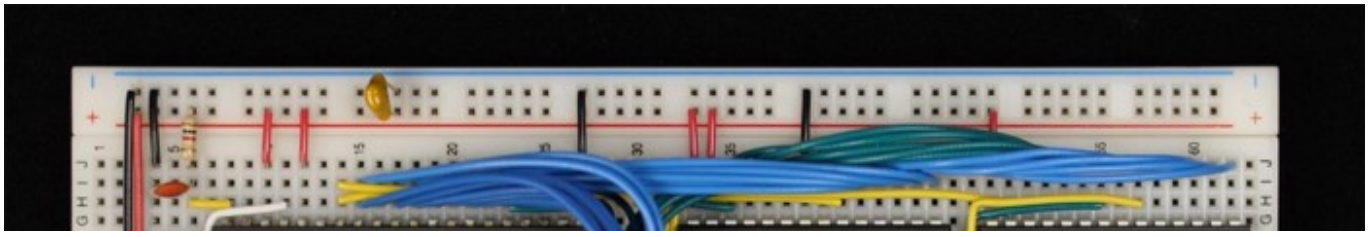
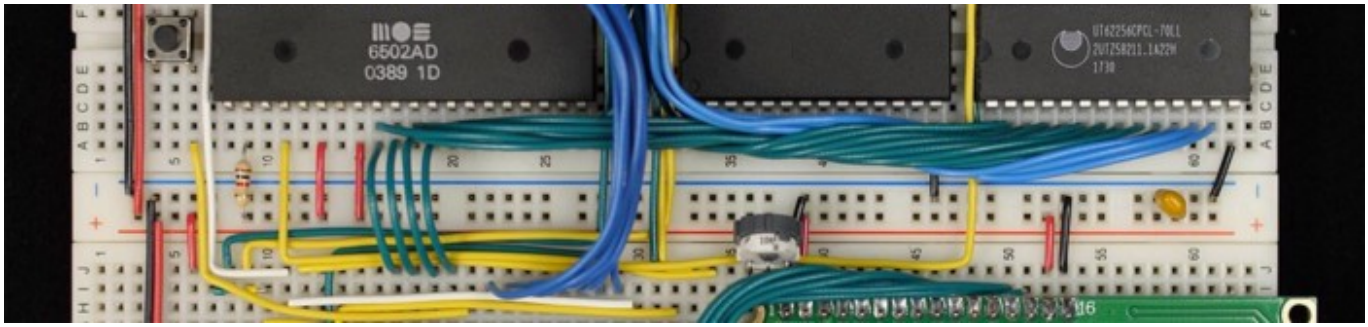


Diagram of a simple RISC microprocessor.

You can think of the colored bars as pipes for pushing data through to different parts of the CPU. Our main interest here is the blue ones which push the data we operate on as well as instructions through the system. The green pipes are address locations for memory cells.

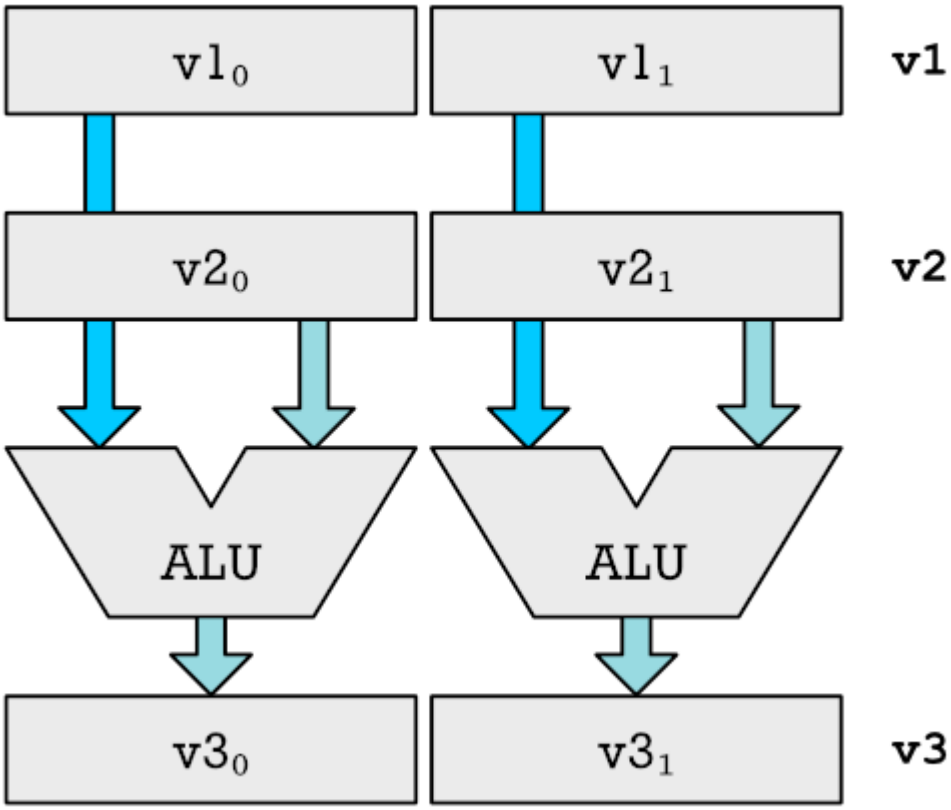




A 6502 computer built on a breadboard by Ben Eater. The colored wires are the data and address busses as well as the control lines.

In a simple microprocessor you only have one single Arithmetic Logic Unit (ALU). An example of such a processor would be the 6502, used in Commodore 64. The ALU is like the calculator of the CPU. It can add and subtract numbers. It takes two numbers as input, and then add or subtract them and spit the output out at the bottom. Inputs comes from register and output goes back to a register (memory cells with holding numbers you operate on).

To turn our CPU into a SIMD executing monster that can chew through dozens of numbers at the same time, we need to make some changes. Below is a simplified example of an upgrade that allows the addition to two numbers at the same time. Please note we are only showing the part related to the registers and ALUs.



How multiple ALUs are used to allow SIMD execution.

`v1`, `v2` and `v3` are what we call vector registers. They are chunked into different parts shown as `v1₀` and `v1₁`. We can feed each part, or element of the vector, into a separate ALU. This allows us to perform multiple additions at the same time. For a real CPU, we don't just add one extra ALUs. We add a dozen. Actually we get more crazy. We add a dozen multipliers and other functional units capable of doing all the different operations of a CPU. For very simple CPUs you don't have multipliers because you can simulate multiplication through repeated additions and shifts (adding and removing digits).

How we got SIMD

So how did these SIMD instructions come about? The need for fast image processing was the starting point. Each pixel in an image is made up of four 8-bit values (RGBA) which needs to be treated as separate numbers. It is slow to add those values separately for millions of pixels. SIMD instructions was an obvious way to boost performance of such tasks.

Each pixel is made up of four components: Red, Green, Blue and Alpha value. Each is a byte which should be calculated on separately. If a 32-bit register is a vector register with 4 components we can do that.

SIMD is also used inside GPUs as they will add position vectors, multiply matrices. Composite pixel color values etc.

Benefits of SIMD

It is difficult to parallelize the execution of code. But performing the same operation on multiple elements of data is fairly straight forward when dealing with things such as images, geometry, machine learning and a lot of scientific computing.

SIMD thus gives us a way of easily speeding up these calculations. If you can e.g. add 8 numbers by just executing 1 instruction, then you basically have a 8x speedup. Hence it is not surprising that x86 and ARM microprocessors have piled on a ton of SIMD instructions over the years.

And GPUs basically contain a bucket load of cores doing lots of SIMD computation. This is what has increased graphics performance a lot and why scientific code is increasingly using GPUs.

But if SIMD is so awesome, why are the RISC-V ditching it and going for Vector processing instead? Or more specifically instead of adding a SIMD instruction-set extension, they are adding a Vector instruction-set extension.

The Problem With SIMD Instructions

RISC-V designers David Patterson and Andrew Waterman wrote an article: SIMD Instructions Considered Harmful.

It is an interesting read, but it gets into a lot more technical depth than I will here. Patterson and Waterman describe the problem:

> *Like an opioid, SIMD starts off innocently enough. An architect partitions the existing 64-bit registers and ALU into many 8-, 16-, or 32-bit pieces and then computes on them in parallel. The opcode supplies the data width and the operation. Data transfers are simply loads and stores of a single 64-bit register. How could anyone be against that?*

Here is the kicker:

> *The IA-32 instruction set has grown from 80 to around 1400 instructions since 1978, largely fueled by SIMD.*

For this reason the specifications and manuals for x86 and ARM are enormous. In contrast you can get an overview of all the most important RISC-V instructions on a single double sided sheet of paper. This has implications for those creating the chips in silicon as well as those making assemblers and compilers. Support for SIMD instruction often get added late.

The designers of RISC-V wants a practical CPU instruction-set which can be used for teaching for a long time. E.g. until RISC-V they used MIPS which stopped being important in the industry long time ago. Academia prefers to not base their teaching on fads and hypes of the industry. Universities emphasize teaching knowledge with long durability. That is why they are more likely to teach to data structures and algorithm rather than say how to use a debugging tool or and IDE.

Thus SIMD as it has developed is untenable. There are new instructions every few years. Nothing is very durable. Thus Patterson and Waterman argue:

> *An older and, in our opinion, more elegant alternative to exploit data-level parallelism is vector architectures. Vector computers gather objects from main memory and put them into long, sequential vector registers.*

A Return to Cray Style Vector Processing?

So the RISC-V designers have created an extension with vector instructions instead of SIMD instructions. But if this is so much better, why did it not happen earlier and why did vector processing fall out of favor in the past?

Before we can answer any of that, we need to actually understand what vector processing is.

Vector vs SIMD Processing

The best way of understanding the difference is by looking at some C/C++ code. In SIMD the vector are fixed size and treated as fixed length types like this:

```
struct Vec3 {
   int x0;
   int x1;
   int x2;
};

struct Vec4 {
   int x0;
   int x1;
   int x2;
   int x4;
};
```

This means that vector addition function are dealing with fixed lengths:

```
Vec3 vadd3(Vec3 v1, Vec3 v2) {
   return Vec3(v1.x0 + v2.x0,
               v1.x1 + v2.x1,
               v1.x2 + v2.x2);
}
```

We can think of `Vec3`, `Vec4` and `vadd3` as existing in hardware. Developers however want higher level functionality and may compose these operations to create more general purpose functions:

```
void vadd(int v1[], int v2[], int n, int v3[]) {
    int i = 0;
    while (i < n) {
        u = Vec3(v1[i], v1[i+1], v1[i+3]);
        v = Vec3(v2[i], v2[i+1], v2[i+3]);

        w = vadd3(u, v); // Efficient vector operation

        v3[i]   = w.x0;
        v3[i+1] = w.x1;
        v3[i+2] = w.x2;
        i += 3;
    }
}
```

You can think of this a bit as pseudo-code. The point to get across is that you can build functionality to process vectors of any length upon functions which processes smaller fixed length vectors.

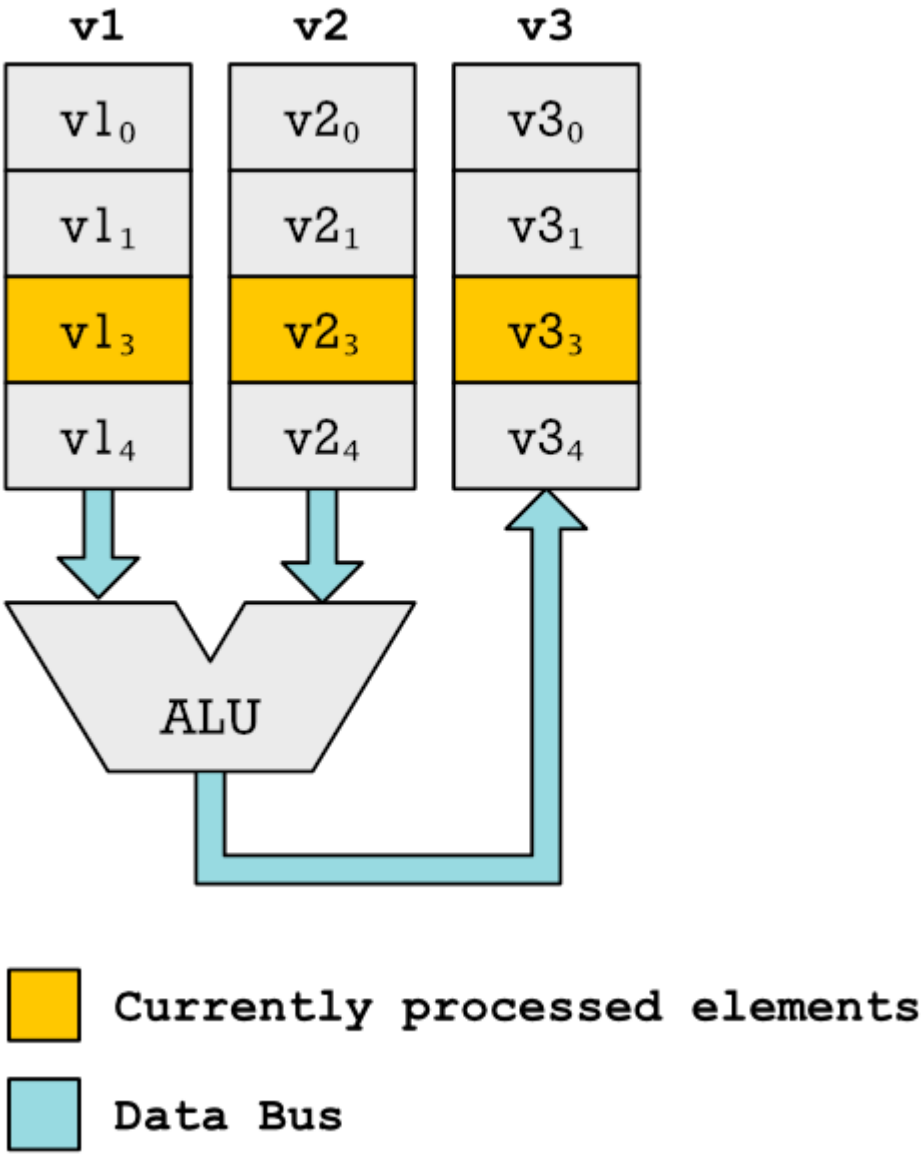
Vector processing as was done with old Cray super computers and which essentially is what the RISC-V guys are proposing is to put functions such as `vadd` into hardware.

What does that mean in practice?

Implementation of Vector Processing in Hardware

It means internally we still got SIMD units that operate on some fixed width vectors. However that is not what the assembly programmers sees. Instead just like with `vadd` the assembly code instructions are not tied to specific vector lengths. There are special status and control registers (CSR) which the programmer can set to the length of the vector he or she is operating on. This is kind of similar to how `vadd` takes the `n` argument specifying the length of the vector.

Instead we got some really long vectors. Significantly longer than the vector registers used by SIMD instructions. It could be hundreds of elements that could fit. It is not practical to create ALUs and multipliers for every one of these elements as we do with SIMD style vector registers.



A vector unit processes elements in sequence. The current elements processed are highlighted. Here we to single pairs of elements. But realistically we use multiple ALUs and process multiple elements in sequence.

Instead what happens when the CPU reads a `vadd` function is it starts looping over these large registers just like we did in the pseudo code example. Here is a code example:

```
vsetlen r1 , 16, 120   ; 120 element vector. Each element 16-bit
vload  v1, 14          ; Load 120 elements start at address 14
vload  v2, 134         ; Load elements starting at address 123
vadd   v3, v1, v2      ; Add all 120 elements.
vstore v3, 254         ; Store result at address 254
```

Before performing operations one has to configure the vector processor, by setting the number of elements in a vector and the size and type of each element. In this example I made it simple. We are always dealing with signed integers. But in a real system you have to be able to specify whether you are dealing with floating point number and signed or unsigned integers.

What `vload` does depends on the configuration. In this case we will load 120 elements, each 16-bit wide from memory.

`vadd` iterates over all 120 elements in both the `v1` and `v2` vector register. Each element is added and result written to register `v3`. To better understand how this works, let us talk about about the number of clock cycles involved.

A clock cycle is what a microprocessor needs to perform one simple task. It could be one clock cycle to decode an instruction, one to add two numbers. More complex operations such as multiplication could take multiple clock cycles.

Say behind the scenes we got four ALUs. Thus we can perform four additions every clock cycle. That means `vadd` will require 30 clock cycles to complete:

```
120/4 = 30
```

This may not sound great. Why not do the same directly using an assembly program that loops and performs these SIMD instructions directly. Why implement in hardware something that has to be done in an iteration anyway?

Benefits of Vector Processing

One major benefit is that we need much smaller programs. We don't need to write programs with multiple loads, comparisons and loops.

Patterson and Waterman make an example program for comparison in their article: SIMD Instructions Considered Harmful. Here is their observation of difference in program size:

> *Two-thirds to three-fourths of the code for MIPS-32 MSA and IA-32 AVX2 is SIMD overhead, either to prepare the data for the main SIMD loop or to handle the fringe elements when n is not a multiple of the number of floating-point numbers in a SIMD register.*

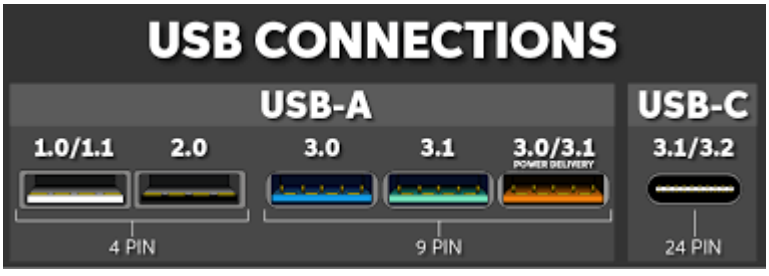
But more importantly with vector instructions you don't have to keep decoding the same instructions repeatedly. Performing repeated conditional branching etc. In the code examples Patterson and Waterman are using they remark that the SIMD programs require 10 to 20 times more instructions to be executed compared to the RISC-V version using vector instructions.

The reason being that a SIMD loop only process 2 to 4 elements on each iteration. In the vector code, one assumes the hardware supports vector registers with 64 elements. Thus batches of 64 elements are handled on each iteration, cutting down on the number of times one needs to iterate.

The max vector length can be queried at runtime, so one does not need to hardcode 64 element long batch sizes.

Decoding fewer instructions reduce power usage as decoding and fetching consume a fair amount of power.

In addition we achieve what all good interface design should strive towards: Hiding implementation details. Why is that important? Look at USB plugs? We love it when USB standard increase performance without physically changing the plug.



From USB-1 to USB-3 we avoided change to interface. We could use the same cables. Likewise Vector extensions let us keep the same interface while doing internal improvements.

When chip technology improves you are allowed to use more transistors. You can use this to add more ALUs and multiplier to handle more vector elements in parallel. For

CPUs with SIMD instructions that means adding a couple of hundred new instructions for the new vector lengths you can now handle.

Programs will have to be recompiled to handle these newly added long vectors to get a performance boost. With the RISC-V approach that is not necessary. The code looks the same. Only thing changing is that `vadd` will finish in fewer cycles because it has a larger SIMD unit allowing it to process a larger batch of elements each clock cycle.

If Vector Machines Are So Great Why Did They Get Abandoned?

Here is a piece of the puzzle I don't think David Patterson have explained that well in his previous writing. Cray vector processing machines basically died out.

Truck or Racing Car?

To understand why, we need to understand tradeoffs. If you want to transport a maximum amount of cargo from A to B, you can basically do it in two ways. Use a racing car to drive fast back and forth with small quantities of cargo.

Or you can use a large slow moving truck which can haul a large amount of cargo but which move slowly.

Small amount of data delivered quickly or large amounts of data delivered slowly?

Most general purpose software is best served by a racing car. General purpose programs are not easily serializable. What data they need depends on instructions executed. There

are all sort of conditional branches and random access of memory to take into account. You simply cannot pick up a lot of cargo (data) on each visit to the warehouse (memory) because you don't know what will be required further down the line.

Thus general purpose microprocessors tend to have large fast memory caches, so the CPU can quickly get what it needs when it needs it (racing car analogy).

Vector processor in contrast work much like GPUs. They did not deal with general purpose programs. Usually they where used for scientific software such as weather simulations where you need large amounts of data which can be processed in parallel. GPUs likewise can process lots of pixels or coordinates in parallel.

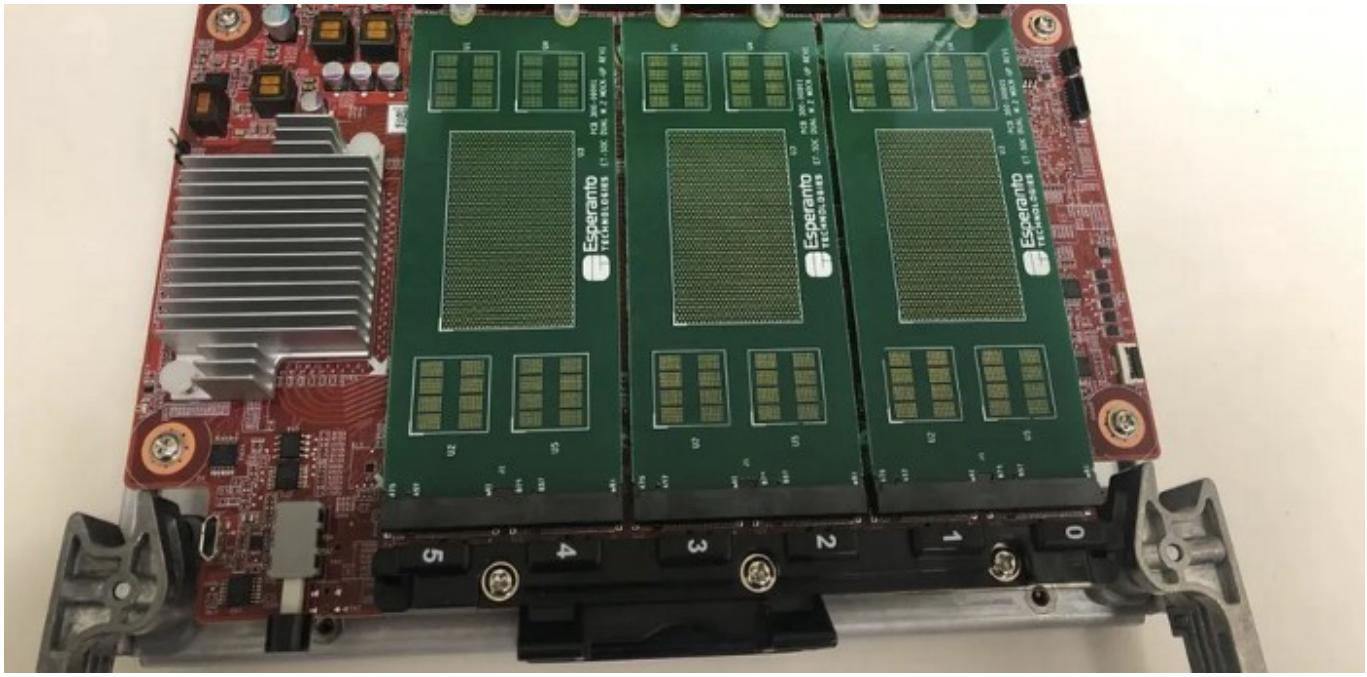
Thus you don't need to move fast, because you can pick up huge chunks of data each time. Thus GPUs and Vector machines typically have low clock frequency and small caches. Instead their memory system is setup to fetch a lot of data in parallel. In other words, they move data like a truck moves cargo. A lot at a time, but slowly.

Vector machines actually have data in pipelines because it is very predictable what the next data needed will be.

Vector Processors are Tiny

Sure you could up the clock frequency of vector processors and give them fat caches, but what is the point when you got much better choices available? All the transistors you don't spend on cache you can use to expand ability to process more elements in parallel. Also watt usage and heath doesn't grow linearly with clock frequency. It grows much faster. Thus to keep thermal budget down it pays to keep clock frequency low.

If you look at the ET-SoC-1 solution from Esperanto Technologies you see all of these tradeoffs taken into consideration. Their SoC is about the same size as Apple's M1 SoC in terms of number of transistors. Yet a vector processing core require a lot less silicon because we are not aiming for high single thread performance. The M1 Firestorm cores are fat beasts because they ned a lot of transistors to implement Out-of-Order Execution (OoOE), branch prediction, deep pipelines and many other things to get single thread performance to scream.



Board with 6 ET-SoC-1 chips. Image: Enterpriseai

The ET-SoC-1 in contrast can fit over 1000 RISC-V CPU cores implementing the Vector instruction extension. That is because vector processor can be made really tiny:

- Cache needs are minimal.
- They are in order, so you save a lot of silicon by not implementing complex OoOE controller logic.
- Lower clock frequency simplifies a lot.

Thus if you can describe your problem as operations on large vectors, then you can get some crazy performance gains using the same number of transistors by going for a vector machine design.

Vector Processors Suck At General Purpose Computing

But here is the kicker: If your program cannot be expressed that way you are in for a world of hurt. Executing regular desktop software which does not operate on large vectors will get terrible performance. Why?

Your clock frequency is low. You got no OoOE, and your cache is tiny. Hence every instruction which needs to get some data has to wait a long time. This was the problem for Cray. They where simply not usable for general purpose computing. Due to the rest of the market using more traditional CPUs those got cheaper to make and a lot of the kind

of software Cray computer ran could be done well by simply running on multicore machines, using clusters or whatever.

When regular computers started to need vector processing, this was for multimedia applications. Image processing and stuff like that. In this case you typically work with small short vectors. Then SIMD instruction was an obvious and simple solution. They are pretty straight forward to setup. Just add some vector registers and operations to work on them. Vector instructions require much more thought and planning. You need ways to setup vector length and element types. Large vectors are impractical to save and restore when switching between programs. And anyway these programs didn't need long vectors.

Thus there was initially no obvious advantage to vector extensions over SIMD. Because vector processing is not great for general purpose computing the ET-SoC-1 developed by Esperanto Technologies e.g. has four fat RISC-V cores, called ET-Maxion, designed for general purpose computing. These are more like the M1 Firestorm cores:

- Larger caches
- Smart branch predictor
- Multiple instruction decoders
- Out of Order Execution

These will run the operating system and schedule work tasks to the smaller RISC-V cores (ET-minion) with vector extensions. This might be the type of architectural choices we will see more of: Mixing of different kinds of cores with different strengths.

General purpose computing cannot really benefit that much for having lots of cores. However for specialized tasks it is much better to use unconventional cores than large cores made for general purpose computing.

For instance a vector processor does all these tasks really well:

- Machine learning
- Compression both for images, zip files etc

- Cryptography
- Multimedia: audio, video
- Speech and handwriting
- Networking. Parity checks, checksums
- Databases. Hash/joins

Thus given a budget of X number of transistors, to speed up these tasks you would better of opting for a vector processor design than adding more large general purpose cores.

Thus we are really back to the argument I made in: Apple M1 foreshadows Rise of RISC-V.

As well as my talk of heterogenous computing in: Why Is Apple's M1 Chip So Fast?

In both cases I was talking about how Apple was getting a speed boost from their M1 chips by using specialized coprocessors. And that is what this is about as well. In principle we could simply add the vector extension to any RISC-V general purpose CPU. But you could could choose to tailor if for vector processing by removing the large cache, kick the Out-of-Order Execution Unit to the curb, reduce clock frequency, go with much simpler branch predictor, and widen the memory access (read more data in at a time into a data pipeline).

If you do that you get a much smaller chip with much lower watt usage that will likely perform better for vector processing than a fatter chip made for fast general purpose computing. Because it is small and low watt, you can have a lot of them.

This is really just another way to validate the case for specialized coprocessor that I made earlier.

Why Vector Processors Matter Again

So this takes us a full loop back to what we have really been trying to answer this whole time. Why vector instruction now make sense, but got abandoned in the past.

The question has already been partially answered. The SIMD approach has painted us into a corner. But more importantly our computers are now doing a lot more variety of

tasks. In particular Machine Learning has become huge. This has become a major focus in data centers. It is not without reason that Apple added the Neural Engine on their iPad and iPhone Apple Silicon chips.

It is not without reason that Google made their on Tensor Processing Units (TPU) to offer higher speed machine learning in the crowd. Processing really large arrays is back in business thanks to the rise of deep learning.

Thus in my opinion, that RISC-V is going for Vector extensions instead of SIMD extensions is a very smart move. SIMD came about in a world where the need was primarily needed for short vectors in multimedia settings. We are no longer in that world. Vector extensions kills two birds with one stone:

- Vector instructions don't bloat the ISA. We don't need to keep adding new ones.
- Far more future proof.
- No recompile needed after we have added more ALUs, multipliers and other functional units.
- They are an excellent choice for machine learning applications.

Criticism of Vector Instructions

Of course not everyone share my enthusiasm for vector instructions. We got to look at some of the criticism. One accusation often seen is that the whole system is way more complicated and that SIMD is much easier. The belief is that vector extensions will bloat the chips.

This is frankly not criticism we should take serious. Esperanto Technologies have already demonstrated that they can make small efficient chips with the RISC-V vector extensions.

David Patterson himself is not a newbie in this area. He knows what he is doing. Not only was he one of the key architects behind the first RISC processor, but he was also actively involved in another less well known project in the 1990s called the IRAM project.

This was in many ways an alternative to RISC, where one took a vector processing approach instead. In fact the RISC-V guys may be more influenced by having worked last

on vector processing than having invented the original RISC. Patterson and others really began believing in the power and elegance of vector processing from their IRAM project. Hence the *V* in RISC-V actually stands for both 5 and for *Vector*. RISC-V was from the beginning conceived as an architecture for vector processing.

The IRAM project is an interesting read as it foreshadows a lot of what later has happened with Apple's M1 chip. In talks extensively about using DRAM on an SoC and the advantages of that. Hint think Unified Memory.

Asked about the complexity and difficulty of using vector processing instructions Patterson has written:

> *It's not that hard. We did this in the IRAM project long ago with parallel execution of smaller data types. Many are building RISC-V processors with this style of vector architecture, and having a mask that says which elements at the end of the vector are written is not rocket science.*

Passing Data Between Vector Functions

But here is perhaps a more serious criticism. You can easily create concrete vector types of fixed length and element type which you can utilize in a higher level language. Thus you can create a series of functions which can get argument data passed through vector registers. Thus we can combine multiple functions where all the data is passed between them using vector registers.

For instance if functions look like this, we can easily have this work with other functions taking `Vec3` arguments:

```
Vec3 vadd3(Vec3 v1, Vec3 v2) {
   return Vec3(v1.x0 + v2.x0,
               v1.x1 + v2.x1,
               v1.x2 + v2.x2);
}
```

If vector extensions require that you pass data as arrays or pointers to memory then that is much harder. It would imply that data between two vector functions will always have

to exchange data by going throw main memory. That will surely slow things down. This is thrust of the argument:

> *The approach you're advocating for can't quite do that. You can't invent a sane calling convention for a function that takes or returns kilobytes of data. Current architectures all pass arguments and return values in these vector registers, because their count and size are part of the ISA, i.e. stable and known to compilers.*

I can however see some ways around this. For instance with GPU programming people use CUDA arrays which can be of any length. These are really just wrappers around handlers to arrays in graphics memory. I don't see why Vector registers cannot work the same way. You simply use special array types as arguments to these vector functions.

If this isn't possible then Just in Time Compilation may actually be a good fit. E.g. in the Julia programming language, the interface between different function calls is often eliminated by the Just in Time Compiler. One could imagine a JIT merging together several vector based functions in this manner to avoid writing result out to memory before all the processing is finished.

But honestly this is an area I would love to hear some comments on.

Why Not Use a GPU?

The last criticism I have seen is that if you need to operate on long vectors, you should simply use a GPU instead. The idea here is that if you need to operate on large chunks of data, then you can also likely afford the performance hit of transferring large chunks of data to the GPU for processing and then reading back the result.

There are two ways of looking at criticism. If vector extensions where only used for fat general purpose CPUs it would have some merit. A CPU optimized for operating on scalar data fast, is going to have disadvantage in vector processing anyway.

However we cannot assume this. As we have seen with Esperanto Technologies, it is perfectly valid to design specialized RISC-V processors such as the ET-Minion which is tailor mode for fast vector processing.

And if we are to believe Esperanto Technologies' claim their solution would outperform a GPU based solution for machine learning.

This is in general a problem I see with a lot of RISC-V criticism. It often misses the mark because it assumes we are always talking about a general purpose processor for running Windows, Linux or macOS. However RISC-V is supposed to be for anything from microcontrollers, coprocessor to super computers.

And the vector extension still makes sense for a general purpose CPU, simply because it hides implementation detail of the vector processing. Something SIMD doesn't do and which has created a lot of ISA bloat.

Benefits of Common Instruction-Set

Even if a specialized processor is better for long vector, that does't mean the general purpose CPU and the specialized core cannot both be RISC-V processor with vector extensions.

One of the problems with the novel Cell architecture of the Playstation 3 was that the general purpose PowerPC CPU did not share instruction set with the more limited coprocessors. The PS3 architecture had in fact a lot in common with the Esperanto ET-SoC-1. Instead of the ET-Maxion core as a general purpose CPU, the PS3 used a PowerPC CPU to run the operating system. Instead of the ET-Minion cores to handle the main vector oriented workload the PS3 used Synergistic Processing Elements (SPE).

It was an early attempt at doing what M1 is doing today and what ET-SoC-1 may do in the future.

While countless pages have been written about the failure of the PS3, one reason mentioned was that the central processor and the SPEs did not share instruction-set.

For a developer it would likely have been much easier to develop and troubleshoot vector processing code on the main PowerPC CPU first. Thus having a RISC-V vector extension on a general purpose CPU can be useful for developers and for troubleshooting.

Final Remarks

I hope this was interesting. This got a lot longer than planned and I will have to contemplate a shorter version of this one. But then again that was my belief about my

M1 story, and that proved wrong. Would be useful to hear feedback from anyone who can make their way all the way to this final remark 😉

Risc V Cpu Arm X86 Simd