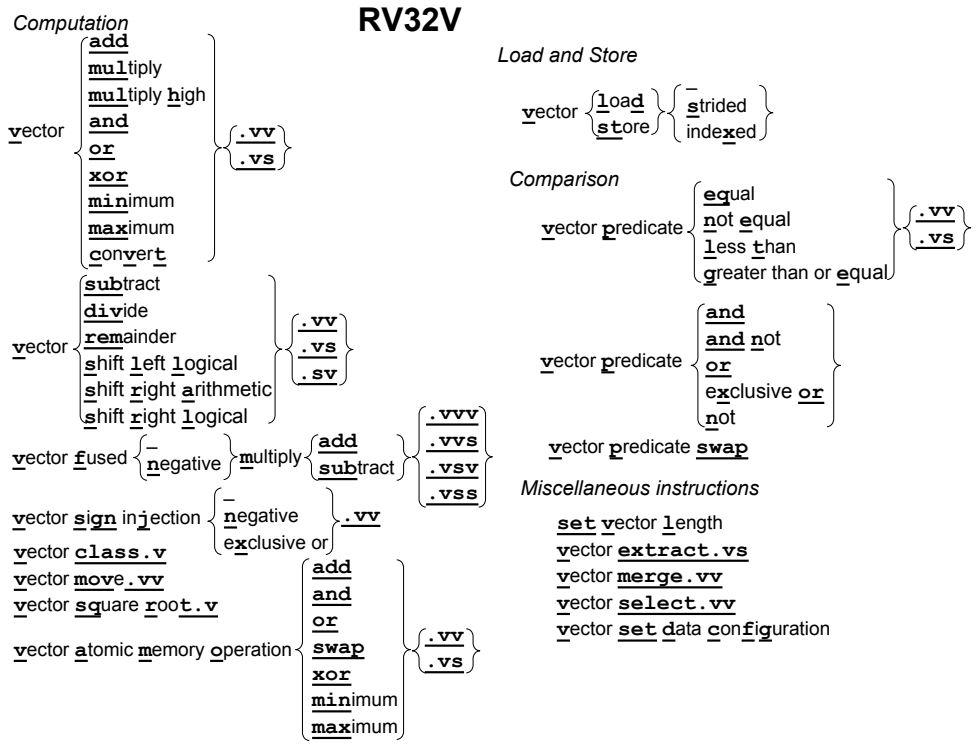
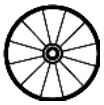


Figure 8.1: Diagram of the RV32V instructions. Because of dynamic register typing, this instruction diagram also works without change for RV64V in Chapter 9.

Vector architectures are rarer than SIMD architectures, so fewer readers know vector ISAs. Thus, this chapter will have a more tutorial flavor than earlier ones. If you want to dig deeper into vector architectures, read Chapter 4 and Appendix G of [Hennessy and Patterson 2011]. RV32V also has novel features that simplify the ISA, which requires more explanation even if you already are familiar with vector architectures.



8.2 Vector Computation Instructions

Figure 8.1 is a graphical representation of the RV32V extension instruction set. The RV32V encoding has not been finalized, so this edition does not include the usual instruction-layout diagram.

Virtually every integer and floating-point computation instruction from an earlier chapter has a vector version: Figure 8.1 inherits operations from RV32I, RV32M, RV32F, RV32D, and RV32A. There are several types of each vector instruction depending on whether the source operands are all vectors (`.vv` suffix) or a vector source operand and a scalar source operand (`.vs` suffix). A scalar suffix means an `x` or `f` register is an operand along with a vector register (`v`). For example, our DAXPY program (Figure 5.7 on page 55 in Chapter 5) calculates $Y = a \times X + Y$, where X and Y are vectors, and a is a scalar. For vector-scalar operations, the rs1 field specifies the scalar register to be accessed.

Asymmetric operations like subtraction and division offer a third variation of vector in-

structions where the first operand is scalar and the second is vector (`.sv` suffix). Operations like $Y = a - X$ use them. They are superfluous for symmetric operations like addition and multiplication, so those instructions have no `.sv` version. The fused multiply-add instructions have three operands, so they have the largest combination of vector and scalar options: `.vvv`, `.vvs`, `.vsv`, and `.vss`.

Readers may notice that Figure 8.1 ignores the data type and width of the vector operations. The next section explains why.

8.3 Vector Registers and Dynamic Typing

RV32V adds 32 vector registers, whose names start with `v`, but the number of *elements* per vector register varies. That number depends on both the width of the operations and on the amount of memory dedicated to vector registers, which is up to the processor designer. For example, if the processor allocated 4096 bytes for vector registers, that is enough for all 32 vector registers to have 16 64-bit elements, 32 32-bit elements, 64 16-bit elements, or 128 8-bit elements.

To keep the number of elements flexible in a vector ISA, a vector processor calculates the *maximum vector length* (`mvl`) that programs use to run properly on processors with differing amounts of memory for vector registers. The vector length register (`vl`) sets the number of elements in a vector for a particular operation, which helps programs when a dimension of an array is not a multiple of `mvl`. We'll demonstrate `mvl`, `vl`, and the eight predicate registers (`vp`*i*) in more detail in the following sections.

RV32V takes the novel approach of associating the data type and length with the *vector registers* rather than with the *instruction opcodes*. A program tags the vector registers with their data type and width before executing the vector computation instructions. *Dynamic register typing* slashes the number of vector instructions, important because there are often six integer and three floating-point versions of each vector instruction as Figure 8.1 shows. As we shall see in Section 8.9 when we confront the numerous SIMD instructions, a dynamically typed vector architecture reduces the cognitive load on the assembly language programmer and the difficulty of the compiler's code generator.

Another advantage of dynamic typing is that programs can disable unused vector registers. This feature allocates all the vector memory to the enabled vector registers. For example, suppose only two vector registers are enabled, they are type 64-bit floats, and the processor has 1024 bytes of vector register memory. The processor would halve the memory, giving each vector register 512 bytes or $512/8 = 64$ elements and therefore set `mvl` to 64. Thus, `mvl` is dynamic, but its value is set by the processor and cannot be directly changed by software.

The source and destination registers determine the type and size of the operation and the result, so conversions are implicit with dynamic typing. For example, a processor can multiply a vector of double-precision floating-point numbers by a single-precision scalar without first having to convert the operands to the same precision. This bonus benefit reduces the total number of vector instructions and the number of instructions executed.

The `vsetdcfg` instruction sets the vector register types. Figure 8.2 shows the vector register types available to RV32V plus more types for RV64V (see Chapter 9). RV32V requires that vector floating-point operations have the scalar versions also. Thus, you must have at least RV32FV to use the F32 type and RV32FDV to use the F64 type. RV32V introduces a 16-bit floating-point format type F16. If an implementation supports both RV32V and RV32F, then it must support both F16 and F32 formats.

Type	Floating Point		Signed Integer		Unsigned Integer	
Width	Name	`vetype`	Name	`vetype`	Name	`vetype`
8 bits	–	–	X8	10 100	X8U	11 100
16 bits	F16	01 101	X16	10 101	X16U	11 101
32 bits	F32	01 110	X32	10 110	X32U	11 110
64 bits	F64	01 111	X64	10 111	X64U	11 111

Figure 8.2: RV32V encodings of vector register types. The rightmost three bits of the field show the width of the data, and the two leftmost bits give its type. X64 and U64 are available only for RV64V. F16 and F32 require the RV32F extension and F64 requires RV32F and RV32D. F16 is the IEEE 754-2008 16-bit floating-point format (binary16). Setting `vetype` to 00000 disables the vector registers. (Table 17.4 of [Waterman and Asanović 2017] is the basis of this figure.)

■ ***Elaboration: RV32V can switch context quickly.***

One reason vector architectures were less popular than SIMD architectures was concern that adding large vector registers would stretch the time to save and restore a program on an interrupt, called a *context switch*. Dynamic register typing helps. The programmer must tell the processor which vector registers are being used, which means processor needs to save and restore only those registers on a context switch. The RV32V convention is to disable *all* vector registers when the vector instructions aren't being used, which means a processor can have the performance benefit of vector registers but pay the extra context switch time only if an interrupt occurs while the vector instructions are executing. Earlier vector architectures had to pay the worst-case context switch cost of saving and restoring all vector registers whenever an interrupt occurred.

Concern about slow context switch times led Intel to avoid adding registers in the original MMX SIMD extension. It simply reused the existing floating-point registers, which meant no extra context to switch, but a program couldn't intermix floating-point and multimedia instructions.

8.4 Vector Loads and Stores

The easiest case for vector loads and stores is dealing with single-dimension arrays that are stored sequentially in memory. Vector load fills a vector register with data from sequential addresses in memory starting with the address in the `vld` instruction. The data type associated with the vector register determines the size of the data elements and the vector length register `vl` sets the number of elements to load. Vector store `vst` does the inverse operation of `vld`.

Each load and store has a 7-bit unsigned immediate offset that is scaled by the element type in the destination register for loads and the source register for stores.

For example, if `a0` has 1024, and the type of `v0` is X32, then `vld v0, 0(a0)` will generate the addresses 1024, 1028, 1032, 1036, ... until reaching the limit set by `vl`.

For multi-dimension arrays, some accesses will not be sequential. If stored in row major order, sequential column accesses in a two-dimensional array want data elements separated by the size of the row. Vector architectures support these accesses with *strided* data transfers: `vlds` and `vsts`. While one could get the same effect as `vld` and `vst` by setting the stride to the size of the element in `vlds` and `vsts`, `vld` and `vst` guarantee that all accesses will be sequential, which makes it easier to deliver high memory bandwidth. Another reason is that providing `vld` and `vst` reduces code size and instructions executed for the common case of unit stride. These instructions specify two source registers, with one giving the starting address and the other specifying the stride in bytes.

For example, assume the starting address in `a0` was address 1024, and the size of a row in `a1` was 64 bytes. `vlds v0,a0,a1` would send this sequence of addresses to memory: 1024, 1088 (1024 + 1 × 64), 1152 (1024 + 2 × 64), 1216 (1024 + 3 × 64), and so on until the vector

length register `vl` tells it to stop. The returning data is written into sequential elements of the destination vector register.

Thus far, we have assumed that the program is working with dense arrays. To support sparse arrays, vector architectures offer *indexed* data transfers: `vldx` and `vstx`. One source register for these instructions refers to a vector register and the other to a scalar register. The scalar register has the starting address of the sparse array, and each element of the vector register contains the index in bytes of the nonzero elements of the sparse array.

Indexed load is also called *gather* and indexed store is often named *scatter*.

Suppose the starting address in `a0` was address 1024, and vector register `v1` had these byte indices in the first 4 elements: 16, 48, 80, 160. `vldx v0,a0,v1` would send this sequence of addresses to memory: 1040 (1024 + 16), 1072 (1024 + 48), 1104 (1024 + 80), 1184 (1024 + 160). It loads the returning data into sequential elements of the destination vector register.

We used sparse arrays as our motivation for indexed loads and stores, but there are many other algorithms that access data indirectly via a table of indices.

8.5 Parallelism During Vector Execution

While a simple vector processor might execute one vector element at a time, element operations are independent by definition, and so a processor could theoretically compute all of them simultaneously. The widest data for RV32G is 64 bits, and today's vector processors typically execute two, four, or eight 64-bit elements per clock cycle. Hardware handles the fringe cases when the vector length is not a multiple of the number of the elements executed per clock cycle.

Like SIMD, the number of smaller data operations is the ratio of the widths of the narrow data to the wide data. Thus, a vector processor that computes 4 64-bit operations per clock cycle would normally launch 8 32-bit, 16 16-bit, and 32 8-bit operations per clock cycle.

In SIMD, the ISA architect determines the maximum number of data parallel operations per clock cycle *and* the number of elements per register. In contrast, the RV32V processor designer picks both of them without having to change the ISA or the compiler, while every doubling of SIMD register width doubles the number of SIMD instructions and requires changes to the SIMD compilers. This hidden flexibility means the identical RV32V program runs without change on the simplest and most aggressive vector processors.

8.6 Conditional Execution of Vector Operations

Some vector computations include `if` statements. Rather than rely on conditional branches, vector architectures include a mask that suppresses operations on some elements of a vector operation. The predicate instructions in Figure 8.1 perform conditional tests between two vectors or a vector and scalar and writes into each element of the vector mask a 1 if the condition holds or a 0 otherwise. (The vector mask must have the same number of elements as the vector registers.) Any subsequent vector instruction can use that mask, with a 1 in bit i means that element i is changed by vector operations, and a 0 means that element i is unchanged.

A program is called *vectorizable* if most operations are performed by vector instructions. Gather, scatter, and predicate instructions increase the number of vectorizable programs.

RV32V provides 8 *vector predicate registers* (`vp`*i*) to act as vector masks. The instructions `vpand`, `vpandn`, `vpor`, `vpxor`, and `vpnot` perform logical instructions to combine them together to allow efficient processing of nested conditional statements.

RV32V instructions specify either `vp0` or `vp1` to be the mask that controls a vector operation. To perform a normal operation on all elements, one of those two predicates registers

must be set to all ones. To swap one of the other six predicate registers quickly into vp0 or vp1, RV32V has the vpswap instruction. The predicate registers are also enabled dynamically, and disabling them clears all the predicate registers quickly.

For example, suppose all the even-numbered elements of vector register v3 were negative integers and all the odd-numbered elements were positive integers. The result of this code:

```
vplt.vs     vp0,v3,x0  # set mask bits when elements of v3 < 0
add.vv,vp0  v0,v1,v2   # change elements of v0 to v1+v2 when true
```

would set all the even bits of vp0 to 1, all the odd bits to 0, and would replace all the even elements of v0 with the sum of the corresponding elements of v1 and v2. The odd elements of v0 would be unchanged.

8.7 Miscellaneous Vector Instructions

Adding to the instruction that configures the data types of vector registers mentioned above (vsetdcfg), setvl sets the vector length register (vl) and the destination register with the smaller of the source operand and the maximum vector length (mvl). The reason for picking the minimum is to decide in loops whether the vector code can run at the maximum vector length (mvl) or it must run at a smaller value to cover the remaining elements. Thus, to handle the tail, setvl is executed every loop iteration.

RV32V also has three instructions that manipulate elements within a vector register.

Vector select (vselect) produces a new result vector by gathering elements from one source data vector at the element locations specified by the second source index vector:

```
# vindices holds values from 0..mvl-1 that select elements from vsrc
vselect vdest, vsrc, vindices
```

Thus, if the first four elements of v2 contain 8, 0, 4, 2, then vselect v0,v1,v2 will replace the zeroth element of v0 with eighth element of v1, the first element of v0 with the zeroth element of v1, the second element of v0 with the fourth element of v1, and the third element of v0 with the second element of v1.

Vector merge (vmerge) resembles vector select, but it uses a vector predicate register to choose which of the sources to use. It produces a new result vector by gathering elements from one of two source registers depending on the predicate register. The new element comes from vsrc1 if the predicate vector register element is 0 or from vsrc2 if it is 1:

```
# vp0 bit i determines whether new element i for vdest
# comes from vsrc1 (if bit i == 0) or vsrc2 (if bit i == 1)
vmerge,vp0 vdest, vsrc1, vsrc2
```

Thus, if the first four elements of vp0 contain 1, 0, 0, 1, the first four elements of v1 contain 1, 2, 3, 4, and the first four elements of v2 contain 10, 20, 30, 40, then vmerge,vp0 v0,v1,v2 will make the first four elements of v0 be 10, 2, 3, 40.

The vector extract instruction takes elements starting from the middle of one vector and places these at the beginning of a second vector register:

```
# start is scalar reg holding element starting number of vsrc
vextract vdest, vsrc, start
```

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0             # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0           # vl = t0 = min(mvl, n)
  c:  vld    v0, a1           # load vector x
 10:  slli   t1, t0, 3        # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2           # load vector y
 18:  add    a1, a1, t1       # increment C pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1  # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0       # n -= vl (t0)
 24:  vst    v1, a2           # store Y
 28:  add    a2, a2, t1       # increment C pointer to y by vl*8
 2c:  bnez   a0, loop         # repeat if n != 0
 30:  ret                     # return
```

Figure 8.3: RV32V code for DAXPY in Figure 5.7. The machine language is missing because the RV32V opcodes are yet to be defined.

For example, if vector length `vl` is 64 and `a0` contains 32, then `vextract v0,v1,a0` will copy the last 32 elements of `v1` into the first 32 elements of `v0`.

The `vextract` instruction assists reductions by following a recursive-halving approach for any binary associative operator. For example, to sum all the elements of a vector register, use vector extract to copy the last half of a vector into the first half of another vector register and halve the vector length. Next, add these two vector registers together and repeat the recursive-halving with their sum until vector length equals 1. The result in the zeroth element will be the sum of all the original elements in the vector register.

8.8 Vector Example: DAXPY in RV32V

The V in RISC-V is also for vector. The RISC-V architects had extensive positive experience with vector architectures and were frustrated that SIMD dominated microprocessors. Hence, the V is for the fifth Berkeley RISC project *and* because their ISA would highlight vectors.

Figure 8.3 shows the RV32V assembly language for DAXPY (Figure 5.7 on page 55 in Chapter 5), which we'll explain a step at a time.

RV32V DAXPY starts by enabling the vector registers needed for this function. It requires only two vector registers to hold portions of `x` and `y`, which are double-precision floating-point numbers each 8 bytes wide. The first instruction creates a constant and the second writes it to the control status register that configures vector registers (`vcfgd`) to get two registers of type F64 (see Figure 8.2). By definition, the hardware allocates the configured registers in numerical order, yielding `v0` and `v1`.

Let's assume our RV32V processor has 1024 bytes of memory dedicated to vector registers. The hardware allocates the memory evenly between the two vector registers, which hold double-precision floating-point numbers (8 bytes). Each vector register has $512/8 = 64$ elements, so the processor sets the maximum vector length (`mvl`) for this function to 64.

The first instruction in the loop sets the vector length for the following vector instructions. The instruction `setvl` writes the smaller of the `mvl` and `n` into `vl` and `t0`. The insight is that if the number of iterations of the loop is larger than `n`, the fastest the code can crunch the data is 64 values at time, so set `vl` to `mvl`. If `n` is smaller than `mvl`, then we can't read or write beyond the end of `x` and `y`, so we should compute only on the last `n` elements in this final

iteration of the loop. `setvl` also writes to `t0` to help with later loop bookkeeping at location 10.

Vector architectures without `setvl` have extra *strip-mining* code to set `vl` to the last `n` elements of the loop and to check if n is initially zero.

The instruction `vld` at address c is a vector load from the address of `x` in scalar register `a1`. It transfers `vl` elements of `x` from memory to `v0`. The following shift instruction `slli` multiplies the vector length by the width of the data in bytes (8) for later use in incrementing pointers to `x` and `y`.

The instruction at address 14 (`vld`) loads `vl` elements of `y` from memory into `v1` and the next instruction (`add`) increments the pointer to `x`.

The instruction at address 1c is the jackpot. `vfmadd` multiplies `vl` elements of `x` (`v0`) by the scalar `a` (`f0`) and adds each product to `vl` elements of `y` (`v1`) and stores those `vl` sums back into `y` (`v1`).

All that is to left is store the results in memory and some loop overhead. The instruction at address 20 (`sub`) decrements `n` (`a0`) by `vl` to record the number of operations completed in this iteration of the loop. The following instruction (`vst`) stores `vl` results into `y` in memory. The instruction at address 28 (`add`) increments the pointer to `y` and the following instruction repeats the loop if `n` (`a0`) is not zero. If `n` is zero, the final instruction `ret` returns to the calling site.

The power of vector architecture is that each iteration of this 10-instruction loop launches $3 \times 64 = 192$ memory accesses and $2 \times 64 = 128$ floating-point multiplies and additions (assuming that `n` is at least 64). That averages about 19 memory accesses and 13 operations per instruction. As we shall see in the next section, these ratios for SIMD are an order of magnitude worse.

8.9 Comparing RV32V, MIPS-32 MSA SIMD, and x86-32 AVX SIMD

We'll now see the contrast between how SIMD and vector executes DAXPY. If you tilt your head, you can see SIMD as a restricted vector architecture with short vector registers—eight 8-bit "elements"—but it has no vector length register and no strided or indexed data transfers.

ARM-32 has a SIMD extension called NEON but it doesn't support double-precision floating-point instructions, so it doesn't help DAXPY.

MIPS SIMD. Figure 8.5 on page 83 shows the MIPS SIMD Architecture (MSA) version of DAXPY. Each MSA SIMD instruction can operate on two floating-point numbers since the MSA registers are 128 bits wide.

Unlike RV32V, because there is no vector length register, MSA requires extra bookkeeping instructions to check for problem values of `n`. When `n` is odd, there is extra code to compute a single floating-point multiply-add since MSA must operate on pairs of operands. That code is found in locations 3c to 4c in Figure 8.5. In the unlikely but possible case when `n` is zero, the branch at location 10 will skip the main computation loop.

Such bookkeeping code is considered part of `strip mining` in vector architectures. As the caption of Figure 8.5 explains, the vector length register `vl` renders such SIMD bookkeeping code moot for RV32V. Traditional vector architectures need extra code to handle the corner case of n = 0. RV32V just makes vector instructions act like nops when n = 0.

If it doesn't branch around the loop, the instruction at location 18 (`splati.d`) puts copies of `a` in both halves of the SIMD register `w2`. To add scalar data in SIMD, we need to replicate it to be as wide as the SIMD register.

Inside the loop, the `ld.d` instruction at location 1c loads two elements of `y` into SIMD register `w0` and then increments the pointer to y. It then does the a load of two elements of `x` into the SIMD register `w1`. The following instruction at location 28 increments the pointer to x. The payoff multiply-add instruction at location 2c is next.

The (delayed) branch at the end of the loop tests to see if the pointer to `y` has been incremented beyond the last even element of `y`. If it hasn't, the loop repeats. The SIMD store in the delay slot at address 34 writes the result to two elements of `y`.

ISA	MIPS-32 MSA	x86-32 AVX2	RV32FDV
Instructions (static)	22	29	13
Bytes (static)	88	92	52
Instructions per Main Loop	7	6	10
Results per Main Loop	2	4	64
Instructions (dynamic, n=1000)	3511	1517	163

Figure 8.4: Number of instructions and code size of DAXPY for vector ISAs. It lists number of instructions total (static), code size, number of instructions and results per loop, and number of instructions executed (n = 1000). microMIPS with MSA shrinks code size to 64 bytes and RV32FDCV reduces it to 40 bytes.

After the main loop terminates, the code checks to see if n is odd. If so, it performs the last multiply-add using scalar instructions from Chapter 5. The final instruction returns to the calling site.

The 7-instruction loop at the heart of the MIPS MSA DAXPY code does 6 double-precision memory accesses and 4 floating-point multiplies and additions. The average is about 1 memory access and 0.5 operations per instruction.

x86 SIMD. Intel has gone through many generations of SIMD extensions, which we see in the code in Figure 8.6 on page 84. The SSE expansion to 128-bit SIMD led to the `xmm` registers and instructions that can use them, and the expansion to 256-bit SIMD as part of AVX created the `ymm` registers and their instructions.

The first group of instructions at addresses 0 to 25 load the variables from memory, make four copies of `a` in a 256-bit `ymm` registers, and tests to ensure `n` is at least 4 before entering the main loop. It uses two SSE and one AVX instructions. (The caption of Figure 8.6 explains how in more detail.)

The main loop does the heart of the DAXPY computation. The AVX instruction `vmovapd` at address 27 loads 4 elements of `x` into `ymm0`. The AVX instruction `vfmadd213pd` at address 2c multiplies 4 copies of `a` (`ymm2`) times 4 elements of `x` (`ymm0`), adds 4 elements of `y` (in memory at address ecx+edx*8), and puts the 4 sums into `ymm0`. The following AVX instruction at address 32, `vmovapd`, stores the 4 results into `y`. The next three instructions increment counters and repeat the loop if needed.

As was the case for MIPS MSA, the "fringe" code between addresses 3e and 57 deals with the cases when `n` is not a multiple of 4. It relies on three SSE instructions.

The 6 instructions of the main loop in the x86-32 AVX2 DAXPY code do 12 double-precision memory accesses and 8 floating-point multiplies and additions. They average 2 memory accesses and about 1 operation per instruction.

■ ***Elaboration: The Illiac IV was the first to show the difficulty of compiling for SIMD.***

With 64 parallel 64-bit floating-point units (FPUs), the Illiac IV was planned to have more than 1 million logic gates before Moore published his law. Its architect originally predicted 1000 million floating-point operations per second (MFLOPS), but actual performance was 15 MFLOPS at best. Costs escalated from the $8M estimated in 1966 to $31M by 1972, despite the construction of only 64 of the planned 256 FPUs. The project started in 1965 but took until 1976 to run its first real application, the year the Cray-1 was unveiled. Perhaps the most infamous supercomputer, it made a top 10 list of engineering disasters [Falk 1976].

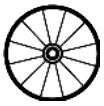
8.10 Concluding Remarks

If the code is vectorizable, the best architecture is vector.

—Jim Smith, keynote speech, International Symposium on Computer Architecture, 1994

Figure 8.4 summarizes the number of instructions and number of bytes in DAXPY of programs for RV32IFDV, MIPS-32 MSA, and x86-32 AVX2. The SIMD computation code is dwarfed by the bookkeeping code. Two-thirds to three-fourths of the code for MIPS-32 MSA and x86-32 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.

RV32V code in Figure 8.3 doesn't need such bookkeeping code, which halves the number of instructions. Unlike SIMD, it has a vector length register, which makes the vector instructions work at any value of `n`. You might think RV32V would have a problem when `n` is 0. It doesn't because RV32V vector instructions leave everything unchanged when `vl` $= 0$.

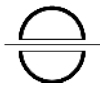


However, the most significant difference between SIMD and vector processing is not the static code size. The SIMD instructions execute 10 to 20 times more instructions than RV32V because each SIMD loop does only 2 or 4 elements instead of 64 in the vector case. The extra instruction fetches and instruction decodes means higher energy to perform the same task.

Comparing the results in Figure 8.4 to the scalar versions of DAXPY in Figure 5.8 on page 29 in Chapter 5, we see that SIMD roughly doubles the size of the code in instructions and bytes, but the main loop is the same size. The reduction in the dynamic number of instructions executed is a factor of 2 or 4, depending on the width of the SIMD registers. However, the RV32V vector code size increases by a factor of 1.2 (with the main loop 1.4X) but the dynamic instruction count is a factor of 43 smaller!

While dynamic instruction count is a large difference, in our view that is the second most significant disparity between SIMD and vector. Lacking a vector length register explodes the number of instructions as well as the bookkeeping code. ISAs like MIPS-32 and x86-32 that follow the incrementalist doctrine must duplicate all the old SIMD instructions defined for narrower SIMD registers every time they double the SIMD width. Surely, hundreds of MIPS-32 and x86-32 instructions were created over many generations of SIMD ISAs and hundreds more are in their future. The cognitive load on the assembly language programmer of this brute force approach to ISA evolution must be overwhelming. How can one remember what `vfmadd213pd` means and when to use it?

In comparison, RV32V code is unaffected by the size of the memory for vector registers. Not only is RV32V unchanged if vector memory size expands, you don't even have to recompile. Since the processor supplies the value of maximum vector length `mvl`, the code in Figure 8.3 is untouched whether a processor raises the vector memory from 1024 bytes to, say, 4096 bytes, or drops it to 256 bytes.



Unlike SIMD, where the ISA dictates the required hardware—and changing the ISA means changing the compiler—the RV32V ISA allows processor designers to choose the resources for data parallelism for their application without affecting the programmer or compiler. One can argue that SIMD violates the ISA design principle from Chapter 1 of isolating the architecture from implementation.

We think the high contrast in cost-energy-performance, complexity, and ease of programming between the modular vector approach of RV32V and the incrementalist SIMD architectures of ARM-32, MIPS-32, and x86-32 might be the most persuasive argument for RISC-V.

8.11 To Learn More

H. Falk. What went wrong V: Reaching for a gigaflop: The fate of the famed Illiac IV was shaped by both research brilliance and real-world disasters. *IEEE spectrum*, 13(10):65–70, 1976.

J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

Notes

[1] `http://parlab.eecs.berkeley.edu`

```
# a0 is n, a2 is pointer to x[0], a3 is pointer to y[0], $w13 is a
00000000 <daxpy>:
  0: 2405fffe  li       a1,-2
  4: 00852824  and      a1,a0,a1        # a1 = floor(n/2)*2 (mask bit 0)
  8: 000540c0  sll      t0,a1,0x3       # t0 = byte address of a1
  c: 00e81821  addu     v1,a3,t0        # v1 = &y[a1]
 10: 10e30009  beq      a3,v1,38        # if y==&y[a1] goto Fringe (t0==0 so n is 0 | 1)
 14: 00c01025  move     v0,a2           # (delay slot) v0 = &x[0]
 18: 78786899  splati.d $w2,$w13[0]     # w2 = fill SIMD register with copies of a
Loop:
 1c: 78003823  ld.d     $w0,0(a3)        # w0 = 2 elements of y
 20: 24e70010  addiu    a3,a3,16         # increment C pointer to y by 2 Fl.Pt. numbers
 24: 78001063  ld.d     $w1,0(v0)        # w1 = 2 elements of x
 28: 24420010  addiu    v0,v0,16         # increment C pointer to x by 2 Fl.Pt. numbers
 2c: 7922081b  fmadd.d  $w0,$w1,$w2      # w0 = w0 + w1 * w2
 30: 1467fffa  bne      v1,a3,1c         # if (end of y != ptr to y) go to Loop
 34: 7bfe3827  st.d     $w0,-16(a3)      # (delay slot) store 2 elts of y
Fringe:
 38: 10a40005  beq      a1,a0,50         # if (n is even) goto Done
 3c: 00c83021  addu     a2,a2,t0         # (delay slot) a2 = &x[n-1]
 40: d4610000  ldc1     $f1,0(v1)        # f1 = y[n-1]
 44: d4c00000  ldc1     $f0,0(a2)        # f0 = x[n-1]
 48: 4c206b61  madd.d   $f13,$f1,$f13,$f0  # f13 = f1 + f0 * f13 (muladd if n is odd)
 4c: f46d0000  sdc1     $f13,0(v1)       # y[n-1] = f13 (store odd result)
Done:
 50: 03e00008  jr      ra                # return
 54: 00000000  nop                       # (delay slot)
```

Figure 8.5: MIPS-32 MSA code for DAXPY in Figure 5.7. The bookkeeping overhead of SIMD is evident when comparing this code to the RV32V code in Figure 8.3. The first part of the MIPS MSA code (addresses 0 to 18) duplicate the scalar variable `a` in a SIMD register and to check to ensure `n` is at least 2 before entering the main loop. The third part of the MIPS MSA code (addresses 38 to 4c) handle the fringe case when `n` is not a multiple of 2. Such bookkeeping code is unneeded in RV32V because the vector length register `vl` and the `setvl` instruction lets the loop work for all values of `n`, whether odd or even.

```
# eax is i, n is esi, a is xmm1, pointer to x[0] is ebx, pointer to y[0] is ecx
 00000000 <daxpy>:
    0: 56                   push   esi
    1: 53                   push   ebx
    2: 8b 74 24 0c          mov    esi,[esp+0xc]   # esi = n
    6: 8b 5c 24 18          mov    ebx,[esp+0x18]  # ebx = x
    a: c5 fb 10 4c 24 10    vmovsd xmm1,[esp+0x10] # xmm1 = a
   10: 8b 4c 24 1c          mov    ecx,[esp+0x1c]  # ecx = y
   14: c5 fb 12 d1          vmovddup xmm2,xmm1     # xmm2 = {a,a}
   18: 89 f0                mov    eax,esi
   1a: 83 e0 fc             and    eax,0xfffffffc  # eax = floor(n/4)*4
   1d: c4 e3 6d 18 d2 01    vinsertf128 ymm2,ymm2,xmm2,0x1 # ymm2 = {a,a,a,a}
   23: 74 19                je     3e              # if n < 4 goto Fringe
   25: 31 d2                xor    edx,edx         # edx = 0
 Loop:
   27: c5 fd 28 04 d3       vmovapd ymm0,[ebx+edx*8] # load 4 elements of x
   2c: c4 e2 ed a8 04 d1    vfmadd213pd ymm0,ymm2,[ecx+edx*8] # 4 mul adds
   32: c5 fd 29 04 d1       vmovapd [ecx+edx*8],ymm0 # store into 4 elements of y
   37: 83 c2 04             add    edx,0x4
   3a: 39 c2                cmp    edx,eax          # compare to n
   3c: 72 e9                jb     27               # repeat loop if < n
 Fringe:
   3e: 39 c6                cmp    esi,eax          # any fringe elements?
   40: 76 17                jbe    59               # if (n mod 4) == 0 goto Done
 FringeLoop:
   42: c5 fb 10 04 c3       vmovsd xmm0,[ebx+eax*8] # load element of x
   47: c4 e2 f1 a9 04 c1    vfmadd213sd xmm0,xmm1,[ecx+eax*8] # 1 mul add
   4d: c5 fb 11 04 c1       vmovsd [ecx+eax*8],xmm0 # store into element of y
   52: 83 c0 01             add    eax,0x1          # increment Fringe count
   55: 39 c6                cmp    esi,eax          # compare Loop and Fringe counts
   57: 75 e9                jne    42 <daxpy+0x42>  # repeat FringeLoop if != 0
 Done:
   59: 5b                   pop    ebx              # function epilogue
   5a: 5e                   pop    esi
   5b: c3                   ret
```

Figure 8.6: x86-32 AVX2 code for DAXPY in Figure 5.7. The SSE instruction `vmovsd` at address a loads `a` into half of the 128-bit `xmm1` register. The SSE instruction `vmovddup` at address 14 duplicates `a` into both halves of `xmm1` for later SIMD computation. The AVX instruction `vinsertf128` at address 1d makes four copies of `a` in `ymm2` starting from the two copies of `a` in `xmm1`. The three AVX instructions at addresses 42 to 4d (`vmovsd`, `vfmadd213sd`, `vmovsd`) handle when mod(`n`,4) $\neq$ 0. They perform the DAXPY computation one element at a time, with the loop repeating until the function has performed exactly `n` multiple-add operations. Once again, such code is unnecessary for RV32V because the vector length register `vl` and the `setvl` instruction makes the loop work for any value of `n`.

9 RV64: 64-bit Address Instructions

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management.

—C. Gordon Bell, 1976

C. Gordon Bell (1934-) was one of the lead architects of two of the most popular minicomputer architectures of their day: the Digital Equipment Corporation PDP-11 (16-bit address), which was announced in 1970, and its successor seven years later, the Digital Equipment Corporation 32-bit address VAX-11 (Virtual Address eXtension).

9.1 Introduction

Figures 9.1 to 9.4 shows graphical representations of the RV64G versions of the RV32G instructions. These figures illustrate the small increase in the number of instructions to switch to a 64-bit ISA in RISC-V. The ISAs typically add only a few `word`, `doubleword`, or `long` versions of the 32-bit instructions and expand all the registers, including the PC, to 64 bits. Thus, `sub` in RV64I subtracts two 64-bit numbers rather than two 32-bit numbers as in RV32I. RV64 is a close but actually different ISA than RV32; it adds a few instructions and the base instructions do slightly different things.

For example, Insertion Sort for RV64I in Figure 9.8 is quite near the code for RV32I in Figure 2.8 on page 27 in Chapter 2. It is the same number of instructions and the same number of bytes. The only changes are that the load and store word instructions become load and store doublewords, and the address increment goes from 4 for words (4 bytes) to 8 for doublewords (8 bytes). Figure 9.5 lists the opcodes of the RV64GC instructions in Figures 9.1 to 9.4.

Despite RV64I having 64-bit addresses and a default data size of 64 bits, 32-bit words are valid data types in programs. Hence, RV64I needs to support words just as RV32I needs to support bytes and halfwords. More specifically, since registers are now 64 bits wide, RV64I adds word versions of addition and subtraction: `addw, addiw, subw`. They truncate their results to 32 bits and write the sign-extended result to the destination register. RV64I also includes word versions of the shift instructions to get 32-bit shift result instead of a 64-bit shift result: `sllw, slliw, srlw, srliw, sraw, sraiw`. To do 64-bit data transfers, it has load and store doubleword: `ld, sd`. Finally, just as there are unsigned versions of load byte and load halfword in RV32I, RV64I must have an unsigned version of load word: `lwu`.

For similar reasons, RV64M needs to add word versions of multiply, divide, and remainder: `mulw, divw, divuw, remw, remuw`. To allow the programmer to synchronize on both words and doublewords, RV64A adds doubleword versions of all 11 of its instructions.