VECTOR CRYPTOGRAPHY IN QEMU

ONE STEP IN THE PATH TO RATIFICATION

GOAL

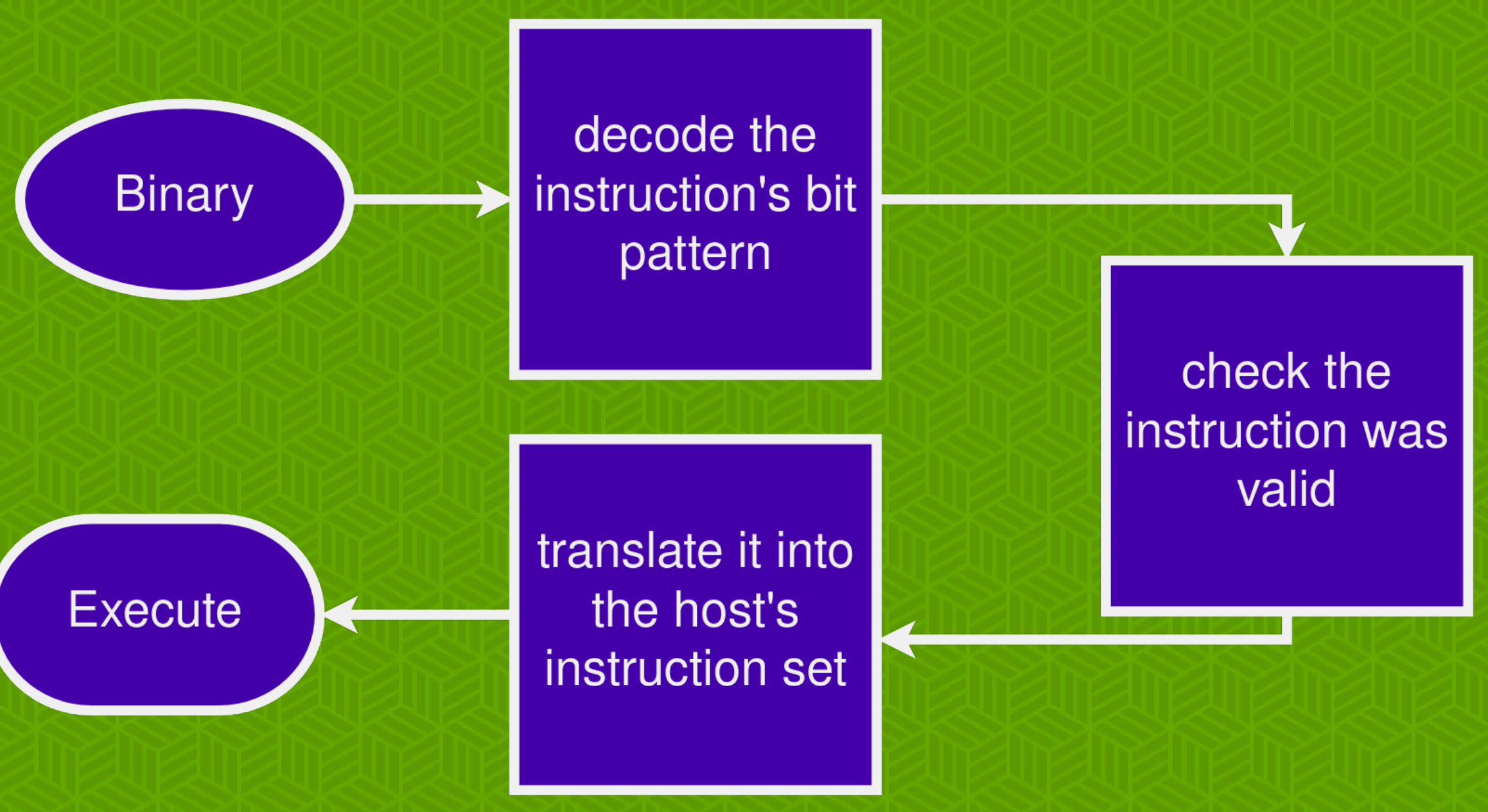
Add support for RISC-V's vector cryptography, *vcrpyto*, extensions to QEMU. This work is necessary for their ratification and was sponsored by SiFive.

The extensions provide instructions for implementing various cryptographic algorithms: AES, SHA-2, ShangMi, etc.

Unlike the scalar equivalents, the *vcrpyto* extensions leverage vector registers to increase the throughput of cryptographic operations.

QEMU

For each instruction in the *vcrypto* extension we had to add support for:


Binary
decode the instruction's bit pattern
check the instruction was valid
translate it into the host's instruction set
Execute

Let's work through `vaesz` as an example - see printout. This instruction is part of the AES block cipher extension.

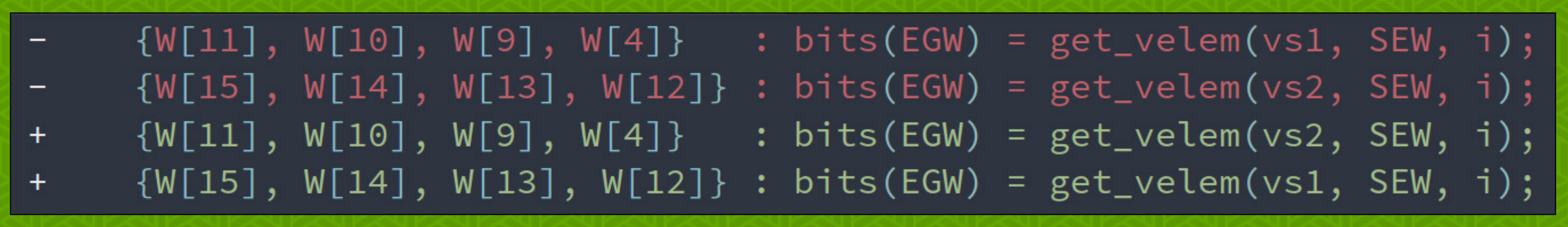
This instruction takes as arguments two vector registers, labelled `vd` and `vs2`. `vd` is then overwritten by the output from the instruction.

The first step is to add instruction's bitwise encoding to QEMU so it can be recognised in binary. The encoding is given by the table in the "Encoding (Vector-Scalar)" section.

The RISC-V vector spec defines several parameters that can be used to tune the behaviour of the vector instructions. Such changes can render an instruction illegal, so the next step is to have QEMU check this. For `vaesz`, the requirements are contained within the `if` clause.

Next we need to handle translating the RISC-V instruction into the host's instruction set. QEMU provides copious tooling for implementing this. We just needed to implement the pseudo code written in the spec as C code and then QEMU can handle the dynamic translation.

TESTING



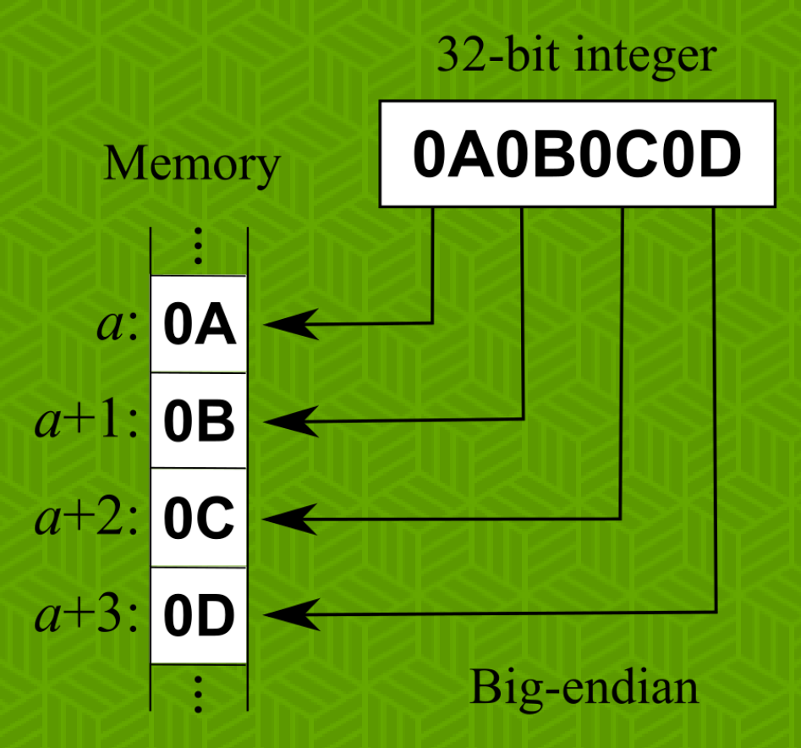
```
-    {W[11], W[10], W[9], W[4]}   : bits(EGW) = get_velem(vs1, SEW, i);
-    {W[15], W[14], W[13], W[12]} : bits(EGW) = get_velem(vs2, SEW, i);
+    {W[11], W[10], W[9], W[4]}   : bits(EGW) = get_velem(vs2, SEW, i);
+    {W[15], W[14], W[13], W[12]} : bits(EGW) = get_velem(vs1, SEW, i);
```

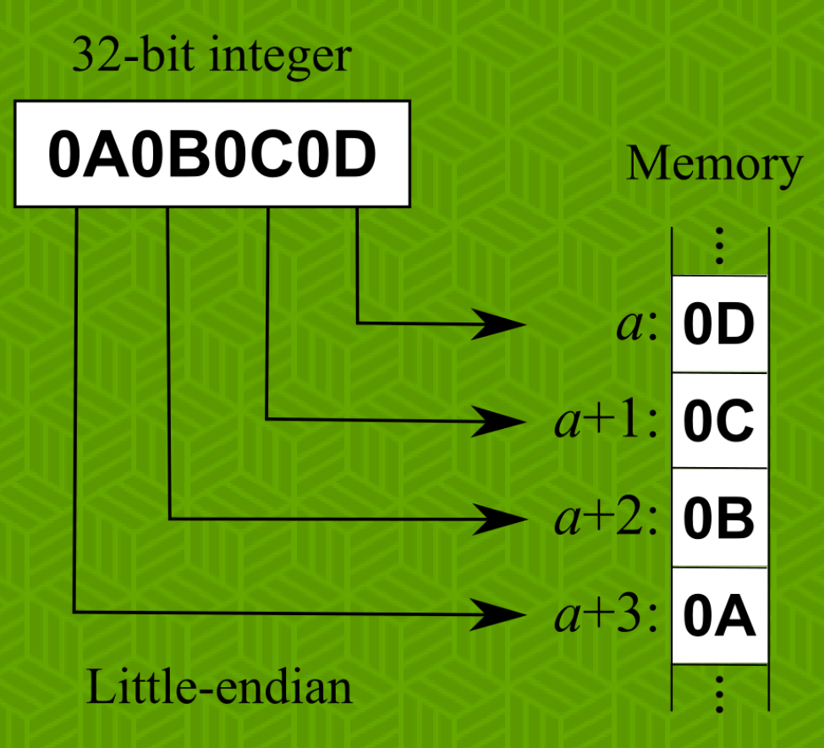
The *vcrypto* spec had no prior implementation and we also targeted the latest version, with breaking changes frequently introduced (see above). So testing was important!

Our sponsor provided a test suite to check our implementation against: auto-generated assembly code containing positive and negative test cases, which we could run within QEMU.

We also wrote our own test framework that ran in Linux userland and generated JIT instructions with random parameters. This allowed us to cover a wider range of cases (although only positive ones).

ENDIANNESS





There are two (sane) standards for loading data in/out of CPU registers: little endian (LE) and big endian (BE). With LE, the data's least significant byte is stored in the smallest memory address and the most significant in the largest. It is vice versa for BE.

RISC-V CPUs are LE. QEMU, on the other hand, can be run on both LE and BE hosts. Hence someone may try emulating an LE RISC-V CPU on some BE host.

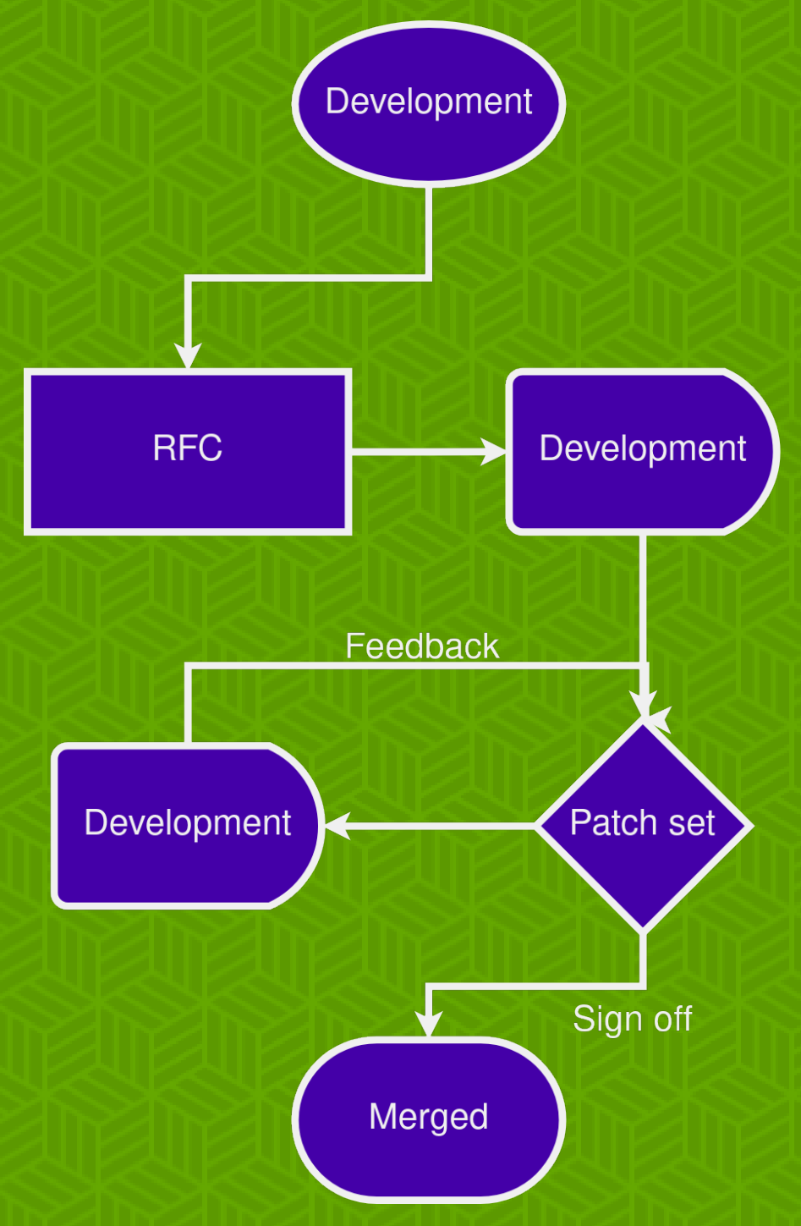
If we weren't careful we could've introduced some subtle bugs in this scenario.

Ideally we would have run our test suites on BE CPUs but we had no access to such hardware. However, QEMU-in-QEMU is actually a supported use of the program. Hence we could run our QEMU tests within a QEMU emulation of BE hardware! We opted for running FreeBSD in a 64-bit PowerPC emulation.

Only downside: building QEMU and running tests went from taking a few minutes on a laptop to multiple hours when done within the PowerPC emulation.

UPSTREAMING

The upstreaming process has been the usual cycle of submitting email patch submissions and implementing feedback. This began with an RFC before posting formal submissions, of which currently the 4th revision is being prepared. The *vcrypto* spec has ostensibly been frozen so hopefully future changes to the patchset will be minimal!



TALK TO US AGAIN

Come see us at booth 2 and find out how we can accelerate your RISC-V workflow!

P.S. We have merch 😎