Ara v2 update report

General progress

The update from v1 (RISC-V V specs v0.5) to v2 (RISC-V V specs v0.9, now v0.10) involves not only technical modifications and some architectural changes to adapt the hardware to the new specifications, but also an important refactoring of the codebase and software updates to provide and maintain a high-quality open-source release.
The introduced modifications must not degrade the functionality of the system and its operating frequency.

Refactoring

Ara was open-sourced recently, and this required refactoring the whole codebase to bring the code compatible with an open source release, together with a style unification to align the code to common style guidelines (lowRISC coding style guidelines). We also made sure that our codebase is compatible with Verilator, which allows Ara to be used with open-source simulation tools.

Memory System of Ara and Ariane

Ara v1

Ara v1 was instantiated in an SoC without any notion of memory coherence and consistency. Ariane featured a private write-back L1 cache, and Ariane and Ara shared a DRAM. In practice, this meant for any stores from Ariane to become visible to Ara and vice versa, a `fence` had to be executed, writing back and flushing the entire cache hierarchy. This caused a significant performance impact.

Ara v2: Memory Coherence

Ariane is configured to use a write-through L1 data cache. Ara and Ariane share a last-level cache (LLC), which is directly accessed by Ara. Hence, writes to the main memory by Ariane become directly visible to Ara. Writes by Ara trigger an invalidation of the corresponding cache set in Ariane’s L1 cache.

Ara v2: Memory Consistency

To prevent race conditions to the shared memory of Ariane and Ara, we implemented a load/store barrier:

- Scalar loads are only issued if there are no incomplete vector stores prior in program order.
- Scalar stores are only issued if there are no incomplete vector loads or stores prior in program order.
- Vector loads and stores are only executed by Ara if there are no pending stores in Ariane.

Ara Frontend

RVV-Agnostic Core

Ariane is now mostly agnostic of the V extension. For what concerns the V extension, inside Ariane there is only a lightweight pre-decoder to fetch immediate operands from the integer and FP register files. Apart from this, Ariane does not decode vector instructions, but it forwards them to Ara through a dispatcher.
The decoder for the V instructions is now inside Ara. This allows to increase the decoupling between processor and coprocessor, enhancing Ara re-usability and verification easiness.

Ariane-Ara interface

We standardized the interface between Ara and Ariane, making it agnostic to the coprocessor it is connected to. Ariane sends Ara only the basic information about the vector instruction it encountered, such as the instruction encoding, and any scalar operands it might need.

Ara Architecture

AraV2's internal organization is very similar to what it was in AraV1. But the open-sourcing of Ara led us to reimplement many of its modules, either to improve Ara's reliability, or to make it compliant with the new vector specification. Some modules were deeply affected by the new specification, and required major changes. The most notable difference is related to the organization of the Vector Register File, which is now mandated by the specification. This means that, in order to convert between mandated VRF organization, and Ara's internal organization, the load/store unit and the mask unit needs to shuffle/deshuffle the elements of the vector registers before writing them into the lanes. While the shuffling is costly, it also means that each lane only accesses its own section of the vector register file, when running normal vector instructions.

Mask unit

The mask unit is a new module which communicates with all the lanes. Its purpose is collecting operands from all the lanes, and then sending mask operands back to them. The reason for this is the new organization of mask elements on the vector register file. Before, the layout of the mask elements in the mask register followed the layout of the normal vectors, and the mask bit was the least significant bit of each "vector element". Now, however, the layout of the mask elements in the mask register is now packed. This implies that the accessing pattern is no more homogeneous, and accessing mask elements means accessing single contiguous bits.

For example, the bit #1 of the element #0 of any vector register (which resides in lane #0) is now used as a mask operand for the element #1 of the vector instruction (which resides in lane 1). The mask unit is responsible for this packing/unpacking of mask elements.

Ara RVV v0.5 to RVV v0.10

Vector Register File

v0.5: each vector register had its independent state and data type, i.e. the state of each vector register and its data type could have been different from the others. Before using the vector register file, it had to be configured, which led to costly configuration hardware (most notably, a divider).
v0.10: the state of the vector register file is now global and kept inside the vector CSRs, and the register file is agnostic on the data-type of the contained elements. This led to the scrapping of the VRF configuration hardware.

Instruction encoding

v0.5: the encoding of the instructions was polymorphic, i.e. the same instruction was used to perform processing on different data-types, depending on the source registers. This puts a lot of pressure on the decoding of vector instructions. since they need to take into account the instruction encoding and the value of the vector CSRs.
v0.10: the encoding of the instructions is monomorphic, i.e. there are different instructions to perform computation on different data-types. The size of the vector ISA is now much larger.

Mask register layout

Mask registers are used to perform predicated execution, i.e. perform computation on selected elements of a vector only. The operation is performed on element X only if the corresponding mask element is set.
v0.5: the layout of the mask elements in the mask register followed the layout of the normal vectors, and the mask bit was the least significant bit of each “vector element”. This implied that the access pattern was the same for both vector elements and mask elements. However, this also meant that we potentially had to load a 64-bit element from the vector register file, only to read its lowest significant bit as the mask operand.
v0.10: the layout of the mask elements in the mask register is now packed, i.e. all the mask bits are contiguous and packed in the least significant bits of the vector register. Moreover, every vector register can be a mask register. This implies that the accessing pattern is no more homogeneous, and accessing mask elements means accessing single contiguous bits.

Vector Instructions

New implemented instructions

- Strided memory operations
- Memory operations misaligned wrt AXI-data width
- VSETIVLI setting instruction

Reimplemented and refactored instructions (currently under verification for 2, 4, 8, 16 number of lanes)

- Slide instructions
- Integer logical/arithmetic instructions
- FP Arithmetic instructions, normal and widening ones
- FP Comparisons
- FP Conversions

Instructions in progress

- Move whole-register
- Whole-register memory operations
- Mask Memory Operations
- Integer reductions

Missing instructions for full compliance with RVV 0.10

- Scatter-Gather memory operations
- Fault-Only-First loads/stores (like normal loads, but they take traps in a different way)
- Fixed-Point arithmetic/logical instructions
- Specific mask instructions (vpopc, vfirst, viota, vid, vmsbf, vmsif, vmsof)
- Gather/Compress registers
- Floating-Point reciprocal, reciprocal sqrt, round towards odd
- Move scalar elements between Ara and Ariane
- FP Reductions

Toolchain progress

We updated the toolchain from GCC to LLVM, as the main efforts of the developers are now on LLVM for what concerns RVV.
The new toolchain and software environment were brought to RVV v0.9, and then to RVV 0.10. This update also allowed our software environment to support the official RVV intrinsics (https://github.com/riscv/rvv-intrinsic-doc).

Verification

The updating process requires first-order verification after every code update. This is implemented using continuous integration via GitHub Actions: the verilated RTL model is verified against the Spike golden model, testing every supported instruction and all the available benchmarks. When we implement a new instruction, we also create the related test. We also plan to fully support Ara variants with 2, 4, 8, and 16 lanes on the GitHub Actions continuous integration, testing that Ara works with a flexible set of architectural parameters.
During the passage from AraV1 to AraV2, the verification level and the maturity of the codebase significantly increased.

Avoid regressions

Before freezing a new feature into the main branch of the codebase, we synthesize the new system to be sure not to have introduced timing regressions. The introduction of a new feature is therefore an iterative process.

Benchmarks

The Arav1 benchmarks (matmul, convolutions) were refactored and adapted to an open-source release. They were also updated to be compatible with RVV 0.10.

- Matrix Multiplication
- Convolution

Then, the following benchmarks were added, and currently to be merged onto the main branch:

- Dropout
- Reduction Benchmark (reduction emulated using vector slides)
- Benchmarks from RALC88:
    - Jacobi2d
    - Pathfinder
    - Particle Filter

RALC88 benchmarks were used to improve the verification on Ara, and allowed to spot and fix bugs in the codebase.

Project plan

WP1: Benchmarks and Performance profiling (30%/100%)
Some benchmark candidates were proposed, namely:

- MatMul
- Convolution
- Softmax (we saw that this is not the best example to test reductions)
- Dropout
- ROI pooling/align
- Hough Transform
- Warp Affine
- RANSAC
- Gaussian Filter
- Reduction-Slide

The plan is to extract useful kernels from these applications to implement the measurement of performance metrics useful to get insights on how to improve the overall architecture.

We refactored the benchmarks of AraV1 (matmul, conv2d) integrating them in the CI with performance measurements (still WIP for conv2d and the other benchmarks), we added a first implementation of Dropout and Reduction-Slide. Reduction-Slide is the emulation of a reduction operation in software using slides, and it is useful for both evaluating and improving the performance of the slide unit and studying the trade-off of different implementation choices for the HW reductions.

We adapted some benchmarks from the RALC88 repository (https://github.com/RALC88/riscv-vectorized-benchmark-suite), namely AXPY, Jacobi2d, Pathfinder, and Particlefilter (still WIP) with the aim of improving the verification level of the system and testing the official RVV 0.10 LLVM intrinsics.

WP2: Update to AraV2 + Release (70%/100%)
All the instructions previously supported for RVV 0.5 are now supported for version RVV 0.10. Moreover, new instructions were added, or are planned to be added in the immediate future. The instructions that are still missing are either planned/WIP (gather/scatter, integer reductions, Fixed Points) or they are low priority (floating point reciprocal of sqrt, round toward odd, ...).
We replaced the GCC toolchain with LLVM, and we worked on improving the status of the compilation flow/CI, e.g. optimizing the flow with Verilator. The modification of the toolchain was required to port Ara to RVV 0.10, and was also motivated by the toolchain developers, who are now focusing on LLVM to support the official RVV intrinsics.
We also finalized the first TO of Ara (http://asic.ethz.ch/2021/Yun.html), and this helped in improving the verification level of the whole system.