Implementing out-of-order issue in CVA6 for efficient support of long variable latency instructions

Eric Guthmuller[1] and Tanuj Khandelwal[1] *

[1]Univ. Grenoble Alpes, CEA, List,
F-38000 Grenoble, France

Abstract

OpenHW Group's RISC-V CVA6 core, while being heavily configurable, even supporting superscalar execution, is still limited to in-order issue. Out-of-order execution allows for more efficient instruction scheduling in the case of non-predictable latency, such as memory accesses for example. We propose adaptations to the CVA6 core pipeline by adding one pipeline stage and reorder queues to issue most instructions out-of-order. Our work is also superscalar-ready as multiple instruction queues are instantiated in parallel. We measured a 10% performance increase on Coremark with our modifications.

Introduction

OpenHW Group 64-bit RISC-V CVA6 [1, 2] core is offering application-class performance for embedded usages. It features scalar and superscalar in-order execution implemented with 6 pipeline stages and runs fully-fledged operating systems such as Linux. A custom extension port also allows for taking advantage of RISC-V ISA extensibility.

In the context of the European Processor Initiative project, we used the CVA6 to implement the VaRiable Precision processor [3] (VRP). It implements the *Xvpfloat* RISC-V ISA extension to support variable and extended precision floating point operations for scientific computing. When optimizing linear algebra kernels, we encountered two difficulties:

1. Some kernels, e.g. sparse matrix-vector multiplication, are highly irregular and it is thus difficult to apply static optimization techniques such as loop unrolling.
2. As latency and throughput of extended precision instructions vary with precision, differently from one instruction to another, perfect instruction scheduling changes with precision. As our goal with this accelerator is to avoid recompilation when changing precision, instruction scheduling may be suboptimal depending on chosen compute precision.

These observations motivated us to implement out-of-order issue of instructions in our modified CVA6 core, leading to the design of the next generation of the VRP, called *VXP* (Variable eXtended Precision processor). While we branched our CVA6 in 2020, our modifications still apply to up-to-date vanilla CVA6.

*Corresponding author: eric.guthmuller@cea.fr

Design Objectives

To efficiently mask high latency of our *Xvpfloat* instructions (up to 20 cycles), we wished to support up to 64 in-flight instructions. Total reordering window had to be in the same ballpark. We also wanted our design to be ready for future superscalar execution, as already proposed for the vanilla CVA6 [4].

As the VXP is targeting linear algebra kernels mostly written in C and assembly, our applications are not branch-heavy, especially regarding indirect branch instructions. That is why we chose to keep branch handling in the first execute stage, blocking issue of new instructions until register dependencies are solved. This is a drawback when dealing with function pointers such as C++ class vtables for example.

Finally, we did not want to deal with too much pipeline flushes to avoid cancelling already issued *Xvpfloat* instructions that are energy intensive. On top of preventing branch speculation after the issue stage, we also prevented load instruction speculation as long as address resolution of previous memory requests is not done. These decisions, while having a negative impact on Instructions Per Cycle (IPC), benefit energy efficiency and simplify the hardware.

Main Pipeline Modifications

An overview of the VXP pipeline featuring 7 stages is shown in Figure 1. First, we chose to implement register renaming of all register banks as it naturally solves a lot of issues regarding clobber of destination registers and allow for more in-flight instructions. Each rename table and corresponding register file depth is configurable. In the VXP, all register files have 64

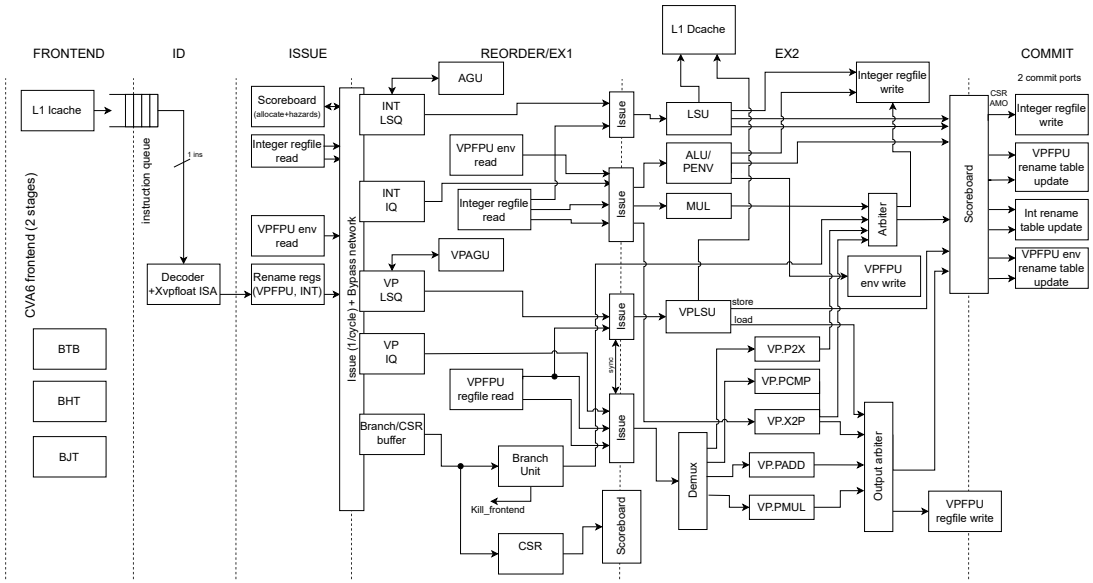

Figure 1: *VXP pipeline microarchitecture*

entries.

We settled on having multiple independent smaller reordering queues (16 entries in VXP) instead of a bigger central one, as: 1/ it would scale better when going superscalar, and 2/ it allows each queue to be specialized. So our design has 4 instructions queues : 2 for IMA instructions, 2 for *Xvpfloat* instructions, in each case with separate load/store queues(LSQ).

As a general design principle, we settled on storing only addresses of memory requests in the reorder queues which are necessary to enforce memory consistency. All other source register values are read directly from the register files or bypassed in the same cycle of issue. This reduces the cost of the reorder queues entries, at the expense of more register files read ports. A wide superscalar design would probably necessitate putting everything in the reorder queues and rely entirely on the register bypass network.

Memory consistency is ensured by implementing a dependency matrix in each LSQ. An address comparator per LSQ entry allows for updates of this matrix. The integer LSQ implements one Address Generation Units (AGU) per entry, while the *Xvpfloat* LSQ (VPLSQ) implements only two as they are more costly, requiring integer multiplication.

The integer register file has 5 read ports: 2 for CSR/branch in EX1 stage, 2 for arithmetic instructions and 1 for store instructions. It also feature 4 write ports: one for the load unit, one for the ALU, one shared between multiplication and branch, and one for commit to handle CSR updates and AMOs.

Finally, we simplified the scoreboard by first removing the 64-bit result as register files are written directly thanks to renaming, and also by removing the exception field as we now only store the oldest exception. This saves 193 bits in each scoreboard entry. We also improved the store buffer design by fusing speculative and committed queues, while allocation is done in issue stage to keep store order.

Conclusion

While designing the *VXP*, the next generation of our extended variable precision accelerator, we identified out of order execution as a key enabler for higher IPC. We added the necessary hardware to our modified CVA6 core to reach up to 64 in-flight instructions and a similar reorder window. The VXP is successfully synthesized at 90 MHz on FPGA and achieves a 10% better performance than the previous generation VRP on Coremark.

References

[1] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (July 2019), pp. 2629–2640. DOI: 10.1109/TVLSI.2019.2926114.

[2] OpenHW Group. *CVA6 RTL code*. https://github.com/openhwgroup/cva6.

[3] Eric Guthmuller et al. "Xvpfloat: RISC-V ISA Extension for Variable Extended Precision Floating Point Computation". In: *IEEE Transactions on Computers* 73.7 (2024), pp. 1683–1697. DOI: 10.1109/TC.2024.3383964.

[4] Côme Allart et al. "Using a Performance Model to Implement a Superscalar CVA6". In: *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*. CF '24 Companion. Ischia, Italy: Association for Computing Machinery, 2024, pp. 43–46. ISBN: 9798400704925. DOI: 10.1145/3637543.3652871.