Hardware-Software Co-design for Low-cost AI processing in Space Processors

SIMD module design discussion

Author: Marc Solé I Bonet
Email: marc.sole.bonet@estudiantat.upc.edu
Director: Leonidas Kosmidis
Email: leonidas.kosmidis@bsc.es
Date:

Abstract

The goal of this Master Thesis is to co-design architectural and microarchitectural extensions of at least one space processor such as Cobham Gaisler's LEON3 (SPARC v8 based) or NOEL (RISC-V based), in order to increase its performance capabilities for AI processing with a low hardware cost. The architectural changes include the extension of the ISA with short SIMD instructions which operate over the existing register file including also predication capabilities. In the microarchitectural side, the processor design will be converted to static superscalar featuring predication. The design decisions will be driven by an analysis of AI processing software and the proposal will be implemented in VHDL. Finally, compiler support will be added to take advantage of the proposed hardware features.

Table of Contents

List of Figures

List of Tables

1 Project presentation

1.1 Motivation

This project begun the previous semester during the Processor Design course. I decided that the project for said course could be the initial steps for my master thesis, and after discussing it with the professor we agreed on a topic that could be developed within the expected course but that might be further expanded as my thesis.

I was conflicted weather to do my thesis about hardware architecture or about compilers. Taking into account the discussed before I decided to mix both. I have always enjoyed the hardware design courses during my degree and later in the master, but I also felt a great deal of interest in compilers as they are the *interpreters* of human intentions to the computers. Therefore, being able to work in both fields for my project felt like the most appropriate direction.

My first take on the project was to add predication to an existing processor and then add the compiler support for this feature. However, after consulting the RISC-V specification I realized that doing so would not add any significant improvement [1]. On the other hand, I was aware of the increasing interest in implementing efficient machine learning in space processors, and decided to turn on this area but keep my original idea of predication as an additional feature.

With the main topic decided, the selection of the base processor was a natural given it had to be an space processor which was available under a public license and written in a language I know. With this conditions in mind, I selected the LEON3 [2] processor designed by Cobham Gaisler. However, recently they have released the NOEL-V [3] which follows the RISC-V standard, because of this, I decided to work using both processors.

1.2 Objectives

As introduced in the previous section the main goal for this project is to add additional support for efficient machine learning in the space processors LEON3 and NOEL-V. To do so I will follow two different approaches. The first one is to add a SIMD module that performs vector operations aimed at improving performance in artificial intelligence applications. By checking the most common instructions and characteristics in said applications the hardware will be optimized for an improved performance.

The second approach is to turn both processors into static superscalars with a dual issue pipeline. Furthermore, the integer pipeline will be extended with predication support allowing to execute or not instructions depending on a predicate value, thus turning control dependencies into data dependencies. Finally, to properly take advantage of the modifications the compiler will be modified to use the new instructions in the SIMD module and the predication characteristics that will now be available.

2 Module description

2.1 Main characteristics

The Single Instruction Multiple Data (SIMD) module, was initially conceived as a SIMD within a register (SWAR), this means that it will use the same registers as the integer pipeline but will do the operations in smaller subsets.

This characteristic has the advantage of not requiring an additional register file to store the larger registers, which is ideal for a module with minimal impact on space and power consumption. Also, is important to note that artificial intelligence applications work with small values of data *CITATION NEEDED*. Therefore, I decided that each component of the vector register would be 8 bits long.

The functions implemented were decided through an analysis of which operations are used in artificial intelligence applications, and it was seen that the dot product, is a pivotal operation in machine learning[4]. This gave the conclusion that it would be beneficial to implement in a single instruction the multiplication and addition.

It can be seen that there are also other situations in which it may be interesting to perform two operations consecutively. And in many cases the second part would be a reduction performed on the result of the first computation. With this in mind I divided the module in two stages. Since there is no need for the module to access memory, as it works using the same registers as the integer pipeline, the second stage of the module can match with the memory access stage with no drawback.

The module has also support for saturated instructions, this means that for those operations the result will not overflow over the data type representation. The saturation of the operation is given in the opcode for the first stage, and in case there is saturation, the same is done in the second stage. In section 2.3.1 the justification for this characteristic is explained in more detail.

2.2 Additional features

Aside from the defined characteristics the module was expanded with new functionalities in order to simplify some operations and allow other interesting capabilities.

2.2.1 Masking/Predication

Previously, I have stated my interest in adding to the processors predication, this feature is also compatible with a vector module but is more commonly known as masking. The idea is simply to add a boolean vector which determines weather the operation should modify the result in the corresponding vector component.

To keep the design modular, it is impossible to not overwrite the result register if this is not also passed to the module, but this idea was discarded in favor of passing the value of the first source register. This is due to many times being the destination register also the source one, and if needed the same result could be achieved with an additional instruction.

The approach taken in the module is that the value of the mask vector must be set by a specific instruction and will keep this value until it is overwritten or the processor is reset. In the later case the value will go back to the default one which is no mask.

2.2.2 Swizzling

The swizzling allows to reorder and duplicate the source vector components to perform an operation using any combination. This is simply a multiplexor which picks which of the components is used in each byte operation.

Same as in the mask case, the swizzling configuration is defined by an instruction and is constant until it's set again. In this case the default configuration does not alter the original order of the register bytes.

2.3 Design decisions

2.3.1 Second stage saturation

Initially there wasn't supposed to be saturation in the second stage, the reason being that since in it reduction operations were performed, there was no need to be concerned on the overflow in the result. However, this produced inconsistencies regarding the input and output data types, therefore, I decided to add saturation in the second stage.

However, there were no available bits in the instruction to extend the second operand code to add saturation support. Thus, I considered three different approaches to solve this issue.

- At this point of the project it hadn't been considered to add the possibility to operate using an immediate, and the bit that identifies the operand as such was kept in regards of coherence with the rest of the ISA.

  Therefore, I first thought of using the immediate bit as a *saturation* bit, that for indicated for both stages weather the operation should use saturation or not. However, this meant a loss of many bit combinations as there are operations incompatible with saturation.

- Similarly, taking advantage of the immediate bit I considered to increase the size of the second opcode to include the immediate bit as well. But this altered the coherence of the instruction and blocked to use in the future of immediate operands and thus the idea was also discarded.

- Finally, I opted for the final approach, which is the one that was finally implemented, that links the saturation of the second stage with the first stage. This means that if the instruction on the first stage used saturation so will the second stage.

There were some considerations to make before deciding for this last option. It may seem that with this approach if there is no instruction in the first stage we can't have saturation in the second. However, if in the first stage we do a saturated addition with zero there is no issue.

The only drawback, which happens also in the first approach, is that we cannot have saturation in only one stage. Nonetheless, the result of said operation can still be achieved by using two consecutive instructions. And we also have to consider that when using saturation in a data value all consecutive operations will likely also use saturation.

References

[1] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.

[2] Cobham Gaisler. LEON3 processor, 2021. URL `https://www.gaisler.com/index.php/products/processors/leon3`.

[3] Cobham Gaisler. NOEL-V processor, 2021. URL `https://www.gaisler.com/index.php/products/processors/noel-v`.

[4] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao. *ACM SIGARCH Computer Architecture News*, 43, 2015. ISSN 0163-5964. doi: 10.1145/2786763.2694358.