Implementing Vector Extensions to RISC-V in Gem5

Xinyu Ma
205080043
xinyuma@g.ucla.edu

Zhaoning Kong
005223528
jonnykong@cs.ucla.edu

Weijia Yuan
704946874
weijia@cs.ucla.edu

Yajun Shi
304946079
yajuns@outlook.com

Abstract

Vector architecture is a scheme that utilizes data level parallelism. The idea is to take vector registers as operands, so that instructions and memory accesses operate on whole registers. Since elements within a vector have no dependencies on each other, vector processing allows more efficient computation through data parallelism. Due to no support for RISC-V vector instructions in gem5, our project is intended to provide a working subset that supports this kind of data-parallel execution.

The goal of this project is to implement a subset of vector extensions to RISC-V in gem5, in order to achieve faster dense linear algebra computation. The implementation basically follows an existing RISC-V vector extension proposal [1], with a few modifications as the draft is not completed. This project implements some of the proposed components including vector registers and vector instructions for a basic working subset. The evaluation results show that the implemented vector processing improves the matrix computation performance by a large factor, and there exists future optimizations that can be made for a more complete architecture.

1. Introduction

RISC-V is an open source implementation of a reduced instruction set computing-based ISA. It's completely open and freely available to academic and industry, and it's compatible to all kinds of software and programming languages. The guiding principle of RISC-V is to make an ISA suitable for nearly any computing device. RISC-V is stable and keeps its own basic instruction set architecture unchanged. Unlike almost all the old architectures, RISC-V can achieve those features by keeping independent from any single company's decision.

Gem5 is a modular, open source simulation platform that supports different ISAs including x86. Its advanced simulation features provide RISC-V applications with a great environment to simulate. RISC-V implements the instruction sets for single core simulations in SE mode [1], including integer and multiply instructions, atomic instructions and floating-point instructions.

The traditional approach to computer architecture is incremental ISA, which must not only implement new ISA extensions but also implement all of the old extensions for the purpose of binary compatibility. But this requirement actually has significantly increased the content size of ISA over time. For RISC-V, it's a modularized ISA based on RV32l core. RV32I is the base 32-bit integer ISA (shown in figure 1) RV32l core is fixed and will never change. This provide a stable goal for developers and programmers to develop standard extensions. RISC-V can have 32 integer registers and 32 floating point registers. The memory is addressed by 8-bit bytes, but instructions will be formed into 32-bit address. Load and store of 8-16-bit items are the only instructions that can access to main memory. RISC-V increases a computer's speed and reduces its cost by fixing the location of its most significant bits [2]. It develops a specific bit arrangement to reduce multiplexers. RISC-V also wants to develop an efficient support for data level parallelism. Unlike the traditional SIMD architecture which needs old hardware recompilation, the vector design for RISC-V should be scalable, so that same program can get a full performance on machines with any vector length [2]. In this project, we only focus on the vector extension of RISC-V, as an approach to enable faster dense linear algebra computation by exploiting data parallelism.

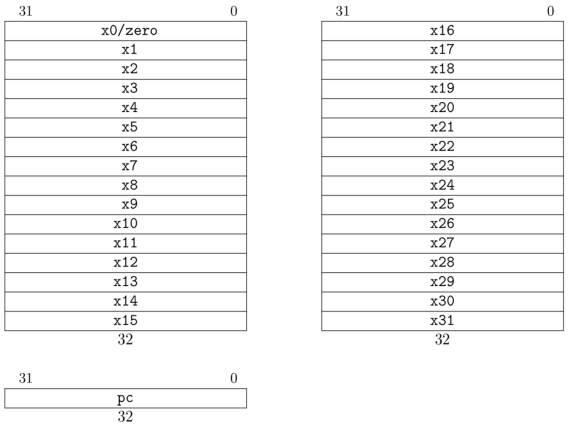


Figure 1: RV32I user-visible architectural state

31 29	28	27	26 25	24 20	19 15	14	13 12	11 9	8 7	6 0	Opcode
1001		000		vs2	vs1	1	m	vd		1010111	VADD
1001		001		vs2	vs1	1	m	vd		1010111	VSUB

Figure 2: instruction table

2. Related Work

This project is implemented under the specification in the GitHub repository riscv-v-spec, which is a base vector extension proposal to RISC-V, aiming at providing general support for data-level parallel processing [3]. The riscv-v-spec repository contains several specification files: v-spec.adoc, inst-table.adoc, vector-op-base.csv, vector-op-vtypes, etc. There exists lots of differences in the specifications across different versions, and we use this particular v-spec.adoc version 0.5 because it comes with a detailed vector instruction table.

The v-spec.adoc file introduces a general specification of the vector extension without specific implementation details, and the content covers vector CSRs, vector configuration instructions, vtypes, and vector instructions, which provides a basis of the essential components that need to be implemented to simulate RISC-V vector processing in gem5.

The inst-table.adoc file illustrates all vector instructions and their bitfield encodings. These vector instructions are of size 32 bits, with most instructions indicating opcodes, vector register operands, destination vector registers, and a two-bit mask field. For example, figure 2 shows VADD and VSUB from the table.

3. Vector Extension

In this section we describe the implementation of vector extension to RISC-V in gem5. While there are existing code pieces leaving space to vector architecture in current gem5 RISC-V, they are incomplete and lack explanations. In order to have the basic functionalities of vector computations, we need to build the essential related architecture components. Therefore, our implementation includes vector CSRs, vector registers as operands, decode of vector instruction bitfields, a subset of vector instructions and corresponding formats.

3.1 Vector Extension

In this project, we implement the fundamental vector CSRs, vl, along with 32 vector registers. vl is a WARL CSR that holds the current active vector length, where WARL stands for write any values, read legal values. The active vector length determines the number of elements processed by each vector instruction, and its value can be set with the instruction vsetvl. The vsetvl instruction details are explained in section 3.3. The implementation also includes 32 vector registers, naming from v0 to v31, with each holding 4 elements of type uint32_t by default.

3.2 Vector Operands

There are two extended operand types used by the vector operands, sv and svf, representing int32_t and float element types respectively of the vector registers. The vector operands therefore include destination vector (vd) and source vectors (vs1, vs2, and vs3) of both types, and v0 is used for masking. In addition, vl CSRs is also included, it is an int register holding the value of type uint64_t, and can be set by instruction VSETVL. The code of this section can be found in gem5/src/arch/riscv/isa/operands.isa file.

3.3 Vector Instructions

Among all the 32-bit encoding instructions proposed by riscv-v-spec, we implement the ones that are necessary to our linear algebra computation experiments. These include vector configuration instructions, vector integer and floating-point compute instructions, vector load/store instructions, and vector register element movement instructions.

The instruction encoding includes two bits of masking, indicating the scalar/vector shape of the result and the type of masking. m = 00 represents scalar shape result; m =01 represents vector shape result; m = 10 represents vector operation that is enabled when v0[i] = 0; m = 11 represents vector operation that enabled when v0[i] = 1.

VSETVL (vsetvl rd, rs1) - set active vector length in vl. If the requested application vector length (AVL) in rs1 is less than or equal to the maximum vector length (set to 4 in this project), set vl to AVL; otherwise set vl to the maximum vector length. This new active vector length is also written to rd.

VADD (vadd vd, vs1, vs2) - add vs1 and vs2 element-wise, write each result to vd.

VSUB (vsub vd, vs1, vs2) - subtract vs2 from vs1 element-wise, write each result to vd.

VXOR (vxor vd, vs1, vs2) - xor vs1 and vs2 element-wise, write each result to vd.

VSEQ (vseq vd, vs1, vs2) - compare vs1 and vs2 element-wise, write 1 to vd if they are equal, write 0 otherwise.

VSLT (vslt vd, vs1, vs2) - compare vs1 and vs2 element-wise, write 1 to vd if the element of vs1 is less than the element of vs2, write 0 otherwise.

VMUL (vmul vd, vs1, vs2) - multiply vs1 and vs2 element-wise, write each result to vd.
VREM (vrem vd, vs1, vs2) - divide vs1 by vs2 element-wise, write each remainder to vd.
VMADD (vmadd vd, vs1, vs2) - multiply vs1 and vs2 element-wise, write accumulated result to vd.
VREDSUM (vredsum vd, vs1) - sum up the elements in vs1, write the result to vd. This takes a vector shape as input and produces a scalar shape.
VLW (vlw vd, rs1) - load elements in continuous sequence from memory in 4-byte stride starting at the base address stored in rs1, write to vd.
VSW (vsw vs3, rs1) - store elements in vs3 in continuous sequence to memory in 4-byte stride starting at the base address stored in rs1.
VSXW (vsxw vs3, offset(rs1), vs2) - store elements in vs3 to memory, with base address offset(rs1), and each element address calculated with corresponding offset indicated in vs2.
VEXTRACT (vextract rd, vs1, rs2) - extract one element (indicated by rs2) from vs1, write to rd.
VMERGE (vmerge vd, vs1, vs2) - merge elements from vs1 and vs2 to vd, the mask determines whether to pick element from vs1 or element from vs2 to write. If mask is 00, write first element of vs1 to destination.
VMERGEX (vmergex vd, rs1, vs2) - merge rs1 and elements from vs2 to vd, the mask determines whether to pick rs1 or element from vs2. If mask is 00, write rs1 to destination.
We also implement floating-point versions of some of the instructions described above, including VFADD, VFSUB, VFMIN, VFREDSUM, VFMADD, and VFEXTRACT. The functionalities are quite similar to the integer version, except VFMADD.
VFMADD (vfmadd vd, vs1, vs2, vs3) - multiply vs1 and vs2 element-wise, add corresponding element in vs3, write each result to vd.

4. Evaluation

First, we choose PolyBench to test our ISA structure. PolyBench is a collection of benchmarks containing static control parts. We can uniformize the execution and monitoring of kernels in this way. The features of PolyBench include:
* Single file and tunable at compile time
* non-null data initialization
* no dead code elimination
* clear kernel marking

In this simulation, we start with optimizing 2D multiplications kernels go on to optimize the code in the PolyBench suite. Our experiments explore the performance of RISC-V Vector Extension. In this work, we focus on the time consumption for the matrix multiplications. We run the experiment with in order CPU core and DDR3_1600_8x8 memory type. The reason we choose in order CPU rather than OOO is that the current pipelined CPU design in gem5 does not support multiple memory access in a single instruction. Especially, load instructions need to specify the EA in initiating phase and get the data during retirement.

3mm is a linear algebra kernel that consists of three matrix multiplications. The program will take A, B, C, D four matrix as input and gives G as the output.

Input	
A	P * Q
B	Q * R
C	R * S
D	S * T
Output	
G	P * T = （A * B) * (C * F)

Table 1: Input and output of 3mm benchmark

Floyd Warshall problem is a graph algorithm which tries to find the shortest distances between every pair of vertices in a weighted directed graph. But in PolyBench we only compute the shortest path length. It takes a N x N matrix named “w” as input (w stands for weight/cost in the graph) and gives a N x N matrix named paths as output, where path represents the shortest path length. Linear sieve is an algorithm that finds all prime numbers between 2 and N. It takes N as an input and the prime number list as an output.

We also implemented some functions in assembly, which contains the implemented vector instructions. These functions will be linked with C codes, to help with the PolyBench suite. These functions include vecadd, muladd, dotprod, vextract, fmuladd and fdotprod. The details of these functions can be found in Table X.

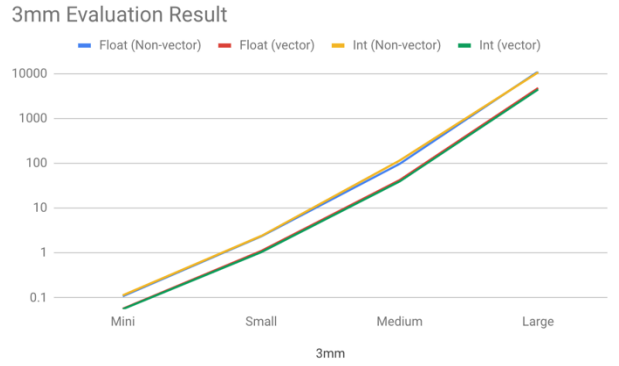
Function	Signature in C	Description
vecadd	void vecadd(int len, int *X, int *Y, int *W);	W[i] = X[i] + Y[i];
muladd	void muladd(int len, int *X, int *Y, int *W);	W[i] += X[i] + Y[i];
dotprod	int dotprod(int len, int *X, int *Y, int stride);	return sum_i{X[i] * Y[i][stride]};
vectract	int vextract(int *X, int idx);	return X[idx];
fmuladd	void fmuladd(int len, float *X, float *Y, float *Z, float *W);	W[i] = X[i] * Y[i] + Z[i];
fdotprod	float fdotprod(int len, float *X, float *Y, int stride);	return sum_i{X[i] * Y[i][stride]}

Table 2: Functions implemented in assembly with vector instructions

We evaluated all 3 benchmarks using the default settings of se.py in gem5. Figure7 shows the time consuming（time is measured in the unit of tick, it also can be represented by the number of instructions if divided by 500) of benchmarks in four different configurations: non-vector float, vector-float, non-vector int, vector-int. Integers(int) and Float are different data types. Integer are whole numbers while floating types can hold real numbers such as “2.33”. Vector prefix means this is a data set for the test of RISC-V vector extension and non-vector prefix means this is for the original RISC-V testing. Mini, small medium and large means the sample dataset sizes which can be found in the head file of each benchmark. We can find out that RISC-V with vector extensions runs much faster compared to the original ones.

3mm	Float (Non-vector)	Float (vector)	Int (Non-vector)	Int (vector)
Mini	107862500	56632500	112928000	55393000
Small	2334746500	1103654500	2383676500	1037788500
Medium	96942195500	42035237000	114918685500	39169078000
Large	11156878222500	4769997427000	10663644964000	4432291467500

Table 3: Original data of 3mm benchmark evaluation



Outliers: RISC-V vector extension can achieve higher performance for both float and int matrix multiplications

Floyd	Float (Non-vector)	Float (vector)	Int (Non-vector)	Int (vector)
Mini	1676146000	528316000	1640950000	446467000
Small	38965215500	12981408500	46661422500	10787466500
Medium	830245293500	270407742500	995236582500	223476992500

Figure 3: 3mm Evaluation Result

Table 4: Original data of Floyd benchmark evaluation

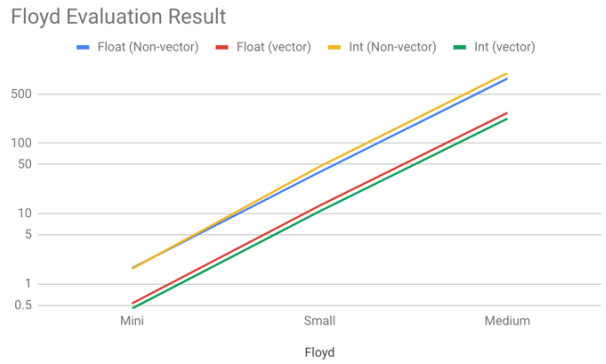


Figure 4: Floyd Evaluation Result

Outliers: RISC-V vector extension can achieve higher performance for both float and int in graph problems.

However, for Euler’s Sieve, the simulation results show that the performance of vector extension is even worse than the original settings. We think there are two reasons that may cause this situation:

1. Too many multiplications of small integer such as 2, 3 and 5 exist in calculation. For every composite integer X with its least prime factor P, we only sieve numbers in forms of X*Q s.t. Q <= P is a prime. Thus, for most numbers, the multiplication times are not large enough to benefit from this vector extension.
2. Vector executes 4 multiplication in parallel in one circle while non-vector executes 1 serial multiplication per cycle. This is the basic logic of vector accelerator. But since we don't implement pipeline in the simulator, all instructions use 1 cycle equally. Henceforth, the optimization by reducing cycle-consuming instructions will not be shown in the results.

Sieve	Int (Non-vector)	Int (vector)
1000	10419000	13643000
100000	1226985500	1529949500
10000000	105233622500	152039048000

Table 5: Original data of Sieve benchmark evaluation

Figure 5: Sieve Evaluation Result

These three evaluation cases represent 3 common situation which RISC-V might run into:

1. 3mm is a standard matrix multiplication which will obviously benefit from vector accelerator. In our implementation, stride-load instructions are used to optimize the inner loop.

2. Floyd has higher data dependence and a branch inside the loop, which are challenges to SIMD implementations.

3. Euler's sieve is a quite challenging scenario with very high data dependency and complex control flow, which was considered not able to be vectorized. It might only gain a little performance in the real-world CPU but definitely will perform worse in the simulator.

5. Conclusion

In this project, we implemented vector extensions to RISC-V in gem5, an architecture that supports data level parallel execution. Under the existing vector extension proposal, we completed a subset of components, including vector CSRs, vector operands, and vector instructions. Overall, RISC-V vector extension will gain around more than 90% performance compared to original configuration. We use three different types of test cases to evaluate how will the vector extension react to unpredictable programs. For this version of proposal, future work such as additional vector CSRs and vector instructions can be done to get a well-designed implementation. Besides, the proposal is keeping updated and newer versions are also available, it's worthwhile to compare performance between those versions of draft.

6. Statement of Work

Every member of our team made a valuable contribution to this project, with each has following emphasis:
Xinyu Ma: cectorization of benchmarks, help adding vector instructions.
Zhaoning Kong: working on gem5 to add vector instructions support.
Weijia Yuan: evaluation and analyze the result of test case. Construct the report and introduction.
Yajun Shi: constructing the report, presenting background information, related work, and implementation details.

References

[1] RISC5: Implementing the RISC-V ISA in gem5. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. https://carrv.github.io/2017/papers/roelke-risc5-carrv2017.pdf
[2] Design of the RISC-V Instruction Set Architecture. Technical Report No. UCB/EECS-2016-1 http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html
[3] RISC-V, riscv-v-spec, 2018. GitHub repository, https://github.com/riscv/riscv-v-spec
[4] PolyBench 4.0 http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/polybench.pdf