# Artifact Evaluation Guide for Paper #58: Zoozve: A Strip-Mining-Free RISC-V Vector Extension with Arbitrary Register Grouping Compilation Support (WIP)

Siyi Xu, Limin Jiang, Yintao Liu, Yihao Shen, Yi Shi, Shan Cao, Zhiyuan Jiang

## 1 Introduction

This document provides a comprehensive guide for reviewers to interact with the artifact associated with Paper #58: Zoozve: A Strip-Mining-Free RISC-V Vector Extension with Arbitrary Register Grouping Compilation Support (WIP) at the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2025). Due to the large size of the artifact, we have provided three access methods for convenience:

- GitHub Repository: This option allows reviewers to compile the artifact on their own machine. The repository includes all necessary code and dependencies.
- Zenodo with Docker Environment: For a streamlined experience, we have also uploaded a pre-configured Docker image containing the entire compilation environment. This ensures that reviewers can bypass manual setup and focus on the evaluation itself.
- Google Drive: If neither of the above methods successfully reproduces the results, reviewers can download the text files generated during the compilation process from Google Drive for evaluation.

In the following sections, we will provide step-by-step instructions for both methods. After the environment is successfully deployed, reviewers will be able to compile and test the code, and compare the results with the claims presented in the paper.

## 2 Getting Started Guide

### 2.1 GitHub Repository

To compile the artifact locally, please use the GitHub repository provided. The repository is divided into two main parts:

- Zoozve Compiler: This part includes our custom modifications to LLVM, forming the core of the artifact.
- Official RISC-V GNU Toolchain: This part contains the official RISC-V GNU toolchain, which is linked to the Zoozve repository to provide necessary functionality.

To begin, install the necessary dependencies and clone the repository to your local machine using the following commands:

```
sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip libmpc-dev libmpfr-
                                                    dev libgmp-dev gawk build-essential bison flex
                                                    texinfo gperf libtool patchutils bc zlib1g-dev
                                                    libexpat-dev ninja-build git cmake libglib2.0-dev
                                                     ccache
git clone https://github.com/ACELab-SHU/LCTES-25-Artifact.git
cd LCTES-25-Artifact
git submodule update --init --remote riscv-gnu-toolchain
```

Once the repositories are cloned, navigate to the directory of the RISC-V GNU Toolchain to start the compilation process:

```
cd riscv-gnu-toolchain
./configure --with-arch=rv32im --prefix=/path/to/install --with-abi=ilp32
make -j16
```

After successfully compiling the RISC-V GNU Toolchain, the next step is to compile the Zoozve Compiler

```
cd ../zoozve
mkdir -p build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=/path/to/install -
                                        DLLVM_TARGETS_TO_BUILD="RISCV" -
                                        DLLVM_CCACHE_BUILD=ON -DLLVM_ENABLE_PROJECTS="
                                        clang;llvm" -DLLVM_USE_LINKER=gold -
                                        DLLVM_DEFAULT_TARGET_TRIPLE="riscv32-unknown-elf"
                                         ../llvm && ninja install
```

The process may take some time depending on your system. Once the compilation is complete, you will have successfully set up the Zoozve Compiler environment, ready for testing and evaluation.

## 2.2 Zenodo with Docker Environment

The Zoozve Docker environment can be obtained from here. Once downloaded, please proceed with extracting the files:

```
gunzip lctes25.tar.gz
```

If Docker is not available, please follow the instructions below:

```
sudo apt-get clean
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/
                                        docker.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://
                                        download.docker.com/linux/ubuntu $(lsb_release -
                                        cs) stable" | sudo tee /etc/apt/sources.list.d/
                                        docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
sudo service docker restart
sudo service docker status # Check if Docker is active
```

Once the Docker environment is set up, please use the following command to mount the image:

```
sudo docker load < lctes25.tar
sudo docker run --interactive --tty --detach --name lctes25 --privileged=true lctes25:latest
sudo docker exec -it lctes25 bash
```

At this point, the image has been successfully mounted. By running the

```
ls
```

command, you should see a `zoozve_debug` folder in the current directory.

# 3 Step-by-Step Instructions

After successfully setting up the Zoozve Compiler and RISC-V GNU Toolchain, you can begin testing the compiled code. Follow the steps below to run the tests and compare the results with the claims made in the paper.

The first step is to navigate to the `venus_test` directory, where the test scripts and related files are located. Use the following command:

```
cd zoozve/venus_test # For Github Users
cd ~/venus_test # For Docker Users
```

The folder contains a `test.c` file, which implements a 2048-point fast Fourier transform (FFT) used for orthogonal frequency division multiplexing (OFDM) modulation and demodulation in wireless communication. The algorithm performs butterfly operations in full parallel for the 2048 points and then reorders the computed results. Below is a code snippet from `test.c`:

```
// Stage 0
vshuffle(data_real_down, move_2048to1024, Data_without_CP_real, SHUFFLE_GATHER, calculate_length);
vshuffle(data_imag_down, move_2048to1024, Data_without_CP_imag, SHUFFLE_GATHER, calculate_length);
// a + b
tempAddResult_real = vsadd(Data_without_CP_real, data_real_down, MASKREAD_OFF, calculate_length);
tempAddResult_imag = vsadd(Data_without_CP_imag, data_imag_down, MASKREAD_OFF, calculate_length);
tempAddResult_real = vsra(tempAddResult_real, fft_shift_vec[0], MASKREAD_OFF, calculate_length);
tempAddResult_imag = vsra(tempAddResult_imag, fft_shift_vec[0], MASKREAD_OFF, calculate_length);
```

The `vshuffle`, `vsadd`, and `vsra` functions in the code snippet correspond to **Step 1** in **Figure 2(c)** of the paper. These are all built-in functions defined in Clang. Additionally, some of the vector variables in the code, such as `tempAddResult_real`, have the data type `__v2500i8`, which represents a sequence of 2500 `int8` values. These data types are also defined within Clang.

In the current directory, run:

```
make all
```

to compile the code. For GitHub users, its important to adjust the `LLVMPATH` and `RVPATH` variables in the `Makefile` to the paths set in 2.1. The `make` command, after a series of operations, compiles the source code into executable files and disassembly files for the target platform. Next, we will explain the process step by step.

First, the source code is compiled by the Clang compiler frontend to generate an intermediate representation (IR) files, named `test.0.ll`. Below is a code snippet:

```
%165 = call <2500 x i8> @llvm.venus.shuffle.test.v2500i8.v2048i16.v2500i8.i32.i32(<2048 x i16> %
                                                  160, <2500 x i8> %161, <2500 x i8> %162, i32 0,
                                                  i32 1, i32 %164, i32 10)
store <2500 x i8> %165, ptr %data_real_down, align 4096
%166 = load <2500 x i8>, ptr %data_imag_down, align 4096
%167 = load <2048 x i16>, ptr %move_2048to1024, align 4096
%168 = load <2500 x i8>, ptr %Data_without_CP_imag, align 4096
%169 = load i16, ptr %calculate_length, align 2
%170 = load <2048 x i16>, ptr %move_2048to1024, align 4096
%171 = load <2500 x i8>, ptr %Data_without_CP_imag, align 4096
%172 = load <2500 x i8>, ptr %data_imag_down, align 4096
%173 = load i16, ptr %calculate_length, align 2
%174 = zext i16 %173 to i32
%175 = call <2500 x i8> @llvm.venus.shuffle.test.v2500i8.v2048i16.v2500i8.i32.i32(<2048 x i16> %
                                                  170, <2500 x i8> %171, <2500 x i8> %172, i32 0,
                                                  i32 1, i32 %174, i32 10)
store <2500 x i8> %175, ptr %data_imag_down, align 4096
%176 = load <2500 x i8>, ptr %Data_without_CP_real, align 4096
%177 = load <2500 x i8>, ptr %data_real_down, align 4096
%178 = load i16, ptr %calculate_length, align 2
%179 = load <2500 x i8>, ptr %Data_without_CP_real, align 4096
%180 = load <2500 x i8>, ptr %data_real_down, align 4096
%181 = load i16, ptr %calculate_length, align 2
%182 = zext i16 %181 to i32
%183 = call <2500 x i8> @llvm.venus.sadd.ivv.v2500i8.i32.i32(<2500 x i8> %179, <2500 x i8> %180,
                                                  i32 0, i32 0, i32 %182, i32 10)
store <2500 x i8> %183, ptr %tempAddResult_real, align 4096
%184 = load <2500 x i8>, ptr %Data_without_CP_imag, align 4096
%185 = load <2500 x i8>, ptr %data_imag_down, align 4096
%186 = load i16, ptr %calculate_length, align 2
%187 = load <2500 x i8>, ptr %Data_without_CP_imag, align 4096
%188 = load <2500 x i8>, ptr %data_imag_down, align 4096
%189 = load i16, ptr %calculate_length, align 2
%190 = zext i16 %189 to i32
%191 = call <2500 x i8> @llvm.venus.sadd.ivv.v2500i8.i32.i32(<2500 x i8> %187, <2500 x i8> %188,
                                                  i32 0, i32 0, i32 %190, i32 10)
store <2500 x i8> %191, ptr %tempAddResult_imag, align 4096
%192 = load <2500 x i8>, ptr %tempAddResult_real, align 4096
%193 = load i16, ptr @fft_shift_vec, align 2
%194 = load i16, ptr %calculate_length, align 2
%195 = load <2500 x i8>, ptr %tempAddResult_real, align 4096
%196 = load i16, ptr @fft_shift_vec, align 2
%197 = zext i16 %196 to i32
%198 = load i16, ptr %calculate_length, align 2
%199 = zext i16 %198 to i32
```

```
%200 = call <2500 x i8> @llvm.venus.sra.ivx.v2500i8.i32.i32(<2500 x i8> %195, i32 %197, i32 0, i32
                                                 0, i32 %199, i32 10)
store <2500 x i8> %200, ptr %tempAddResult_real, align 4096
%201 = load <2500 x i8>, ptr %tempAddResult_imag, align 4096
%202 = load i16, ptr @fft_shift_vec, align 2
%203 = load i16, ptr %calculate_length, align 2
%204 = load <2500 x i8>, ptr %tempAddResult_imag, align 4096
%205 = load i16, ptr @fft_shift_vec, align 2
%206 = zext i16 %205 to i32
%207 = load i16, ptr %calculate_length, align 2
%208 = zext i16 %207 to i32
%209 = call <2500 x i8> @llvm.venus.sra.ivx.v2500i8.i32.i32(<2500 x i8> %204, i32 %206, i32 0, i32
                                                 0, i32 %208, i32 10)
```

As seen, the original built-in functions (`vshuffle`) are converted into LLVM intrinsic functions (`@llvm.venus.shuffle.test.v2500i8.v2048i16.v2500i8.i32.i32`), while retaining the variable types from the source code (`<2500 x i8>`). This part corresponds to **Step 2** in **Figure 2(c)** of the paper.

Next, the LLVM `opt` tool splits the existing intrinsics. The opt tool primarily uses the `mem2reg` and `venusplit` passes for optimization, and the optimized result is output to `test.split.ll`. Below is a code snippet:

```
call void @llvm.venus.delimit(i32 10)
%1030 = call <256 x i8> @llvm.venus.shuffle.test.v256i8.v128i16.v256i8.i32.i32(<128 x i16> %897, <
                                                256 x i8> %757, <256 x i8> %929, i32 0, i32 1,
                                                i32 %1029, i32 10)
%1031 = call <256 x i8> @llvm.venus.shuffle.test.v256i8.v128i16.v256i8.i32.i32(<128 x i16> %898, <
                                                256 x i8> %758, <256 x i8> %930, i32 0, i32 1,
                                                i32 %1029, i32 10)
... # Eight more split shuffle intrinsics
call void @llvm.venus.delimit(i32 0)
...
%1040 = zext i16 %calculate_length.1 to i32
call void @llvm.venus.delimit(i32 10)
%1041 = call <256 x i8> @llvm.venus.shuffle.test.v256i8.v128i16.v256i8.i32.i32(<128 x i16> %897, <
                                                256 x i8> %767, <256 x i8> %939, i32 0, i32 1,
                                                i32 %1040, i32 10)
%1042 = call <256 x i8> @llvm.venus.shuffle.test.v256i8.v128i16.v256i8.i32.i32(<128 x i16> %898, <
                                                256 x i8> %768, <256 x i8> %940, i32 0, i32 1,
                                                i32 %1040, i32 10)
... # Eight more split shuffle intrinsics
call void @llvm.venus.delimit(i32 0)
...
%1051 = zext i16 %calculate_length.1 to i32
call void @llvm.venus.delimit(i32 10)
%1052 = call <256 x i8> @llvm.venus.sadd.ivv.v256i8.i32.i32(<256 x i8> %757, <256 x i8> %1030, i32
                                                 0, i32 0, i32 %1051, i32 10)
%1053 = call <256 x i8> @llvm.venus.sadd.ivv.v256i8.i32.i32(<256 x i8> %758, <256 x i8> %1031, i32
                                                 0, i32 0, i32 %1051, i32 10)
... # Eight more split sadd intrinsics
call void @llvm.venus.delimit(i32 0)
%1062 = zext i16 %calculate_length.1 to i32
call void @llvm.venus.delimit(i32 10)
%1063 = call <256 x i8> @llvm.venus.sadd.ivv.v256i8.i32.i32(<256 x i8> %767, <256 x i8> %1041, i32
                                                 0, i32 0, i32 %1062, i32 10)
%1064 = call <256 x i8> @llvm.venus.sadd.ivv.v256i8.i32.i32(<256 x i8> %768, <256 x i8> %1042, i32
                                                 0, i32 0, i32 %1062, i32 10)
... # Eight more split sadd intrinsics
call void @llvm.venus.delimit(i32 0)
%1073 = load i16, ptr @fft_shift_vec, align 2
%1074 = load i16, ptr @fft_shift_vec, align 2
%1075 = zext i16 %1074 to i32
%1076 = zext i16 %calculate_length.1 to i32
call void @llvm.venus.delimit(i32 10)
%1077 = call <256 x i8> @llvm.venus.sra.ivx.v256i8.i32.i32(<256 x i8> %1052, i32 %1075, i32 0, i32
                                                 0, i32 %1076, i32 10)
%1078 = call <256 x i8> @llvm.venus.sra.ivx.v256i8.i32.i32(<256 x i8> %1053, i32 %1075, i32 0, i32
                                                 0, i32 %1076, i32 10)
... # Eight more split sra intrinsics
call void @llvm.venus.delimit(i32 0)
%1087 = load i16, ptr @fft_shift_vec, align 2
%1088 = load i16, ptr @fft_shift_vec, align 2
%1089 = zext i16 %1088 to i32
```

```
%1090 = zext i16 %calculate_length.1 to i32
call void @llvm.venus.delimit(i32 10)
%1091 = call <256 x i8> @llvm.venus.sra.ivx.v256i8.i32.i32(<256 x i8> %1063, i32 %1089, i32 0, i32
                                                         0, i32 %1090, i32 10)
%1092 = call <256 x i8> @llvm.venus.sra.ivx.v256i8.i32.i32(<256 x i8> %1064, i32 %1089, i32 0, i32
                                                         0, i32 %1090, i32 10)
... # Eight more split sra intrinsics
call void @llvm.venus.delimit(i32 0)
```

After the split, the intrinsic is surrounded by two `delimiter` intrinsics. The number of splits depends on the hardware parallelism, which can be configured by adjusting the `VENUSROW` and `VENUSLANE` variables in the `Makefile`. These variables represent the number of hardware registers and the parallelism, respectively. Currently, the configuration sets the length of one physical vector register to 256 bytes. For a type of `__v2500i8`, it requires $\lceil 2500/256 \rceil = 10$ registers. The split operation corresponds to **Step 3** in **Figure 2(c)** of the paper. For hardware design details, please refer to this paper.

Subsequently, the LLVM `llc` tool converts the virtual registers into physical registers using the modified register allocation algorithm, with the output file being `test_before_merge.s`. This step corresponds to **Step 4** in **Figure 2(c)** of the paper. Below is a snippet of the code:

```
vns_shuffle.test  vns76, vns30, vns60, 0, 1, a3, 10
vns_shuffle.test  vns77, vns31, vns61, 0, 1, a3, 10
... # Eight more split shuffle assemblies
vns_shuffle.test  vns86, vns40, vns60, 0, 1, a3, 10
vns_shuffle.test  vns87, vns41, vns61, 0, 1, a3, 10
... # Eight more split shuffle assemblies
vns_sadd.ivv  vns0, vns76, vns30, 0, 0, a3, 10
vns_sadd.ivv  vns1, vns77, vns31, 0, 0, a3, 10
... # Eight more split sadd assemblies
vns_sadd.ivv  vns10, vns86, vns40, 0, 0, a3, 10
vns_sadd.ivv  vns11, vns87, vns41, 0, 0, a3, 10
... # Eight more split sadd assemblies
lui a1, %hi(fft_shift_vec)
lhu a0, %lo(fft_shift_vec)(a1)
vns_sra.ivx vns20, a0, vns0, 0, 0, a3, 10
vns_sra.ivx vns21, a0, vns1, 0, 0, a3, 10
... # Eight more split sra assemblies
lhu a0, %lo(fft_shift_vec)(a1)
vns_sra.ivx vns50, a0, vns10, 0, 0, a3, 10
vns_sra.ivx vns51, a0, vns11, 0, 0, a3, 10
... # Eight more split sra assemblies
```

Finally, we added a pass for physical register coalescing within `llc`, and the final output file is `test.s`. This assembly file is then compiled into an executable for the target hardware. This part corresponds to **Step 5** in **Figure 2(c)** of the paper. Below is a snippet of the code:

```
vns_shuffle.test  vns76, vns30, vns60, 0, 1, a3, 10
vns_shuffle.test  vns86, vns40, vns60, 0, 1, a3, 10
vns_sadd.ivv  vns0, vns76, vns30, 0, 0, a3, 10
vns_sadd.ivv  vns10, vns86, vns40, 0, 0, a3, 10
lui a1, %hi(fft_shift_vec)
lhu a0, %lo(fft_shift_vec)(a1)
vns_sra.ivx vns20, a0, vns0, 0, 0, a3, 10
lhu a0, %lo(fft_shift_vec)(a1)
vns_sra.ivx vns50, a0, vns10, 0, 0, a3, 10
```

This concludes the entire artifact evaluation.