Project Report

On

RISC-V & HPC

Submitted
In partial fulfilment

For the award of the Degree of

PG-Diploma in Embedded Systems and Design (PG-DESD)

C-DAC, ACTS (Pune)

Guided By:

Mr. Surendra Billa **Mr. Hrushikesh Jadhav**

Submitted By:

NAME	PRN
Mr. Katkar Tejas Bhanudas Tara	230940130035
Mr. Bhanu Praskash Agrawal	230940130015
Ms. Muskan Sharma	230940130037
Ms. Ghorpade Akanksha Abhay	230940130025

Centre for Development of Advanced Computing (C-DAC), ACTS

(Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Mr. Surendra Billa** and **Mr. Hrushikesh Jadhav** C-DAC ACTS, Pune for her constant guidance and helpful suggestion for preparing this project **RISC-V & HPC**. We express our deep gratitude towards her for inspiration, personal involvement, constructive criticism that she provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Risha P R (Program Head) Mrs. Srujana Bhamidi** (Course Coordinator, PG-DESD) for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

NAME	**PRN**
Mr. Katkar Tejas Bhanudas Tara	**230940130035**
Mr. Bhanu Praskash Agrawal	**230940130015**
Ms. Muskan Sharma	**230940130037**
Ms. Ghorpade Akanksha Abhay	**230940130025**

ABSTRACT

The RISC-V Vector Extension (RVV) is a pivotal advancement in processor architecture, designed to enhance performance for a wide range of computational tasks. This abstract explores the key features and benefits of the RVV, including its scalable vector length, customizable data types, and support for parallel processing. Additionally, it examines the potential impact of RVV on various application domains, such as scientific computing, machine learning, and multimedia processing. By providing a flexible and efficient framework for vector operations, the RVV promises to revolutionize the landscape of processor design and accelerate the development of next-generation computing systems.

Table of Contents

Chapter 1
Introduction

1.1 Introduction

RISC-V, pronounced "risk-five", is an open-source instruction set architecture (ISA) shaking up the world of processor design. Unlike traditional architectures controlled by specific companies, RISC-V is freely available and modular, allowing for customization and innovation at every level. This introduction will guide you through the key features of RISC-V and its potential impact on the future of computing. Dive into the world of parallel processing with RISC-V Vector Extension (RVV), an exciting addition to the open-source RISC-V instruction set architecture. RVV equips RISC-V processors with the ability to perform operations on multiple data elements simultaneously, boosting performance for numerous applications like multimedia, signal processing, and scientific computing.

The open-source nature of RISC-V allows for customization and experimentation at every level, including building your own toolchain for the exciting RISC-V Vector Extension (RVV). This introduction will guide you through the process, highlighting key steps and resources to get started.

RVV is still evolving, with ongoing development and optimization. Its open-source nature and vast potential make it a key player in shaping the future of vector processing across various computing domains.

By understanding RVV, we can equip our self with knowledge about a powerful technology poised to revolutionize how we handle data and tackle demanding computational tasks.

Chapter 2

LITERATURE REVIEW

Zhenhao Li : Wei Hu : Shuang Chen [1] Design and Implementation of CNN Custom Processor Based on RISC-V Architecture

With the rapid development of CNN (convolutional neural networks), the traditional CPU platform cannot make full use of the parallelism of CNN. We decide to adopt a new and popular processor architecture: the risc-v architecture for experimental design. In this paper, a new convolutional neural network processor is designed based on risc-v architecture. The processor can take advantage of the parallelism of CNN and is more flexible. This paper completely designed a CNN processor based on the risc-v architecture. The processor uses a classic five-stage pipeline structure, and implements instruction buffer memory and data buffer memory, and adds peripherals such as FLASH, SRAM, and SDRAM. And, this paper designed custom instructions. Given the convolution operation frequently occurring in CNN, vector store instruction, vector load instruction, vector addition instruction, and convolution operation instruction are designed to accelerate the execution of the convolution process. The design has passed the simulation experiment. It can not only complete the general instructions but also run the custom instructions. The final simulation test verified the correctness of the design.

Zhenhao Li : Wei Hu : Shuang Chen [2] Performance Modelling-driven Optimization of RISC-V Hardware for Efficient SpMV

The growing need for inference on edge devices brings with it a necessity for efficient hardware, optimized for particular computational kernels, such as Sparse Matrix-Vector Multiplication (SpMV). With the RISC-V Instruction Set Architecture (ISA) providing unprecedented freedom to hardware designers, there is now a greater opportunity to tailor these microarchitectures to both the application requirements and the data it is expected to

process

Chapter 3

Methodology and Techniques

3.1 INTRODUCTION

The project involves implementing the RISC-V vector extension on an emulator and developing a cross-compilation toolchain. This extension enables efficient parallel processing of data, enhancing the performance of RISC-V processors for vectorized workloads. The emulator will simulate the execution of vectorized instructions, providing a platform for testing and debugging vectorized code. Concurrently, the cross-compilation toolchain will facilitate the development of vectorized applications by allowing developers to compile code on one architecture (e.g., x86) for execution on RISC-V processors with vector extensions. Overall, the project aims to advance the accessibility and utilization of RISC-V vector processing capabilities for various computational tasks.

3.1.1 Cross Compilation

Cross-compilation refers to the process of compiling code on one architecture or platform (the host) to run on a different architecture or platform (the target). This is commonly done when developing software for embedded systems, mobile devices, or different operating systems.

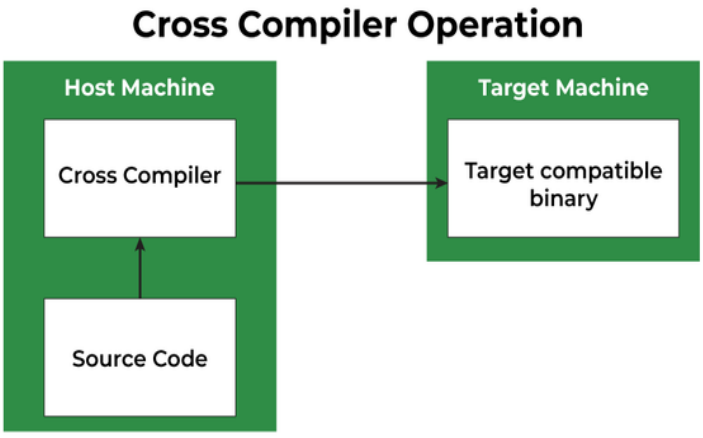


Fig 3.1 Cross Compiler Operation

Cross-compilers operate by examining the source code and producing machine code tailored for a different processor and/or operating system than that of the compilation host. These compilers facilitate the development of software for platforms distinct from the one on which the compilation occurs. Similar to native compilers, cross-compilers can implement optimizations like loop unrolling, function inlining, and instruction scheduling to enhance the performance and efficiency of the generated code for the target system.

In the context of RISC-V vector extension toolchain where RISC-V is the target and x86 is the host, cross-compilation involves compiling software on an x86-based system (the host) to run on a RISC-V processor (the target). The cross-compilation toolchain includes:

Cross Compiler: This is a compiler specifically configured to generate machine code for the target architecture (RISC-V) while running on the host architecture (x86). It translates high-level source code (such as C or C++) into machine code that the target architecture can execute

Cross Assembler: Similar to the compiler, the cross-assembler translates assembly language code written for the target architecture into machine code while running on the host architecture.

Cross linker: The cross-linker is responsible for linking together object files and libraries generated by the cross-compiler into an executable binary that can run on the target architecture.

Cross debugger: This tool helps in debugging programs compiled for the target architecture. It allows developers to step through the code, inspect variables, and analyze runtime behavior, all while running on the host system.

Cross-compilation allows developers to take advantage of the more powerful and familiar development environment of the host system while targeting different architectures or platforms. It's particularly useful for embedded systems where the target hardware might not have the resources or tooling necessary for compiling software directly on the device. Additionally, cross-compilation can improve development efficiency by enabling faster compilation and testing cycles.

3.1.2 QEMU

QEMU (Quick Emulator) is a free and open-source emulator. It emulates a computer's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. It can interoperate with Kernel-based Virtual Machine (KVM) to run virtual machines at near-native speed. QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.[4]

QEMU supports the emulation of various architectures, including x86, ARM, PowerPC, RISC-V, and others.

QEMU can emulate network cards (of different models) which share the host system's connectivity by doing network address translation, effectively allowing the guest to use the same network as the host. The virtual network cards can also connect to network cards of other instances of QEMU or to local TAP interfaces. Network connectivity can also be achieved by bridging a TUN/TAP interface used by QEMU with a non-virtual Ethernet interface on the host OS using the host OS's bridging features.

QEMU integrates several services to allow the host and guest systems to communicate; for example, an integrated SMB server and network-port redirection (to allow incoming connections to the virtual machine). It can also boot Linux kernels without a boot loader.

QEMU does not depend on the presence of graphical output methods on the host system. Instead, it can allow one to access the screen of the guest OS via an integrated VNC server. It can also use an emulated serial line, without any screen, with applicable operating systems.

Simulating multiple CPUs running SMP is possible.

QEMU does not require administrative rights to run unless additional kernel modules for improving speed (like KQEMU) are used or certain modes of its network connectivity model are utilized.

3.2 FLOWCHART

3.2.1 Developing RISC-V Vector extension Toolchain

The following flowchart shows flow of developing the RISC-V vector extension toolchain:

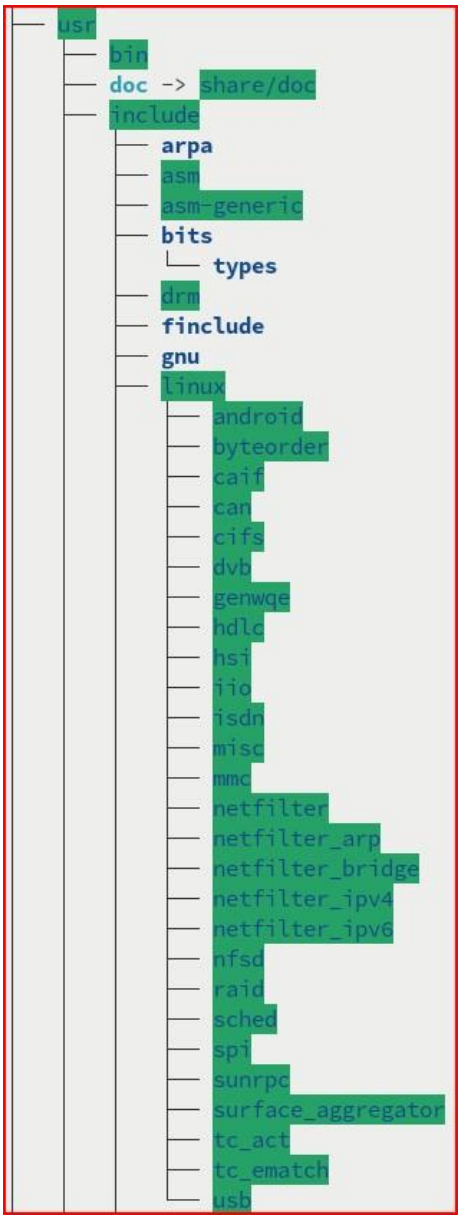
1. Start
2. |
3. |----- **Set Up Development Environment**
4. | |
5. | |----- Install Required Dependencies (e.g., build tools, libraries)
6. | |
7. | |----- Set Up RISC-V Toolchain Sources
8. | |
9. | |----- Set Up RVV Extension Sources
10. | |
11. |----- **Configure Toolchain**
12. | |
13. | |----- Run configure script (e.g., ./configure)
14. | |
15. | |----- Specify Target Architecture (e.g., RISC-V RV64GV)
16. | |
17. | |----- Specify Installation Directory
18. | |
19. | |----- Enable RVV Extension Support
20. | |
21. |----- **Build Toolchain**
22. | |
23. | |----- Run make command
24. | |
25. | |----- Compilation of Binutils (assembler, linker, etc.)
26. | |
27. | |----- Compilation of GCC (compiler)
28. | |
29. | |----- Compilation of Runtime Libraries
30. | |
31. | |----- Compilation of RVV Extension Support
32. | |

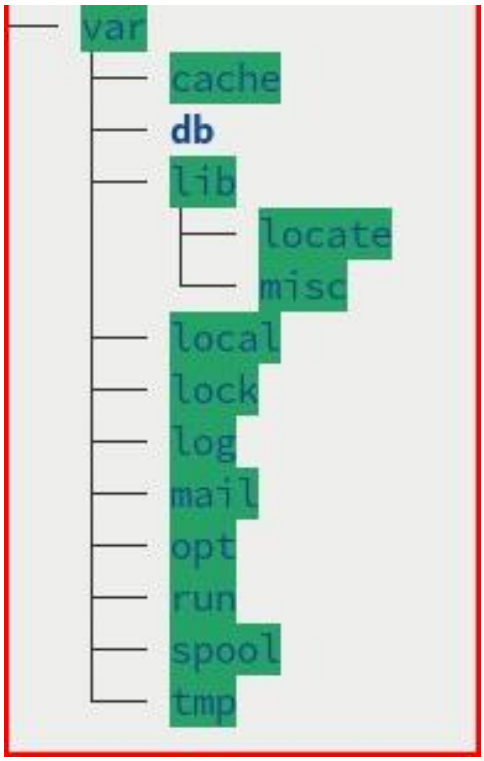
33. |----- **Install Toolchain**
34. | |
35. | |----- Run make install command
36. | |
37. | |----- Install Binutils
38. | |
39. | |----- Install GCC
40. | |
41. | |----- Install Runtime Libraries
42. | |
43. | |----- Install RVV Extension Support
44. |
45. **End**

Fig 3.2 Flowchart of developing toolchain

```
bhanu@fedora:~/Documents/tars_glibc/tars$ tree -d -L 2
.
├── binutils
│   ├── binutils-2.39
│   └── binutils-build
├── busybox
├── clfs
├── gcc
│   ├── gcc-13.0.1-20230401
│   ├── gcc-build
│   └── gcc-static
├── glibc
│   ├── glibc-2.37
│   └── glibc-build
├── gmp
│   └── gmp-6.2.1
├── isl
│   └── isl-0.24
├── linux
│   └── linux-6.7.3
├── mpc
│   └── mpc-0.34
├── mpfr
│   └── mpfr-4.1.1
└── zlib
```


bhanu@fedora:/lj-os$ tree -d -L 1
.
bin
boot
cross-tools
dev
etc
home
lib
lib64
media
mnt
opt
proc
root
sbin
srv
sys
tmp
usr
var
dev
etc
init.d
opt
home
lib
firmware
modules
lib64
lp64d
media
cdrom
floppy
mnt
opt
proc
root
sbin
srv
sys
tmp


usr
bin
doc -> share/doc
include
arpa
asm
asm-generic
bits
types
drm
finclude
gnu
linux
android
byteorder
caif
can
cifs
dvb
genwqe
hdlc
hsi
iio
isdn
misc
mmc
netfilter
netfilter_arp
netfilter_bridge
netfilter_ipv4
netfilter_ipv6
nfsd
raid
sched
spi
sunrpc
surface_aggregator
tc_act
tc_ematch
usb


var
cache
db
lib
locate
misc
local
lock
log
mail
opt
run
spool
tmp

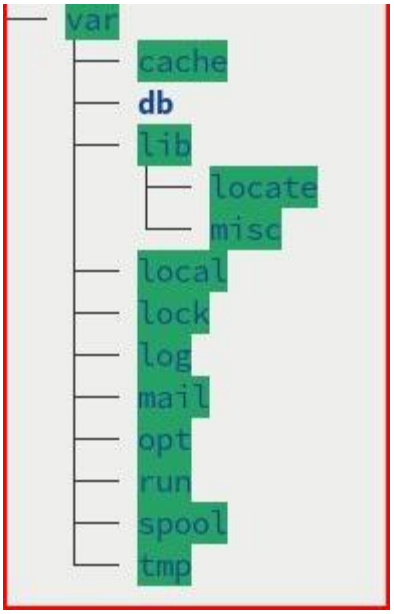
The following are the steps to build the cross compilation toolchain for RISC-V vector extension on x86 host machine:

- **Configure the build environment:** This involves setting up your host machine and creating the directories that will hold the cross-compilation toolchain and target image.

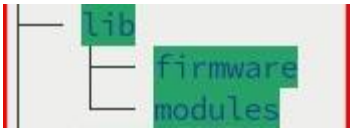
Create the target image's file system hierarchy: This involves creating the basic directory structure for your target image, including directories for things like

/bin, /boot 

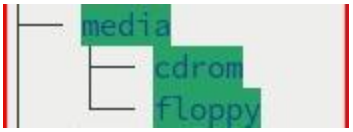
/usr 

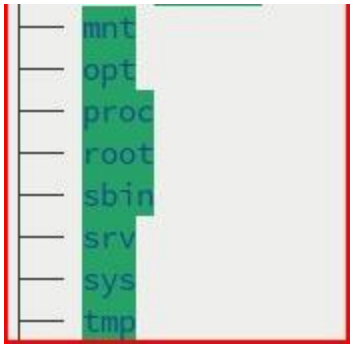


/var

/lib 

/lib64 

/media 

/mnt, /opt, /proc, /root, /sbin, /srv, /sys, /tmp 

/dev, /home, /etc 
.

- **Build the cross compiler:** This is a compiler that can be used to compile code for a different architecture than the one you are running on. You will need a cross compiler to build the kernel and other software for your target image.
- **Build the C library:** The C library is a collection of essential functions that are used by many programs. You will need to build a C library for your target image before you can build other software.

```
bhanu@fedora:~/Documents/tars_glibc/tars$ tree -d -L 2
.
├── binutils
│   ├── binutils-2.39
│   └── binutils-build
├── busybox
├── clfs
├── gcc
│   ├── gcc-13.0.1-20230401
│   ├── gcc-build
│   └── gcc-static
├── glibc
│   ├── glibc-2.37
│   └── glibc-build
├── gmp
│   └── gmp-6.2.1
├── isl
│   └── isl-0.24
├── linux
│   └── linux-6.7.3
├── mpc
│   └── mpc-0.34
├── mpfr
│   └── mpfr-4.1.1
└── zlib
```

- **Build the target image:** This involves compiling the kernel and other software for your target image, and then packaging it all up into a bootable image.
- **Install the target image on your target machine:** This can be done on a physical machine or a virtual machine.

3.2.2 RISC-V Vector extension on emulator

The following flowchart shows the flow of RISC-V vector extension on the emulator:

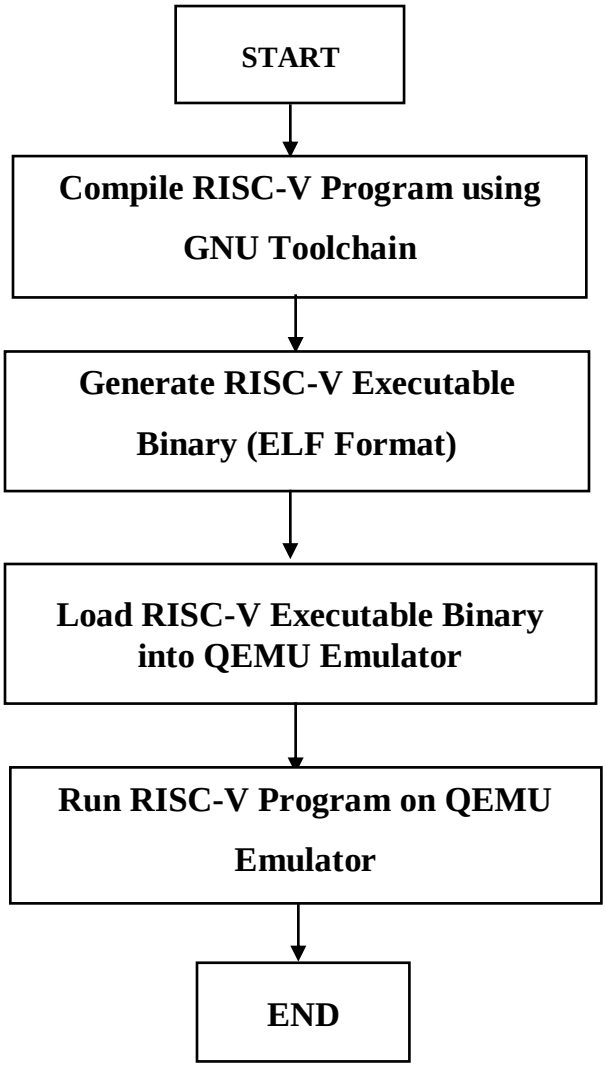


Fig 3.3 Flowchart of RISC-V Vector on the emulator

Compile RISC-V Program using GNU Toolchain:

- Write or obtain the source code of a RISC-V program.
- Use the RISC-V GNU toolchain, which includes the GNU Compiler Collection (GCC) for RISC-V, to compile the source code.

Generate RISC-V Executable Binary (ELF Format):

- After compilation, the RISC-V program is transformed into an executable binary in the ELF (Executable and Linkable Format) format.

Load RISC-V Executable Binary into QEMU Emulator:

- QEMU is a versatile emulator that supports various architectures, including RISC-V.
- Load the generated RISC-V executable binary into the QEMU emulator environment.

Run RISC-V Program on QEMU Emulator:

- Execute the loaded RISC-V program within the QEMU emulator environment.
- QEMU emulates the RISC-V instruction set architecture, allowing the program to run as if it were running on real RISC-V hardware.

3.3 PROCEDURE FOR DEVELOPING A RISC-V TOOLCHAIN

Configuring the Environment

Before beginning this process, you need to configure the build environment. First, turn on Bash hash functions:

$ set +h

Make sure that newly created files/directories are writable only by the owner

$ umask 022

You'll use your home directory as the main build directory. (this isn't a requirement). This is where the cross-compilation toolchain and target image will be installed and put into the lj-os subdirectory.

```
$ export LJOS=~/lj-os
$ mkdir -pv ${LJOS}
$ export LC_ALL=POSIX
$ export PATH=${LJOS}/cross-tools/bin:/bin:/usr/bin
```

After setting the above environment variables, create the target image's filesystem hierarchy:

```
$ mkdir -pv
${LJOS}/{bin,boot{,grub},dev,{etc/,}opt,home,lib/{firmware,modules},lib64,mnt}
$ cd lj-os
$ cd boot
$ mkdir grub
$ cd ../..
$ mkdir -pv ${LJOS}/{proc,media/{floppy,cdrom},sbin,srv,sys}
$ mkdir -pv ${LJOS}/var/{lock,log,mail,run,spool}
$ mkdir -pv ${LJOS}/var/{opt,cache,lib/{misc,locate},local}
$ install -dv -m 0750 ${LJOS}/root
$ install -dv -m 1777 ${LJOS}{/var,}/tmp
$ install -dv ${LJOS}/etc/init.d
$ mkdir -pv ${LJOS}/usr/{,local/}{bin,include,lib{,64},sbin,src}
$ mkdir -pv ${LJOS}/usr/{,local/}share/{doc,info,locale,man}
$ mkdir -pv ${LJOS}/usr/{,local/}share/{misc,terminfo,zoneinfo}
$ mkdir -pv ${LJOS}/usr/{,local/}share/man/man{1,2,3,4,5,6,7,8}
$ for dir in ${LJOS}/usr{,/local}; do      ln -sv share/{man,doc,info} ${dir};    done
```

This directory tree is based on the Filesystem Hierarchy Standard (FHS), Create the directory for a cross-compilation toolchain:

$ install -dv ${LJOS}/cross-tools{,/bin}

Use a symlink to /proc/mounts to maintain a list of mounted filesystems properly in the /etc/mtab file:

$ ln -svf ../proc/mounts ${LJOS}/etc/mtab

Then create the /etc/passwd file, listing the root user account

$ cat > ${LJOS}/etc/passwd << "EOF"
root::0:0:root:/root:/bin/ash
EOF

Create the /etc/group file with the following command:

$ cat > ${LJOS}/etc/group << "EOF"
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tty:x:4:
daemon:x:6:
disk:x:8:
dialout:x:10:
video:x:12:
utmp:x:13:
usb:x:14:
EOF

The target system's /etc/fstab:

```
$ cat > ${LJOS}/etc/fstab << "EOF"
# file system  mount-point  type   options         dump  fsck
#                                                  order

rootfs         /             auto    defaults       1     1
proc           /proc         proc    defaults       0     0
sysfs          /sys          sysfs   defaults       0     0
devpts         /dev/pts      devpts  gid=4,mode=620 0     0
tmpfs          /dev/shm      tmpfs   defaults       0     0
EOF
```

The target system's /etc/profile to be used by the Almquist shell (ash) once the user is logged in to the target machine:

```
$ cat > ${LJOS}/etc/profile << "EOF"
export PATH=/bin:/usr/bin
if [ `id -u` -eq 0 ] ; then
      PATH=/bin:/sbin:/usr/bin:/usr/sbin
      unset HISTFILE
fi
# Set up some environment variables.
export USER=`id -un`
export LOGNAME=$USER
export HOSTNAME=`/bin/hostname`
export HISTSIZE=1000
export HISTFILESIZE=1000
export PAGER='/bin/more '
```

```
export EDITOR='/bin/vi'
EOF

$echo "ljos-test" > ${LJOS}/etc/HOSTNAME

$cat > ${LJOS}/etc/issue<< "EOF"
Linux Journal OS 0.1a
Kernel \r on an \m

EOF
```

You won't use systemd here. Instead, you'll use the basic init process provided by BusyBox. This requires that you define an /etc/inittab file:

```
$ cat > ${LJOS}/etc/inittab<< "EOF"
::sysinit:/etc/rc.d/startup

tty1::respawn:/sbin/getty 38400 tty1
tty2::respawn:/sbin/getty 38400 tty2
tty3::respawn:/sbin/getty 38400 tty3
tty4::respawn:/sbin/getty 38400 tty4
tty5::respawn:/sbin/getty 38400 tty5
tty6::respawn:/sbin/getty 38400 tty6

::shutdown:/etc/rc.d/shutdown
::ctrlaltdel:/sbin/reboot
EOF
```

Also as a result of leveraging BusyBox to simplify some of the most common Linux system functionality, you'll use mdev instead of udev, which requires you to define the following

/etc/mdev.conf file:

```
$ cat > ${LJOS}/etc/mdev.conf<< "EOF"
# Devices:
# Syntax: %s %d:%d %s
# devices user:group mode

# null does already exist; therefore ownership has to
# be changed with command
null    root:root 0666  @chmod 666 $MDEV
zero    root:root 0666
grsec   root:root 0660
full    root:root 0666

random  root:root 0666
urandom root:root 0444
hwrandom root:root 0660

# console does already exist; therefore ownership has to
# be changed with command
console root:tty 0600 @mkdir -pm 755 fd && cd fd && for x
 ↪in 0 1 2 3 ; do ln -sf /proc/self/fd/$x $x; done

kmem    root:root 0640
mem     root:root 0640
port    root:root 0640
ptmx    root:tty 0666

# ram.*
ram([0-9]*)     root:disk 0660 >rd/%1
loop([0-9]+)    root:disk 0660 >loop/%1
```

```
sd[a-z].*       root:disk 0660 */lib/mdev/usbdisk_link
hd[a-z][0-9]*   root:disk 0660 */lib/mdev/ide_links

tty             root:tty 0666
tty[0-9]        root:root 0600
tty[0-9][0-9]   root:tty 0660
ttyO[0-9]*      root:tty 0660
pty.*           root:tty 0660
vcs[0-9]*       root:tty 0660
vcsa[0-9]*      root:tty 0660

ttyLTM[0-9]     root:dialout 0660 @ln -sf $MDEV modem
ttySHSF[0-9]    root:dialout 0660 @ln -sf $MDEV modem
slamr           root:dialout 0660 @ln -sf $MDEV slamr0
slusb           root:dialout 0660 @ln -sf $MDEV slusb0
fuse            root:root  0666

# misc stuff
agpgart         root:root 0660  >misc/
psaux           root:root 0660  >misc/
rtc             root:root 0664  >misc/

# input stuff
event[0-9]+     root:root 0640 =input/
ts[0-9]         root:root 0600 =input/

# v4l stuff
vbi[0-9]        root:video 0660 >v4l/
video[0-9]      root:video 0660 >v4l/

# load drivers for usb devices
```

```
usbdev[0-9].[0-9]       root:root 0660 */lib/mdev/usbdev
usbdev[0-9].[0-9]_.*    root:root 0660
EOF
```

You'll need to create a /boot/grub/grub.cfg for the GRUB bootloader that will be installed on the target machine's physical or virtual HDD

```
$ cat > ${LJOS}/boot/grub/grub.cfg<< "EOF"
set default=0
set timeout=5
set root=(hd0,1)
menuentry "Linux Journal OS 0.1a" {
      linux   /boot/vmlinuz-4.16.3 root=/dev/sda1 ro quiet
}
EOF
```

Finally, initialize the log files and give them proper permissions:

```
$ touch ${LJOS}/var/run/utmp ${LJOS}/var/log/{btmp,lastlog,wtmp}

$ chmod -v 664 ${LJOS}/var/run/utmp ${LJOS}/var/log/lastlog
```

- **Building the Cross Compiler**

The cross compiler is a toolchain of various compilation tools built for the system on which it's executing but designed to compile for an architecture or microprocessor that's not necessarily compatible with the system on which you're using it. In my environment, I'm running a 64-bit x86 architecture (x86-64) and will be cross compiling to a RISC-V Vector extension target architecture.

You never can be too sure with what is set in a currently running environment, which is why you'll unset the following C and C++ flags:

$ unset CFLAGS
$ unset CXXFLAGS

Next, define the most vital parts of the host/target variables needed to create the cross-compiler toolchain and target image:

$ export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/_[^-]*/-cross/")
$ export LJOS_TARGET=riscv-unknown-linux-gnu
$ export LJOS_CPU=rv64gv_zfh
$ export LJOS_ARCH=riscv
$ export LJOS_ENDIAN=little
$ export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")

- **Kernel Headers**

The kernel's standard header files need to be installed for the cross compiler. Uncompress the kernel tarball and change into its directory. Then run:

$ make ARCH=${LJOS_ARCH} INSTALL_HDR_PATH=dest headers_install

$ cp -rv dest/include/* ${LJOS}/usr/include

- **Binutils**

Binutils contains a linker, assembler and other tools needed to handle compiled object files. Uncompress the tarball. Then create the binutils-build directory and change into it:

$ mkdir binutils-build
$ cd binutils-build/

Then run :

**$ ../binutils-2.30/configure --prefix=${LJOS}/cross-tools **
**--target=${LJOS_TARGET} --with-sysroot=${LJOS} **
--disable-nls --enable-shared --disable-multilib

$ make configure-host && make
$ ln -sv lib ${LJOS}/cross-tools/lib64
$ make install

Copy over the following header file to the target's filesystem:

$ cp -v ../binutils-2.30/include/libiberty.h ${LJOS}/usr/include

- **GCC (Static)**

Before building the final cross-compiler toolchain, you first must build a statically compiled toolchain to build the C library (glibc) to which the final GCC cross compiler will link.
Uncompress the GCC tarball, and then uncompress the following packages and move them into the GCC root directory:

$ tar xjf gmp-6.1.2.tar.bz2
$ mv gmp-6.1.2 gcc-7.3.0/gmp
$ tar xJf mpfr-4.0.1.tar.xz
$ mv mpfr-4.0.1 gcc-7.3.0/mpfr

$ tar xzf mpc-1.1.0.tar.gz

$ mv mpc-1.1.0 gcc-7.3.0/mpc

Now create a gcc-static directory and change into it:

$ mkdir gcc-static

$ cd gcc-static/

now Run the following commands

$ AR=ar LDFLAGS="-Wl,-rpath,${LJOS}/cross-tools/lib" ../packages/gcc-12.2.0/configure --prefix=${LJOS}/cross-tools --build=${LJOS_HOST} --host=${LJOS_HOST} --target=${LJOS_TARGET} --with-sysroot=${LJOS}/target --disable-nls --disable-shared --with-mpfr-include=$(pwd)/../packages/gcc-12.2.0/mpfr/src --with-mpfr-lib=$(pwd)/mpfr/src/.libs --without-headers --with-newlib --disable-decimal-float --disable-libgomp --disable-libmudflap --disable-libssp --disable-threads --enable-languages=c,c++ --disable-multilib --with-arch=${LJOS_CPU} --with-abi=lp64d

$ make all-gcc all-target-libgcc -j4

$ make all-gcc all-target-libgcc -j4

$ make all-gcc all-target-libgcc all-target-libstdc++-v3 -j4

Follow this process if libstdc++ is not getting configured with gcc-static :

$ cd gcc-static/

$ mkdir libstdc++

$ cd libstdc++/

$ ls

$ ../../packages/gcc-12.2.0/libstdc++-v3/configure --host=${LJOS_HOST} --target=${LJOS_TARGET} --prefix=${LJOS}/cross-tools --disable-multilib --disable-nls --disable-libstdcxx-pch

$ ls

$ make install

Now, Check whether libstdc++ is present in lj-os --> cross-tools

$ cd ../..

$ cd lj-os/cross-tools/

$ ls

$ cd lib64/

$ cd ../../..

$ ls

$ cd gcc-static/

Create a symbolic link in gcc-static:

$ ln -vs libgcc.a `${LJOS_TARGET}-gcc -print-libgcc-file-name | sed 's/libgcc/&_eh/'`

$ cd ..

- **Glibc**

Uncompress the glibc tarball. Then create the glibc-build directory and change into it:

$ tar xJf glibc-2.36.tar.xz

$ mkdir glibc-build

$ cd glibc-build/

Configure the following build flags:

$ echo "libc_cv_forced_unwind=yes" > config.cache

$ echo "libc_cv_c_cleanup=yes" >> config.cache
$ echo "libc_cv_ssp=no" >> config.cache
$ echo "libc_cv_ssp_strong=no" >> config.cache

Then run:

$ BUILD_CC="gcc" CC="${LJOS_TARGET}-gcc" AR="${LJOS_TARGET}-ar" RANLIB="${LJOS_TARGET}-ranlib" CFLAGS="-O2" ../packages/glibc-2.36/configure --prefix=/usr --host=${LJOS_TARGET} --build=${LJOS_HOST} --disable-profile --enable-add-ons --with-tls --enable-kernel=2.6.32 --with-__thread --with-binutils=${LJOS}/cross-tools/bin --with-headers=${LJOS}/usr/include --cache-file=config.cache

$ make -j4
$ echo $LJOS
$ make && make install_root=${LJOS}/ install

- **GCC (Final)**

As I mentioned previously, you'll now build the final GCC cross compiler that will link to the C library built and installed in the previous step. Create the gcc-build directory and change into it:

$ cd ..
$ ls
$ mkdir gcc-build
$ cd gcc-build/

$ AR=ar LDFLAGS="-Wl,-rpath,${LJOS}/cross-tools/lib" ../packages/gcc-12.2.0/configure --prefix=${LJOS}/cross-tools --build=${LJOS_HOST} --target=${LJOS_TARGET} --host=${LJOS_HOST} --with-sysroot=${LJOS} --disable-nls --enable-shared --enable-

languages=c,c++ --enable-c99 --enable-long-long --with-mpfr-include=$(pwd)/../packages/gcc-12.2.0/mpfr/src --with-mpfr-lib=$(pwd)/mpfr/src/.libs --disable-multilib --with-arch=${LJOS_CPU}

$ make -j4
$ make install -j4

$ cp -v ${LJOS}/cross-tools/${LJOS_TARGET}/lib64/libgcc_s.so.1 ${LJOS}/lib64
$ cd ..
$ cd lj-os/
$ cd cross-tools/bin/
$ ls
$ readelf riscv-unknown-linux-gnu-gcc
$ readelf -a riscv-unknown-linux-gnu-gcc | less
$ cd ../..

Now that you've built the cross compiler, you need to adjust and export the following variables:

$ export CC="${LJOS_TARGET}-gcc"
$ export CXX="${LJOS_TARGET}-g++"
$ export CPP="${LJOS_TARGET}-gcc -E"
$ export AR="${LJOS_TARGET}-ar"
$ export AS="${LJOS_TARGET}-as"
$ export LD="${LJOS_TARGET}-ld"
$ export RANLIB="${LJOS_TARGET}-ranlib"
$ export READELF="${LJOS_TARGET}-readelf"
$ export STRIP="${LJOS_TARGET}-strip"

- **Building the Target Image**

The hard part is now complete—you have the cross compiler. Now, let's focus on building the components that will be installed on the target image. This includes various libraries and utilities and, of course, the Linux kernel itself.

- **BusyBox**

BusyBox is one of my all-time favorite open-source projects. BusyBox combines a large collection of tiny versions of the most commonly used Linux utilities into a single distributed package. Those tools range from common binaries, text editors and command-line shells to filesystem and networking utilities, process management tools and many more.

Uncompress the tarball and change into its directory. Then load the default compilation configuration template:

$ make CROSS_COMPILE="${LJOS_TARGET}-" defconfig

The default configuration template will enable the compilation of a default defined set of utilities and libraries. You can enable/disable whatever you see fit by running menuconfig:

$ make CROSS_COMPILE="${LJOS_TARGET}-" menuconfig

Compile and install the package:

$ make CROSS_COMPILE="${LJOS_TARGET}-"
**$ make CROSS_COMPILE="${LJOS_TARGET}-" **
CONFIG_PREFIX="${LJOS}" install

Install the following Perl script, as you'll need it for the kernel build below:

$ cp -v examples/depmod.pl ${LJOS}/cross-tools/bin
$ chmod 755 ${LJOS}/cross-tools/bin/depmod.pl

- **The Linux Kernel**

Change into the kernel package directory and run the following to set the default x86-64 configuration template:

**$ make ARCH=${LJOS_ARCH} **
CROSS_COMPILE=${LJOS_TARGET}- x86_64_defconfig

I'm going to be running this target image in a VirtualBox virtual machine (defaulted in defconfig).For the sake of simplicity, these modules are configured to be compiled statically into the kernel image—that is, set to * instead of m.
Compile and install the kernel:

**$ make ARCH=${LJOS_ARCH} **
CROSS_COMPILE=${LJOS_TARGET}-

**$ make ARCH=${LJOS_ARCH} **
**CROSS_COMPILE=${LJOS_TARGET}- **
INSTALL_MOD_PATH=${LJOS} modules_install

You'll need to copy a few files into the /boot directory for GRUB:

$ cp -v arch/riscv/boot/Image ${LJOS}/boot/vmlinuz-4.16.3
$ cp -v System.map ${LJOS}/boot/System.map-4.16.3
$ cp -v .config ${LJOS}/boot/config-4.16.3

Then run the previously installed Perl script provided by the BusyBox package:

```
$ ${LJOS}/cross-tools/bin/depmod.pl \
-F ${LJOS}/boot/System.map-4.16.3 \
-b ${LJOS}/lib/modules/4.16.3
```

- **The Bootscripts**

The Cross Linux From Scratch (CLFS) project (a fork of the original LFS project) provides a wonderful set of bootscripts that I use here for simplicity's sake. Uncompress the package and change into its directory. Out of box, one of the package's makefiles contains a line that may not be compatible with your current working shell. Apply the following changes to the package's root Makefile to ensure that you don't experience any issues with package installation:

```
$ tar xJf clfs-embedded-bootscripts-1.0-pre5.tar.xz
```

```
@@ -19,7 +19,9 @@ dist:
        rm -rf "dist/clfs-embedded-bootscripts-$(VERSION)"

 create-dirs:
-       install -d -m ${DIRMODE}
 ↪${EXTDIR}/rc.d/{init.d,start,stop}
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/init.d
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/start
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/stop

 install-bootscripts: create-dirs
        install -m ${CONFMODE} clfs/rc.d/init.d/functions
        ↪${EXTDIR}/rc.d/init.d/
```

Then run the following commands to install and configure the target environment appropriately:

```
$ make DESTDIR=${LJOS}/ install-bootscripts
$ ln -sv ../rc.d/startup ${LJOS}/etc/init.d/rcS
```

- **Zlib**

Now you're at the very last package for this tutorial. Zlib isn't a requirement, but it serves as a great guide for other packages you may want to install for your environment. Feel free to skip this step if you'd rather format and configure the physical or virtual HDD. Uncompress the Zlib tarball and change into its directory. Then configure, build and install the package:

```
$ tar xJf zlib-1.2.12.tar.xz
$ sed -i 's/-O3/-Os/g' configure
$ ./configure --prefix=/usr --shared
$ make && make DESTDIR=${LJOS}/ install
```

Now, because some packages may look for Zlib libraries in the /lib directory instead of the /lib64 directory, apply the following changes:

```
$ mv -v ${LJOS}/usr/lib/libz.so.* ${LJOS}/lib
$ ln -svf ../../lib/libz.so.1 ${LJOS}/usr/lib/libz.so
$ ln -svf ../../lib/libz.so.1 ${LJOS}/usr/lib/libz.so.1
$ ln -svf ../lib/libz.so.1 ${LJOS}/lib64/libz.so.1
```

- **Installing the Target Image**

All of the cross compilation is complete. Now you have everything you need to install the entire cross-compiled operating system to either a physical or virtual drive, but before doing that, let's not tamper with the original target build directory by making a copy of it:

```
$ cp -rf ${LJOS}/ ${LJOS}-copy
```

Followed by the now unneeded statically compiled library files

```
$ FILES="$(ls ${LJOS}-copy/usr/lib64/*.a)"
$ for file in $FILES; do
 rm -f $file
 done
```

Now strip all debug symbols from the installed binaries. This will reduce overall file sizes and keep the target image's overall footprint to a minimum:

```
$ find ${LJOS}-copy/{,usr/}{bin,lib,sbin} -type f -exec sudo strip --strip-debug '{}' ';'
$ find ${LJOS}-copy/{,usr/}lib64 -type f -exec sudo strip --strip-debug '{}' ';'
```

Finally, change file ownerships and create the following nodes:

```
$ sudo chown -R root:root ${LJOS}-copy
$ sudo chgrp 13 ${LJOS}-copy/var/run/utmp ${LJOS}-copy/var/log/lastlog
$ sudo mknod -m 0666 ${LJOS}-copy/dev/null c 1 3
$ sudo mknod -m 0600 ${LJOS}-copy/dev/console c 5 1
$ sudo chmod 4755 ${LJOS}-copy/bin/busybox
```

Change into the target copy directory to create a tarball of the entire operating system image:

$ cd {LJOS}-copy/

$ sudo tar cfJ ../ljos-build-21April2018.tar.xz *

$ sudo virt-make-fs --size=+25G -t ext4 lj-os-copy/ riscv64gc_distro.img

Chapter 4

Implementation

1. Use of fedora Linux Platform for writing the code, building toolchain and exploring it QEMU.
2. Software Configuration:

QEMU: QEMU, a virtual machine emulator and hypervisor, can emulate RISC-V in software. This is useful when you don't have RISC-V hardware or don't have hardware that supports a particular extension. The software emulation in QEMU is called the Tiny Code Generator (TCG).

TCG works by translating basic blocks as they are encountered. The translated code is stored in a QEMU Translation Block (TB) structure and referenced through a hash table of (CPU state, physical address). TBs persist so that code doesn't need to be retranslated, but it can be invalidated by things such as writes happening to the same code page.

TCG defines a set of basic operations like integer adds, loads, stores, labels, branches, and so on. You can recognize these when you see tcg_gen_* called in QEMU code. For example, tcg_gen_qemu_ld_i64 would be called when translating a block of code and would generate a TCG instruction to do a 64-bit load and append it to the list of translated instructions.

However, anything complicated (such as CSRs or vector instructions) is translated into a call to a helper function. You will see helper functions defined in QEMU using the macro-HELPER (<name>). When translating, a call to the helper would be generated using gen_helper_<name>. Since most RISC-V extensions are complicated, they are almost always implemented as a set of helpers.

A file in the QEMU source target/riscv/insn32.decode describes how instruction

bit patterns are decoded. Extensions must list their new instructions here.

The file target/riscv/cpu.c contains two tables listing ISA extensions, their names, and versions. This is a very useful reference for finding out which extensions have been implemented in QEMU.

At the time of writing, QEMU supports these extensions, making it probably the most capable RISC-V platform:

- The base RV32I and RV64I ISAs
- The classic extensions: M A F D
- Compressed instructions: C, Zca, Zcb, Zcf, Zcd, Zce, Zcmp, Zcmt
- The embedded extension: E
- The hypervisor extension: H
- User and Supervisor modes (but note, not Machine mode): U S
- Dynamic languages: J
- Cache management (partial): Zicbom, Zicboz
- Conditional ops: Zicond
- Read and write CSRs: Zicsr
- FENCE.I instruction: Zifencei
- Pause hint: Zihintpause
- Wait on reservation set: Zawrs
- Additional scalar FP: Zfa
- Bfloat16 (partial): Zfbfmin
- Half-width FP: Zfh, Zfhmin
- FP using integer regs: Zfinx, Zdinx, Zhinx
- Bit manipulation: Zba, Zbb, Zbc, Zbs
- Crypto scalar: Zbkb, Zbkc, Zbkx, Zk*
- Vector (mostly complete): V, Zv*
- Advanced Interrupt Architecture: Smaia, Ssaia
- State enable: Smstateen
- Count overflow & filtering: Sscofpmf
- Time compare: Sstc
- Hardware update of PTE A/D bits: Svadu
- Fast TLB invalidation: Svinval

- NAPOT pages: Svnapot
- Page-based memory types: Svpbmt
- T-HEAD multiple custom extensions
- Ventana custom extensions for conditional ops

RISC-V VECTOR EXTENSION EXAMPLE:

1. C CODE

a. TEST code including vector instruction assembly

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define MAXELEM         1000

uint64_t get_time_us() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return (tv.tv_sec * 1000000) + tv.tv_usec;
}


extern void vvaddint32(size_t n, const int *x, const int *y, int *z);

int main()
{
        int x[MAXELEM] = {0};
        int y[MAXELEM] = {0};
        int z[MAXELEM] = {0};

        srand(time(NULL));

        for(int i=0; i < MAXELEM; i++)
        {
                x[i] = rand() % 1000;
                y[i] = rand() % 1000;
        }

        uint64_t begin = get_time_us();
        vvaddint32(MAXELEM, x, y, z);
        uint64_t elapsed = get_time_us() - begin;

        printf("\nTime taken for vector addition : %lu us\n", elapsed);



}
"main.c" 40L, 686B                                                    1,1           All
```

b. TEST program for normal C language for comparison

```c
#include <time.h>
#include <sys/time.h>

#define MAXELEM         1000

uint64_t get_time_us() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return (tv.tv_sec * 1000000) + tv.tv_usec;
}


extern void vvaddint32(size_t n, const int *x, const int *y, int *z);

int main()
{
        int x[MAXELEM] = {0};
        int y[MAXELEM] = {0};
        int z[MAXELEM] = {0};

        srand(time(NULL));

        for(int i=0; i < MAXELEM; i++)
        {
                x[i] = rand() % 1000;
                y[i] = rand() % 1000;
        }

        uint64_t begin = get_time_us();
        for(int i=0; i< MAXELEM; i++)
    {
        z[i] = x[i] + y[i];
    }
        uint64_t elapsed = get_time_us() - begin;

        printf("\nTime taken for non-vector addition : %lu us\n", elapsed);
}


"novect.c" 43L, 730B                                              40,1          Bot
```

2. ASSEMBLY CODE- with vector extension

```asm
    .text
    .balign 4
    .global vvaddint32
    # vector-vector add routine of 32-bit integers
    # void vvaddint32(size_t n, const int*x, const int*y, int*z)
    # { for (size_t i=0; i<n; i++) { z[i]=x[i]+y[i]; } }
    #
    # a0 = n, a1 = x, a2 = y, a3 = z
    # Non-vector instructions are indented
vvaddint32:
    vsetvli t0, a0, e32, ta, ma  # Set vector length based on 32-bit vectors
    vle32.v v0, (a1)         # Get first vector
      sub a0, a0, t0         # Decrement number done
      slli t0, t0, 2         # Multiply number done by 4 bytes
      add a1, a1, t0         # Bump pointer
    vle32.v v1, (a2)         # Get second vector
      add a2, a2, t0         # Bump pointer
    vadd.vv v2, v0, v1       # Sum vectors
    vse32.v v2, (a3)         # Store result
      add a3, a3, t0         # Bump pointer
      bnez a0, vvaddint32    # Loop back
      ret                    # Finished
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"vvadd.s" 22L, 908B                                               1,1           All
```

3. MAKEFILE

```
CC = riscv64-unknown-elf-gcc

bin:
	$(CC) -c vvadd.s
	$(CC) -c main.c
	$(CC) main.o vvadd.o -o main
	$(CC) novect.c -o novect

run:
	@-qemu-riscv64-static -cpu rv64,v=true main
	@-qemu-riscv64-static -cpu rv64,v=true novect


~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"Makefile" 13L, 227B                                                    1,1           All
```

VERIFICATION & ANALYSIS

1. OBJDUMP- shows vector instruction are present in 1(a) code

```
   1027c:       6906                    ld      s2,64(sp)
   1027e:       79e2                    ld      s3,56(sp)
   10280:       6125                    addi    sp,sp,96
   10282:       8082                    ret

0000000000010284 <vec_len_rvv>:
   10284:       010676d7                vsetvli a3,a2,e32,m1,tu,mu
   10288:       4205e007                vlseg3e32.v     v0,(a1)
   1028c:       920011d7                vfmul.vv        v3,v0,v0
   10290:       b21091d7                vfmacc.vv       v3,v1,v1
   10294:       b22111d7                vfmacc.vv       v3,v2,v2
   10298:       4e3011d7                vfsqrt.v        v3,v3
   1029c:       020561a7                vse32.v v3,(a0)
   102a0:       8e15                    sub     a2,a2,a3
   102a2:       20a6c533                sh2add  a0,a3,a0
   102a6:       20d6a6b3                sh1add  a3,a3,a3
   102aa:       20b6c5b3                sh2add  a1,a3,a1
   102ae:       fa79                    bnez    a2,10284 <vec_len_rvv>
   102b0:       8082                    ret
        ...

00000000000102b4 <__fp_lock>:
   102b4:       00000513                li      a0,0
   102b8:       00008067                ret

00000000000102bc <stdio_exit_handler>:
   102bc:       0001e637                lui     a2,0x1e
   102c0:       000155b7                lui     a1,0x15
   102c4:       0001e537                lui     a0,0x1e
   102c8:       5f060613                addi    a2,a2,1520 # 1e5f0 <__sglue>
   102cc:       6e058593                addi    a1,a1,1760 # 156e0 <_fclose_r>
   102d0:       60850513                addi    a0,a0,1544 # 1e608 <_impure_data>
   102d4:       3400006f                j       10614 <_fwalk_sglue>

00000000000102d8 <cleanup_stdio>:
   102d8:       00853583                ld      a1,8(a0)
   102dc:       ff010113                addi    sp,sp,-16
   102e0:       00813023                sd      s0,0(sp)
   102e4:       00113423                sd      ra,8(sp)
   102e8:       2b018793                addi    a5,gp,688 # 1f220 <__sf>
:
```

2. READELF -indicating flags for vector instruction file and no vector instruction file

```
muskan@cdackoji:~/self$ ls
main  main.c  main.o  Makefile  novect  novect.c  vvadd.o  vvadd.s
muskan@cdackoji:~/self$ riscv64-unknown-elf-readelf -A main
Attribute Section: riscv
File Attributes
  Tag_RISCV_stack_align: 16-bytes
  Tag_RISCV_arch: "rv64i2p1_m2p0_a2p1_f2p2_d2p2_v1p0_zicsr2p0_zifencei2p0_zmmul1p0_zve32f1p0_zve32x1p0_zve64d1p0_zve64f1p0_zve64x1p0_zvl128b1p0_zvl32b1p0_zvl64b1p0"
  Tag_RISCV_priv_spec: 1
  Tag_RISCV_priv_spec_minor: 11
muskan@cdackoji:~/self$ riscv64-unknown-elf-readelf -A novect
Attribute Section: riscv
File Attributes
  Tag_RISCV_stack_align: 16-bytes
  Tag_RISCV_arch: "rv64i2p1_m2p0_a2p1_f2p2_d2p2_v1p0_zicsr2p0_zifencei2p0_zmmul1p0_zve32f1p0_zve32x1p0_zve64d1p0_zve64f1p0_zve64x1p0_zvl128b1p0_zvl32b1p0_zvl64b1p0"
  Tag_RISCV_priv_spec: 1
  Tag_RISCV_priv_spec_minor: 11
muskan@cdackoji:~/self$
```

Therefore mentioned instances encompass an illustration of the RISC-V vector extension, juxtaposed with a conventional C-code executable operative on the QEMU static emulator tailored for RISC-V architecture, both of which have been seamlessly executed with efficacy. Moreover, a comprehensive demonstration of vector instructions is elucidated via Objdump, while the delineation of flags is elucidated through the utilization of readelf.

Unable to compile via the QEMU emulator due to the absence of vector instruction support, which is disabled within the QEMU environment.

COMPARISON WITH x_86 ARCHITECTURE EXAMPLE

1. C CODE

a. TEST program for normal C language for comparison

```c
#include <stdio.h>
#include <sys/time.h>
#include <stdint.h>

#define NUM_ENTRIES 100000

double get_time_us() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec * 1000000) + tv.tv_usec;
}

void fmaddsub(double *a, double *b, double *c, double *result, int n) {
    for (int i = 0; i < n; i++) {
        result[i] = a[i] * b[i] + ((i % 2 == 0) ? c[i] : -c[i]);
    }
}

int main() {
    double veca[NUM_ENTRIES];
    double vecb[NUM_ENTRIES];
    double vecc[NUM_ENTRIES];
    double result[NUM_ENTRIES];

    // Initialize vectors
    for (int i = 0; i < NUM_ENTRIES; i++) {
        veca[i] = 6.0;
        vecb[i] = 2.0;
        vecc[i] = 7.0;
    }

    // Perform the operation
    uint64_t begin = get_time_us();
    fmaddsub(veca, vecb, vecc, result, NUM_ENTRIES);
    uint64_t elapsed = get_time_us() - begin;

    // Display the result
    printf("Result for the first %d entries:\n", NUM_ENTRIES);
    for (int i = 0; i < NUM_ENTRIES; i++) {
        printf("%lf ", result[i]);
    }
    printf("\n\nTime taken for normal c operation: %lu us\n", elapsed);

    return 0;
}
```

36,0-1 All

b. TEST code including vector instruction assembly

```
#include <immintrin.h>
#include <stdio.h>
#include <sys/time.h>
#include <stdint.h>

#define n 100000
uint64_t get_time_us() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return (tv.tv_sec * 1000000) + tv.tv_usec;
}
int main() {
    // Initialize vectors with constant values
    __m256d veca = _mm256_set1_pd(6.0);
    __m256d vecb = _mm256_set1_pd(2.0);
    __m256d vecc = _mm256_set1_pd(7.0);

    // Initialize result array to store 1000 result values
    double result[n];

        uint64_t begin = get_time_us();
    // Perform the operation for 1000 entries
    for (int i = 0; i < n; i += 4) {
        __m256d result_vec = _mm256_fmaddsub_pd(veca, vecb, vecc);
        _mm256_storeu_pd(&result[i], result_vec);
    }

     uint64_t elapsed = get_time_us() - begin;

    // Display a few elements of the result array
    printf("Results for first few elements:\n");
    for (int i = 0; i < n; i++) {
        printf("%lf ", result[i]);
    }
    printf("\n");
     printf("\nTime taken for vector addition : %lu us\n", elapsed);

    return 0;
}
~
~
~
~
~
"x86_vector_ext.c" 40L, 1053B
```

VERIFICATION & ANALYSIS

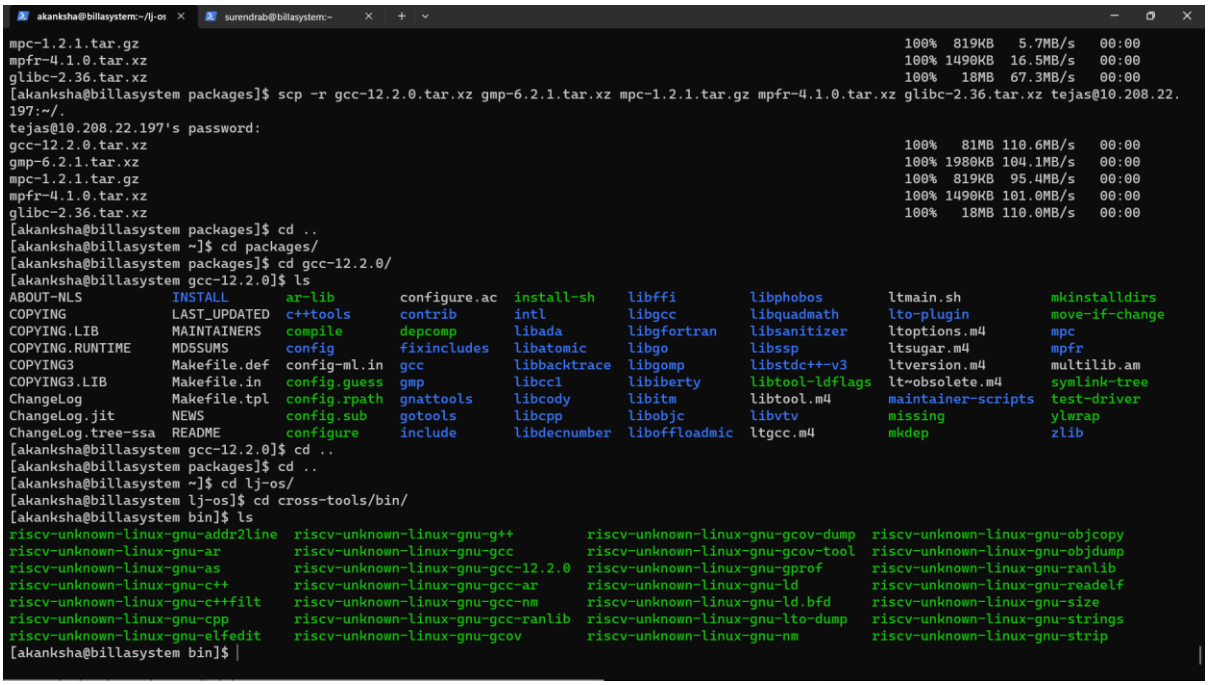
1. **OBJDUMP- shows vector instruction are present in 1(b) code**

```
0000000000401183 <main>:
  401183:       4c 8d 54 24 08          lea    0x8(%rsp),%r10
  401188:       48 83 e4 e0             and    $0xffffffffffffffe0,%rsp
  40118c:       41 ff 72 f8             push   -0x8(%r10)
  401190:       55                      push   %rbp
  401191:       48 89 e5                mov    %rsp,%rbp
  401194:       41 52                   push   %r10
  401196:       48 81 ec a8 20 00 00    sub    $0x20a8,%rsp
  40119d:       c5 fb 10 05 c3 0e 00    vmovsd 0xec3(%rip),%xmm0        # 402068 <__dso_handle+0x60>
  4011a4:       00
  4011a5:       c5 fb 11 85 18 ff ff    vmovsd %xmm0,-0xe8(%rbp)
  4011ac:       ff
  4011ad:       c4 e2 7d 19 85 18 ff    vbroadcastsd -0xe8(%rbp),%ymm0
  4011b4:       ff ff
  4011b6:       c5 fd 29 45 b0          vmovapd %ymm0,-0x50(%rbp)
  4011bb:       c5 fb 10 05 ad 0e 00    vmovsd 0xead(%rip),%xmm0        # 402070 <__dso_handle+0x68>
  4011c2:       00
  4011c3:       c5 fb 11 85 20 ff ff    vmovsd %xmm0,-0xe0(%rbp)
  4011ca:       ff
  4011cb:       c4 e2 7d 19 85 20 ff    vbroadcastsd -0xe0(%rbp),%ymm0
  4011d2:       ff ff
  4011d4:       c5 fd 29 45 90          vmovapd %ymm0,-0x70(%rbp)
  4011d9:       c5 fb 10 05 97 0e 00    vmovsd 0xe97(%rip),%xmm0        # 402078 <__dso_handle+0x70>
  4011e0:       00
  4011e1:       c5 fb 11 85 28 ff ff    vmovsd %xmm0,-0xd8(%rbp)
  4011e8:       ff
  4011e9:       c4 e2 7d 19 85 28 ff    vbroadcastsd -0xd8(%rbp),%ymm0
  4011f0:       ff ff
  4011f2:       c5 fd 29 85 70 ff ff    vmovapd %ymm0,-0x90(%rbp)
  4011f9:       ff
  4011fa:       b8 00 00 00 00          mov    $0x0,%eax
  4011ff:       e8 52 ff ff ff          call   401156 <get_time_us>
  401204:       48 89 85 68 ff ff ff    mov    %rax,-0x98(%rbp)
  40120b:       c7 45 ec 00 00 00 00    movl   $0x0,-0x14(%rbp)
  401212:       e9 8f 00 00 00          jmp    4012a6 <main+0x123>
  401217:       c5 fd 28 45 b0          vmovapd -0x50(%rbp),%ymm0
  40121c:       c5 fd 29 85 d0 fe ff    vmovapd %ymm0,-0x130(%rbp)
  401223:       ff
  401224:       c5 fd 28 45 90          vmovapd -0x70(%rbp),%ymm0
  401229:       c5 fd 29 85 b0 fe ff    vmovapd %ymm0,-0x150(%rbp)
  401230:       ff
  401231:       c5 fd 28 85 70 ff ff    vmovapd -0x90(%rbp),%ymm0
  401238:       ff
  401239:       c5 fd 29 85 90 fe ff    vmovapd %ymm0,-0x170(%rbp)
  401240:       ff
:
```

The preceding demonstration juxtaposes conventional C code with its counterpart augmented with vector extensions, elucidating the inherent superiority of vector instructions in facilitating parallel processing. This superiority manifests itself upon successful compilation, particularly when vector extensions are enabled on the hardware platform. Furthermore, the utilization of objdump serves to visually depict the incorporation and execution of vector instructions within the codebase.

Chapter 5

Results



- **We have successfully build the cross-compilation toolchain for risc-v Vector extension (riscv-unknown-linux-gnu-gcc).**

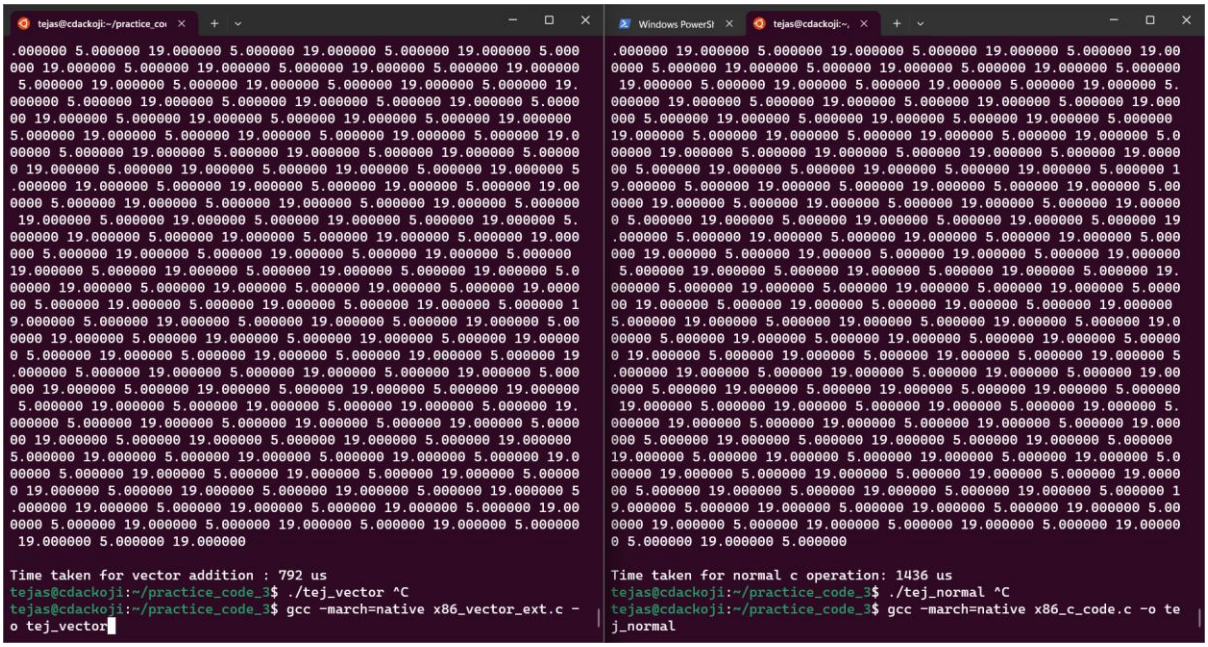
```
muskan@cdackoji:~$ ls
lj-os  muskan  packages  packages1  qemu  riscv  self
muskan@cdackoji:~$ cd muskan/opt/bin/
muskan@cdackoji:~/muskan/opt/bin$ ls
riscv64-unknown-elf-addr2line      riscv64-unknown-elf-lto-dump          riscv64-unknown-linux-gnu-gcc-nm
riscv64-unknown-elf-ar             riscv64-unknown-elf-nm                riscv64-unknown-linux-gnu-gcc-ranlib
riscv64-unknown-elf-as             riscv64-unknown-elf-objcopy           riscv64-unknown-linux-gnu-gcov
riscv64-unknown-elf-c++            riscv64-unknown-elf-objdump           riscv64-unknown-linux-gnu-gcov-dump
riscv64-unknown-elf-c++filt        riscv64-unknown-elf-ranlib            riscv64-unknown-linux-gnu-gcov-tool
riscv64-unknown-elf-cpp            riscv64-unknown-elf-readelf           riscv64-unknown-linux-gnu-gdb
riscv64-unknown-elf-elfedit        riscv64-unknown-elf-run               riscv64-unknown-linux-gnu-gdb-add-index
riscv64-unknown-elf-g++            riscv64-unknown-elf-size              riscv64-unknown-linux-gnu-gfortran
riscv64-unknown-elf-gcc            riscv64-unknown-elf-strings           riscv64-unknown-linux-gnu-gprof
riscv64-unknown-elf-gcc-13.2.0     riscv64-unknown-elf-strip             riscv64-unknown-linux-gnu-ld
riscv64-unknown-elf-gcc-ar         riscv64-unknown-linux-gnu-addr2line   riscv64-unknown-linux-gnu-ld.bfd
riscv64-unknown-elf-gcc-nm         riscv64-unknown-linux-gnu-ar          riscv64-unknown-linux-gnu-lto-dump
riscv64-unknown-elf-gcc-ranlib     riscv64-unknown-linux-gnu-as          riscv64-unknown-linux-gnu-nm
riscv64-unknown-elf-gcov           riscv64-unknown-linux-gnu-c++         riscv64-unknown-linux-gnu-objcopy
riscv64-unknown-elf-gcov-dump      riscv64-unknown-linux-gnu-c++filt     riscv64-unknown-linux-gnu-objdump
riscv64-unknown-elf-gcov-tool      riscv64-unknown-linux-gnu-cpp         riscv64-unknown-linux-gnu-ranlib
riscv64-unknown-elf-gdb            riscv64-unknown-linux-gnu-elfedit     riscv64-unknown-linux-gnu-readelf
riscv64-unknown-elf-gdb-add-index  riscv64-unknown-linux-gnu-g++         riscv64-unknown-linux-gnu-run
riscv64-unknown-elf-gprof          riscv64-unknown-linux-gnu-gcc         riscv64-unknown-linux-gnu-size
riscv64-unknown-elf-ld             riscv64-unknown-linux-gnu-gcc-13.2.0  riscv64-unknown-linux-gnu-strings
riscv64-unknown-elf-ld.bfd         riscv64-unknown-linux-gnu-gcc-ar      riscv64-unknown-linux-gnu-strip
muskan@cdackoji:~/muskan/opt/bin$
```

```
muskan@cdackoji:~/self
muskan@cdackoji:~/self$ ls
main  main.c  main.o  Makefile  novect  novect.c  vvadd.o  vvadd.s
muskan@cdackoji:~/self$ make
riscv64-unknown-elf-gcc -c vvadd.s
riscv64-unknown-elf-gcc -c main.c
riscv64-unknown-elf-gcc main.o vvadd.o -o main
riscv64-unknown-elf-gcc novect.c -o novect
muskan@cdackoji:~/self$ make run

Time taken for vector addition : 218 us

Time taken for non-vector addition : 106 us
muskan@cdackoji:~/self$
```

- **When we try to run our vector addition program on QEMU static (Virtualize vCPU) because of this is emulator that's why the time taken by the vector addition is showing more and time taken by the non-vector addition program is less. Even for android emulator they turn off the vector extension.**
- **But if we can try this on real hardware it will show totally opposite, the time taken by the vector addition program is much less as compare to non-vector addition program. But currently World Wide only one Microcontroller is there and that is**

also very slow.

- **The RISC-V Vector Extension (RVV) Version 1.0 was ratified by RISC-V International in 2021. Since this public debut, there has been growing excitement about vector processing across a wide spectrum of applications since vectors promise to solve multiple current industry design and development challenges. Robust RISC-V ecosystem embraces the advantages of RISC-V vector solutions.**
- **The program is not working on our Fedora's QEMU because at the time building the kernel the only general purpose and compressed extension are used because of that we get illegal instruction there, it is not possible to run over there our vector extension program.**



- **We try on Intel x86 SIMD AVX and we get the above results, for this the vector extension take much less time than non-vector extension.**
  - **As per above result the time taken for vector addition is 792 micro-seconds.**
  - **The time taken for normal C operation is 1436 micro-seconds.**

Chapter 6

Conclusion

6.1 Conclusion

- In this RISC-V and HPC project, we have successfully built a RISC-V Vector extension cross compilation tool chain.
- Vector can change whole landscape when this vector extension arrive with RISC-V ISA hardware.
- RISC-V flexibility and open-source nature empower HPC developers to customize processor architectures, optimizing them for specific computational tasks. This adaptability facilitates the creation of specialized processors that can enhance performance for scientific simulations, data analytics, and other HPC applications.
- Custom RISC-V processors, designed with a focus on HPC workloads, have the potential to deliver performance gains compared to traditional architectures. This can lead to advancements in scientific research, simulations, and data-intensive tasks critical for pushing the boundaries of HPC capabilities.

6.2 Future Enhancement –

- **Integration with AI Accelerators:** As machine learning and AI become increasingly important, future enhancements may involve better integration between the RISC-V vector extension and specialized AI accelerators to address the requirements of AI workloads more efficiently.
- **Energy Efficiency Improvements:** Future enhancements might focus on making the vector extension more energy-efficient, aligning with the growing emphasis on energy efficiency in modern computing.
- **Standardization and Ecosystem Support:** Ongoing efforts to standardize and improve the software ecosystem around RISC-V, including compilers, libraries, and tools, will play a crucial role in the success and adoption of the vector extension.
- **Community Collaboration:** The collaborative nature of the RISC-V community fosters knowledge exchange and innovation. Continued collaboration between researchers, engineers, and industry stakeholders is essential for further advancing RISC-V in the HPC domain.

Chapter 7

References

[1] **RISC-V Vector (RVV) Specification:**

- The specification for the RISC-V Vector (RVV) extension provides details about the vector instructions and their usage.
- RISC-V Foundation's documentation section or the official GitHub repository for the latest specifications.
- RISC-V GitHub Repository
  https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

[2] **RISC-V Specifications:**

- The official RISC-V specifications can be found on the RISC-V Foundation's website. The specifications include the base ISA as well as various extensions, including the vector extension.
- RISC-V Specifications

[3] **DIY:** Build a custom minimal linux Distribution from source.

https://www.linuxjournal.com/content/diy-build-custom-minimal-linux-distribution-source

[4] **Simple RISC-V Vector example:** Test Code for Vector Extension.

https://github.com/brucehoult/rvv_example

[5] **Simple x_86 Vector example:** Test Code for Vector Extension.

https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX

[6] **RISC-V GNU Compiler Toolchain:** Toolchain for cross-compilation.

https://github.com/riscv-collab/riscv-gnu-toolchain

[7] **ExCALIBUR H&ES RISC-V testbed:** A RISC-V test environment for scientific and data-science codes.

https://riscv.epcc.ed.ac.uk/community/isc23-workshop/