A Simulated Environment For RISC-V Using Spike And RV8 Simulators

Dharma Teja Bandaru and Sreeja Matturu

New Mexico State University, Las Cruces, New Mexico

May 9, 2019

1 Introduction

RISC-V is an open source hardware instruction set which is abbreviated as Reduced Instruction Set Computer. This project came into existence in 2010, and is implemented at University of California at Berkeley and is now taken care by RISC-V foundation. The recent platforms of open hardware mainly reduce the cost which is required to design the device architectures. For next generation embedded systems and many IoT devices, RISC-V instruction set became so popular. The instruction set of RISC-V is an open hardware instruction set which is initially designed to develop an extensible and open instruction set that can be used for research and commercial purposes. The RISC-V architecture can be implemented using a multitude of different type of microarchitectures. The RISC-V is defined as a small reduced instruction set computer-based architecture that can be divided into different modules which supports floating-point computation, vector and atomic operations. Each module mainly focuses on upcoming targets like IoT embedded devices and some of the cloud servers. [7]

RISC-V is also defined as frozen ISA which means that the base instructions as well as the optional extensions which are approves are frozen. The frozen ISA deploys a strong foundation to preserve the investments that are made in software. Along with that, the hardware designers are also having the flexibility during processor design and the ISA takes a major part in implementing this. Through this, the input which is provided by the software to hardware designers about developing a RISC-V core can be more software centric. Since RISC-V ISA consists of high stability, the software development is using the ISA and the code which is designed for RISC-V of a software will be able to run on all cores of RISC-V forever. The RISC-V ISA is equivalent to everyone who carries the microarchitecture license. The ISA can optimize performance, security and some designs of lower power... etc. by having the full compatibility with the other designs. RISC-V supports the custom instructions for the designs which need specific functions. In addition to that, RISC-V also provide some additional benefits. If the engineers are developing a soft RISC-V core in FPGA, then the RTL source code is available by the making the process easier and enables deep inspection. This also helps in creating the significant flexibility to port a RISC-V design. Some of the key features of RISC-V are:

- It has the ability to deliver the new of hardware and software architecture in open source mode.
- Broad range of operating systems, software vendors and too developers are supported by RISC-V ISA.
- RISC-V does not depend on only a single supplier, it supports for multiple suppliers and massive potential for future growth.
- RISC-V ISA allows the user extensibility without breaking the previous extensions or any kind of software fragmentation.

1.1 Spike Simulator

Spike is one of the major functional simulators that are used for RISC-V instruction set and its principal public API is RISC-V ISA. It is an instruction set simulator, which excludes all internal delays like cache misses, some of the memory transactions and IO accesses. Initially, Spike simulator is implemented to support high performance, rapid evaluation of RISC-V and some microarchitecture changes. [5]One of the major concept is that, it has the ability to support for user defined extensions as well as to simulate the extensions of a full suite standard RISC-V for 32 bit and 64 bit RISC-V cores. One of the major drawback in Spike is that, it

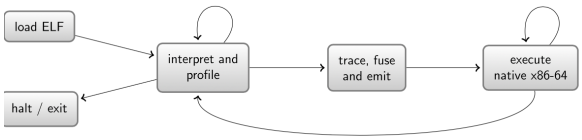

Figure 1: *RV8 Simulator For RISC-V [4]*

does not have a fully cache model, rather it considers cache as a tracer. In RISC-V, the specifications which are given, determines the functionality of each instruction but it does not determine the number of cycles taken by an each instruction. In that manner, Spike simulator does not have the ability to count the number of cycles. Spike consists of a simple cache and memory model and no one has made any attempt to design the spike memory subsystem in a cycle-based manner. The Spike has the ability to insert the memory requests and helps the researchers to scale the experiments in an easy way. [6]

1.2 RV8 Simulator

RV8 is another major functional simulator that is used in RISC-V instruction set. It mainly comprises the best performance for x86-64binary translator, a user mode simulator, a full system translator and an ELF binary analysis tool. This simulator suits mainly consists of different libraries and the command line tools for different systems that helps in implementing the RISC-V and an x86-64 binary translator.

The binary translator engine for RV8 mainly works by interpreting the code while profiling for hot paths. This mainly supports for hybrid binary translation and interpretation so that it can be able to handle the instructions that does not have any native translations. The RV8 binary translator supports for optimization's like:

- Compiling the hot code paths and hybrid interpretation.
- Increasing the translation with the help of dynamic trace linking.
- Making the inline caching function calls in the hot code paths.
- The single cache layer jumps target cache for calls and returns in an indirect manner.
- Macro-op fusion for common RISC-V instruction sequences.

The RV8 full system emulator has a homogenous debugger that allows to handle the breakpoints, single step and disassemble the instructions in the way they are executed. This system has some important characteristics like, it has an address space which is protected. The ISA metadata generated ab extensible interpreter. It consists of simple debugging command line interface.

The RV8 user mode simulator is defined as a single address space implementation which represents the system call to the underlying operating system. This kind of simulator runs RISC-V Linux libraries on non-linux OS with the help of system call emulation. This is almost similar to the full system emulator. But the only difference is that user mode has an address space which is a shared one while full system emulator has protected address space. [8] The optimization of the simulator can be done in different ways. One is register allocation which mainly deals with the solving of the size of the register set by using static register allocation. Due to this, some amount of performance is lost, but there exists multiple number of available registers and low frequent stack spills. Another technique is using the translator temporaries. It uses several host registers to point at internal structures. The factors that are required to quantify the performance differences between the original x86-64 and translated RISC-V code, the bench marks should measure the sign extension overhead, indirect call or return overhead, improvement in Macro-op fusion, register allocation and work per instruction. [3]

2 Methodology

Our experiment with RISC-V instruction set is made with two simulators. One is Spike and the other one is RV8 simulator. While coming to the execution part, we installed the Spike in MacOS and RV8 on Linux system, and executed the required benchmarks. In order to install and execute the benchmarks using these simulators, there are some of the steps that are need to be followed.

2.1 Spike Installation

In order to install Spike, it requires some base tools like device-tree-compiler and OpenBSD. The device-tree-compiler is mainly used to compile the given source code into binary form. The files which are included in this can be divided into multiple files so that it can get compatible with the system. Hence the primary step that is needed to install Spike is to install the device-tree-compiler. The following command helped us in installing this compiler.
[1]

$ apt-get install device-tree-compiler

Later we created a folder named build and created a path by using the command,

$ mkdir build $ cd build

Now, it is requires to configure the prefix and set it as RISCV and it done by using the command,

$ ../configure –prefix=$RISCV

By doing this, in the build folder it creates all the required files along with a makefile. Hence, by executing the makefile, the device-tree-compiler will be installed successfully by following the command,

$ [sudo] make install

Another important aspect in installing the Spike is to install bash, game, etc and clang on a OpenBSD. An OpenBSD is defined as a free source which mainly emphasizes the portability, standard, correctness and security. The following build steps helps in installing the prerequisites for Spike on OpenBSD.

The bash, game and etc can be installed as a package within the terminal by using the pkg_add command which helps in installing the software packages,

$ pkg_add bash gmake dtc

Since the packages are already downloaded, we execute the bash software by using exec command. The exec is mainly used for system calls.

$ exec bash

Now, we export the c++ file by using the command export,

$ export CC=cc; export CXX=c++.

We create the same build file again and create a path in the terminal like the way the device-tree-compiler did,

$ mkdir build $ cd build

Now, we configure the build folder by having the prefix as RISCV,

$ ../configure –prefix=$RISCV

This creates the list of program files and these are executed using the gmake command,

$ gmake

Finally, the docs executes the command as per the instructions given by the user and install the packages successfully.

$ [doas] make install

For compiling and running a sample program using Spike, it requires some prerequisite tool sets like riscv-gnu-toolchain and riskv-pk.

The GNU tool chain of RISC-V is the cross-compiler and it supports two different modules, one is a generic ELF/Newlib tool chain and the other one is Linux-ELF/glibc tool chain. Hence, in order to download the gnu tool chain from internet we do by using the command called git clone, [9]

$ git clone –recursive https://github.com/riscv/riscv-gnu-toolchain

Now, in order to install and run the programs using gnu toolchain, some of the prerequisites must be installed. They are like,

$ brew install gawk gnu-sed gmp mpfr limbic isl zlib expat

This requires a case-insensitive file system and the simple format to build the glib is to create and add a new disk image with the case sensitive format. The overall process gets started by downloading the upstream sources and then they will patch, build and finally they install the toolchain. By default, the system requires 8GB of disk space, so that the installation will be completed successfully.

Finally, in order to build the cross-compiler which supports for 32 and 64-bit, the commands that are required are:

$ ./configure —prefix=/opt/riscv —enable-multilib

$ make

Finally, the riscv-gnu-toolchain is installed in the system.

The riscv-pk works as a proxy kernel for RISC-V and it is essential to install gnu tool chain and the commands that are required to build the proxy kernel are:

$ Mkdir build

$ Cd build

$ ../configure —prefix=$RISCV —host=riscv64-unknown-elf

$ Make

$ Make install

The building steps from OpenBSD are:

$ pkg_add riscv-elf-binutils riscv-elf-gcc riscv-elf-newlib

Finally the proxy kernel for the RISC-V is also installed.

In order to compile and run the programs, we took two different programs. One is to multiply two matrices and the other one is to find whether the given number is a prime number or not. The commands that we took to compile and run both the programs are:

$ riscv64-unknown-elf-gcc -o multiply multiply.c

$ spike pk multiply

$ riscv64-unknown-elf-gcc -o prime prime.c

$ spike pk prime

We also made an interactive debug mode by launching the spike with -d command:

$ spike -d pk multiply/prime

In order to see the contents of the floating point and memory register, we use the commands:

: reg 0 a0

: fregs 0 ft0

: mem 2020

:mem 0 2020

In this way, we installed the spike and executed the programs by using the riscv commands.

2.2 RV8 Installation

The prerequisites that are required to install the RV8 simulator in RISC-V are musl-riscv-toolchain and riscv-qemu. The commands and the process of installing RV8 simulator are: [4]

$ git clone https://github.com/rv8-io/rv8.git

$ cd rv8

$ git submodule update —init

$make -j4 && sudo make install

Qemu is defined as an open source machine emulator and a virtualizer which can run OSs and programs that are made for one machine. It delivers the good performance by using dynamic translation. Hence the steps that are required to install the riscv-qemu are defined as follows:[2]

$ git clone https://github.com/riscv/riscv-qemu.git

$ cd riscv-qemu

$ git submodule update —init

$ ./configure —target-list=riscv64-softmmu,riscv32-softmmu

$ make -j4 && sudo make install

Through this it creates a makefile by including all the required files and programs and benchmarks.

$ cd rv8-bench

$ make

Finally, it executes all the files along with the benchmarks and successfully installs the RV8 simulator in the system.

3 Experimental Results

The List of screen shots that are available after compiling and running the programs on both Spike and RV8 are described below.

3.1 Spike Results

```
dharma836e:AC-2 dharmatejab$ riscv64-unknown-elf-gcc -o multiply  Multiply.c
dharma836e:AC-2 dharmatejab$ spike pk multiply
bbl loader
Result matrix is
10 10 10 10
20 20 20 20
30 30 30 30
40 40 40 40
dharma836e:AC-2 dharmatejab$ spike -d pk multiply
: reg 0 a0
0x0000000000000000
:
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
:
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
:
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
:
core   0: 0x0000000000001010 (0x00028067) jr      t0
:
core   0: 0x0000000080000000 (0x1f80006f) j       pc + 0x1f8
:
core   0: 0x00000000800001f8 (0x00000093) li      ra, 0
:
core   0: 0x00000000800001fc (0x00000113) li      sp, 0
:
core   0: 0x0000000080000200 (0x00000193) li      gp, 0
:
core   0: 0x0000000080000204 (0x00000213) li      tp, 0
```

Figure 2: *Spike Multiplication of Two Matrices*

```
: fregs 0 ft0
nan
:
core   0: 0x0000000080000208 (0x00000293) li      t0, 0
:
core   0: 0x000000008000020c (0x00000313) li      t1, 0
:
core   0: 0x0000000080000210 (0x00000393) li      t2, 0
:
core   0: 0x0000000080000214 (0x00000413) li      s0, 0
:
core   0: 0x0000000080000218 (0x00000493) li      s1, 0
:
core   0: 0x000000008000021c (0x00000613) li      a2, 0
:
core   0: 0x0000000080000220 (0x00000693) li      a3, 0
```

Figure 3: *Spike Multiplication Floating Point Register*

```
: mem 2020
:
core   0: 0x0000000080000224 (0x00000713) li      a4, 0
:
core   0: 0x0000000080000228 (0x00000793) li      a5, 0
:
core   0: 0x000000008000022c (0x00000813) li      a6, 0
:
core   0: 0x0000000080000230 (0x00000893) li      a7, 0
:
core   0: 0x0000000080000234 (0x00000913) li      s2, 0
: mem 20
:
core   0: 0x0000000080000238 (0x00000993) li      s3, 0
:
core   0: 0x000000008000023c (0x00000a13) li      s4, 0
```

Figure 4: *Spike Multiplication Memory Register*

```
: q
dharma836e:AC-2 dharmatejab$ spike -d pk prime
: reg x1
:
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
:
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
:
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
:
core   0: 0x0000000000001010 (0x00028067) jr      t0
:
core   0: 0x0000000080000000 (0x1f80006f) j       pc + 0x1f8
:
core   0: 0x00000000800001f8 (0x00000093) li      ra, 0
:
core   0: 0x00000000800001fc (0x00000113) li      sp, 0
:
core   0: 0x0000000080000200 (0x00000193) li      gp, 0
:
core   0: 0x0000000080000204 (0x00000213) li      tp, 0
:
core   0: 0x0000000080000208 (0x00000293) li      t0, 0
:
core   0: 0x000000008000020c (0x00000313) li      t1, 0
:
core   0: 0x0000000080000210 (0x00000393) li      t2, 0
```

Figure 5: *Spike Result For Prime Number*

3.2 RV8 Results

```
bin    CMakeLists.txt  LICENSE   meta       scripts  third_party
build  doc             Makefile  README.md  src
dharmateja@Linux1:~/riscv-gnu-toolchain/rv8$ rv-bin dump -c -a build/riscv64-unknown-elf/bin/
prime

---[ ELF Header ]----------------------------------------------------------------------------
------------------------------

File                    build/riscv64-unknown-elf/bin/prime
Class                   ELF64
Data                    little-endian
Type                    shared object
Machine                 x86-64
Interp                  /lib64/ld-linux-x86-64.so.2

---[ Section Headers ]-----------------------------------------------------------------------
------------------------------

Shdr Name                Type         Flags              Addr               Offset     Siz
e       Align  Ents Link Info
[ 0]                     NULL                            0x0                0x0        0x0
        0x0       0    0    0
[ 1] .interp             PROGBITS     +ALLOC             0x2a8              0x2a8      0x1
c       0x1       0    0    0
[ 2] .note.ABI-tag       NOTE         +ALLOC             0x2c4              0x2c4      0x2
0       0x4       0    0    0
[ 3] .note.gnu.build-id  NOTE         +ALLOC             0x2e4              0x2e4      0x2
4       0x4       0    0    0
[ 4] .gnu.hash           UNKNOWN      +ALLOC             0x308              0x308      0x2
4       0x8       0    5    0
[ 5] .dynsym             DYNSYM       +ALLOC             0x330              0x330      0xd
8       0x8      24    6    1
[ 6] .dynstr             STRTAB       +ALLOC             0x408              0x408      0xb
8       0x1       0    0    0
[ 7] .gnu.version        GNU_VERSYM   +ALLOC             0x4c0              0x4c0      0x1
2       0x2       2    5    0
[ 8] .gnu.version_r      GNU_VERNEED  +ALLOC             0x4d8              0x4d8      0x4
0       0x8       0    6    1
[ 9] .rela.dyn           RELA         +ALLOC             0x518              0x518      0xc
0       0x8      24    5    0
[10] .rela.plt           RELA         +ALLOC+INFO_LINK   0x5d8              0x5d8      0x4
8       0x8      24    5   22
[11] .init               PROGBITS     +ALLOC+EXEC        0x1000             0x1000     0x1
7       0x4       0    0    0
```

Figure 6: *RV8 Output For Prime Program*

```
---[ Symbol Table ]--------------------------------------------------------------------------
------------------------------

Symbol Value              Size  Type     Bind     Vis       Index Name

[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x2a8                 0 SECTION  LOCAL    DEFAULT       1
[   2] 0x2c4                 0 SECTION  LOCAL    DEFAULT       2
[   3] 0x2e4                 0 SECTION  LOCAL    DEFAULT       3
[   4] 0x308                 0 SECTION  LOCAL    DEFAULT       4
[   5] 0x330                 0 SECTION  LOCAL    DEFAULT       5
[   6] 0x408                 0 SECTION  LOCAL    DEFAULT       6
[   7] 0x4c0                 0 SECTION  LOCAL    DEFAULT       7
[   8] 0x4d8                 0 SECTION  LOCAL    DEFAULT       8
[   9] 0x518                 0 SECTION  LOCAL    DEFAULT       9
[  10] 0x5d8                 0 SECTION  LOCAL    DEFAULT      10
[  11] 0x1000                0 SECTION  LOCAL    DEFAULT      11
[  12] 0x1020                0 SECTION  LOCAL    DEFAULT      12
[  13] 0x1060                0 SECTION  LOCAL    DEFAULT      13
[  14] 0x1070                0 SECTION  LOCAL    DEFAULT      14
[  15] 0x12a4                0 SECTION  LOCAL    DEFAULT      15
[  16] 0x2000                0 SECTION  LOCAL    DEFAULT      16
[  17] 0x2088                0 SECTION  LOCAL    DEFAULT      17
[  18] 0x20c8                0 SECTION  LOCAL    DEFAULT      18
[  19] 0x3da8                0 SECTION  LOCAL    DEFAULT      19
[  20] 0x3db0                0 SECTION  LOCAL    DEFAULT      20
[  21] 0x3db8                0 SECTION  LOCAL    DEFAULT      21
[  22] 0x3fa8                0 SECTION  LOCAL    DEFAULT      22
[  23] 0x4000                0 SECTION  LOCAL    DEFAULT      23
[  24] 0x4010                0 SECTION  LOCAL    DEFAULT      24
[  25] 0x0                   0 SECTION  LOCAL    DEFAULT      25
[  26] 0x0                   0 FILE     LOCAL    DEFAULT     ABS crtstuff.c
[  27] 0x10a0                0 FUNC     LOCAL    DEFAULT      14 deregister_tm_clones
[  28] 0x10d0                0 FUNC     LOCAL    DEFAULT      14 register_tm_clones
[  29] 0x1110                0 FUNC     LOCAL    DEFAULT      14 __do_global_dtors_aux
[  30] 0x4010                1 OBJECT   LOCAL    DEFAULT      24 completed.7930
[  31] 0x3db0                0 OBJECT   LOCAL    DEFAULT      20 __do_global_dtors_aux_fini_
array_entry
[  32] 0x1150                0 FUNC     LOCAL    DEFAULT      14 frame_dummy
[  33] 0x3da8                0 OBJECT   LOCAL    DEFAULT      19 __frame_dummy_init_array_en
try
[  34] 0x0                   0 FILE     LOCAL    DEFAULT     ABS prime.c
```

Figure 7: *Symbol Table for Prime Program*

In RV8 compilation process, we took a sample Hello World program and when the program is executed, it creates different header files. Primarily it creates an ELF header files. This file will help to choose either 32-bit or 64-bit addresses by using its fields like file, class, data, type, machine and an interpreter. Next It produces the section headers, which consists of different sections like shdr name, type, flags, address, offset, size and aligned ents link information. Later it creates a symbol table which is defined as a data structure that is used by the language translator like compiler or interpreter. It creates the table with the contents like the value of the symbol, size, type, bind, vis and the type of index that is used.

After the symbol table is created, it creates the program header files which is defined as an array of structures that gives the information to the system about the requirements for the execution. It creates with the contents like phdr, memsize type, flags, offset align, virtual address, physical address and the size of the file. The above headers are all meant to be about the assembling the code. While coming to the disassembling the code, it processes with different functions like init, plt...etc. Finally, the makemap of the program is created by defining all the registers and executing all the required programs successfully.

```
---[ Program Headers ]-----------------------------------------------------------------------
------------------------------

Phdr Type    Flags     Offset             VirtAddr           PhysAddr           FileSize
       MemSize              Align
[ 0] PHDR    +R        0x40               0x40               0x40               0x268
       0x268                8
[ 1] INTERP  +R        0x2a8              0x2a8              0x2a8              0x1c
       0x1c                 1
[ 2] LOAD    +R        0x0                0x0                0x0                0x620
       0x620             4096
[ 3] LOAD    +R+X      0x1000             0x1000             0x1000             0x2ad
       0x2ad             4096
[ 4] LOAD    +R        0x2000             0x2000             0x2000             0x1d0
       0x1d0             4096
[ 5] LOAD    +R+W      0x2da8             0x3da8             0x3da8             0x268
       0x270             4096
[ 6] DYNAMIC +R+W      0x2db8             0x3db8             0x3db8             0x1f0
       0x1f0                8
[ 7] NOTE    +R        0x2c4              0x2c4              0x2c4              0x44
       0x44                 4
[ 8] GNU_EH_FRAME +R       0x2088             0x2088             0x2088             0x3c
       0x3c                 4
[ 9] GNU_STACK +R+W      0x0                0x0                0x0                0x0
       0x0                 16
[10] GNU_RELRO +R        0x2da8             0x3da8             0x3da8             0x258
       0x258                1
```

Figure 8: *Program Headers of Prime Program*

```
 META src/app/test-cc.cc
dharmateja@Linux1:~/riscv-gnu-toolchain/rv8$ make map
 META src/asm/strings.cc
 META src/asm/strings.h
 META src/asm/meta.cc
 META src/asm/meta.h
 META src/asm/jit.cc
 META src/asm/operands.h
 META src/asm/switch.h
 META src/asm/constraints.h
 META src/asm/jit.h
 META src/test/test-fpu-gen.h
 META src/test/test-fpu-gen.c
 META src/emu/interp.h
 META src/app/test-cc.cc
//  3         2         1         0
// 10987654321098765432109876543210
// SSSSSSSSSSSSSSSSSSSSRRRRR0110111 lui rd,imm                  # rv32i
// 00000000000000000000RRRRR0010111 auipc rd,offset             # rv32i
// 00000000000000000000RRRRR1101111 jal rd,offset               # rv32i
// 000000000000RRRRR000RRRRR1100111 jalr rd,rs1,offset          # rv32i
// 0000000RRRRRRRRRR000OOOOO1100011 beq rs1,rs2,offset          # rv32i
// 0000000RRRRRRRRRR001OOOOO1100011 bne rs1,rs2,offset          # rv32i
// 0000000RRRRRRRRRR100OOOOO1100011 blt rs1,rs2,offset          # rv32i
// 0000000RRRRRRRRRR101OOOOO1100011 bge rs1,rs2,offset          # rv32i
// 0000000RRRRRRRRRR110OOOOO1100011 bltu rs1,rs2,offset         # rv32i
// 0000000RRRRRRRRRR111OOOOO1100011 bgeu rs1,rs2,offset         # rv32i
// 000000000000RRRRR000RRRRR0000011 lb rd,offset(rs1)           # rv32i
// 000000000000RRRRR001RRRRR0000011 lh rd,offset(rs1)           # rv32i
// 000000000000RRRRR010RRRRR0000011 lw rd,offset(rs1)           # rv32i
// 000000000000RRRRR100RRRRR0000011 lbu rd,offset(rs1)          # rv32i
// 000000000000RRRRR101RRRRR0000011 lhu rd,offset(rs1)          # rv32i
// 0000000RRRRRRRRRR000OOOOO0100011 sb rs2,offset(rs1)          # rv32i
// 0000000RRRRRRRRRR001OOOOO0100011 sh rs2,offset(rs1)          # rv32i
// 0000000RRRRRRRRRR010OOOOO0100011 sw rs2,offset(rs1)          # rv32i
// SSSSSSSSSSSSRRRRR000RRRRR0010011 addi rd,rs1,imm             # rv32i
// SSSSSSSSSSSSRRRRR010RRRRR0010011 slti rd,rs1,imm             # rv32i
// SSSSSSSSSSSSRRRRR011RRRRR0010011 sltiu rd,rs1,imm            # rv32i
// SSSSSSSSSSSSRRRRR100RRRRR0010011 xori rd,rs1,imm             # rv32i
//  3         2         1         0
// 10987654321098765432109876543210
// SSSSSSSSSSSSRRRRR110RRRRR0010011 ori rd,rs1,imm              # rv32i
// SSSSSSSSSSSSRRRRR111RRRRR0010011 andi rd,rs1,imm             # rv32i
// 00000UUUUUUURRRRR001RRRRR0010011 slli rd,rs1,imm             # rv128i
```

Figure 9: *Makemap of Prime Program*

4 Related Work

The main reference for this project is taken from GitHub and followed the commands that are needed to install the simulators. Also, we referred to the main websites like RISC-V, RV8 that has all the guidance to process the implementation. The Wikipedia as well as other websites has the best reference about Spike and RV8 simulators that are really helpful. Other than these simulators we also tried to install Gem5 and Firesim in our system, but we didn't had the time to complete the installation process. Thus we continued with Spike and RV8 simulators. More than that, GitHub will be the major resource for the information about processors and the simulators.

5 Conclusion

Even though RISC-V is considered as an open source instruction set, it is not yet been recognized completely. But for the upcoming years, RISC-V will become popular and will become helpful in installing different simulators on it. There is a lot of research that is been going on about RISC-V, many simulators were built. Other than Spike and RV8, we have Gem5, Firesim which works with the amazon web services and many more are yet to come. Overall, in the coming years, RISC-V will play a major role in improving the performance in computing applications.

References

[1] ASWaterman. *Spike, a RISC-V ISA Simulator*.

[2] Michael Clark. *RISC-V QEMU*.

[3] Michael Clark. rv 8 : a high performance risc-v to x 86 binary translator. 2017.

[4] Michael J Clark. *rv8.io-rv8 bench*.

[5] Powei Huang. *Tutorial on Spike Internal*.

[6] John D. Leidel. Stake: A coupled simulation environment for risc-v memory experiments. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, pages 365–376, New York, NY, USA, 2018. ACM.

[7] RISC-V. *RISC-V: The Free and Open RISC Instruction Set Architecture*.

[8] RV8. *RV8 RISC-V Simulator For X86-64*.

[9] Jim Wilson. *GNU toolchain for RISC-V, including GCC*.