gem-5 eXtensions for RISC-V: Full System Manual

[*]Embedded Systems Laboratory,
Swiss Federal Institute of Technology, Lausanne (EPFL)

[‡] REDS Institute
School of Engineering and Management Vaud (HEIG-VD),
University of Applied Sciences Western Switzerland (HES-SO)

Joshua Klein[*], Yasir Qureshi[*], Marina Zapater[*‡], and David Atienza[*]

June 2020

Contents

1 Executive Summary

1.1 Abstract

While RISC-V has enjoyed both strong functional simulation support via ISA simulators such as QEMU (2) and Spike (3) and RTL simulation support via Chisel and other HDL simulators (4), it has little support in the realm of full system-level simulators, especially for simulation of Linux-capable systems. This presents a bottleneck in the RISC-V hardware development process because it is difficult to quickly and reliably prototype and verify the performance of hardware designs for complex high-level applications, such as deep learning. To resolve this bottleneck, we present in this technical manual, *gem5-eXtensions for RISC-V*, or GXR5 : a functioning Linux-capable full system simulator built into the gem5 system-level architectural simulator and gem5-X (5)(6). We extend prior work by implementing the RISC-V privileged specification in gem5 (7). Our contributions include implementing privileged specification instructions and control and state registers (CSRs), support for user and supervisor privilege modes, a RISC-V compliant MMU capable of processing virtual memory, and ISA devices and interrupters, in addition to creating and configuring a gem5-compatible bootloader, device tree, Linux kernel, and disk image file system. With the privileged specification implemented and external components configured, we are able to demonstrate functionally correct execution of basic programs and benchmarks fetched from the disk image and executed on top of the Linux kernel.

1.2 Release Information

Version	Date	Changes
v1.0	June 2020	Initial release.

1.3 Collaboration and Contact Information

The maintainers of this project can be contacted via email at {joshua.klein, yasir.qureshi, david.atienza}@epfl.ch and marina.zapater@heig-vd.ch.

Because the scope of this project is very large, we are always interested in potential collaboration efforts to develop new features and bring GXR5 to gem5 master. For inquiries, source code, and additional information, please contact one of the aforementioned emails.

1.4 Licenses

2 Introduction

2.1 Motivation

Due to its status as an open-source and free instruction set, the RISC-V ISA has long been a popular candidate for implementing new hardware systems-on-chip (SoCs) and accelerators by striving to be the Linux of hardware (1). Even though it was only introduced in 2014, the open-source nature of RISC-V has led to varying levels of support across the computer architecture stack. Towards the software end, there are multiple Linux ports for various distributions (11) (12), as well as Linux kernel support upstream made possible by the RISC-V GNU toolchain (13). The hardware end is more limited however: there are multiple different kinds of simulators currently available for RISC-V and each represents a significant trade-off.

On one end of the spectrum, there are RTL simulators that typically interpret hardware description languages (HDLs) like Chisel and Verilog. They can provide extremely accurate simulations of hardware interactions, leading to precise performance results with respect to speed, latencies, bandwidth, power, and energy. However, even on the most powerful machines, RTL simulators can take on the order of weeks or even months to run and generate statistics for high-level, complex applications. This can significantly increase the time-to-market of a hardware product.

On the other end of the spectrum, there are functional ISA simulators such as QEMU (2) and spike (3) for RISC-V. These simulators only model the execution of instructions to verify the functional results of a program. While it is not possible to attain precise performance results of the underlying hardware with these simulators, one can very quickly load and run a program on top of an operating system using these simulators. What may take on the order of weeks or months in an RTL simulator can easily run on a functional simulator in minutes.

The middle ground between RTL and functional simulators, and the focus of this work, are system-level simulators. While not as fast to load and run programs as functional simulators, system-level simulators can represent major hardware components and interconnects as high-level software models with timing information, leading to the attainment of functionally accurate results as well as reasonable hardware performance statistics in significantly less time than RTL simulators. With extensions such as McPat, power and energy data can also be asserted by the generated performance statistics (14). While not being able to boast the same level of precision offered by RTL simulators, the end result is the ability to rapidly prototype and redesign hardware with reasonable insight into performance ramifications, thus decreasing the time to market of a hardware product.

Unfortunately, the premier system-level simulator in academia, gem5 (5), only has limited support for RISC-V. Prior work has implemented the unprivileged instruction set (15) as well as limited bare metal full system support for RISC-V (10). The RISC-V privileged ISA specification has not been implemented in gem5 however, and so it is impossible to leverage the benefits of this system-level simulator for complex applications running on top of a Linux system. Therefore the goal of GXR5 is to implement a Linux-capable full system simulator to allow for rapid design-space exploration of new system architectures for RISC-V.

2.2 Background

In this section we introduce the basic terms and ideas of the RISC-V instruction set, compare the unprivileged and privileged specifications, describe the target RISC-V execution stack, and introduce gem5 and gem5-X.

2.2.1 The RISC-V Instruction Set Architecture

Introduced in 2014, RISC-V is a free and open-source ISA that is built to have a minimalist base instruction set that is highly extensible and can still meet the demands of modern computer systems. The ISA comes in 32-bit, 64-bit, and 128-bit formats, which are denoted as RV32, RV64, and RV128, respectively, and all instructions are 32-bit (with the exception of shorter instructions introduced with the compressed ISA extension). The specification is split into multiple documents, including the base unprivileged specification, privileged specification, external debug specification, trace specification, and compliance framework (1) (9).

In order for a RISC-V system to run programs on top of the Linux kernel, it needs to, at minimum, implement the G (general purpose) and C (compressed) extensions from the unprivileged ISA specification, as well as the privileged ISA specification, which are explained below with respect to this work.

2.2.2 The RISC-V Unprivileged ISA Specification

The RISC-V unprivileged specification (in revision v2.1 as of writing this manual) specifies the existence, operation, formats, and bit codes for the base instruction set as well as numerous extensions. In addition to instruction listings, the unprivileged ISA also includes directives for interrupt subroutines (exceptions, traps, and interrupts), counters, and registers, as well as definitions for numerous RISC-V terms including different execution environments (EEs) and hardware threads (harts).

A Linux-capable RISC-V system with the G and C extensions is referred to as a RV32GC, RV64GC, or RV128GC system depending on the word size. The compressed extension defines 16-bit instruction layouts and the general purpose extension is a composite extension comprised of the *IMAFDZicsrZifencei* extensions, outlined in table 1.

Extension	Version	Description
RV32I	2.1	Base 32-bit integer extension.
RV64I	2.1	Base 64-bit integer extension.
M	2.0	Multiply/divide extension.
A	2.1	Atomic operations extension.
F	2.2	Single-precision floating point extension.
D	2.2	Double-precision floating point extension.
C	2.0	Compressed instruction formats extension.
Zicsr	2.0	CSR interface instructions extension.
Zifencei	2.0	Instruction-fetch fence instruction extension.

Table 1: RISC-V Unprivileged Specification GC extension versions as of writing this manual. All extensions presented above are ratified.

2.2.3 The RISC-V Privileged ISA Specification

The RISC-V privileged specification (in revision 1.11 as of writing this manual) specifies different privilege modes of operation (user, supervisor, hypervisor, and machine), and it is split into Supervisor-Level and Machine-Level ISAs (7).

The Machine-Level ISA contains the definitions and layouts of real CSRs and the privileged specification instructions accessible in M-mode (machine mode), as well as a description of physical memory protection and attributes (PMP/PMAs). The CSRs, unlike normal argument and temporary registers housed in a CPU's register file, may be memory-mapped and contain a lot of specific information pertaining to a variety of system functions, including the current status of the system, interrupt information, system information, and counters. The new instructions introduced in thee Machine-Level ISA include environment call and breakpoint instructions, trap-return instructions, and a wait for interrupt instruction.

The Supervisor-Level ISA contains the definition and layouts of supervisor CSRs, most of which shadow the existing Machine-Level ISA CSRs. It also introduces a supervisor fence instruction, virtual memory management, and a paging algorithm.

2.2.4 Introduction to gem5 and gem5-X

Introduced initially in 2011, gem5 is a modular computer architecture simulator that, unlike RTL simulators, enables system-level design space exploration by simulating high-level event-driven software models for processors, peripheral devices, and memory. It comes with numerous CPU, RAM, and device models right out of the box and includes varying levels of support for numerous ISAs including x86, ARM, and RISC-V.

gem5's primary running modes are Syscall Emulation (SE) and Full System (FS). SE mode enables program simulation with simple, emulated interrupt and syscall handling. FS mode on the other hand enables program simulation on top of a full system stack, including memory hierarchy, operating system, disk image, and full interrupt service routines by in interrupt controllers (5).

gem5-X, published in April 2019 by the Embedded Systems Laboratory at EPFL, extends gem5 by introducing architectural extensions such as in-cache computing and 3D-stacked HBM models, as well as a methodology for optimizing the power and performance of many-core systems (6). It also implements several enhancements in gem5, listed below:

- ARM-64 Full System support by way of an Ubuntu 16.04 LTS disk image with kernel v4.3.
- Profiling capabilities within the simulation using the gperf profiler.
- Enhanced checkpointing by way of Region-Of-Interest (ROI) marking in applications.
- 9P over Virtio for fast modification of files without modifying the root file system, enabled by default and built into the kernel.

2.3 Prior Work

Varying levels of support for RISC-V have been introduced to the main gem5 repository over the years, but unfortunately there has been no full published effort bringing Linux-capable FS mode systems for RISC-V in gem5.

The first published effort bringing RISC-V to gem5 was published in CARRV (workshop on Computer Architecture Research with RISC-V) 2017 by Alec Roelke and Mircea R. Stan (15). In their work, "RISC5: Implementing the RISC-V ISA in gem5", they implemented RISC-V unprivileged GC extensions and verified their results against a single-core system running in gem5 SE mode. This was extended by Tuan Ta, Lin Cheng, and Christopher Batten at CARRV 2018 with their work, "Simulating Multi-Core RISC-V Systems in gem5", which brought multi-core SE mode simulation to gem5 (16).

The work closest to bringing Linux-capable FS mode to gem5 for RISC-V systems is a Master's Thesis written by Robert Scheffel of Technische Universität Dresden (TUD) (10). In this work, Scheffel implemented a RISC-V bare-metal-capable FS mode system in gem5. In this case, Scheffel could run binaries stored on a simulated flash device loaded in RAM without an operating system or MMU. This required implementing interrupts and exceptions, in addition to a few peripheral devices implicitly required by the RISC-V ISA.

Finally, Nils Asmussen of the Barkhausen Institut contributed a RISC-V Sv39 MMU to the main gem5 repository for use with microkernels (24). This MMU is model is based off of the x86 MMU model in gem5 but does not support Linux.

2.4 Contributions

The base target execution environment for this work is a combination software and hardware execution environment with both an application binary interface (ABI) and supervisor binary interface (SBI). In other words, **our main contribution with gem5-eXtensions for RISC-V is creating the first Linux-capable FS mode system in gem5**. The work leading up to this is as follows:

1. We extend the FS mode configuration in gem5 for Linux-capable RISC-V systems.
2. We implement instructions from the RISC-V privileged specification and extend or verify instructions implemented in prior work.
3. We implement the missing Zifencei extension from the unprivileged specification.
4. We extend CSR implementations from prior work.
5. We develop a RISC-V compliant MMU capable of processing virtual memory, checking PMP/PMA, and interfacing a page table walker.
6. We implement RISC-V ISA devices, including a PLIC (platform-level interrupt controller) and CLINT (core-local interrupter).
7. We develop and configure a gem5-compatible bootloader, Linux kernel, device tree, and buildroot image for storage.

With these contributions combined, we are able to demonstrate running programs on top of the Linux kernel, on top of a disk image. In the next section we discuss and describe how to set up and use GXR5. In the rest of this technical manual we detail the high-level implementation of our contributions.

3 Running gem5 Full System Mode with RISC-V and Linux

In this chapter we describe how we configured and ran our RISC-V model, ending with a quick-start guide.

3.1 Necessary Files

Because our model is run in FS mode with a full Linux environment, we need several major system components not included with gem5. This includes,

- A bootloader
- A static kernel binary, e.g., vmlinux
- A file system/disk image
- A device tree binary

Additionally, all of the aforementioned components must be compatible with the RV64GC flavor of ISA.

All of the configuration files are located under system/riscv/ in the bootloader, disk, dt, and linux folders. In the following sections we describe the configuration options and how to use them in brief. Our steps for building each of the components follow from each respective component's own quick-start guide and it is assumed you have followed their guide and set up their respective environments. Once you have all of the files configured, it is possible to test them using the RISC-V variant of soft-mmu QEMU (2). Note that a common dependency is the RISC-V GNU toolchain, which includes a cross-compiler necessary for compiling the bootloader and kernel.

Additionally, you should also set up gem5 full system mode by creating the folders src/full_system_images/disks and src/full_system_images/binaries to house your additional files. These folders should sit on your M5 path after you set up gem5, which can be done with the following:

```
export M5_PATH=/path/to/gXR5/full\_system\_images
```

3.1.1 Device Tree

Our device tree is custom built, initially modified from a device tree generated by QEMU for a Fedora Linux emulation (2) (11). Once you have the device tree compiler set up, you can compile the device tree structure (dts) file using the following:

```
dtc -I dts -O dtb gem5-simple-rv64.dts -o gem5-simple-rv64.dtb
```

The output of this command is a device tree binary (dtb) file that can then be used by copying it to your GXR5 FS mode directory, under full_system_images/binaries.

3.1.2 Bootloader

The bootloader we used for testing is the Open Supervisor Binary Interface, or OpenSBI. OpenSBI is a first-stage bootloader and platform-independent M-mode firmware designed to link a library with a simple set of defined methods for a specific platform. The RISC-V manual also includes a SBI specification that OpenSBI conforms to (9) (17).

For our experiments, we tested the OpenSBI bootloader compiled for the qemu/virt platform using a custom configuration file. Copy the configs.mk, objects.mk, and platform.c files to the qemu/virt directory in your OpenSBI folder. After this, you can run the following commands to build the bootloader:

```
git clone https://github.com/riscv/opensbi.git
cd ./opensbi
git checkout 813f7f4c250af9f7c9546f64778e9b35bb7d7dcb
export CROSS_COMPILE=/path/to/riscv/compiler/here
export PLATFORM_RISCV_XLEN=64
make PLATFORM=qemu/virt O=/path/to/your/build/directory
make PLATFORM=qemu/virt I=/path/to/your/install/directory
```

The result of these commands, if run successfully, should include a file ”fw_jump.elf”. This can be used as a standalone bootloader which launches a vmlinux file during run-time. To install for GXR5 , copy the file to full_system_images/binaries.

3.1.3 Linux Kernel

Under most circumstances, gem5 is only able to directly run static binaries, and thus we must use a static version of the Linux kernel. This refers specifically to the file generated by building the Linux kernel, vmlinux.

For our tests, we built and used Linux kernel version 5.5 directly from the Linux repository (18). You will need to git checkout this version and then rebuild Linux with our custom configuration file. Copy one of the provided Linux configuration files into your Linux directory as the file ”.config”. You can run the following:

```
git clone https://github.com/torvalds/linux.git
cd ./linux
git checkout v5.5
make ARCH=riscv CROSS_COMPILE=/path/to/riscv/compiler/here
make ARCH=riscv CROSS_COMPILE=/path/to/riscv/compiler/here all
```

The result of running these commands should be both a vmlinux and Image file containing a RV64GC-compatible kernel. Though the Image file is not used in gem5, you can use it in QEMU to verify a successful RISC-V build in lieu of the vmlinux file.

Like before, in order to use your vmlinux file you will need to copy over to your GXR5 directory under full_system_images/binaries.

3.1.4 Disk Image

The disk image we used is a minimal rootfs created by buildroot (19). Similarly with building Linux, in order to build the root file system you will need to copy over the configuration file and make. **Note that if you previously did not set up buildroot with RISC-V, buildroot will need to download and set up the entire RISC-V GNU toolchain, so your first time creating the minimal rootfs will likely take tens of minutes** (if not longer) depending on your computer and internet connection.

So just like with Linux, you will need to copy the provided buildroot.config file into the .config file of the buildroot directory. By default, the rootfs file will be a 60MB image with only the minimal root file system. The username and password are set to root/buildroot by default, but this can be changed in the nconfig menu for buildroot, or for auto-login, in /etc/inittab of the generated rootfs.ext2. Note that the rootfs file is technically marked as an ext4 file system, but what is generated is an ext2 file system with an ext4 link. Linux will still treat this file system as ext4 however.

So after you have completed configuration of your rootfs file, you are ready to build it. Simply call ”make” in the buildroot directory and the disk image, rootfs.ext2, will be available in output/images. In order to use it, copy it over to your gXR5 directory under full_system_images/disks.

Lastly, just like with the Image file in Linux, you can use this rootfs file with RISC-V QEMU to verify its contents. Because it is a miniaml root file system however, it will not have the proper utilities for creating test programs, so simply mount it and copy over files from your host system using the following:

```
    sudo mount <path_to_rootfs>/rootfs.ext2 /path/to/mount/point
    cp /your/binary /path/to/mount/point
    sudo umount /path/to/mount/point
```

3.2 Recommended Script Options

In addition to the files included above, we tested our model using the AtomicSimpleCPU model, 2GB of RAM, and the DDR3_2133_8x8 memory type. Future work will include testing and verification of other CPU models with respect to real hardware.

3.3 Quick-Start Guide

In this brief start-up guide, we will guide you through the basic steps to running your first experiment with gXR5. This guide assumes you have already built a working bootloader, device tree, static kernel file, and disk image using the aforementioned configuration files as described in the previous sections and placed them in the proper file locations: gem5-simple-rv64.dtb, fw_jump.elf, and vmlinux in gXR5/full_system_images/binaries and rootfs.ext2 in gXR5/full_system_images/disks.

3.3.1 Prerequisites

You will need to set up the gem5 environment in order to compile and run the gem5 binary using the scons (SConstruct) builder. If running on an Ubuntu-based host system, you can use the following command to get all the required libraries:

```
    sudo apt install build-essential git m4 scons zlib1g \
    zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev \
    libgoogle-perftools-dev python-dev python-six python \
    libboost-all-dev
```

3.3.2 Building the gem5 Binary

Once the above is done, you will need to build a RISC-V gem5 binary. You can create multiple builds including .fast, .opt, and .debug. If you are only concerned about running experiments, it is recommended to only create gem5.fast. However, if you need to debug anything or want to generate traces, you will need to build gem5.opt or gem5.debug. Do this with the following:

```
    cd gXR5
    scons build/RISCV/gem5.{fast, opt, debug}
```

Additionally, if you would like to speed up the compilation process, you can use the option ”-jN” on the scons build line where N is the number of threads you want to assign for compilation.

3.3.3 Setting Up Your Experiment

Before you start running gem5, you will need to set up the rootfs image to run your desired program. Copy over your desired program to the rootfs, and then modify the /etc/inittab in your rootfs to include the program you want to run. The inittab is responsible for initializing various file systems and the login screen but simple programs should be safe to run this way because the kernel has already entirely booted by the time inittab is run.

For example, if you want to run a cross-compiled binary called ”rvhello” sitting in my file system’s bin folder, you would run the following:

```
    riscv64-linux-gnu-gcc rvhello.c -o rvhello
    mkdir /mnt
    sudo mount gXR5/full\_system\_images/disks/rootfs.ext2 /mnt
    sudo cp rvhello /mnt/bin
    sudo vi /mnt/etc/inittab
    sudo umount /mnt
```

When you run ”sudo vi” in the above, you will modify the inittab to have the following:

```
# /etc/inittab
#
# Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>

...

# Run your binary
::sysinit:/bin/rvhello

...
```

Exit vi with escape + ”:wq” + enter, and then you can unmount your file system and you are ready to run your experiment.

3.3.4 Running Your Experiment

Once your program sits in your rootfs and your inittab file is modified appropriately, run GXR5 using a script that has some variation of the following:

```
    cd gXR5
    ./build/RISCV/gem5.{fast, opt, debug} \
    -d /path/to/your/output/directory \
    ./configs/example/fs.py \
    --disk-image=full_system_images/disks/rootfs.ext2 \
    --kernel=full_system_images/binaries/vmlinux \
    --os-type=linux \
    --dtb-filename=full_system_images/binaries/gem5-simple-rv64.dtb \
    --cpu-type=AtomicSimpleCPU \
    -n 1 \
    --mem-size=8GB \
    --mem-type=DDR3_2133_8x8 \
    --cpu-clock=1GHz \
```

At this point you should be able to connect to your running gem5 instance in another shell with,

```
    telnet localhost 3456
```

Upon connecting to your gem5 instance, you should be able to see the OpenSBI logo as well as the kernel dmesg, followed by whatever output is apart of the program you are running (including basic ”Hello, world!” programs and other benchmarks).

4 Full System Configuration Overview

The goal with this first revision of a RISC-V FS mode-compatible model was to create the simplest possible computer system for running basic benchmarks that still aligns with performance and power models of real RISC-V processors. In this chapter we describe the FS mode configuration in detail, and, in brief, our platform.

4.1 Full System Model and Child Tree

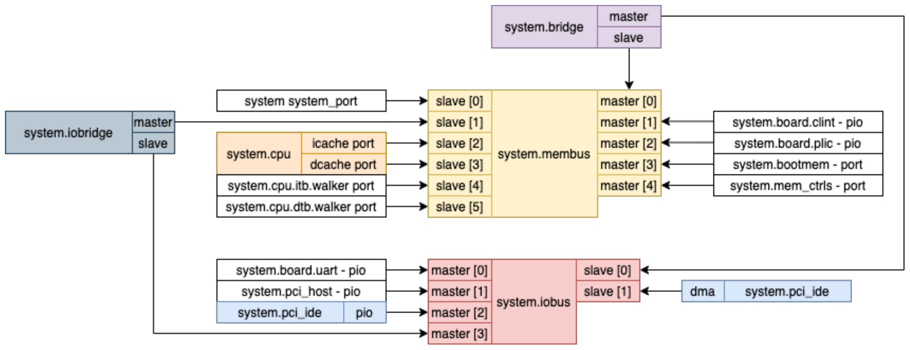

Figure 1: RISC-V Full System configuration with all models used for simulation and their interfaces. The diagram on the left An arrow from object A pointing to object B indicates that A sets its associated port to B within the gem5 configuration script.

The RISC-V Full System model consists of numerous built-in gem5 models as well as custom ISA-specific models. As shown in figure 1, the model includes a single-core CPU with instruction and data caches, TLB walkers for each of the aforementioned caches, a (currently unused) bootmem, a Platform-Level Interrupt Controller (PLIC), a Core-Local Interrupter (CLINT), a PCI host with IDE controller, and lastly a UART module. All models are connected to either a memory bus (membus) or IO bus (iobus), and these buses are interfaced through both a system bridge and IO bridge. Of these models mentioned, only the PLIC and CLINT are RISC-V-specific models required by the ISA. Additionally, the PCI host is customized for our implementation. The rest of the models are ISA-independent and are provided by gem5's built-in utilities.

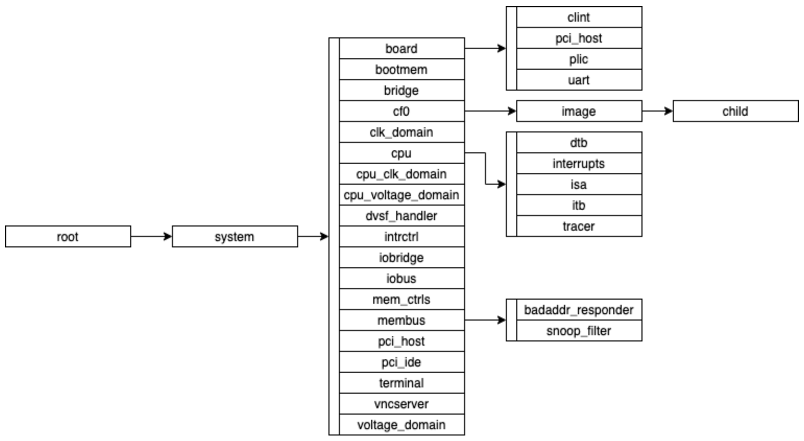

Figure 2: RISC-V Full System gem5 child tree.

The child tree is shown in figure 2 and shows the gem5 model hierarchy of the simulated system.

The full system configuration is implemented in configs/example/fs.py and configs/common/FSConfig.py. fs.py contains the general FS-mode setup while FSConfig.py contains the specific RISC-V system setup, including mapping memory and making all port connections.

4.2 Full System Memory Map

The full system memory map can be seen in table 2. It is defined in configs/example/fs.py and src/dev/riscv/SimpleBoard.py.

Device	Address Range
PLIC	`0x0c000000:0x0c2fffff`
UART	`0x10000000:0x10000100`
CLINT	`0x20000000:0x2effffff`
PCI	`0x2f000000:0x5fffffff`
RAM	`0x80000000:0xffffffff`
– bootloader	`0x80000000:0x801fffff`
– kernel	`0x80200000:0x87ffffff`
– device tree	`0x88000000:0x8fffffff`

Table 2: Rough memory map of simulated full system.

For a precise listing of all CSRs are their offsets, please refer to the ISA specification (7) (8).

4.3 SimpleBoard Platform

To interface ISA-specific devices as well as external devices, we further develop the SimpleBoard platform that was initially introduced in Scheffel's thesis (10). The relation between the CPU core and SimpleBoard SoC, as well as external devices, can be seen in figure 3.

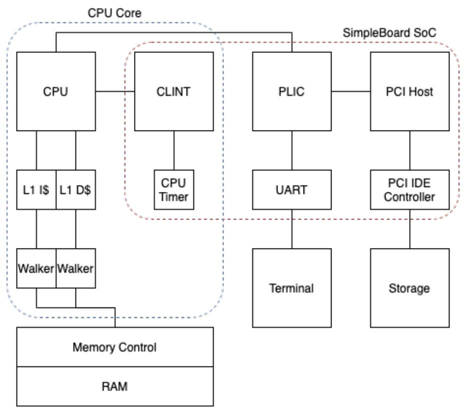

Figure 3: Target hardware system for emulation.

In order to enable gem5 FS mode, models for both a CLINT and PLIC needed to be developed and interfaced through the SimpleBoard platform. Additionally, a platform in gem5 is used to also interface certain external interrupts such as interrupts from PCI devices as well as implement timers, connect a terminal, etc. All code for the SimpleBoard platform and associated devices can be found in src/dev/riscv.

5 Implementation of Privileged RISC-V ISA Specification

In this section we go into more detail about how the privileged ISA specification was implemented in GXR5 , including the CSRs, instructions, interrupt handling, and virtual memory subsystems.

5.1 Control and State Registers (CSRs)

In this section, we discuss CSRs in RISC-V and their implementation in gem5.

5.1.1 CSRs versus Other Registers

The RISC-V ISA specification implicitly designates two kinds of registers: the aforementioned Control and State Registers and what this manual will refer to as 'normal' CPU registers. The normal registers include the common registers seen across many ISAs, including argument registers, temporary registers, stack pointer, program counter, etc. CSRs however contain very specific information with respect to the operation of RISC-V systems. Furthermore, CSRs don't have to be in the register file of the CPU and can instead be *memory-mapped* to help interface external devices. Keep in mind however that a CPU will still see and access memory-mapped CSRs as though they were in the register file.

Some of the most important CSRs include *mstatus*, which contains information about the current running status of the system, *mip* and *mie*, which show the pending and enabled interrupts in the system, and *mtime*, which is used as a global system timer. The full CSR listings can be found in the RISC-V privileged specification (7).

CSR names that start with 'm' refer to a CSR only accessible in machine (M) mode. The RISC-V privileged specification also lists supervisor (S) mode CSRs, most of which are not actually CSRs separate from their M-mode counterparts but are shadows of the same M-mode CSRs where M-mode-only bits cannot be reliably read or written from. There are some CSRs completely unique to S-mode however, such as the *satp* CSR.

5.1.2 CSRs in gem5-X

The register file for RISC-V is located in src/arch/riscv/registers.hh. This file defines the maps that contain both the normal register file registers as well as the CSRs. It is in this file that CSR indices, names, and offsets are stored and referenced in gem5. Additionally, the registers.hh file defines BitUnions for easy access to the different fields of the CSRs. Most of these CSRs were previously defined in prior work, and only some performance and timing registers needed to be added for GXR5 .

5.1.3 Interfacing and Accessing CSRs

As the register file is usually unique for each hart, the usual way to interface the CSRs is via the threadContext class using the methods getMiscReg and setMiscReg. These methods are defined and implemented for RISC-V in src/arch/riscv/isa.hh and src/arch/riscv.isa.cc, respectively.

Hardware events usually call the ISA methods when a CSR needs to be checked or modified. For example, a CPU read to the *mtime* CSR must read the external system clock, and therefore the ISA interface will access the system's CLINT to read that CSR. Another example is when an external interrupt needs to be posted by the PLIC, it will update the *mip* CSR through the ISA interface.

Note that with memory-mapped CSRs, the ISA interface only defines direct access to the register. Usually when a CSR must be accessed directly by address, it is through gem5's packet interface (typically associated with PIO devices), and so another interface is defined to connect external devices and the ISA interface in src/arch/riscv/isa_device.hh (and subsequently implemented in src/arch/riscv/isa_device.cc).

5.2 RISC-V Instructions

In this section, we discuss the new instruction implementations in RISC-V and gem5.

5.2.1 Instructions in gem5

gem5 implements instructions by using ISA template files that then generate all of the instruction classes and functionality during the SConstruct build process. The ISA templates for RISC-V are located in src/arch/riscv/isa/. The primary file used for implementing individual instruction functions is src/arch/riscv/isa/decoder.isa, which decodes machine code to assign and process instruction types. Most of the RV64GC instructions from the unprivileged ISA specification were implemented in prior work.

5.2.2 Privileged Specification Instructions

The privileged RISC-V spec adds very few instructions to the base ISA. These instructions are URET, SRET, MRET, SFENCE.VMA, WFI, HFENCE.BVMA, and HFENCE.GVMA. As of writing this manual, the Hypervisor specification has not yet been ratified, so we forego the implementation of the HFENCE instructions. The table of instruction layouts (Table 5.1 in the unprivileged specification document) is recreated in table 3 for convenience.

00000000	00010	00000	000	00000	1110011	URET
00010000	00010	00000	000	00000	1110011	SRET
00110000	00010	00000	000	00000	1110011	MRET
00010000	00101	00000	000	00000	1110011	WFI
00000000	rs2	rs1	000	00000	1110011	SFENCE.VMA
00000000	rs2	rs1	000	00000	1110011	HFENCE.BVMA
00000000	rs2	rs1	000	00000	1110011	HFENCE.GVMA

Table 3: Instructions introduced in the RISC-V Privileged ISA specification and their bit layouts.

The URET, SRET, and MRET instructions are all trap return instructions for specific privilege modes. These instructions were initially implemented in prior work and only verified for use with Linux-capable FS mode in this work.

The WFI instruction is a ”wait for interrupt” instruction. The RISC-V privileged specification states that a no-operation (NOP) is a valid implementation for WFI, so we left the instruction as such.

The SFENCE.VMA instruction is a supervisor fence that is used to enforce memory ordering. While the RISC-V privileged specification goes into very tedious and specific detail about memory ordering and constraints, the result of the specification's discussion is a relatively simple hardware cache flush, implemented in gem5 using the overridden demapPage method found in src/arch/riscv/tlb.cc. A call to SFENCE.VMA will flush a specific page or set of pages of the L1 ITB and DTB caches by address space number (ASN), virtual address, both, or neither (resulting in a full cache flush) depending on the values of its arguments rs1 and rs2.

5.2.3 Unprivileged Specification Instructions

Only two instructions necessary for Linux-capable FS mode were missing from prior work. The instructions and their layouts are in tables 4 and 5.

fm	pred	succ	rs1	000	rd	0001111	FENCE

Table 4: Missing fence instruction from the RV32/64 Base Instruction Set.

imm[11:0]	rs1	001	rd	0001111	FENCE.I

Table 5: Missing fence instruction from the RV32/RV64 *Zifencei* Standard Extension.

The FENCE instruction is simply implemented as a flush instruction for the entire L1 instruction and data caches (ITB and DTB). The FENCE.I instruction is specifically for L1 ITBs, so only the ITB is flushed. The cache flush implementation in gem5 is the same as described in the previous section for the SFENCE.VMA instruction.

5.3 Fault Handling

In this section, we discuss faults, exceptions, and interrupts in RISC-V and their implementation in gem5.

5.3.1 RISC-V Terminology

The RISC-V ISA specification defines numerous terms relating to trap/faults. A synchronous fault is referred to as an *exception*, while an asynchronous fault is referred to as an *interrupt*.

Exceptions include faults due to misaligned instructions/addresses, invalid instruction/address access, illegal instruction calls, breakpoints, environment calls, and page faults. Interrupts include software, timer, and external asynchronous faults in different privilege modes.

The fault number of an exception or interrupt is stored in the cause CSR, which holds fault causes in a one-hot representation and reserves its top bit to indicate if the fault is an interrupt or an exception.

5.3.2 RISC-V Fault Handling Algorithm

Because RISC-V relies on its various binary interfaces for fault handling, a lot of the algorithm for handling various faults is delegated to software. This makes the implementation of fault handling in hardware extremely easy. The RISC-V hardware simply saves the current context (privilege mode, current PC value), escalates the privilege mode (unless delegated via the mideleg and medeleg CSRs), and sets the PC to the saved address of the fault handler. The cause value stored in the m/s/ucause CSR will tell the software fault handler which specific routine needs to be taken. When the fault is handled, it calls the m/s/uret instruction to restore context and continue program execution.

5.3.3 Exceptions in gem5

Prior work had defined and implemented RISC-V exceptions in src/arch/riscv/faults.hh and src/arch/riscv/faults.cc, respectively. All fault types derive from a base RiscvFault class and most faults will use the same invoke method which implements the fault handling described in the previous section. The derived fault classes are mostly for implementation simplicity, but in this work we also had to ensure privilege modes were implemented for the ecall fault type.

Address, misalignment, instruction, and page faults of all types are typically posted by the TLB, defined in src/arch/riscv/tlb.cc.

5.3.4 Interrupts in gem5

Interrupts in RISC-V use the same fault handling algorithm as exceptions with some minor changes based on the fault code and cause CSR, and therefore use the same aforementioned fault handler defined for exceptions in gem5. To use the same fault handler, prior work had already defined and implemented an InterruptFault class in src/arch/riscv/faults.hh and src/arch/riscv/-faults.cc, respectively.

Interrupts are posted from different (usually external) sources at asynchronous times. gem5 offers a CPU interface with postInt and clearIInt methods that set and clear an interrupt pending array defined in src/arch/riscv/interrupts.hh.

5.4 Virtual Memory

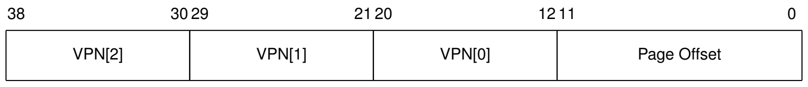

Figure 4: Sv39 virtual address.

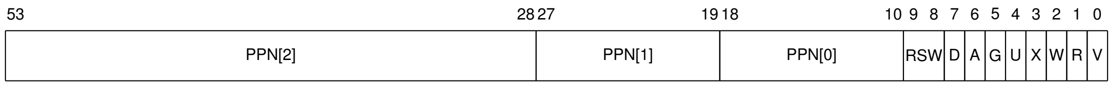

Figure 5: Sv39 page table entry.

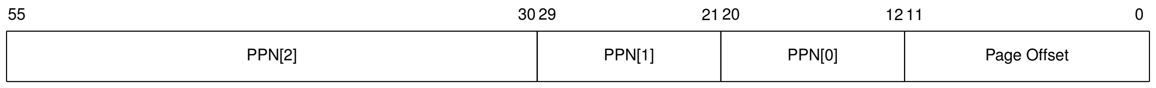

Figure 6: Sv39 physical address.

The satp (supervisor address translation and protection) CSR is responsible for determining the status of virtual memory. It contains three fields: the mode of translation, the address space identifier (ASID), and finally the physical page number of the root page table. When the mode is set to bare or the privilege mode is in M-mode, address translation is direct and all addresses are considered ”physical”. When the mode field is not bare, there are four modes of virtual address translation available.

In GXR5, we support only Sv39 virtual address translation. This is 39-bit virtual addressing that translates a 39-bit virtual address to a 56-bit physical address. The bit formats for Sv39 virtual address, page table entries, and physical addresses, are found in figures 4, 5, and 6. Virtual address translation is mostly implemented in src/arch/riscv/tlb.cc, and is discussed in more detail in the next chapter.

6 Virtual Memory Subsystem

In this section we expand upon the previous by going in-depth into our virtual memory implementation, which includes a RISC-V-compliant Sv39 MMU consisting of a TLB and page table walker.

6.1 TLB Implementation

gem5 implements a base TLB class that is used for instruction and data caches. Our gem5 TLB implementation is largely based on the TLB implementation for the ARM ISA, but with the RISC-V virtual address translation algorithm implemented. In other words, functional, atomic, and timings-based address translation calls are routed either through the translateFs or translateSe methods. In our case, we move prior work to the translateSe method and focus on implementing the translateFs method. The TLB description and implementation are located in src/arch/riscv.hh and src/arch/riscv/tlb.cc, respectively. Additionally, some page table structures are implemented and defined in src/arch/riscv/pagetable.hh.

The local page table of the TLB is stored in a CPP map data structure for simplicity. This data structure maps virtual address bases (virtual address without offset) to TlbEntry structures. The TlbEntry structures include the virtual address, physical address, page table entry, and ASID.

The actual address translation takes place in the translateFs method of the TLB. This method is the main workhorse for all address translation types, PMP/PMA checking, and cache management with respect to other TLB methods.

6.2 Physical Address Translation

When the satp CSR is set to bare translation mode, or the translation is occurring in machine mode, the physical address is translated directly and hence there is no need to cache the address. The address is simply checked against the PMP/PMA checker for potential access faults, before the physical address field of the requesting packet is simply set to the unchanged physical address.

6.3 Virtual Address Translation

When the satp CSR is not set to bare translation mode and the mode is supervisor or user, virtual address translation is required. The translation process starts with checking the TLB's map data structure. If there is a TLB hit, we simply translate the virtual address using the cached physical address. If there is a TLB miss, we proceed with the virtual address translation algorithm explained in the RISC-V privileged ISA specification, section 4.3.2. The source code is annotated with each individual step of the virtual address translation algorithm.

6.4 Page Table Walker Implementation

The page table walker definition and implementation are in src/arch/riscv/table_walker.hh and src/arch/riscv/table_walker.cc, respectively. Because a page table walker is a purely hardware component with no RISC-V specification, we use a very simple design that only acts to facilitate DMA transactions via the walk method.

The table walker is set up using gem5's ports interface to connect it both to the L1 caches and directly to the memory. Memory is accessed directly via the dmaAction method and the result is stored locally.

7 SimpleBoard Platform

The SimpleBoard platform, extended from prior work, inherits from the gem5 platform class. Platforms are typically used to define SoCs and various on-chip interconnects. Though our target SoC in future work will be the HiFive Unleashed 1 board, this SimpleBoard platform represents the simplest possible implementation of a platform for RISC-V that can interface its I/O devices for Linux-capable FS mode.

7.1 SimpleBoard Implementation

Our definition and implementation of the SimpleBoard platform can be found in src/dev/riscv/ simpleboard.hh and src /dev/riscv/simpleboard.cc, respectively. Additionally, the simulation object parameters for the SimpleBoard platform, in addition to the pythonic class definitions for the other devices hosted on the SimpleBoard are located in src/dev/riscv/SimpleBoard.py.

In addition to hosting SoC devices for RISC-V systems, the SimpleBoard platform is also used to interface PCI and console interrupts, as well as host a generic UART device. The PCI interrupt interface is required by our implementation for disk access, however the console interrupt interface is currently unused.

The SimpleBoard SoC is instantiated and connected to the main system through the gem5 FS mode configuration script.

7.2 SimpleBoard Devices and Parameters

All of the SoC devices are defined and linked to the SimpleBoard platform in src/dev/riscv/SimpleBoard.py. These devices include the PLIC, CLINT, UART, and a RISC-V PCI host. A visual representation of these devices can be found in figure 7.

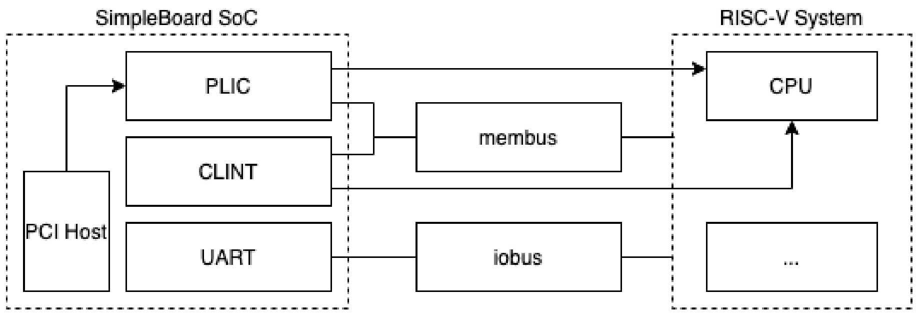

Figure 7: A rough diagram showing the interconnects between the SimpleBoard SoC devices and the RISC-V system. The lines with arrows represent interrupt lines to the PLIC and CPU while the other connections are either peripheral I/O (PIO) interconnects or interconnects via the isa_device interface. PIO interfaces use packets and addresses to connect these devices.

8 Platform-Level Interrupt Controller (PLIC)

In order to handle potentially simultaneous interrupts signals that come from peripheral devices, the RISC-V ISA manual specifies the existence of a PLIC. The goal of the PLIC is to receive all external interrupt signals (so-called 'global interrupts'), order them by priority, and then delegate them to an available hart for handling.

For every potential source of a global interrupt, the PLIC defines a source number and highest allowable priority mask. An interrupt source numbered 1 with a priority 7 has the highest interrupt priority. Interrupt source 0 is reserved to mean 'no interrupt'.

8.1 PLIC Implementation

In our implementation, the PLIC is a BasicPioDevice on the SimpleBoard platform, and the first of our ISA devices. The definition and implementation of the PLIC are found in src/dev/riscv/plic.hh and src/dev/riscv/plac.cc, respectively. The configuration of the registers is defined as a map in the source code and follows the same layout at the PLIC described in the FU540-C000 core manual (20). This layout is reproduced in the next section.

8.2 PLIC Registers and Memory Layout

All PLIC registers are 32-bit and are listed in table 6. Additionally, all PLIC registers are read-write registers, except for the pending array registers which are read-only. 64-bit registers are split into two 32-bit registers with a low and high field that is automatically interfaced by RISC-V software when accessing the PLIC.

Note that this PLIC was designed for the HiFive Unleashed SoC with five harts (one small CPU core and 4 large CPU cores), even though our simulated FS mode system only simulates one core (and therefore one hart) currently. We keep the registers for other harts for future work.

8.2.1 PLIC Source Registers

Each source number refers to a specific external device and designates a priority. For example, in the FU540-C000 manual, sources 1-3 refer to the L2 cache controller, source 4 refers to UART0, and so on. The values stored in the registers are 3-bit source priority values. A value of zero indicates that interrupts are disabled for the source, while a value of seven indicates that interrupts are of the highest priority for the source.

Like with the additional registers for additional harts, we preserve all of the source registers for future work. In our FS mode simulation, we technically only use the sources for UART and the PCI host.

8.2.2 PLIC Interrupt Pending Array

The PLIC interrupt pending array is a one-hot representation of pending interrupts where the bit index refers to the source number. For example, if bit 12 of the interrupt pending array is high, it means source 12 is awaiting interrupt handling.

Offset	Register
0x0c000004	Source 1 Priority Register
0x0c000008	Source 2 Priority Register
...	...
0x0c0000d8	Source 54 Priority Register
0x0c001000	Interrupt Pending Array (low)
0x0c001004	Interrupt Pending Array (high)
0x0c002000	Hart 0 M-Mode Interrupt Enable (low)
0x0c002004	Hart 0 M-Mode Interrupt Enable (high)
0x0c002080	Hart 1 M-Mode Interrupt Enable (low)
0x0c002084	Hart 1 M-Mode Interrupt Enable (high)
0x0c002100	Hart 1 S-Mode Interrupt Enable (low)
0x0c002104	Hart 1 S-Mode Interrupt Enable (high)
...	...
0x0c002380	Hart 4 M-Mode Interrupt Enable (low)
0x0c002384	Hart 4 M-Mode Interrupt Enable (high)
0x0c002400	Hart 4 S-Mode Interrupt Enable (low)
0x0c002404	Hart 4 S-Mode Interrupt Enable (high)
0x0c200000	Hart 0 M-Mode Priority Threshold
0x0c200004	Hart 0 M-Mode Claim/Complete
0x0c201000	Hart 1 M-Mode Priority Threshold
0x0c201004	Hart 1 M-Mode Claim/Complete
0x0c202000	Hart 1 S-Mode Priority Threshold
0x0c202004	Hart 1 S-Mode Claim/Complete
...	...
0x0c207000	Hart 4 M-Mode Priority Threshold
0x0c207004	Hart 4 M-Mode Claim/Complete
0x0c208000	Hart 4 S-Mode Priority Threshold
0x0c208004	Hart 4 S-Mode Claim/Complete

Table 6: PLIC register listings.

8.2.3 PLIC Interrupt Enable Registers

The PLIC interrupt enable registers masks the available interrupts using the same one-hot representation used by the interrupt pending array. If an interrupt is pending and a hart is ready to handle an external interrupt, the hart will check the interrupt pending array against its assigned interrupt enable array, based on the current hart's number and privilege mode.

8.2.4 PLIC Priority Thresholds

The PLIC priority threshold register holds a 3-bit priority value. In order for a hart of specific number and privilege mode to be able to handle an interrupt, its threshold value must be lower than the source priority value.

8.2.5 PLIC Claim/Complete Registers

The PLIC claim and complete registers are used for the PLIC claims process, described in the next section.

8.3 PLIC Claim/Complete Operations

At a high level, the PLIC is meant to perform two major operations in addition to standard reads and writes to its registers: PLIC claim, and PLIC claim complete.

When a hart is ready to handle an external (global) interrupt, it 'claims' the interrupt (making it unavailable to other harts) by sending a read request to the hart's claim register. This will read the PLIC's interrupt pending array, clear the bit associated with the highest priority pending interrupt, and return the source number to the hart. The hart can then use the source number to jump to the correct interrupt handler for the source.

When a hart is finished handling an external interrupt, it performs a 'claim complete', where it sends a write request to its claim/complete register. This register will hold the source number of the last completed external interrupt.

8.4 PLIC postInt and clearInt Methods

In our FS mode implementation, we use postInt and clearInt to send interrupts to the CPU. These methods are used primarily by the platform to forward PCI interrupts. Additionally, we only send M-mode and S-mode interrupts to the CPU; U-mode external interrupts can be delegated depending on the mideleg CSR.

The sendInt method will send an external interrupt to the CPU if the source priority exceeds the source threshold as well as set the appropriate bit in the PLIC pending array. Likewise, the clearInt method will clear an interrupt in the CPU and clear the appropriate bit in the PLIC pending array.

9 Core-Local Interrupter

The CLINT, expanded from prior work (10), houses the system's main clock. As a result, the CLINT is responsible for posting timer interrupts to its local CPU and associated harts. While there would usually only be one PLIC in a real system, there is usually one CLINT per hart.

In addition to posting timer interrupts, the CLINT is also responsible for posting software interrupts. Our implementation loosely follows that of the CLINT implementation of the HiFive Unleashed SoC, so our CLINT will post only M-mode software interrupts (20). As software interrupts are typically only used for cross-hart communication, they remain untested in GXR5.

9.1 CLINT Implementation

Like the PLIC, our CLINT is implemented as a BasicPioDevice on the SimpleBoard platform that is also an ISA device. The definition and implementation of the CLINT are found in src/dev/riscv/clint.hh and src/dev/riscv/clint.cc, respectively. In GXR5, the CLINT is set up such that there is only one instance of the object in the whole system, but to handle multi-core and multi-hart systems in future work, we define a CpuTimer class that contains core-local and hart-local registers. The timer in the CLINT is, by default, set to 1MHz. The registers and layout of the CLINT are shown and described in the following section.

9.2 CLINT Registers and Memory Layout

Like with the PLIC, the CLINT register layout follows that of the FU540-C000 core from the HiFive Unleashed SoC, and can be seen in table 7. There is only one mtime register in the CLINT, but there is one mtimecmp and msip register per hart. All registers are read-write. Unlike the PLIC, the offsets of the mtimecmp and msip registers are generated as a function of the base offset added to the width of those register.

Offset	Bit Width	Register
0x00000000	4	Hart 0 Interrupt Pending
0x00004000	8	Hart 0 Time Compare
0x0000bff8	8	Hart 0 Timer

Table 7: CLINT register listings.

For example, for hart 1 the offset of the interrupt pending (msip) and time compare (mtimecmp) CSRs would be 0x00000004 and 0x00004008, respectively.

9.2.1 CLINT Timer Register and the mtime CSR

The mtime CSR from the CPU's register file is a memory-mapped CSR. A read/write operation to the CSR must perform the operation to the actual register location, which in our case, is the CLINT. The CLINT stores the mtime CSR as a simple 64-bit unsigned integer, which can be read/written to directly via address or via the ISA device interface.

In a real (non-simulated) system, the CLINT's timer would be tied to an external oscillator tuned to a constant frequency. Reading the mtime CSR means reading the timer register, which returns the number of cycles the oscillator has processed since system start. As gem5 is an event-based simulator however, implementing a timer directly in this way would be incredibly inefficient and increase simulation times significantly due to the number of added discrete events to the event queue. Therefore, the timer only changes with discrete events – primarily reads/writes to the CLINT.

When the timer needs to be updated, it is done through the updateTime method. This method calculates the timer's current value as a function of the default clock frequency of the timer and the current tick of the simulator. Note that it is also possible to write to the timer register, and therefore we preserve an offset value as well to incorporate into the timer update method.

9.2.2 CLINT Time Compare Register

The CLINT timer compare register, mtimecmp, is a register that determines when a timer interrupt must be posted. A timer interrupt is posted whenever the mtime CSR is greater than or eqeual to mtimecmp. Setting mtimecmp to INT_MAX effectively disables timer interrupts. The mtimecmp register only determines when a timer interrupt is posted to the hart it is attached to, and in a MPSoC, there would be one mtimecmp register per hart.

9.2.3 CLINT Software Interrupt Pending Register

The msip register is supposed to hold one bit that indicates of a software interrupt is pending or not. In our implementation however, we tie the msip register directly to the mip CSR, and thus a read or write operation to the msip register in the CLINT is directly reflected in the mip CSR instead.

9.3 CLINT Timer Interrupts

Timer interrupts occur whenever a CPU's mtimecmp register is greater than or equal to the mtime CSR. In our implementation, we use gem5's event queue to schedule ”timer alarms” that go off and post either M-mode or S-mode timer interrupts to the CPU.

Initially, the simulated system starts with mtime at 0 and timecmp at INT_MAX, so timer interrupts are disabled. mtime will be modified any time it is read or written to by the simulation. When mtimecmp is written to, mtime is updated and an event (timer alarm) is scheduled for when mtime should be greater than or equal to mtimecmp. This timer alarm, when run, will verify mtimecmp and mtime, and then post a timer interrupt to the CPU. The software interrupt handler will then reset mtimecmp, which writes mtimecmp and consequently starts the timer alarm process again.

If mtimecmp is written to again in between the aforementioned cycle, the previous timer alarm event is descheduled and a new one is scheduled in its place. Note that even setting mtimecmp back to INT_MAX would still set a timer alarm, although at a tick extremely far into the future.

10 Miscellaneous SimpleBoard SoC Devices

10.1 RISC-V PCI Host

For access to a disk image, gem5 uses a simulated PIIX4 IDE controller connected via PCI bus to a generic PCI host model. Our SimpleBoard SoC defines a GenericRiscvPciHost object that inherits from the base gem5 GenericPciHost object, and is defined and implemented in src/dev/riscv/pci_host.hh and src/dev/riscv/pci_host.cc, respectively.

The sole purpose of the custom PCI host is to map the correct source interrupt number from the PCI host to the PLIC. In our implementation, this is the base interrupt source number added to the PCI interrupt number. The base source number for the PCI host is 0x20, and thus the source numbers for all potential PCI interrupts are 0x21, 0x22, 0x23, and 0x24, although it is observed that only source number 0x21 is used.

10.2 UART

The UART module is incorporated into the RISC-V system using the Uart8250 model that comes with the vanilla gem5 release.

11 Future Work

In this section, we discuss the trajectory of this project and its short and long-term goals. Ideally we would like to simulate most of the features of a full-fledged RISC-V SoC so that system configurations ranging from embedded systems to high-performance computing systems can be verified and analyzed. The sections below outline some of the most crucial steps towards achieving this goal.

11.1 Formal Verification and Validation

While having a functionally correct simulation of RISC-V hardware is ideal, it is only useful if it can accurately represent real-world hardware as well. The next major milestone is to tune the latencies and models (including different CPU types) against a real RISC-V system. This will show that GXR5 can be used to accurately represent the performance of RISC-V systems, and thus can be used to conduct design-space exploration of new RISC-V systems.

11.2 MPSoC Support

Given that even the most basic embedded systems nowadays include multiple CPU cores, it is crucial that our RISC-V simulator be able to simulate multiple cores and their interactions, including parallel workloads, shared caches, etc.

12 References and Acknowledgements

12.1 Acknowledgements

This work has been partially supported by the ERC Consolidator Grant COMPUSAPIEN (GA No. 725657), the EC H2020 WiPLASH (GA No. 863337) and the RISC-AI project from HEIG-VD/HES-SO

We would also like to thank Professor Katzalin Olcoz of the Complutense University of Madrid for her advice and help in developing the virtual memory subsystem of this project, and general advice about kernel operations.

12.2 Links Reference

- GXR5 Homepage: https://www.epfl.ch/labs/esl/research/2d-3d-system-on-chip/gxr5
- Embedded Systems Laboratory (ESL) at EPFL: https://esl.epfl.ch/
- gem5-X Homepage: https://www.epfl.ch/labs/esl/open-source-software-projects/gem5-x/gem5-x-documentation/
- gem5: http://gem5.org/Main_Page
- OpenSBI: https://github.com/riscv/opensbi
- Linux: https://github.com/torvalds/linux
- buildroot: https://buildroot.org/
- Device Tree standard: https://www.devicetree.org/
- The RISC-V Foundation: https://riscv.org/
- The RISC-V GNU Toolchain: https://github.com/riscv/riscv-gnu-toolchain
- QEMU: https://www.qemu.org/
- spike: https://github.com/riscv/riscv-isa-sim

13 Glossary

13.1 Acronyms

Acronym	Term	Description
CLINT	Core-Local Interrupt Controller	Interrupt controller responsible for delegating timer and software interrupts to a local hart.
CSR	Control and State Register	RISC-V term for registers responsible for preserving system information and state.
DTB	Data Table Buffer	The L1 cache for data within the CPU.
hart	RISC-V Hardware Thread	RISC-V term hardware thread within a CPU.
ITB	Instruction Table Buffer	The L1 cache for instructions within the CPU.
MMU	Memory Management Unit	Hardware component that contains a TLB and page table walker.
PCI	Peripheral Control Interface	A standard meant to facilitate control of peripheral devices. A PCI system typically includes a host connected to a bus, connected to a series of PCI-compatible devices.
PLIC	Platform-Level Interrupt Controller	Interrupt controller responsible for delegating external interrupts to RISC-V harts.
RV64GC	RISC-V 64-bit General and Compressed extensions	RISC-V 64-bit-word-width machine with General and Compressed extensions. The General and Compressed extensions are the minimum extensions required to run a Linux-capable system.
PIO	Peripheral Input/Output	Usually used in reference to a PIO device, refers to external device such as a UART model.
Sv39	39-bit Supervisor Virtual Address	One of several RISC-V virtual addressing schemes specifying a 39-bit virtual address width.

Acronym	Term	Description
TLB	Translation Lookaside Buffer	Generic term for hardware memory cache, typically apart of the MMU.

13.2 Terms

Term	Description
Exception	A synchronous fault with respect to CPU cycles.
Fault	General term in RISC-V referring to various kinds of traps (e.g., page fault).
Interrupt	An asynchronous fault with respect to CPU cycles.

Bibliography

[1] K. Asanovic, D. A. Patterson, ”Instruction Sets Should Be Free: The Case For RISC-V”, in *Technical Report No. UCB/EECS-2014-146*, August 6, 2014. Accessed September 6, 2019 at https://people.eecs.berkeley.edu/ krste/papers/EECS-2014-146.pdf.

[2] F. Bellard, ”QEMU, a fast and portable dynamic translator.”, in *USENIX Annual Technical Conference*, 2005.

[3] A. Waterman and Y. Lee, ”Spike, a RISC-V ISA Simulator”, online. Accessed April 20, 2020 at https://github.com/riscv/riscv-isa-sim.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. “Chisel: constructing hardware in a scala embedded language”. in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE.*, 2012, pp. 1212– 1221.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, ”The gem5 simulator”, in *ACM SIGARCH Computer Architecture News*, Volume 39 Issue 2, May 2011.

[6] Y. M. Qureshi, W. A. Simon, M. Zapater, D. Atienza, K. Olcoz, ”Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms”, in *2019 Spring Simulation Conference (SpringSim)*, 29 April - 2 May 2019.

[7] A. Waterman, K. Asanovic, *et al.*, ”The RISC-V Instruction Set Manual Volume II: Privileged Architecture”, online, June 8 2019. Accessed September 6, 2019 at https://riscv.org/specifications/privileged-isa/.

[8] A. Waterman, K. Asanovic, *et al.*, ”The RISC-V Instruction Set Manual Volume I: Unprivileged ISA”, online, June 8, 2019. Accessed September 6, 2019 at https://riscv.org/specifications/.

[9] A. Waterman, K. Asanovic, *et al.*, ”The RISC-V Instruction Set Manual”, online, June 8 2019. Accessed September 6, 2019 at
https://github.com/riscv/riscv-isa-manual.

[10] R. Scheffel, ”Simulation of RISC-V based Systems in gem5”, Master’s thesis, TU Dresden, August 2018.

[11] ”Architectures/RISC-V”, web page. Accessed May 20, 2020 at https://fedoraproject.org/wiki/Architectures/RISC-V.

[12] ”Debian port information”, web page. Accessed May 20, 2020 at https://wiki.debian.org/RISC-V.

[13] P. Dabbelt, K. Cheng, J. Wilson, A. Waterman, *et al.*, ”RISC-V GNU Compiler Toolchain”, online, September 12 2014. Accessed May 20, 2020 at https://github.com/riscv/riscv-gnu-toolchain/graphs/contributors.

[14] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, ”McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures”, in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2009, pp. 469-480.

[15] A. Roelke and M. Stan, ”RISC5: Implementing the RISC-V ISA in gem5”, in *First Workshop on Computer Architecture Researc h with RISC-V (CARRV)*. 2017.

[16] T. Tuan, L. Cheng, and C. Batten, ”Simulating multi-core RISC-V systems in gem5.”, in *Workshop on Computer Architecture Research with RISC-V*. 2018.

[17] ”RISC-V Open aSource Supervisor Binary Interface (OpenSBI)”, online. Accessed May 20, 2020, at https://github.com/riscv/opensbi.

[18] ”Linux kernel”, online. Accessed May 20, 2020, at https://github.com/torvalds/linux.

[19] ”Buildroot: Making Embedded Linux Easy”, online. Accessed May 20, 2020, at https:// buildroot.org/.

[20] ”SiFive FU540-C000 Manual v1p0”, online. Accessed May 20, 2020, at https://static.dev.sifive.com/FU540-C000-v1.0.pdf.

[21] Sakalis, Christos, *et al.*, ”Splash-3: A properly synchronized benchmark suite for contemporary research.”, in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016.

[22] ”SPEC CPU2017”, online. Accessed May 20, 2020, at https://www.spec.org/cpu2017/.

[23] ”riscv-tests”, online. Accessed May 20, 2020, at https://github.com/riscv/riscv-tests.

[24] N. Asmussen, H. Härtig, and G. Fettweis, ”A Modular and Secure System Architecture for the IoT”, at the *3rd gem5 Users' Workshop*. June 2020.