Chapter 4

Architectural Simulation with gem5

Iago Caran Aquino (UNICAMP)
i198921@dac.unicamp.br
http://lattes.cnpq.br/4465472589884134
Lucas Wanner (UNICAMP)
wanner@unicamp.br
http://lattes.cnpq.br/4458409544983212
Sandro Rigo (UNICAMP)
srigo@unicamp.br
http://lattes.cnpq.br/8308517667746974

Abstract

This chapter introduces the fundamental concepts of architectural simulation and explores its critical role in modern computer architecture. It highlights how simulation enables designers to explore, verify, and optimize processor architectures by modeling their behavior and interaction with key system components, such as memory hierarchies and accelerators. The chapter illustrates how gem5's flexibility can be used to easily evaluate different architectural configurations, helping designers make informed decisions and avoid costly physical prototyping. It also discusses the challenges of simulation in increasingly complex systems, strategies for balancing detail and abstraction, and how gem5 can help explore this ample design space.

A case study involving the RISC-V architecture demonstrates how new instructions, particularly matrix-based operations, can be integrated into an existing gem5 model. Matrix multiplication, a key operation in AI workloads, is a practical example of expanding the RISC-V instruction set. The case study illustrates the addition of matrix load, store, and multiply-and-accumulate instructions in gem5, providing readers with hands-on experience in extending architectural models.

Resumo

Este capítulo introduz os conceitos fundamentais de simulação arquitetural e explora seu papel crítico na arquitetura de computadores moderna. Ele destaca como a simulação permite que os projetistas explorem, verifiquem e otimizem arquiteturas de processadores, modelando seu comportamento e interação com componentes chave do sistema como hierarquias de memória e aceleradores. O capítulo ilustra como a flexibilidade do gem5 pode ser usado para avaliar facilmente diferentes configurações arquiteturais, ajudando os projetistas a tomar decisões informadas e evitar a construção de protótipos físicos caros. Ele também discute os desafios da simulação de sistemas cada vez mais complexos, estratégias para equilibrar detalhes e abstração e como 0 gem5 pode ajudar a explorar esse vasto espaço de projeto.

Um estudo de caso envolvendo a arquitetura RISC-V demonstra como novas instruções, particularmente operações baseadas em matrizes, podem ser integradas a um modelo existente no gem5. A multiplicação de matrizes, uma operação essencial em cargas de trabalho de IA, é um exemplo prático de expansão do conjunto de instruções do RISC-V. O estudo de caso ilustra a adição de instruções de carga, armazenamento e multiplicação-acumulação de matrizes no gem5, proporcionando aos leitores uma experiência prática em estender modelos arquiteturais.

4.1. Introduction

Simulation is an indispensable tool across all fields of science, enabling researchers to study and reason about complex systems that would otherwise be too difficult, costly, or impossible to explore through direct experimentation. Whether simulating the behavior of subatomic particles, modeling climate systems, or evaluating the performance of cutting-edge computer architectures, simulations allow scientists to create controlled virtual environments to test hypotheses, analyze interactions, and predict outcomes.

In this chapter, we explore the idea of architectural simulation, which is essential in modern computer architecture because it allows designers to explore, verify, and optimize their designs. Architectural simulation refers to modeling and emulating a processor's behavior and its interaction with key components of a computer system. This includes simulating not just the processor's execution of instructions but also how it communicates with and manages caches, memory hierarchies, buses, and specialized hardware accelerators. For the sake of simplicity, we will refer to architectural simulation as just simulation from now on.

As computing systems become more complex, the importance of simulation grows. Designing hardware involves significant trade-offs between performance, power consumption, and cost. Due to time and financial constraints, physical prototyping for every potential design iteration is impractical. This is where simulation comes into play, offering several benefits:

1. **Design Space Exploration:** Simulation enables the evaluation of multiple design configurations, such as different processor architectures or cache sizes, to determine their impact on performance. This accelerates innovation and enables more informed decision-making early in the design process.
2. **Performance Prediction:** With accurate models, architectural simulations can predict how a system will behave under different workloads, helping designers optimize hardware for specific use cases or applications.
3. **Verification and Validation:** Simulating an architecture allows for early detection of functional errors and design flaws, reducing the risk of costly mistakes during manufacturing. Simulation can serve as a platform for functional validation.
4. **Cost Reduction:** By identifying issues or bottlenecks early, simulation reduces the need for multiple physical prototypes, leading to significant cost savings.

While architectural simulation offers many advantages, it also presents several challenges, particularly when scaling simulations for modern, increasingly complex architectures. Simulation models must balance detail and abstraction:

1. **Detailed Modeling:** In detailed models (e.g., cycle-accurate simulations and RTL), every aspect of the hardware is modeled as accurately as possible, down to the timing of individual components like ALUs or memory controllers. This level of detail ensures high accuracy in predicting performance but drastically increases simulation time and computational resource requirements.

2. **Abstraction in Functional Simulation:** Functional simulations, on the other hand, abstract away the timing details and focus solely on whether the hardware can correctly execute instructions. This abstraction makes functional simulation much faster, which is critical for exploring architectural designs in the early stages. However, these simulations are less suitable for performance analysis, as they don't capture how long operations take, limiting their ability to model real-world execution times.

3. **Hybrid Approaches:** Many modern simulators, such as gem5, offer hybrid approaches where parts of the simulation can be more detailed (for critical components) while others are abstracted. This balance between functional accuracy and timing detail allows for more practical simulations while maintaining reasonable execution times.

4. **Parameterization and Configurability:** Architectural simulators often rely on parameterization to handle the complexity of modern architectures. By adjusting parameters like cache size, memory latency, or core counts, designers can explore various configurations without creating entirely new models for each change. This enables quick experimentation with different design choices.

4.1.1. RISC-V

RISC-V is an open-source, reduced instruction set computing (RISC) architecture that has gained significant traction recently due to its flexibility, scalability, and lack of licensing fees. Unlike traditional proprietary instruction set architectures (ISAs) such as x86 or ARM, RISC-V is designed to be freely available, allowing researchers, developers, and companies to customize it for their own needs without the constraints of intellectual property restrictions. This makes RISC-V particularly appealing for innovation in both academia and industry, as it can be adapted for a wide range of applications, from embedded systems to high-performance computing and AI. Its modular design, with a minimal base instruction set and optional extensions, provides a strong foundation for building general-purpose processors and specialized accelerators.

Simulation plays a critical role in the development of new architectures like RISC-V. Providing a virtual environment for design space exploration accelerates innovation by enabling rapid iteration and evaluation of performance under various workloads, even before physical processors reach the market.

4.1.2. Our Case Study

To make things more interesting, we will introduce gem5 using a case study architecture. The goal is to start with an existing gem5 RISC-V simulator and discuss how to expand it to simulate new instructions.

Matrix multiplication is a cornerstone operation in many artificial intelligence (AI) workloads, particularly Deep Learning. It forms the backbone of neural network computations, where it is used extensively in the forward and backward propagation of data through the network. This operation is vital for the training and inferencing phases of

neural networks, where the multiplication of weight matrices with input data matrices—or subsequent layer outputs—generates the activations that move through the network.

Following this trend, several popular processor architectures are introducing new Matrix Extensions to their ISAs. RISC-V is no exception, although the specification for this extension is still in its discussion phase.

Our case study will introduce new matrix-based instructions in an existing RISC-V gem5 model. Figure 4.1 illustrates the architecture of an Attached In-core Matrix Engine (AIME). The AIME resides inside the existing core, introducing new matrix-specialized registers. The core decodes and dispatches the new instructions to the matrix processing units (MPU). For example, we will introduce matrix-based versions of the load, store, and multiply-and-accumulate (mac) instructions.

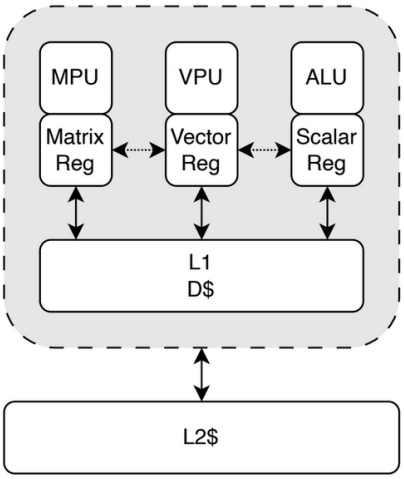

Figure 4.1: An Attached In-core Matrix Accelerator Architecture

4.1.3. Text Organization

Section 4.2 presents a high-level view of the gem5 simulation framework. Section 4.3 includes instructions on setting up gem5 and running basic RISC-V code on the simulator. Section 4.4. Section 4.4 introduces gem5 configuration scripts. Section 4.5 presents a case study wherein the RISC-V model in gem5 is extended to support new instructions for optimized matrix multiplication. Finally, Section 4.6 introduces strategies for performance analysis with gem5.

4.2. The gem5 Simulation Framework

The gem5 simulation framework[Binkert et al. 2011, Lowe-Power et al. 2020] is a widely-used, modular simulation framework designed for computer architecture research and development. It supports a broad range of instruction set architectures (ISAs), including x86, ARM, and RISC-V, and provides flexible simulation modes for both functional and cycle-accurate (timing) simulations. With gem5, users can model complex systems that span from single-core processors to full-system simulations with multicore architectures, memory hierarchies, and peripherals. It offers extensive configurability, making it ideal

for exploring design choices and conducting performance analysis under various scenarios.

gem5 originates in a fusion between two well-established simulators: M5 and GEMS. The M5 simulator[Binkert et al. 2006], developed at the University of Michigan, was known for its focus on full-system simulation, providing detailed models of CPUs and memory systems. GEMS[Martin et al. 2005], developed at the University of Wisconsin-Madison, offered advanced memory system modeling and support for parallel simulations. In 2011, these two simulators were combined to form gem5, bringing together both strengths. Since then, gem5 has evolved significantly, with ongoing contributions from a global community of researchers and developers. The gem5 simulator is released under a BSD (Berkeley Software Distribution) 3-clause license.

One of gem5's standout capabilities is its dual-mode simulation: full-system (FS mode) and syscall emulation (SE mode). In FS mode, gem5 simulates the complete hardware environment in detail, enabling the execution of unmodified operating systems. This comprehensive approach allows for intricate interactions between the application and the underlying system, capturing low-level operational details. Conversely, in SE mode, gem5 focuses solely on running the application as a user-level process, with the simulator intercepting and responding to system calls. This mode reduces the granularity of the simulation, thereby accelerating execution times. SE mode is particularly beneficial when the primary objective is to analyze the application's interaction with the memory hierarchy.

4.2.1. The gem5 System Architecture Model

In gem5, the design of the CPU model exemplifies a clear delineation between the microarchitecture and the instruction set architecture (ISA). This distinction is implemented through two primary classes: Static Instruction and Execution Context. The Static Instruction class and its derivatives articulate the behavior of individual instructions with methods that detail their interaction with various stages of the CPU pipeline. Conversely, the Execution Context class manages all alterations to the CPU's state, storing essential information such as register values, program counters, and memory mappings. It further facilitates access to memory ports and other architectural features offered by the CPU. This architectural separation enables the association of various CPU models with different ISAs (e.g., RISC-V, ARM, x86) without necessitating specific adaptations for each configuration. Figure 4.2, adapted from documentation [gem5 2022, Black et al. 2010] illustrates the components of the gem5 CPU Model, indicating which elements depend on the ISA and which do not. gem5 includes five processor models:

- **AtomicSimpleCPU:** a purely functional, in-order model that is suited for cases where a detailed model is not necessary, like warm-up periods, client systems that are driving a host, or just testing to make sure a program works. It uses atomic memory accesses;
- **TimingSimpleCPU:** similar to the AtomicSimpleCPU, but uses timing memory accesses and stalls on cache accesses, waiting for the memory system to respond before proceeding;

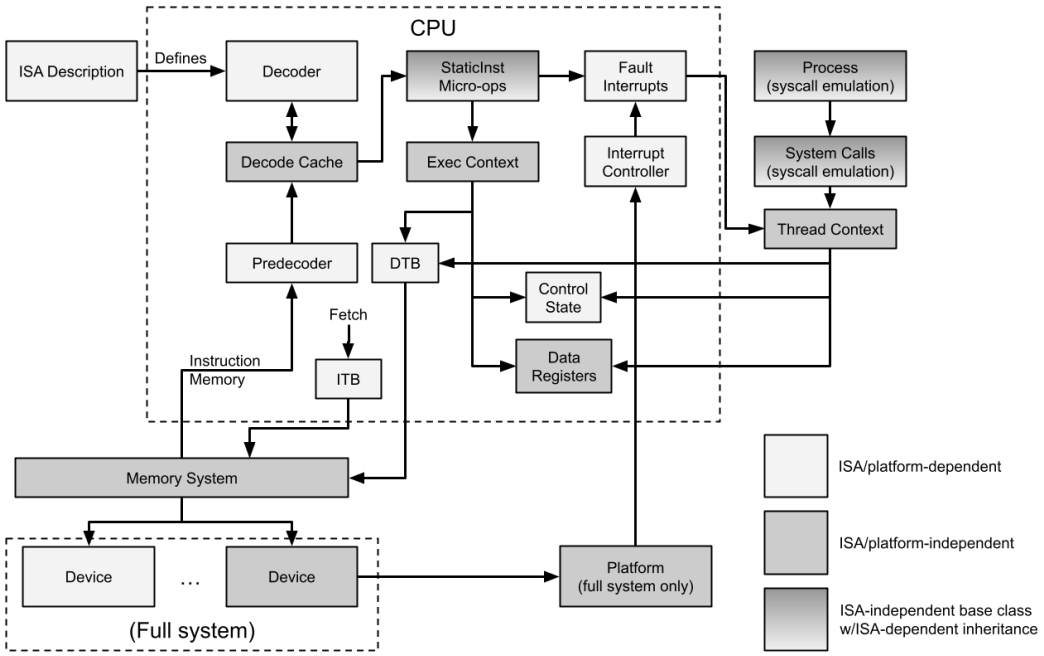

Figure 4.2: gem5's architecture diagram adapted from[gem5 2022, Black et al. 2010].

- **MinorCPU:** is an in-order processor model with a fixed pipeline but configurable data structures and execution behavior designed to model processors that enforce strict in-order execution.;

- **O3CPU:** is a detailed, timing-accurate out-of-order CPU model, loosely based on the Alpha 21264, which executes instructions at the execute stage of the pipeline, unlike most other simulators;

- **TraceCPU:** is a model that replays dependency- and timing-annotated traces from the O3 model, allowing for faster yet reasonably accurate memory-system performance exploration.

4.2.2. gem5 standard library

The **gem5 standard library (stdlib)** is a set of components that allows users to create systems quickly. These components act as reusable and modular blocks that can be connected to simplify the process of constructing custom systems, they range from memory and cpu models to complete cache hierarchies and systems. These components will be extended while developing your custom configurations using object-oriented semantics.

Let's say we want to implement a cache model for our system. We can extend the Cache class to express our L1 cache configuration, as shown in Listing 4.1, and then extend this L1 class to differentiate the data cache from the instruction cache.

```
class L1Cache(Cache):
    assoc = 2
    tag_latency = 2
    data_latency = 2
    response_latency = 2
    mshrs = 4
    tgts_per_mshr = 20

    def connectCPU(self, cpu):
        raise NotImplementedError

    def connectBus(self, bus):
        self.mem_side = bus.cpu_side_ports


class L1ICache(L1Cache):
    size = '16kB'

    def connectCPU(self, cpu):
        self.cpu_side = cpu.icache_port


class L1DCache(L1Cache):
    size = '64kB'

    def connectCPU(self, cpu):
        self.cpu_side = cpu.dcache_port
```

Listing 4.1: Example of new L1 Cache component

4.3. Getting started with gem5

The gem5 Getting Started Guide [Gem5 Project 2024] includes detailed instructions on setting up, building, and running basic examples. Here, we provide a summarized outline of this process, which includes setting up required packages, installing a RISC-V compiler toolchain, downloading and building gem5, and running a simple example using preconfigured systems. These steps were tested for an Ubuntu 22.04.5 LTS Linux distribution, both natively and in a Docker container. Reference Dockerfiles for the toolchain and gem5 are available at https://github.com/LSC-Unicamp/gem5-sscad-2024.

4.3.1. Required Packages

As illustrated in Listing 4.2 required packages for the GNU toolchain for RISC-V (line 1) and gem5 (line 2) may be installed in Ubuntu using `apt`.

```
1 sudo apt install -y autoconf automake autotools-dev curl python3 python3-pip libmpc-dev
      libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
      patchutils bc zlib1g-dev libexpat-dev git
2 sudo apt install -y build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev
      protobuf-compiler libprotoc-dev libgoogle-perftools-dev python-is-python3
```

Listing 4.2: APT command for installing requires packages for the RISC-V GNU Toolchain (line 1) and gem5 (line 2)

4.3.2. GNU Toolchain for RISC-V

To compile RISC-V code on platforms that don't natively support this architecture, we use the GNU RISC-V toolchain, a collection of compilers, assemblers, and related tools

tailored for the RISC-V instruction set architecture (ISA). This setup allows for cross-compilation, where code is compiled on one platform (e.g., an x86 machine) to run on a different target platform, in our case a RISC-V processor.

The build process for the GNU RISC-V toolchain is illustrated in Listing 4.3. The tools are installed under `/opt/riscv` and must be added to the system path. After building the toolchain, two common targets are available: ELF (Executable and Linkable Format) and Linux. ELF is typically used for bare-metal applications, running directly on hardware without an operating system, while the Linux target generates binaries designed to run on systems with glibc and a Linux OS.

```
export RISCV=/opt/riscv
export PATH=$PATH:$RISCV/bin

git clone --recursive https://github.com/riscv/riscv-gnu-toolchain -b 2024.04.12
cd riscv-gnu-toolchain
./configure --prefix=$RISCV --enable-multilib
make -j8
make linux -j8
cd ..
rm -rf riscv-gnu-toolchain
```

Listing 4.3: Build steps for the GNU RISC-V toolchain

Once the toolchain is installed, code can be compiled using gcc with the O3 optimization flag, as illustrated in Listing 4.4

```
riscv64-unknown-elf-gcc -O3 code.c -o code.riscv
```

Listing 4.4: Compiling with GCC

The instructions of the generated binary can be inspected using the `objdump` tool, which disassembles the binary back into assembly language, as shown in Listing 4.5:

```
riscv64-unknown-elf-objdump -S code.riscv | less
...
0000000000010104 <main>:
10104:       6549                    lui     a0,0x12
10106:       1141                    addi    sp,sp,-16
10108:       5c850513                addi    a0,a0,1480 # 125c8 <__errno+0x8>
1010c:       e406                    sd      ra,8(sp)
1010e:       418000ef                jal     10526 <puts>
10112:       60a2                    ld      ra,8(sp)
10114:       4501                    li      a0,0
10116:       0141                    addi    sp,sp,16
10118:       8082                    ret
...
```

Listing 4.5: Inspecting the Binary with objdump

For applications not requiring extensive OS support, gem5 can be used to run the resulting binary in SE mode, simulating the RISC-V architecture without an operating system.

4.3.3. Building and Running gem5

Gem5 is compiled using the scons build system, which is a Python-based build tool. To build Gem5 for the RISC-V architecture, the process starts by cloning the repository, followed by navigating to the Gem5 directory, and then compiling for RISC-V. The build

process can be accelerated by utilizing all available CPU cores through the use of the -j option with the number of processing units, as shown in Listing 4.6.

```
git clone https://github.com/gem5/gem5
cd gem5
scons build/RISCV/gem5.opt -j`nproc`
```

Listing 4.6: Building Gem5

4.3.4. Using gem5 default configuration scripts

A collection of configuration scripts is available within the `configs` directory covering different features of the framework, such as checkpoints, full system simulation, different CPU models, among others. We recommend exploring the examples there.

Taking `configs/learning_gem5/part1/simple-riscv.py` as an example, we have a basic system using the RISC-V ISA with the TimingSimple CPU model running a program that prints "Hello World!", the complete output of the simulation is exemplified by the output in Listing 4.7.

```
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 24.0.0.1
gem5 compiled Oct  5 2024 20:32:53
gem5 started Oct  5 2024 23:35:42
gem5 executing on desktop, pid 41833
command line: ./gem5/build/RISCV/gem5.opt
    ./gem5/configs/learning_gem5/part1/simple-riscv.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
src/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match
    the address range assigned (512 Mbytes)
src/arch/riscv/isa.cc:276: info: RVV enabled, VLEN = 256 bits, ELEN = 64 bits
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat is a
    stat that does not belong to any statistics::Group. Legacy stat is deprecated.
system.remote_gdb: Listening for connections on port 7000
Beginning simulation!
src/sim/simulate.cc:199: info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 499797000 because exiting with last active thread context
```

Listing 4.7: Basic hello world running in gem5

4.4. gem5 configuration

Gem5's flexibility comes from the Python interface for describing systems, which allows multiple configurations to be executed without recompiling the entire framework. The basic components of such a script describe various aspects of the system, such as CPU type, memory hierarchy, cache configuration, and simulation parameters. The script specifies how to wire together different system components and how the simulation should be executed.

Expanding on the example presented by the `simple-riscv.py` script, we will add a simple L1 cache and read the binary to be executed from the command line argument. Listing 4.8 shows the modified script. From here, we could, for example, add an

```python
from m5.objects import *

# Define system components
system = System()
system.clk_domain = SrcClockDomain(clock='1GHz', voltage_domain=VoltageDomain())

# Set up the CPU
system.cpu = RiscvTimingSimpleCPU()

# Set up the memory system
system.mem_mode = 'timing'  # Use timing model for memory accesses
system.mem_ranges = [AddrRange('512MB')]
system.membus = SystemXBar()
system.system_port = system.membus.cpu_side_ports

# Configure L1 cache
system.cpu.icache = Cache(size='32kB', assoc=2)
system.cpu.dcache = Cache(size='32kB', assoc=2)
system.cpu.icache.cpu_side = system.cpu.icache_port
system.cpu.dcache.cpu_side = system.cpu.dcache_port
system.cpu.icache.mem_side = system.membus.cpu_side_ports
system.cpu.dcache.mem_side = system.membus.cpu_side_ports

# Set up memory controller
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports

# Load binary
thispath = os.path.dirname(os.path.realpath(__file__))
binary = os.path.join(
    thispath,
    sys.argv[1],
)
system.workload = SEWorkload.init_compatible(binary)

# Run simulation
process = Process()
process.cmd = [binary] + sys.argv[2:]
system.cpu.workload = process
system.cpu.createThreads()

root = Root(full_system=False, system=system)
m5.instantiate()

print(f"Beginning simulation!")
exit_event = m5.simulate()
print(f"Exiting @ tick {m5.curTick()} because {exit_event.getCause()}")
```

Listing 4.8: A Simple RISC-V System Configuration

L2 cache just by instantiating another Cache object and connecting it between the L1 and the memory bus.

4.4.1. Full System simulation with gem5-resources

Beyond the `stdlib` components, there is gem5-resources, which is a repository and a framework designed to provide a collection of pre-built resources, such as disk images, binaries, benchmarks, and workloads, known and proven compatible with the gem5 architecture simulator.

One of the uses of this repository is to easily simulate full systems since it eliminates the need to prepare an image with the kernel and operating system for execution. For example, to run a test with Ubuntu 20.04, we set the workload to `workload = obtain\_resource("riscv-ubuntu-20.04-boot", resource\_version="3.0.0")` and the image with the system will be downloaded automatically.

The `riscvmatched-fs.py` file in `configs/example/gem5_library/` illustrates the complete configuration for this workload using a prebuilt system. The details of configuring a script for Full System simulation are beyond our scope, but there are many more details to be configured, such as I/O devices and device tree generation.

4.5. Expanding gem5

Extending the ISA (Instruction Set Architecture) is a fundamental process in processor development, especially when adapting a general-purpose architecture like RISC-V for specific workloads, such as matrix operations. This section focuses on adding new instructions to optimize matrix multiplication, a key operation in many high-performance and AI-related applications, as detailed in Section 4.1.2.

4.5.1. The ISA Extension

Instruction specification is critical to processor architecture development, as it defines how a processor communicates with software and executes tasks. An instruction's behavior determines its function and must be clearly defined to ensure the processor implementation correctly interprets and executes each one as intended.

RISC-V's Matrix Specification is still in the discussion stages, meaning it can change a lot. A matrix multiplication, for example, can be performed by considering each line and column as vectors or by working on small blocks of the full matrix. Both approaches have different design considerations that impact the overall performance and efficiency of the system.

One of the key challenges in adding new instructions to RISC-V is the strict limitation on instruction sizes and opcode availability. The base RISC-V ISA supports fixed 32-bit instructions, so we must carefully pack essential information such as operation type, data type, operands, and other metadata into a limited space. This constraint impacts the immediate design of the instruction and could limit future extensibility if not handled carefully. Efficiently encoding these instructions while preserving flexibility is crucial for balancing the immediate needs of matrix operations with the long-term goals of keeping the ISA lightweight and adaptable.

Mnemonic	Format	Behavior
mls	load/store	
mss	load/store	
mmacf	arithmetic	multiply and accumulate
maddf	arithmetic	add matrices
msubf	arithmetic	subtract matrices
mm_xmu	reg-reg mov	fill matrix with scalar value
mzero	reg-reg mov	move zero to the entire destination register

Table 4.1: Matrix Extension to the RISC-V ISA

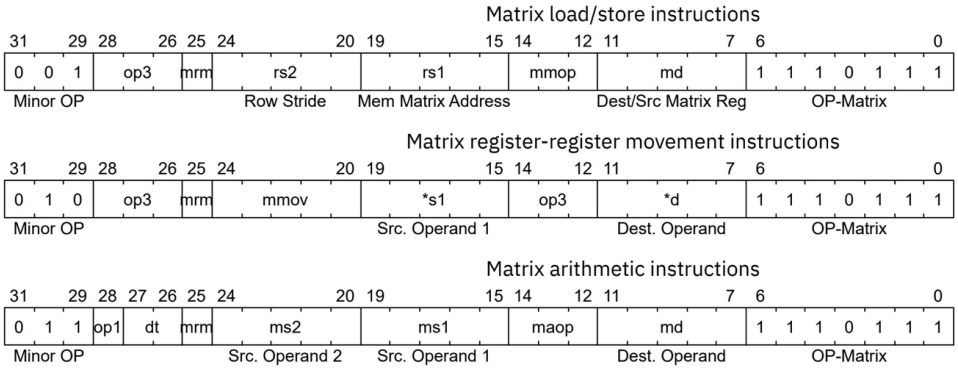

Figure 4.3: Instruction encodings for the Matrix Extension.

Since the final matrix specification for RISC-V has not yet been published, the already-established vector extension has inspired our design choices. The vector extension efficiently handles data parallelism, and its flexible encoding allows for operations on various data types and sizes. Drawing from these principles, we have drafted a preliminary set of instructions to handle matrix operations efficiently. These instructions leverage matrix-specific registers and provide the essential functionality for matrix-based software, including load/store operations and arithmetic functions such as addition, subtraction, and multiplication.

The following instructions highlight the key operations that form the basis of this extension. These instructions, detailed in Figure 4.3 and summarized in Table 4.1, represent a first best estimate of the necessary operations to support matrix computations. These include basic arithmetic instructions, data movement between matrix and scalar registers, and memory operations. For the sake of simplicity, in our example, the instructions are fixed to 32-bit elements and a fixed four-by-four panel. This fixed format was chosen to streamline early development and debugging. Later development phases can introduce more sophisticated encoding and support for different data types. Although the exact encoding may evolve as the specification matures, these examples illustrate the core functionality we aim to integrate into the RISC-V model.

4.5.2. How to Include a New Instruction

This section discusses how we add the new matrix registers to our gem5 model and then advances to including the new instructions in the decoder.

4.5.2.1. Adding new registers

In this example, we are adding new registers that hold panels of our matrix. These panels will be 4x4. We do not have a ready-to-use type for this shape, so we are borrowing the *MatStore* container from ARM's implementation to use as our register datatype. For simplicity, we took only the essential part of it. Following the same template as the other register types, we will create a file inside the *regs* folder to define the matrix registers. Note that we are using the MatStore class that was described before. We have 32 matrix registers of 256 bits named m0– m31.

```
namespace gem5 {
namespace RiscvISA {
constexpr unsigned MaxMLenInBits = 256;
constexpr unsigned MaxMLenInBytes = MaxMLenInBits >> 3;

// Import the MatStore type that will be our register type
using MatRegContainer = gem5::MatStore;
using mreg_t = MatRegContainer;

// Define the number of registers this architecture uses
const int NumMatrixStandardRegs = 32;
const int NumMatrixInternalRegs = 0;
const int NumMatrixRegs = NumMatrixStandardRegs + NumMatrixInternalRegs;

// Define the register names
const std::vector<std::string> MatrixRegNames = {
     "m0",   "m1",   "m2",   "m3",   "m4",   "m5",   "m6",   "m7",
     "m8",   "m9",  "m10",  "m11",  "m12",  "m13",  "m14",  "m16",
    "m17",  "m18",  "m19",  "m20",  "m21",  "m22",  "m23",  "m24",
    "m25",  "m26",  "m27",  "m28",  "m29",  "m30",  "m31",  "m32"
};

static inline TypedRegClassOps<RiscvISA::MatRegContainer> matRegClassOps;
inline constexpr RegClass matRegClass =
    RegClass(MatRegClass, MatRegClassName, NumMatrixRegs, debug::MatRegs).
        ops(matRegClassOps).
        regType<MatRegContainer>();
} // namespace RiscvISA
} // namespace gem5
```

Listing 4.9: Extract of the register definition.

The matrix register header file must be with `#include "arch/riscv/regs/matrix.hh"` added to the following files located under `gem5/src/arch/riscv/`: `utility.hh`, `pcstate.hh`, `isa/includes.isa` and `isa.cc`.

In the `isa.cc` file, we also need to comment on the following line to allow the ISA to include the new registers we defined in the matrix header:

```
RegClass matRegClass(MatRegClass, MatRegClassName, 0, debug::MatRegs);
```

We can now rebuild gem5. However, nothing will apparently change; we need to add instructions to start using these registers.

4.5.2.2. Expanding the decoder

Following the encodings we defined in Figure 4.3, we will expand the ISA description. In the *decoder.isa* file, we will first find a suitable place for our new extension. From the RISC-V ISA specification [Waterman et al. 2014], we found that the reserved OPCODE 1110111 is not in use yet, thus we will be using it for our matrix extension. The decoder structure is similar to multiple nested switch clauses, each defining a bitfield that will be used to index the instruction, so we will first decide in which QUADRANT our instruction is, 0b11 or 0x3 for 32b instructions. Then, we are addressing the 5-bit opcode, 0b11101 or 0x1D. With this, we can finally add our new instructions as illustrated by the structure in Listing 4.10.

```
decode QUADRANT default Unknown::unknown() {
    ...
    0x3: decode OPCODE5 {
        ...
        0x1D: // Our new instructions here
        ...
    }
    ...
}
```

Listing 4.10: Extract of the ISA definition.

Our operations will be decoded by the *Minor OP* field, bits 31 to 29, which we will be adding to the bitfield list file, as any other entry does not define it. We will also add the bitfield definitions for the source and destination matrix registers used by the instructions, as illustrated by Listing 4.11. We will not add the other fields described in the encoding because they map details we will not address in this chapter. The caption of the following Listings will show the path in our repository where the reader can find the file being modified.

```
// Matrix instructions
def bitfield MMINOP     <31:29>;

def bitfield MRD        mrd;
def bitfield MRS1       mrs1;
def bitfield MRS2       mrs2;
def bitfield MRS3       mrs3;
```

Listing 4.11: gem5/src/arch/isa_parser/isa/bitfields.isa

The registers don't define the bit ranges. They reference the definitions made in the *types.hh* file as shown in 4.12.

```
// matrix
Bitfield<11,  7>    mrd;
Bitfield<19, 15>    mrs1;
Bitfield<24, 20>    mrs2;
Bitfield<11,  7>    mrs3;
```

Listing 4.12: gem5/src/arch/riscv/types.hh

For the registers to be usable in our instructions, we must add them to the *operands.isa* file as shown in Listing 4.13. This tells gem5 that there is some information encoded in the instruction that the instruction behavior and the expected type will use.

```
'Mrd':  MatRegOp('mc', 'MRD',  'IsMatrix', 1),
'Mrs1': MatRegOp('mc', 'MRS1', 'IsMatrix', 2),
'Mrs2': MatRegOp('mc', 'MRS2', 'IsMatrix', 3),
'Mrs3': MatRegOp('mc', 'MRS3', 'IsMatrix', 4),
```

Listing 4.13: gem5/src/arch/riscv/isa/operands.isa

Now, we can specify our instructions in the decoder file as shown in Listing 4.14. For memory and arithmetic, we also use the *FUNCT3* field to differentiate between load and store operations or multiply and accumulate, addition, or subtraction operations.

```
...
0x1d: decode MMINOP {
    0x1: decode FUNCT3 {
        0x4: MatrixLoadOp::mls({{
            for (int k = 0; k < 4; k++) {
                Mrd_uw[i + k] = Mem_mc.as<uint32_t>()[k];
            }
        }});
        0x5: MatrixStoreOp::mss({{
            for (int k = 0; k < 4; k++) {
                Mem_mc.as<uint32_t>()[k] = Mrd_uw[i + k];
            }
        }});
    }
    0x2: ScalarMatrixMoveOp::mmv_xmu({{
        Mrd_uw[i] = Rs1_uw;
    }});
    0x3: decode FUNCT3 {
        0x0: MatrixArithLineOp::mmacf({{
            uint32_t row = i / 4;
            uint32_t col = i % 4;
            uint32_t row_stride = 4;

            float acc = Mrs3_mc->as<float>()[i];

            for (int j = 0; j < 4; j++) {
                acc += Mrs1_mc->as<float>()[row * row_stride + j] *
                    Mrs2_mc->as<float>()[col * row_stride + j];
            }

            Mrd_sf[i] = acc;
        }});
        0x1: MatrixArithOp::maddf({{
            Mrd_sf[i] = Mrs1_sf[i] + Mrs2_sf[i];
        }});
        0x2: MatrixArithOp::msubf({{
            Mrd_sf[i] = Mrs1_sf[i] - Mrs2_sf[i];
        }});
    }
}
...
```

Listing 4.14: Extract of the riscv/isa/decoder.isa definition.

Each instruction derives from one template that we will be declaring and a name. There is also the specification of the instruction behavior that will be composed inside

the template. A script named isa_parser aggregates and expands all the *.*isa* files into the decoder *.*cpp* files.

Note that each register reference is followed by a suffix defined in the `src/arch/riscv/isa/operands.py` file. These are used to indicate which data type should be used to interpret the contents of the registers. Since we added a new register type, we must add the code in Listing 4.15 to the operand_types list in *operands.py*.

```
def operand_types {{
    ...
    'vc'    : 'RiscvISA::VecRegContainer',
    'mc'    : 'RiscvISA::MatRegContainer'
}};
```

Listing 4.15: Extract of the riscv/isa/operands.py file

We still need to define the instruction formats, i.e., the templates that isa_parser uses to expand the instruction definitions. Taking the *mls* as an example, we have the *MatrixLoadOp* format, which is defined in the `formats/matrix.isa` file that we are going to create.

In the format definition exemplified in Listing 4.16, we specify this operation's default parameters and base template, which will generate a tuple with all the necessary parts to compose one instruction. The base template used for the instruction defines which templates are used for each one of these parts, which are listed below:

- **Header output:** used to define the header file declaring the instruction class. Here, we declare the number of source and destination registers and the functions of the instruction in the pipeline. Most will have only an execute method, but memory instructions may have initiateAcc and completeAcc methods;
- **Decoder output:** defines a constructor for the instruction. This is where the necessary registers are specified;
- **Decode block:** defines the decoder entry, which is how a new object of the class should be created;
- **Exec output:** specifies the implementation of the instruction, manipulating the registers and ExecContext.

The base template shows one important characteristic of gem5: the division of instructions into macro-ops and micro-ops. The macro ops are used by the frontend portion of the simulation, aggregating the necessary information for the instruction and advancing in the pipeline as necessary. The micro-ops are used on the backend portion, allowing complex operations to be broken up into fine-grained operations, allowing for more simulation detail.

Listing 4.17 illustrates the templates used to compose the format. They reflect the parts mentioned before and illustrate how we define the number of registers the instruction will use in the Declare portion, the behavior of the instruction in the Execute portion, and

```
def MMemBase(name, Name, ea_code, code, base_class, mem_flags, inst_flags,
             declare_template_base=MatrixMacroDeclare, decode_template=MatrixDecode,
             exec_template_base='', macro_construtor=MatrixMacroConstructor,
             op_class='', is_macroop=True):
    iop = InstObjParams(name, Name, base_class, { 'code': code, 'ea_code': ea_code,
        'op_class': op_class },
          inst_flags)
    header_output   = declare_template_base.subst(iop)
    decoder_output  = macro_construtor.subst(iop)
    decode_block    = decode_template.subst(iop)
    exec_output     = ''

    if not is_macroop:
        return (header_output, decoder_output, decode_block, exec_output)

    micro_class_name = exec_template_base + 'MicroInst'
    microiop = InstObjParams(name + '_micro',
        Name + 'Micro', exec_template_base + 'MicroInst', {'ea_code': ea_code, 'code':
            code},
        inst_flags)

    microDeclTemplate = eval(exec_template_base + 'MicroDeclare')
    microConsTemplate = eval(exec_template_base + 'MicroConstructor')
    microExecTemplate = eval(exec_template_base + 'MicroExecute')
    microInitTemplate = eval(exec_template_base + 'MicroInitiateAcc')
    microCompTemplate = eval(exec_template_base + 'MicroCompleteAcc')
    header_output = microDeclTemplate.subst(microiop) + header_output
    decoder_output = microConsTemplate.subst(microiop) + decoder_output
    micro_exec_output = (microExecTemplate.subst(microiop) +
        microInitTemplate.subst(microiop) +
        microCompTemplate.subst(microiop))
    exec_output += micro_exec_output

    return (header_output, decoder_output, decode_block, exec_output)

def format MatrixLoadOp(
    code, ea_code={{ EA = Rs1 + ((uint64_t) microIdx / 4) * Rs2 + (microIdx % 4) * 4;
        }},
    mem_flags=[], *flags ) {{
    (header_output, decoder_output, decode_block, exec_output) = \
        MMemBase(name, Name, ea_code, code, 'MatrixLoadMacroInst',
                 mem_flags, flags, op_class='MemReadOp',
                 macro_construtor=MatrixMacroConstructor,
                 exec_template_base='MatrixLoad')
}};
```

Listing 4.16: Extract of the matrix format specification

```
def template MatrixLoadMicroDeclare {{
    class %(class_name)s : public %(base_class)s
    {
      private:
        RegId srcRegIdxArr[2];
        RegId destRegIdxArr[1];
      public:
        %(class_name)s(ExtMachInst _machInst);
        %(class_name)s(ExtMachInst _machInst, uint32_t _microIdx);

        Fault execute(ExecContext *, trace::InstRecord *) const override;
        Fault initiateAcc(ExecContext *, trace::InstRecord *) const override;
        Fault completeAcc(PacketPtr, ExecContext *,
                          trace::InstRecord *) const override;

        using %(base_class)s::generateDisassembly;
    }; }};

def template MatrixLoadMicroExecute {{
    Fault
    %(class_name)s::execute(ExecContext *xc,
        trace::InstRecord *traceData) const
    {
        Addr EA;
        %(op_decl)s;
        %(op_rd)s;
        uint32_t i = microIdx;
        for (int row; row < 4; row++) {
            EA = Rs1 + ((uint64_t) i / 4) * Rs2 + (i % 4) * 4;
            const size_t mem_size = 16;
            std::vector<bool> byte_enable(mem_size, true);
            Fault fault = xc->readMem(EA, (uint8_t*) &Mem, mem_size,
                                      memAccessFlags, byte_enable);
            if (fault != NoFault)
                return fault;
            %(code)s;
            i += 4;
        }
        %(op_wb)s;
        return NoFault;
    } }};

def template MatrixLoadMicroConstructor {{
%(class_name)s::%(class_name)s(ExtMachInst _machInst, uint32_t _microIdx)
  : %(base_class)s(
        "%(mnemonic)s", _machInst, %(op_class)s, _microIdx)
{
    %(set_reg_idx_arr)s;
    _numSrcRegs = 0;
    _numDestRegs = 0;
    setDestRegIdx(_numDestRegs++, matRegClass[_machInst.mrd]);
    _numTypedDestRegs[MatRegClass]++;
    setSrcRegIdx(_numSrcRegs++, intRegClass[_machInst.rs1]);
    setSrcRegIdx(_numSrcRegs++, intRegClass[_machInst.rs2]);
} }};
```

Listing 4.17: Extract of the matrix templates specifications

the register values that need to be fetched in the Constructor portion. Note the presence of `%(...)s` structures, these define smaller template pieces that will be put expanded during the build process. The `%(code)s` template is an important one, as it is the one that brings the behavior we defined in the `decoder.isa` into the instruction definition.

These different parts can be shaped to specific instructions or be flexible and reused between multiple instructions. The *isa_parser* will generate the complete C++ classes representing our instructions.

Note that the base template has a *base_class* parameter, this indicates which base class the instruction follows, these being defined in the `src/arch/riscv/insts/matrix.hh` file. These classes group the instructions by common behavior and allow us to define parameters that are shared between them, as is the case of the flags that define if an instruction is a matrix operation, a load or store, etc. These flags are used by gem5 to identify the functional units that will process the instructions for more detailed CPU models. The base classes for the matrix load instructions are illustrated in Listing 4.18.

```
class MatrixLoadMacroInst : public MatrixMemMacroInst {
  protected:
    MatrixLoadMacroInst(const char* mnem, ExtMachInst _machInst, OpClass __opClass)
        : MatrixMemMacroInst(mnem, _machInst, __opClass)
    { this->flags[IsLoad] = true; }
};

class MatrixLoadMicroInst : public MatrixMicroInst {
  protected:
    Request::Flags memAccessFlags;
    MatrixLoadMicroInst(const char* mnem, ExtMachInst _machInst, OpClass __opClass,
        uint32_t _microIdx)
        : MatrixMicroInst(mnem, _machInst, __opClass, _microIdx), memAccessFlags(0)
    { this->flags[IsLoad] = true; }
};
```

Listing 4.18: Extract of the matrix instruction base classes

Some files need other specific changes for the instructions to work correctly, so we recommend looking at the full details in the GitHub repository to get a reference of all the necessary adjustments. These are mostly one-off changes that you don't need to perform again if you're going to add more instructions.

The same logic applies to the definitions of arithmetic instructions, which differ only in that they do not implement the *initiateAcc* and *completeAcc* methods.

4.5.3. The Matrix Multiplication Kernel

Testing is one of the most important aspects of designing new instructions for an architecture. Usually, the software development kit is not done or fully updated to include the ISA extensions. So, the designer does not have a compiler, assembler, or other tools to validate the simulation model. Let's discuss strategies to overcome this obstacle.

Suppose you want to test your matrix ISA extension using the most basic implementation of matrix multiplication, also called naive or iterative multiplication. It consists of three nested loops. The first two go through every one of the elements of the destination, while the third accumulates the sum of every multiplication of the element of a line

by the element of a column.

Due to the complexity and time required to modify a compiler, we wrote a kernel to use our new instructions with inline assembly instead of compiler-generated instructions. Inline assembly can be quickly adapted or modified, making it easier to maintain across different header versions, which is necessary for faster iteration and testing. To simplify our programming, a header file defines Macros to generate the correct encoding for matrix instructions in a similar form to how it would be done with compiler support.

The Macros define a function-like interface that receives as parameters the necessary operands to compose the entire instruction, like the source and destination registers. In addition, the instruction names encode some of the information in their own names, as is the case of the move instructions, where the *XMU* part of it indicates that the source register is scalar (X), the destination is a matrix (M), and the data should be interpreted as unsigned (U). The definition gets every operand as a string of bits. It merges it with the fixed parts of the instruction, generating a 32-bit word incorporated inside the code as text but interpreted correctly by the simulators. There is also a pseudo-instruction for MZERO, which fills a matrix register with zeroes. The macro definitions can be seen in Listing 4.19. Notice that `mX` denotes the `Xth` matrix register. Other register names follow the usual naming convention for RISC-V.

```
#define MLS(md, rs1, rs2)    ".word 0b0010001" rs2  rs1 "100" md "1110111"
#define MSS(md, rs1, rs2)    ".word 0b0010001" rs2  rs1 "101" md "1110111"

#define MMV_XMU(md, rs1)     ".word 0b010000100001" rs1 "000" md "1110111"
#define MZERO(md)            MMV_XMU(md, X0)

#define MMACF(md, ms1, ms2) ".word 0b0110101" ms2  ms1 "000" md "1110111"
#define MADDF(md, ms1, ms2) ".word 0b0110101" ms2  ms1 "001" md "1110111"
#define MSUBF(md, ms1, ms2) ".word 0b0110101" ms2  ms1 "010" md "1110111"
```

Listing 4.19: Extract of the definitions of the RISC-V Matrix Header.

Using these definitions, we can adapt the matrix multiplication kernel presented in Listing 4.20 to leverage the matrix instructions and perform the multiplication by panels instead of individual elements, enabling higher throughput. The result is shown in Listing 4.21.

As our registers store 4x4 matrices, note that the kernel using matrix multiplication advances four elements per iteration instead of one in the naive multiplication example.

```
void naive_multiply(int M, int N, int K, int ldc, int ldb, int lda, float *C, float *A,
    float *B) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < K; k++) {
                sum = (double) A[i * lda + k] * (double) B[k * ldb + j] + (double) sum;
            }
            C[i * ldc + j] = sum;
        }
    }
}
```

Listing 4.20: Basic multiplication kernel.

```
void matrix_multiply(int M, int N, int K, int ldc, int ldb, int lda, float *C, float
    *A, float *B){
  float *B_t = (float *) malloc(N*K*sizeof(float));
  float *C_ptr = C, *A_ptr = A, *B_ptr = B_t;
  transpose(B, B_t, N, K);
  ldb = K;
  int elem_per_row = 4;                  // Number of elements per row

  // Matrix Multiplication
  for (int i = 0; i < M; i += elem_per_row) {
    for (int j = 0; j < N; j += elem_per_row) {
      C_ptr = C + j + i*ldc;

      asm(MZERO(M2));                    // Reset the accumulator

      for (int k = 0; k < K; k += elem_per_row) {
        A_ptr = A + k + i*lda;
        B_ptr = B_t + k + j*ldb;

        asm(
          "mv t0, %0\n\t"
          "mv t1, %1\n\t"
          MLS(M0, T1, T0)                // Load A
          "mv t0, %2\n\t"
          "mv t1, %3\n\t"
          MLS(M1, T1, T0)                // Load B
          MMACF(M2, M0, M1)              // Multiply-Accumulate
          :: "r" (lda * 4), "r" (A_ptr), "r" (ldb * 4), "r" (B_ptr)
          : "t0", "t1", "memory"
        );
      }

      asm(
        "mv t0, %0\n\t"
        "mv t1, %1\n\t"
        MSS(M2, T1, T0)                  // Store C
        :: "r" (ldc * 4), "r" (C_ptr)
        : "t0", "t1", "memory"
      );
    }
  }
}
```

Listing 4.21: Basic multiplication kernel using matrix instructions.

4.6. Performance Analysis

Performance analysis is a central task for researchers and developers to gain insights into the performance characteristics of architectural models. gem5 provides several features and options supporting detailed performance analysis and high-level performance evaluation. We start with an overview of the primary options and features available in gem5 for performance analysis. Then, we present more detailed examples of some simulation statistics and the Python-based configuration and controlling of simulations.

4.6.1. Simulation Modes

As mentioned above, two different simulation modes affect the performance analysis options available to the user.

- **Full-System Simulation (FS Mode):** gem5 simulates the entire system, including the operating system, peripheral devices, and hardware interactions in this mode.

This mode is crucial for analyzing how architectural decisions impact the overall system performance, including I/O operations, memory management, and interaction with accelerators.

- **System-Call Emulation (SE Mode):** SE mode focuses on user-level processes, making it faster for testing application performance without simulating the entire OS or hardware devices. While less detailed, this mode allows performance analysis on individual components like CPU and cache.

4.6.2. CPU Models

gem5 supports multiple types of CPU models that enable different types of performance analysis:

- **In-order models:** Simulate simple processors where instructions are executed in the order they are issued. These are useful for evaluating simpler designs or embedded systems.
- **Out-of-order models:** These are more complex and allow for performance analysis of advanced, modern CPUs where instructions are executed out of order. This helps identify performance bottlenecks like pipeline stalls, branch mispredictions, and cache misses.

Please refer to Section 4.2.1 for a list of the processor models.

4.6.3. Instrumentation and Statistics Collection

gem5 has built-in instrumentation for collecting detailed statistics about various aspects of the simulated system. Some key features include:

- **Cycle counts:** For tracking the number of cycles required for executing instructions or completing specific tasks.
- **Cache:** gem5 provides cache hit/miss rates, latency statistics, and other detailed information about cache behavior. This is vital for analyzing how different cache configurations affect performance.
- **Pipeline:** Out-of-order CPU models provide detailed insights into how instructions are scheduled, executed, and retired. This allows you to analyze pipeline efficiency and the impact of hazards, stalls, and dependencies.
- **Branch prediction:** gem5 can capture data on branch mispredictions, which is critical for analyzing speculative execution and its effect on performance.

4.6.4. Memory System Analysis

The memory hierarchy is one of the most important aspects of the performance of any real-world application. It is important to be able to simulate and compare different configurations.

- **Cache Models:** gem5 supports configurable cache hierarchies (L1, L2, L3 caches), with options to adjust cache sizes, associativity, and policies (e.g., replacement and write-back/write-through policies). This enables a detailed analysis of how the memory subsystem impacts overall performance.
- **DRAM and Interconnect Models:** For more comprehensive memory performance analysis, gem5 includes detailed models for DRAM memory and interconnects. Researchers can study how memory latency, bandwidth, and contention affect the system's behavior under various workloads.

4.6.5. Interconnect

gem5 allows users to simulate detailed interconnect architectures such as crossbars and network-on-chip (NoC) systems, which are crucial for performance analysis in multicore or many-core systems. These models allow latency, bandwidth usage, and contention analysis, giving insight into how inter-core communication affects overall performance. This topic is out of the scope of our chapter, but the following link has more information on the subject: `https://www.gem5.org/documentation/general_docs/ruby/interconnection-network/`.

4.6.6. Power and Energy Modeling

gem5 integrates an energy consumption estimate into its memory components based on a DRAMPower model. Thus, when analyzing the statistics generated by a simulation, values in Joules correspond to the estimated energy consumption of the memory modules.

McPAT (a power, area, and timing analysis tool) can be combined with gem5 to estimate power consumption, enabling a comprehensive analysis that includes both performance and energy efficiency—critical for modern processor designs that aim to optimize performance per watt.

As with DRAMPower, work has already been done linking gem5 outputs with McPAT inputs. Some scripts can be found on GitHub to do this, allowing gem5 parameters to be ported effortlessly to generate estimates of energy consumption and design area.

Similarly, gem5 integrates with DRAMPower to model and estimate the energy consumption of memory systems, aiming at collecting insights into how different memory configurations and access patterns affect overall energy usage. These integrations allow researchers to perform detailed power and energy evaluations across the entire system, from processors to memory hierarchies.

4.6.7. Performance Profiling and Debugging Tools

gem5 provides powerful tools for performance analysis, including detailed tracing and performance counters. Detailed tracing, as in Listing 4.22, allows users to track simulation events at a fine-grained level, capturing specific instruction executions, memory accesses, and interconnect traffic. This detailed information is invaluable for identifying bottlenecks and performing in-depth performance analysis. Additionally, gem5 includes performance counters that monitor key metrics in real-time, such as instructions per cycle

(IPC), memory accesses, and cache hit rates. These counters enable users to dynamically observe performance trends and system behavior during the simulation, offering insights that help optimize architectural designs. Listing 4.23 shows a sumarized and abridged example of statistics collected from our matrix multiplication implementation.

```
...
3996666: system.cpu: T0 : 0x104c0 @mm+182 : c_slli a2, 2
3996999: system.cpu: T0 : 0x104c2 @mm+184 : c_add a0, s4
3997332: system.cpu: T0 : 0x104c4 @mm+186 : c_add s8, s7
3997665: system.cpu: T0 : 0x104c6 @mm+188 : c_add s9, s3
3997998: system.cpu: T0 : 0x104c8 @mm+190 : c_add a2, s6
3998331: system.cpu: T0 : 0x104ca @mm+192 : c_li a6, 0
3998664: system.cpu: T0 : 0x104cc @mm+194 : mmv_xmu m2, zero
```

Listing 4.22: Abridged extract from matrix multiplication execution trace

```
---------- Begin Simulation Statistics ----------
simSeconds                   0.000385 # Number of seconds simulated
simTicks                    385437177 # Number of ticks simulated
hostSeconds                      0.62 # Real time elapsed on the host
hostTickRate                619899155 # Number of ticks simulated per host second
hostMemory                     650268 # Number of bytes of host memory used
simInsts                       967487 # Number of instructions simulated
simOps                         967487 # Number of ops (including micro ops) simulated
hostInstRate                  1555710 # Simulator instruction rate (inst/s)
hostOpRate                    1555693 # Simulator op (including micro ops) rate
system.cpu.numCycles          1157470 # Number of cpu cycles simulated
system.cpu.cpi               1.196330 # CPI: cycles per instruction (core level)
system.cpu.ipc               0.835889 # IPC: instructions per cycle (core level)
...
dram.bytesRead::cpu.inst      4629880 # bytes read from this memory
dram.bytesRead::cpu.data      1098880 # bytes read from this memory
dram.bytesRead::total         5728760 # bytes read from this memory
dram.bytesInstRead::cpu.inst   462988 # instructions bytes read from this memory
dram.bytesInstRead::total     4629880 # instructions bytes read from this memory
dram.bytesWritten::cpu.data    880270 # bytes written to this memory
dram.bytesWritten::total       880270 # bytes written to this memory
dram.numReads::cpu.inst       1157470 # read requests responded to by this memory
dram.numReads::cpu.data        173591 # read requests responded to by this memory
dram.numReads::total          1331061 # read requests responded to by this memory
dram.numWrites::cpu.data       125132 # write requests responded to by this memory
dram.numWrites::total          125132 # write requests responded to by this memory
```

Listing 4.23: Abridged extract from matrix instruction multiplication stats.txt

4.6.8. Python-based Configuration

In gem5, the Python-based scripting framework provides flexibility and ease of use for configuring and automating simulations. gem5's simulation environment is driven by Python scripts, allowing users to dynamically configure all aspects of the simulation. These scripts can define CPU models, memory hierarchies, cache configurations, interconnects, and peripherals.

Instead of manually adjusting numerous parameters or creating multiple static configuration files for different simulation scenarios, Python scripting allows for programmatic control. Users can easily modify simulation parameters by changing values in their Python scripts, making it easier to explore various architectural designs. For example, you can dynamically change cache sizes, adjust memory latency, or configure the number of CPU cores using Python constructs. This allows for rapid experimentation and

design space exploration, as a single script can be reused with different configurations by adjusting parameters programmatically.

One significant advantage of Python scripting is the ability to automate repetitive tasks, such as running a series of simulations with different configurations (often called parameter sweeping). Instead of manually configuring each scenario, you can write a Python script to loop over various parameter values (e.g., different cache sizes or clock frequencies) and run simulations automatically. For example, as depicted in Listing 4.24, you could write a loop in Python to run multiple simulations with different L1 cache sizes and gather performance data, greatly speeding up the process of design space exploration and optimization.

```
cache_sizes = ['32kB', '64kB', '128kB']
for size in cache_sizes:
    system.cpu.icache = L1ICache(size=size)
    system.cpu.dcache = L1DCache(size=size)
    m5.instantiate()
    print(f"Running simulation with L1 cache size: {size}")
    exit_event = m5.simulate()
    print(f"Simulation with {size} cache size ended at tick {m5.curTick()}")
```

Listing 4.24: Parameter Sweeping

Python-based scripts, like Listing 4.25, allow easy integration with external tools to process the simulation results. After running the simulation, you can use Python to analyze the generated statistics, format the output, and even automate performance data visualization using libraries like Matplotlib or Pandas. This post-processing capability helps to automate the performance evaluation and analysis pipeline, allowing users to generate plots, tables, or reports directly from the simulation results.

```
import pandas as pd
import matplotlib.pyplot as plt

# Post-process the simulation data and generate a report
results = pd.read_csv('stats.txt', sep=' ')
plt.plot(results['sim_ticks'], results['IPC'])
plt.title('IPC Over Time')
plt.xlabel('Simulation Ticks')
plt.ylabel('Instructions Per Cycle (IPC)')
plt.show()
```

Listing 4.25: Post-processing Simulation Data

4.7. Conclusion

This tutorial has explored the fundamentals of architectural simulation using gem5, emphasizing its flexibility in modeling and evaluating complex computer architectures. Using practical examples, we demonstrated how to extend a RISC-V architecture with new instructions for matrix multiplication, a key operation in AI workloads.. By following this guide, readers should now be equipped to experiment with architectural models and make informed design decisions.

References

[Binkert et al. 2011] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

[Binkert et al. 2006] Binkert, N., Dreslinski, R., Hsu, L., Lim, K., Saidi, A., and Reinhardt, S. (2006). The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60.

[Black et al. 2010] Black, G., Binkert, N., Reinhardt, S. K., and Saidi, A. (2010). *Modular ISA-Independent Full-System Simulation*, pages 65–83. Springer US, Boston, MA.

[gem5 2022] gem5 (2022). gem5: Execution Basics. `https://www.gem5.org/documentation/general_docs/cpu_models/execution_basics`. [Accessed 23-09-2024].

[Gem5 Project 2024] Gem5 Project (2024). Getting started with gem5. `https://www.gem5.org/getting_started/`. Accessed: 2024-10-02.

[Lowe-Power et al. 2020] Lowe-Power, J., Ahmad, A. M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Bharadwaj, S., Black, G., Bloom, G., Bruce, B. R., Carvalho, D. R., Castrillón, J., Chen, L., Derumigny, N., Diestelhorst, S., Elsasser, W., Fariborz, M., Farahani, A. F., Fotouhi, P., Gambord, R., Gandhi, J., Gope, D., Grass, T., Hanindhito, B., Hansson, A., Haria, S., Harris, A., Hayes, T., Herrera, A., Horsnell, M., Jafri, S. A. R., Jagtap, R., Jang, H., Jeyapaul, R., Jones, T. M., Jung, M., Kannoth, S., Khaleghzadeh, H., Kodama, Y., Krishna, T., Marinelli, T., Menard, C., Mondelli, A., Mück, T., Naji, O., Nathella, K., Nguyen, H., Nikoleris, N., Olson, L. E., Orr, M. S., Pham, B., Prieto, P., Reddy, T., Roelke, A., Samani, M., Sandberg, A., Setoain, J., Shingarov, B., Sinclair, M. D., Ta, T., Thakur, R., Travaglini, G., Upton, M., Vaish, N., Vougioukas, I., Wang, Z., Wehn, N., Weis, C., Wood, D. A., Yoon, H., and Zulian, É. F. (2020). The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152.

[Martin et al. 2005] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. (2005). Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99.

[Waterman et al. 2014] Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. (2014). The risc-v instruction set manual, volume i: User-level isa, version 2.0. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, page 4.