marss-riscv-docs Documentation

Release 4.1a

Gaurav Kothari

Dec 09, 2020

CONTENTS:

CHAPTER

ONE

IN-ORDER CORE MICROARCHITECTURE

MARSS-RISCV includes a highly configurable cycle-level model of a generic pipelined in-order scalar core. It consists of six pipeline stages: pcgen, fetch, decode, execute, memory, commit. For 5-stage pipeline configuration, pcgen and fetch unit operate as a single stage. The pipeline is shared by the integer and floating-point instructions and supports data forwarding (bypassing)—the figure below a high-level overview of the simulated in-order pipeline.

Fig. 1: Simulated In-order Pipeline

1.1 Overview of Pipeline Stages

The simulated in-order core has the 6 generic pipeline stages: **pcgen**, **fetch**, **decode**, **execute**, **memory**, **commit**. For 5-stage pipeline, pcgen and fetch stages are unified.

1.1.1 PC Generator (pcgen)

By default, pcgen increments PC to point to the next instruction in the sequence. Branch resolution overrides the address in pcgen upon misprediction signal from the memory stage. Branch target buffer (BTB) delivers the target address to pcgen for branches on a BPU hit.

1.1.2 Fetch

The fetch stage probes the instruction TLB, BPU, and instruction cache in parallel using the pcgen stage generated address. On BPU hit, the predicted address is sent to the pcgen unit. The fetch stage will stall until the latency for cache-lookup and memory access (if required) is simulated.

Note: Memory access latency can include latency for reading the cache line containing instruction and hardware page-table walk if required.

1.1.3 Decode

The decode stage decodes the instruction and, if illegal, raises an exception. The return address is pushed onto RAS if a function call is detected. Operands are read from forwarding buses or the register files if valid. Stage stalls until all the data dependencies (RAW and WAW) are satisfied and the required (functional unit) FU is free.

Note: For simplicity, SYSTEM major opcode instructions (*csr{r/w}, ecall, sret, etc.) are handled as software exceptions due to their complex logic. For this purpose, TinyEMU complex opcode helper functions are used.

1.1.4 Execute

The execution pipeline stage is sub-divided into five distinct functional units. They are Integer ALU, Integer Multiplier, Integer Divider, Floating-Point ALU, and Floating-Point Fused-Multiply-Add unit. FPU-ALU is strictly non-pipelined, whereas the rest can be configured to run iteratively or in a pipelined fashion. All the execution units have dedicated data forwarding buses. After simulating the execution latency, the FU writes the result on the forwarding bus on which it **remains valid exactly for one cycle**.

Note: Integer ALU performs memory address calculation for loads, stores, atomic instructions, branch target calculation, and branch condition evaluation.

1.1.5 Memory

Non-memory instructions (arithmetic and branches) pass through the memory stage in a single cycle and push their results, which were calculated during execution on the memory stage's forwarding bus.

For loads, stores, and atomic operations, the memory stage probes the data TLB and data cache in parallel. The stage will stall until the latency for cache-lookup and memory access (if required) is simulated. After simulating the delay, data read by load and atomic instruction is written on the dedicated forwarding bus for the memory stage, which **remains valid exactly for one cycle**.

The entries for branches in the branch predictor structures are created for branches if not present and updated with the branch's new outcome. The target address is forwarded to pcgen, followed by a flushing of preceding stages on misprediction. Entries

---

Note: Memory access latency can include latency for reading/writing the cache line containing data and hardware page-table walk if required.

1.1.6 Commit

The results are written to the register files (Int and FP) in the commit stage.

CHAPTER

TWO

OUT-OF-ORDER CORE MICROARCHITECTURE

MARSS-RISCV includes a highly configurable cycle-accurate model of a generic pipelined out-of-order scalar core. The simulated design does not have separate physical register files. However, it uses the re-order buffer (ROB) slots as the physical registers and separate rename tables for integer and floating-point data. The simulated out-of-order core has seven generic stages: fetch, decode, rename/dispatch, issue, execute, memory, commit. Speculative execution is supported, and instructions are executed on a speculated path as long as required resources are available for dispatch. Figure 2 shows a high-level overview of the simulated out of order core model.

Fig. 1: Simulated Out-of-Order Pipeline

2.1 Overview of Pipeline Stages

The simulated out-of-order core has the 7 generic pipeline stages: **fetch**, **decode**, **dispatch**, **issue**, **execute**, **memory**, **commit**.

2.1.1 Fetch

Fetch stage probes the instruction TLB, BPU, and instruction cache in parallel using the address provided by the pcgen stage. On BPU hit, the predicted address is sent to the pcgen unit. Fetch stage will stall until the latency for cache-lookup and memory access (if required) is simulated.

Note: Memory access latency can include latency for reading the cache line containing instruction and reading/writing page table entries on a TLB miss.

2.1.2 Decode

Decode stage decodes the instruction and, if illegal, raises an exception. Entries in the branch predictor structures are created for branches if not present. The return address is pushed onto RAS if a function call is detected.

Note: For simplicity, SYSTEM major opcode instructions (*csr{r/w}, ecall, sret etc.*) are executed in a **single cycle** due to their complex logic. For this purpose, TinyEMU complex opcode helper functions are used.

2.1.3 Dispatch

The dispatch stage performs register renaming and allocates the required entries in IQ, LSQ, and ROB. The stage stalls if any of the required resources are not available. Valid operands are read from the architectural register file.

2.1.4 Issue

There is a single global issue queue (IQ) for all the instructions, in which the entries are checked sequentially every cycle for issuing to one of the five execution units. When an IQ entry is processed, remaining operands are read from the designated ROB slot if they are ready. IQ entry is issued once all the operands are available, and the target execution unit is vacant.

2.1.5 Execute

The execution pipeline stage is sub-divided into five distinct functional units. They are Integer ALU, Integer Multiplier, Integer Divider, Floating-Point ALU, and Floating-Point Fused-Multiply-Add unit. Integer ALU and FPU ALU are strictly non-pipelined, whereas the rest can be configured to run iteratively or in a pipelined fashion. In out of order core design, all of these five functional units strictly operate in parallel. After simulating the execution latency, the FU writes the result produced in the corresponding ROB entry and marks it as ready to commit.

Note: Integer ALU performs memory address calculation for loads, stores, atomic instructions, branch target calculation, and branch condition evaluation.

2.1.6 Memory

A single unified load-store queue (LSQ) handles all the memory traffic of loads, stores, and atomic instructions in the program sequence from where they are issued to (load-store unit) LSU from the head of LSQ. Though loads are speculatively issued, stores and atomics are issued to memory only if they reach ROB top.

LSU accepts a single memory request from the head of LSQ and probes the data TLB and data cache in parallel. The LSU will stall until the latency for cache-lookup and memory access (if required) is simulated. After simulating the delay, data read by load and atomic instruction is written to the corresponding ROB entry and is marked ready to commit.

Note: Memory access latency can include latency for reading the cache line containing data and reading/writing page table entries on a TLB miss.

2.1.7 Commit

Retirement logic works asynchronously and commits the instructions from the head of Reorder buffer (ROB) when the entry is ready to commit. Results are written to the architectural register file from the corresponding ROB entry. The number of commit ports on ROB is configurable.

2.2 Stage-wise activities based on the instruction type

This section describes stage by stage pipeline activities for various instruction types from the instant they enter the pipeline until they commit.

2.2.1 Operate Instructions

Consider the following operate instruction in the logical format: **OPCODE RD, RS1, RS2**

- **Fetch**
  - Read instruction from memory
- **Decode**
  - Decode the instruction
- **Dispatch (Rename-Dispatch)**
  - Stall, if ROB or IQ is full
  - Read the rename table to get the latest physical register mappings (or ROB indexes) for `RS1` and `RS2`, say `PRS1` and `PRS2` respectively
  - If the value of `PRS1` or `PRS2` is `-1`, this indicates that `RS1` or `RS2` can be safely read from the architectural register file
  - Update `RD` mapping in the rename table to `PRD` (or the ROB index of the current instruction), and save old mapping for the destination `RD` as `PREV_PRD`
  - After renaming, the instruction becomes: `OPCODE PRD, PRS1, PRS2`
- **Issue**
  - If the value of `PRS1` or `PRS2` is not `-1`, then the values of `RS1` or `RS2` are read from ROB slots `PRS1` or `PRS2` respectively if they are ready
  - If the required execution unit is free, issue the instruction to the appropriate execution unit and remove IQ entry
  - Instruction is not issued until all the source operands are read, and the required execution unit is available
- **Execute**
  - Calculate the result, write it to the corresponding ROB entry and mark the ROB entry as ready to commit
- **Commit**
  - Once this instruction comes to ROB top and no exception has occurred, and entry is ready to commit, write the results from ROB entry to the architectural register file
  - Update the rename table mapping for `RD` to `-1`, which indicates that `RD` can now be read directly from the architectural register file

  - Deallocate the ROB entry

2.2.2 Loads

Consider the following load instruction in the logical format: **LOAD RD, RS1, IMM**

- **Fetch**
  - Read instruction from memory
- **Decode**
  - Decode the instruction
- **Dispatch (Rename-Dispatch)**
  - Stall, if ROB, IQ or LSQ is full
  - Read the rename table to get the latest physical register mappings (or ROB indexes) for `RS1`, say `PRS1`
  - If the value of `PRS1` is `-1`, this indicates that `RS1` can be safely read from the architectural register file
  - Update `RD` mapping in the rename table to `PRD` (or the ROB index of the current instruction), and save old mapping for the destination `RD` as `PREV_PRD`
  - After renaming, the instruction becomes: `LOAD PRD, PRS1, IMM`
- **Issue**
  - If the value of `PRS1` is not `-1`, then the value of `RS1` is read from ROB slot `PRS1`, if ready
  - If the `Int-ALU` is free, issue the instruction to `Int-ALU` and remove IQ entry
  - Instruction is not issued until all the source operands are read, and the `Int-ALU` is available
- **Execute**
  - Calculate the memory address, write it to the LSQ entry and mark it as ready
- **Memory**
  - Once the LSQ entry for this load reaches to LSQ top and is valid, issue the load to memory
  - Remove LSQ entry after memory access completes and write the result to corresponding ROB entry and mark the ROB entry as ready to commit
- **Commit**
  - Once this instruction comes to ROB top and no exception has occurred, and entry is ready to commit, write the results from ROB entry to the architectural register file
  - Update the rename table mapping for `RD` to `-1`, which indicates that `RD` can now be read directly from the architectural register file
  - Deallocate the ROB entry

2.2.3 Stores

Consider the following store instruction in the logical format: **STORE RS1, RS2, IMM**

- **Fetch**
  - Read instruction from memory
- **Decode**
  - Decode the instruction
- **Dispatch (Rename-Dispatch)**
  - Stall, if ROB, LSQ or IQ is full
  - Read the rename table to get the latest physical register mappings (or ROB indexes) for `RS1` and `RS2`, say `PRS1` and `PRS2` respectively
  - If the value of `PRS1` or `PRS2` is `-1`, this indicates that `RS1` or `RS2` can be safely read from the architectural register file
  - After renaming, the instruction becomes `STORE PRS1, PRS2, IMM`
- **Issue**
  - If the value of `PRS1` or `PRS2` is not `-1`, then the values of `RS1` or `RS2` are read from ROB slots `PRS1` or `PRS2` respectively if they are ready
  - If the `Int-ALU` is free, issue the instruction to `Int-ALU` and remove IQ entry
  - Instruction is not issued until all the source operands are read, and the `Int-ALU` is available
- **Execute**
  - Calculate the memory address, write it to the LSQ entry, mark it as valid
- **Commit**
  - Once this store comes to ROB top and LSQ top (LSQ entry must be valid), the store is issued to the memory
  - Deallocate the ROB and LSQ entry once the store completes its memory access

2.2.4 Atomics

Atomic instructions are handled similarly to the store instructions are dispatched only when all the prior instructions in the ROB are committed (ROB is empty). The memory access initiates once the ROB entry for atomics reaches the head of the ROB and LSQ.

2.2.5 Branches

Consider the following branch instruction in the logical format: **BRANCH RS1, RS2, IMM**

- **Fetch**
  - Read instruction from memory
- **Decode**
  - Decode the instruction
- **Dispatch (Rename-Dispatch)**

  - Stall, if ROB or IQ is full
  - Read the rename table to get the latest physical register mappings (or ROB indexes) for `RS1` and `RS2`, say `PRS1` and `PRS2` respectively
  - If the value of `PRS1` or `PRS2` is `-1`, this indicates that `RS1` or `RS2` can be safely read from the architectural register file
  - After renaming, the instruction becomes: `BRANCH PRS1, PRS2, IMM`
- **Issue**
  - If the value of `PRS1` or `PRS2` is not `-1`, then the values of `RS1` or `RS2` are read from ROB slots `PRS1` or `PRS2` respectively if they are ready
  - If the `Int-ALU` is free, issue the instruction to `Int-ALU` and remove IQ entry
  - Instruction is not issued until all the source operands are read, and the `Int-ALU` is available
- **Execute**
  - Evaluate the branch condition and calculate the target address
  - On a misprediction,
    * Flush fetch, decode and dispatch stages
    * Set the fetch PC to the target address
    * Flush all the instructions from ROB, LSQ, LSU, and IQ on the miss speculated path, followed by this branch
    * Revert all the rename table mappings, to the point of the dispatch of this branch instruction
  - Mark ROB entry of the branch as ready to commit
- **Commit**
  - Once this branch comes to ROB top and is ready to commit, deallocate the ROB entry

CHAPTER

THREE

BRANCH PREDICTION UNIT

Branch prediction unit (BPU) is configurable and can run a simple bimodal predictor or complex 2-level adaptive predictors like GShare, GSelect, GAg, GAp, PAg, or PAp. This section describes the major structures used in BPU and their interaction with the CPU pipeline.

Fig. 1: BPU High-level Overview

3.1 Branch Target Buffer

Branch Target Buffer (BTB) is modeled as a set-associative data structure. Supported eviction policies are random eviction and bit PLRU.

Note: For a simple bi-modal predictor, we keep the prediction bits in a separate Branch History Table (BHT).

3.2 Return Address Stack

BPU consists of an optional return address stack (RAS), which keeps track of the last `N` function return addresses, where `N` is the number of entries in RAS. The return address is pushed onto RAS from the decode pipeline stage if the decoded instruction becomes a function call. Similarly, if the decoded instruction turns out to be a function return, the address is popped from RAS and forwarded to the fetch stage.

3.3 Two-Level Adaptive Predictor

Level 1 of the adaptive predictor consists of a Global History Table (GHT), and level 2 consists of a Pattern History Table (PHT). Each GHT entry consists of a PC and History Register (HR) of `N` bits. Each PHT entry consists of a PC and an array of 2^N 2-bit saturating up-down counters. Based on the number of GHT and PHT entries, four different prediction schemes are possible, shown in the following table.

GHT entries	PHT Entries	Predictor
= 1	= 1	GAg
= 1	> 1	GAp
> 1	= 1	PAg
> 1	> 1	PAp

3.4 Interaction with the CPU Pipeline

CPU pipeline interacts with the Branch Prediction Unit in mainly four ways as follows:

- **Probing the branch predictor**
  - In the Fetch stage, the branch predictor is probed using the virtual address of the instruction
  - If the address is present in the BTB, then probe returns hit
  - In case of conditional branches, for two-level adaptive predictor, address must also be present in GHT (For PAg and PAp based schemes) and PHT (For GAp and PAp based schemes)
- **Acquiring the target address**
  - On branch prediction unit hit, fetch stage requests the target address from the BPU
  - For unconditional branches, the target is immediately returned from the BTB entry
  - For conditional branches, only if the prediction is taken, the target is returned from the BTB entry
- **Adding entries to branch predictor structures**
  - A BTB entry is created for a branch instruction missed in the BTB when the branch is resolved.
  - In case of two-level adaptive predictor, required entries are also created in GHT and PHT based on the prediction scheme, for conditional branches
- **Updating branch predictor entries**
  - Branch Prediction unit is probed again in memory stage (or `Int-ALU` in the out of order pipeline) after the branch resolves
  - If probe misses, then the updates are skipped
  - If probe results in a hit, BTB entry of the branch is updated with the current target address
  - For the conditional branches, prediction counters are updated based on the latest branch outcome

3.5 Timings

- Probe and target address retrieval: **one cycle** (overlaps with the TLB and cache probes)
- Add and update entry: **one cycle each** (overlaps with relevant stage processing)

CHAPTER

FOUR

SIMULATION OF MEMORY ACCESS DELAY

This section briefly describes the memory access delay simulation and the simulated components of the memory hierarchy. The memory hierarchy includes four main components:

1. Memory controller
2. Translation look-aside buffers (TLB)
3. Caches
4. DRAM

Note: We do not model the actual data in the memory hierarchy for simplicity, but just the addresses (or cache tag arrays) for simulating the delays.

4.1 Translation look-aside buffers (TLB)

TinyEMU emulates a hardware memory management unit (MMU) that manages emulated guest memory and supports *sv32*, *sv39*, and *sv48* paged virtual memory schemes. It translates the virtual memory addresses issued by the emulated CPU to the host system's correct physical memory locations using three distinct direct-mapped address translation buffers (or TLB) for code, loads, and stores. It checks the validity of the CPU issued memory addresses, performs a page table walk in case of a TLB miss, and generates a page fault exception in the emulated CPU on a page miss in the emulated memory.

MARSS-RISCV uses the MMU emulated by TinyEMU to fetch the instructions and data into the simulated CPU pipeline. TLB lookup and address translation on a TLB hit are modeled to complete in a **single cycle** and in parallel with the cache lookup. On a TLB miss, the simulator creates a memory request per page table entry containing the entry's physical address and appends it to the memory controller queue.

4.2 Caches

The cache hierarchy model consists of two levels of physically indexed, physically tagged blocking caches. Level-1 caches include a separate instruction and data cache, whereas Level-2 consists of an optional unified cache. The cache accesses are non-pipelined L2 cache can be accessed in parallel by split L1 caches. A miss in the last level cache (LLC) or eviction creates a memory request appended to the memory controller queue.

4.3 Memory controller

The memory controller includes a single FIFO queue known as `mem_req_queue` comprising all the pending memory access requests. Requests are processed sequentially, one at a time, from the head of `mem_req_queue`.

Memory access requests are added to `mem_req_queue` by cache lookup functions on a cache miss or TLB lookup functions on a TLB miss (for modifying page table entries).

Note: We do not stall the CPU pipeline stage for the `write` requests to complete. However, the delay for a `write` request is nevertheless simulated asynchronously through the memory controller.

4.4 DRAM

MARSS-RISCV supports three DRAM model: `Base Model`, `Ramulator` and `DRAMSim3`.

4.4.1 Base DRAM model

When MARSS-RISCV is configured to run with the `base` DRAM model, requests are processed sequentially, from the head of `mem_req_queue`. Processing solely involves simulating a fixed configurable latency in CPU cycles, known as `mem_access_latency`. After simulating the latency, the stall on the waiting CPU pipeline stage is removed, and the current entry is dequeued from `mem_req_queue`.

The `Base DRAM` model keeps track of the physical page number of the latest request processed. Any subsequent access to the same physical page occupies a lower delay, roughly 60 percent of the fixed `mem_access_latency`.

Fig. 1: Simulated Memory Hierarchy

CHAPTER

FIVE

SIMULATING BENCHMARKS

This section describes how to configure the simulator as per the desired configuration and simulate benchmarks in full system mode.

For this tutorial, we shall be configuring MARSS-RISCV to simulate a simple 5-stage 32-bit in-order RISC-V system on a chip (SoC) and run CoreMark. Coremark is an industry-standard benchmark that measures the performance of central processing units (CPU) and embedded microcontrollers (MCU).

5.1 System Requirements and Dependencies

Check the system requirements and install the required dependencies.

5.1.1 System requirements

- 32-bit or 64-bit Linux machine
- Libcurl, OpenSSL and SDL Libraries
- Standard C and C++ compiler

5.1.2 Installing the dependencies

Ensure all the dependencies (`ssl`, `sdl`, and `curl` libraries) are installed on the system. For Debian-based (including Ubuntu) systems, the packages are: `build-essential`, `libssl-dev`, `libsdl1.2-dev`, `libcurl4-openssl-dev`.

```
$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install libssl-dev
$ sudo apt-get install libsdl1.2-dev
$ sudo apt-get install libcurl4-openssl-dev
```

5.2 Compiling the simulator

First, clone the simulator repository:

```
$ git clone https://github.com/bucaps/marss-riscv
```

Then, `cd` into the simulator source directory:

```
$ cd marss-riscv/src/
```

Open Makefile and set `CONFIG_XLEN` to `32`. Then, compile the simulator using:

```
$ make
```

5.3 Preparing the bootloader, kernel, and userland image

The simplest way to start is by using a pre-built bootloader, kernel, and userland image. The pre-built 32-bit and 64-bit RISC-V userland, bootloader, and kernel are available here: marss-riscv-images

```
$ wget http://cs.binghamton.edu/~marss-riscv/marss-riscv-images.tar.gz
$ tar -xvzf marss-riscv-images.tar.gz
```

The userland image needs to be decompressed first:

```
$ cd marss-riscv-images/riscv32-unknown-linux-gnu/
$ xz -d -k -T 0 riscv32.img.xz
```

Note that the file system on the disk image has almost no space. Hence we need to resize it to the desired size.

Grow the image file to the desired size (`8GB` for this tutorial):

```
$ truncate --size 8G riscv32.img
```

Note: Below steps may require `sudo` access.

Find the first available `losetup` device. On my system, the below command returned: `/dev/loop8`

```
$ sudo losetup -f
```

Attach the disk image to the given loopback device:

```
$ losetup /dev/loop8 riscv32.img
```

Run fsck before growing the file system:

```
$ e2fsck -f /dev/loop8
```

Note: You may require `e2fsck` version `1.43.1` or greater.

Grow the file system to its maximum size:

```
$ resize2fs /dev/loop8
```

Run fsck post resize:

```
$ e2fsck -f /dev/loop8
```

Detach the loopback device:

```
$ losetup -d /dev/loop8
```

At this point, you should have a 32-bit RISC-V Linux image of size 8GB ready to use.

5.4 Configuring the Simulator

Simulation and TinyEMU SoC parameters are configured using the TinyEMU JSON configuration file provided in the configs directory. We will now configure MARSS-RISCV to simulate a single-core 32-bit RISC-V SoC with the following configuration:

- 32-bit in-order core with a 5-stage pipeline, 1GHz clock
- 32-entry instruction and data TLBs
- 32-entry 2-way branch target buffer with a simple bimodal predictor, with 256-entry history table
- 4-entry return address stack
- single-stage integer ALU with one cycle delay
- 2-stage pipelined integer multiplier with one-cycle delay per stage
- single-stage integer divider with eight cycles delay
- All the instructions in FPU ALU with a latency of 2 cycles
- 3-stage pipelined floating-point fused multiply-add unit with one-cycle delay per stage
- 32KB 8-way L1-instruction and L1-data caches with one cycle latency and LRU eviction
- 2MB 16-way L2-shared cache with 12 cycle latency and LRU eviction
- 64-byte cache line size with write-back and write-allocate caches
- 1024MB DRAM with base DRAM model with 75 cycles for main memory access

Based on the above configuration, the `configuration` will look like the below.

```
/* VM configuration file */
{
    version: 1,
    machine: "riscv32", /* riscv32, riscv64 */
    memory_size: 1024, /* MB */
    bios: "riscv32-unknown-linux-gnu/bbl32.bin",
    kernel: "riscv32-unknown-linux-gnu/kernel-riscv32.bin",
    cmdline: "console=hvc0 root=/dev/vda rw",
    drive0: { file: "riscv32-unknown-linux-gnu/riscv32.img" },
    eth0: { driver: "user" },

    core: {
        name: "32-bit inorder riscv CPU",
        type: "incore", /* incore, oocore */
```

(continues on next page)

(continued from previous page)

```
        cpu_freq_mhz: 1000,
        rtc_freq_mhz: 10,

        incore : {
            num_cpu_stages: 5, /* 5, 6 */
        },

        oocore: {
            iq_size: 16,
            iq_issue_ports: 3,
            rob_size: 64,
            rob_commit_ports:4,
            lsq_size: 16,
        },

        /* Note: Latencies for functional units, caches and memory are␣
↪specified in CPU cycles */
        functional_units: {
            num_alu_stages: 1,
            alu_stage_latency: "1",

            num_mul_stages: 2,
            mul_stage_latency: "1,1",

            num_div_stages: 1,
            div_stage_latency: "8",

            /* Note: This will create a pipelined FP-FMA unit with 4 stages␣
↪with a
             * latency of 1 CPU cycle(s) per stage */
            num_fpu_fma_stages: 2,
            fpu_fma_stage_latency: "1,1",

            /* Note: FP-ALU is non-pipelined */
            fpu_alu_stage_latency: {
                fadd: 2,
                fsub: 2,
                fmul: 2,
                fdiv: 2,
                fsqrt: 2,
                fsgnj: 2,
                fmin: 2,
                fmax: 2,
                feq: 2,
                flt: 2,
                fle: 2,
                cvt: 2,
                fcvt: 2,
                fmv: 2,
                fclass: 2,
            },

            /* Latency for RISC-V SYSTEM opcode instructions (includes CSR␣
↪and privileged instructions)*/
            system_insn_latency: 3,
        },
```

(continues on next page)

(continued from previous page)

```
        bpu: {
            enable: "true", /* true, false */
            flush_on_context_switch: "false", /* true, false */

            btb: {
                size: 32,
                ways: 2,
                eviction_policy: "lru", /* lru, random */
            },

            bpu_type: "bimodal", /* bimodal, adaptive */

            bimodal: {
                bht_size: 256,
            },

            adaptive: {
                ght_size: 1,
                pht_size: 1,
                history_bits: 2,
                aliasing_func_type: "xor", /* xor, and, none */

                /* Given config for adaptive predictor will create a Gshare␣
↪predictor:
                *   1) global history table consisting of one entry, entry␣
↪includes a 2-bit history register
                *   2) pattern history table consisting of one entry, entry␣
↪includes an array of 4 saturating counters
                *   3) value of history register will be `xor` ed with␣
↪branch PC to index into the array of saturating counters
                */
            },

            ras_size: 4, /* value 0 disables RAS */
        },

        caches: {
            enable_l1_caches: "true", /* true, false */
            allocate_on_write_miss: "true", /* true, false */
            write_policy: "writeback", /* writeback, writethrough */
            line_size: 64, /* Bytes */

            icache: {
                size: 32, /* KB */
                ways: 8,
                latency: 1,
                eviction: "lru", /* lru, random */
            },

            dcache: {
                size: 32, /* KB */
                ways: 8,
                latency: 1,
                eviction: "lru", /* lru, random */
            },

            l2_shared_cache: {
```

(continues on next page)

(continued from previous page)

```
                enable: "true",
                size: 2048, /* KB */
                ways: 16,
                latency: 12,
                eviction: "lru", /* lru, random */
            },
        },
    },

    memory: {
        tlb_size: 32,

        /* Memory controller burst-length in bytes */
        /* Note: This is automatically set to cache line size if caches are␣
↪enabled */
        burst_length: 64, /* Bytes */

        base_dram_model: {
            mem_access_latency: 75,
        },

        dramsim3: {
            config_file: "DRAMsim3/configs/DDR4_4Gb_x16_2400.ini",
        },

        ramulator: {
            config_file: "ramulator/configs/DDR4-config.cfg",
        },
    },
}
```

5.5 Run the simulator

```
$ ./marss-riscv -rw -ctrlc -sim-mem-model base <path-to-config-file>
```

5.6 Load the benchmark and the simulation utility programs inside the guest VM

Now we will load the CoreMark benchmark and MARSS-RISCV simulation utility programs using `git clone` inside the guest VM. Before that, you may want to set the time to the current time in the VM manually. So in the guest terminal, type:

```
$ date --set="9 Dec 2019 10:00:00"
```

To clone the repos, type:

```
$ git clone https://github.com/eembc/coremark.git
$ git clone https://github.com/bucaps/marss-riscv-utils.git
```

To install the simulation utility programs, type:

```
$ cd marss-riscv-utils
$ make
```

This installs the following commands (programs): `simstart`, `simstop` and `simulate`, which will help us to enable and disable simulation mode, before and after running CoreMark, respectively.

At this point, we are pretty much ready to run CoreMark.

5.7 Run Benchmark

Switch to CoreMark directory inside the guest VM and compile the benchmark:

```
$ cd ../coremark
$ make compile
```

This will generate the coremark executable: `coremark.exe`. It has 3 set of inputs and the command lines are as follows (based on Makefile):

- `./coremark.exe 0x0 0x0 0x66 0 7 1 2000 > ./run1.log`
- `./coremark.exe 0x3415 0x3415 0x66 0 7 1 2000 > ./run2.log`
- `./coremark.exe 8 8 8 0 7 1 2000 > ./run3.log`

Then, to simulate the benchmark inside the guest VM, type:

```
$ simstart; ./coremark.exe  0x0 0x0 0x66 0 7 1 2000 > ./run1.log; simstop;
$ simstart; ./coremark.exe  0x3415 0x3415 0x66 0 7 1 2000  > ./run2.log;␣
↪simstop;
$ simstart; ./coremark.exe 8 8 8 0 7 1 2000 > ./run3.log; simstop;
```

After every `simstop` command, the summary of the performance stats is printed on the console, and the stats file for every run is generated based on the current time-stamp.