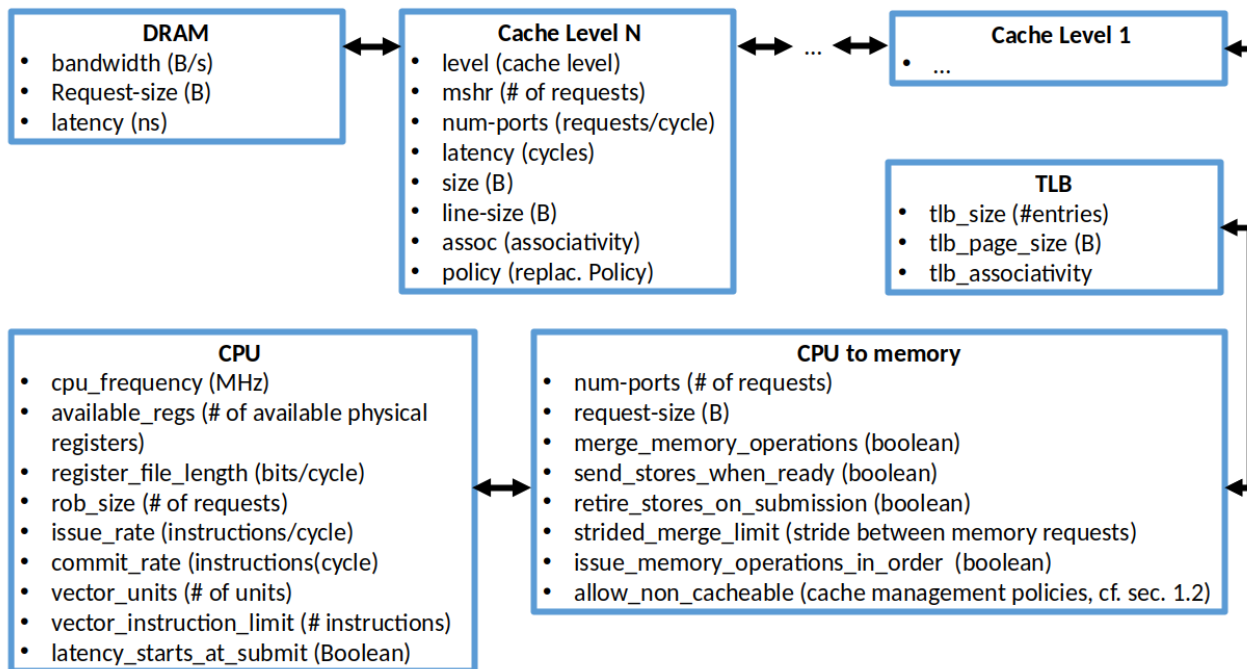


MUSA-RISCV processor model and its architecture parameters

This document describes the architectural parameters that the MUSA-RISCV model uses to describe the architecture it simulates (Section 1), the instruction latency format it uses to compute the duration of non-memory operations (Section 2), the pipeline processor model it implements (Section 3), and the additional events the MUSA-RISCV model adds in the output Paraver trace (Section 4).



1. Architecture Parameters

1.1 CPU section

- **cpu_frequency**: frequency in MHz of the simulated CPU (affects interaction with DRAM).
- **available_regs**: number of available physical vector registers. The RISC-V ISA specification requires 32 logical registers, so values < 32 might cause the simulation to fail. `available_regs = 0` means unlimited registers; `available_regs = -1` means there is no renaming. The EPI EPAC has 40 physical registers, although high-performance computing machines have typically much more. For example, the NEC SX-Aurora VE has 192 physical registers.
- **register_file_length**: throughput of the vector functional units file, in bits/cycle. The EPI SDV designs has a throughput of 512-bits / cycle, which means eight double precision lanes.
- **rob_size**: reorder buffer size, in terms of vector instructions. The EPI SDV has an 8-entry reorder buffer for vector instructions.

- **issue_rate:** maximum number of instructions the CPU can be issue every cycle. The EPAC has an issue rate of 1.
- **commit_rate:** maximum number of instructions the CPU can commit every cycle. The EPAC has a commit rate of 1.
- **vector_units:** it limits the maximum number of vector instructions that run in the functional units at the same cycle. Some instruction types need exclusive use of the vector unit a determined number of cycles (see the instruction latency section). Simulating a single vector unit implies setting this parameter to 1.
- **vector_instruction_limit:** it limits the number of vector instructions that can be inside the ROB any given time (0: not limited). This an experimental option to limit non-memory instructions, we recommended setting it to 0.
- **latency_starts_at_submit:** it defines whether the processor pays instructions startup latency at submit or at decode stages. The latency format file, which Section 2 describes, defines startup latency. It is a Boolean parameter. By default latency starts to be paid at decode, that is, `latency_starts_at_submit = 0` by default.

1.2 CPU-to-memory section

- **num-ports:** number of requests that can be sent every cycle to L1Cache, also the number of ACKs that can be received every cycle. Typically, there is 1 port.
- **request-size:** maximum size in Bytes of the requests to memory. This value is typically set to 64Bytes in common architectures.
- **merge_memory_operations:** it allows discarding vector stores operations. When a subsequent store contains their same memory address range, and vector loads when a previous load/store contains it. It is a Boolean parameter. By default it is enabled, that is, `merge_memory_operations = 1` by default.
- **send_stores_when_ready:** it allows submitting store vector operations once they become ready. Otherwise, they are submitted when they are at the head of the reorder buffer. It is a Boolean parameter, By default, it is enabled, that is, `send_stores_when_ready = 1`.
- **retire_stores_on_submission:** it allows removing stores from the reorder buffer once their memory requests have been sent, without waiting for the acknowledgment from memory. It is a Boolean parameter. It is enabled by default, that is, `retire_stores_on_submission = 1` by default.
- **strided_merge_limit:** it defines the maximum stride for which the requests of a strided load or strided store are merged into one request per cache line. The default value is set to 4. This parameter impacts accesses triggered by the same instruction and it is independent of the `merge_memory_operations` parameter.
- **issue_memory_operations_in_order:** a Boolean parameter that forces memory instructions to be executed in-order. Setting this parameter to 0 lets memory instructions to be executed out-of-order. Be default, the parameter is set to 0.
- **allow_non_cacheable:** it allows the generation of requests that will not store the data in the L1Cache. Similar to what *non-temporal* hints do in other architectures. Its default value is 0. Section 1.5 describes how to create the lists required by parameters 2-5, 7-10.
 - o 0: Everything is cached.
 - o 1: Nothing is cached.
 - o 2: Listed ADDRs are non cached.
 - o 3: Listed PCs are non cached.
 - o 4: Listed ADDRs are cached.
 - o 5: Listed PCs are cached.
 - o 6: Region reuse predictor.

- o 7: Multiperspective reuse predictor + List of cached ADDRs.
- o 8: Multiperspective reuse predictor + List of cached PCs.
- o 9: Multiperspective reuse predictor + List of bypassed ADDRs.
- o 10: Multiperspective reuse predictor + List of bypassed PCs.
- o 11: Source-code instructions reuse predictor.

1.3 Cache section

- **mshr:** it defines the number of concurrent outstanding requests that the Cache can handle at any given time. For the case of the EPI EPAC, its value is 32.
- **level:** it defines the level of the cache (for output statistics purposes). It starts at 1. The cache hierarchy can have a generic number of levels.
- **num-ports:** it sets the number of requests that can be sent to the **next** memory level each cycle. The value should typically be set 1.
- **latency:** it defines the latency in cycles of the cache. A typical 32KB L1 should have from 1 to 4 cycles latency. The L2 cache of EPAC has 40 cycles latency.
- **size:** it sets the cache size in bytes. EPI L2 cache has 262144 Bytes per vector unit.
- **line-size:** it defines the cache line size in bytes. Its value is typically 64 Bytes.
- **assoc:** it defines the cache associativity. The L2 cache associativity is 8.
- **victim-lines:** it sets the size of the victim cache in lines. We recommend setting it to 0.
- **policy:** it defines the name of the cache replacement policy. We recommend to set it to LRU, that is, policy=LRUPOLICY. All supported policies are LRU, NRU, Random. To use them, just set policy=XPOLICY where X={LRU, NRU, Random}.
- **tlb_size:** it represents the number of entries for the TLB. The simulator reports TLB hits and misses, but it does not apply any performance penalty for the later. As such, the TLB configuration does not influence performances. Its default value is 4096.
- **tlb_page_size:** it represents the TLB page size in Bytes. Its default value is 4096. Like the other TLB-related parameters, it does not influence performance.
- **tlb_associativity:** it represents the TLB cache associativity. Its default value is 4.

1.4 DRAM Section

- **bandwidth:** maximum DRAM bandwidth [bytes/s].
- **request-size:** request size in bytes.
- **latency:** single request latency in nanoseconds.

This is a very simplified model, which only accepts bandwidth and request latency as parameters. To achieve that maximum bandwidth the number of requests is limited to:

$$\text{max_requests} = \text{bandwidth} * \text{latency} / (\text{request-size} * 10^6 * \text{cpu_frequency})$$

The 10^6 is added to cancel the units as the `cpu_frequency` is input in MHz ($[\text{bytes/s}] * [\text{ns}] / ([\text{bytes}] 10^6 * [\text{MHz}])$).

The simulator supports more complex DRAM models than the 3-parameter model described above. These complex DRAM models are based on RAMULATOR and DRAMSim. Please, contact marc.casas@bsc.es or francesc.martinez@bsc.es to figure out how to use these more complex models.

1.5 Parameters describing some aspects of the simulator behavior:

- **latency_file:** instruction latency file location (Section 2 describes its format).
- **detail_mode:** CPU simulation mode (always use 'RISCV').
- **deadlock_detection_interval:** interval between each inactivity check. Expressed in cycles. We recommend to set it to 1000000.
- **detail_mode:** must be set to "RISCV" when simulating RISCV traces.
- **flush_final_trace:** This parameter allows generating the output trace as the simulation runs. When disabled, the whole trace is stored in memory until the simulation ends. We recommend setting this parameter to 1 when dealing with large paraver traces.
- **disable_paraver_trace:** It allows disabling the emission of the output paraver trace.
- **generate_reuse_trace:** generates an output trace describing the reuse behavior of all instruction PCs and addresses. It can be used as a guide to generate the lists driving some of the **allow_non_cacheable** configurations.
- **params:** contains the path of the file with the list of addresses/PCs needed for options [2-5] and [7-10] of **allow_non_cacheable**.

1. Instruction latency format

The instruction latency file contains a list of instructions and, for each instruction. It defines the following parameters:

- **opcode:** It defines the instruction type. For example, "vadd".
- **startup_latency:** It defines the value of instructions startup latency in cycles. This startup latency is typically paid once the instruction is inserted in the ROB, although parameter **latency_starts_at_submit** defines when startup latency is paid.
- **scaling_factor:** It is a coefficient that multiplies the $GVL * SEW / RFL$ ratio.

Each instruction belongs to one of the four categories described below. These categories define the model the simulator uses to compute the total latency of the instruction. The simulator uses parameters like instructions granted_vector_length (GVL), single_element_width (SEW), and FP units throughput (called register_file_length - RFL) to compute the total latency.

- **ALU_FP_INST:** For the instructions belonging to this category, the **startup_latency** does not require the use of a vector functional unit. Scaling latency ($scaling_factor * GVL * SEW / RFL$) is paid once the instruction is submitted to the FP functional units and it requires exclusive use of a vector functional unit.
- **ALU_FP_INST_NON_BLOCKING:** For the instructions belonging to this category, neither the startup latency, nor the scaling latency, requires the use of vector functional units.
- **ALU_NON_FP_INST:** These instructions have a startup latency of 0 cycles. The total instruction latency ($scaling_factor * GVL * SEW / RFL$) does not need the exclusive use of a vector unit.
- **NON_ALU_INST:** These instructions have a constant latency defined by the **startup_latency** parameter. They do not need the exclusive use of a vector unit.

1. Pipeline Processor Model.

The pipeline model that MUSA RISC-V implements requires all instructions go through the decode, ready, submit, complete and commit steps. Along these steps, the simulator generates the events **decode_cycle**, **ready_cycle**, **submit_cycle**, **complete_cycle** and **commit_cycle** and adds them to the output Paraver trace when the instruction finishes executing.

- An instruction is decoded when there is an empty entry in the reorder buffer (**decode_cycle**).
- First, we check that all its input dependencies have been computed, otherwise we wait.
- When the input dependencies are ready, if it is a memory instruction we check for dependencies with the other memory instructions in the reorder buffer (cf. Section 3.3 memory dependencies).
- We check that whether there is an empty register to write the result (for non store instructions) (**ready_cycle**).
- At this point the handling of memory and non memory instructions separates.

For **memory** instructions:

- They are added in the load/store queue, where they will be issued whenever the output port from the CPU to the L1Cache is available (it may be full if the L1Cache has the maximum number of request pending) (**submit_cycle**).
- Whenever the output port is available, the oldest ready memory instruction's requests are submitted. Newer instructions can be submitted before older ones if those are not ready, unless parameter **issue_memory_operations_in_order** is set to 1.
- The output buffer will be filled with one Request per cache line that is contained in the request. For gather/scatter or strided load/stores with stride > 4 (configurable with the **strided_merge_limit** parameter) Requests will be issued per vector element (one per cache line that they touch).
- Whenever all the ACKs from the L1Cache have arrived, the instruction is marked as complete (**complete_cycle**).

For **non-memory** operations:

- The instruction is submitted to execution (**submit_cycle**)
- The instruction contains two latency values
 - o Fixed latency: latency due to the use non-blocking resources.
 - o Variable latency: latency due to the length of the operands, it requires exclusive use of a vector unit (configurable via **vector_units** parameter).
- We wait until the fixed latency has elapsed ($\text{current_cycle} > \text{decode_cycle} + \text{fixed_latency}$).
- Then we wait for a vector unit to be available to execute the variable latency (number of **vector_units** is indicated in **MemCPU::vector_units**).
- After the instruction has spent its variable latency on the vector unit, the vector unit is liberated and the instruction is marked as complete (**complete_cycle**).
- For all instructions:
 - When an instruction is complete and is the first in the reorder buffer, it can be committed (**commit_cycle**).

The number of instructions issued and committed every cycle is controlled by the configuration parameters **issue_rate** and **commit_rate**.

3.1 Requests life cycle in the memory hierarchy.

- When a request arrives to the Cache, it waits for the latency of the Cache.
- After the latency has elapsed:
 - If the requested line is available, it is served (hit), and touched for the cache replacement policy.
 - If the line is unavailable, but present on the victim_cache, it is swapped back into the main cache and served (hit).
 - If the line is unavailable, but a MSHR entry for it is already present, we add the request to that MSHR entry and wait for the ACK for that Cache line to arrive (half_miss).
 - If the MSHR entry is not present, and we can allocate the MSHR entry and the new cache line, we create the MSHR entry and send the request to the next memory level.
 - If either the MSHR entry cannot be allocated (reached the limit), or the cache line cannot be allocated (all lines in the block are either waiting for an ACK or dirty, or we cannot evict a line to the victim cache because all victim cache lines are dirty), we stall the Cache until an ACK resolves the resource issue.
- When a request arrives to the DRAM, we check if it can be processed.
 - This is due to the simplified mode having a request latency and a bandwidth configuration, which forces a maximum number of simultaneous processing requests in order to avoid going over the specified bandwidth, **this number is always ≥ 1** .
- If we can process it, we add the request latency, and send the ACK once that request latency has elapsed.

3.2 Other details

The number of requests that a hardware component, either a CPU or a cache, can send to the next level in a cycle is controlled by the **num_ports** parameter. This parameter also limits the number of acknowledgment messages that can be read in a cycle from that same next level.

Therefore, the bandwidth of the link would be:

$$BW = (\text{request-size} * \text{frequency} * \text{num-ports}) \text{ [bytes/second]}$$

3.3 Memory dependencies

When a new memory instruction is decoded, we check for dependencies with all pending memory instructions from newer to oldest, for those instructions that have not been discarded.

For contiguous accesses (load, stores, strided with stride ≤ 4):

- If it is a store instruction and the following three conditions are true: i) It fully contains a previous non-submitted store, ii) there is no load/store instructions between the two stores with a dependency with the first one, and iii) the **merge_memory_operations** parameter is set to 1, we discard the previous store instruction.
- If it is a load that is fully contained in a previous load, and we have the **merge_memory_operations** parameter set to 1, we discard the new instruction's requests, and we consider all dependencies on the second load fulfilled once the first one completes.

- If it overlaps with a previous pending memory instruction, and they are not both loads, we add a dependency between the first and the second instruction and hold we hold the second one until the first instruction completes.

For non-contiguous accesses (strided >4, gather and scatter):

- If it overlaps with a previous memory instruction, and they are not both loads, **we add a dependency** and **hold** it until the previous instruction has been completed.

If the instruction is not discarded or put on hold until the dependency is fulfilled, it is considered **ready**.

1. Events present in the output Paraver trace

In the Paraver trace each instruction is represented by a 1 ns. Event, making the X axis the number of instructions instead of time. There are several events available:

- **decode_cycle**, **ready_cycle**, **submit_cycle**, **complete_cycle** and **commit_cycle** from the instruction lifecycle logic are printed at the end of the instruction lifecycle.
- **L1_hit**, **L2_hit**, **L3_hit**, **MM_hit**: number of requests to the caches or main memory that are a hit.
- **MEM-merge**: if the memory requests have been discarded by the memory dependency logic.
- **TLB-misses**: misses in the L1Cache TLB for the requests generated by the instruction.
- **DRAM-requests**: number of DRAM requests that have been fulfilled before the instruction **commit_cycle**.

The events **complete_cycle**, **commit_cycle** and **DRAM_requests** are printed at the end of the time step representing the instruction in the Paraver trace, so the view “**Next Event Value**” should be used in Paraver. All other events are in the initial line and “**Last Event Value**” should be used for them.