*Technical Report*

UCAM-CL-TR-972
ISSN 1476-2986

Number 972

# Muntjac multicore RV64 processor: introduction and microarchitectural guide

Xuan Guo, Daniel Bates, Robert Mullins, Alex Bradbury

June 2022

This work was kindly supported by lowRISC CIC.

ISSN 1476-2986

**Abstract**

Muntjac is an open-source collection of components which can be used to build a multicore, Linux-capable system-on-chip. This includes a 64-bit RISC-V core, a cache subsystem, and TileLink interconnect supporting cache-coherent multicore configurations and I/O. Each component is easy to understand, verify, and extend, with most being configurable enough to be useful across a wide range of applications. In its current state, Muntjac achieves 2.17 DMIPS/MHz and 3.01 CoreMark/MHz, and can achieve 80+ MHz (with FPU enabled) when targeting a Xilinx Kintex 7 FPGA. This document provides an overview of Muntjac, the standards it follows and an explanation of design decisions and implementation details.

# 1 Introduction

Muntjac [5] is an open-source collection of components which can be used to build a multicore, Linux-capable system-on-chip. This includes a 64-bit RISC-V core, a cache subsystem, and TileLink interconnect supporting cache-coherent multicore configurations and I/O. Each component is easy to understand, verify, and extend, with most being configurable enough to be useful across a wide range of applications.

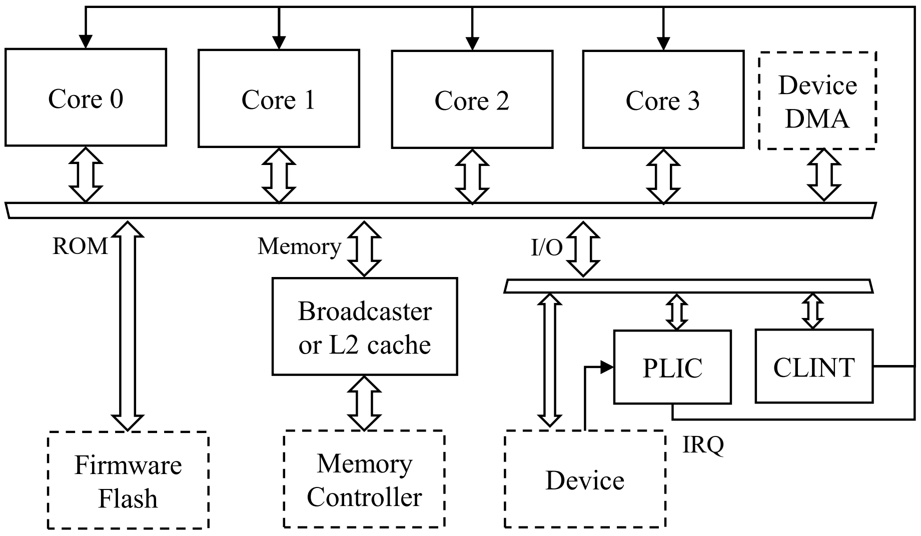

Figure 1: Example multi-core Muntjac system. All solid-border components are provided by Muntjac.

Inspired by the lowRISC's Ibex project [6], we prioritise verification and documentation so users can make rapid progress and be confident that Muntjac's behaviour adheres to all appropriate standards. We focus on having clean, well-tested designs with clear routes to further customisation and improvements. Muntjac's performance is competitive with other open-source projects, and we aim for a power/performance/area compromise that maximises the value of Muntjac as a baseline design for others to use.

We anticipate Muntjac being valuable in the following situations, among others:

- Education: the complete code for a high-quality modular processor is available, with documentation, verification support and examples, and lends itself to self-contained extensions (e.g. a new branch predictor). This document also discusses some of the nuances of real processor design, and some of the key design decisions we made.
- Research: Muntjac can serve as a solid baseline for experimentation, removing the need to build and test huge swathes of general-purpose infrastructure needed to support a single novel component.
- Industry: many real-world applications value "time to solution" over absolute performance or energy efficiency. Muntjac is a reliable, configurable, complete starting point, allowing engineers to focus on the unique selling points of their target system.

Why use Muntjac? At the project's inception, there were a number of similar RISC-V projects available, but none had the right mix of features for us. We believe we have now reached a unique point in terms of performance, documentation, verification and ease of understanding. We have a base design that we have written entirely ourselves, so we have a consistent coding style, all components have been designed together to complement each other, and we are able to explain the design better to new contributors. ccessibility is aided by a relatively small and well-structured codebase, and usage of popular, well-supported hardware description languages (SystemVerilog) and standards (RISC-V, TileLink) to maximise interoperability with other tools and IP. Everything "just works" out-of-the-box, and can be configured to suit new designs.

We are keen to see the community of Muntjac users grow, and together we will extend and improve the range of components we offer.

## 2 Core overview

The Muntjac core is single-issue, in-order, scalar, and supports the RV64GC instruction set with machine and supervisor ISA. Floating-point extensions (F and D) are optional and can be configured with SystemVerilog parameters. The list of supported instruction-set extensions and the standards they conform to is detailed in Table 1.

| **Standard** | **Version** |
|---|---|
| RV64I: Base Integer Instruction Set, 64-bit | 2.1 |
| M: Standard Extension for Integer Multiplication and Division | 2.0 |
| A: Standard Extension for Atomic Instructions | 2.1 |
| C: Standard Extension for Compressed Instructions | 2.0 |
| F: Standard Extension for Single-Precision Floating-Point | 2.2 (Optional) |
| D: Standard Extension for Double-Precision Floating-Point | 2.2 (Optional) |
| ZiCSR: Control and Status Register (CSR) | 2.0 |
| Zifencei: Instruction-Fetch Fence | 2.0 |
| Machine ISA | 1.11 |
| Supervisor ISA | 1.11 |

Table 1: List of RISC-V instruction-set extension standards implemented

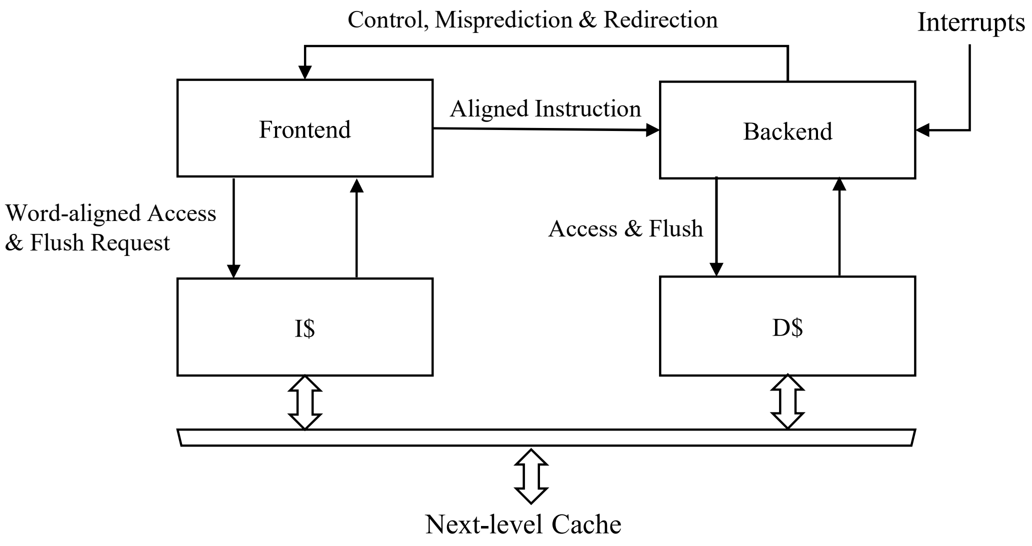

Figure 2: High-level overview of Muntjac core components

The Muntjac core is designed with modularity in mind. It aims to be easy to understand, verify and extend. As shown in Figure 2, the core is separated into 4 components. The frontend and the backend are loosely coupled and valid-ready signals are used for stalling and back-pressure. The instruction and data caches are also loosely coupled with the frontend and backend, respectively. Separating pipeline design and cache design allows a differently designed cache to be swapped in easily without having to adjust the design of the pipeline.

In general, we choose distributed stall signals over a global stall signal or a stall-free design. Valid-ready signals are widely used both within components and between components, so each component and each pipeline stage can be mostly self-contained. Skid buffers are used when distributed stall signals start to cause timing issues.

## 3 Pipeline

### 3.1 Frontend

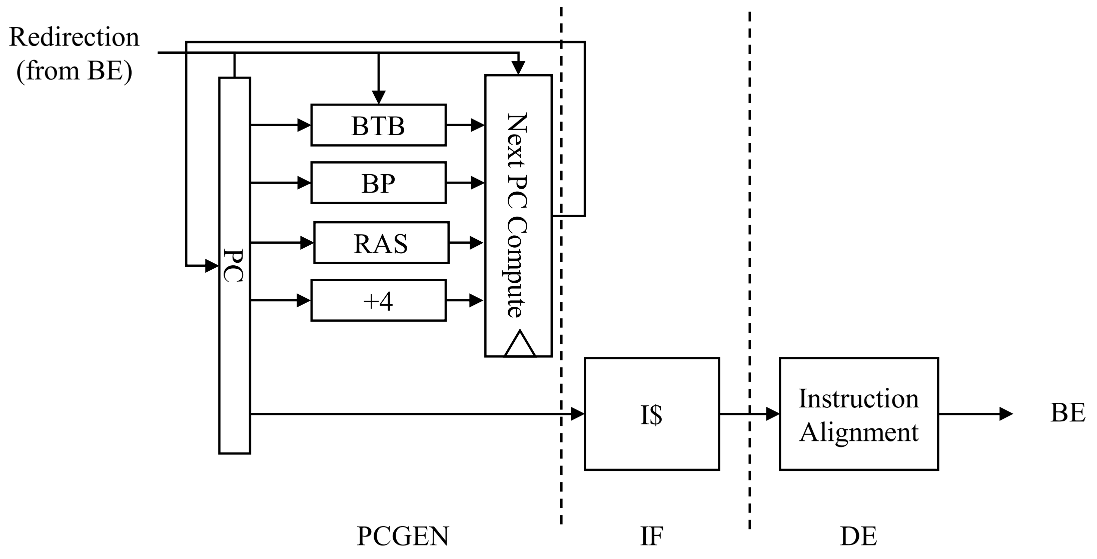

Figure 3: Muntjac frontend design

Muntjac's frontend design is illustrated in Figure 3. It is a fairly classical frontend design, except that variable-length instructions are supported. There is a separate PCGEN stage that generates the next PC, without peeking into the fetched instruction bytes.

Muntjac supports the compressed instruction (C) extension. The C extension is desirable because it brings significant code size reductions, improves instruction cache efficiency and is also required by most Linux distributions. We made a deliberate choice to not allow the C extension to be turned off via a Verilog parameter, because doing so requires us to add support for the misaligned instruction exception. A RISC-V misaligned instruction exception is triggered at the jump instruction that causes it; this is the only possible exception generated by jump instructions, so requiring the C extension can make the backend simpler.

It should be noted that the C-extension does not only adds 2-byte compressed instructions, but also relaxes the alignment requirement of 4-byte instructions to be only 2-byte aligned. The relaxed alignment requirement does mean that Muntjac has to deal with the complexity of a misaligned 4-byte instruction, potentially crossing a cache line or even a page boundary. For any instructions that are not 4-byte aligned, two bytes will be fetched first; the following bytes might be prefetched, but the exception triggered during the prefetch would be discarded if the first two bytes indicate that the instruction is compressed.

The complexity of compressed instructions and misaligned instructions is entirely handled within the frontend; the instruction cache only needs to support accesses aligned to word (32-bit) boundaries. When the PCGEN stage generates a PC, it will also generate a mask indicating which half-words of the fetched word are significant. For example, when a branch lands on a misaligned location (i.e. $PC\%4 = 2$), a mask of 0b10 is generated; when the PCGEN stage predicts that control flow will deviate after an instruction that ends on a misaligned location (e.g. a misaligned 4-byte predicted-taken branch), a mask of 0b01 is generated; a mask of 0b11 is generated for most other scenarios where all half-words are significant. The instruction alignment logic takes the fetched words and the masks and re-segments into individual instructions to be decompressed and decoded.

The branch target buffer (BTB), return address stack (RAS) and branch predictor are used to predict the next PC. In the current version of Muntjac, a direct-mapped BTB is used. The branch target buffer will output 3 pieces of information:

- the type of instruction: whether the instruction is a conditional branch, jump, call, or return instruction, or not a control flow instruction at all;
- the target PC in case the instruction is a branch or a jump;
- whether the second half-word is part of the instruction stream (in case the control-flow instruction ends on a misaligned location).

The RAS is pushed when a BTB predicts a call or a yield instruction, and is popped when BTB predicts a return or a yield instruction. The RAS is implemented like a ring; pushing when the RAS is full will overwrite the first entry, and popping when the RAS is empty reads the last entry. A separate set of buffer pointers are maintained that are only adjusted when a call/return/yield instruction is committed; in case of a misprediction, this counter will override the speculative counters to re-balance the return address stack, so that a single misprediction will not cause cascade RAS mispredictions.

The branch predictor is used when the BTB predicts a conditional branch. Currently, the branch predictor implemented is a simple bi-modal 2-bit saturating counter, but like other components, the interface between frontend and branch predictor is well-defined and a different implementation can be easily swapped in.

The backend notifies the frontend for all control-flow instructions committed whether correctly predicted or not. The frontend makes use of this information to train the branch predictor, and in case of a misprediction, also trains the BTB, adjusts RAS pointers, and re-initiates the fetch.

## 3.2 Backend

The aligned instructions from the frontend are expanded into 4-byte instructions if they are compressed, decoded and have operands fetched in the DE stage as shown in Figure 4. The issue logic manages control, data and structural hazards, and issues instructions into one of the functional units.

The arithmetic logic unit (ALU) and the branching unit complete in a single cycle, while the latency of other functional units may be multiple and variable number of cycles. To hide the latency of data cache accesses (usually 2 cycles when cache hits), there are two execution stages, EX1 and EX2. All instructions progress from EX1 to EX2 before their result is written back; a register is placed between the two stages to hold the result if the functional unit completes before it could progress into EX2.

CSR accesses and other system instruction executions are handled outside the normal data flow. When a system instruction is decoded, the issue logic blocks it from continuing down the pipeline, and instead waits for all previously issued instructions to commit (or trap). It will then dispatch the instruction to the control state machine when the pipeline

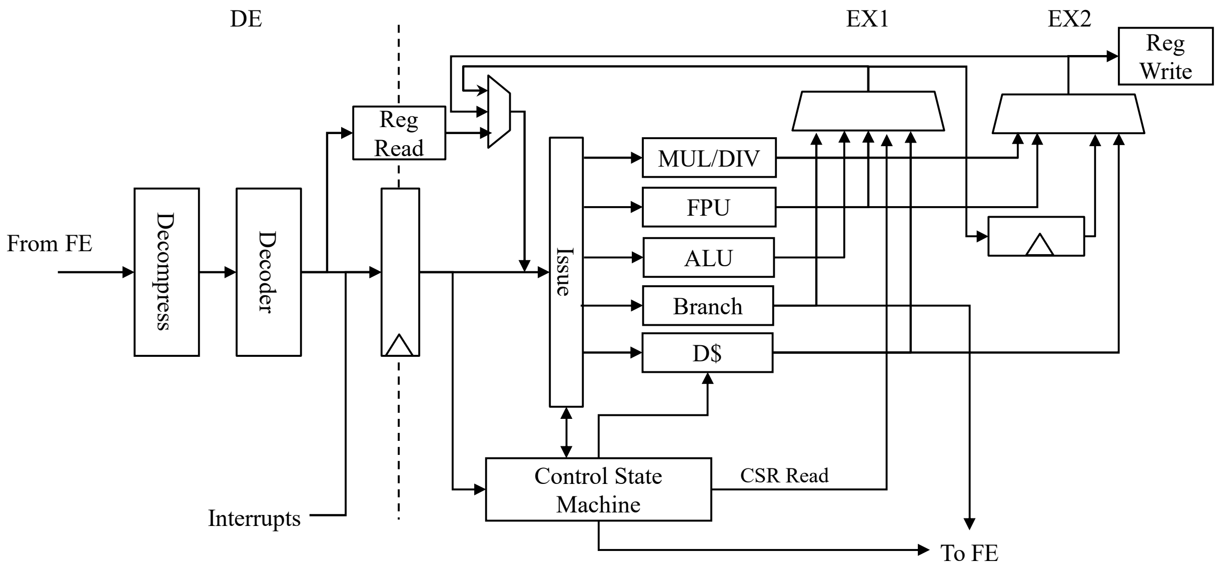

Figure 4: Muntjac backend design

is empty. After the control state machine deals with the instruction, it will inject the execution result (e.g. CSR value read) into the pipeline. This design ensures that all backend control signals are stable, so functional units do not have to latch control signals internally; global states can only change when the pipeline is empty.

Except for the data cache interface, no functional units nor the control state machine may generate exceptions. Insufficient privilege or illegal CSR accesses are all decoded as illegal instructions by the decoder. If a load or store instruction triggers an exception, all instructions in flight are cancelled by setting their destination register to x0. This simple cancellation process is made possible by the fact that all functional units other than the data cache are stateless and cannot generate exceptions, and that the control state machine is outside the pipeline.

Two implementation options are provided for the multiplier. A slower implementation that has a 17x17 multiplier and splits 32-bit multiplication into 3 narrow multiplications, and 64-bit multiplication into 10-16 narrow ones. A faster one uses a 33x33 multiplier and splits 64-bit multiplications into 3-4 narrow multiplications. 1 extra cycle is inserted on the input path for timing purposes. The division unit implemented performs 1-bit per cycle long division. 1 extra cycle each is inserted on the input and output path for timing. If a pipelined multiplier or multi-bit-per-cycle divider is needed it could easily be plugged into the existing interface.

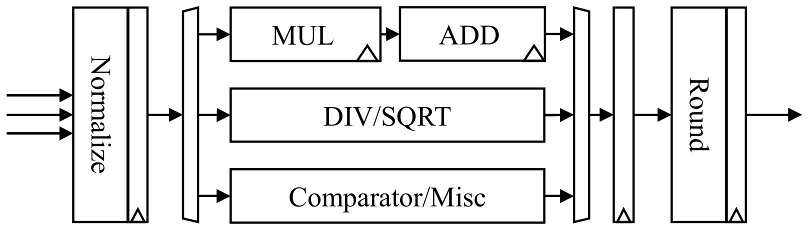

Figure 5: Muntjac FPU design

An FPU is available with Muntjac, supporting both single-precision and double-precision floating-point arithmetics. It is divided into 3 stages as shown in Figure 5. The “normalize” stage converts subnormal numbers into normalized form and decodes zero, infinity and NaNs. Single-precision floating-point numbers will be expanded to double-precision at this stage so arithmetic pipelines are unaware of the difference. There is a multiplication-and-add pipeline, which as its name implies, handles multiplication, add, and fused multiplication-and-add (FMA). Division and square root using long division are in the same pipeline. Another pipeline handles comparison, min-max and other miscellaneous operations. After the arithmetic pipeline, the result is fed into one of the single-precision, double-precision or integer rounding modules to convert from the internal normalized form back to IEEE format or to integers. The number of cycles depends on the operation and ranges between 3 and 59. Unlike many other implementations, we opted not to use a recoded format internally, so that NaN-boxing behaviour is more easily implemented and hard-to-test bugs related to recoding can be avoided.

We provide a few options for floating-point support:

- No FP support: No FP registers are supported, and FPU is not instantiated
- Full FP support: FP registers are supported, and FPU is instantiated
- FP register only: FP registers are supported, but FPU is not instantiated. In this mode, FP load and store operations are supported and handled by the hardware, but other FP operations will cause an illegal instruction exception. This mode can be used if the user only uses FP instructions occasionally on cold paths, wants the core to appear as RV64GC, but does not want to bear the cost of FPU. In this mode, M-mode firmware can trap and emulate the floating-point operations, but trapping is not needed when the operating system (OS) merely saves or restores FP contexts without performing actual computations.

It is also possible to have multiple cores sharing a single FP unit, but this is not yet implemented.

## 4 Protocol choice for cache coherency

In Muntjac, both instruction and data caches communicate with the pipeline via generic valid-ready interfaces, allowing flexibility in cache designs and their protocol choice. However we do want to provide a reference cache implementation which could suit general use cases.

Since cache coherency protocols have profound influences on cache designs and a lot of work is to be spent on implementing all necessary utilities and infrastructures, we thoroughly evaluated a range of cache coherency protocols to be used for the reference cache implementations. Candidates include a custom 3 channel protocol and open-standards such as AXI/ACE and TileLink.

### 4.1 AXI

AXI [1] is short for Advanced eXtensible Interface, and is one protocol out of the Advanced Microcontroller Bus Architecture (AMBA) family. The specification is developed and published by ARM. The specification is royalty-free, but it does come with a restriction that if a design uses AXI and includes a CPU, then the CPU should either be licensed from ARM, or must not be compatible with ARM ISA. Since Muntjac is not compatible with ARM ISA, the license restriction would not apply.

The latest version of the AXI specification when the project began was AXI 4, and the latest version of AXI as of the time of writing is AXI 5. There is a trimmed down version called AXI-lite for peripheral IOs that have low performance requirements.

AXI/AXI-lite have 5 channels: Read Address (AR), Read Response (R), Write Address (AW), Write Data (W) and Write Response (B). Each channel has its own valid and ready signals and handshaking happens upon assertion of both valid and ready signals. AXI requires the valid signal be constantly asserted until the handshake happens. AXI requires that no combinational path exists between input and output signals.

AXI has two basic transactions:

- Read: host sends address, size and length on AR channel, and device responds on R channel with data and status.
- Write: host sends address, size and length on AW channel, and sends data on W channel. The device responds on B channel with status.

Size is the size of a single burst, while length is the number of bursts minus 1. A "last" signal is required for channels that support multiple bursts (R/W in AXI). The separation of size and length leads to a possibility called narrow bursts, when the size is smaller than the native size given the width of data signals but the length is non-zero. Narrow bursts are not supported by all AXI implementations.

AXI also places AW and W in separate channels. In AXI 3, both AW and W channels carry transaction IDs, but it could be different to handle if addresses and data are not arriving in the same order. The ID of the W channel was subsequently removed in AXI 4 and reordering of write is forbidden.

AXI supports load-reserved store-conditional (LRSC) by exclusive accesses. Read request can have ARLOCK set, and the device can respond with EXOKAY if it supports exclusive accesses. A subsequent write request can have AWLOCK set, and if the memory has not been modified between the read and the write request, the write will be accepted by the device and an EXOKAY status is returned. Otherwise, the memory is not updated and OKAY is returned. Starting in AXI 5, AW channel has an additional atomic operation signal AWATOP; the supported atomic operations are a super set of atomic operations supported by RISC-V.

## 4.2 ACE

ACE, short for AXI coherence extensions, is also from the AMBA protocol family. ACE extends AXI with 5 additional channels: Snoop Address (AC), Snoop Response (CR), Snoop Data (CD), Read Acknowledgement (RACK) and Write Acknowledgement (WACK). AC, CR and CD are flow controlled using valid and ready signals like AXI channels, while RACK and WACK channels are one-way and have no ready signals.

A typical read transaction that needs invalidation makes use of AR, AC, CR, CD, R and RACK channels:

- Host issues a read transaction on AR channel.
- Device forwards the address to the AC channel.
- Other hosts respond on CR channel, optionally provide data on CD channel.
- If data is present, device forwards the data to R channel; otherwise the device handles the read transaction itself, e.g. serve request from the cache or forward to the next-level cache.
- Host completes the transaction on RACK channel.

Write-back or eviction makes use of AW, W, B and WACK channels:

- Host issues a write transaction on AW channel.
- If cache line is dirty, data is transferred on W channel.
- Device responds on B channel.
- Host completes the transaction on WACK channel.

In ACE a cache line can be in one of 5 states: Invalid (I), UniqueDirty (UD), SharedDirty (SD), UniqueClean (UC), SharedClean (SC), i.e. it is a MOESI protocol.

## 4.3 TileLink

TileLink [10] is a family of open-standard interconnect protocols originally designed by UC Berkeley Architecture Research group and subsequently SiFive. Unlike ACE's MOESI, TileLink follows MESI model. There are two permission levels in TileLink: N (None), B (Branch) and T (Trunk), some quite atypical naming differing from normal `None, Read, Read/Write` nomenclature.

In TileLink, caches are viewed as a tree with a single device as the root (e.g. LLC), L1 caches as the leaf and intermediary caches (if any) as the middle nodes. and For any particular cache line, all agents that contain cache copies of the cache line forms a subtree, known as *coherence tree* in TileLink. The point of write serialization is known as the `Tip`; a node on the path between the `Tip` and the root (including both) is called a `Trunk`, and children of the `Tip` are known as `Branch`es. For example, when L1 requests `Trunk` permission from L2, L2 requests `Trunk` from memory controller, then all agents have the cache line in the `Trunk` state. TileLink formally defines that only Trunk Tip with no Branches have write permission to the cache line to reflect the fact that only L1 can write the cache line in this case.

There are 5 channels for a TileLink link: A, B, C, D and E. Out of these channels, A, C and E flow from hosts to devices, while B and D flow from devices to hosts.

- A channel carries requests such as `Get` and `PutPartialData/PutFullData` message for uncached memory accesses, or `AcquireBlock/AcquirePerm` message for obtaining data and permission for a cache line.
- B channel carries `ProbeBlock/ProbePerm` messages to invalidate a cache line from hosts.
- C channel carries `Release/ReleaseData` voluntary cache line permission release (and writeback) messages, or `ProbeAck/ProbeAckData` messages in response to invalidation messages sent over the B channel.
- D channel carries `AccessAck/AccessAckData` messages in response to `Get/ PutPartialData/PutFullData` messages sent over A channel, or `Grant/GrantData` cache line permission grant messages in response to `AcquireBlock/AcquirePerm` messages sent over the A channel, or `ReleaseAck` messages in response to voluntary cache line release messages sent over the C channel.
- E channel carries `GrantAck` message in response to `Grant/GrantData` messages sent over the D channel.

If sorted according to causal order:

- Uncached read: host sends `Get` and device responds with `AccessAckData`.

- Uncached write: host sends `PutPartialData/PutFullData` and device responds with `AccessAck`.
- Increase cache line permission: host sends `AcquireBlock/AcquirePerm`, device responds with `Grant/GrantData` and host responds with `GrantAck`.
- Non-voluntary decrease cache line permission: device sends `ProbeBlock/ProbePerm`, host responds with `ProbeAck/ProbeAckData`.
- Voluntary decrease cache line permission: host sends `Release/ReleaseData` and device responds with `ReleaseAck`.

There are also atomic read-modify-write transactions and a hint/prefetch transaction. They are handled similarly to `Get` and `PutFullData`. Muntjac does not use these transactions so we will ignore them.

The channels are of ascending priority ($A < B < C < D < E$). Naturally, all responses are on a higher priority channel than the original request. If a link agent is waiting for the response on a channel, it must be able to process messages sent over higher-priority channels. For example, a host waiting the response of `AcquireBlock` must be able to process a `ProbeBlock` message. However, a host waiting for the response of `ReleaseData` would not have to process `ProbeBlock` until it has received `ReleaseAck`. The priority requirement ensures that if a message is not processed, there must be a higher priority message in process. With limited (5) priority levels, this property guarantees the deadlock freedom of the TileLink network.

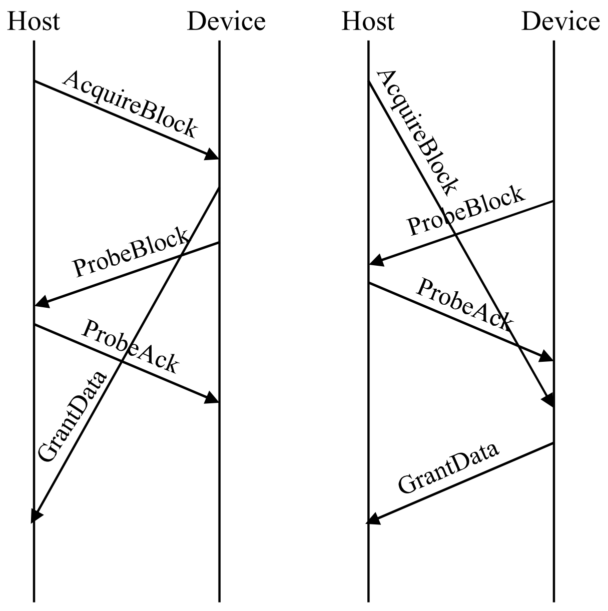

Figure 6: Ambiguous TileLink transaction if E channel is absent

TileLink does not guarantee any ordering of messages over channels. This relaxed requirement is the rationale behind the three-way handshake of cache line permission acquisition requests. If the E channel is absent, the transactions shown in Figure 6 are no different to the host; however in the left figure, the device would assume that the host no longer owns the cache line, while in the right figure, the device would assume that the host owns the cache line. This ambiguity is removed by requiring a `GrantAck` response to `Grant/GrantData` so the device would not send a `ProbeBlock` message until it has received the `GrantAck`.

There are three variants: TileLink Uncached Lightweight (TL-UL), TileLink Uncached Heavyweight (TL-UH) and TileLink Cached (TL-C). TL-C supports all messages. TL-UH supports `Get` and `PutPartialData/PutFullData` (and hint/atomics) with their corresponding response messages. TL-UL supports `Get` and `PutPartialData/PutFullData` (without hint/atomics) and does not support multi-cycle bursty transfers. Since TL-UL and TL-UH do not support any messages on channel B, C and E, they only need channel A and D.

## 4.4 Custom protocol

We have also considered a custom 3-channel protocol. It contains a `Req`, `Resp` and `Wb` channel. For simplicity we will use TileLink names for messages:

- Cache line permission is increased by sending out an `AcquireBlock` request on `Req` channel, and the device responds `GrantData` on the `Resp` channel.
- Voluntary release of cache line permissions is done by sending `Release` on the `Wb` channel and is not acknowledged.
- Invalidation is performed by the device sending a `ProbeBlock` on the `Resp` channel and the host responds with `ProbeAck` on the `Wb` channel.

This design is simplest and was used during the early prototyping stages. However the lack of acknowledge on `Wb` means that a reordering of `AcquireBlock` (on `Req` channel) and `Release` (on `Wb` channel) is problematic. Consider the scenario where a host sends out a `Release` followed by a `AcquireBlock` to the same address. Because the `Release` message does not require acknowledgement, if message on the `Wb` channel is delayed, the device may receive the `AcquireBlock` first followed by `Release`, different from the sending order. The host would assume that it owns the block while the device would mistakenly believe that the block is released, causing inconsistency.

The most straightforward fix is adding new channels, which brings us close to TileLink. While fixing the issue without adding new channels is possible, e.g. adding reordering constraints across multiple channels, we conclude that additional complication would make a custom protocol not worthwhile given that TileLink only requires 2 additional channels and has no reordering constraints.

## 4.5 Comparison

One difference between AXI/ACE is the specification of transaction sizes. In AXI, two fields are used: size and length. TileLink instead has a single size field that can be only used to specify sizes of powers of two: if the size is smaller than the bus data width, it is a narrow transaction; otherwise the transaction is bursty. Data width conversion is complicated in AXI due to the possibility of narrow bursts; in comparison it is very simple for TileLink, and data width downsizing could even be stateless. TileLink's power of two requirement however means that burst length is also power of two, so DMA devices are harder to implement compared to AXI because they have to divide accesses to aligned chunks.

Another significant difference between AXI/ACE and TileLink is the usage of IDs. In AXI, the same ID can be used for multiple in-flight transactions. Devices are required to respond to transactions of the same ID in FIFO order and reordering is forbidden. In TileLink, the ID must not be reused until the transaction has been completed. AXI's design simplifies the design of simple hosts by allowing them to issue multiple transactions under the same ID and process them one by one; crossbar implementations are more complicated if two transactions to different devices are under the same ID, since the crossbar has to block the second request (otherwise the response to host could be out-of-order). On the other hand, TileLink crossbars only need to arbitrate messages going towards the same channel. Without reordering requirements, TileLink crossbars can be stateless.

In AXI, control and data signals are in separate channels; in TileLink they are sent in the same channel, and control signals are held constant for all data beats for bursty transactions. TileLink's approach makes interconnects and devices simpler because control and data will not be reordered; the AXI 4's ditch of ID in W channel manifests the disadvantage of separate control and data signals. A minor disadvantage of TileLink's design is that FIFOs either have to store redundant control signals or have to store control and data signals separately despite being from a single channel, but we consider it to be insignificant.

| | AXI4-Lite | AXI5-Lite | AXI4 | AXI5 | ACE | TL-UL | TL-UH | TL-C |
|---|---|---|---|---|---|---|---|---|
| # of Channels | 5 | | | | 10 | 2 | | 5 |
| Data width | 32 or 64 | Up to 1024 | | | | Up to 4096 | | |
| Transaction Size | Full bus width | Up to bus width | | | | Up to bus width | Up to 4096 | |
| Burst Length | 1 | | Up to 256 | | | 1 | Length must be power of 2<br>Address must be aligned | |
| Control & Data | Separate | | | | | Combined | | |
| "Last" Signal | Always true | | Explicit | | | Always true | Implied | |
| Atomics | No | | LRSC Only | Yes | Yes | No | Atomics Only | Yes |
| IDs | Optional | Required, same ID indicates FIFO | | | | Required, same ID forbidden | | |

Table 2: Summary of cache coherency protocol differences

Table 2 summarises differences between the cache coherency protocol evaluated. AXI-Lite, TL-UL and TL-UH are mainly for device IOs and they are not suitable as the cache coherency protocol due to lack of atomic support needed to implement the entirety of RISC-V's A extension. We also ruled out the usage of bare AXI. While AXI 5 does support LRSC and atomic operations, the need of bus transactions for all atomic accesses is unfavourable. While traditionally it is believed that synchronisations are rare, the statement is no longer true given the rise of multi-threaded programs. To avoid data races, very fine-grained locking is becoming widely adopted, and their performance and overhead has been greatly improved by ParkingLot [8]. The Rust programming language [11] takes further steps to eliminate data races by ensuring all data shared between threads are protected by thread-safe reference counters and locks, all of which use atomic operations underneath. As a result of the fine granularity, the vast majority of atomic memory accesses are uncontended. The performance in such cases becomes critical, and the requirement of bus transactions for all atomic operations is undesirable.

Considering simplicity, number of channels, and the fact that Ibex [6] and OpenTitan [7] already makes use of TileLink, we ultimately selected TL-C over ACE for cache coherent links, TL-UH for uncached accesses and TL-UL for I/O memory access. TileLink specification includes provision for update-based protocols; Muntjac does not implement them. This removes the possibility of a bursty B channel, and saves 73 bits per 64-bit data width TL-C link.

One additional aspect worth mentioning is not about the intrinsic design of protocols, but their current adoptions. Many existing third party IPs use AXI. So for compatibility/interoperability reasons, a bridge needs to be feasible between the protocol chosen (TileLink) and AXI. Fortunately, TileLink to AXI bridge is fairly trivial; AXI to TileLink bridge requires splitting AXI transactions into chunks due to the address alignment requirements of TileLink, but we consider this burden to be acceptable. We have implemented these bridges to be used along with Muntjac.

## 5 Cache design

Muntjac provides cache implementations with interfaces that comply with the TileLink protocol. Both instruction and data caches communicate with the pipeline via generic valid-ready interfaces, so they have to manage the request replays themselves without help from the frontend/backend.

### 5.1 Data cache

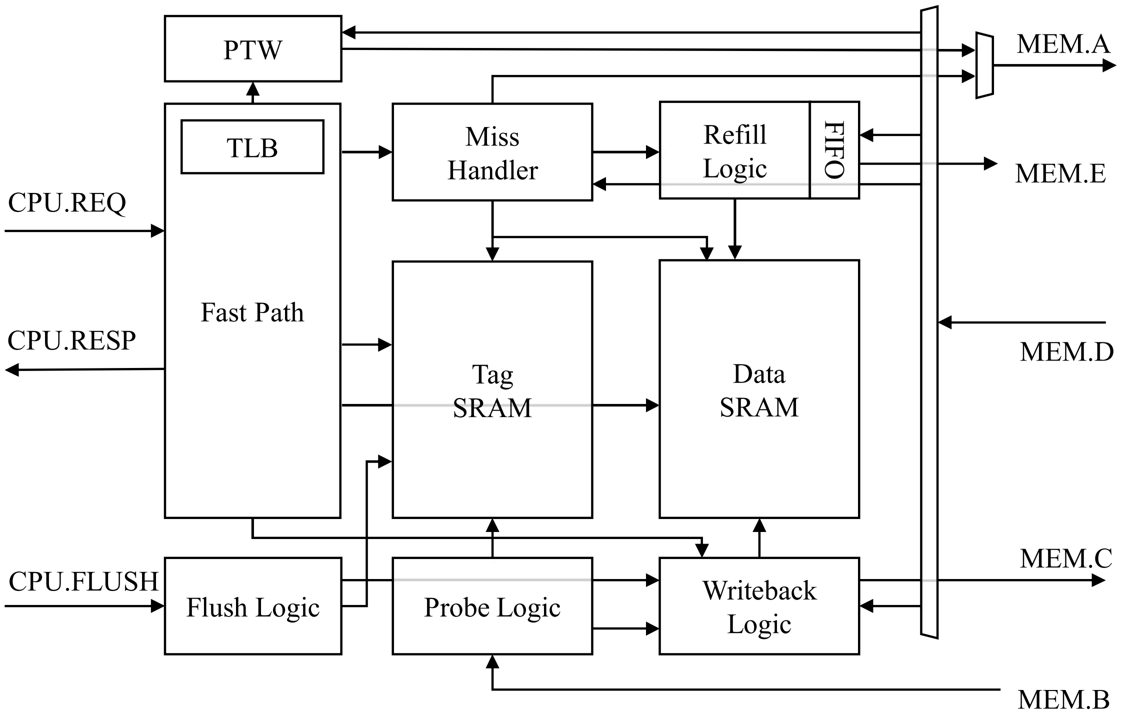

Figure 7: Muntjac L1 data cache subcomponents

Muntjac's data cache implements the TileLink Cached (TL-C) protocol for its memory-facing interface and is therefore multi-core-capable. It is organised into several state machines as shown in Figure 7. The cache is set-associative, and for each way, tags and data are stored in separate single-port SRAMs. Our initial implementation used simple dual-port (1-read 1-write) SRAMs, but benchmarks showed only a small performance difference, so Muntjac switches to single-port SRAMs for less transistor usage. Data SRAMs are 64-bit wide and do not have byte masks for writes.

When an access request arrives from the backend (via `CPU.REQ`), a parallel lookup to all of the tag SRAM, data SRAM and TLB take place as shown in Figure 8. The physical address translated by the TLB is used for tag comparison (i.e. this is a physically-indexed physically-tagged (PIPT) cache). For a load or atomic fetch-and-update request, if everything hits, the data word fetched and multiplexed will be aligned if the access is narrow (less than 64-bit) and returned to the pipeline.

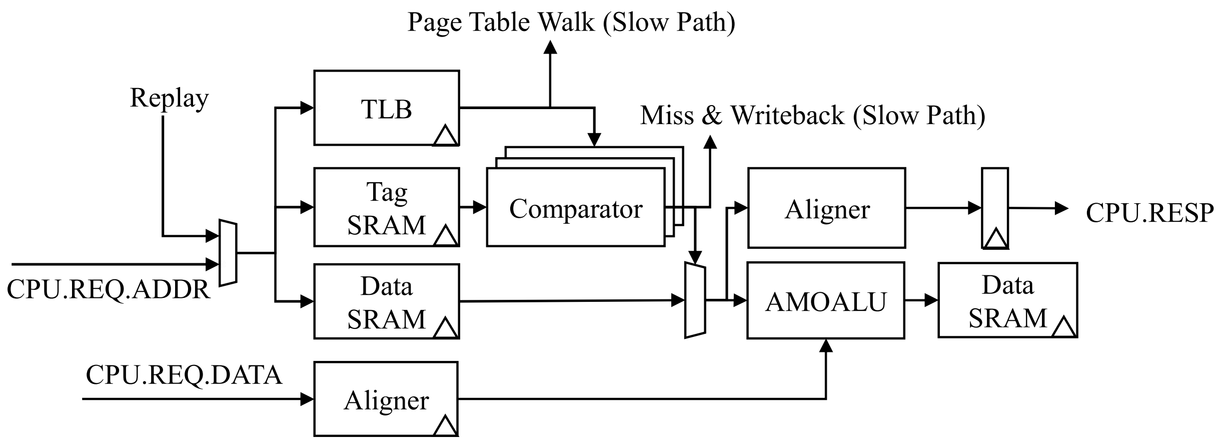

Figure 8: Muntjac L1 data cache fast path

The cache has its own ALU, the AMOALU, for atomic operations and narrow stores. For a store or an atomic fetch-and-update request, the supplied data from the pipeline will first be aligned, i.e. have the bytes shifted to their position in the containing 64-bit word for a narrow store. The current value from the data SRAM and the aligned data will then enter AMOALU, recombined with the current value for the inactive bytes before being written back into the data SRAM. Plain stores are implemented similar to an AMOSWAP: while no computation is performed, AMOALU is still used for byte recombination. Recombining bytes is preferred to using byte masks on SRAMs to ease future implementation of ECC.

The SRAM accesses are arbitrated between all subcomponents of the cache, so the SRAM lookups might be blocked when another subcomponent (e.g. the probe logic) is accessing it, in which case the request will be replayed. Similarly, a TLB miss will trigger a page table walk and TLB refill, and the request will be replayed afterwards. If the accessed cache line is nonexistent in the cache, the fast path will trigger the miss handler, and optionally trigger the writeback logic if eviction is needed for a replacement.

The fast path only interacts with the SRAMs and all TileLink protocol details are handled in other subcomponents. To avoid SRAM access races, refill, probe, flush and writeback logic are mutually exclusive and at most one of them can be active at a time. Upon a cache miss, the miss handler will send a TileLink A-channel `AcquireBlock` message. The corresponding D-channel `Grant`/`GrantData` response is processed by the refill logic, which updates the tag and data SRAM and sends back `GrantAck` message over channel E. Because writeback logic (C-channel) might be active when the response is received and TileLink mandates D-channel messages takes priority over the C-channel, a FIFO is needed for the refill logic. I/O accesses are not specially treated by the fast path; they are treated as cache misses and the miss handler will still send out an `AcquireBlock` message. If the address space accessed is I/O, the request will be responded with a denied `Grant` message, and then the miss handler will retry the request using `Get` or `PutPartialData` uncached access.

Writeback logic is responsible for both voluntary releases (e.g. eviction or a flush) or for responding to probes initiated by the next-level cache. In the case of a voluntary release, writeback will issue `Release`/`ReleaseData` and wait for `ReleaseAck`. Probe logic is activated when a `ProbeBlock` message is received over the TileLink B-channel, and it will look up the tag and leave the task of sending `ProbeAck` or `ProbeAckData` message to the writeback logic. The probe logic is required even in a single core configuration, e.g. to keep instruction cache and page table walkers coherent.

RISC-V has load-linked/store-conditional (LL/SC) instructions, which may cause live lock in a multi-core setup; a forward progress requirement is thus included in the specification. In Muntjac, when a cache access misses and the cache line is refilled, a 16-cycle timer starts ticking. Within 16 cycles, any `ProbeBlock` request to the cache line is blocked. This invalidation protection is lifted when a memory store has been made, another cache miss happens or when the timer expires, whichever happens first.

Muntjac's data cache, as it is currently implemented, is sequential consistent, and thus all fences are treated as no-ops. This is means that Muntjac is currently compliant with both RISC-V weak memory model (RVWMO) and RISC-V total store order (RVTSO). However, it should be noted that the cache is implemented in this way solely due to simplicity rather than a deliberate design decision. Future changes and improvements could weaken it so that it is no longer compliant with RVTSO. RVWMO compliance is however guaranteed as mandated by relevant RISC-V standards.

## 5.2 Instruction cache

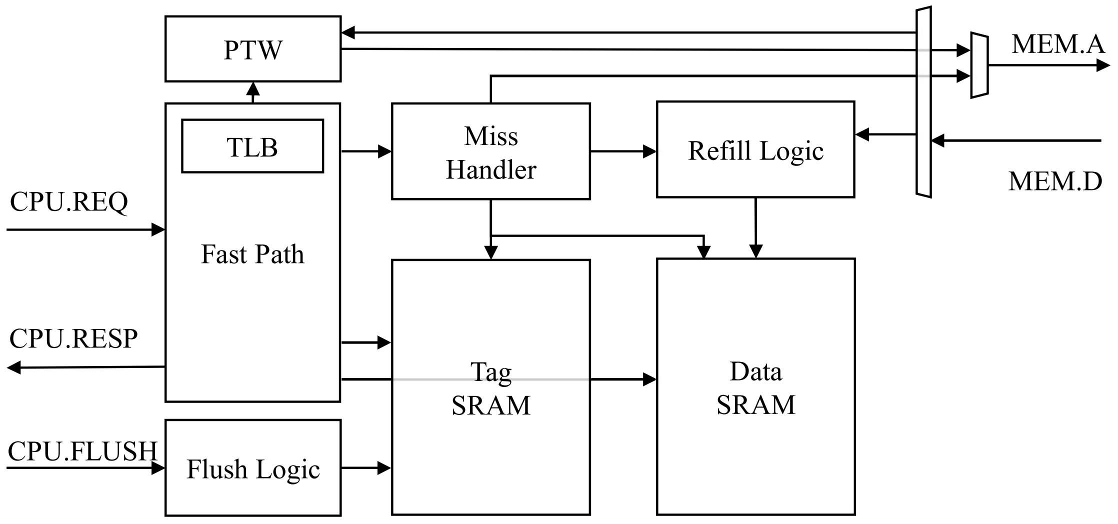

Figure 9: Muntjac L1 instruction cache subcomponents

Muntjac's instruction cache is derived from the data cache but with unnecessary features removed. The high-level structure is portrayed in Figure 9. As a derivative, the overall design and operation are similar to those of data cache.

In RISC-V, an explicit FENCE.I instruction is necessary to ensure modified code is visible to instruction fetch. Therefore, we implemented a non-coherent version of the instruction cache; it will be fully flushed when a FENCE.I or SFENCE.VMA instruction is executed. All cache coherence related logic is removed from this instruction cache, and the TileLink variant it conforms to is TileLink Uncached Heavyweight (TL-UH). When the cache misses, a bursty `Get` request is sent instead of `AcquireBlock`. A failed `Get` request will cause an instruction access fault immediately rather than initiating a retry as an I/O memory access. If needed, a coherent instruction cache that communicates via TL-C is also available.

Write support is not needed in the instruction cache which simplifies the fast path compared to the data cache, shown in Figure 10. Also, unlike the data cache which needs to support variable size accesses, the instruction cache supports only 32-bit access; the alignment stage is therefore also removed from the instruction cache (note that the compressed instruction handling is performed entirely in the frontend). This narrower access also allows us to reduce the data SRAM's width to 32-bits.

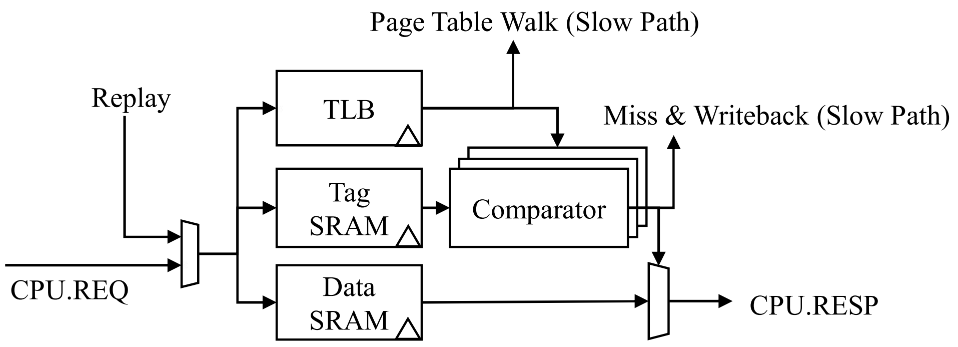

Figure 10: Muntjac L1 instruction cache fast path

## 5.3 SRAM word interleaving

The native word width of the data cache's data SRAM is 64-bit, and 32-bit for the instruction cache's data SRAM. However, the bus width is often configured to be wider than 64-bit to improve bandwidth and reduce latency caused by a high number of beats in a bursty transaction. Adding a TileLink data width converter could solve the data width discrepancy, but the performance is not optimal.

In Muntjac's set associative cache design, data SRAMs are already banked; each set needs its own data SRAM bank to allow parallel lookup for cache access. For example, a 64-bit 4-way data cache therefore has a SRAM bandwidth of 256 bits/cycle for lookup. An interleaving technique would allow all the bandwidth to be utilised for refilling as well.

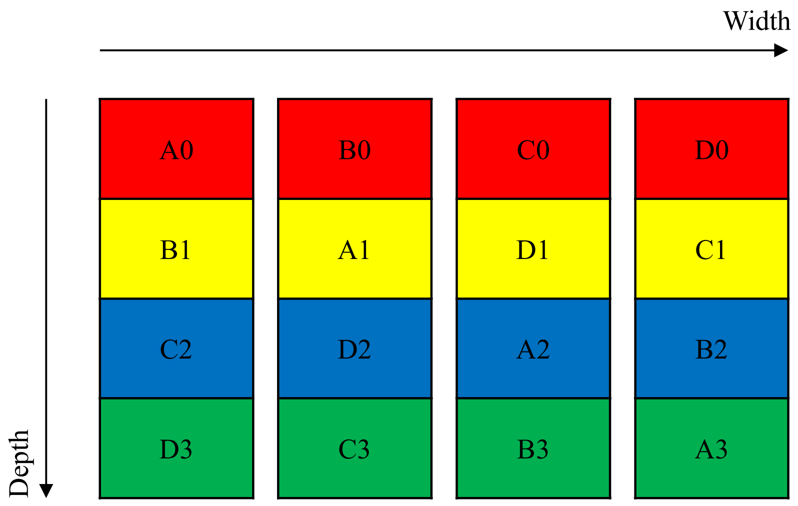

Figure 11: Parallel data SRAM lookup across multiple ways, given a fixed offset

A naive design would have each data SRAM bank storing data for a specific way, and data in the same cache line are indexed by their offsets within the cache line. With interleaving, words in the same cache line may not be stored in the same bank of data SRAM. The index and way number used to access a word depends on both its offset within the cache line and the way number of the cache line.

Figure 11 and Figure 12 illustrates the interleaving mechanism and how data access is performed with this scheme with a hypothetical 4-way cache where each cache line contains 4 words. ABCD indicates the way number of a cache line, and 0123 indicates the offset within the cache line. A0-3 therefore are the same cache line, but it can be observed that they are spread out in different banks.

Cells with the same colour are simultaneously accessed in the same cycle. Figure 11 demonstrates a fast-path read operation that must simultaneously read all ways of a

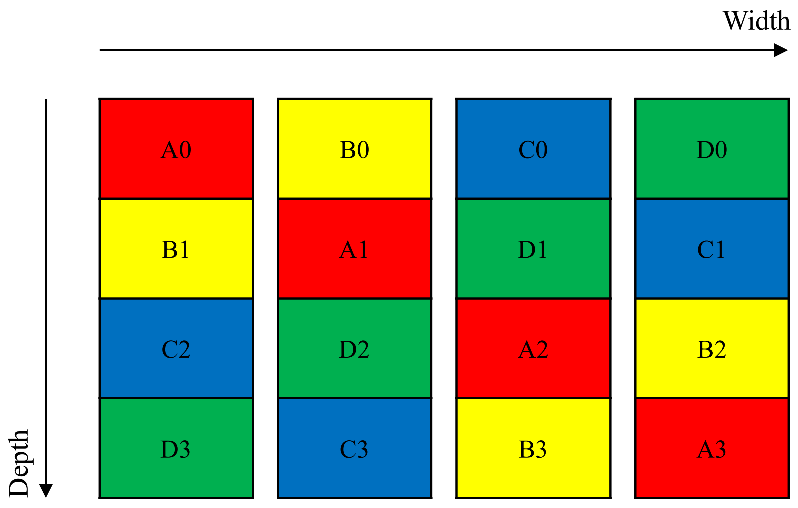

Figure 12: Wide data SRAM access of multiple words within the same cache line

particular set in parallel. The same index is used for SRAMs in each way, so words of the same offset are fetched for all ways (all cells sharing the same colour have the same offset). Figure 12 on the other hand reflects the scenario where wide data access is needed for writeback or refilling. A different index is used for each SRAM, so that all fetched words belong to the same cache line (all cells sharing the same colour are in the same way).

This interleaving scheme reduces the number of cycles needed for writeback and refilling, expanding the maximum bus width from 64 bits to 256 bits for a 4-way set-associative data cache without the need to expand the bit width of each SRAM or adding TileLink adapters. Similarly, a 4-way set associative instruction cache supports a maximum bus width of 128 bits.

## 6 Multicore support

Both data cache and instruction cache have a memory-facing TileLink link, and they are aggregated as shown in Figure 2. The aggregation is performed by a passive stateless 2:1 multiplexer that only switches messages based on source IDs. Currently 4 IDs are used per core: the data cache and instruction cache are allocated with one ID each, and their page table walkers are each allocated with a distinct ID.

The instruction cache and data cache are considered as distinct hosts in the TileLink network, because they are not inherently coherent to each other; when an instruction cache requests a dirty cache line that resides in the data cache of the same core, the dirty data should be written back and used, instead of the stale data in the next-level cache or the main memory. Similarly, the page table walker for the data TLB is considered a distinct host from the data cache itself, so that up-to-date page table entries are used.

These requirements means that each core consists of multiple hosts from the TileLink network's perspective. Therefore, there are no fundamental differences between a single-core Muntjac system and a multi-core Muntjac system. A broadcaster or an L2 cache will always be necessary for correct operation.

In a typical system like Figure 1, TileLink links from each core and possibly DMA-capable devices will be further aggregated with a stateless m:1 multiplexer; the link will then be split into multiple links with a stateless 1:n demultiplexer depending on address ranges and sink IDs. The handling will differ depending on the properties of the address spaces:

- IO: all caching `AcquireBlock` requests are denied. `Get/PutPartialData` are allowed and further demultiplexed to core local interrupt (CLINT) controller, platform-level interrupt controller (PLIC) or devices. Interrupt controllers and devices follow TileLink Uncached Lightweight (TL-UL), which does not support bursty transactions.
- ROM: `AcquireBlock` messages that request `Trunk` (read-write) permission are denied. `PutPartialData` requests are denied. `Get` is allowed, and read-only `AcquireBlock` requests are converted into `Get` requests by a "TL-C ROM terminator" component.
- Memory: unlike IO and ROM, memory-like addresses can both be cached and modified. A broadcaster or an L2 cache is required as previously mentioned. Ultimately, after potentially multiple levels of caches, it would be connected to a memory controller that talks TL-UH.

## 6.1 Broadcaster

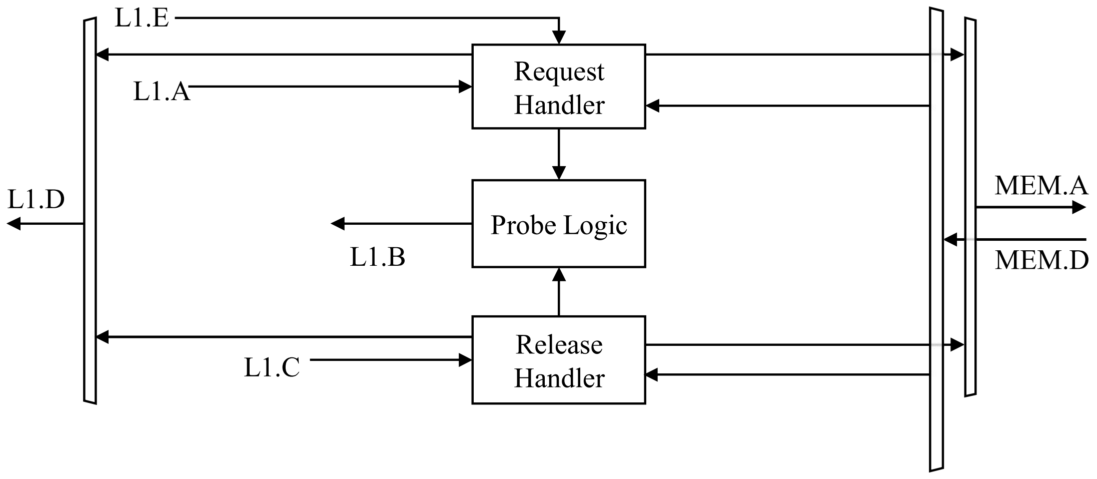

Figure 13: Simple broadcasting bus to bridge TL-C multiple hosts to a TL-UH port

For 1 or 2 cores, a broadcaster like Figure 13 is simplest and requires no additional SRAMs. For each incoming request on the L1's A channel, the broadcaster will send out a `ProbeBlock` to all caching hosts except the initiator. The message will be converted to `Get` only after all `ProbeAck` messages are received. `ProbeAckData` and `ReleaseData` are converted to `PutFullData` while `Release` is responded by the broadcaster directly with `ReleaseAck`.

A broadcaster will generate probing traffic to all cores for all requests even if these cores are not using the cache line. This will hurt the performance of L1 caches. For multi-core systems, it is recommended that an L2 cache is used.

## 6.2 L2 cache

Muntjac provides a reference L2 cache design. This cache uses TL-C for both the CPU-facing link and memory-facing link, so despite its name, it could be cascaded and used as L3 caches as well. This cache design can also be used as individual banks of a larger cache, multiplexed by address.

When the cache is used as a last-level cache, a "RAM terminator" is provided which converts TL-C protocol to TL-UH (this is allowed for the L2 cache because there is only a single host (the cache) while for cores there are multiple hosts). Bridge IPs are provided to further convert it to AXI.

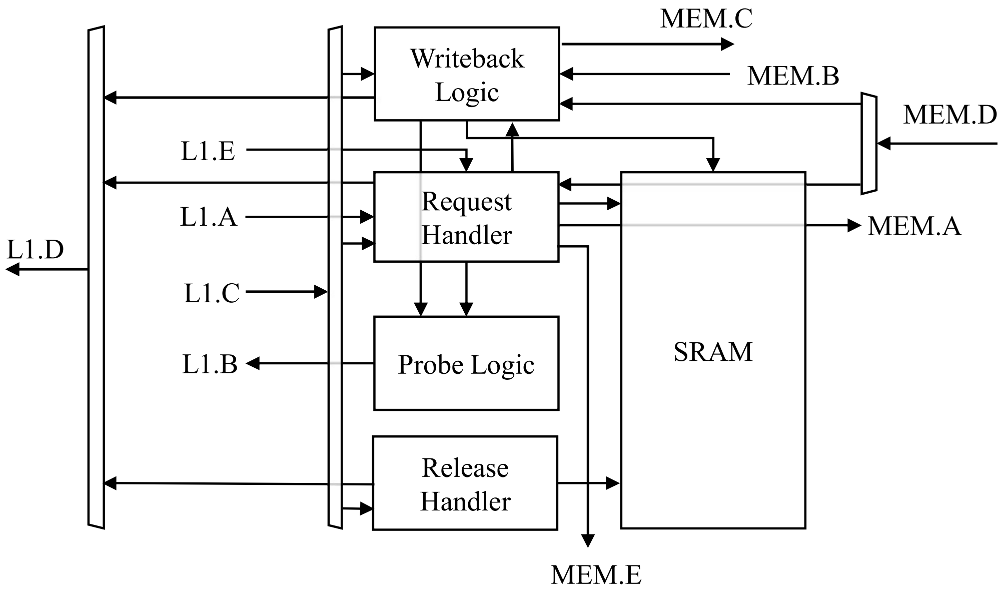

Figure 14: Muntjac L2 cache design

The cache consists of SRAMs for tags and data, probe logic, writeback logic and handlers for requests and releases, as shown in Figure 14. Multiple copies of request and release handlers can exist for parallelism, allowing messages for different addresses to be handled differently. Typically the number for each type of handler matches the number of cores. It should be noted that to ensure correctness, only one of the logic can be active for a particular address.

The probe logic is a very simple sequencer. A cache line may be shared by multiple L1 caches, so invalidation could potentially be multicast. TileLink is a unicasting protocol, so a sequencer is needed to separate the multicasting invalidation to a number of unicasting `ProbeBlock` messages.

The request handler, as its name suggests, handles request messages like `Get` or `AcquireBlock` from A channel. If cache access hits and no probing is necessary, the response gets sent back on the D channel. For `Grant`/`GrantData` responses, recipient `GrantAck` on E channel will bring the request handler to idle state for new requests. The request handler is also responsible for refilling the cache if the cache access misses. To reduce latency, the refilled data is simultaneously forwarded back to the L1 and written back to the data SRAM. If cache line eviction is necessary before refilling, probing of the evicted cache line and dirty data writeback is delegated to the writeback logic.

When probing is necessary, the request handler sends out the probe via probe logic, and is expected to handle the `ProbeAck`/`ProbeAckData` itself, as well as `Release`/`ReleaseData` messages to the same address. This requirement is essential for the deadlock-free operation of TileLink: another core may issue a `Release` to the same cache line concurrently as the current one in progress by the request handler; the release handler cannot operate on the same cache line to avoid conflicts. Whenever a C channel message with data is received, the request handler simultaneously writes it back to the SRAM and forwards it to the requester of the original initiating A channel message. The handler will read data from SRAM and respond to the requester if none of the C channel messages contain data.

The writeback logic is responsible for both eviction and probes from next-level caches. Similar to the request handler, it sends out the probe via probe logic and handles all C channel messages to the cache line of concern. Though, in this case, data are forwarded as

ReleaseData or ProbeAckData to the memory-facing C channel rather than CPU-facing D channel.

Because the writeback and request logic is responsible for C channel messages for their cache lines, the release handler is very simple. It only ever deals with Release/ReleaseData messages. It simply writes the data back to the SRAM if necessary and responds with ReleaseAck.

## 7 Utilisation and performance

| **Instruction Type** | **Stall Cycles** | **Notes** |
|---|---|---|
| Integer Arithmetic | 0 | No RAW data hazard |
| Load | 0 | |
| Store/Atomics | 0/1 | 1 cycle stall if succeeded immediately by a load or atomic |
| Integer Multiplication | 3/10/16 (Slow Multiplier)<br>1/3/4 (Fast Multiplier) | Numbers in MULW/MUL/MULH order |
| Integer Division | 33/65 | 33 for DIVW/REMU, 65 for DIV/REM |
| FP Arithmetic | 4 | Same for FADD/FSUB/FMUL/FMA |
| FP Division/Sqrt | 57 | No separate treatment for single precision |
| FP Bitcasts/Classification | 0 | |
| FP Comparison | 1 | |
| FP Misc | 2 | Includes sign manipulation, conversion |
| Jump/Branch (Predicted)<br>Jump/Branch (Mispredicted) | 0/1<br>3/4 | +1 cycle stall if jump target is a misaligned 4-byte instruction. |
| CSR Access | 2 | Pipeline is flushed before access happens. |
| FENCE.I/SFENCE.VMA/ERET<br>Traps<br>Interrupts | 7/8 | Implemented as a pipeline flush followed by a redirection (similar to a mispredicted jump/branch). |

Table 3: Number of stall-cycles for different instructions

Table 3 lists the approximate number of stall cycles an instruction will cause. All instruction and data caches accesses are assumed to hit. As described in Section 3.2, Muntjac backend has two execution stages and therefore can usually hide the latency of two cycles instructions. For example, memory loads or bitcasting a float to integer would create no stalls. It should be noted though these are still two cycle instructions, so if the succeeding instruction needs to use their output register, 1 cycle stall will be induced due to read-after-write (RAW) data hazard. Only integer instructions from the base instruction set (RV64I) is single-cycle.

For the Dhrystone and CoreMark benchmarks, we use the default configuration parameters. The L1 data and instruction cache are 16 KiB each, 4-way associative. The L1 data and instruction TLBs are 32 entry each and 4-way associative. The L2 cache is 64KiB/core in size and also 4-way associative. The compiler being used is GCC 9.2.0, with optimisations enabled and some jump target alignment
(-falign-functions=4 -falign-jumps=4 -falign-loops=4). Muntjac achieves Dhrystone score of 2.17 DMIPS/MHz and CoreMark score of 3.01 CoreMark/MHz.

While Muntjac aims to be both FPGA and ASIC friendly, and does not employ FPGA-specific optimisations, we do test it on FPGA hardware extensively. We tested Muntjac on Digilent Genesys 2 board (with Xilinx Kintex 7) and Nexys A7 board (with Xilinx Artix 7). The result for Xilinx Kintex 7 is shown in Table 4. Synplify is used for its better ability to perform retiming compared to Vivado. With these utilisations Genesys 2 board can comfortably fit a 4-core system and Nexys A7 a 2-core system with full FPU. Table 4 also shows that the option to turn on FP registers but not FPUs has very limited overhead comparing to turning the floating point support fully off, making this option a good choice for running Linux kernel and userspace compiled for RV64GC target.

|  | **Frequency** | **Pipeline** |  | **Core** |  |
|---|---|---|---|---|---|
|  | (MHz) | LUTs | Registers | LUTs | Registers |
| FPU on | 89 | 13663 | 4538 | 17420 | 6683 |
| FPU off | 97 | 6022 | 3098 | 9902 | 5266 |
| FPU off, FP registers on | 95 | 6111 | 3121 | 9924 | 5281 |

Table 4: Synthesis result for Xilinx Kintex 7 with Synopsys Synplify

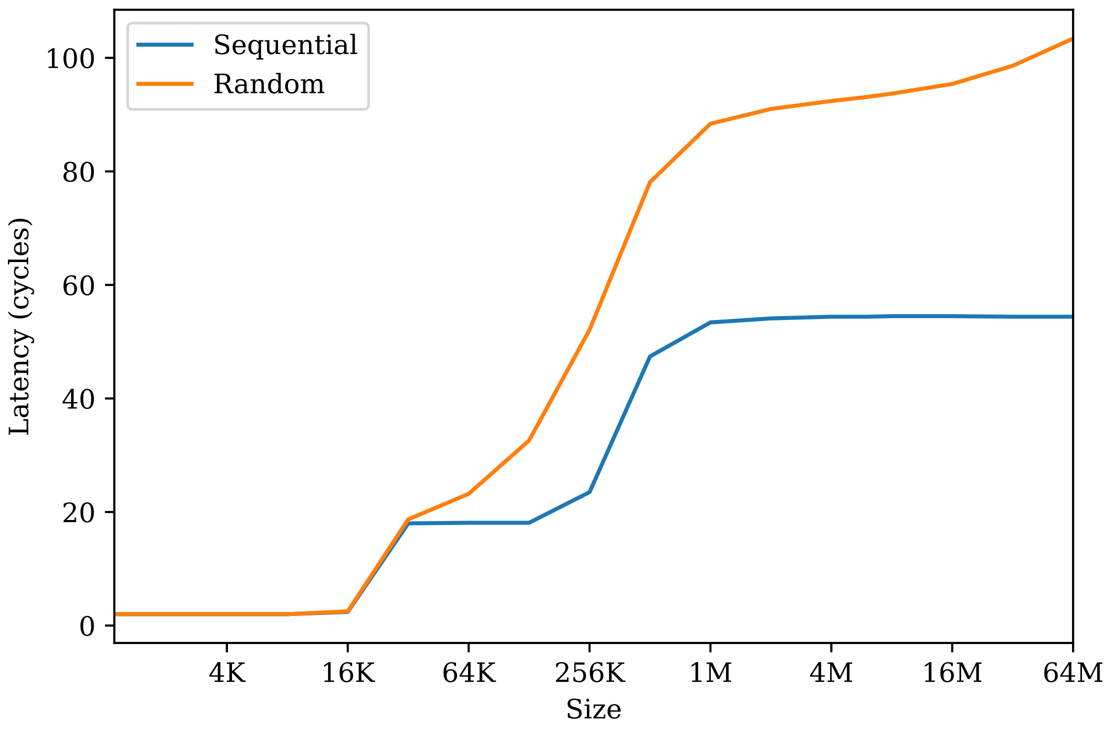

Figure 15: Memory access latency vs working set size

Figure 15 shows the memory access latency plotted against size of working set, with both result of linear and random access shown. The test is carried out using Linus Torvalds' test-tlb tools [12]. The test is performed on a Genesys 2 board with 4 cores instantiated with default parameters, so has a total L2 of 256KiB. The DDR3 memory on the development board is used, with an approximate latency of 30 cycles per cache line.

We have also measured the core-to-core communication latency of a multicore Muntjac system, using a tiny benchmark tool [4] that creates two spin-based semaphores and let two threads up and down them alternatively. The latency between a core `UP` the semaphore and another core `DOWN` it is measured to be around 44 cycles. Two bus transactions are involved in such operation, we estimate each transaction to take 20 cycles (4 bus messages, `AcquireBlock`/`ProbeBlock`/`ReleaseData`/`GrantData` plus a few cycles delay in the L1 and the L2 responding to messages). This is similar to the latency of a L1 miss but L2 hit indicated in Figure 15.

## 8 Verification

We have put considerable effort into verification given its importance for encouraging adoption of the IP and growing its user base. Our hope is that this could make Muntjac a capable starting point for further extensions by simplifies the process of validating any modifications.

We have broken our verification efforts down into two main areas:

- Core: ensure Muntjac adheres to the RISC-V specification
- Memory system/TileLink: ensure the memory system behaves in a consistent way and that our TileLink IP adheres to the TileLink specification

All testing is heavily scripted to allow both automation and a fast route to useful output for new users. Further details on each test suite can be found in the `test` directory of the Muntjac source repository.

As well as these regression tests, we also regularly run ad-hoc system tests, which involve tasks such as booting Linux (with Debian userspace) and running parallel benchmarks like PARSEC [2].

### 8.1 Core & pipeline testing

Testing of the core and pipeline aims to ensure that Muntjac is able to execute RISC-V programs in a way that is consistent with the RISC-V specification. The standards supported are detailed in Table 1.

- `riscv-tests` [9]: ensure that we meet RISC-V ISA specifications
- `riscv-dv` [3]: random test generation to improve test coverage

We run all tests on both an isolated Muntjac pipeline, and a whole core (pipeline + L1/L2 caches). This allows us to narrow down the source of issues (Table 5), and also gives us more control over cache latencies, allowing us to exercise more internal states of the pipeline.

With these tests, we have been able to achieve the following coverage.

| | **Pass on core** | **Fail on core** |
|---|---|---|
| **Pass on pipeline** | All good | Bug in memory system |
| **Fail on pipeline** | Bug in pipeline testbench | Bug in pipeline or shared testbench infrastructure |

Table 5: Verification results and their likely meanings.

| **Component** | **Line coverage** |
|---|---|
| Pipeline | 93% |
| Core (excluding pipeline) | 90% |

## 8.2 TileLink testing

Muntjac uses TileLink 1.8 [10] as the on-chip communication protocol between all of the caches in the memory system. A cache coherence protocol runs on top of TileLink, but is orthogonal to it.

For each of the TileLink protocol variants (TL-UL, TL-UH, TL-C), we provide:

- A module which monitors a TileLink port and checks that all communication adheres to the TileLink specification.
- A collection of functional coverage points, enumerating all of the states we want to see when testing (Table 6).

| **Scope** | **Coverpoints** |
|---|---|
| Each channel (A, B, C, D, E) | Two messages were sent with/without a pause between them<br>A valid message was/was not accepted by the recipient<br>The corrupt/denied bits were used (if appropriate) |
| Each field (`A.address`, `D.data`, etc.) | The value changed/stayed the same while waiting for a message to be accepted<br>Consecutive messages used the same/different values |

Table 6: Coverpoints used by Muntjac's TileLink verification.

We also provide a TileLink traffic generator capable of generating (random) valid requests and responses. Testing may then proceed in one of two ways. First, by simulating an isolated TileLink network and pushing random valid traffic through it, we can determine whether the TileLink components are configured properly and behave as expected. Second, by enabling assertions in a standard Muntjac system and running software on it, we can ensure that all components which interact with the TileLink network respect the relevant protocols and produce valid requests/responses.

The coverage achieved through this process is outlined in Table 7. We do not quote functional coverage figures since not every coverpoint applies to every component (e.g. coverpoints concerning message reordering are not relevant for components which require/ensure FIFO ordering).

| Component | Line coverage |
|---|---|
| `tl_adapter` | N/A |
| `tl_broadcast` | 100% |
| `tl_data_downsizer` | 100% |
| `tl_data_upsizer` | 100% |
| `tl_fifo_async` | N/A |
| `tl_fifo_converter` | 100% |
| `tl_fifo_sync` | N/A |
| `tl_io_terminator` | 100% |
| `tl_ram_terminator` | 99% |
| `tl_regslice` | 100% |
| `tl_rom_terminator` | 96% |
| `tl_sink_upsizer` | 100% |
| `tl_size_downsizer` | 100% |
| `tl_socket_1n` | 100% |
| `tl_socket_m1` | 100% |
| `tl_source_downsizer` | 100% |
| `tl_source_shifter` | N/A |

Table 7: Test coverage of Muntjac's TileLink IP. Coverage is N/A when there are no coverpoints in the component (e.g. all work is delegated to subcomponents).

## References

[1] ARM. AMBA AXI and ACE Protocol Specification. `https://developer.arm.com/documentation/ihi0022/hc`, 2021. Accessed: 2021-09-22.

[2] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[3] Google. RISCV-DV: open-source instruction generator for RISC-V processor verification. `https://github.com/google/riscv-dv`, 2021. Accessed: 2021-11-08.

[4] Xuan Guo. InterCoreBench: Intercore performance benchmark. `https://github.com/nbdd0121/InterCoreBench`, 2021. Accessed: 2021-04-05.

[5] Xuan Guo and Daniel Bates. Muntjac RISC-V core. `https://github.com/lowRISC/muntjac`, 2022. Accessed: 2022-04-23.

[6] lowRISC. Ibex RISC-V core. `https://github.com/lowRISC/ibex`, 2021. Accessed: 2021-09-24.

[7] lowRISC. OpenTitan bus specification. `https://docs.opentitan.org/hw/ip/tlul/doc/`, 2021. Accessed: 2021-09-24.

[8] Filip Pizlo. Locking in WebKit. `https://webkit.org/blog/6161/locking-in-webkit/`, 2016. Accessed: 2021-09-22.

[9] RISC-V International. riscv-tests: unit tests for RISC-V processors. `https://github.com/riscv-software-src/riscv-tests`, 2021. Accessed: 2021-11-08.

[10] SiFive. TileLink specification 1.8.1. https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf, 2020. Accessed: 2021-09-24.

[11] The Rust Project Developers. The Rust programming language. https://github.com/rust-lang/rust, 2021. Accessed: 2021-09-22.

[12] Linus Torvalds. test-tlb: Stupid memory latency and tlb tester. https://github.com/torvalds/test-tlb, 2018. Accessed: 2022-05-05.