Aquila-MP: A RISC-V Multi-Core SoC with Level-2 Cache Coherence

Tzu-Chen Yang, Ye Chen, Department of Computer Science, National Yang Ming Chiao Tung University

***Abstract*—In this project, we present Aquila-MP, a multi-core system-on-chip featuring a 32-bit RISC-V processor with the MESI cache coherence protocol for maintaining data consistency across L1 and L2 caches in a shared memory system. The processor supports atomic instructions for efficient synchronization and mutual exclusion in multicore environments. Achieved a 7.27× speed-up on MNIST (MLP) and a 7.53× speed-up on parallel matrix multiplication using an 8-core configuration, compared to a single-core processor. Aquila-MP operates at 50 MHz on a Xilinx Kintex-7 FPGA.**

***Index Terms*—RISC-V, multi-core processors, two-layer cache, MESI cache coherence protocol**

I. Introduction

A. Background and Motivation

Since its introduction by UC Berkeley in 2011, the open-source RISC-V ISA [1] has gained traction in both academia and industry, enabling flexible and customizable processor designs. Notable implementations like Rocket Core [2] and Ariane [3] have demonstrated its potential across applications ranging from embedded systems to high-performance computing.

As RISC-V adoption grows, the demand for multicore architectures in SoCs has increased. By integrating multiple cores, SoCs improve computational performance, parallel processing, and scalability. However, ensuring data consistency through efficient cache coherence mechanisms is essential to avoid computational errors.

A well-designed cache hierarchy is key to maximizing multicore performance. L1 cache provides fast access for frequently used data, while L2 cache offers larger capacity at the cost of higher latency. [4] Optimizing L2 cache structure and replacement policies helps balance speed and efficiency in RISC-V-based multicore systems.

B. Project Overview and Contribution

Aquila-MP is a multi-core system-on-chip featuring a 32-bit RISC-V processor with the MESI cache coherence protocol to ensure data consistency across L1 and L2 caches in a shared memory system. Designed for clarity and verifiability, it serves as a versatile platform for a wide range of applications.

Expanding on these principles, Aquila-MP enhances performance and usability, making it a versatile platform for research and real-world applications.

Key features include:

- **Cache Coherence Management** – Implements the MESI protocol to ensure L1 and L2 cache consistency, reducing memory access overhead.
- **Atomic Extension for Multicore Processing** – Supports RISC-V atomic instructions for efficient parallel execution and synchronization.
- **Flexible Hardware Verification** – Tested on a Xilinx Arty and kintex FPGA for reliable real-world performance.
- **Scalable and Extensible Architecture** – Designed for easy modification and expansion in multicore research.

C. Report Organization

The report is organized as follows: Section II reviews related work on quad-core systems and cache coherence protocols. Section III covers the design of the Aquila-Quad system architecture. Section IV and Section V details the implementation of the MESI cache coherence protocol, including the coherence unit, data cache, and level-2 cache. Section VI discusses the ”A” standard extension instruction. Section 6 presents experimental results, including test programs, cache coherence verification, performance, and resource utilization. Section VIII and Section IX concludes the report and suggests future work.

II. Related Work

A. Aquila SoC

This project is based on the Aquila processor core, a 32-bit RISC-V processor [5]. It features a five-stage in-order pipeline, including instruction fetch (IF), instruction decode (ID), execute (EXE), memory access (MEM), and write-back (WB), which simplifies design and enhances execution efficiency. Fig. 1 shows the pipeline organization.

Aquila supports the RV32I base instruction set [6], the RV32M multiplication and division extension, and the RV32A atomic operation extension, making it suitable for both embedded and general-purpose computing. It employs a Harvard architecture with separate instruction and data caches and integrates tightly-coupled memory (TCM) for better performance. The processor connects to DDR3 DRAM via the Xilinx MIG7Series memory controller for high-speed data access. Fig. 2 shows the complete system architecture.

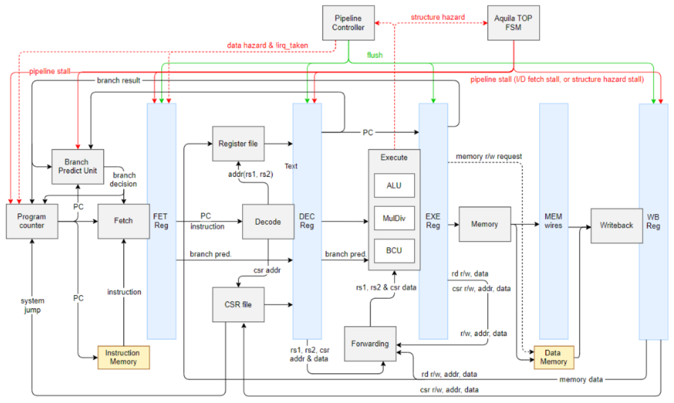
Fig. 1: Aquila SoC pipeline organization. [5]

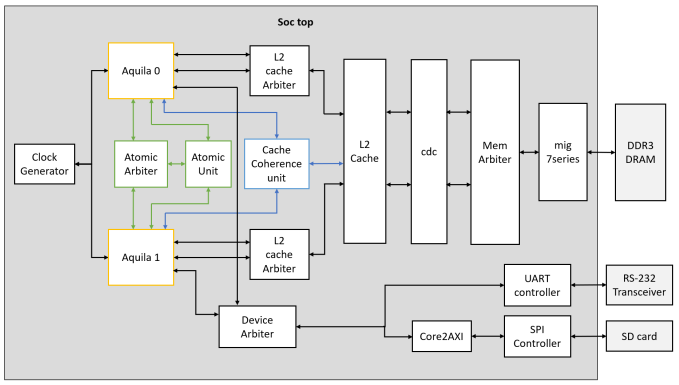
Fig. 3: Aquila-dual system architecture. [7]

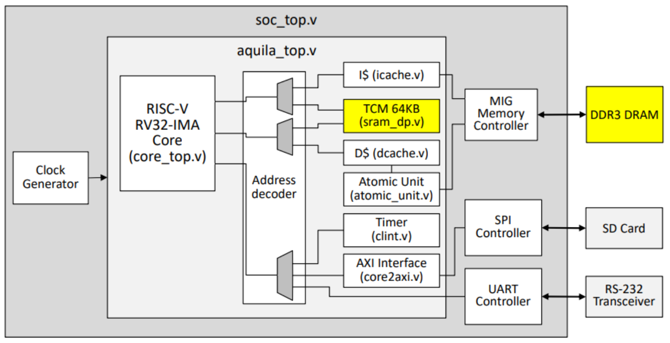
Fig. 2: Aquila SoC architecture. [5]

B. *Aquila-dual*

The Embedded Intelligent Systems Laboratory developed a dual-core Aquila-dual [7] processor consisting of two independent Aquila cores interconnected via an Atomic Arbiter and a Cache Coherence Unit. The Atomic Arbiter ensures stable, coordinated operations, particularly for atomic memory tasks like locking and synchronization. The Cache Coherence Unit utilizes the MESI protocol to maintain L1 cache coherence, preventing data conflicts and computational errors.

Aquila-dual's memory architecture (Fig. 3) is designed for flexibility and efficiency. The shared L2 cache connects both cores and serves as a buffer for external DDR3 DRAM, reducing memory access frequency and improving data retrieval performance. The L2 Cache Arbiter ensures fair and efficient cache request handling to minimize conflicts and latency, while the Memory Arbiter manages DDR3 DRAM access among the L2 cache, external memory, and peripherals to optimize overall system performance. This SoC is implemented in Verilog and operates at 100 MHz on a Xilinx KC-705 FPGA [8].

However, the current MESI implementation is limited to dual-core configurations, lacks circuit area optimizations, and does not fully adhere to the MESI protocol. Therefore, we redesign the system based on relevant literature to overcome these limitations.

C. *Gem5 multicore simulation*

GEM5 [9] [10] provides comprehensive simulation capabilities and detailed documentation, particularly the reference book "A Primer on Memory Consistency and Cache Coherence [11]" authored by professors specializing in memory coherence protocols. The book thoroughly describes MESI protocol states, cache state transitions, and their correctness. In the MESI protocol, each core possesses its own L1 cache, divided into instruction and data caches, while the L2 cache is shared among all cores. Further details regarding our application of GEM5 and the referenced MESI technology will be discussed in Section IV-A.

III. Multi-core System Design

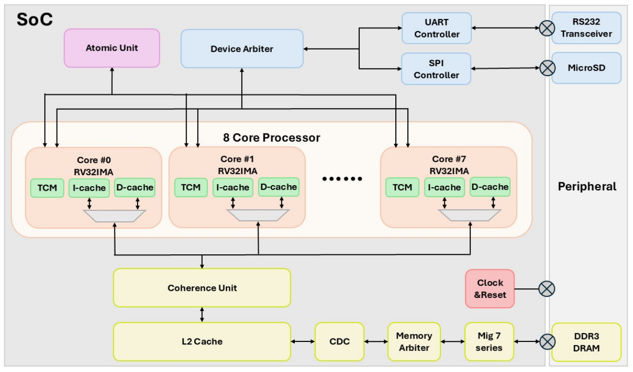
Fig. 4: Aquila-MP multi-core system architecture.

The preliminary system architecture of this research is shown in Fig. 4.

The cache coherence unit developed in this study connects four Aquila processor cores using a Snooping Bus architecture. It manages memory read and write requests from each core and ensures cross-core data coherence through a broadcasting mechanism. Detailed implementation specifics will be discussed in Section IV-B.

The L2 Cache serves as a shared system cache, providing additional cache capacity to reduce direct accesses to external DDR3 DRAM, thereby improving memory access latency and overall system performance. Internally, the L2 Cache employs a finite state machine (FSM) based on the MESI protocol to maintain data coherence with L1 caches.

Additionally, the Atomic Arbiter and Device Arbiter are critical new modules introduced in this study. They handle atomic instruction execution and arbitration among the four cores, as well as device access arbitration. Their functionalities include:

1) **Core ID Priority Arbitration**: Ensures only one core executes atomic operations or accesses a device at a time by prioritizing core IDs.
2) **Contention Avoidance Mechanism**: Prevents simultaneous resource requests by multiple cores, ensuring system stability and efficient execution.

IV. MESI Protocol

A. Introduction to the MESI Protocol

1) L1 Cache: In this project, MESI protocol is used to maintain the behaviour between L1 caches and main memory. This protocol defines four states: Modified (M), Exclusive (E), Shared (S), and Invalid (I). This four state help manage how data is stored and accessed in the cache. Table I provides an overview of these states in an L1 cache.

The Modified (M) state means the cache has an updated version of the data. In this state the data in the block will be different from main memory. The Exclusive (E) state indicates that the cache holds the only copy of the data which is same as the main memory. The Shared (S) state means multiple L1 caches store the same version of the data. Lastly, the Invalid (I) state signifies that the cache block is no longer valid, requiring a fetch from memory or another cache. In the next section, we'll explore how these states transition when a core requests data based on different events.

State	Access Permission	Description
Modified (M)	Read/Write	Data is modified (dirty) but valid.
Exclusive (E)	Read	Data is exclusively owned by a single L1 cache.
Shared (S)	Read	Data is shared among multiple L1 caches and the L2 cache.
Invalid (I)	None	Data is invalid.

TABLE I: L1 cache states in the MESI protocol. [11]

2) L2 cache: In the MESI cache coherence protocol, four cores share the L2 cache. TABLE IIexplains how each block in the L2 cache is assigned one of four states based on its current usage and validity.

3) Event Descriptions in the MESI Protocol: Before discussing the state transitions in L1 and L2 caches, we define three types of memory access operations: GetS, GetM, and PutM. Their details are listed in TABLE III. These operations are designed based on the Write-Back memory strategy.

State	Corresponding L1 Cache State	Description
Non Owner	Exclusive/modified	L2 cache is invalid, but corresponding cache block in L1 is in E or M.
Share	Share	At least one L1 cache contains the same data in the S state.
Owner	Invalid	L2 cache holds the only copy of the data.
Invalid	Invalid	Data is stored in memory.

TABLE II: L2 cache states in the MESI protocol. [11]

Event	Trigger Condition	Behavior
GetS	Read miss in L1 Cache	Broadcasts a request via the snooping bus; the core or L2 cache holding the data returns it.
GetM	Write miss in L1 Cache	Similar to GetS, but with different state transition rules upon completion.
PutM	An L1 Cache block in M/E state is evicted	Writes the block back to the L2 cache to ensure data consistency, making the L2 cache the data owner.

TABLE III: Memory events defined in the MESI protocol. [11]

4) FSM of the MESI protocol: The left FSM in Fig. 5 shows how an L1 cache block changes states in the MESI protocol to keep data consistent. It moves between four states—Modified, Exclusive, Shared, and Invalid—based on read and write operations. If a core reads a block in Exclusive or Shared, nothing changes (silent hit). But if it writes to an Exclusive block, it switches to Modified.

When another cache requests data from an Exclusive block, it moves to Shared. If the block is Modified and another cache issues a GetS (Get Shared) request, it also becomes Shared to ensure consistency before being shared. But if a different core sends a GetM (Get Modified) request, the block turns Invalid, meaning another cache now owns it. If an L1 cache needs data that's currently invalid (Own-GetS or Own-GetM), it transitions to Shared, Exclusive, or Modified, depending on whether it can claim exclusive access.

The right FSM in Fig. 5 shows how L2 cache manages coherence and ownership for L1 requests. It has three states—Owner, Non-Owner, and Share—to track whether it holds the latest data. When L1 requests a GetS or GetM, L2 becomes the Owner if it has the newest copy. If L1 later gives up ownership (PutM), L2 shifts to Non-Owner, meaning it no longer holds the authoritative version.

The Share state means L2 has a shared copy but doesn't own it. If L1 later sends a GetM request, ownership transfers to maintain coherence. This FSM helps manage multiple L1 caches efficiently, ensuring only one valid copy of modified data exists at any time.

B. Implementation of the MESI Protocol

The MESI protocol implementation follows the methodology used in Aquila-dual. In the L1 cache, as shown in TABLE IV , the traditional Valid Bit and Dirty Bit are extended

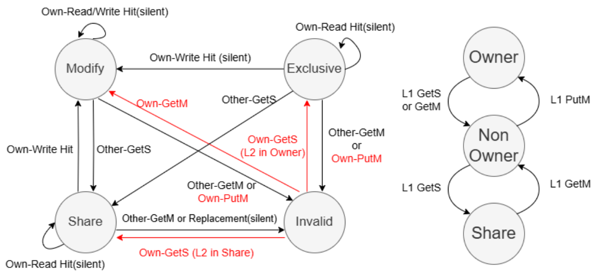

Fig. 5: Finite State Machine (FSM) of the MESI protocol: L1 cache (left) and L2 cache (right). [11]

with an additional Share Bit. The combination of these three bits represents the four MESI states. This method not only fully maps the MESI states but also aligns with the existing hardware conventions for cache status bits, ensuring feasibility and efficiency in hardware implementation.

State	Share Bit	Valid Bit	Dirty Bit
Modified (M)	0	1	1
Exclusive (E)	0	1	0
Shared (S)	1	1	0
Invalid (I)	0	0	0

TABLE IV: MESI state representation in L1 cache. [7]

The L2 cache, on the other hand, uses a 2-bit state representation, as detailed in TABLE V. This design choice optimizes memory bandwidth usage by ensuring that L2 only stores the necessary metadata required for coherence management.

State	State Bits
Invalid (I)	00
Owner (O)	01
Non Owner (NO)	10
Shared (S)	11

TABLE V: L2 cache state representation.

In reference to the Muntjac Multicore [12], a quad-core RV64 SoC utilizing the TileLink [13] cache coherence protocol, we establish the priority of different memory events based on TileLink's handling of cache coherence. The priority order is as follows:

1) **Write-back event:** This occurs when data is written back to the L2 cache or main memory to maintain data coherence. Given its importance in ensuring consistency, it is assigned the highest priority.
2) **Invalidate event:** When one core modifies a shared data block, it must promptly notify other cores to ensure coherence. Therefore, invalidation events should be processed with high priority.
3) **Read request event:** Read requests can be queued and processed in order, as they do not immediately impact data coherence.

This ordering ensures efficient memory operations while maintaining cache coherence across multiple cores.

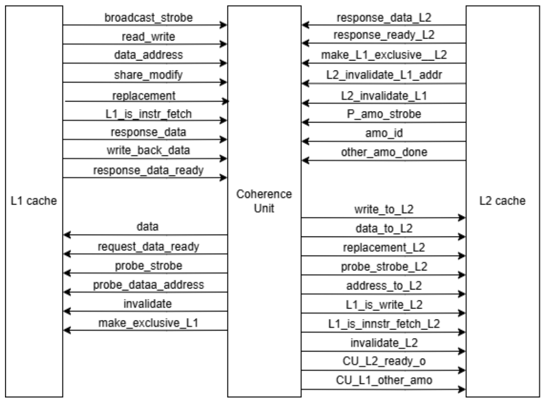

Fig. 6: Coherence unit architecture

V. Implementation of Cache Coherence Controller

A. Coherence Unit

Fig. 6. illustrates the datapath between the coherence unit and the L1/L2 caches. The coherence unit implements the MESI protocol to maintain cache coherence among four cores, managing data requests and ensuring consistency between L1 and L2 caches. The unit handles four primary memory events: read miss, write miss, write hit on a shared state, and cache replacement.

1) Read Miss: Fig. 8. presents the workflow of a read miss event in the system. When a core encounters a read miss (broadcast_strobe asserted with read_write = 0), the coherence unit first checks whether the requested data is present in another core's L1 cache in the Modified (M) or Exclusive (E) state. If so, the coherence unit directly supplies the data to the requesting core and subsequently performs a write-back to the L2 cache to ensure compliance with the MESI protocol, where the L2 cache acts as the owner of shared-state data. The corresponding timing diagram of the coherence unit for this scenario is shown in Fig. 7..

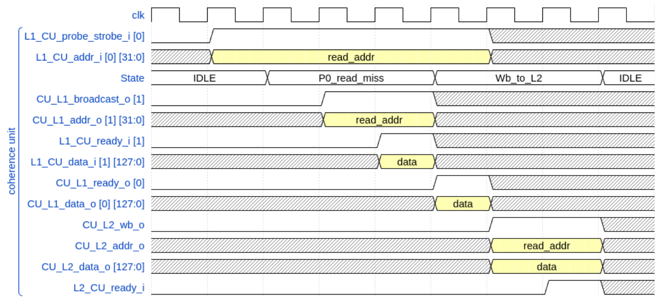

Fig. 7: Timing diagram of read miss event in coherence unit, in this case, data is sent from other core.

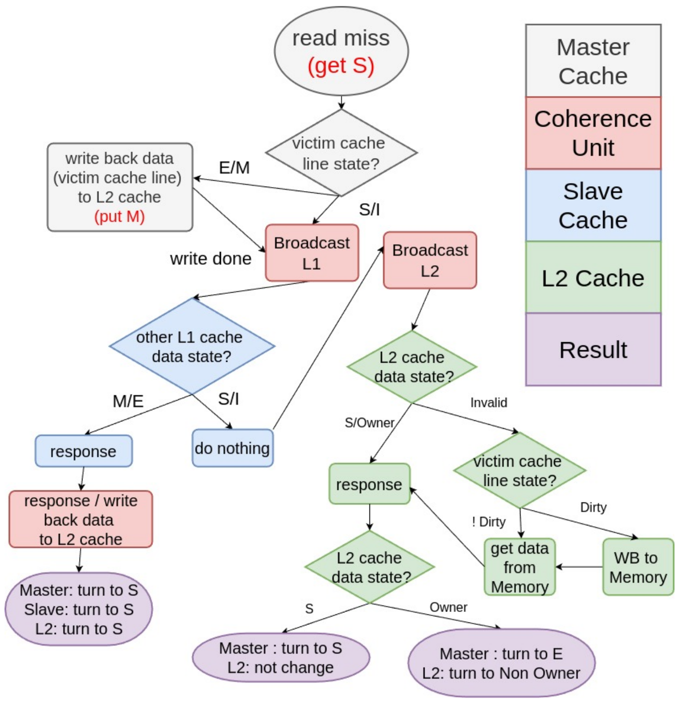

Fig. 8: Workflow of a read miss event in the system.

If no core has the requested data, the coherence unit fetches it from the L2 cache by asserting probe_strobe_L2, then forwards the data to the requesting core as an L2 cache response. Fig. 9. illustrates this timing scenario, highlighting the two additional cycles required to confirm the absence of data in other caches.

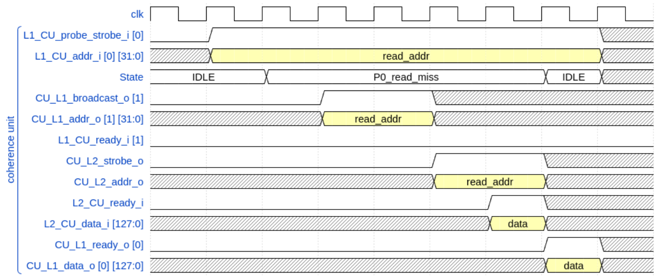

Fig. 9: Timing diagram of read miss event in coherence unit, in this case, data is sent from L2 cache.

2) Write Miss: The operation for a write miss event is similar to that of a read miss, except the coherence unit asserts an invalidate signal when broadcasting to other cores and the L2 cache. Any cache holding identical valid data will transition to the Invalid (I) state, adhering to the write-invalidate protocol.

3) Write Hit on Shared: If a core attempts to write to a cache block currently in the Shared (S) state, the coherence unit issues an invalidation broadcast (invalidate asserted) to all other cores sharing that cache line, forcing them to invalidate their copies in compliance with the write-invalidate protocol.

4) Replpacement: When a core needs to evict a cache line in the Modified (M) or Exclusive (E) state due to cache replacement (replacement asserted), the coherence unit ensures the data is written back to the L2 cache (write_to_L2 asserted), returning ownership of the data to L2. If another core subsequently requests this evicted address, it will retrieve the data directly from the L2 cache.

Additionally, the coherence unit supports atomic memory operations (AMO) to ensure correct execution of operations such as amoswap.w.aq or amoor.w. When one core initiates an AMO (P_amo_strobe asserted), all other cores are temporarily stalled (CU_L1_other_amo_o asserted) until the atomic operation completes. This prevents interference and ensures that AMO operations execute in a serialized manner, avoiding race conditions.

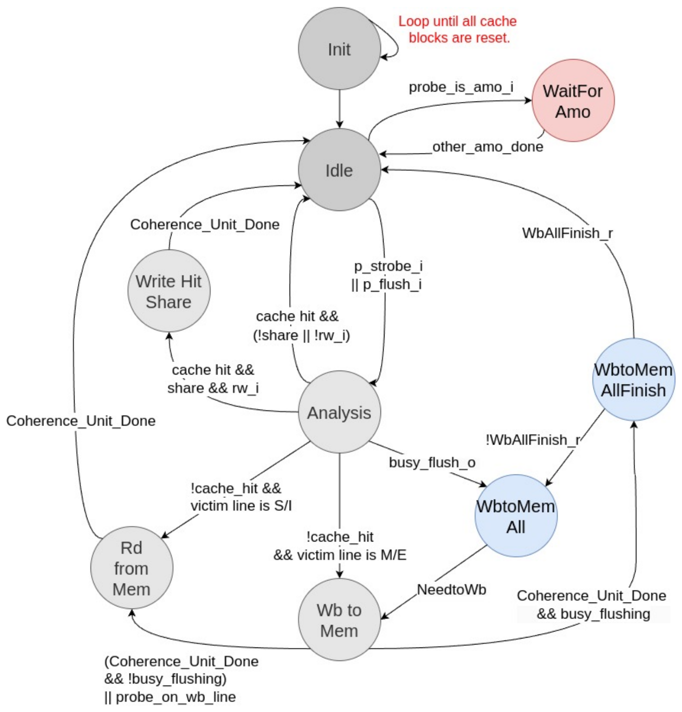
Init
Loop until all cache blocks are reset.
probe_is_amo_i
WaitFor Amo
other_amo_done
Idle
Coherence_Unit_Done
WbAllFinish_r
Write Hit Share
p_strobe_i || p_flush_i
cache hit && (!share || !rw_i)
cache hit && share && rw_i
WbtoMem AllFinish
Coherence_Unit_Done
Analysis
!WbAllFinish_r
busy_flush_o
!cache_hit && victim line is S/I
WbtoMem All
Rd from Mem
!cache_hit && victim line is M/E
NeedtoWb
Coherence_Unit_Done && busy_flushing
Wb to Mem
(Coherence_Unit_Done && !busy_flushing) || probe_on_wb_line

Fig. 10: Finite state machine (FSM) of L1 D-cache.

B. Data Cache

The original Aquila data cache is a 4-way associative cache utilizing single-port SRAMs for storing tags and data. To support more cores and resolve timing violations, we reduced the cache to 2-way associativity. Additionally, for MESI protocol support, we added another I/O-port to handle probe events.

Fig. 10. illustrates the finite state machine of the data cache. Upon a memory request, the cache initially checks data validity. If a cache miss occurs, it examines the victim cache line's state and manages the write-back process for M/E states. Once resolved, it sends a data miss event to the coherence unit and waits for data reception. For probe events, when a cache hit occurs, the cache returns the requested data and updates its state based on the event.

When a processor experiences a write-hit on a shared cache line; it must trigger an invalidation event for other cache lines, causing the core to stall until completion. Specific FSM states (highlighted in red and blue) handle unique events. The red state represents stalls during lock-based atomic instructions performed by other cores, ensuring atomicity. The blue state handles the RISC-V fence.i instruction by writing back all data in M/E states.

C. Level-2 Cache

The L2 cache is a 4-way associative cache that uses single-port SRAMs to store both tags and data. Fig. 11. shows the finite state machine (FSM) of the data cache. The FSM starts in the Idle state, waiting for a request. When an access occurs, it moves to Analysis, where it checks whether the requested data is in the cache (cache hit). If the data is present and clean, the FSM simply returns to Idle, sending the data back to L1 without further action. If the data is not in the cache (cache miss), the FSM checks whether the victim block (the block that will be replaced) is dirty. If it is, the FSM writes the dirty block back to memory before fetching the new data. If the victim block is clean, it skips the writeback and directly loads the new data from memory.

A special state, Invalidate L1, handles cases where the block is in a shared state between L1 and L2, meaning both levels of cache contain the data, but it differs from the version in main memory. If this shared block is chosen as the victim block, the FSM first instructs L1 to invalidate its copy before replacing the data in L2. This ensures that L1 does not retain an outdated version of the data, maintaining coherence across the memory hierarchy.

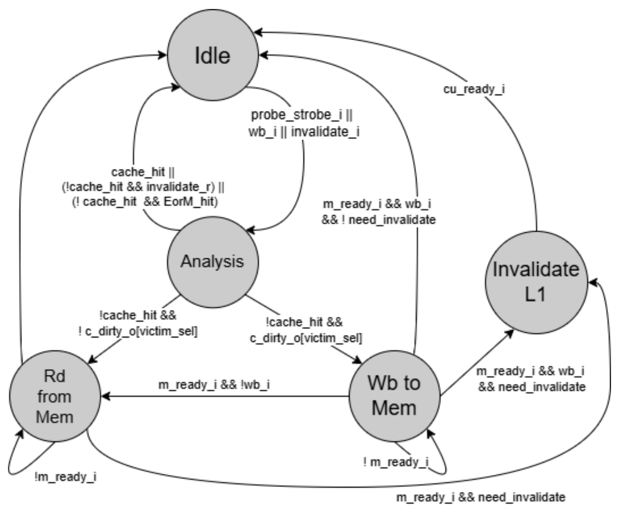
Idle
probe_strobe_i || wb_i || invalidate_i
cu_ready_i
cache_hit || (!cache_hit && invalidate_r) || (! cache_hit && EorM_hit)
m_ready_i && wb_i && ! need_invalidate
Analysis
Invalidate L1
!cache_hit && ! c_dirty_o[victim_sel]
!cache_hit && c_dirty_o[victim_sel]
Rd from Mem
m_ready_i && !wb_i
Wb to Mem
m_ready_i && wb_i && need_invalidate
!m_ready_i
! m_ready_i
m_ready_i && need_invalidate

Fig. 11: Finite state machine (FSM) of L2 cache.

VI. "A" Standard Extension Instruction

In a multi-core system like Aquila-quad, the RISC-V ”A” extension [14] is essential for efficient synchronization and communication between cores. When multiple cores access shared resources, atomic operations ensure data integrity and prevent race conditions. Without hardware support for atomic instructions, synchronization would rely on software-based solutions, which are not only slower but also more complex to implement.

A. Atomic Unit

The Atomic Unit [15] in Aquila is designed to handle atomic memory operations (AMOs), which are crucial for ensuring synchronization and consistency in a multi-core system. It manages Load-Reserved (LR) and Store-Conditional (SC) operations, along with other atomic operations like swap, add, and bitwise logic updates on shared memory. The unit operates through a finite state machine (FSM) that coordinates reads and writes to memory, ensuring that atomic operations are completed without interference from other cores. It tracks reservation addresses for LR/SC operations, ensuring that a store-conditional (SC) succeeds only if no other core has modified the data.

B. Software Design

Since the quad-core system shares a single UART resource managed by the device arbiter, it is essential to prevent multiple cores from simultaneously executing UART-related functions, such as `printf()` or `inbyte()` (used in the bootloader). To ensure proper synchronization and avoid conflicts, a mutex lock is employed.

In addition to mutual exclusion, synchronization between cores is required, as computation tasks typically consist of multiple sequential stages. To coordinate task completion, a semaphore mechanism is implemented using the atomic OR operation (AMOOR). Initially, the semaphore flag is set to `0`, and the mask is defined as `4'b1111`. After completing its designated computation, each core performs an atomic OR operation on the flag using the expression: $1 << \text{core\_id}$. Each core then waits until the flag reaches `4'b1111`, indicating that all cores have completed their respective tasks, allowing the process to proceed to the next stage.

```
__attribute__((optimize(”O0”))) static void
    atomic_or(volatile unsigned int *addr, int val)
    {
    int old = *addr;
    asm volatile (
        ”amoor.w %0, %2, %1\n”
        : ”=r”(old), ”+A”(*addr)
        : ”r”(val)
        : ”memory”
    );
}
```

VII. Experimental Result

A. Resource Utilization

TABLE VI summarizes the resource utilization of the quad-core SoC implemented on the Arty A7-100T FPGA. The design includes the RISC-V cores, 16KB TCM, 8KB instruction cache (I-cache), 8KB data cache (D-cache), 64KB L2 cache, and additional supporting components. Due to the addition of extra I/O ports in the L1 cache to handle probe events, the implementation consumes additional Block RAM and LUT resources. To scale the cache sizes and TCM size further while managing resource constraints, transitioning to single-port caches may become necessary.

Name	LUTs as Logic	LUTs as Memory	Block RAM
Amo_arbiter	661	0	0
Atomic_unit	64	0	0
Coherence_Unit	1,243	0	0
Core 0	6,953	350	22
Core 1	7,195	350	22
Core 2	6,784	350	22
Core 3	6,668	350	22
L2 Cache	3,642	192	18
Total	37,957 (59.87%)	2,257 (11.88%)	107 (79.26%)

TABLE VI: Resource utilization of Aquila-Quad on the Arty A7-100T FPGA.

B. Experimental Environment

Our quad-core system is implemented in Verilog HDL, synthesized using Xilinx Vivado EDA software, and deployed onto Xilinx Arty A7-100T [16] FPGA board (Fig. 12.). The RISC-V software components are compiled using the official GNU Toolchain. The detailed specifications of our experimental setup are listed below.

- **RISC-V Toolchain:** riscv32-unknown-elf-gcc (version 13.2.0)
- **EDA Tool:** Xilinx Vivado 2024.1
- **FPGA Board:** Xilinx Arty A7-100T Evaluation Kit

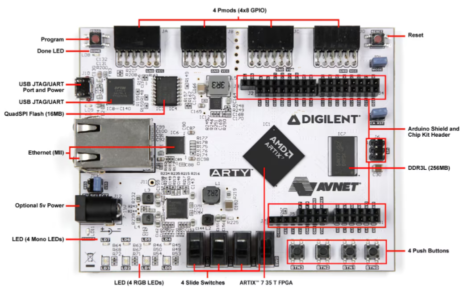

Fig. 12: Xilinx Arty A7-100T evaluation kit.

C. Test Program: A Multi-Layer Perceptron Application

Fig. 13. represents a fully connected artificial neural network with an input layer (784 nodes), a hidden layer (48 nodes), and an output layer (10 nodes), where different processing cores (Core 0, Core 1, Core 2, Core 3) are assigned specific computational responsibilities. Each color corresponds to a different core, indicating which nodes in the hidden and output layers are processed by which core. The hidden layer is divided among the cores in groups of 12 nodes per core, and the output layer is distributed as 3 nodes for core 0 to core 2 and 1 node for core 3. This suggest a parallel processing architecture, where each core is responsible for computing activations for its assigned nodes, optimizing computational efficiency. It further validates the system's ability to execute complex programs correctly.

Since our system does not support Linux or virtual memory management, we manually organize the memory layout within the DRAM address space, with reference to [7]. As shown in Fig. 14., each core has its own stack, read-only data, and code section while sharing a common heap and global data section. This design allows efficient shared memory usage while preserving individual stacks for independent execution.

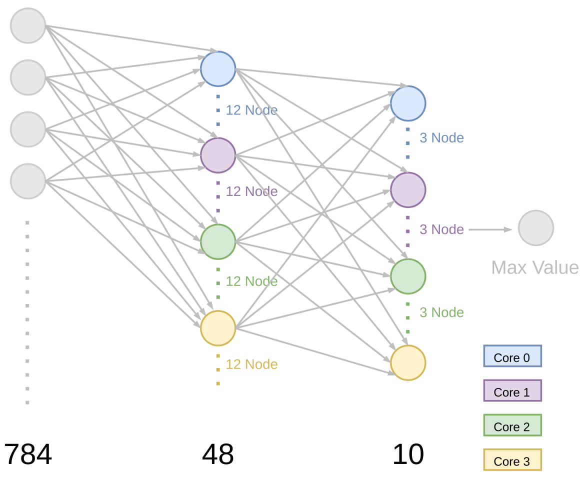

Fig. 13: Neural Network distribution for the quad-core system.

D. Test Program: Parallel Matrix Multiplication

To evaluate the impact of cache performance and coherence in a four-core system, we designed a parallel matrix multiplication program. Fig. 15. illustrates the task distribution across four cores, where each core is assigned its own computation section. At the end of the process, Core 0 verifies whether the computed matrix matches the expected result.

To assess cache utilization and sharing overhead, we perform a single 256×256 matrix multiplication. This test helps evaluate whether the working dataset fits within cache levels and how cache coherence mechanisms affect data sharing across cores.

To analyze cache contention and coherence maintenance, we execute 64×64 matrix multiplication 100 times. This test focuses on the frequent data synchronization overhead caused by cache coherence mechanisms and examines L1/L2 cache hit rates in small-scale, repetitive computations.

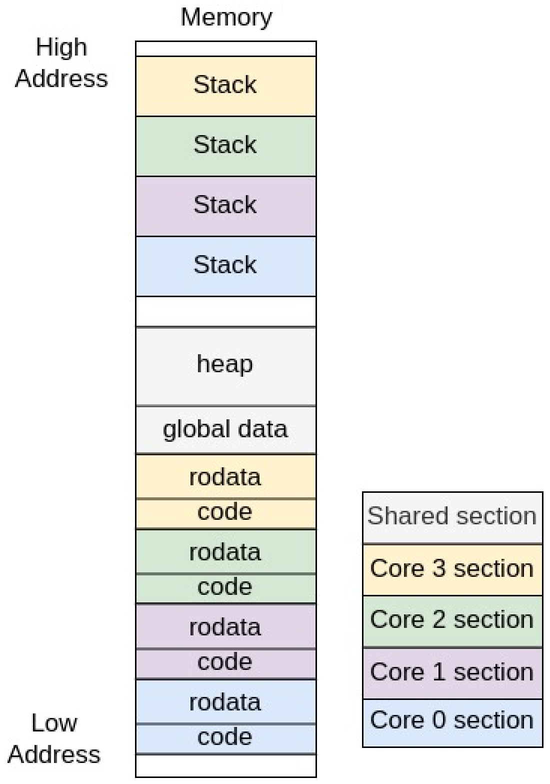

Fig. 14: Memory layout for the quad-core system.

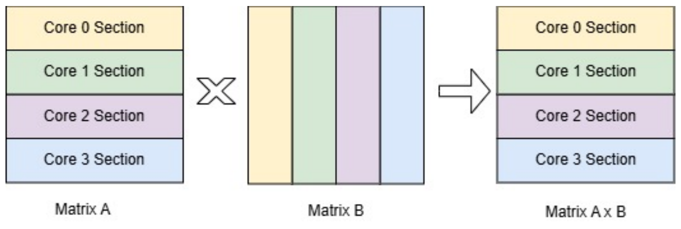

Fig. 15: Matrix multiplication computing distribution.

E. Array Sorting

We also design a parallel array sorting to test the performance on 8 core. In our implementation the array is first divided into eight equal segments. Each segment is independently sorted using the Bubble Sort algorithm. To exploit parallelism, these sorting tasks are distributed across eight cores, allowing all segments to be processed simultaneously. Once the local sorting within each segment is complete, the results are merged in successive stages to form a fully ordered array. This design leverages both task parallelism (across the eight cores) and a hierarchical merge process, ensuring efficient utilization of available computing resources while maintaining correctness of the overall sort.

F. Performance

The experimental results demonstrate the performance improvements as the number of cores increases. For the MLP-based OCR benchmark, the system achieved up to 7.27× speedup with eight cores compared to the baseline, showing nearly linear scalability. Similarly, in the Array Sorting experiment with 40K elements, parallel execution across eight cores delivered a 7.41× speedup, highlighting the efficiency of dividing the dataset into segments and merging results hierarchically. The Matrix Multiplication benchmark (64×64) exhibited the highest parallel efficiency, reaching a 7.53× speedup with 8 cores. These results confirm that the proposed multi-core RISC-V system effectively leverages parallelism to accelerate computational workloads, with scalability depending on the characteristics of each application—compute-intensive tasks such as matrix multiplication benefit most, while memory movement and merging overheads slightly limit sorting performance.

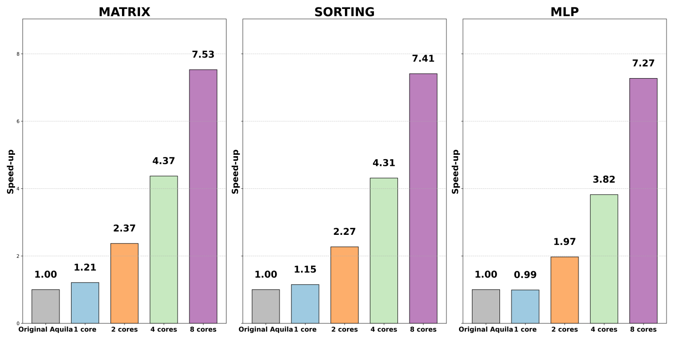

Fig. 16: Speed up ratio for different core number

G. Cache Latency Analysis

To evaluate the scalability limits of the snooping bus architecture, we extended our experiment on the matrix benchmark to 16 cores. The results in Fig. 17. show that the speed-up reached 11.71× compared to the baseline, which is significantly below the ideal linear scaling of 16×.

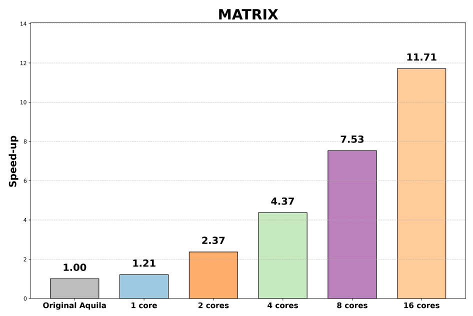

Fig. 17: Speed-up of Matrix Benchmark Extended to 16 Cores

Fig. 18. shows the latency trends of cache coherence events as the number of cores increases. While Write Miss (I $\rightarrow$ M) remains relatively stable, rising only from 28.9 cycles at 1 core to 32.1 cycles at 16 cores, the latency of Read Miss (I $\rightarrow$ E) and Read Miss (I $\rightarrow$ S) increases sharply, reaching 15.2 and 22.3 cycles at 16 cores, respectively. This significant growth in coherence latency directly explains why the system fails to scale efficiently at 16 cores: the higher cost of serving read

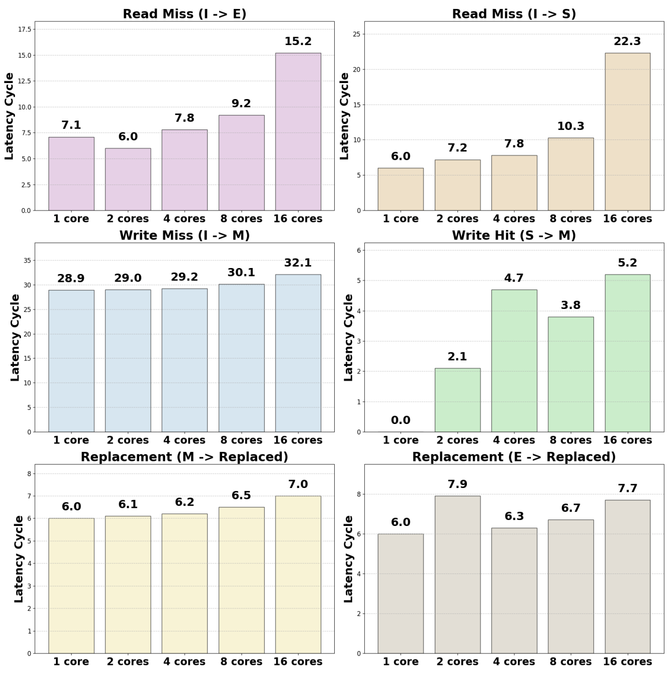

Fig. 18: Average Latency of Cache Coherence Events under Different Core Counts

misses limits parallel execution and reduces the achievable speed-up.

Fig. 19. further illustrates the per-core latency distribution of coherence events. It shows that as the number of cores grows, the variance in latency between different harts widens considerably, with some harts experiencing much higher delays. This demonstrates the inherent limitation of the snooping bus: when the core count becomes large, every coherence event must be serialized and broadcast to all participants, causing individual cores to wait excessively for others to acknowledge or respond. Such prolonged waiting times for each coherence transaction make the snooping bus inefficient at larger scales.

H. Cache Size

The TABLE VII shows how different L1 and L2 cache sizes impact performance for a matrix (64) workload. The big takeaway here is that when the L2 cache is reduced from 16 KB to 8 KB, performance drops dramatically. This happens because of conflict misses—when multiple memory addresses compete for the same cache space, causing frequent evictions and forcing the system to fetch data from slower memory. You can also see that while the L1 cache size makes a difference, the L2 cache size has a much bigger effect. With a larger 16 KB L2 cache, the system handles memory access more efficiently, reducing the number of costly trips to main memory. But when L2 is just 8 KB, it struggles to keep frequently used data, leading to a major slowdown.

VIII. Conclusion

Aquila-Quad demonstrates efficient multicore performance through its MESI-based cache coherence and atomic instruction support for synchronization. Performance evaluation indicates that it achieves near-linear speedup in tasks with high cache locality, such as MLP and small-scale matrix multiplication, reaching up to 3.65× and 9.25x acceleration, respectively. However, in write-intensive workloads with low locality, such as large matrix multiplication, cache coherence overhead leads to diminished gains. The addition of an L2 cache improves performance for specific workloads but introduces latency penalties in tasks with frequent write misses. Since many shared memory read/write programs require an OS capable of handling multicore execution, supporting a robust software environment is crucial for fully leveraging Aquila-Quad's potential.

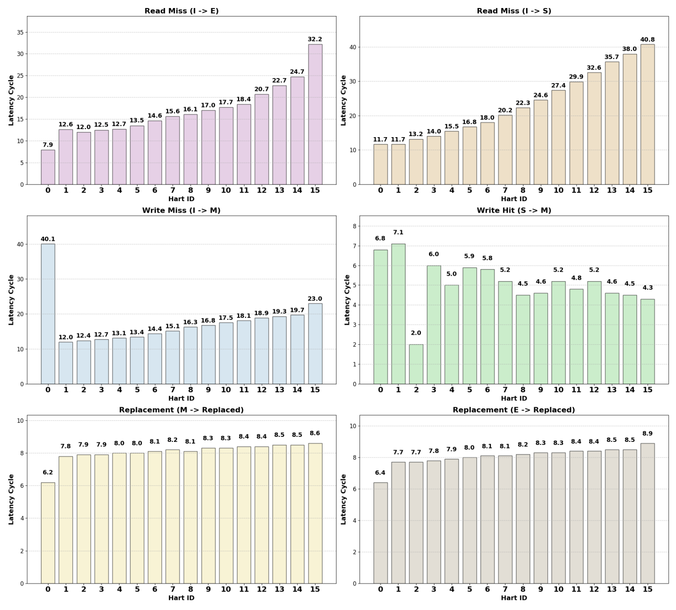

Fig. 19: Latency Distribution of Coherence Events Across 16 Cores

L2 Cache	L1 Cache	matrix(64)
16 KB	4 KB	9018 ms
	2 KB	9795 ms
8 KB	4 KB	33486 ms
	2 KB	34324 ms

TABLE VII: L2 and L1 cache configurations with matrix (64) results.

IX. Future Work

A. Linux Support

To extend the capabilities of Aquila-Quad, we plan to integrate a Memory Management Unit (MMU) to enable Linux operating system support. This enhancement will significantly expand the software ecosystem, facilitating broader application compatibility and improving system usability. With Linux support, Aquila-Quad will be capable of handling more complex workloads, further enhancing its practicality for high-performance computing applications.

B. Optimization of Resource Utilization

As discuss in Section VII-F, The Block RAM utilization of L1 Cache can be reduced as we modify the data and tag ram to single port. the extra block ram resource can be used to enlarge the cache size and TCM size to put a more powerful System Initialization Code in the TCM.

C. Optimization of Memory Access Latency

In a multi-core system, the memory request/response path tends to be longer. In the current system, any attempt to shorten critical paths in the memory access module may introduce timing violations, potentially destabilizing processor operation. However, optimization may still be achievable by simplifying the protocol rules outlined in the reference documentation or refining the implementation logic to enhance efficiency.

References

[1] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA,", May 13th, 2011.

[2] RISC-V, Rocket Chip Generator Project. [Online] Available: https://github.com/freechipsproject/rocket-chip

[3] ETH Zurich, PULP Platform Project. [Online] Available: https://pulp-platform.org/

[4] W. Lei and F. Xiao-ya, "Study on L2 cache of multi-core processor and optimization for embedded," 2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), Xi'an, China, 2011, pp. 1-5.

[5] Embedded Intelligent Systems Lab, NYCU, Hsinchu, Taiwan. The Aquila SoC, available online at: https://github.com/eisl-nctu/aquila, accessed: July 02, 2024.

[6] A. Waterman, K. Asanovi c, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213", December 13th , 2019, [On-line] Available: https://riscv.org/specifications/

[7] YEN, LIN-EN. "Two-Level Coherent Cache Design for Multi-Core RISC-V Processors." Mater thesis, Dept. C.S., NYCU, Hsinchu, Taiwan, 2024.

[8] Xilinx Kintex-7 FPGA KC705 Evaluation Kit. [Online] Available: https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html

[9] The Gem5 simulator, available online at: https://github.com/Gem5/Gem5, accessed: July 02, 2024.

[10] The Gem5 documentation for MESI on the official website, available online at: https://www.gem5.org/documentation/general_docs/ruby/MESI_Two_Level/, accessed: July 06, 2024.

[11] M. D. Hill and M. R. Marty, A Primer on Memory Consistency and Cache Coherence, 2nd ed. Madison, WI, USA: University of Wisconsin-Madison, 2020. [Online]. Available: https://pages.cs.wisc.edu/ markhill/-papers/primer2020_2nd_edition.pdf

[12] X. Guo, D. Bates, R. Mullins, and A. Bradbury, "Muntjac multicore RV64 processor: introduction and microarchitectural guide" lowRISC CIC, June 2022.

[13] SiFive. "TileLink Specification." Version 1.8.1, January 27, 2020.

[14] A. Waterman and K. Asanovic, The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20191213, RISC-V Foundation, Tech. Rep, 2019.

[15] Hsiang, Chih-Yu. "Design of the Atomic Instructions and the Data-Coherent Cache for a Multi-Core RISC-V Processor." Mater thesis, Dept.C.S., NCTU, Hsinchu, Taiwan, 2020.

[16] Xilinx, Arty A7-100T Evaluation Board for the Artix-7 FPGA, reference manual, available online at https://digilent.com/reference/programmable-logic/arty-a7/reference-manual?s rsltid=A