Proyecto 1: MESI & cache coherence

[illegible]

Soto, Camacho, Montero

Introduction

In the contemporary computing landscape, the proliferation of multicore systems has revolutionized the way we harness processing power. The integration of multiple processing elements within a single system presents an array of challenges, particularly concerning memory management and cache coherency. This paper focuses on the implementation of a multicore system that interconnects numerous PEs while ensuring cache coherence through the MESI protocol.

Within this context, we explore the complexities of managing memory subsystems in a multicore environment and highlight the pivotal role played by a well-designed memory controller. By examining benchmarking data, we assess the impact of memory controller features tailored to enhance multicore traffic performance and cache coherence.

Contents

Term	Definition
MM	Avalon Memory Mapped Interface
ST	Avalon Streaming Interface
PE	Processing element
Token	Hazard prevention unit that allows a node to send messages to others
Interconnect	Modules and hardware interfaces required for managing communication between the PEs and their caches
MESI	Invalidate-based cache coherence protocol

Table 1. Various terms and their definitions.

1 Design

1.1 Design process

The two main objectives with this design are simplicity and effectiveness.

The system must me simple to design and implement. It must be feasible to implement it in 5 weeks, it must fit in the 32070 ALMs available in a DE-SoC-1 FPGA. It must be simple enough to be implemented by 3 people. It must be compiled and synthesized in a reasonable amount of time with consumer level machines. [1]

The system must be effective. Despite being small, it must be able to boot Linux and have an acceptable performance. It must be able to execute a small program to test its functionality.

In order to find a suitable design, research papers and past industry practices were taken into consideration.

1.1.1 Core

For the core implementation, the initial plan was to develop one capable of writing and reading data from memory and executing three instructions: write addr reg, read addr reg, incr reg.

However, the decision was made to use a much more complete core developed in previous projects.

1.1.2 Interconnect

The first major decision for the design was the interconnect. Three aspects of it are of special importance:

- The specific bus protocol (or protocols) that will communicate the caches
- Topology
- Hazards

1.1.2.1 Topology

Our research yielded the following conclusions regarding topology:

Ring interconnects are a good compromise between the centralized design of a bus or a crossbar design and the complexity and overhead associated with a packet-switched network, as they use short core-to-core connections and, in our case, doesn't need a router. [2]

We have found that ring topologies outperform other designs when used in small configurations (8 cores or less). [3]

Furthermore, Intel has used a ring topology to interconnect physical cores together. They have since moved on from this approach due to their increasing number of cores, but our design has only 4 cores, so this should be of no concern for us. [4]

Thus, we have decided to implement a Ring topology for our interconnect.

1.1.2.2 Bus protocol

For the specific bus protocol we have chosen to implement both ST and MM buses from the AVALON specification.

ST is used for communication between the cores. It is a stream oriented bus that transmits messages related to:

1. Token
2. MESI
3. Requests and responses

MM is used for non arbitrated core-cache (32bit) communication and arbitrated (128bit) cache-system communication. Note that 128bits is the size of a cache line.

1.1.2.3 Token

During the design process, we noticed that the ring topology behaves very similarly to a pipeline. This implies that there must be hazards principally due to the overlapping stages of instruction execution.

Some papers mentioned techniques on how to solve this problem. Nevertheless we chose to implement our own system. [3]

We have named our hazard prevention system “Token”. It can be summarized in 3 rules:

1. The token jumps to the next cache every clock cycle.
2. Only the core which holds the token can emit a new message.
3. The value of the token is shifted every clock cycle.

This system was inspired by IEEE 802.5 and will be further explained in the Design Proposal section. [5]

1.1.3 Cache

For the cache, two main coherence strategies available given the topology we chose: directory or snooping.

In a snooping cache, each cache monitors or "snoops" the system's ring to detect and resolve data coherence issues by invalidating or updating its own cached data when another core writes to the same memory location.

In a directory cache, a centralized directory keeps track of the memory locations cached by each core and manages data coherence by forwarding read and write requests to the appropriate caches.

In our research, we also found that snooping outperforms directory-based strategy in our use case. [3]

Thus, we have decided to implement a snooping based strategy for our cache.

Finally, due client requirements, the MESI protocol was used for cache coherence.

1.2 Design proposal

The system consists of 4 cores. Each core is connected to its respective cache controller through an MM link. Each cache controller is connected to the next one through an interconnect, which contains the Token Ring and Message Ring. In addition to that, the cache controllers are all connected to memory through an MM link. Finally, the cache conntroller is the manager of the debug subordinate and of course contains the cache memory itself.

Figure 1 shows a general diagram of the interconnection between the elements of the system.

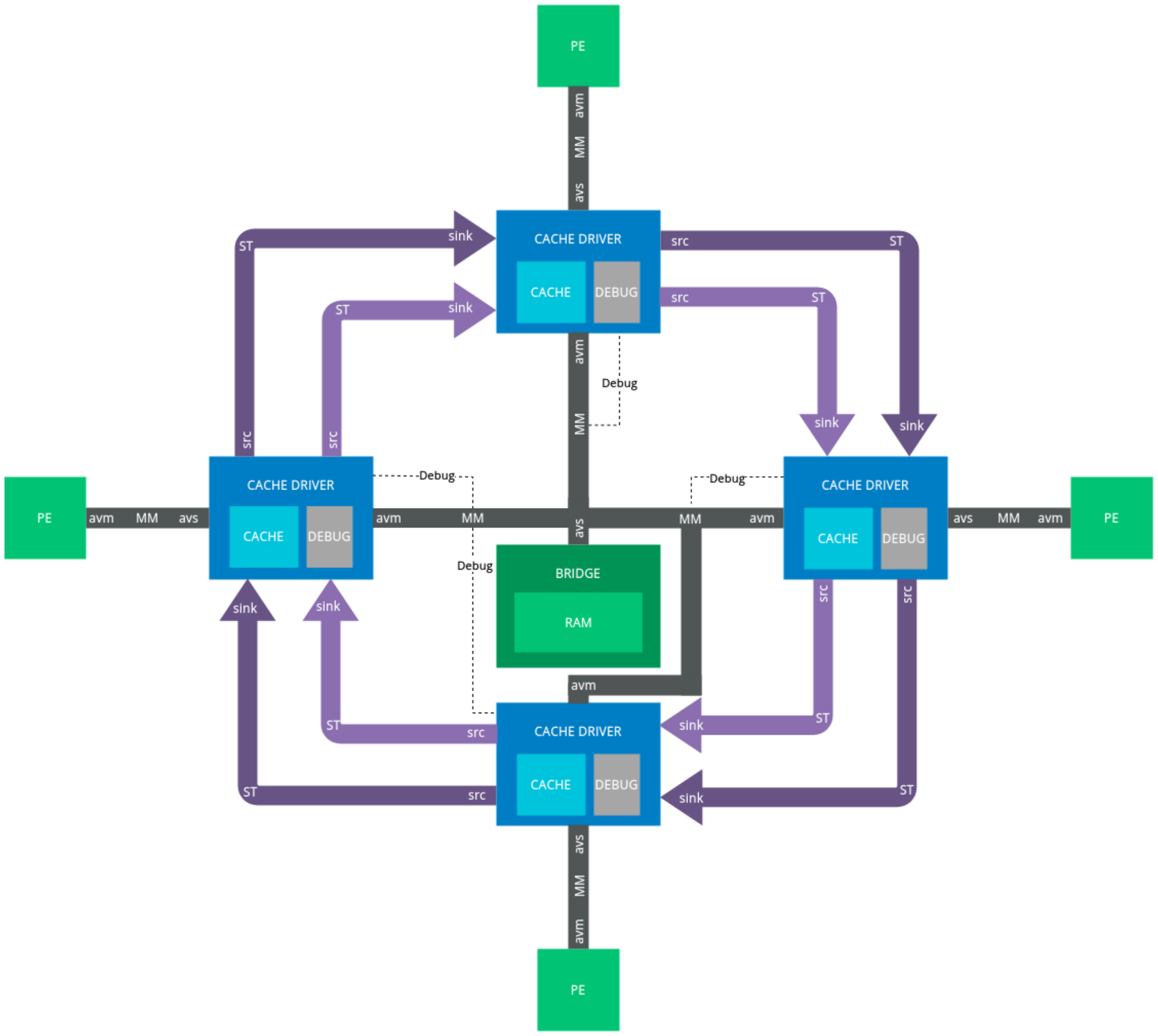


Figure 1. System interconnection.

1.2.1 Core

The core is the instruction processing unit, corresponding to each of the processing elements (PE) in figure 1.

The core implements the ARMv4 architecture and includes the following components:

- Fetch: The first stage of the core's pipeline. It retrieves the next instructions from memory that need to be processed.
- Decode: The second stage of the pipeline, which decodes the instruction obtained in the fetch stage so that it can be executed later.
- Porch: Determines whether an instruction should be executed or not by evaluating the conditional field that all ARMv4 instructions have in bits 31 to 28.

- Shifter: Performs bit-shifting operations.
- ALU (Arithmetic Logic Unit): Handles all arithmetic calculations and logical operations.
- PSR (Program Status Register): Manages the program status register, which is the register where the current program flags are stored.
- MUL (Multiply): Performs multiplication operations.
- Writeback: Writes to memory registers once an operation has been performed.

  Figure 2 shows a simplified structure of the core.

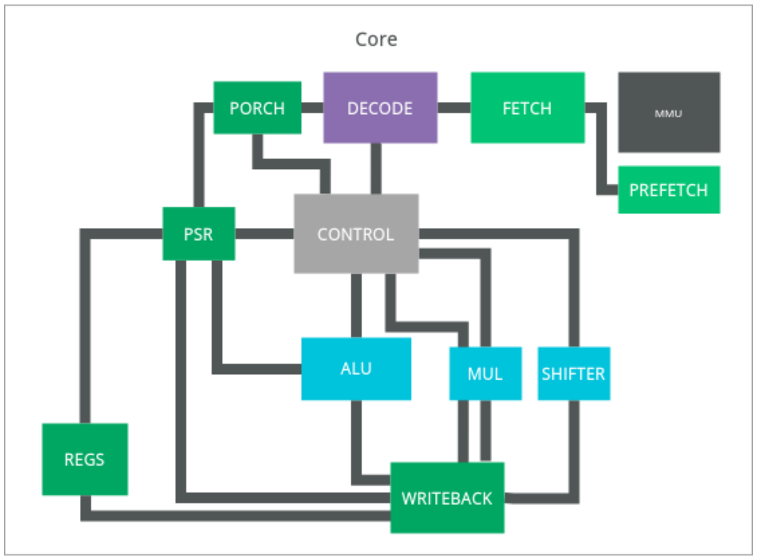


Figure 2. Core design.

Additionally, the PE has options for stepping and starting to process instructions, allowing for control over when and how instructions are executed.

Interestingly, ARMv4 was not designed to support multicore, so additional implementation of ldrex and strexeq was requiered.

1.2.2 Interconnect

This system implements the cache driver and the interconnect in a single module. The cache driver includes different state machines that control various aspects, including both rings implementing the MESI protocol.

The interconnect contains seven ports:

- Token in/out:
  - ST bus protocol, packet oriented
  - 78 bits wide

- 1 beat per cycle
- Supports idling but not back pressure
- Unidirectional

- Message in/out:
  - ST bus protocol, packet oriented
  - 158 bits wide
  - 1 beat per cycle
  - Supports idling and back pressure
  - Unidirectional
- PE interface:
  - MM bus protocol, transaction oriented
  - 32 bits wide
  - Non pipelined, but supports back pressure
  - Byteenable support
  - Bus lock support
  - Response support for locked transactions
- Memory interface:
  - MM bus protocol, transaction oriented
  - 128 bits wide
  - Pipelined and supports back pressure
  - Byteenable support
- Debug interface:
  - MM subordinate, transaction oriented
  - 32 bits wide
  - Has a register map which is detailed in a separate document
  - Enables dumping of cache lines

Figure 3 shows a diagram of this.

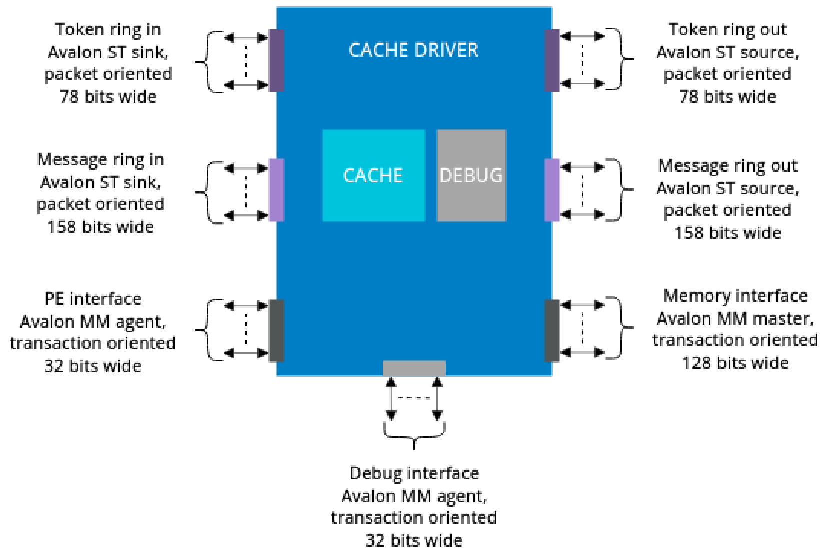


Figure 3. Cache interfaces.

All of these connections and their transactions are managed by a master state machine that defines what the SRAM controller is prioritizing, as SRAM has a single read/write port. So, it needs to decide whether to prioritize the core or the bus. Depending on the case, it can be in one of the following states:

- ACCEPT: After all other states, it transitions to ACCEPT.
- CORE: Actions related to each specific core (monitor, etc.).
- SNOOP: Monitoring memory and executing MESI. This happens when a message arrives, and the message's TTL is greater than 0.
- REPLY: This core received a response.

The initial state of this machine is ACCEPT. Upon starting or whenever it is in the ACCEPT state, it will decide whether to stay or transition to one of the following cases:

- If an incoming message arrives, it transitions to SNOOP or REPLY based on the TTL (Time To Live) value of the message.
  - If the TTL is greater than 0, it transitions to SNOOP, where it performs actions according to the MESI protocol.
  - If the TTL is zero, it transitions to REPLY, where it decides what to do based on the nature of the received message. For example, if a reply was expected but what arrived is not a reply, the message should be sent on the bus.

While in one of these states, it returns to ACCEPT in the next cycle. It's important to note that messages can continue to enter and exit during this process, as not all of them need to go through the SNOOP state. For instance, if a message does not have the invalid2 flag set (indicating it has already been read) and a reply has been sent, then it simply needs forwarding, which does not affect the state machine.

When any operation involving back pressure is necessary, meaning that not every cycle allows for a transmission, the node that was going to generate the transaction must abort it and retry it later.

The cache driver implements the logic of the MESI coherence protocol. In this driver, the next state of a cache line is determined based on its current state, the operations being performed (read or write), and whether the core accessing it is local or remote. Transitions occur by sending and receiving messages using the two implemented rings.

Following is a brief description of the specific connections in the interconnect:

1.2.2.1 Core-cache

The connection for communication between cores and caches is given by the implementation of Avalon Memory-Mapped Interface. This communication standard allows the transfer of data between different blocks of hardware known as Host and Agent.[6]

This interface is based on a memory addressing scheme, which offers several advantages in the context of this kind of communication.

Data is transferred between the core and the cache using read and write operations to shared memory addresses based on the state of various control signals. The core, being the host or manager (`avm` in the diagram in figure 1) in this context, it can request data stored in the cache (which is the agent or `avs` in figure 1) or write new data to it.

In this interconnect, the transaction unit is a word.

1.2.2.2 Cache-cache

Communication between caches is done by implementing the MESI protocol and it is done through a ring topology.

This interconnection consists of two unidirectional and independent rings, one for messages and the other for tokens (sections 1.2.2.2.1 and 1.2.2.2.2 respectively).

Both rings implement the Avalon Streaming (Avalon-ST) interface. This is useful for communication between components that handle high-speed, low-latency unidirectional data. [6]

The operation of the Avalon-ST is based on a constant flow of data from the transmitter to the receiver (`src` and `sink` respectively in figure 1). As data is generated or requested in a cache, control signals indicate that it should be packaged and sent across the interface. When a receiver is ready, it receives them and processes them as required.

In this case, as shown in figure 1, because of the ring topology configuration, each cache controller itself constitutes both a `src` and a `sink`. This is because each cache driver needs to be able to transmit information to the next cache driver in the ring or receive data from the previous one.

1.2.2.2.1 Message ring

This connection handles ring transactions that combine a request and a MESI reply in a single structure. It has a latency of two cycles per node.

Operations in this message ring has a TTL (time-to-live) that ensures that the transactions start and end at the node that generates them. It works as follows:

- Transactions operate with a time-to-live (TTL) that starts at a value of 3 when a node initiates it. Each time the message passes through a node, the TTL is decremented by 1.
- If a node observes that the TTL is not zero, it knows it didn't originate the transaction and processes the message according to MESI, modifies it, and retransmits it. Replies are simple modifications of flags.
- On the other hand, when a node sees that the TTL is equal to 0, the node knows it originated the transaction, making it trivial to identify whether there was a reply or not. In this case, the transaction is terminated.

Additionally, it supports multiple simultaneous transactions, as a consequence, the ring can be understood as a feedback pipeline with all the features that this implies, such as the following:

- The ring transactions are analogous to instructions
- Stalls exist (if `ready=0`)
- Dependencies and *hazards* (resolved by token ring (see 1.2.2.2.2)) exist.

Message transmission can fail due to bus collisions or if the token doesn't allow it. In such cases, the ring doesn't come to a complete halt but instead clears all the settings made for sending the message and retries the transmission.

In this message ring, priority is given to forwarding messages. This is necessary because if priority were given to other types of operations, there's a possibility of encountering deadlocks or some messages might never complete transmission.

The priority order is as follows:

1. Forwarding a message not originating from the current core.
2. Forwarding a message generated within the current core.
3. Handling debugging tasks.
4. Attending to core operations.

Each message in this ring is composed as follows:

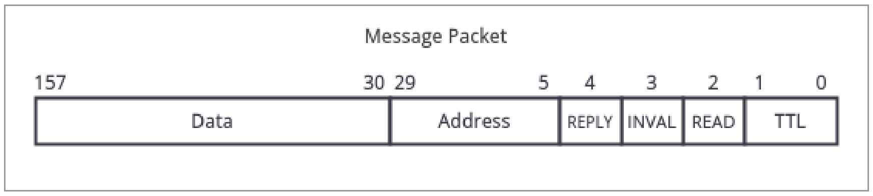


Figure 4. Message packet bit structure.

1.2.2.2.2 Token ring

The second interconnection ring's main function is to avoid inconsistencies in the message ring. Unlike the latter, the token ring does not carry transactions, but can be thought of as a kind of hazard unit.

Additionally, it never experiences *stalls* (there is no `ready` signal, since it is understood that `ready` signal is always asserted), and it has a constant period of 4 cycles.

Its operation relies on the presence of a single token that passes to each node during the execution, so that exactly one node perceives that `valid` is asserted in any given cycle. Therefore, it is said that this node "holds the token".

The token functions as a shift register of three elements (e0, e1, e2). At each clock's positive edge, the following occurs:

- Old e2 is discarded.
- e1 becomes e2 (e2 <= e1)
- e0 becomes e1 (e1 <= e0)
- The new e0 is provided by the newly traversed node

Two conditions must be met in the same cycle to initiate a transaction: the current node has the token and no valid token element matches the same line.

Upon initiating a transaction, the line in the token is locked. For this, `e0.valid` is asserted, preventing other nodes from initiating transactions on that line until it's released.

For the same line, it enforces a global and total order of ring transactions. This is important since a single ring offers neither globality nor totality and the MESI protocol needs both characteristics to be fulfilled simultaneously.

Between lines the system does not guarantee any order. However, this is not a consistency requirement. Therefore, the system is still considered as strongly consistent.

This system allows for momentarily inconsistent hits, which largely justify the usefulness of both rings. These hits are permitted by the second coherence condition and can occur whenever the MESI protocol does not require a cache-cache transaction.

There is no global ordering of core transactions, locally there is partial ordering.

The system complies with the guarantee of eventual coherence, which is time-bounded, specifically after a complete cycle of the message ring.

The token works primarily by arbitrating which core can emit a request and send it to the interconnect. It does this by only allowing the token holder to emit a message.

The token abides the following rules:

1. The token jumps to the next cache every clock cycle.
2. Only the core which holds the token can emit a new message.
3. The value of the token is shifted every clock cycle.

Each message in this ring is composed as follows:

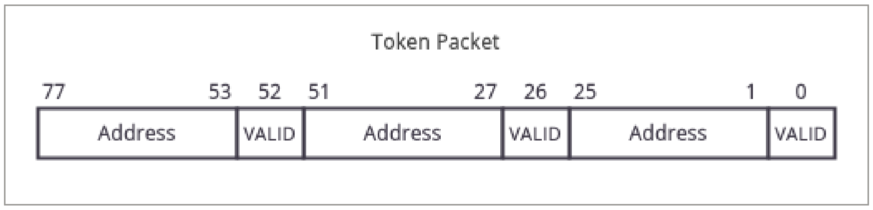


Figure 5. Token packet bit structure.

1.2.3 Cache

Each core has a private cache that consists of 4096 lines in a direct-mapped configuration. Each of these cache lines stores 4 units of 32-bit words, making the line size 128 bits (or 16 bytes). It is composed of the three main regions that are presented in figure 6:

- Data Storage: This region is used to store the actual data that is cached.
- Tags: This region stores the tags associated with each line in the cache.
- Cache Line Status: This region keeps track of the state of each cache line, indicating whether the data is invalid, modified, exclusive or shared, depending on the MESI protocol.

Having these three regions in the cache allows the core to efficiently manage and access cached data while maintaining cache coherence and consistency.

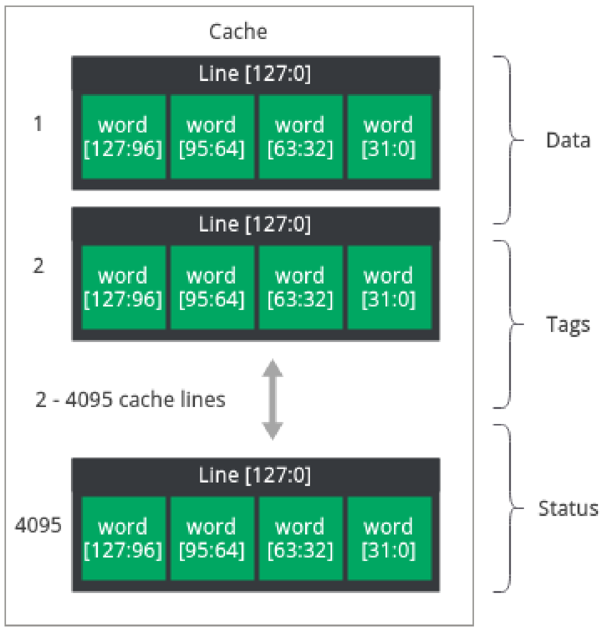


Figure 6. Cache structure.

The core only performs aligned operations, each 32-bit address has the following structure:

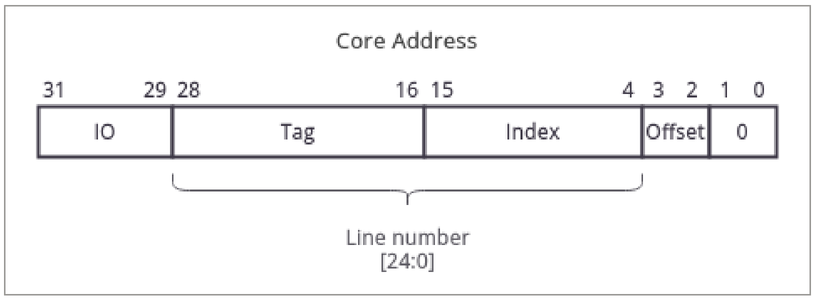


Figure 7. Core address bit structure.

These last 3 bits of IO indicate whether the address corresponds to a cache or not. This is possible because the RAM memory size is a power of 2, and its addresses start at zero. Therefore, based on the analysis of the 3 most significant bits, it will be possible to determine whether the address is for the cache or for I/O.

A series of modules were implemented to aid the operation of the cache:

1.2.3.1 Cache offset

This module serves to simplify offsets, making them transparent to the cache. Since the core operates in words while the cache operates in lines, this module acts as the bridge between these two data types. In this way, the cache never sees the offset part present in the addresses.

1.2.3.2 Cache routing

This module is designed for routing operations either to the cache or to memory. This is necessary because there are certain write operations that should not be cached under any circumstances.

An example of this is with peripherals, where if their values are stored in the cache rather than in memory, subsequent reads would result in incorrect data since the correct values are held in the cache. This module ensures that operations are directed appropriately to avoid coherency issues and to guarantee that specific critical data is written directly to memory instead of being cached.

1.2.3.3 Cache monitor

This module provides the capability for spin locks, which are essentially mutexes used to protect a code section through spin lock/unlock operations. It implements ARM instructions like ldrex and strexeq, which are not originally part of ARMv4. This implies that the core is a combination of ARMv4 and ARMv6.

This functionality is necessary to handle situations where multiple cores attempt to access the same variable simultaneously. In such scenarios, a locking system must be established to ensure that only one core at a time can access a critical code region, preventing conflicts and ensuring data integrity.

1.2.3.4 Cache debug

The debug module takes a cache address and captures the current state of the cache to return the data (each of the words) from the queried line, as well as metadata about the line, such as its current MESI state, whether it's cached, the index, tag, and more.

This module has the particularity of having an interface with the central Avalon MM interconnect and with its respective cache driver. This implies that each cache driver has an interface with itself.

1.2.4 Performance Unit

This module is responsible for monitoring a CPU in order to obtain various performance counters, such as the following:

- Requests (sends, forwards, replies)
- Requests sent (read, invalid, read invalid)
- Memory (misses, writebacks)
- Uncached I/O (reads, writes)
- Ring cycles (minimum, maximum)
- Memory read cycles (minimum, maximum)
- Memory write cycles (minimum, maximum)

1.2.5 SMP Controller

This controller is responsible for coordinating the execution of the various processing elements in the system. When the system boots up, only one of the cores (PE0) initiates its operation by setting its halt signal low. Therefore, this core is designated as the central node to orchestrate the execution of the others.

Each core can run normally, step by step, or be in a completely halted state. Additionally, cores can send signals to trigger a change in the execution state of others.

One of these signals is the breakpoint signal. If a request of this type is received, the halt signal is raised, indicating that the processor must stop, and the step signal is lowered. It's important to note that the halt is in effect until the next cycle after the breakpoint request to stop the execution at the precise point where it should be halted.

On the other hand, if a signal requesting step-by-step execution is received, the halt signal is lowered to continue processing, and the step signal is raised to indicate the mode in which it should continue.

In this way, the execution of the processing elements in the system is controlled.

1.3 System components

1.3.1 Core

ARMv4 processor with additional implementation of ldrex and strexeq for multicore support.

1.3.2 Interconnect

1.3.2.1 Core-cache interconnect

Avalon MM Communication Interface with a word as transaction unit.

1.3.2.2 Cache-cache ring interconnect (MESI protocol)

Avalon ST Communication Interface.

1.3.2.2.1 Message ring

This connection carries ring transactions that combine a request and a MESI reply in a single structure. Its latency is two cycles per node.

The structure is composed as follows:

- TTL (Time-to-live) that starts in 3
- 12 bits of index
- 13 bits of tag

- Bits of request: `read`, `invalidate` (both or only one can be raised)
- Bit of reply.
- Line data, significant only if `read=1` and `reply=1`.

1.3.2.3 Token ring

The token consists of three elements (e0, e1, e2) in a shift register, and each of them has the following structure:

- 1 bit of Valid (do not confuse with valid of sinks/sources)
- 12 bits of index
- 13 bits of tag

The token ring has a constant period of 4 cycles (it does not have back pressure).

1.3.3 Cache

Each core has a cache memory with the following specifications:

- Storage size of 64 KiB.
- Direct-mapped (one way).
- 4096 cache lines for each of the three cache sections (data, tag, state).
- Line size of 16 bytes.
- 4 words of 32 bits per line.
- Each cache line can have one of these states: `INVALID, SHARED, EXCLUSIVE, MODIFIED`. (Dirty and valid bits are considered implicit in these states)
- Single read/write port.

Since the core only performs aligned operations, each 32-bit address consists of:

- 2 bits that are always 0 (translated to byteenable by core).
- 2 bits offset bits (since the cache's addressable unit is the word)
- 12 bits of index
- 13 bits of tag
- 3 bits that are == 0 if cached, != 0 if uncached

These last 3 bits of IO indicate whether the address corresponds to a cache or not. This is possible because the RAM memory size is a power of 2, and its addresses start at zero. Therefore, based on the analysis of the 3 most significant bits, it will be possible to determine whether the address is for the cache or for I/O.

1.3.3.1 Cache driver

Each cache driver has a total of 7 interfaces:

- 2 source interfaces (one for each ring).
- 2 sink interfaces (one for each ring).
- 1 interface with its respective core.
- 1 for debugging.
- 1 with memory.

2 Testing and simulation

2.1 Reference model

The reference model consists of two simulations written in python, available in `tb/models`:

- **Message ring**: Simulation of a segment of message ring, has a queue of messages that enter the cache and exit it. This simulates the full behavior if the cache controller when it receives messages.
- **SMP controller**: Simulation of the SMP controller. It can simulate the action of halting and running a CPU.
- Core: Single core simulation.

2.2 Test plan and coverage

2.2.1 Test plan

- testRING: Takes the ring model, instantiates it 4 times, synthesizes it with Verilator 4 times. Each of the four segments are used by actors, which sends messages randomly and a scoreboard checks that there are no faults in the system.
- testSMP: Tests the SMP controller by allowing the halting and running of cores.
- SMP Boot: Starts all four cores and makes the generate contention in a single cache line.
- SDRAM: Tests correctness in SDRAM writes.
- Strex: Tests atomic operations by executing them and checking for expected behavior.
- Subword: Checks for byte-enable operations.
- Shift: Checks for all shifts available in ARMv4 architecture.

2.3 Simulations

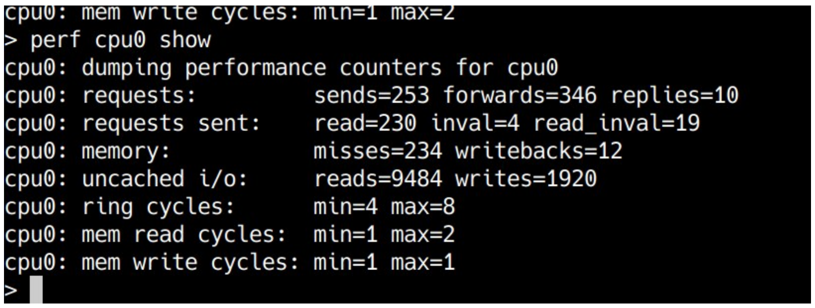
Simulation at the block level is used for testing the proposed models.

On the other hand, system-level simulations are much more complex, as they involve a hand-written C++ implementation of the entire system. This implementation is integrated with Verilator to run system-wide simulations. Simulation allows for the execution of programs that correspond to the tests.

There are integration tests (listed in the test plan) and also functional tests.

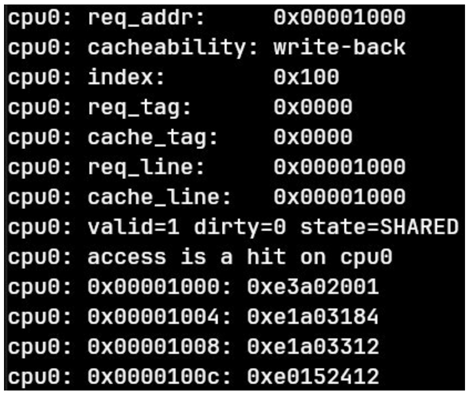
The demo is a C++ simulation like the one mentioned earlier, running the DEMO program.

2.4 Debug and monitoring/counting logic examples



```
cpu0: mem write cycles: min=1 max=2
> perf cpu0 show
cpu0: dumping performance counters for cpu0
cpu0: requests:          sends=253 forwards=346 replies=10
cpu0: requests sent:     read=230 inval=4 read_inval=19
cpu0: memory:            misses=234 writebacks=12
cpu0: uncached i/o:      reads=9484 writes=1920
cpu0: ring cycles:       min=4 max=8
cpu0: mem read cycles:   min=1 max=2
cpu0: mem write cycles:  min=1 max=1
> 
```

Figure 8. Performance unit's output.



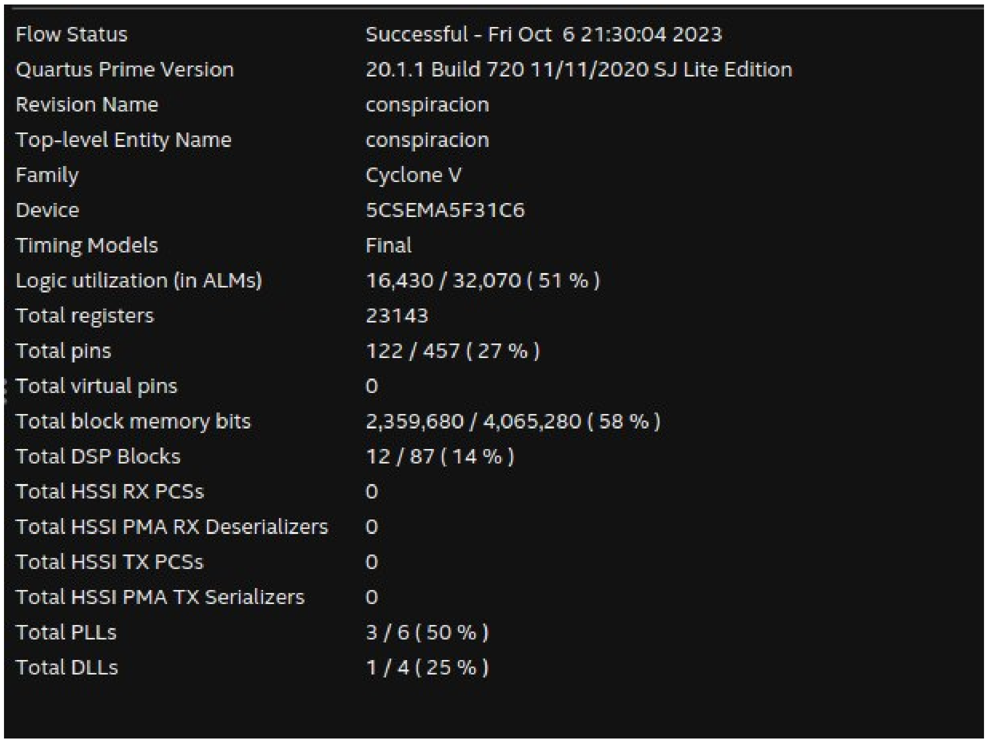
```
cpu0: req_addr:      0x00001000
cpu0: cacheability: write-back
cpu0: index:         0x100
cpu0: req_tag:       0x0000
cpu0: cache_tag:     0x0000
cpu0: req_line:      0x00001000
cpu0: cache_line:    0x00001000
cpu0: valid=1 dirty=0 state=SHARED
cpu0: access is a hit on cpu0
cpu0: 0x00001000: 0xe3a02001
cpu0: 0x00001004: 0xe1a03184
cpu0: 0x00001008: 0xe1a03312
cpu0: 0x0000100c: 0xe0152412
```

Figure 9. Cache line dump.

2.5 Results

2.5.1 Quartus results

- We used a DE-1-SoC which has 3270ALMs, of which we used 16430, which corresponds 51% of usage.
- 58% of SRAM was used
- 3 PLLs
- 12 DSP blocks
- 80Mhz Core/cache frequency
- 50Mhz bus frequency
- No timing violations



```
Flow Status                        Successful - Fri Oct  6 21:30:04 2023
Quartus Prime Version              20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name                      conspiracion
Top-level Entity Name              conspiracion
Family                             Cyclone V
Device                             5CSEMA5F31C6
Timing Models                      Final
Logic utilization (in ALMs)        16,430 / 32,070 ( 51 % )
Total registers                    23143
Total pins                         122 / 457 ( 27 % )
Total virtual pins                 0
Total block memory bits            2,359,680 / 4,065,280 ( 58 % )
Total DSP Blocks                   12 / 87 ( 14 % )
Total HSSI RX PCSs                 0
Total HSSI PMA RX Deserializers    0
Total HSSI TX PCSs                 0
Total HSSI PMA TX Serializers      0
Total PLLs                         3 / 6 ( 50 % )
Total DLLs                         1 / 4 ( 25 % )
```

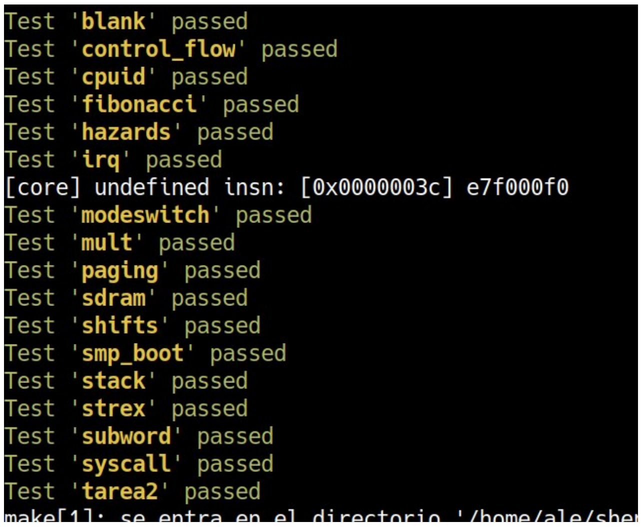
Figure 10. Quartus results report.

2.5.2 Linux results

- 5.19 bogomips

2.5.3 Block and integration results

All tests were successful, see attached logs, reports and traces.



```
Test 'blank' passed
Test 'control_flow' passed
Test 'cpuid' passed
Test 'fibonacci' passed
Test 'hazards' passed
Test 'irq' passed
[core] undefined insn: [0x0000003c] e7f000f0
Test 'modeswitch' passed
Test 'mult' passed
Test 'paging' passed
Test 'sdram' passed
Test 'shifts' passed
Test 'smp_boot' passed
Test 'stack' passed
Test 'strex' passed
Test 'subword' passed
Test 'syscall' passed
Test 'tarea2' passed
make[1]: se entra en el directorio '/home/ale/she
```

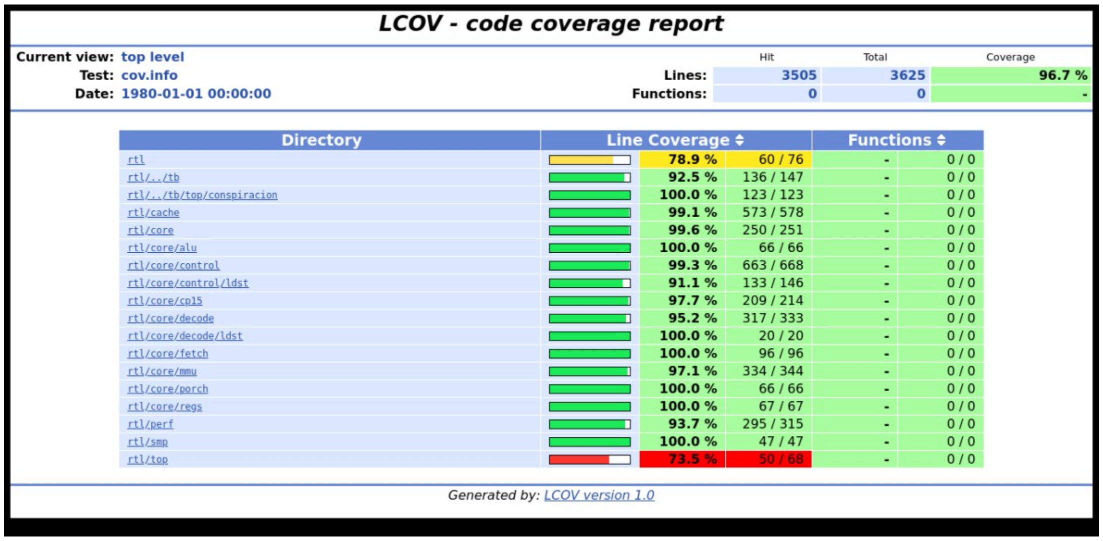
Figure 11. Integration results.

2.5.4 Coverage

Automatic line-level coverage was generated using Verilator.

The following figure 12 provides details of the code coverage report for the system.

Coverage percentage is 96.%



LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level		Lines:	3505	3625	96.7 %
Test:	cov.info		Functions:	0	0	-
Date:	1980-01-01 00:00:00					

Directory	Line Coverage			Functions	
rtl		78.9 %	60 / 76	-	0 / 0
rtl/../tb		92.5 %	136 / 147	-	0 / 0
rtl/../tb/top/conspiracion		100.0 %	123 / 123	-	0 / 0
rtl/cache		99.1 %	573 / 578	-	0 / 0
rtl/core		99.6 %	250 / 251	-	0 / 0
rtl/core/alu		100.0 %	66 / 66	-	0 / 0
rtl/core/control		99.3 %	663 / 668	-	0 / 0
rtl/core/control/ldst		91.1 %	133 / 146	-	0 / 0
rtl/core/cp15		97.7 %	209 / 214	-	0 / 0
rtl/core/decode		95.2 %	317 / 333	-	0 / 0
rtl/core/decode/ldst		100.0 %	20 / 20	-	0 / 0
rtl/core/fetch		100.0 %	96 / 96	-	0 / 0
rtl/core/mmu		97.1 %	334 / 344	-	0 / 0
rtl/core/porch		100.0 %	66 / 66	-	0 / 0
rtl/core/regs		100.0 %	67 / 67	-	0 / 0
rtl/perf		93.7 %	295 / 315	-	0 / 0
rtl/smp		100.0 %	47 / 47	-	0 / 0
rtl/top		73.5 %	50 / 68	-	0 / 0

Generated by: LCOV version 1.0

Figure 12. System testing coverage.

3 References

[1] M. Thompson, *What are alms, les and aluts?* 2013. [Online]. Available: https://electronics.stackexchange.com/questions/81312/what-are-alms-les-and-aluts.

[2] M. R. Marty and M. D. Hill, *Coherence ordering for ring-based chip multiprocessors*, 2006. [Online]. Available: https://research.cs.wisc.edu/multifacet/papers/micro06_ring.pdf.

[3] L. A. Barroso and M. Dubois, "The performance of cache-coherent ring-based multiprocessors," vol. 29, no. 6, pp. 61–72, 1993. DOI: 10.1145/165123.165162. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/165123.165162.

[4] WikiChip, *Mesh interconnect architecture - intel*, 2021. [Online]. Available: https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture.

[5] Wikipedia, *Token ring*, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Token_Ring.

[6] Intel, *Avalon® interface specifications*, 2022. [Online]. Available: https://cdrdv2-public.intel.com/667068/mnl_avalon_spec-683091-667068.pdf.