**Project Book – Multicore DLX Processor with MESI Coherence Protocol**

**Project Number:** 24-1-1-3024

**Student:** Yohai Shiloh  
**ID:**   
**Student:** Yarin Koren  
**ID:**

**Supervisor:** Oren Ganon

**Project Carried Out at:** Faculty of Engineering, Tel Aviv University

Index

[Figures List 3](#_Toc204642253)

[Tables List 4](#_Toc204642254)

[Abstract 5](#_Toc204642255)

[1. Introduction 6](#_Toc204642256)

[Goals of the Project 6](#_Toc204642257)

[Motivation 6](#_Toc204642258)

[Approach 6](#_Toc204642259)

[Comparison to Other Multicore Protocols 6](#_Toc204642260)

[2. Theoretical Background 7](#_Toc204642261)

[Cache 7](#_Toc204642262)

[The MESI Protocol 7](#_Toc204642263)

[Cache Coherence in Shared Bus Architectures 8](#_Toc204642264)

[Instruction Memory Unit 8](#_Toc204642265)

[General Design Guidelines and Execution Method 8](#_Toc204642266)

[3. Simulation 9](#_Toc204642267)

[Environment 9](#_Toc204642268)

[Simulation Objectives 9](#_Toc204642269)

[Key Observations 9](#_Toc204642270)

[Execution example – Bubble Sort 9](#_Toc204642271)

[Summary of Simulation Coverage: 13](#_Toc204642272)

[4. Implementation 14](#_Toc204642273)

[4.1 Hardware Description 16](#_Toc204642274)

[Detailed explanation: 17](#_Toc204642275)

[4.2 Software Description 25](#_Toc204642276)

[5. Analysis of Results 26](#_Toc204642277)

[Functional Verification 26](#_Toc204642278)

[Performance Metrics 26](#_Toc204642279)

[Comparison to other Cache Coherence Protocols: MESI vs MOESI, MSI, and Dragon 28](#_Toc204642280)

[6. Conclusions and Further Work 30](#_Toc204642281)

[Achievements 30](#_Toc204642282)

[Improvements Observed 30](#_Toc204642283)

[Further Work 30](#_Toc204642284)

[Takeaways 30](#_Toc204642285)

[7. Project Documentation 31](#_Toc204642286)

[User guide 31](#_Toc204642287)

[8. References 32](#_Toc204642288)

[Appendix A. Inputs outputs tables 33](#_Toc204642289)

[Appendix B. The Simplified DLX Instruction Set Architecture 40](#_Toc204642290)

# Figures List

[Figure 1 : System Block Diagram 5](#_Toc204636049)

[Figure 2 : MESI State Diagram 7](#_Toc204636050)

[Figure 3: Example of execution through simulation 9](#_Toc204636051)

[Figure 4: Multicore architecture's workflow diagram 14](#_Toc204636052)

[Figure 5 : Multicore architecture Top Level diagram 14](#_Toc204636053)

[Figure 6 : Core architecture diagram 15](#_Toc204636054)

[Figure 7: Cache structure diagram 15](#_Toc204636055)

[Figure 8: Cache DataPath architecture diagram 16](#_Toc204636056)

[Figure 9: Bus Control Unit 17](#_Toc204636057)

[Figure 10: bus control state diagram 18](#_Toc204636058)

[Figure 11: Core Unit 18](#_Toc204636059)

[Figure 12: Fetch State Check Unit 18](#_Toc204636060)

[Figure 13: Cache Unit 19](#_Toc204636061)

[Figure 14: AS\_N – DLX control signal DEMUX Unit 19](#_Toc204636062)

[Figure 15: Instruction Memory Unit 19](#_Toc204636063)

[Figure 16: DLX data in MUX Unit 20](#_Toc204636064)

[Figure 17: ACK signal MUX Unit 20](#_Toc204636065)

[Figure 19: Cache control state diagram 21](#_Toc204636066)

[Figure 18: Cache Control Unit 21](#_Toc204636067)

[Figure 20: Address Calculator Unit 22](#_Toc204636068)

[Figure 21: Cache DataPath Unit 22](#_Toc204636069)

[Figure 22: Tag Memory flow diagram 23](#_Toc204636070)

[Figure 23: Tag Memory Unit 23](#_Toc204636071)

[Figure 24: Data In MUX Unit 24](#_Toc204636072)

[Figure 25: Cache Data Memory Unit 24](#_Toc204636073)

[Figure 26: Programs runtime graph 26](#_Toc204636074)

[Figure 27: Speedup ratio graph 27](#_Toc204636075)

[Figure 28: Execution example – the array before sorting 28](#_Toc204636076)

[Figure 29: Execution example – the array after sorting 28](#_Toc204636077)

# Tables List

[Table 1: Verification coverage 13](#_Toc204636031)

[Table 2: Run time on each processor for different programs 26](#_Toc204636032)

[Table 3: Speedup ratio for any two processors 27](#_Toc204636033)

[Table 4: Different coherence protocol comparison 29](#_Toc204636034)

[Table 5: Bus Control Unit Input Output table 33](#_Toc204636035)

[Table 6: Core Unit Input Output table 33](#_Toc204636036)

[Table 7: Fetch State Check Unit Input Output table 34](#_Toc204636037)

[Table 8: Cache Unit Input Output table 34](#_Toc204636038)

[Table 9: AS\_N – DLX control signal DEMUX Unit Input Output table 35](#_Toc204636039)

[Table 10: Instruction Memory Unit Input Output table 35](#_Toc204636040)

[Table 11: DLX data in MUX Unit Input Output table 35](#_Toc204636041)

[Table 12: ACK signal MUX Unit Input Output table 36](#_Toc204636042)

[Table 13: Cache Control Unit Input Output table 36](#_Toc204636043)

[Table 14: Address Calculator Unit Input Output table 37](#_Toc204636044)

[Table 15: Cache DataPath Unit Input Output table 37](#_Toc204636045)

[Table 16: Tag Memory Unit Input Output table 38](#_Toc204636046)

[Table 17: Data in MUX Unit Input Output table 39](#_Toc204636047)

[Table 18: Cache Data Memory Unit 39](#_Toc204636048)

# Abstract

This project extends the classical DLX architecture (which was implemented during the ACSL course) into a dual-core multicore system, addressing performance limitations by implementing parallelism and cache coherence. The project enhances a single-core DLX CPU by adding instruction memory and data caches to each core and introducing a cache coherence mechanism based on the MESI protocol. This ensures data consistency across local caches and enables scalable parallel processing.

A physical implementation was also performed using an FPGA board and the RESA environment, validating both functionality and performance.

The project includes a Verilog based implementation, using both code files and schematic files into a full processor functional design, while supporting the original architecture and instruction set of the DLX processor.

A diagram of a computer program

AI-generated content may be incorrect.

Figure 1 : System Block Diagram

# 1. Introduction

## Goals of the Project

* Extend a single-core DLX processor to a parallel, dual-core system.
* Reduce access time and improve memory bandwidth through per-core instruction and data caches.
* Ensure correctness through the implementation of the MESI coherence protocol.

## Motivation

Traditional single-core systems face bottlenecks when attempting to improve performance solely through higher frequencies. Multicore architectures address these challenges by enabling concurrent execution of multiple threads. However, this requires careful management of shared memory and coherence to avoid erroneous behavior, e.g. using an invalid memory that has been modified by other thread, etc.

The use of MESI enables efficient sharing and synchronization of data between caches while reducing unnecessary communication with main memory.

## Approach

We reused the basic DLX core structure developed in the ACSL course[[1]](#footnote-1) and expanded it into a multicore platform with the following key additions:

* Independent instruction memory per core to avoid bus contention.
* Data caches for each core.
* A shared bus with arbitration logic.
* A MESI state machine per cache controller.
* A bus monitor to support snooping.

## Comparison to Other Multicore Protocols

Whereas some simpler architectures use a shared memory without coherence mechanisms, our design integrates MESI to ensure correctness in data sharing. Compared to directory-based coherence models, MESI provides a balance between complexity and efficiency, suitable for small-scale multicore systems.  
More broad comparison is done in section 5.

# 2. Theoretical Background

## Cache

The processor uses cache memory to accelerate access to data and instructions. Cache is a small, fast storage located close to the CPU, designed to hold recently accessed data or data predicted to be reused soon. In our system, each core has a separate instruction cache and data cache, allowing independent memory access and avoiding contention. In addition, each cache is connected to a MESI controller that ensures data consistency across the multi-core system.

The cache address mapping is based on dividing the original addresses into three parts – tag, which identifies the block in the cache compared to the main memory and share the higher bits of the address in the main memory; index, which determine the location of the block in the cache and offset – the requested word inside the block.

## The MESI Protocol

MESI (Modified, Exclusive, Shared, Invalid) is a write-invalidate coherence protocol. It ensures consistency across caches through four possible states per cache block:

* **Modified (M):** This cache and this cache only has the only valid copy, and it has been changed from main memory.
* **Exclusive (E):** This cache has the only copy, but it matches the data in the main memory.
* **Shared (S):** The block may be in multiple caches, and it matches the data in the main memory.
* **Invalid (I):** The block is not valid.

תמונה שמכילה טקסט, תרשים, עיגול, קו

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.State transitions are driven by processor-initiated actions (e.g., read or write) and snooping on bus actions from other processors. This protocol reduces traffic by avoiding unnecessary writes to memory and enabling cache-to-cache transfers.

Figure 2 : MESI State Diagram

## Cache Coherence in Shared Bus Architectures

All caches snoop bus transactions to keep track of changes to shared data. Each cache determines its actions based on the bus signals (BusRd, BusRdX, etc.), enabling decisions about exclusivity.

This passive coherence mechanism avoids the complexity of directory tracking and is ideal for dual-core systems.

## Instruction Memory Unit

Each core is assigned a dedicated instruction memory module to avoid contention during instruction fetch. These ROM-based memories are preloaded using custom compilation and initialization tools, enabling both deterministic performance and independent operation of the cores.

## General Design Guidelines and Execution Method

The basic engineering point of view was to take the original DLX as a "close box" and build the advanced multicore processor to wrap around it. As a result, the execution flow of the processor includes the following steps:

* The DLX processor fetches an instruction from the instruction memory unit
* If there is an approach to data (e.g. read or write) the DLX addresses the cache to check if the data is available inside the cache.
* If not, the cache will start a transaction to the main memory to import/export data from it
* If the required data is available only in cache of another core (when the block is in the Modified mode), it will be imported from this cache, while simultaneously updating the main memory

# 3. Simulation

The project included a suite of simulations aimed at verifying functionality and coherence.

## Environment

The simulations were written in Verilog and run using Xilinx's Isim platform. Each DLX core ran independently, executing test programs involving shared memory.

The simulation was executed using Verilog based text-bench, performing various

## Simulation Objectives

* Validate the functional correctness of the new additions (the cache, instruction memory etc.)
* Validate MESI protocol transitions.
* Ensure correctness of shared data accesses.
* Measure cycle counts and identifies speedup.

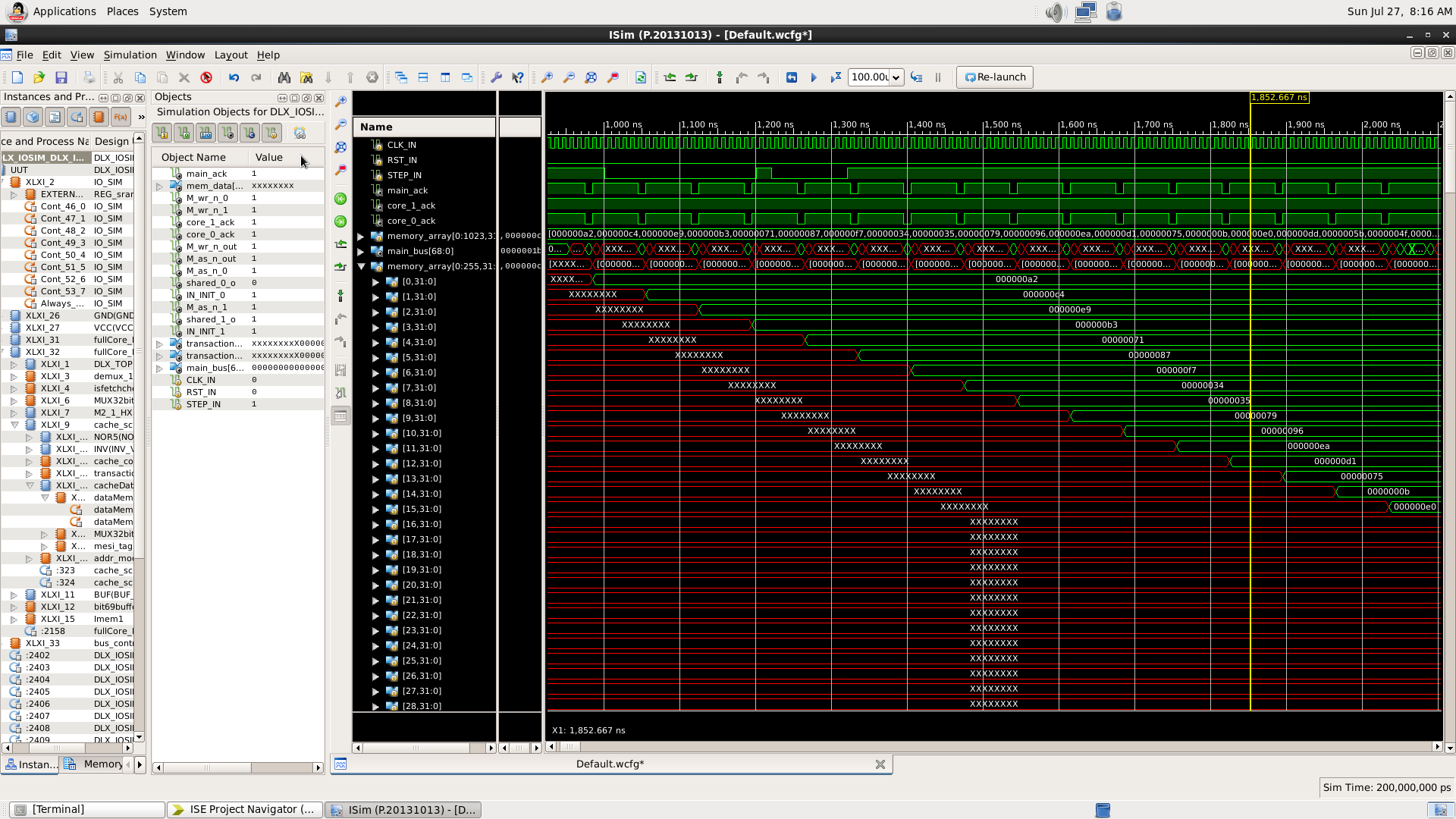
### Key Observations

* Cache state transitions behaved as expected.
* The shared line was correctly asserted during shared reads.
* Write invalidation occurred precisely when needed.

### Execution example – Bubble Sort

To demonstrate the work of the project, below is attached the execution of one of the programs, the bubble sort program. To avoid excessive length, only key stages are shown, and only main signals and data values are presented.

Loading block into cache No. 0:

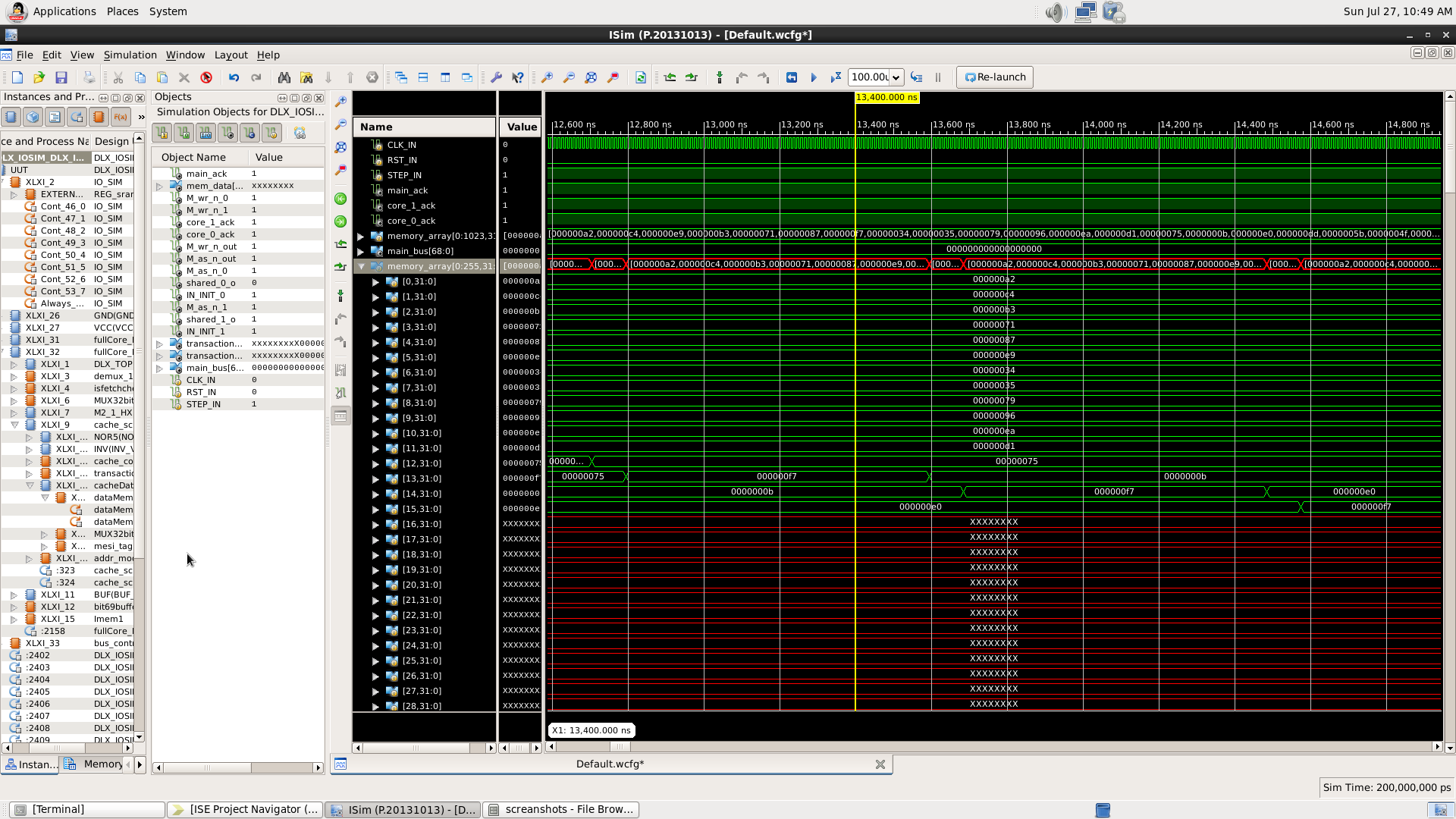
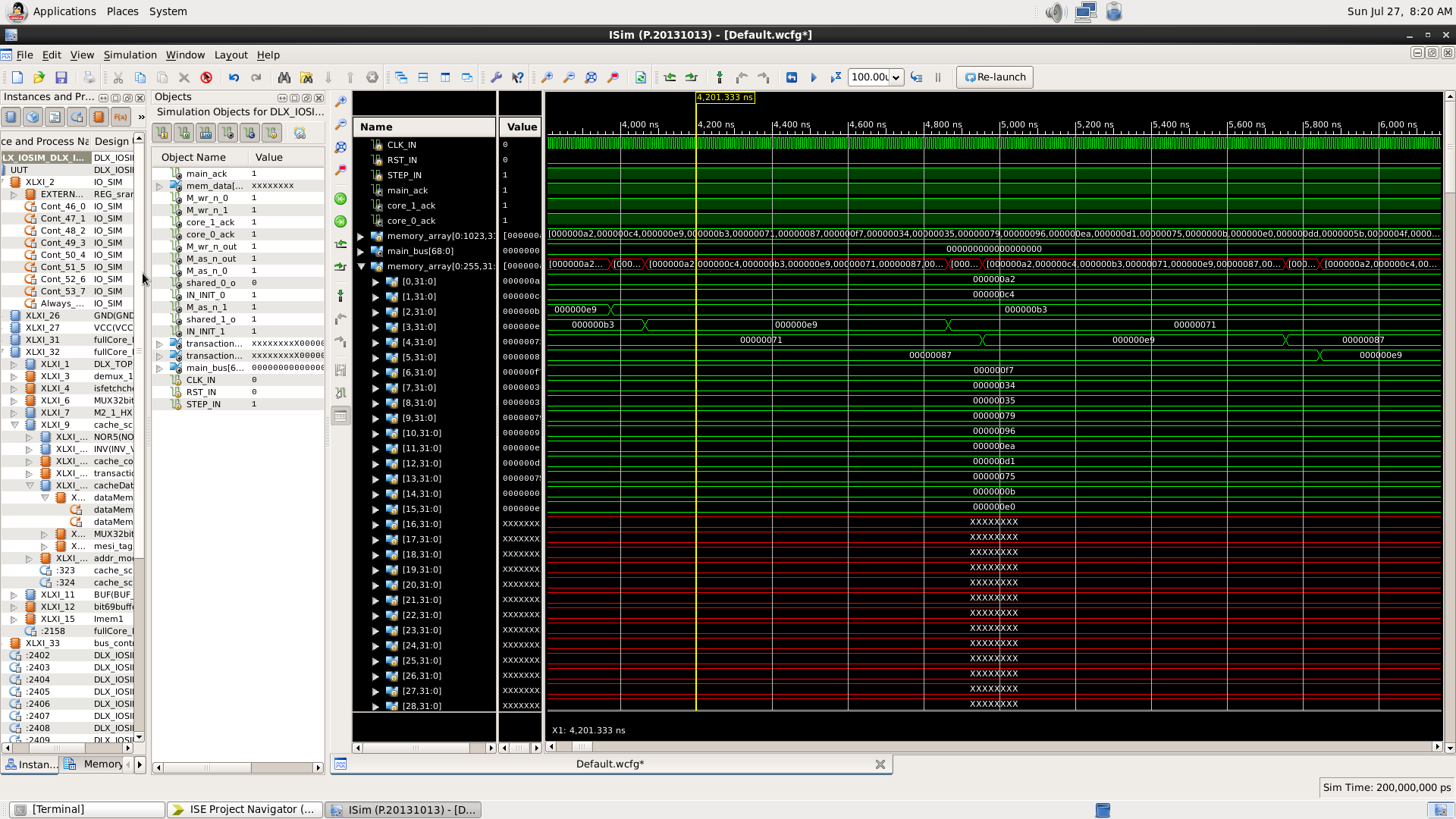


The memory of cache No. 0

Last word in the block arrives

Figure 3: Example of execution through simulation

Starting the sort in core 0:   
Finding the biggest word(000000f7) and moving it to the last place

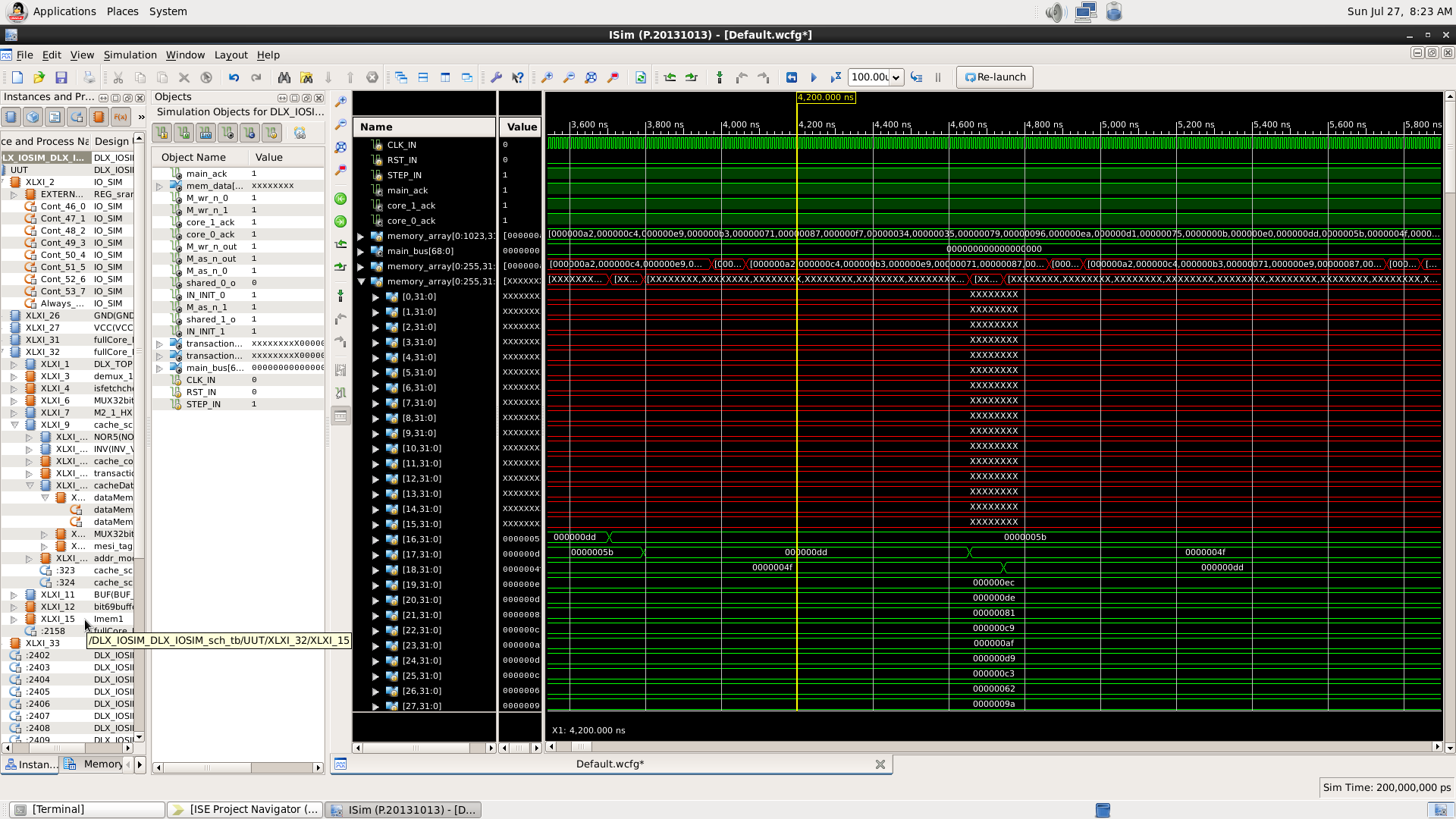


Two first words in order already

000000e9 is moved down until bigger word is found

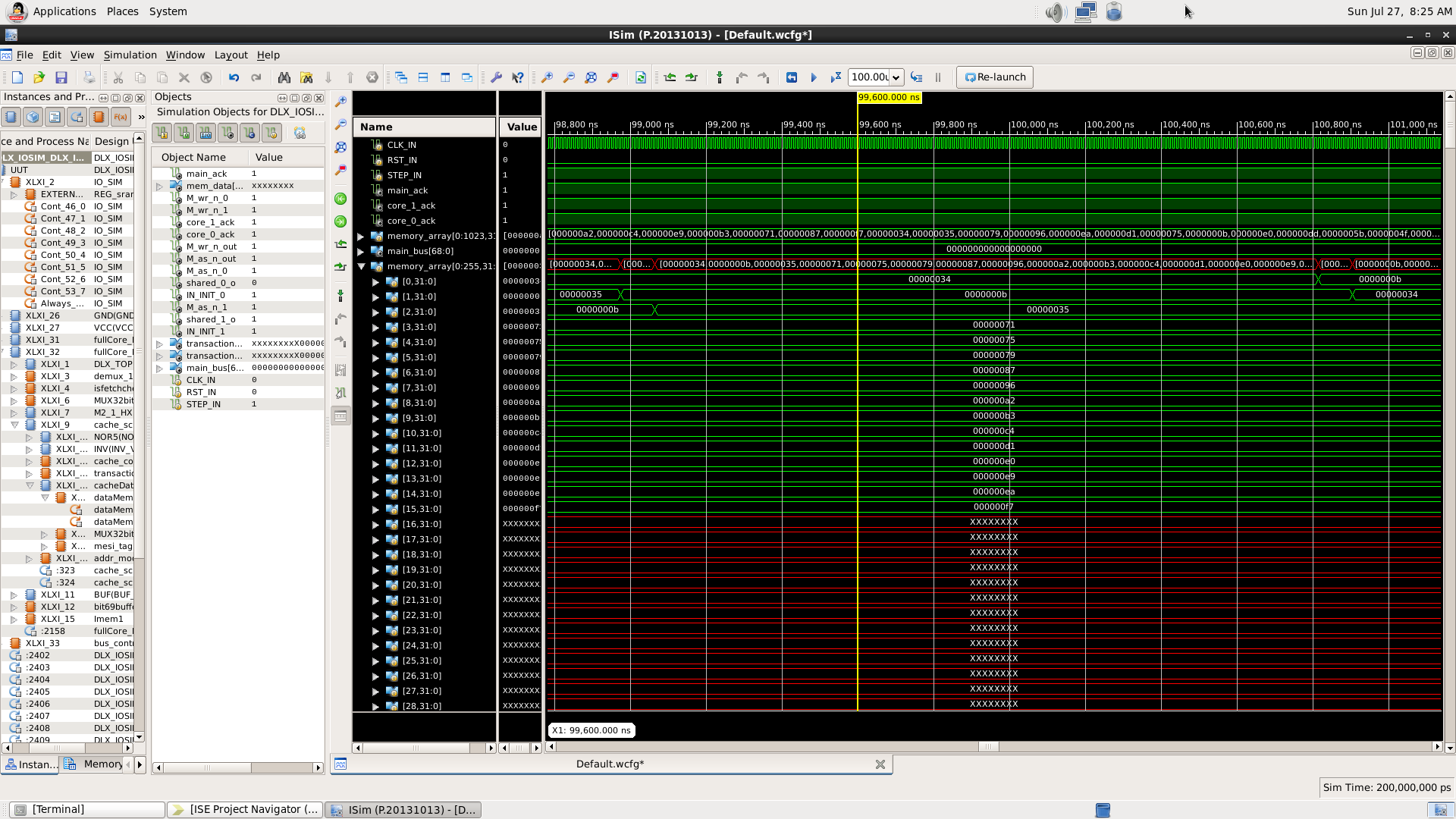
000000f7 placed as the last word in the array

Start sorting in core 1(simultaneously):

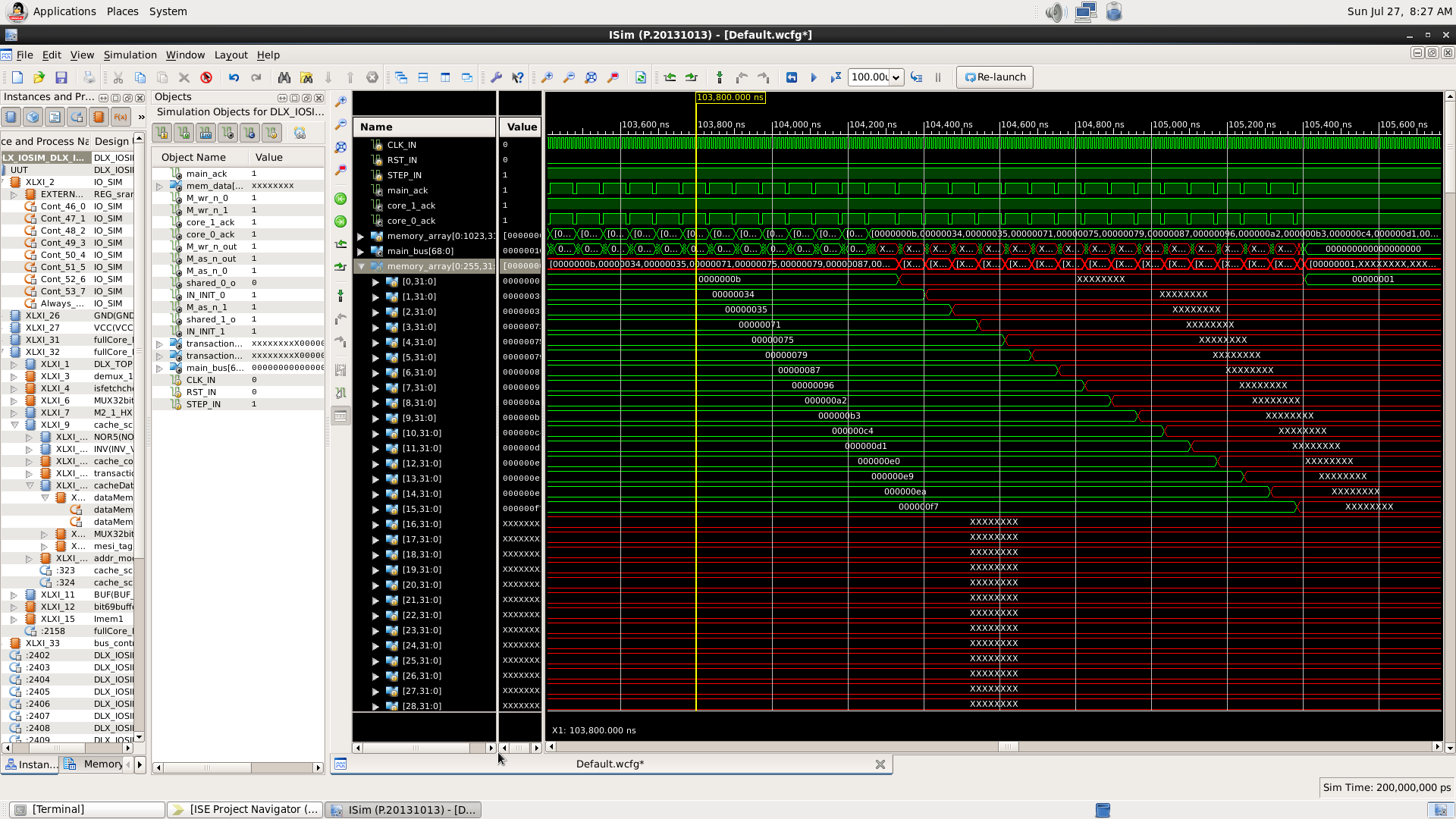


The memory of cache No. 1

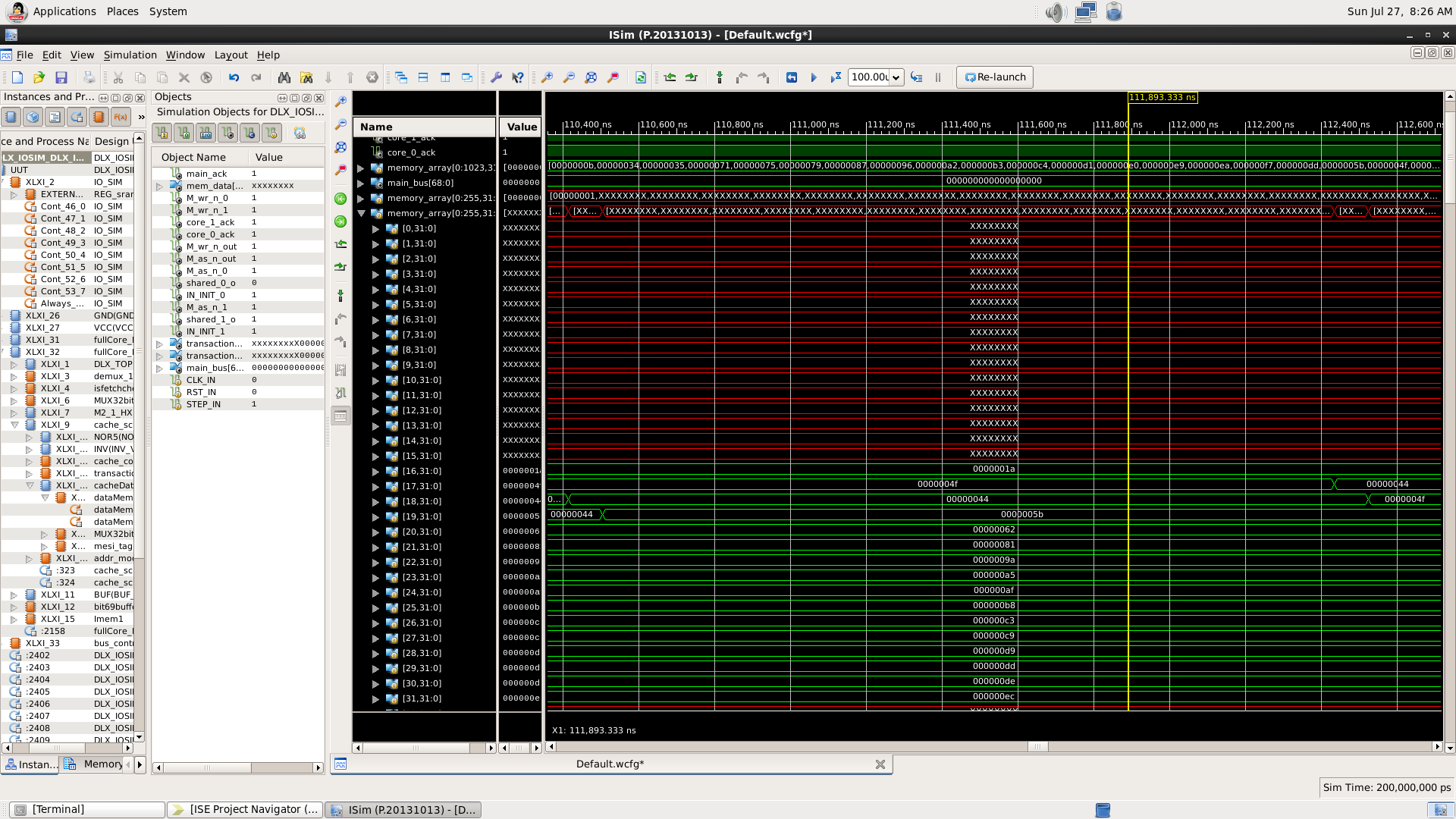
Done the sort in core 0:



At this point, the block in cache 0 is fully sorted

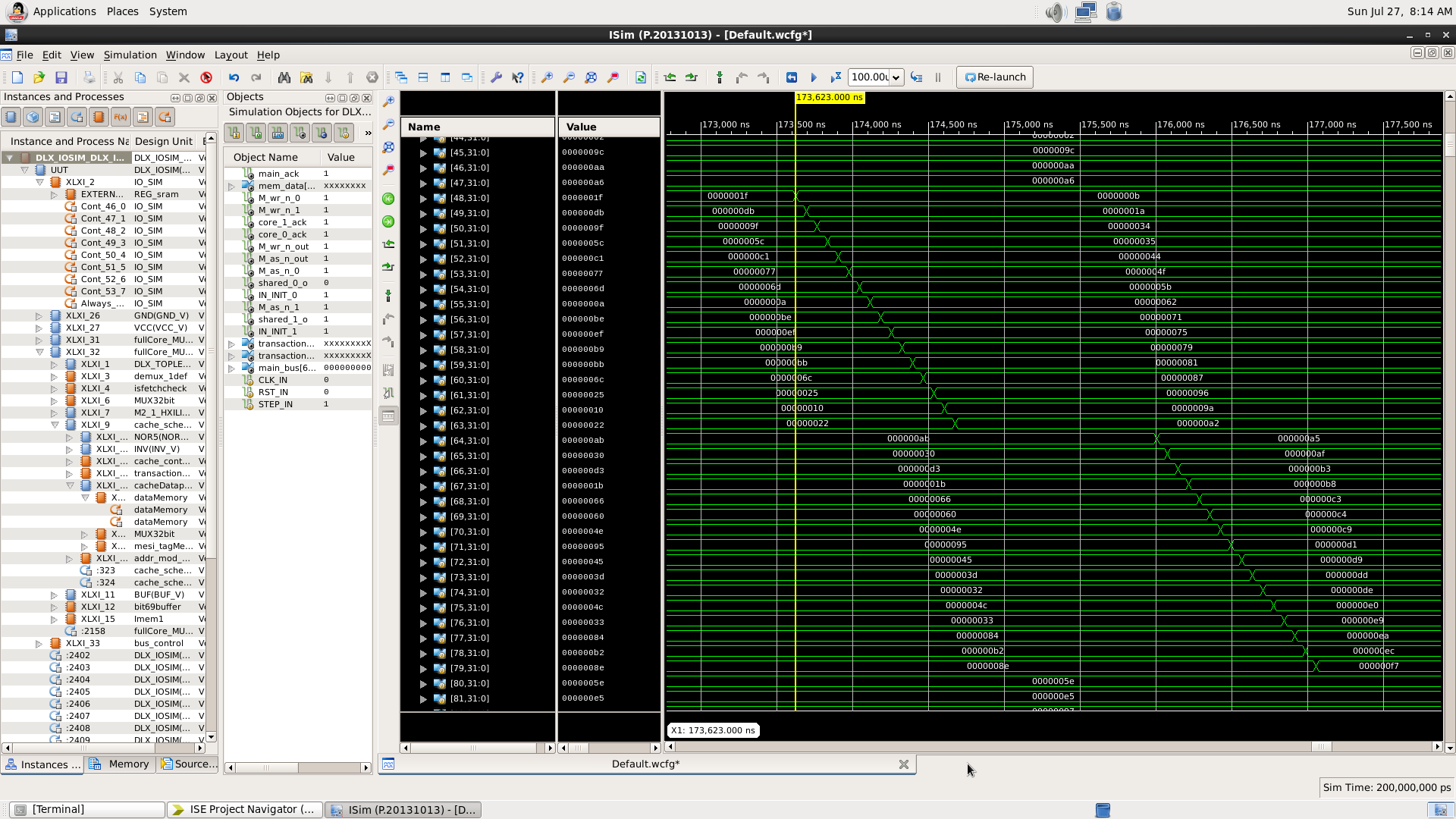
Clearing the block from cache No. 0 to the main memory:

Done sorting in core 1:



At this point, the block in cache 1 is fully sorted

Merging the two sorted blocks into a fully sorted array and saving it in the main memory (in addresses 48-79):



The first block is saved in the main memory

The second block is saved in the main memory

The main memory

## Summary of Simulation Coverage:

Using the simulation and special Assembly code (see section 4.2), the design was validated using coverage table, a common method to verify the functionality of a system[[2]](#footnote-2):

Table 1: Verification coverage

|  |  |  |  |
| --- | --- | --- | --- |
| **Unit under test** | **Test/Tests** | **Tested by -** | **Notes** |
| **Instruction memory unit** | Small input handling | Very short program |  |
| Very large input handling | Very long program |  |
| Parallel execution by multiple cores from separated units | Parallel programs |  |
| **Cache unit** | Data transmission from the main memory | Read/Write instructions on missing blocks ("cache miss") | On single-core DLX with cache extension[[3]](#footnote-3) |
| Data transmission to the core | Read/Write instructions on existing blocks ("cache hit") | On single-core DLX with cache extension |
| Updating data in the main memory | Read/Write instructions on missing blocks, while its place in the cache is occupied ("clearing "dirty" block") | On single-core DLX with cache extension |
| **MESI Unit** | Transition I to S, E, M | Read/Write instructions |  |
| Transition S to I | Read/Write was performed by another core |  |
| Transition S to E, M | Read/Write instructions |  |
| Transition E, M to I, S | Read/Write was performed by another core |  |
| Transition E, M to E, M | Read/Write instructions | Both by read/write to the same block and by calling to another block |
| **Bus controller** | Correct arbitration between the cores | Various combination of bus usage request from the cores in every possible order |  |
| Data transmission |  |  |
| Signal transmission | IN\_INIT, ack\_N, wr\_N etc. from/to each MESI unit and main memory |  |
| Snooping | Read/Write instructions | This test is for both bus control and MESI units |

# 4. Implementation

The design implements the following workflow as the general operation of the processor:

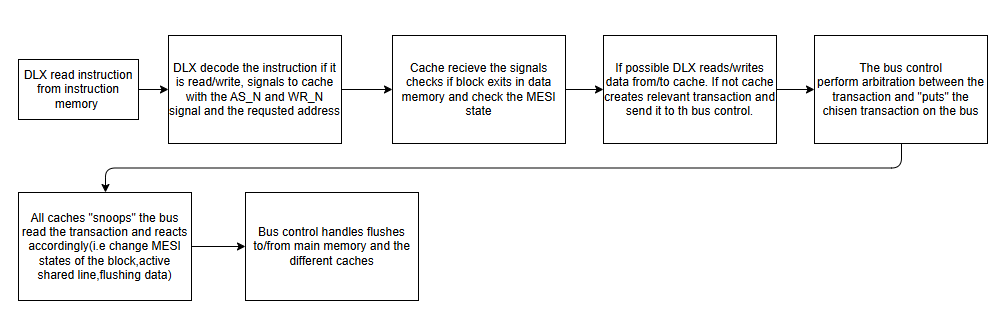


Figure 4: Multicore architecture's workflow diagram

The design structure is made of four main blocks – main memory, bus control unit and two identical processing cores, and are described in the following block diagrams:

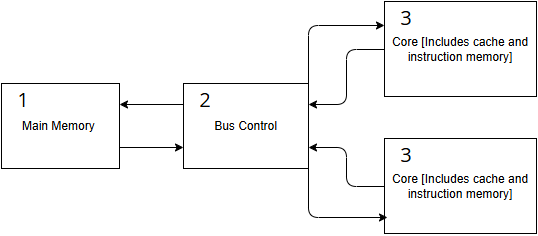


Figure 5 : Multicore architecture Top Level diagram

Each core consists of the following structure:   
A simple DLX core, a cache memory unit, and supporting logic and small units around the first two, to allow correct data flow and keep the compatibility with the original DLX structure and operation (each component will be discussed in detail in the following sections).

The internal structure of each core is as shown in the following diagram:

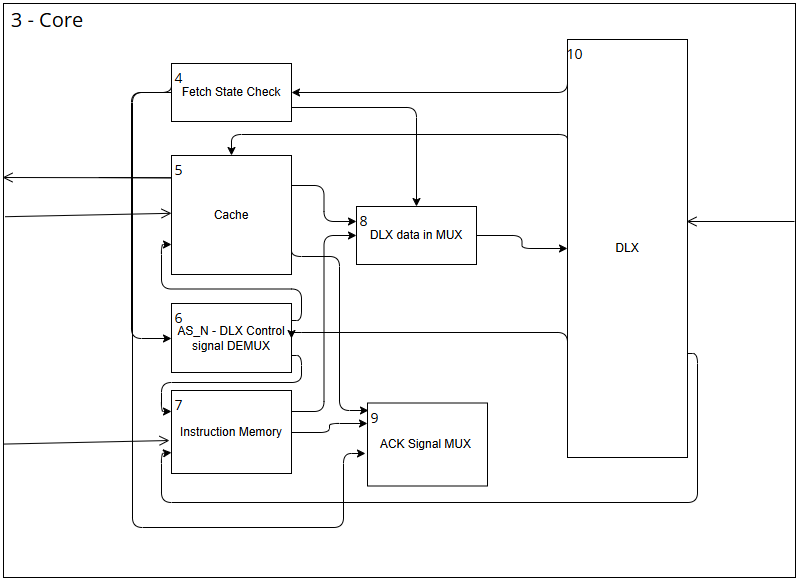


Figure 6 : Core architecture diagram

Each cache memory unit is built of several parts as well – a controller to control its different functions, Cache datapath that store, edit and route the data into and from the cache and address calculator to modify the raw address values the cache receives from the DLX into the cache address form.  
The structure of the cache and of the datapath unit is shown in the next diagrams:

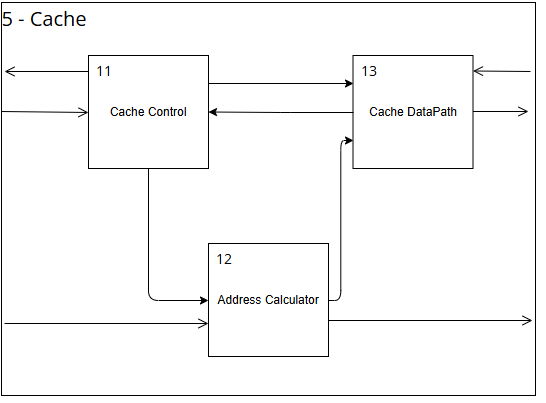


Figure 7: Cache structure diagram

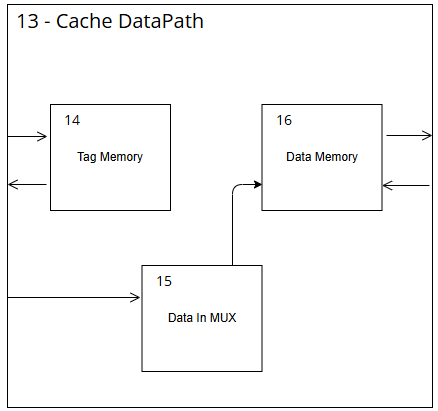


Figure 8: Cache DataPath architecture diagram

## 4.1 Hardware Description

The system was implemented in Verilog codes and schematic draws and loaded into a FPGA platform using the following hardware blocks:

* **DLX Based Core (x2):** Each core supports the DLX instruction set[[4]](#footnote-4).
* **Instruction Memory (x2):** One per core to eliminate fetch collisions.
* **Data Cache (x2):** Each core has a local data cache.
* **Main Memory:** Shared between cores, accessed via the bus.
* **Bus Controller:** Manages access scheduling and coherence signals.
* **MESI Controller:** FSM logic enforcing cache coherence for each core, as extension of the simple cache unit[[5]](#footnote-5).

In the following sub-sections, the different units, modules and components that we built and used during the project will be discussed and, when deemed necessary, an abstract figure will be attached, showing the modules' ports and connections[[6]](#footnote-6)

### Detailed explanation:

1. Main Memory: The main memory is implemented by the FPGA memory components in the physical implementation, and during simulations by the I/O Simul module, which was given in the ACSL course[[7]](#footnote-7).   
   Both consist of ordinary DRAM modules, and used as close boxes, the same they have been used during the ACSL course.
2. Bus Control: The bus control unit serves three main goals: data routing between the cores and the main memory, signal transmission and routing, and arbitration – managing the queue of the cores to memory access. To accomplish this the bus control receives all the data buses and signal lines from each of the above, holding in an internal memory the last core to address the maim memory and perform the necessary logic computations to allow fast and accurate operation of the full system.

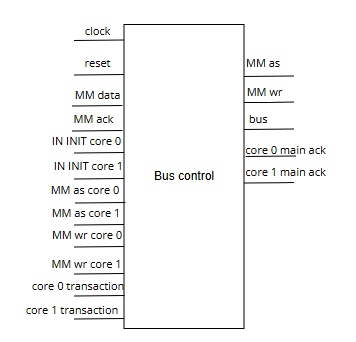


Figure 9: Bus Control Unit

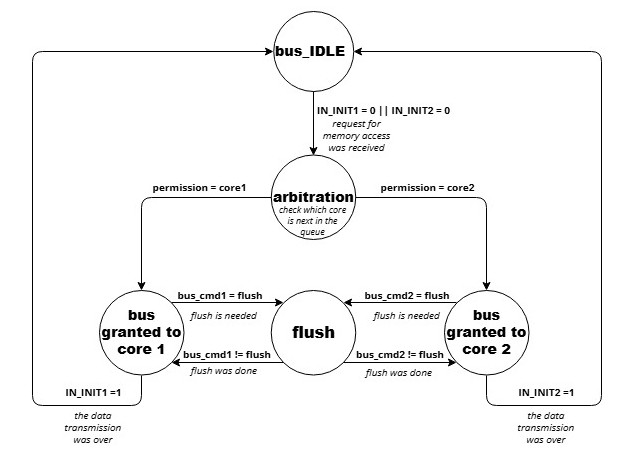


Figure 10: bus control state diagram

1. Core: This unit is a single core of the multicore processor architecture. The unit contain the DLX processor, the cache, the instruction memory, fetch state check unit, AS\_N – DLX control signal DEMUX unit, DLX data in MUX unit, and the ACK signal MUX unit all detailed further on.

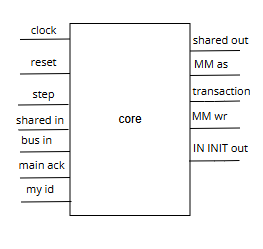


Figure 11: Core Unit

1. Fetch State Check: This unit checks the state of the DLX state machine and indicates if the DLX fetches an instruction.



Figure 12: Fetch State Check Unit

1. Cache: This unit is the cache memory of the core as explained in section 2. The unit contains the cache control, cache DATAPATH and address calculator unit detailed further on.

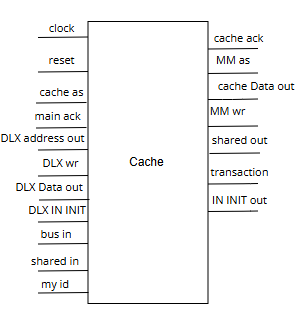


Figure 13: Cache Unit

1. AS\_N – DLX control signal DEMUX: A DEMUX to route the AS\_N signal of the DLX to the correct destination. If DLX fetches an instruction the signal goes to the instruction memory and if the DLX read/write data, the signal goes to the cache.

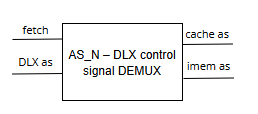


Figure 14: AS\_N – DLX control signal DEMUX Unit

1. Instruction Memory: Pre-loaded with the compiled assembly code. Act as an instruction cache memory for the DLX.

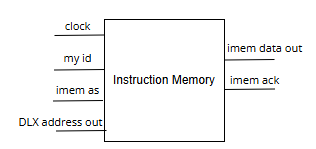


Figure 15: Instruction Memory Unit

1. DLX data in MUX: A MUX for the DLX 'data in' input if the DLX fetch an instruction the data arrives from the instruction memory and if the DLX read data the data arrives from the cache.

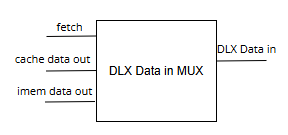


Figure 16: DLX data in MUX Unit

1. ACK signal MUX: A MUX for the DLX ack signal if the DLX fetch an instruction the ack arrives from the instruction memory and if the DLX read/write data the ack arrives from the cache.

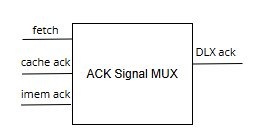


Figure 17: ACK signal MUX Unit

1. DLX: The DLX is a simple processor, implemented following a simplified version of the DLX architecture. It was implemented during the ACSL course[[8]](#footnote-8).
2. Cache Control: The cache control unit manages both data transmissions from and to the main memory, and access to the data saved inside the cache. This is done using a finite state machine, which is described in the diagram below (Fig. 19), and its decisions determined by the validity of the data inside the cache, the MESI state of the relevant block and additional information and requests which may will be received from the other cache.  
   It can perform either internal or external data access (depends if the block exists inside the cache and its MESI state, which determines its validity), data transmission to the main memory ("clearing block"), data transmission to the other cache ("flush", appears in purple in the diagram) or inform the other cache about a change in the status of certain block (if the block exists in both caches and it will be modified by 'write' instruction; appears in blue in the diagram). Since each block is 16-words size, each block transmission lasts 16 cycles, and the end of the cycle will be determined by internal counter (appears in italic form in the diagram).

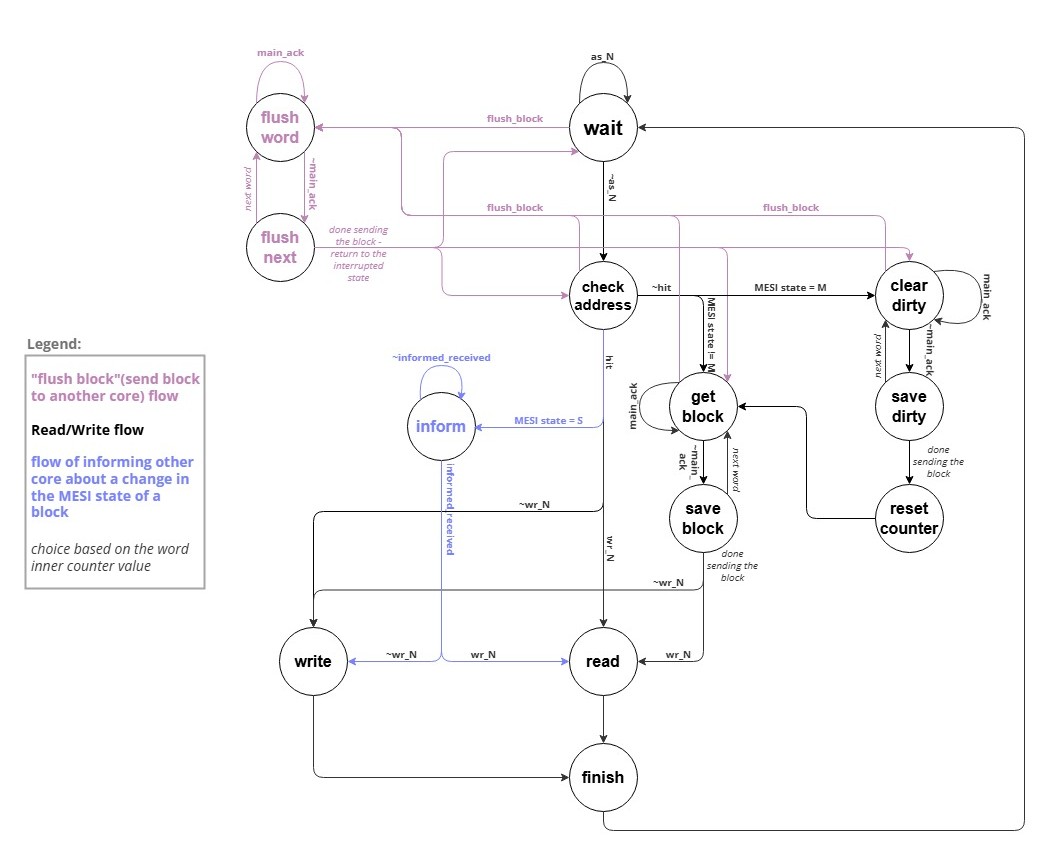


Figure 18: Cache control state diagram

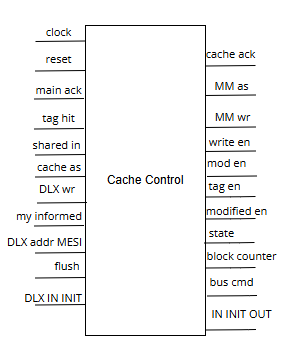


Figure 19: Cache Control Unit

1. Address Calculator: Address Calculator unit calculates the relevant address based on the required action. For example, when a full block is read or written the calculator outputs the 16 words addresses 1 by 1. Additional cases are flush, replacing modified block with new block and more. The calculator receives signals from the cache control and based on them perform the calculation.

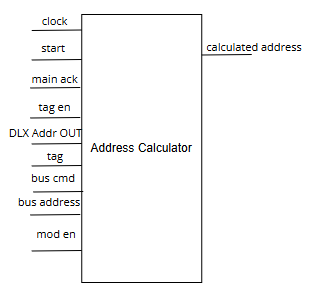


Figure 20: Address Calculator Unit

1. Cache DataPath: The Cache Datapath contains the units: Tag memory, Data in MUX and Data memory which are detailed further on.



Figure 21: Cache DataPath Unit

1. Tag Memory: The tag memory unit contains the tags of the blocks that are in the cache and their MESI state. The unit "snoops" the bus and based on the transaction it detects the unit changes the state of the relevant block. The unit turns on the "shared" line when a block that is in the cache memory is requested by the other cache and if "flush" is needed the unit signals to the cache control.

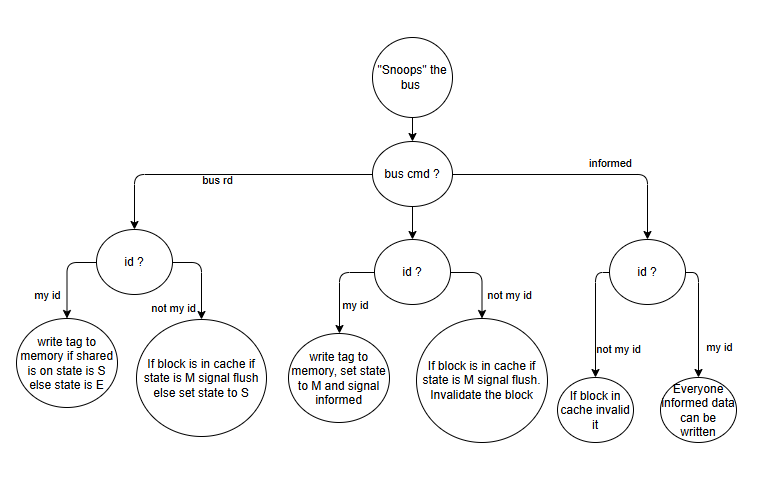


Figure 22: Tag Memory flow diagram

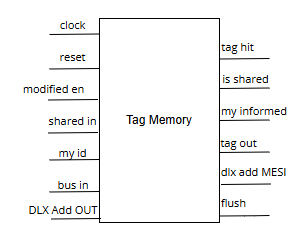


Figure 23: Tag Memory Unit

1. Data in MUX: Data in MUX is a 32bit MUX which based on the select signal chooses the data to be sent to the cache. When data from main memory or the other cache need to be written to the cache the select chooses 'bus data' input and in case the data is from the DLX the 'DLX data' input is chosen.

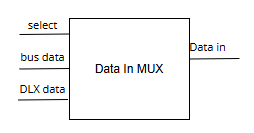


Figure 24: Data In MUX Unit

1. Data Memory: This unit is the memory of the cache. It contain up to 256 words divided into 16 blocks of data. Each clock cycle if 'write en' is on the word in 'Data in' input is written to the address based on the 'address' input and 'Data Out' output is the data that is in the requested address.



Figure 25: Cache Data Memory Unit

## 4.2 Software Description

Test programs were written in DLX assembly to exercise parallel access patterns, and demonstrate and validate coherence behavior under different parallelism conditions:

* **Bubble Sort –** Sorting a 32-words array from the main memory.   
  Each core sorting part of the array. This tests concurrent writing and enforces invalidations.
* **Vector Summing –** Addition of two 16-words vectors and save the result back in the main memory.  
  Independent sums are computed and shared, validating read-modify-write operations.
* **Basic image processing –** Importing and image, which made of 16X16 array and performing a basic image processing function – segmentation by a given threshold value, e.g. 128.  
  Each core processes a block of image data in parallel, simulating independent access patterns.
* Additional code was written to validate the correctness of the design, by going through every transition possibly for the MESI protocol, every new function of the processor and performing additional tests.

All programs were compiled using RESA compilation software and loaded using Xilinx's ISE into the instruction memory units.  
For each program we expected a significant speedup with minimal contention, particularly in memory-localized tasks

# 5. Analysis of Results

## Functional Verification

All MESI transitions were verified through simulation and hardware traces. Cache states responded correctly to bus events. No inconsistencies in memory were observed.

## Performance Metrics

After execution of all the programs on each of the processors (original simple DLX, DLX with cache extension and multicore DLX) the following results were recorded:

Table 2: Run time on each processor for different programs

|  |  |  |  |
| --- | --- | --- | --- |
| **Run time for program** | **Multicore[us]** | **Cache single-core[us]** | **Simple DLX[us]** |
| Vector Addition | 30.43782 | 46.43142 | 45.24023 |
| Image Processing | 172.7642 | 344.81202 | 427.14574 |
| Bubble Sort | 296.00655 | 695.87154 | 1120.2184 |

The results are also shown in the graph below for visual information and easy comparison between the different processors:

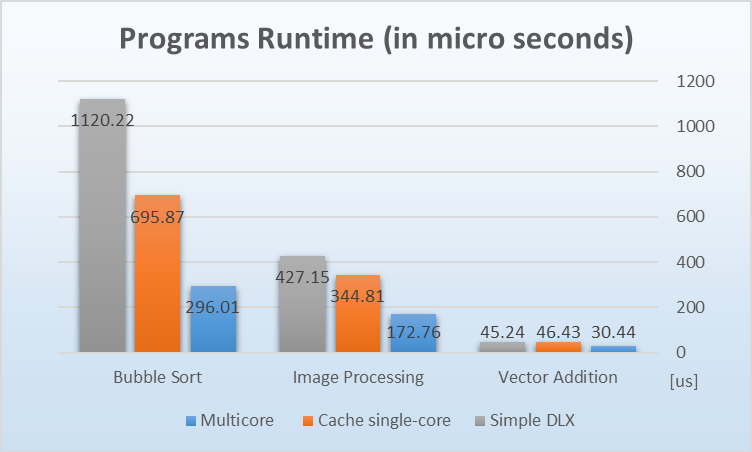


Figure 26: Programs runtime graph

By analysis of the results in the table above, the speedup in performance that was received is calculated below. As more complete analysis, the speedup was calculated not only between the original DLX and the multicore DLX, but between each of the processors above – for each program and the total average.

Table 3: Speedup ratio for any two processors

|  |  |  |  |
| --- | --- | --- | --- |
| **Speedup per program** | **DLX/multicore** | **DLX/cache single-core** | **cache single-core/multicore** |
| Vector Addition | 1.486316366 | 0.974345 | 1.525452 |
| Image Processing | 2.472420444 | 1.238779 | 1.995853 |
| Bubble Sort | 3.784437878 | 1.609806 | 2.350865 |
| Average | 2.581058229 | 1.27431 | 1.95739 |

A graphic representation and comparison can be seen in the graph below:

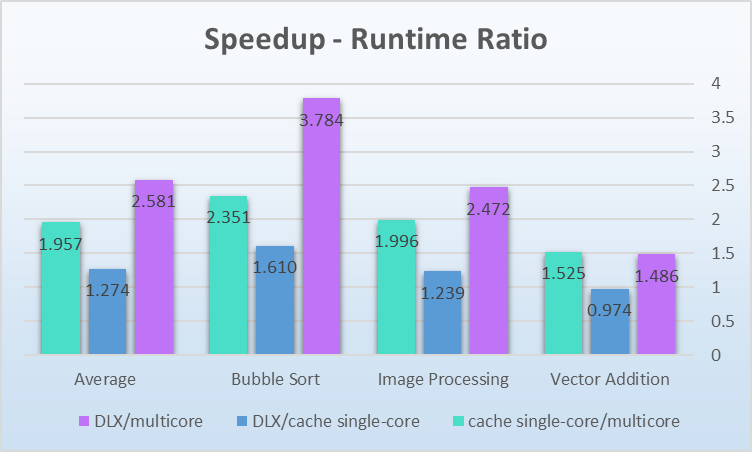


Figure 27: Speedup ratio graph

Several insights can be seen from the results above:

* Although some speedup was achieved by the addition of the cache to a single core processor, the majority of the improvement was due to the parallelism, which took advantage of the ability not only to do multiple computations in the same time but also to perform calculations by each core, when the other core might be IDLE and waiting for data transactions with the main memory to be completed
* The greatest improvement can be received in workloads with high amount of memory access actions, mostly in close addresses and local areas in the memory, with high number of computations and inter-core actions such as the bubble sort program.
* In programs with minimal number of computations, such as the vector addition, the speedup is less significant due to the cache structure, which requires the transmission of an entire block each time instead of a single word, and very little amount of calculation done internally in the cores.

Although several improvements contributed to superiority of the multicore processor in performance, it is clear that the main factor is parallelism, and its significance is well appreciated in the modern hardware industry and therefore its presence in every modern computer.

The execution of the different programs was not only achieved using the simulator, but on the physical implementation of the design, using the FPGA board. For example, the results of the bubble sort program can be seen below, as it was received using the RESA program.

The unsorted array - before the execution:

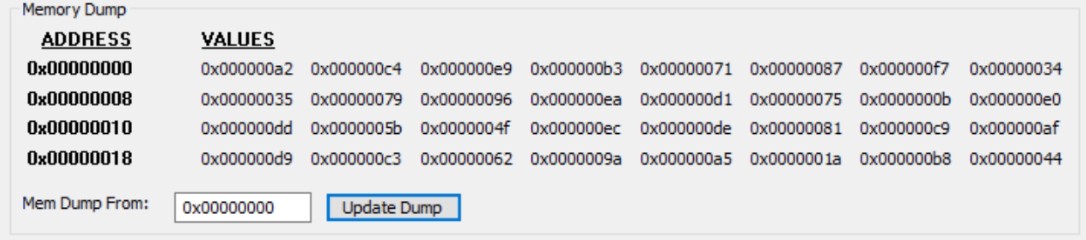


Figure 28: Execution example – the array before sorting

The sorted array – after the execution:

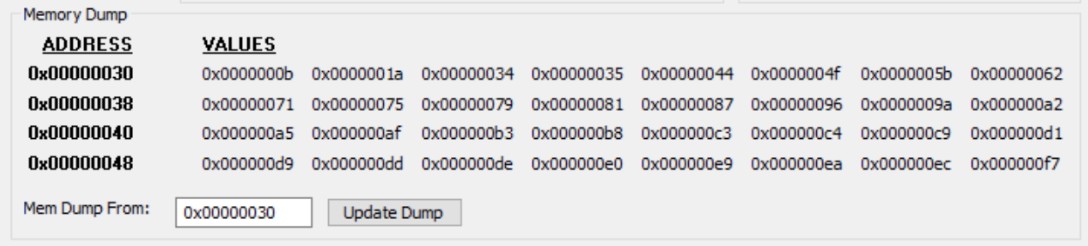


Figure 29: Execution example – the array after sorting

## Comparison to other Cache Coherence Protocols: MESI vs MOESI, MSI, and Dragon

Modern multiprocessors rely on cache coherence protocols to ensure consistency between private caches. The most widely known is MESI, offering a good balance of efficiency and complexity. More advanced protocols like MOESI optimize memory traffic by allowing dirty sharing, while simpler ones like MSI trade performance for ease of implementation. Dragon, a write-update protocol, is suited for systems with frequent sharing and minimizes invalidations. Below is a compact comparison across key features.

Table 4: Different coherence protocol comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **MESI** | **MOESI** | **MSI** | **Dragon** |
| States | 4: Modified, Exclusive, Shared, Invalid | 5: MESI + Owned | 3: Modified, Shared, Invalid | 4: Modified, Shared, Exclusive, Shared-Modified |
| Write Policy | Write-back | Write-back | Write-back | Write-back + Write-through |
| Shared Read | S state | S/O states | S state | S/Sm states |
| Exclusive Ownership | E state | E/O states | None | E state |
| Dirty Sharing | None | O state | None | Sm state |
| Bus Traffic Efficiency | Medium | High | Low | High |
| Write Miss Cost | High | Moderate | High | Moderate |
| Read Miss Cost | Medium | Low (can get from O state) | High | Low |
| Implementation Complexity | Moderate | High | Low | High |
| Use Case | General purpose | High-performance CPUs | Simpler designs | Systems with frequent sharing |

It can be seen that although more advanced protocols like MOSEI and Dragon offer better performance, the MESI protocol strikes an effective balance between simplicity and performance

# 6. Conclusions and Further Work

## Achievements

This project successfully extended a basic single-core DLX processor into a functional dual-core architecture with coherent memory access. The addition of local instruction and data caches to each core, combined with a carefully designed MESI-based coherence protocol, significantly improved overall system performance while maintaining data correctness. The design was thoroughly verified through simulation and deployed on FPGA hardware, demonstrating practical feasibility and robustness.

## Improvements Observed

The evaluation across multiple benchmark programs—such as bubble sort, vector addition, and image processing—revealed substantial performance gains, particularly in memory-intensive tasks. The integration of parallel execution and local caching proved highly effective, with speedups of up to 3.78× over the baseline DLX processor. While caching alone provided modest improvements, the real impact stemmed from parallelism and efficient memory management (such as separated instructions memory unit, to reduce bottle neck in memory access).

## Further Work

Several potential enhancements can further elevate the system’s capabilities. These include both improvements in single-core level such as pipelining the DLX core for improved instruction throughput, introducing dynamic scheduling for out-of-order execution, and integrating branch prediction mechanisms, and on the multicore level – exploring advanced coherence protocols such as MOESI or Dragon, as well as dynamic cache replacement strategies and OS-level interrupt handling.   
Implementing those improvements could enhance scalability and efficiency to provide significantly better performance of each core and the system as a whole.

## Takeaways

This project illustrates the transformative value of multiprocessing in computer architecture, even in small-scale systems. By combining hardware-based parallelism with thoughtful cache and memory management, the system achieves substantial performance gains with manageable complexity. The results reflect trends in modern processor design and underscore the relevance of coherence protocols in maintaining data integrity across cores.

# 7. Project Documentation

The project was mainly documented using GitHub repository, using different branches for each part of the project (by stages).

Project files include:

* Verilog source files for cores, caches, and control units.
* Testbenches and simulation scripts, written in Verilog.
* FPGA .bit files. Used to load the design into the FPGA board.

Documentation and source code repository: [Final-Project GitHub repository](https://github.com/yohaishiloh/Final-Project)

## User guide

Simulation:

* save two complied assembly code file name imem0 and imem1 for each core as .data file in the directory of all the project files named HOME\_VER. Maximum of 128 instructions per file
* In the same directory save a sram.data file contains the initial main memory data in eight hexadecimal numbers, when empty lines considered as zeros. The file has a maximum of 1024 lines, when each line is a single word
* open HOME\_VER.xise and run the testbench via xilinix platform and the waveform window will open.
* Review the waveforms as needed.

FPGA:

* save two complied assembly code file name imem0 and imem1 for each core as .data file in the directory of all the project files named SOURCE\_VER. Maximum of 128 instructions per file.
* Create a .cod file via RESA software with the main memory initial data.
* Open SOURCE\_VER.xise and create a bit file for the project.
* Using the RESA software upload the bit file and the .cod file to the FPGA
* Use continuous mode and let the programs run to halt.

# 8. References

1. Hennessy, J.L., & Patterson, D.A. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
2. Shen J. P., Lipasti M. K., "Modern Processor Design–Fundamentals of Superscalar Processors", McGraw-Hill, 1st ed. ,2005
3. Wikipedia contributors. (2025). MESI protocol. In Wikipedia, The Free Encyclopedia.  
   <https://en.wikipedia.org/wiki/MESI_protocol>
4. G. Even, M. Markov & M. Medina (2015). *Computer Structure Lab Notes – Implementing a DLX processor on an FPGA*, Tel Aviv University.
5. *Multiprocessors Cache Coherence* and *Cache Coherence Examples* course handouts, from Computer Architecture course notes. G. Oxman (2024), Tel Aviv University

# Appendix A. Inputs outputs tables

**Bus control:**

Table 5: Bus Control Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| reset | Input | 1 | Global reset |
| MM data | Input | 32 | Main memory data out |
| MM ack | Input | 1 | Main memory ack signal |
| IN INIT core 0 | Input | 1 | IN INIT signal from core 0 |
| IN INIT core 1 | Input | 1 | IN INIT signal from core 1 |
| MM as core 0 | Input | 1 | AS\_N signal received from core 0 dedicated for the main memory |
| MM as core 1 | Input | 1 | AS\_N signal received from core 1 dedicated for the main memory |
| MM wr core 0 | Input | 1 | WR\_N signal received from core 0 dedicated for the main memory |
| MM wr core 1 | Input | 1 | WR\_N signal received from core 1 dedicated for the main memory |
| Core 0 transaction | Input | 69 | Core 0 requested transaction |
| Core 1 transaction | Input | 69 | Core 1 requested transaction |
| MM as | Output | 1 | AS\_N signal to main memory |
| MM wr | Output | 1 | WR\_N signal to main memory |
| bus | Output | 69 | Bus data |
| Core 0 main ack | Output | 1 | ACK signal to core 0 |
| Core 1 main ack | Output | 1 | ACK signal to core 1 |

**DLX Core:**

Table 6: Core Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| reset | Input | 1 | Global reset |
| step | Input | 1 | Step enable for DLX |
| Shared in | Input | 1 | The other core shared signal |
| Bus in | Input | 69 | Bus data |
| Main ack | Input | 1 | Ack from bus control |
| My id | Input | 1 | Core id |
| Shared out | Output | 1 | Shared signal to the other core |
| MM as | Output | 1 | Connection between cache 'MM as' output to bus control 'MM as core' input |
| transaction | Output | 69 | Requested transaction |
| MM wr | Output | 1 | Connection between cache 'MM wr' output to bus control 'MM wr core' input |
| IN INIT out | Output | 1 | Connection between cache 'IN INIT out' output to bus control 'IN INIT core' input |

**Fetch state check:**

Table 7: Fetch State Check Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| DLX CTRL state | Input | 5 | DLX control state machine state. |
| fetch | Output | 1 | Fetch = 1 if DLX control state is fetch |

**Cache:**

Table 8: Cache Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| reset | Input | 1 | Global reset |
| Cache as | Input | 1 | Connection between AS\_N DEMUX 'cache as' output to the cache control 'cache as' input. |
| Main ack | Input | 1 | Connection between bus control 'Core main ack' output to cache control 'main ack' input. |
| DLX address out | Input | 32 | Connection between DLX 'address out' output to address calculator 'DLX Add OUT' input. |
| DLX wr | Input | 1 | Connection between DLX 'wr\_n' output to cache control 'DLX wr' input |
| DLX data out | Input | 32 | Connection between DLX 'data out' output to cache DATAPATH 'DLX data out' input |
| DLX IN INIT | Input | 1 | Connection between DLX 'IN INIT' output to cache control 'DLX IN INIT' input |
| Bus in | Input | 69 | Bus data |
| Shared in | Input | 1 | The other core shared signal |
| My id | Input | 1 | Core id |
| Cache ack | Output | 1 | Connection between cache control 'cache ack' output to ACK signal MUX 'cache ack' input. |
| MM as | Output | 1 | Connection between cache control 'MM as' output to bus control 'MM as core' input. |
| Cache data out | Output | 32 | Connection between cache DATAPATH 'data out' output to DLX data in MUX 'cache data out' input |
| MM wr | Output | 1 | Connection between cache control 'MM wr' output to bus control 'MM wr core' input. |
| Shared out | Output | 1 | Connection between cache DATAPATH 'is shared' output to the other core 'shared in' input |
| transaction | Output | 69 | Transaction requested. |
| IN INIT out | Output | 1 | Connection between cache control 'IN INIT out' output to bus control 'IN INIT core' input |

**AS\_N – DLX control signal DEMUX:**

Table 9: AS\_N – DLX control signal DEMUX Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| fetch | Input | 1 | Selects where the as signal needs to go. |
| DLX as | Input | 1 | DLX as signal. Signaling of an incoming address to be read/write from/to |
| Cache as | Output | 1 | Cache as = DLX as if fetch = 0 else equals 1 |
| Imem as | Output | 1 | Imem as = DLX as if fetch = 0 else equals 1 |

**Instruction Memory:**

Table 10: Instruction Memory Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| My id | Input | 1 | Core id |
| Imem as | Input | 1 | Signals to imem that an instruction is wished to be read |
| DLX address out | Input | 7 | The pc of the instruction that wished to be read |
| Imem data out | Output | 32 | The instruction that the DLX wishes to read |
| Imem ack | Output | 1 | Ack signal to signal that the instruction is ready to be read |

**DLX data in MUX:**

Table 11: DLX data in MUX Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input /Output | Size [bits] | Description |
| fetch | Input | 1 | Fetch = 0 => DLX data in = cache data out  Fetch = 1 => DLX data in = imem data out |
| Cache data out | Input | 32 | Cache data out |
| Imem data out | Input | 32 | Imem data out |
| DLX Data in | Output | 32 | The data the DLX wish to read base of if it is an instruction or data from the cache. |

**ACK signal MUX:**

Table 12: ACK signal MUX Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| fetch | Input | 1 | Fetch = 0 => DLX ack = cache ack  Fetch = 1 => DLX ack = imem ack |
| Cache ack | Input | 1 | Ack signal from cache control indicates to DLX that the data is ready to be read/written |
| Imem ack | Input | 1 | Ack signal from imem indicates to DLX that the instruction is ready to be read |
| DLX ack | Output | 1 | Ack signal to DLX based on if it reads instruction or reads/write data. |

**Cache control:**

Table 13: Cache Control Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| reset | Input | 1 | Global reset |
| Main ack | Input | 1 | Ack signal from the bus control |
| Tag hit | Input | 1 | Signal from tag memory if the block is in cache or not. |
| Shared in | Input | 1 | The other core shared signal |
| Cache as | Input | 1 | As signal from DLX indicates an address is set to read/write data. |
| DLX wr | Input | 1 | Signal if the DLX write or read |
| My informed | Input | 1 | Signal from tag memory that informed transaction of my core is detected on the bus. |
| DLX addr MESI | Input | 2 | The MESI state of the block of address requested by the DLX |
| flush | Input | 1 | Signal from tag memory that a flush of a block is needed |
| DLX IN INIT | Input | 1 | DLX IN INIT signal |
| Cache ack | Output | 1 | Cache control signal to DLX that the data is ready to be read or the data was written |
| MM as | Output | 1 | Cache control signals to bus control of an address arriving for read/write |
| MM wr | Output | 1 | Cache control signals to bus control if read or write |
| Write en | Output | 1 | Cache control signals to data memory to write the data. |
| Mod en | Output | 1 | Indicates address calculator if a block is read/written or a single word |
| Tag en | Output | 1 | Indicates address calculator where to take the tag from (current in cache or requested) |
| Modified en | Output | 1 | Cache control signals to tag memory to set the block MESI state to M |
| state | Output | 4 | The state of the cache control state machine |
| Block counter | Output | 5 | When block counter equals 0 the address calculator starts calculating the address when a full block is read/written |
| Bus cmd | Output | 3 | One of the indicators for the address calculator. |
| IN INIT out | Output | 1 | IN INIT output to bus control |

**Address calculator:**

Table 14: Address Calculator Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| start | Input | 1 | NOR of cache control 'block counter' output bits. Signals to start calculate the addresses when a whole block is read/written. |
| Main ack | Input | 1 | Ack from main memory signals that the data of the current address was read/written. |
| Tag en | Input | 1 | Indicates address calculator where to take the tag from (current in cache or requested) |
| DLX Add OUT | Input | 32 | DLX address out |
| tag | Input | 24 | Tag of the block of the address on the bus given from tag memory. |
| Bus cmd | Input | 3 | Bus command requested by the cache control determines the calculation of the address |
| Bus address | Input | 32 | The address that is on the bus |
| Mod en | Input | 1 | Indicates address calculator if a block is read/written or a single word |
| Calculated address | Output | 32 | Calculated address |

**Cache DATAPATH:**

Table 15: Cache DataPath Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| reset | Input | 1 | Global reset |
| Write en | Input | 1 | Connection between cache control 'write en' output to data memory 'write en' input |
| Cache control state(0) inverted | Input | 1 | Connection between Bit 0 of cache control state inverted to Data in MUX 'select' input |
| DLX Data out | Input | 32 | Connection between DLX 'data out' output to Data in MUX 'DLX data' input |
| Calculated address | Input | 32 | Connection between address calculator 'Calculated address' output to Tag memory 'DLX Add OUT' input and Data memory 'Address' input |
| Bus in | Input | 69 | Bus data |
| My id | Input | 1 | Core id |
| Modified en | Input | 1 | Connection between cache control 'modified en' output to Tag memory 'modified en' input |
| Shared in | Input | 1 | The other core shared signal |
| Data out | Output | 32 | Cache data out |
| DlX addr MESI | Output | 2 | Connection between Tag memory 'DLX addr MESI' output to cache control 'DLX addr MESI' input |
| My informed | Output | 1 | Connection between Tag memory 'my informed' output to cache control 'my informed' input |
| Is shared | Output | 1 | Connection between Tag memory 'is shared' output to the other core 'shared in' input |
| Tag hit | Output | 1 | Connection between Tag memory 'tag hit' output to cache control 'tag hit' input |
| tag | Output | 24 | Connection between Tag memory 'tag out' output to address calculator 'tag' input |
| flush | Output | 1 | Connection between Tag memory 'flush' output to cache control 'flush' input |

**Tag Memory:**

Table 16: Tag Memory Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| clock | Input | 1 | Clock signal |
| reset | Input | 1 | Global reset all blocks set to invalid |
| Modified en | Input | 1 | Cache control signals to set the block to state modified |
| Shared in | Input | 1 | The other core shared signal |
| My id | Input | 1 | Core id |
| Bus in | Input | 37 | Bus cmd, bus origid and bus address that are read from the bus |
| DLX Add OUT | Input | 32 | The Address the DLX wishes to write/read to/from after address calculator calculation |
| Tag hit | Output | 1 | Signals cache control the block is in the cache |
| Is shared | Output | 1 | On if the block of the address that read from the bus is in the cache |
| My informed | Output | 1 | Signals cache control that an informed transaction with my id is read on the bus |
| Tag out | Output | 24 | The current tag of the block of the address the read from the bus |
| DlX add MESI | Output | 2 | The MESI state of the block of address requested by the DLX |
| flush | Output | 1 | Signals cache control that a flush is required |

**Data in MUX:**

Table 17: Data in MUX Unit Input Output table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| select | Input | 1 | Select = 0 => Data in = bus data  Select = 1 => Data in = DLX data |
| Bus data | Input | 32 | Data that is written on the bus |
| DLX data | Input | 32 | Output data of the DLX |
| Data In | Output | 32 | Cache's data memory data in |

**Data Memory:**

Table 18: Cache Data Memory Unit

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Input/ Output | Size [bits] | Description |
| Clock | Input | 1 | Clock signal |
| Write\_EN | Input | 1 | Write 'Data In' to memory in 'Address' when enabled |
| Address | Input | 8 | Address in data memory array the DLX request to read/write from/to. after address calculator calculation |
| Data In | Input | 32 | Data to be written |
| Data Out | Output | 32 | Data in memory at 'Address' |

# תמונה שמכילה טקסט, צילום מסך, גופן, מספר תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.תמונה שמכילה טקסט, צילום מסך, גופן, מספר תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.תמונה שמכילה טקסט, צילום מסך, גופן, מספר תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.Appendix B. The Simplified DLX Instruction Set Architecture[[9]](#footnote-9)

תמונה שמכילה טקסט, צילום מסך, גופן, מספר

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

1. The Advanced Computer Structure Lab course. During the course, a simplified DLX processor was built by us, under the guide and supervision of the course stuff. [↑](#footnote-ref-1)
2. Since the design was made through the engineering point of view to keep the original DLX design as a close box (see chapter 2, section ***General Design Guidelines and Execution Method***), its functionality was ensured, and it needed no additional validation as part of the project [↑](#footnote-ref-2)
3. The simple cache unit was made as part of the first half of the project, which included a single core DLX with cache extension only. [↑](#footnote-ref-3)
4. Since the simple DLX processor itself is NOT part of the project rather a starting point and used as a "close box", its internal structure will not be addressed extensively in this paper.

   For more details regarding the DLX internal structure see G. Even, M. Markov & M. Medina (2015). *Computer Structure Lab Notes – Implementing a DLX processor on an FPGA*, Tel Aviv University. For more details about the DLX simplified architecture that it implements, see *Appendix B – Simplified DLX Instruction Set Architecture* [↑](#footnote-ref-4)
5. See footnote No. 3 [↑](#footnote-ref-5)
6. More detailed elaboration about the different modules' ports, connections and I/O signals is presented in *Appendix A – Inputs-Outputs Tables* [↑](#footnote-ref-6)
7. Since both was not part of the work in the project and only used as close boxes, we felt there is no need to provide deeply detailed explanation and/or internal structure diagrams regarding them. [↑](#footnote-ref-7)
8. See footnote No. 4 [↑](#footnote-ref-8)
9. From G. Even, M. Markov & M. Medina (2015). *Computer Structure Lab Notes – Implementing a DLX processor on an FPGA*, Tel Aviv University, pages 39-40. [↑](#footnote-ref-9)