Dual core DLX with atomic instructions

Project number: 3030

Final report

ID: 206564106 Ido Shalev

ID: 206867798 Matan Levin

Supervisor: Oren Ganon

Project Carried Out at: Tel-Aviv University, Projects Lab.

Contents

Contents

List of figures

List of tables

Abstract

The goal of this project is to extend the simple DLX processor architecture into a dual-core configuration, incorporating support for atomic operations such as:

- LR (Load-Reserved)
- SC (Store-Conditional)
- SWAP
- AMOADD (Atomic Add)

(Detailed explanations of each operation will be provided later)

Support for these operations enables the parallel execution of two processors (P0 and P1) accessing a shared memory without creating race conditions. A Sync-Box unit was added to the project, containing two entries per processor to store and manage reserved memory addresses, marking the requesting core and validating the reservation status. Additionally, the processor's control unit state machine was upgraded with five new states, enabling hardware implementation of the atomic instructions.

The project deliverables include:

- An architectural extension of the DLX into a dual-core processor with hardware synchronization.
- Simulations demonstrating proper synchronization functionality between P0 and P1.
- Quantitative metrics (area, power consumption, performance) measured using university tools.

The system preserves all original DLX functionalities while increasing hardware size by more than double, and power consumption accordingly. This extension demonstrates the feasibility of using a simple DLX processor for parallel processing scenarios with shared memory.

1 Introduction

In many embedded systems, parallel processing and load distribution across multiple cores have become critical components for enhancing performance and energy efficiency. This project focuses on extending a simple DLX processor to support two independent processors (P0 and P1) running code in parallel while preventing failures in accessing shared memory.

The primary challenge in this project is to implement atomic operations—operations on memory that are executed fully or not at all, ensuring that at any given moment, only one processor has access to the memory. This is aimed at preventing a situation where both cores read and write to the same variable without coordination, which could lead to incorrect results. Such issues are known as race conditions.

In this project, three main operations were added:

- **LR/SC** – Loading information from memory/saving information into memory.
- **Swap** – Exchanging a value between memory and a register.
- **Fetch-and-Op** – Atomic addition to memory (Atomic Add).

To achieve this, a SyncBox—a hardware synchronization management unit between the two cores—was designed, and the control unit was upgraded with new states to manage the mechanism. The project preserves the original functionality of the DLX processor while adding advanced capabilities for safe and efficient parallel operation.

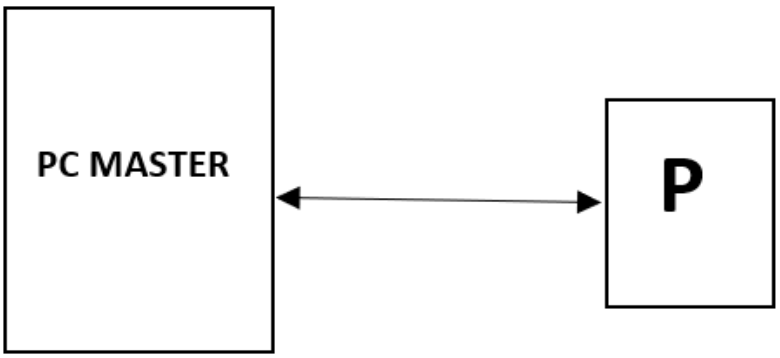


Figure 1 - Simplified DLX processor

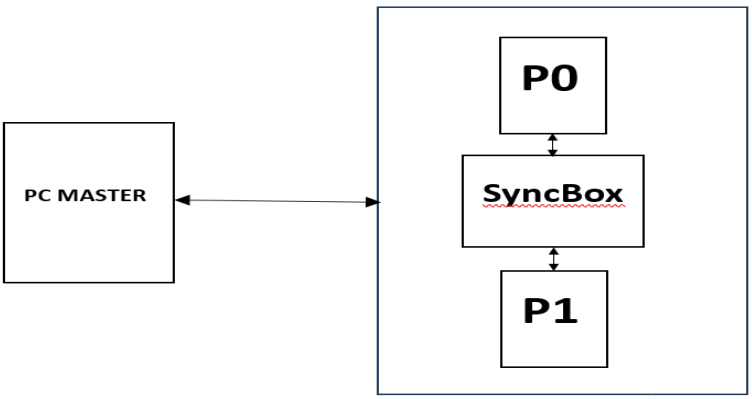


Figure 2 - dual core processor with atomic operations expansion

2 Theoretical background

During the advanced computer architecture lab, advanced technologies were employed for the design, simulation, and implementation of digital hardware, with a focus on building a simple RISC processor. The process involved describing hardware components using the Verilog language within the Xilinx ISE Foundation development environment, followed by functional simulation using ModelSim—a professional simulator for debugging and emulating processor operation prior to physical implementation. The processor was implemented on the XSA-50 development board from XESS, based on an FPGA component, and tested in conjunction with an educational system called RESA-2 and dedicated debugging tools developed in the lab.

The processor built is based on the DLX architecture—a simple RISC architecture used for educational purposes. As part of the project, the architecture was extended to support atomic operations and a multi-core system. These extensions included adding new machine code instructions (opcodes), adapting the datapath to support comparison logic, storage, and selection between regular and atomic execution, and updating the control unit's state machine (Control FSM) to include dedicated stages for these instructions.

The need for atomic operation support arises from the shift to multi-core processors, where multiple processes may access shared memory simultaneously. Uncoordinated access can lead to a race condition—a phenomenon where one process's write or read operation may occur concurrently or overwrite another process's operation, resulting in incorrect outcomes. To prevent this, atomic operations are required—operations that are executed fully or not at all, without the possibility of external interference. This feature is essential for process synchronization and maintaining data consistency.

In this project, a dedicated synchronization unit called SyncBox was implemented as a hardware component centralizing the reservation mechanism for memory addresses. Each of the two cores in the system is allocated a reservation entry, including a memory address, and a validity flag. Before performing an atomic write **(specifically SC)**, the core checks with the SyncBox to confirm that the reservation remains valid and belongs to it. Only if these conditions are met will the operation proceed; otherwise, it is aborted.

This approach is similar to existing atomicity mechanisms in modern architectures, such as LL/SC in MIPS, LDREX/STREX in ARM, and the LOCK prefix in the x86 architecture. However, the extension implemented on the DLX preserves implementation simplicity and adds minimal hardware logic, making it particularly suitable for educational environments and basic research in processor synchronization.

Command	Description	Functionality (register/memory equations)
LR Rd Rs1 imm	Load a word from external memory into Rd and mark the address reserved	Rd ← Mem[Rs1 + imm]
SC Rd Rs1 imm	Store the value in Rd into external memory if allowed; success/failure feedback updated in R29	If store succeeds: Mem[Rs1 + imm] ← Rd, R29 ← 1. Otherwise: R29 ← 0
SWAP Rd Rs1 imm	Exchange the value in Rd with the word in external memory; success/failure feedback updated in R29	If swap succeeds: Rd ← Mem[Rs1 + imm] (old), Mem[Rs1 + imm] ← Rd (old), R29 ← 1. Otherwise: R29 ← 0
AMOADD Rd Rs1 imm	Add the value in Rd to the old word in memory; new sum stored in both external memory and R30; success/failure feedback updated in R29	If addition succeeds: Mem[Rs1 + imm] ← Mem[Rs1 + imm] (old) + Rd, R30 ← Mem[Rs1 + imm] (old) + Rd, R29 ← 1. Otherwise: R29 ← 0

Table 1 – Atomic instructions

3 Simulation

Simulation Environment

The simulation was conducted using ModelSim, a professional simulator for debugging and hardware component simulation at the RTL (Register Transfer Level). We chose this tool due to its ability to provide a direct and detailed view of all internal signals in the system, including those of the SyncBox component and the expanded DLX's. Such simulation helps identify logical errors early in the development process, before physical implementation on an FPGA.

In addition to ModelSim, we utilized Xilinx ISE for creating Netlist files and synthesis testing, as well as the RESA-2 tool. The work environment included:

Verilog files describing all components of the expanded processor (such as Datapath, Control Unit, and SyncBox).

Dedicated testbenches written for each of the new atomic instructions to create diverse test scenarios.

Assembly files for running programs on the processor, as used in the lab environment, enabling us to simulate realistic use of two cores (P0 and P1) in parallel, combining regular and atomic instructions.

During the simulation, we examined sensitive signals such as Reservation Address and Valid from the SyncBox, as well as control signals from the state machine (FSM) of the control unit of the expanded DLX's. Additionally, we monitored the feedback register R29, which holds the success or failure value of the atomic instructions. For example, in the SC instruction, the value in R29 changes based on success or failure (Due to invalid reservation or race condition), while the LR instruction always succeeds and does not alter the external memory.

3.2 Simulation Objectives

The objectives of the simulation were to ensure that the new extensions function correctly, aligning with the overall project goals. Specifically:

Functional Validation: Verify that all new atomic instructions (LR, SC, SWAP, AMOADD) are executed according to their definitions, including the addition of new states to the state machine such as RESERVE, CHECK, ADO, WBA, and WBA_30.

Synchronization Testing: Assess that the SyncBox properly manages memory address reservations, matching between valid table and the addresses so that external memory write (using SC) are performed correctly.

Race Condition Prevention: Demonstrate that in scenarios of shared memory access, the outcome is controlled and logically correct (based on pre-decided core arbitration), free of errors.

Compatibility Preservation: Prove that the basic functionality of the DLX processor remains intact despite the extensions, with all regular instructions continuing to work as before.

Quantitative Metrics: Measure the number of clock cycles required to execute atomic instructions and compare them to regular instructions to evaluate the impact of the extensions on performance.

3.3 Test Scenarios

To validate the system, we defined several representative scenarios simulating realistic use of two cores (P0 and P1) accessing a shared dual-port memory. These scenarios cover all atomic instructions and demonstrate cases of success and failure:

Single Memory Access: One processor (e.g., P1) can perform LR on a specific address and thus loading its value to register and reserving its address, For SC, a valid reservation is required for success; for SWAP or AMOADD, the operation can proceed without a prior reservation. The success or failure value is stored in register R29.

Dual memory access:

Since both processors share a single external memory bus, the SyncBox selects which processor is active at a given time. That core's own AO (address out) and DO (data out) are selected through a multiplexer, while its AS_N and WR_N signals — which together tell the external memory what type of request the core is making — are also selected. The DI (data in) line from external memory is shared by both cores and simply ignored by the inactive one.

In this scenario, there are two possible situations:

1.Access to Two Different Memory Addresses: Each core accesses a different address using LR/SC, SWAP, or AMOADD. We verify dual access without conflicts, and return correct feedback in R29 (especially for SC, which depends on a valid reservation).

2.Parallel Access to the Same Memory Address: Both cores attempt to perform an operation on the same address simultaneously. The expected behavior is as follows:

For SC: If only one core has a valid reservation, it succeeds; if both have valid reservations, P0 takes precedence. The other core fails, reflected in R29 register in both core's RAM.

For SWAP or AMOADD: The operation can proceed without a reservation, but in case of parallel access, P0 takes precedence, and its operation is written to memory. The other core fails, with feedback in R29.

These scenarios cover all atomic instructions, with success or failure feedback (except for LR, which always succeeds) consistently provided in register R29.

It is important to note that any write access to an external memory address resets the reservation of both processors for that same address (Valid set to 0), regardless of whether the writing succeeded or failed.

3.4 Presentation of Results

Several scenarios can be used to illustrate the behavior of dual-core system:

- The following batch of LR/SC instructions serves as a comprehensive and illustrative example, as it demonstrates parallel load by both cores, and same address parallel write attempt to the same address in external memory, encompassing both successful and failing cases, as shown in the results below.

p0 code snippet:

```
90010006 //     lr R1 R0 6
2C210001 //     addi R1 R0 1
B0010006 //     sc R1 R0 6
```

p1 code snippet:

```
90010006 //     lr R1 R0 6
2C210002 //     addi R1 R0 2
B0010006 //     sc R1 R0 6
```

The external memory value at address 6 is 3 at first

	0	1	2	3	4	5	6
0x0	90010006	2C210001	B0010006	FC000000	00000000	00000000	00000003

After executing first 2 instructions (loading from address 6 and adding it different value in each processor) P0's R1 value and P1's R1 value are 4 and 5 respectively

	0	1
0x0	00000000	00000004
0x10	00000000	00000000

	0	1
0x0	00000000	00000005
0x10	00000000	00000000

We see that there 2 valid reservations of address 6 in syncbox, meaning address value in reservations table is 6 and valid value in validity table is 1 for both cores:

Reservations table:

	0	1
0x0	00000006	00000006

validity table:

	0	1
0x0	1	1

After executing the SC last instruction we see that p0 was given precedence and its R1's value was updated in the external memory at address 6 and also we see that p0 R29 value is 1 signaling success and P1's is zero signaling failure and that both reservations are no longer valid:

External memory:

	0	1	2	3	4	5	6
0x0	90010006	2C210001	B0010006	FC000000	00000000	00000000	00000004

P0's RAM:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0	00000000	00000004	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001	00000000	00000000

P1's RAM:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0	00000000	00000005	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Validity table:

	0	1
0x0	0	0

- An additional example is provided by the SWAP instruction. It demonstrates how a single core can atomically exchange the value of a register with the value stored at an external memory address.

  Code snippet:

```
90010005 //     lr R1 R0 5
2C210003 //     addi R1 R1 3
B4010007 //     swap R1 R0 6
FC000000 //     halt
```

  At first, the external memory value at address 5 is 2 and 8 at address 6:

	0	1	2	3	4	5	6
0x0	90010005	2C210003	B4010006	FC000000	00000000	00000002	00000008

  After executing the first 2 instructions (loading to R1 from address 5 and adding 3 to its value) P0's R1 value is 5 and we have a valid reservation for core P0 of address 5:

  P0 RAM:

	0	1
0x0	00000000	00000005
0x10	00000000	00000000

  reservations table:

	0	1
0x0	00000005	00000000

  validity table:

	0	1
0x0	1	0

  After executing the last SWAP instruction we see value at address 6 is swapped with value at R1 and the feedback value in R29 is 1 which means success and we also see that valid value is still 1 since we changed different address than the one reserved

  Updated external memory:

	0	1	2	3	4	5	6
0x0	90010005	2C210003	B4010006	FC000000	00000000	00000002	00000005

  validity table:

	0	1
0x0	1	0

  P0's RAM:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0	00000000	00000008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001	00000000	00000000

- Another illustrative scenario is the AMOADD instruction. It shows how a single core can perform an atomic update (addition of a register's value) on an external memory address:

  Code snippet:

```
90010006 //     lr R1 R0 6
2C210002 //     addi R1 R1 2
B8010006 //     amoadd R1 R0 6
FC000000 //     halt
```

At the beginning, external memory value at address 6 is 2 :

	0	1	2	3	4	5	6
0x0	90010006	2C210006	B8010004	FC000000	00000000	00000000	00000002

After executing the first 2 instructions (loading to R1 from address 6 and adding 2 to its value) P0's R1 value is 4 and we have a valid reservation for core P0 of address 6:

P0 RAM:

	0	1
0x0	00000000	00000004
0x10	00000000	00000000

reservations table:

	0	1
0x0	00000006	00000000

validity table:

	0	1
0x0	1	0

After executing the last AMOADD instruction we see that the external memory value at address 6 was added the value of R1 and the feedback value in R29 is 1 which means success and the calculated result was also loaded to R30 and the validity of the reservation is nullified:

Updated external memory:

	0	1	2	3	4	5	6
0x0	90010006	2C210002	B8010006	FC000000	00000000	00000000	00000006

validity table:

	0	1
0x0	0	0

P0's RAM:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0	00000000	00000004	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001	00000006	00000000

Waveform simulation:

An illustrative waveform example is that of the first LR/SC example comprised by the code:

p0 code snippet:

```
90010006 //     lr R1 R0 6
2C210001 //     addi R1 R0 1
B0010006 //     sc R1 R0 6
```

p1 code snippet:

```
90010006 //     lr R1 R0 6
2C210002 //     addi R1 R0 2
B0010006 //     sc R1 R0 6
```

- First rows (LR instructions)

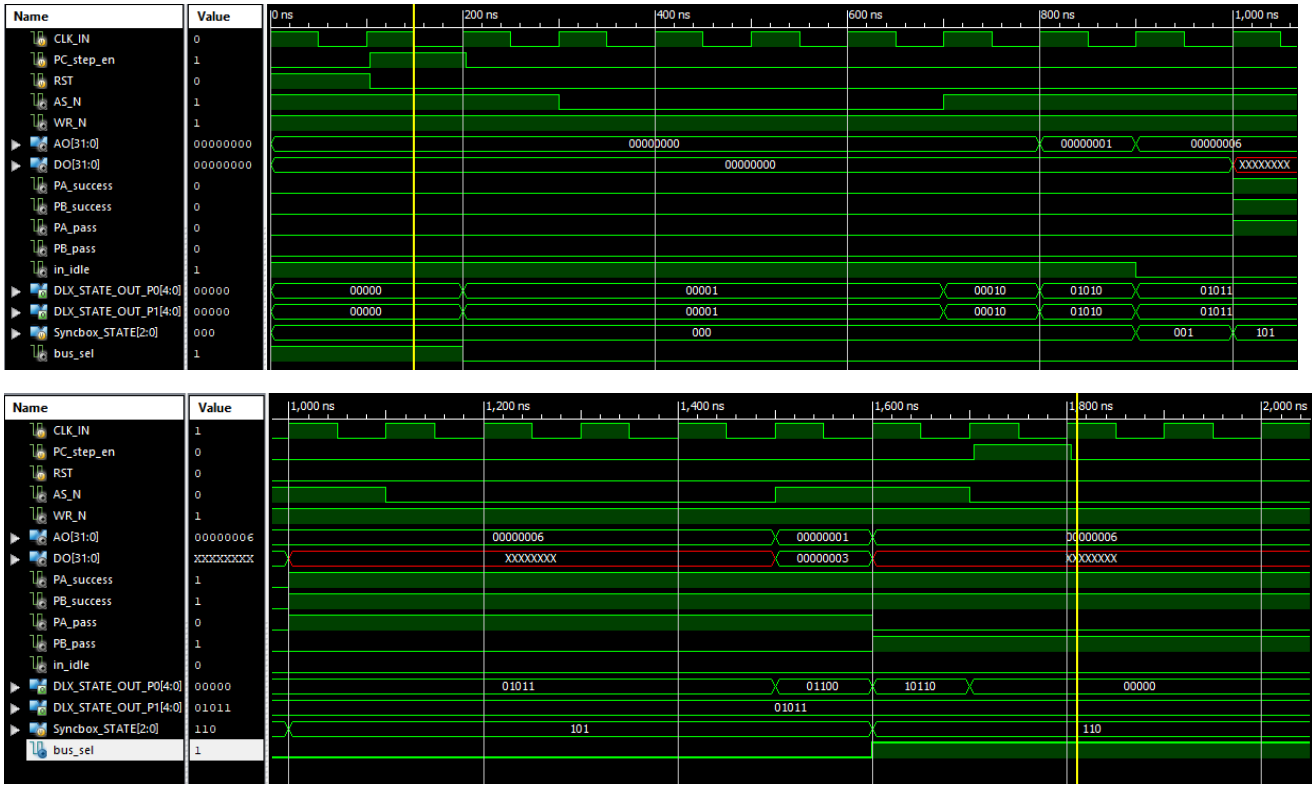


Figure 3 – LR waveform

At first both cores transition through the states INIT ('00000') → FETCH ('00001') → DECODE ('00010') → ADDRESSCMP ('01010') → LOAD ('01011'), and in LOAD ('01011') both need to perform loading from external memory. The SyncBox first grants access to P0 by asserting PA_pass high and bus_sel = 0 selects P0's AS_N, WR_N, AO, and DO to communicate with the external bus. P0 continues while P1 stays stuck in LOAD ('01011') state, and once P0 has finished using the bus the SyncBox asserts PB_pass and bus_sel = 1 selects P1's outputs to communicate with the external bus.

We also see that P0 waits in INIT ('00000') state until P1 finishes its instruction and also reaches INIT ('00000') state, while also waiting for the SyncBox to return to its first state IDLE ('000'). This is crucial for correct synchronization, since all cores must begin their instruction state transitions at the same time from their initial state.

- Second rows (addi instructions)

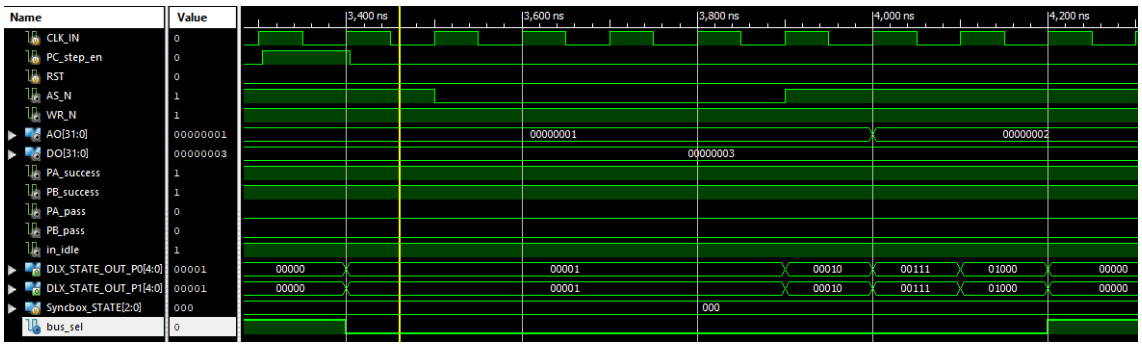


Figure 4 – addi waveform

Here we see that both cores go through the transitions INIT ('00000') → FETCH ('00001') → DECODE ('00010') → ALUI ('00111') → WBI ('01000') in a symmetric way, since no arbitration or synchronization of external bus access is required.

- Third rows (SC instructions)

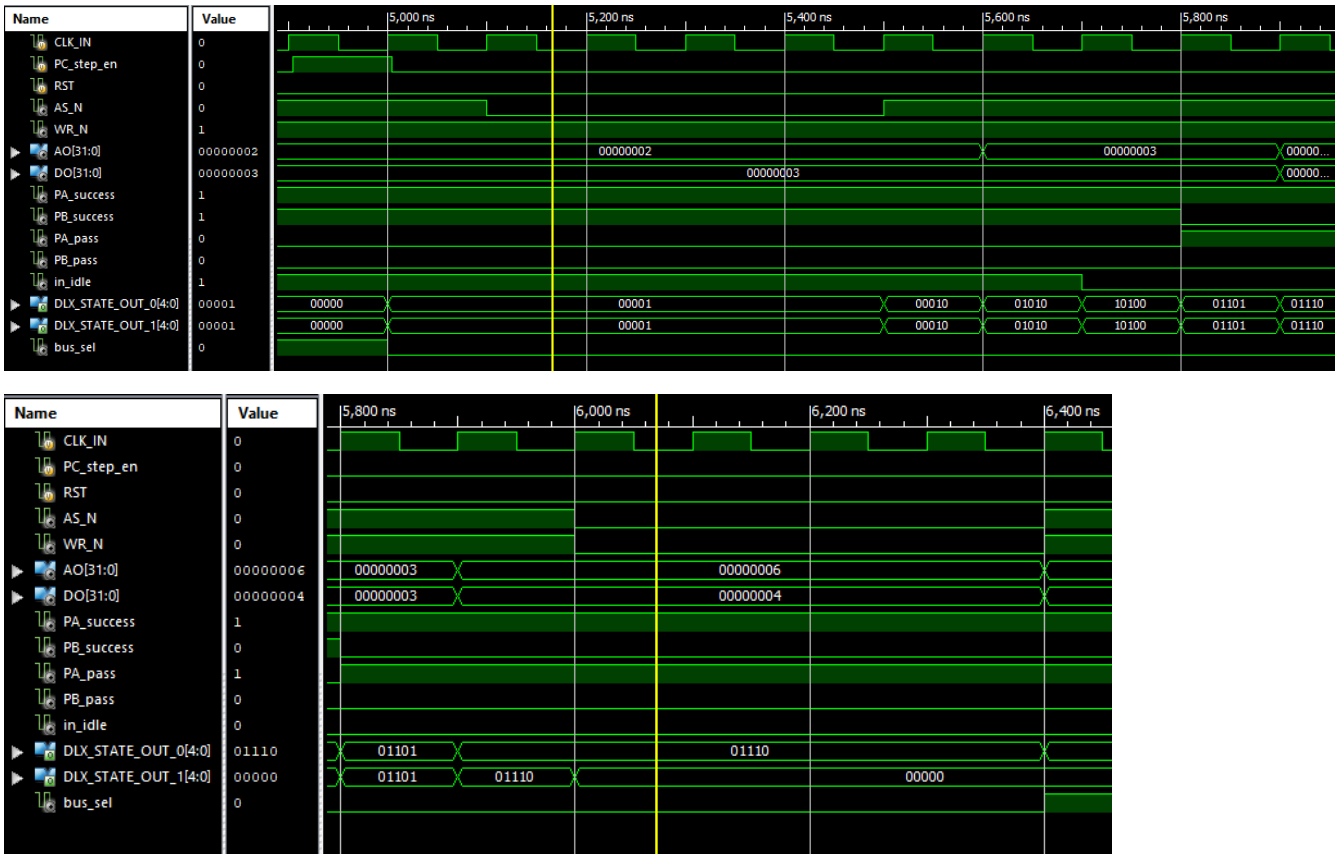


Figure 5 – SC waveform

At first both cores transition through the states INIT ('00000') → FETCH ('00001') → DECODE ('00010') → ADDRESSCMP ('01010') → CHECK_WBA_29 ('10100') → COPYGPR2MDR ('01101') → STORE ('01110') and in STORE ('01110') both need to write to the external memory but we see that PB_success signal turns low which signals failure so The SyncBox only grants access to P0 by asserting PA_pass high and bus_sel = 0 selects P0's AS_N, WR_N, AO, and DO to communicate with the external bus. P0 continues and P1 returns to INIT ('00000') state since its writing request failed.

We now see that P1 is the one waiting in INIT ('00000') state until P0 finishes its instruction and also reaches INIT ('00000') state, and also waiting for the

SyncBox to return to its first state IDLE ('000').

4 Implementation

As was explained above, our implementation is structured around two DLX cores that share a single external memory, coordinated by an arbitration unit. The goal is to preserve the simplicity of the original DLX design while adding only the minimal mechanisms, in terms of power and area, required to support atomic operations within each core. The datapaths of the DLX cores remain unchanged, while the control logic is extended and a SyncBox module is introduced to arbitrate the bus for communication with the external memory. The SyncBox manages reservations, drives the **bus_sel** signal to select which processor is connected to the bus, and grants approval for external memory accesses. This separation allows atomic instructions to be integrated without altering the basic functionality of the processor and provides a clear mechanism for handling arbitration, preventing conflicts, and guaranteeing correct execution when both cores access the external memory.

In our implementation, each DLX core's control state machine is extended by four additional states:

- **CHECK_WBA_29**: verifies success of memory access before executing atomic instruction.
- **ADO**: performs the arithmetic addition step of the AMOADD instruction.
- **WBA**: handles the write-back phase for SWAP and LR/SC.
- **WBA_30**: completes the write-back for AMOADD into register R30.

These extensions enable the correct execution of the newly added atomic instructions—LR, SC, SWAP, and AMOADD. They are particularly useful in parallel computations such as convolution and matrix multiplication, for which dedicated assembly programs were developed. The design was also synthesized and tested on a **Xilinx Spartan-6 FPGA**.

Below is a detailed state machine diagram of the extended control logic.

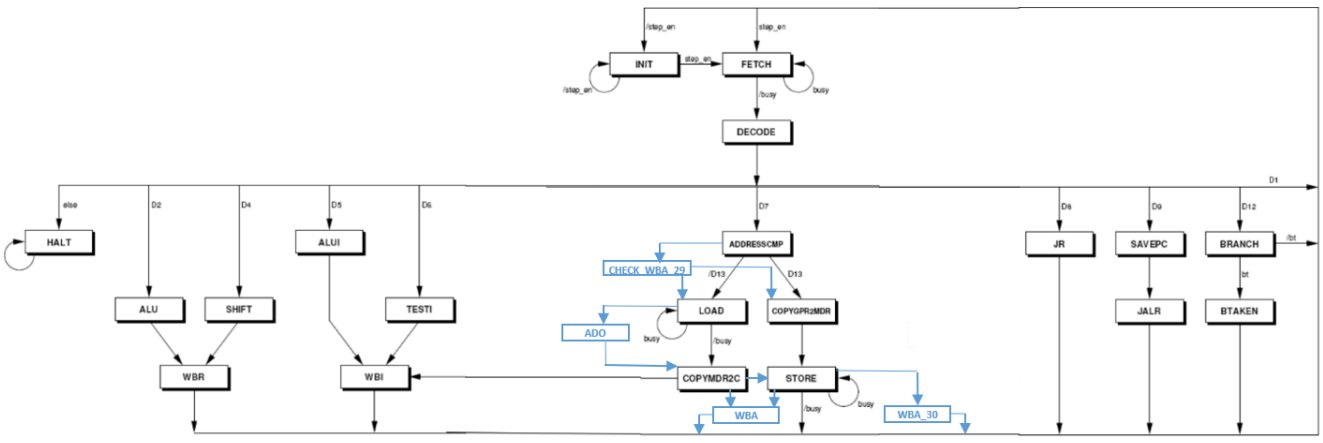


Figure 6- state machine diagram of the extended control logic (modifications in blue)

The atomic instructions follow the I-type encoding format:

6	5	5	16
Opcode	RS1	RD	immediate

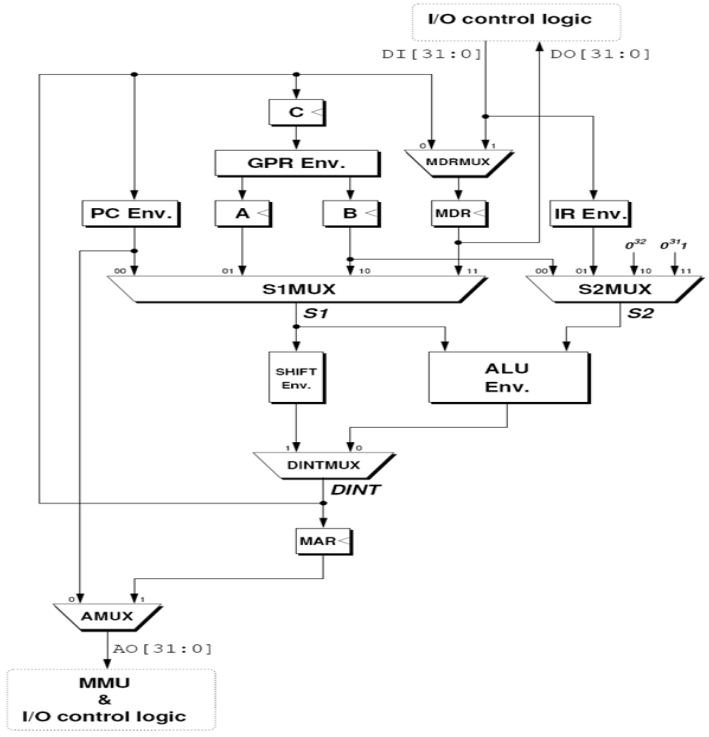


Figure 7 - Datapath of the simplified DLX machine

In addition, the arbitration logic of the SyncBox is implemented as a finite control state machine, whose behavior is summarized in Figure 10. Figure 10 illustrates the finite control state machine of the SyncBox, which coordinates how the two DLX cores access the shared memory. The machine starts from the IDLE state and moves to VALIDATE whenever one or both cores issue a request. In VALIDATE, the SyncBox decides which core (or both, in sequence) is allowed to access memory. After this point, three possible paths exist:

• **P0_ONLY_WAIT**: taken when only Core 0 proceeds to a memory operation; in this state, the FSM waits until Core 0 completes its read or write before returning to IDLE.

• **P1_ONLY_WAIT**: taken when only Core 1 proceeds to a memory operation; in this state, the FSM waits until Core 1 completes its read or write before returning to IDLE.

• **BOTH_WAIT_P0 → BOTH_WAIT_P1**: taken when both cores request simultaneously; the FSM first serves Core 0 and waits for its memory operation to complete, then switches to Core 1 and waits for its completion, before finally returning to IDLE.

Thus, every WAIT state represents the system holding until a core finishes its memory access, ensuring orderly and correct use of the shared bus.

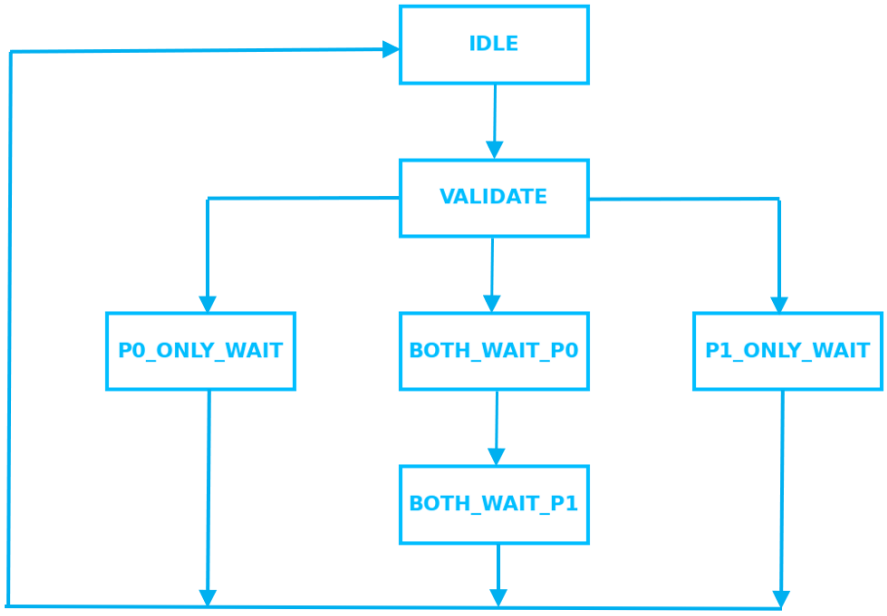


Figure 8 - SyncBox control state diagram.

Beyond control flow, the SyncBox also maintains internal data structures to track reservations. Figure 11 illustrates the data structures of the SyncBox, consisting of a MAR table for storing reservation addresses of each core and a corresponding Validity table indicating whether each reservation is active.

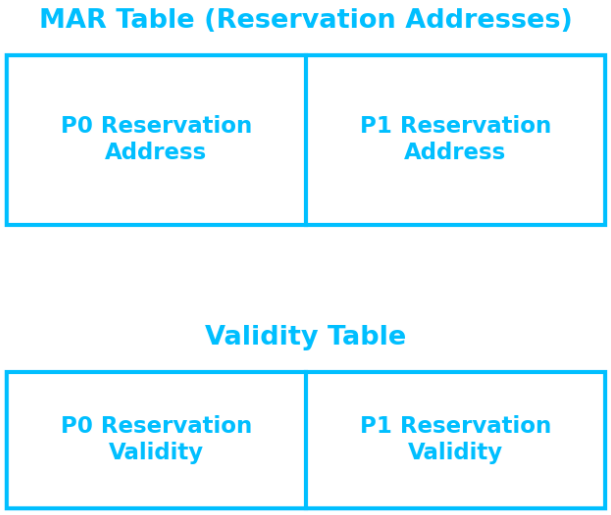


Figure 9 - SyncBox data structures (MAR and Reservation Validity tables).

4.1 Hardware code implementation

The hardware implementation was carried out on the Xilinx Spartan-6 FPGA (xc6slx25), a mid-range device commonly used in teaching and research. It offers about 24,000 logic cells, DSP slices, and block RAM, providing sufficient resources for implementing moderately complex processor designs. The design consists of two DLX cores implemented in Verilog, each built around a control state machine that sequences instruction execution. Every core includes its own ALU, register file, and control logic, while memory operations are coordinated through a shared external memory.

Since only one memory and one external bus are available, a dedicated SyncBox arbitration unit was introduced. The SyncBox manages reservations for atomic instructions, resolves conflicts between cores, and drives the bus_sel signal to determine which core is granted access to the bus for reading from or writing to the external memory at any given time. This ensures correct and ordered execution of all memory operations.

To support newly added atomic instructions (LR, SC, SWAP, AMOADD), the control state machine of each core was extended with four additional states (CHECK_WBA_29, ADO, WBA, WBA_30). These modifications allow atomic operations to be executed reliably within each core while maintaining correct arbitration for the single external memory.

The entire design was coded in Verilog and synthesized using the Xilinx ISE Design Suite. Synthesis produced utilization and timing reports confirming compliance with the available FPGA resources. Functional verification was performed in ModelSim, while final emulation was executed on the Spartan-6 FPGA using the RESA environment. Both environments validated the correct operation of the dual-core system, including synchronization through atomic instructions.

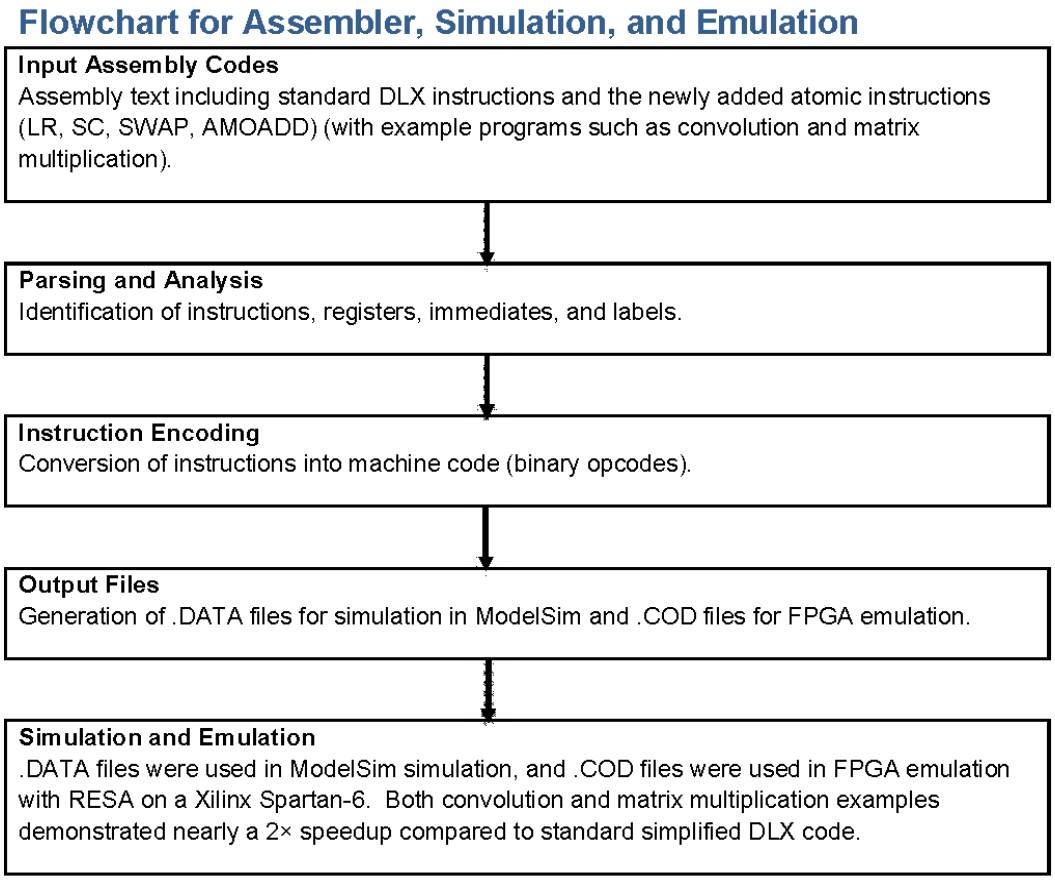
Figure 10 - FPGA xilinx spartan-6

4.2 Software code implementation

The software implementation provided the complete toolchain required to support the extended dual-core DLX processor, covering code generation and compilation, simulation, and FPGA emulation. The processor itself was implemented in Verilog, while a Python-based assembler was developed to convert assembly programs into machine code (.CODE and .DATA files), enabling their execution on the architecture.

To extend the programming model, the instruction set was enhanced with four atomic operations: LR, SC, SWAP, and AMOADD. Dedicated assembly programs for matrix multiplication and convolution were created as representative workloads to evaluate these extensions in parallel computations (achieving nearly a 2× speedup in these codes).

The design flow included functional verification in ModelSim and hardware emulation on a Xilinx Spartan-6 FPGA using the RESA environment, with results compared between the software model and the FPGA implementation. Software simulation proved especially valuable, as it allowed many subtle issues to be identified and resolved without the need to run directly on hardware.



Hardware and software description

Following design and simulation verification, we downloaded our implementation onto the FPGA using the XuLa2-LX25 board, which integrates a 1.5M-gate Spartan-6 device. A bit file was generated with the Xilinx toolchain and programmed into the FPGA via the RESA software environment.

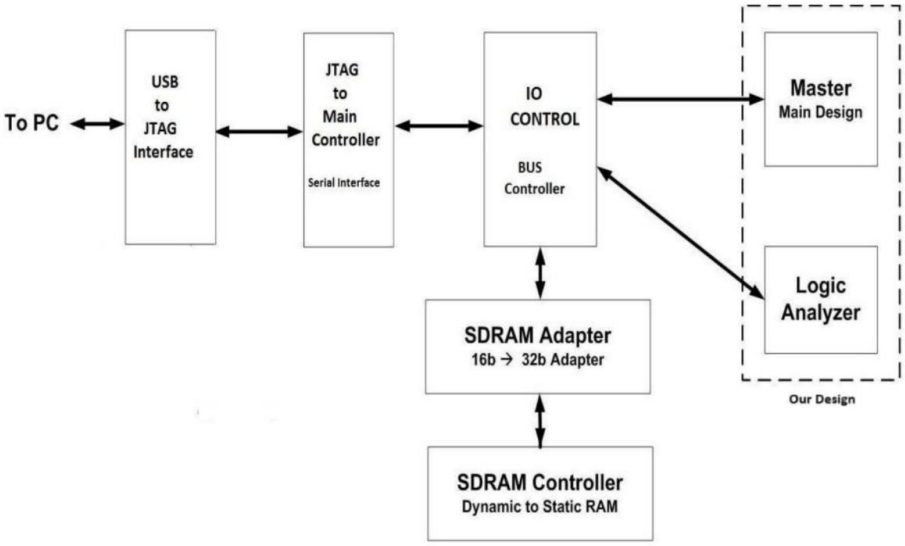


Figure 11 - block diagram of the overall system

Hardware Description and Synthesis:
The design was coded in Verilog, a widely accepted hardware description language for designing and implementing digital logic. We utilized the Xilinx ISE Design Suite for synthesis and deployment, enabling us to map the Verilog code onto FPGA hardware. The Xilinx suite delivered synthesis reports and resource utilization overviews to confirm adherence to our hardware specifications.

Simulation and Functional Verification:
To ensure the accuracy of our implementation, we employed ModelSim, a popular HDL simulation platform. ModelSim facilitated the simulation of our Verilog modules at the signal level, tracked internal states over time, and confirmed the performance of each custom atomic instruction and dual-core synchronization process. This step was crucial for troubleshooting and validating the design prior to synthesis.

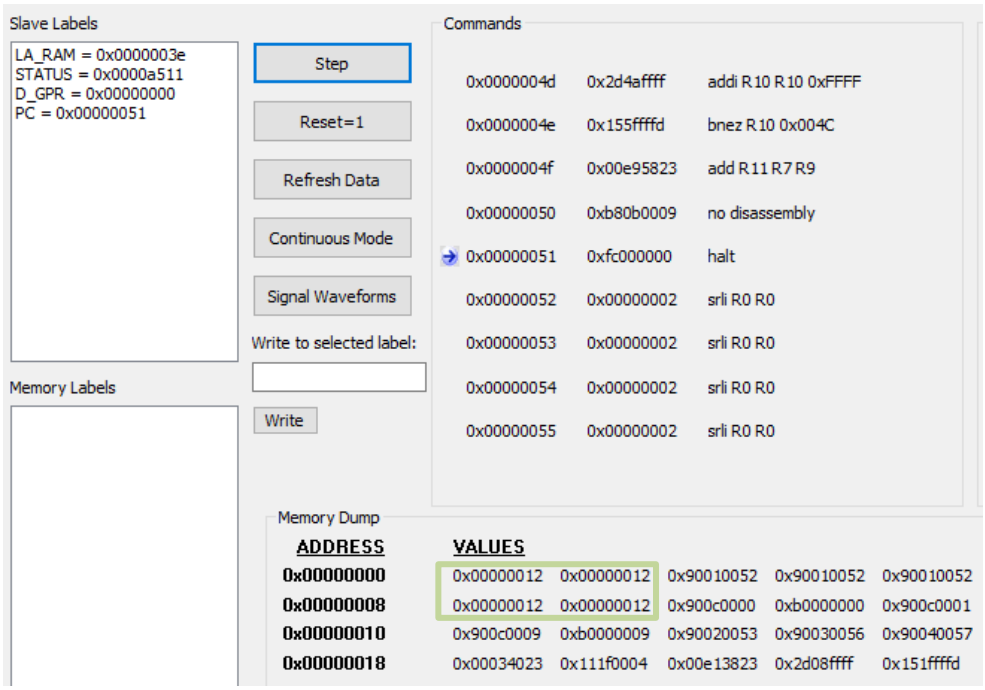
Translating Assembly:
The creation of .data and .code files for simulation and emulation respectively was handled using an enhanced assembler from the lab, adapted to include both the standard DLX instructions and our newly added atomic commands.

Emulation with RESA:
For hardware-level testing, the system was emulated on a Xilinx Spartan-6 FPGA using the RESA environment. This platform provided a bridge between the generated machine code and the physical FPGA, allowing full execution of our assembly programs and verification of system behavior under realistic conditions.

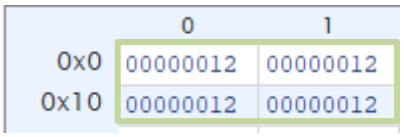
A matrix multiplication code example we compared is:

$$C = A \times B = \begin{bmatrix} 4 & 2 \\ 4 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & 1 \\ 4 & 6 \end{bmatrix} = \begin{bmatrix} 12 & 12 \\ 12 & 12 \end{bmatrix}$$

In RESA:



We verified by comparing to Model_Sim simulations result and indeed both results match:



5 Analysis of results

Power Consumption

The goal was to keep total power growth modest relative to the single-core DLX baseline.
Measured results show 0.086 W → 0.108 W (+28%). The increase is driven mainly by Logic (+150%) and Signals (+75%) due to the second datapath and larger per-core control, while DCMs and IOs remain unchanged and Leakage rises only slightly (+7%).
Conclusion: The dual-core design meets the intent of a restrained power budget; shared clocking/IO and the FPGA's static floor keep total power well below a 2× scaling.

Component	Single DLX (W)	Dual-Core DLX (W)	Δ (%)	Notes
Clocks	0.009	0.02	+100%	More global clock load, 2 cores + sync
Logic	0.004	0.010	+150%	Datapath + larger FSMs per core
Signals	0.004	0.007	+75%	Routing not doubled, many shared nets
DCMs	0.014	0.014	0%	Same clock manager shared
IOs	0.027	0.027	0%	External interface unchanged
Leakage	0.030	0.032	+7%	Baseline dominates, small increase only
Total	**0.086**	**0.11**	**+28%**	Dual ≠ 2×; shared resources limit growth

Table 2 – Power consumption

Area Utilization

Resource usage grows from 2,564 → 3,775 units (+47%) across key metrics (Registers, LUTs, Slices, MUXCYs). This reflects two cores plus expanded control and small arbitration logic, with efficient packing and shared infrastructure (DCMs, IOs, routing) preventing a full 2× increase. Specifically, the FPGA tools are able to co-locate LUTs and registers from different cores into the same slice, improving density. In addition, clock managers, IO buffers, and much of the interconnect fabric are reused rather than duplicated, reducing overhead that would otherwise push the design closer to double the baseline.
Conclusion: Area overhead is contained and aligns with expectations for a dual-core upgrade with richer control, remaining comfortably within device capacity (xc6slx25).

Component	Single DLX	Dual-Core DLX	Δ (%)
Occupied Slices	443	641	+45%
Slice Registers	688	1,001	+45%
Slice LUTs	1,213	1,797	+48%
MUXCYs	220	336	+53%
LUT–FF pairs	1,370	1,962	+43%
Total Area	**2,564**	**3,775**	**+47%**

Table 3 – Area utilization

Speedup

We investigated useful operations such as convolution and matrix multiplication, in which most of the work (in the assembly code) is done through internal R-type calculations (repeated additions used to implement multiplications). Memory access still has to happen in turns between both cores so parallelism is reduced, but in these workloads it is not the main factor because the majority of the operations take place inside each core. As the input values increase, the number of additions required for each multiplication also grows, and the workload is more dominated by these internal calculations, pushing the speedup close to the full 2× theoretical maximum.

We checked the case with h = {3, 6, 8, 10} and x = {1, 2, 4, 5, 7}, the time to calculate all convolution outputs required 7,838 cycles in our old single core simplified DLX, while the dual core finished in 4,242 cycles (1.85× faster). In the 2×2 matrix multiplication with both matrices filled with 4s, in order to calculate the result the old DLX needed 6,783 cycles versus 3,572 needed by the dual core (1.90× faster). This shows that even with small values, the dual-core already came very close to ideal scaling.

6 Conclusions and Further Work

Conclusions

The main objective of this project was to extend the simplified DLX processor into a dual-core system capable of running in parallel on shared memory with correct synchronization. This was achieved through the addition of atomic instructions (LR, SC, SWAP, AMOADD) and the design of the SyncBox arbitration unit, which ensures orderly access and prevents race conditions.

The design was verified in simulation and emulation, showing correct operation of all instructions and proper synchronization between the two cores. Quantitative analysis demonstrated that the additional hardware remains well within FPGA capacity: area overhead was limited to +47% and power to +26%, far below a naive 2x scaling. The dual-core processor proved especially effective for workloads where most of the computation is carried out internally within each core in a balanced way, rather than through frequent reads and writes to external memory. Key examples such as convolution and matrix multiplication achieved speedups close to 2x (1.85–1.90x), demonstrating the efficiency of the design. These results show that a simple and lightweight dual-core DLX, with minimal synchronization hardware, can already deliver substantial performance gains without incurring excessive area or power costs.

Further Work

Building on this project, several improvements could be explored:

1. **Shared data reads**
   When both cores need the same data, there is no need for them to each access memory separately. A single read can be shared by both, which would save cycles and avoid unnecessary turn-taking arbitration. This is a low-cost optimization since it only requires the ability to broadcast a value to both cores.

2. **Sharing computed results**
   If one core already computed a value that the other core also needs, that result could be shared directly instead of recalculating it on both cores. This would reduce redundant work and improve efficiency, especially in repetitive workloads such as convolution. The mechanism could be as simple as allowing a result register to be accessed by both cores under synchronization.

3. **Smarter stalling**
   In the current design, a core may be forced to wait even if it still has internal additions to complete. Smarter stall logic could let a core continue its internal work while the other is using memory, ensuring fewer idle cycles. This would better overlap internal arithmetic with memory arbitration delays.

4. **Multiply instruction**
   Adding a real multiply instruction would replace the current repeated-addition loops, drastically reducing cycle counts for multiplication-heavy workloads. However, this would also lower the measured dual-core speedup, since the single core would gain as much as the dual-core from faster multiplies. Even so, for workloads with many multiplications compared to memory accesses, dual-core execution would remain effective.

5. **Dual-core aware assembler**
   At present, dividing the workload between cores must be done manually. A dual-core aware assembler could automatically balance the work—for example, assigning even and odd outputs to different cores or distributing multiplications more evenly. This would reduce cases where one core finishes earlier than the other, improving overall throughput.

6. **Shallow pipelining**
   Our design is currently single cycle per instruction, without overlapping stages. Introducing a shallow pipeline (fetch, decode, execute, write-back) would allow multiple instructions to be in progress at once, reducing total execution time. However, pipelining also introduces data hazards, where an instruction depends on the result of a previous one that has not yet completed. These hazards could be handled with simple forwarding paths or pipeline stalls, keeping the design lightweight while still improving throughput.

7 Project Documentation

This repository contains the documentation for the Dual-Core DLX with Atomic Instructions Project, which includes both hardware and software components to extend the DLX processor for dual-core synchronization and atomic operations (LR, SC, SWAP, AMOADD).

The documentation covers:

Hardware: Verilog codes of all blocks designed in this project, including the SyncBox, extended Control Unit, and Datapath additions.

Software: Contains the lab's assembler, extended to support the new atomic instructions alongside basic DLX operations. Also includes workloads like matrix multiplication and convolution assembly scripts.

Simulations: ModelSim waveforms, RESA-2 emulations, and testbenches for testing dual-core synchronization. User Guide: Provides a step-by-step guide for setting up the environment, running the assembler, testing the design using RESA-2 on FPGA, and emulating workloads.

The full documentation for the Project can be found on GitHub at the following link: **https://github.com/Idoshalev7/Dual_Core_Project/tree/main**

8 References

1. "Atomic operations" , ARM developer- Home/Documentation/Architectures/Learn the architecture Introducing AMBA CHI, https://developer.arm.com/documentation/102407/0102/Atomic-operations

2. Leo Hou(Virtualization Engineer), " RISC-V-Standard Extension for Atomic Instructions", https://medium.com/@leohou1402/risc-v-standard-extension-for-atomic-instructions-0d8efcb55f53