INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2026, VOL. 72, NO. 3, PP. 1-11
Manuscript received March 23, 2026; revised June, 2026. doi: 10.24425/ijet.2026.1733

Analysis of coprocessor-based acceleration for hardware-based semaphore concurrency control

Michal Walichiewicz, and Adam Milik

***Abstract*—Coprocessors are widely integrated into modern microprocessors, particularly in safety-critical systems. They handle interrupts and accelerate computationally intensive operations, such as cryptographic and floating-point calculations; furthermore, they enable separation between system components. This article adopts a structured approach to provide a formal description of coprocessors and their role within modern microprocessor architectures. In addition, a concurrency control system based on Inter-Processor Interrupts (IPIs) is presented and compared with a polling-based synchronization approach using hardware-based semaphores. Finally, this study evaluates whether coprocessors can be utilized to accelerate multi-core synchronization by executing computations in parallel with synchronization.**

***Keywords*—safety-critical systems; concurrency control; hardware-based semaphores; multi-core systems; coprocessors; inter-processor interrupts**

I. Introduction

COPROCESSORS are often used in safety-critical systems such as programmable logic controllers (PLCs), due to their ability to enable functional separation of hardware modules. This separation enhances both system reliability and fault tolerance, which is essential in environments where failures can have serious consequences.

Safety-critical systems are typically implemented as heterogeneous distributed embedded systems consisting of a network of distributed microprocessors, each performing a dedicated function. These systems are typically interconnected via a Controller Area Network (CAN). In PLC-based systems, communication is performed using the Common Industrial Protocol (CIP), as shown in Fig. 1.

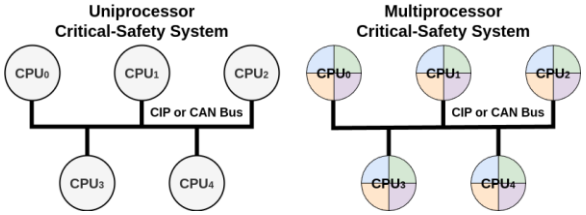

Fig. 1. Diagram of Two Heterogeneous Distributed Embedded Systems

This article investigates the effectiveness of coprocessors utilizing hardware-based semaphores for handling Inter-Processor Interrupts (IPIs) and accelerating computations by enabling parallel processing of shared data during data retrieval, and compare this approach with our polling-based method.

In addition, the proposed hardware implementation demonstrates a predictable multiprocessor architecture that could be implemented on multi-core microprocessors used in safety-critical systems, as shown in Fig. 1.

In safety-critical systems, separation and non-interference between components are essential design principles for ensuring system integrity, and fault isolation. Fig. 2 presents a safety-critical software architecture that adheres to the principles of the Multiple Independent Levels of Security/Safety (MILS) architecture [1], [3].

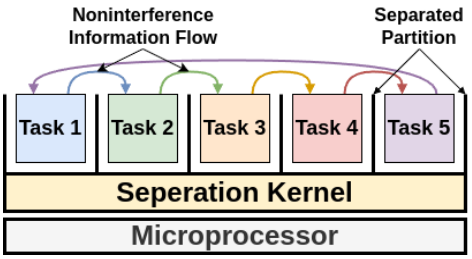

Fig. 2. Diagram of the Safety-Critical Software Architecture

MILS partitions the software into separate domains in accordance with the Greve, Wilding, and Vanfleet separation model. This ensures that data within each partition cannot be corrupted by adjacent partitions. Furthermore, the architecture enforces controlled information flow between these partitions in accordance with Rushby's non-interference model [1]–[2].

In safety-critical systems, various microprocessor architectures are used, with the most common being ARM and MIPS architectures.

MIPS-based microprocessors delegate system control, floating-point, and other specialized operations to coprocessors. Coprocessors enforce separation and non-interference by independently managing specific functionalities, thus reducing complexity and the potential for inter-component interference. Fig. 3 illustrates a classical MIPS-based architecture that includes a system control coprocessor integrated with the main processor [4]–[5].

Michal Walichiewicz and Adam Milik are with Silesian University of Technology, Poland (e-mail: michal.walichiewicz@protonmail.com, adam.milik@polsl.pl).

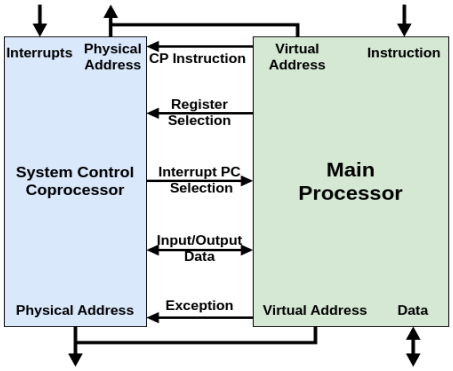

Fig. 3. Diagram of MIPS 2000

The system control coprocessor is responsible for system control functions, including interrupt handling, exception processing, and memory management, within the processor architecture. This architectural choice mirrors the principles of the MILS architecture, which contains potential faults within a single hardware module, thereby reducing the risk of fault propagation and enhancing system reliability. This concept of hardware-level fault isolation and functional separation is illustrated in Fig. 4.

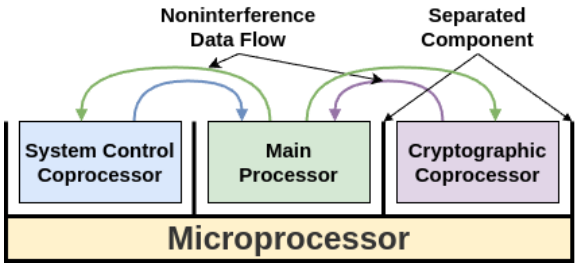

Fig. 4. Diagram of Hardware Separations

Interrupts are a fundamental mechanism in embedded systems, enabling efficient and low-latency communication between peripheral devices and the processor. When a peripheral, such as a general-purpose input/output (GPIO) pin, detects an event, such as a button press or a signal change, it can generate an interrupt signal to notify the processor immediately. This enables the processor to respond in real-time, eliminating the need for continuous polling and thereby conserving computational resources. In addition to GPIOs, other system components can generate interrupts, provided that they are properly configured.

Interrupts can also be used for multi-core synchronization. IPIs are a key mechanism in inter-processor communication (IPC), which encompasses various approaches enabling processing cores to exchange data and synchronize operations. Examples of such mechanisms include the Open Multi-Processor Interrupt Controller (OMPIC) used in OpenRISC-based architectures and mechanisms implemented in Xtensa-based MPSoCs [6]–[8].

A. Hardware

A brief introduction to hardware design essential for understanding the challenges faced by safety-critical systems is provided. Fig. 5 illustrates a typical single-issue, in-order, 3-stage pipeline architecture where all functional units are tightly integrated into the pipeline. This design does not incorporate coprocessors, which means that system control functionalities and memory management are handled internally by the core. Processor architectures such as ARM and OpenRISC commonly follow this design approach. However, such integration may reduce system safety due to the lack of functional isolation between hardware modules [5]–[6].

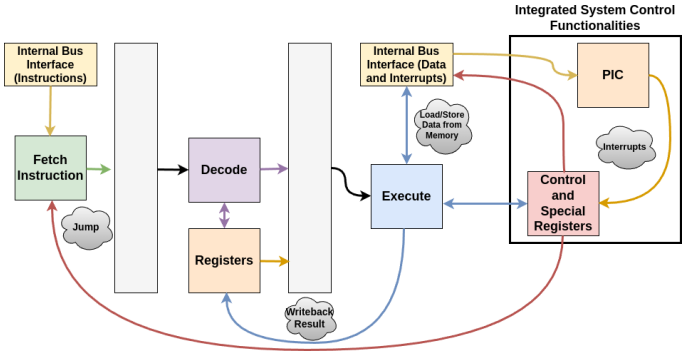

Fig. 5. Diagram of Integrated Architecture

A second common pipeline design used in safety-critical systems is based on the MIPS architecture that incorporates a system control coprocessor alongside the main processor, as depicted in Fig. 6 [4].

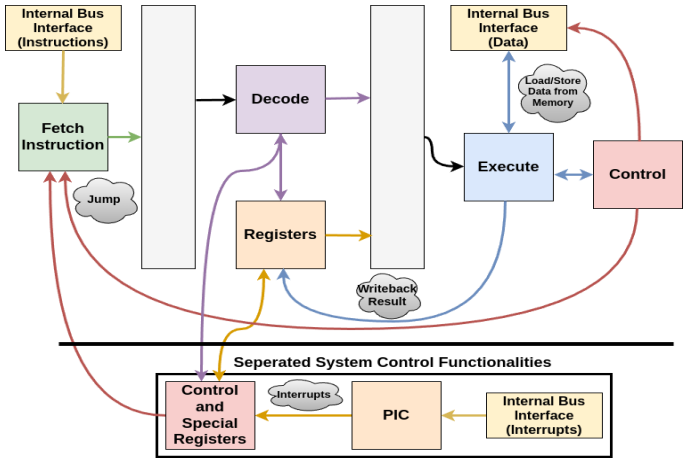

Fig. 6. Diagram of MIPS-based Architecture

In this architecture, the Programmable Interrupt Controller (PIC) is decoupled from the main execution pipeline and implemented within the system control coprocessor, allowing it to operate independently of the main pipeline. This coprocessor manages system control functions and memory management. The core design enforces non-interference information flow between the main core and the coprocessor; this separation enhances the reliability of the microprocessor.

The coprocessor plays a central role in handling tasks such as interrupt management, exception processing, and system monitoring. Fig. 7 presents a typical system control coprocessor that contains registers responsible for managing system control functions. These registers provide essential information on processing mode status, system configuration, interrupt status, exception status, timer values, and other control parameters [4].

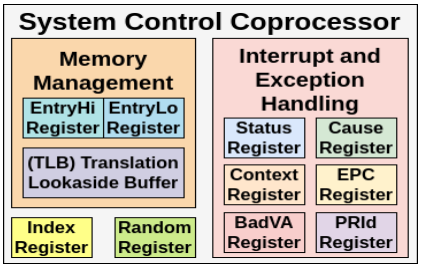

Fig. 7. Diagram of MIPS-based Architecture

The Cause register contains information on the exceptions and interrupts that have occurred, including the type and source of each event. The Exception Program Counter (EPC) register stores the address of the instruction being executed when an exception or interrupt occurred. This enables the processor to resume execution from the correct location after handling the exception or interrupt. Each bit in the Status register represents the state of a specific system control function, providing critical information on the current operating state of the processor [4].

Access to any coprocessor's registers, whether for retrieving information or configuring them, is performed using dedicated instructions. The corresponding MIPS assembly instructions are shown below [9].

Listing 1: Move To/From Coprocessor 0 MIPS Assembly

```
; Coprocessor Number is the Instruction's Digit
; First Reg is Used for the Data Movement
; Second Reg is the Coprocessor Number
; Last 3 Bits is Selection Type

Move_To:
    mtc0 $t0, $a0, 0
Move_From:
    mfc0 $t0, $a0, 0
```

The system control coprocessor is assigned number zero and uses two dedicated instructions to read and modify its register bits. However, other coprocessors, such as floating-point coprocessors, can also execute operation-specific instructions. Fig. 8 illustrates a floating-point coprocessor that employs dedicated instructions to perform floating-point operations. Offloading these computations improves the reliability and fault tolerance of the microprocessor, particularly for floating-point operations.

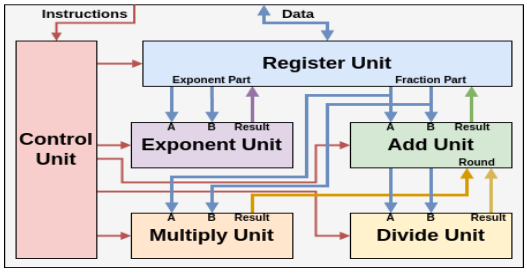

Fig. 8. Diagram of MIPS-based Architecture

These coprocessors contain a control unit that decodes dedicated instructions to execute complex arithmetic operations. In addition to instruction decoding, the coprocessor manages data transfer between the main processor general-purpose registers and dedicated registers, such as floating-point registers in such coprocessors. The following MIPS assembly instructions illustrate floating-point arithmetic instructions used by the coprocessor.

Listing 2: Floating-Point Coprocessor 1 MIPS Instruction

```
; Single Precision Instructions end with .s
; Double Precision Instructions end with .d
; Only Even Numbered Registers are used
; in Double Precision Instructions

FloatingPoint_Subtraction:
    sub.s $f0, $f1, $f2

FloatingPoint_Multiplication:
    mul.d $f2, $f4, $f6
```

B. Inter-Processor Interrupts

Interrupts and exceptions are signals that set the corresponding bits in the Cause register of the system control coprocessor, as depicted in Fig. 9. When these signals are triggered, they cause the processor to jump to a designated program location responsible for handling them. For this mechanism to operate properly, the coprocessor must be configured to handle interrupts and exceptions [4], [9]–[10].

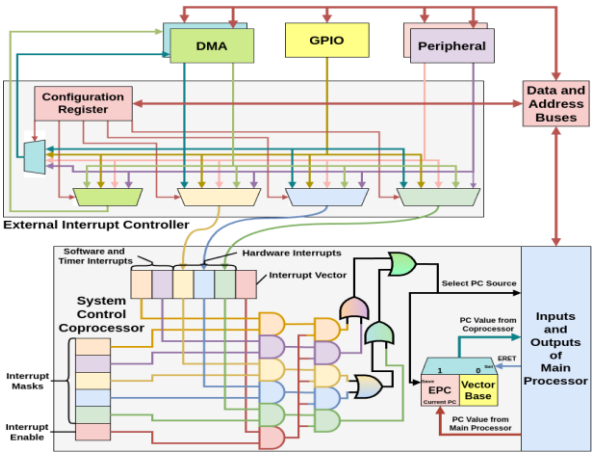

Fig. 9. Diagram Of the Interrupt Handling by Coprocessor

The current program counter (PC) is stored in the EPC register when an interrupt or exception occurs. At the same time, the program counter is updated to jump to the interrupt or exception vector address. These addresses typically contain jump instructions to the interrupt or exception handler, respectively. The following segment presents MIPS assembly code for the interrupt and exception vector [9]–[10].

Listing 3: Exception and Interrupt Vectors MIPS Assembly

```
1: Exception_Vector:
2:     j Exception_Handler
3:     nop
4: Interrupt_Vector:
5:     j Interrupt_Handler
6:     nop
```

The corresponding handler first stores general-purpose registers in memory to preserve the current program state. Next, the handler retrieves information from the Cause and Status registers of the system control coprocessor. The Cause register contains bits corresponding to specific interrupts and exceptions, while the Status register contains masked bits corresponding to these interrupts and exceptions. Each bit is checked sequentially, from the least significant to the most significant, to identify the interrupt or exception source. This enables the processor to jump to the corresponding interrupt service routine (ISR) [9]–[10].

Listing 4: Interrupt Handler MIPS Assembly Code

```
1: Interrupt_Handler:
2:     ; Jump and Link to the Store Part
3:     jal Store_Registers
4:     ; Get Status and Cause Registers
5:     mfc0 $26, $12, 0
6:     mfc0 $27, $13, 0
7:     ; Get Unmasked Interrupt Bits
8:     and $26, $26, $27
9:     srl $26, $26, 8
10:    andi $26, $26, 0x000f
11:    ; Store Return Address to End of the Handler
12:    la $ra, $Handler_End
13:    ; Check Bits Sequentially
14:    addi $t0, $0, 0
15:    beq $26, $t0, First_Interrupt
16:    addi $t0, $0, 1
17:    beq $26, $t0, Second_Interrupt
18:    addi $t0, $0, 2
19:    beq $26, $t0, Third_Interrupt
20:    addi $t0, $0, 3
21:    beq $26, $t0, Fourth_Interrupt
22:    addi $t0, $0, 4
23:    beq $26, $t0, Fifth_Interrupt
24:    addi $t0, $0, 5
25:    beq $26, $t0, Sixth_Interrupt
26:    addi $t0, $0, 6
27:    beq $26, $t0, Seventh_Interrupt
28:    addi $t0, $0, 7
29:    beq $26, $t0, Eighth_Interrupt
30:
31: Handler_End:
32:    ; Jump and Link to the Restore Part
33:    jal Restore_Registers
34:    ; Return to the Interrupted Address
35:    eret
36:
```

Semaphores are one of the simplest and most widely used software-based semaphores are commonly employed for concurrency control, process scheduling, and resource sharing. They ensure that only one thread can access a shared resource at a time, preventing race conditions and ensuring correct coordination between concurrent processes [13].

In multi-core systems, each core is equipped with a cache controller that facilitates shared data exchange between cores. These cache controllers communicate with each other, enabling one core cache to place shared data onto the shared bus, which can then be retrieved by another core cache controller. However, this process may introduce significant latency and may overload the shared bus, causing other cores and hardware components to stall until the cache controllers complete their communication [11]–[12].

To enable efficient data sharing, additional mechanisms must be implemented in multi-core systems. These include a memory consistency model that ensures all cores maintain a coherent view of memory at any time, and atomic instructions that lock memory locations containing shared data and prevent other cores from accessing them until they are released [11]–[12].

As discussed in the previous section, one common hardware-based method for synchronization is Inter-Processor Communication (IPC) that encompasses various models employing different mechanisms, as demonstrated in Xtensa and OpenRISC-based architectures. Inter-Processor Interrupts (IPIs) enable rapid inter-core notifications to indicate when shared data are to be retrieved or when task synchronization is required. Therefore, this work implements interrupt-driven synchronization as part of the proposed hardware-based semaphore solution [6]–[8].

II. Objectives

The objective of this evaluation is to determine whether interrupt-driven synchronization using hardware-based semaphores consistently outperforms its polling-based counterpart using hardware-based semaphores.

Additionally, the second part of the evaluation focuses on examining the performance improvement enabled by our proposed custom coprocessor for hardware-based semaphore synchronization, which is designed to accelerate data exchange by performing arithmetic operations concurrently with shared data retrieval.

III. Methods

The diagram in Fig. 10 provides an overview of the proposed quad-core system. Each core is equipped with two dedicated coprocessors. The first coprocessor is responsible for managing system control functions such as scheduling and handling interrupts from peripherals and other system components. The

second coprocessor is specifically designed to handle interrupt requests (IRQs) from hardware-based semaphores.

In addition, the coprocessor design enables the simultaneous execution of arithmetic operations while retrieving shared data from the hardware-based semaphores. This parallel processing capability aims to improve the performance of hardware-based semaphore implementation in multi-core systems by reducing the number of instructions executed by the cores during data exchange operations.

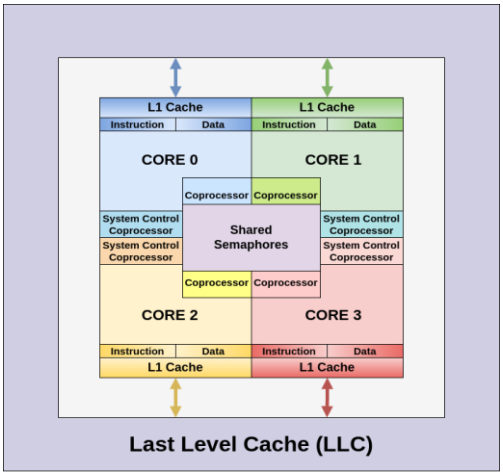

Fig. 10. Diagram of the multi-core system overview

A. Mapping the Hardware Semaphores

Hardware-based semaphores are mapped to a specific region of the physical address space, similar to how GPIOs and other peripherals are mapped. This configuration enables direct access to the hardware-based semaphores by all cores in the system. As a result, the implementation can be integrated into existing multi-core systems without requiring significant system modifications.

The hardware-based semaphores are organized as an array; the array is connected to modules corresponding to the number of cores. Each module includes an interrupt generation mechanism for the corresponding core and contains a register where each bit corresponds to a specific semaphore instance. The module generates interrupt requests for the corresponding core along with the address indicating the semaphore index to which new data has been written.

When a semaphore is triggered, the corresponding bit is set in a register, and a priority encoder is used to determine the lowest-indexed semaphore, even when multiple cores write simultaneously to different semaphores. In addition, the module operates sequentially, waiting for an IRQ acknowledgment from the core before issuing the next IRQ along with the address of the next lowest-indexed semaphore. Furthermore, each core maintains a dedicated queue to handle incoming semaphore-triggered IRQ signals. This mechanism ensures deterministic and orderly processing of semaphore events across all cores. The architecture of this module is shown in Fig. 11.

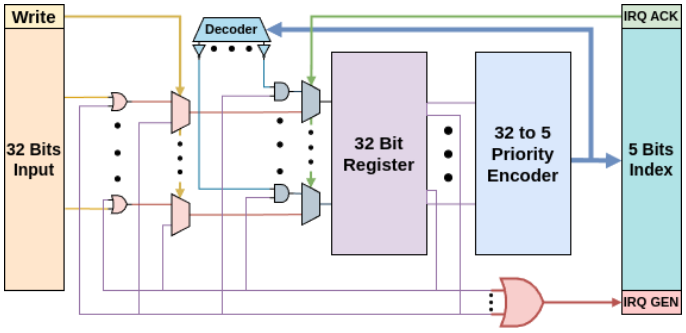

Fig. 11. Diagram of the module Handling IRQs

The hardware-based semaphores are indexed similarly to an array. Each core outputs an index value to indicate the semaphore used to store or load shared data, and this index is included with the generated IRQ signals. In addition, each semaphore includes an additional counter segment, as depicted in Fig. 12. This counter tracks the number of processes that must read shared data from the semaphore before new data can be written. This mechanism ensures proper synchronization and control of data placement.

`The maximum value of the counter segment corresponds to the number of cores in the system. A load signal issued by a core decrements the counter segment of the semaphore specified by the core index value. In addition, the hardware-based semaphores can be configured so that eight bits in the counter segment correspond to eight individual cores. Setting the bit corresponding to a specific core triggers a corresponding interrupt request, enabling targeted and efficient inter-core signaling.

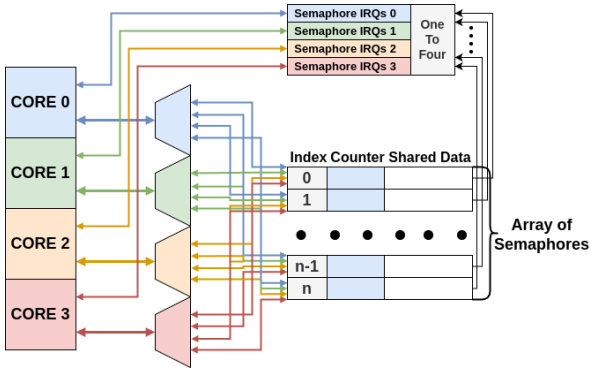

Fig. 12. Diagram of the Shared Semaphore Array

When a core issues a load signal to a shared semaphore specified by the index value, the bit corresponding to the core in the semaphore's counter segment is cleared. This configuration is designed to support IPC in the proposed implementation.

In this configuration, when data is stored in a semaphore and the bit corresponding to a particular core in the counter segment is set, an IRQ is generated by the corresponding module for the core. Along with the IRQ, the index of the hardware-based semaphore is provided to the core. This enables the core to identify the corresponding shared semaphore and trigger the corresponding ISR.

Cores with an unmasked bit corresponding to the semaphore index execute the ISR for the corresponding semaphore. After processing the IRQ, these cores send an IRQ acknowledgement. However, cores with a masked bit send the IRQ acknowledgement immediately without executing the ISR. The IRQ acknowledgement signal from each core clears the interrupt request bit in the corresponding module, but it does not decrement the counter segment or clear the bit associated with that core.

B. The Pipeline Modifications

Hardware-based semaphores are mapped to a specific region of physical memory, allowing direct access to the corresponding memory locations during load and store operations. When a load or store operation is executed on a hardware-based semaphore, the upper 16 bits of the memory address are loaded first. The value is then shifted left by 16 bits. These instructions are used to construct a 32-bit address that points to the memory region where the shared semaphores are located. This process is illustrated by the following MIPS assembly instructions.

Listing 5: Load Operation

```
1: ; Load Shared Data from the Semaphore Index of One
2:
3: addiu $a0, $zero, 0x6000
4: sll $a0, $a0, 16
5: addiu $a1, $a0, 0x0001
6: lw $t0, 0x0000($a1)
```

This process can be executed using only three instructions by placing the semaphore index in the offset field of the load instruction. Subsequent loads of shared data can be executed without the two initial instructions, as the base address of the specific physical memory region can be stored in a general-purpose register.

The same principle applies to storing shared data. To store shared data in a hardware-based semaphore, both the value of the semaphore counter and the semaphore index must be specified. These values are encoded in the address used in the store instruction. The middle segment of the address specifies the value of the semaphore counter, while the lower segment indicates the semaphore index. This value is precomputed by the compiler that generates the MIPS assembly code shown below.

Listing 6: Store Operation with Pre-Calculated Address

```
1: ; Store Shared Data of Two in
2: ; Pre-Calculated Value of 0x0301 that Combines
3: ; the Counter's Value of Three and Index of One
4:
5: addiu $a0, $zero, 0x6000
6: sll $a0, $a0, 16
7: addiu $a1, $a0, 0x0301
8: addiu $t0, $zero, 0x0002
9: sw $t0, 0x0000($a1)
```

This process can be optimized further by eliminating an additional instruction and placing the precomputed value directly in the offset field of the store instruction.

In the proposed implementation, we introduce two custom instructions to accelerate load and store operations involving hardware-based semaphores. The management of these shared semaphores is delegated to a local semaphore controller, similarly to how data memory is accessed through a local memory controller. Moreover, rather than employing spin-locking mechanisms that may lead to inefficient processor utilization, our design adopts a pipeline stalling strategy for cores awaiting access to resource-controlling shared semaphores.

Fig. 13 depicts the local semaphore controller positioned in the memory stage that interacts with hardware-based semaphores similarly to a data memory controller. In addition to supporting custom instructions, the controller supports access via conventional load and store instructions using standard read and write control signals.

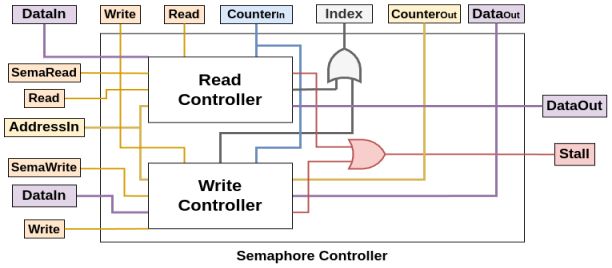

Fig. 13. Diagram of Semaphore Controller

The next two figures illustrate the read and write components of the controller, respectively.

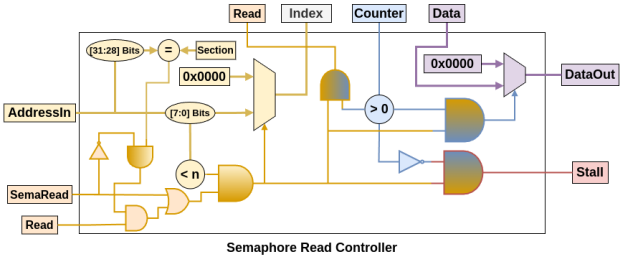

Fig. 14. Diagram of Semaphore Read Controller

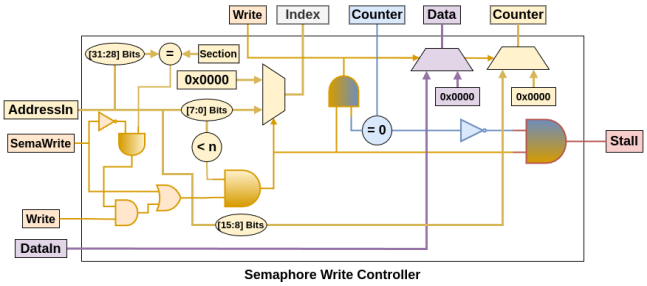

Fig. 15. Diagram of Semaphore Write Controller

The following MIPS assembly code segments illustrate custom load and store instructions, respectively.

Listing 7: Semaphore Load Operation

```
1: ; Load Shared Data from the Semaphore Index of One
3:
4: semaLoad $t0, 0x0000($a1)
```

Listing 8: Semaphore Store Operation

```
; Store Shared Data of Two in future work
; the Counter's Value of Three and Index of One

addiu $a1, $a0, 0x0301
addiu $t0, $zero, 0x0002
semaStore $t0, 0x0000($a1)
```

B. Coprocessor Design

The three-stage pipeline design depicted in Fig. 16 highlights modifications required to support the proposed implementation. The architecture incorporates a dedicated coprocessor alongside the system control coprocessor. Data storage to hardware-based semaphores is performed during the memory access stage of the pipeline. However, data loading from shared semaphores can occur at the decode stage or the memory access stage, depending on whether IPIs are used to notify cores of the availability of shared data.

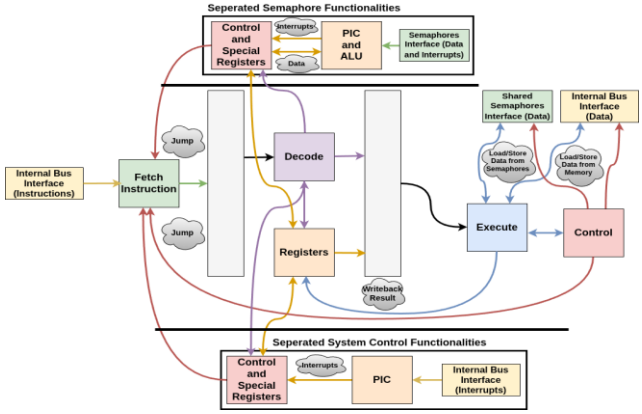

Fig. 16. Diagram of a Pipeline with the Shared Semaphore Interface

In the proposed implementation, a custom coprocessor is integrated alongside the processor to accelerate computation on shared data exchanged between cores. This coprocessor includes an Arithmetic Logic Unit (ALU) designed to perform arithmetic operations on data retrieved from hardware-based semaphores. Computation is performed concurrently with the data retrieval process. This computation mechanism is illustrated in Fig. 17.

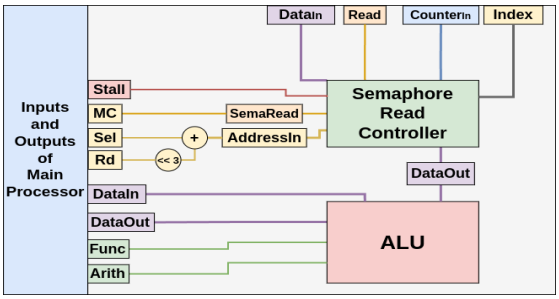

Fig. 17. Diagram Of Arithmetic Part of the Coprocessor

This mechanism operates without unmasking semaphore-related interrupts. Furthermore, the custom coprocessor does not include a register file, unlike a floating-point coprocessor, as all arithmetic computations are performed immediately during data retrieval. This design eliminates the risk of core stalling, thereby accelerating inter-core data exchange. By integrating arithmetic operations directly into the data retrieval process, the core avoids the overhead of executing separate arithmetic instructions after the data is fetched from a shared semaphore Inter-Processor Interrupts.

The following MIPS assembly code illustrates a segment of a control program in which shared data are stored by a core and subsequently retrieved three times using the proposed implementation to accelerate execution.

Listing 10: Semaphore Coprocessor Operation

```
; Store Shared Data of Two in
; the Counter's Value of Three and Index of One

        addiu   $a1,    $a0,    0x0301
        addiu   $t0,    $zero,  0x0002
        semaStore       $t0,    0x0000($a1)
        mfc2.and        $t1,    $zero,  0x01
        mfc2.and        $t1,    $zero,  0x01
mfc2.and $t1, $zero, 0x01
```

The second key feature of the custom coprocessor is its ability to handle semaphore-related interrupts, as depicted in Fig. 18.

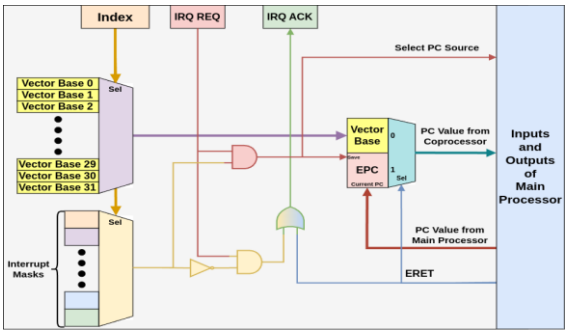

Fig. 18. Diagram Of IRQ Handling Part of the Coprocessor

When a semaphore-related interrupt is triggered, the program counter immediately jumps to the base address associated with that semaphore's interrupt vector and then to the corresponding ISR. This approach avoids the inefficiencies of a centralized interrupt handler, which would otherwise require sequential bit checking of the Cause register to identify the triggered semaphore-related interrupt.

Additionally, to accelerate the handling of semaphore-related interrupts, interrupt acknowledgment is performed automatically upon returning from the ISR. This eliminates the need for explicit acknowledgment instructions within the ISR.

The safety feature of our proposed implementation relies on the fact that the source core must set specific bits in the counter segment of the hardware-based semaphore corresponding to the destination cores. This action ensures that an IRQ is generated for each destination core. Additionally, the destination cores must unmask interrupts for this shared semaphore to properly handle the generated interrupt. However, if interrupts remain masked, then interrupt acknowledgment is generated immediately to clear the interrupt request bit of the corresponding core.

Hardware-based semaphores are always managed as a queue, indexed sequentially from the first shared semaphore to the last, even in this configuration. Although shared semaphores can be assigned for specific data exchanges between multiple cores,

semaphores with lower indices are always given priority when multiple different semaphores are used simultaneously for data exchange targeting the same destination core. In the IRQ configuration, hardware-based semaphores with lower indices are assigned higher priority, similar to standard IRQ behavior.

C. Methodology

To ensure consistency across all test scenarios, it is assumed that the multithreaded program uses a single shared semaphore for each data exchange operation. By focusing on this model, both approaches are compared and their effectiveness is assessed under correct usage.

The part of the proposed custom coprocessor for hardware-based semaphore synchronization that performs arithmetic operations concurrently with shared data retrieval comes at the cost of increased hardware complexity, as it requires an additional ALU, which is resource-intensive in terms of logic gate count and silicon area.

In this evaluation, the ALU consumes approximately 2,000 lookup tables (LUTs), while the entire core occupies around 5,000 LUTs. Therefore, it is important to assess whether the performance gains provided by the ALU within the coprocessor justify its hardware cost.

Therefore, it is essential to compare four distinct system configurations. The first configuration operates without the custom coprocessor and IRQ support, the second operates with the coprocessor but without IRQ support, the third configuration operates with the custom coprocessor and IRQ support, and the fourth operates with IRQ support but without the custom coprocessor.

Furthermore, in the testing scenarios, multithreaded programs utilizing IRQs for concurrency control operate in a loop within the main execution flow while also handling triggered semaphore ISRs. Within these ISRs, shared data are loaded from and stored in shared semaphores. In contrast, multithreaded programs without IRQ support rely on sequential but well-coordinated synchronization between multiple cores.

In order to evaluate the effectiveness of hardware-based semaphore synchronization in the previously described scenarios, it is essential to design test cases that represent fragments of multithreaded programs commonly encountered in safety-critical systems. This includes the previously mentioned MILS architecture, as illustrated in Fig. 2. In this software architecture, non-interference information flow is enforced through the use of Application Programming Interfaces (APIs), which is achieved by using bounded buffers between processes.

D. Test Cases

In this evaluation, three test cases were used. The first test case depicted in Fig. 19 represents a composite scenario that incorporates a mix of different data-flow patterns. The primary objective of this test case is to evaluate the average performance of hardware-based semaphore synchronization using the four configurations mentioned previously.

The second test case represents a scenario that mirrors a common data-flow pattern in safety-critical software systems, where a central (main) module receives collected data from all other modules and subsequently distributes the computed information back to them (Fig.20).

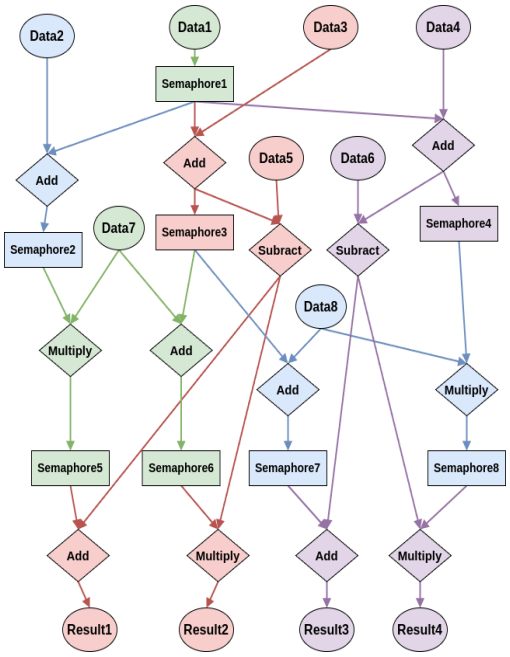

Fig. 19. Diagram Of the First Test Case

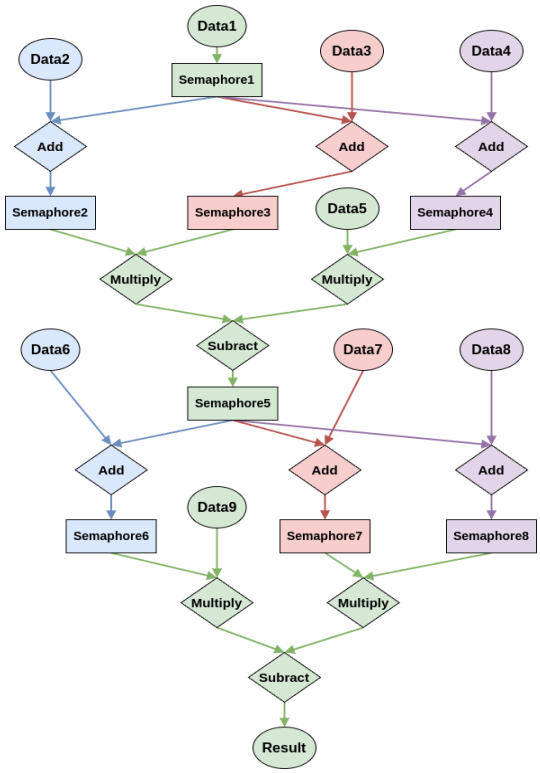

Fig. 20. Diagram Of the Second Test Case

The third test case does not represent a realistic software design for safety-critical systems; however, it simulates a plausible data-flow scenario between multiple modules, representing a single information exchange within the overall system architecture.

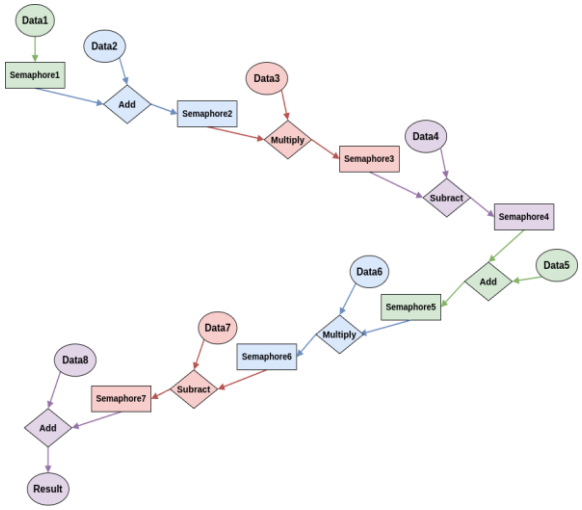

Fig. 21. Diagram Of the Third Test Case

IV. Results

The results of these test cases are shown below for the four configurations mentioned previously.

TABLE I
RESULTS

Test Case	Config 1	Config 2	Config 3	Config 4
First	18 clocks	17 clocks	44 clocks	44 clocks
Second	29 clocks	27 clocks	85 clocks	85 clocks
Third	30 clocks	30 clocks	70 clocks	69 clocks

V. Discussion

Inter-processor interrupts are commonly used for fast synchronization, as used in Xtensa-based and OpenRISC-based MPSoCs. However, as demonstrated in this article, a faster synchronization method can be achieved through polling using hardware-based semaphores. This approach is only effective if the multi-core compiler can generate well-synchronized data-exchange accesses between multiple threads.

The slowdown in multi-core synchronization when using inter-processor interrupts is primarily caused by the hardware mechanisms involved in IRQ handling. As depicted in the waveform in Fig. 22, the delay arises from saving and restoring the program counter, as well as from the vector jump to the interrupt handler and the return sequence. These operations stall and flush the pipeline to ensure accurate restoration of the program counter.

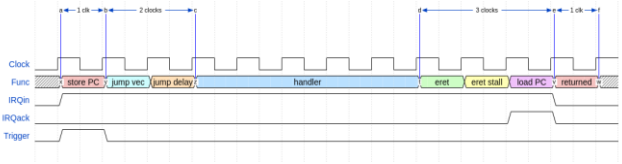

Fig. 22. Waveform of the Delays during Handling IRQ

Additionally, as shown in another waveform in Fig. 23, these IRQ handling operations must be fully completed before another interrupt can be serviced. As a result, these delays accumulate, resulting in poorer performance in our test cases. In contrast, the polling method avoids these costly operations, enabling faster multi-core synchronization by eliminating the need for interrupt handling routines.

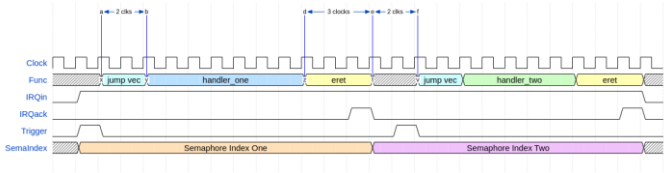

Fig. 23. Waveform of the Delay between IRQs

Furthermore, the performance gains achieved by offloading concurrent arithmetic operations alongside shared data retrieval to the custom coprocessor were minimal and did not justify the associated hardware cost. This is primarily because shared data is loaded during the instruction decode stage, causing half of the pipeline to stall, resulting in a phenomenon similar to a jump instruction, where the pipeline is effectively flushed. Noticeable acceleration occurs only in rare cases where the processor loads multiple shared data items sequentially, without any intervening instructions or delays.

VI. Conclusion

Furthermore, as discussed in this article, coprocessors are commonly used to offload various operations from the main processor. This separation not only improves performance but also enhances system safety by isolating critical tasks. In our implementation, a custom coprocessor was designed to handle IPIs and accelerate computations by offloading arithmetic operations performed after loading shared data. However, the performance gains observed from offloading arithmetic operations during shared data retrieval were minimal.

Interrupt-driven synchronization introduces delays that accumulate over time, thereby resulting in reduced performance. In contrast, polling-based synchronization is only faster when the multithreaded compiler can automatically partition the workload and introduce appropriate synchronization points for a multi-core system.

References

[1] Yongwang Zhao, “A survey on formal specification and verification of separation kernels”, 2016
[2] Adrian Garcia Ramirez, Julien Schmaltz, Freek Verbeek, Bruno Langenstein and Holger Blasum, “On Two Models of Noninterference: Rushby and Greve, Wilding, and Vanfleet”, 2014
[3] Markus Heinrich, Tsvetoslava Vateva-Gurova, Tolga Arul, Stefan Katzenbeisser, Neeraj Suri, Henk Birkholz, Andreas Fuchs, Christoph Krauß, Maria Zhdanova, Don Kuzhiyelil, Sergey Tverdyshev and Christian Schlehuber, “Security Requirements Engineering in Safety-Critical Railway Signalling Networks”, 2019
[4] Gerry Kane and Joe Heinrich, “MIPS RISC Architecture”, 2nd edition, 1991
[5] Muhammad Ali Mazidi, Sarmad Naimi, Sepehr Naimi and Shujen Chen, “ARM Assembly Language Programming & Architecture”, 2016
[6] “OpenRISC 1000 Architecture Manual”, 1.4th, revision, 2022
[7] Tomonari Tohara, “A New Kind of Processor Interface for a System-on-a-Chip Processor with TIE Ports and TIE Queues of Xtensa LX”, 2005
[8] Gulbin Ezer, “Breaking the I/O Bottleneck for High Compute Performance Processing with Xtensa LX Configurable and Extensible Processor Architecture”, 2005
[9] “MIPS Architecture For Programmers Volume I-IV”
[10] Grant Ayers, https://github.com/grantae/OpenMIPS
[11] Yan Solihin, “Fundamentals of Parallel Multicore Architecture”, *Chapman and Hall/CRC*, 2015
[12] Daniel J. Sorin, Mark D. Hill and David A. Wood, “A Primer on Memory Consistency and Cache Coherency,” Morgan & Claypool Publishers, 2020
[13] Michael L. Scott, “Shared-Memory Synchronization”, *Morgan & Claypool Publishers*, 2010