The Journal of Supercomputing (2023) 79:17000–17019
https://doi.org/10.1007/s11227-023-05304-1

Development an efficient AXI-interconnect unit between set of customized peripheral devices and an implemented dual-core RISC-V processor

Demyana Emil[1] · Mohammed Hamdy[1] · Gihan Nagib[1]

Accepted: 13 April 2023 / Published online: 5 May 2023

Abstract

RISC-V set architecture is playing an increasingly important role in processor technology due to its open instructions which allow researchers to build and improve computing systems. However, many RISC-V architectures exist in multi-core architecture with complex designs, large area, and high-power consumption. This paper studies an open-source multi-core RISC-V processor in a simple design with less power consumption. The processor depends on an open-source single RISC-V core processor, Taiga. Two cores of Taiga are integrated on a single chip while addressing issues related to cache coherence, interconnect, and memory design. A solution has been developed to achieve data coherence between implemented caches and the main memory; its architecture depends on the snoopy protocol. A hardware-customized peripheral unit has been implemented to achieve the management process among operated cores' tasks. For more consistent and highly controlled memory storage, the main memory unit has been designed in dual-port based on a specific protocol in the interface, and 8192 lines and word addressable unit. As the UART is common in several devices and processors for communication, the UART as a peripheral customized device has been appended for communication with other devices. The processor has been implemented in System Verilog HDL and extensively tested on various testbenches to ensure correct functionality. Hence, the system performance has been evaluated using the CoreMark benchmark, and achieved 4.605 CoreMark/MHz on Zedboard (FPGA Xilinx family) with a maximum operating frequency 98 MHz. The results indicate that the processor performs comparably to state-of-the-art multi-core processors, while offering a simpler and more power-efficient design. Overall, the research demonstrates the potentialof RISC-V architecture in creating a simple and power-efficient multi-core RISC-V processor.

Keywords Interconnect unit · Cache coherence · RISC-V multicore · Hardware synthesizable memory · System verilog · RTL

✉ Demyana Emil
Dem11@fayoum.edu.eg

Extended author information available on the last page of the article

1 Introduction

Single-core processors have reached their potential due to physical limitations. To achieve higher performance, parallel processing techniques can be used at both the hardware and software levels. Multicore processors have become popular for various applications including personal computers, supercomputers, and smartphones. RISC-V is a new RISC instruction set architecture that is open-source and categorized into several sets. These sets can be implemented based on a specific design target. Since their introduction in 2011, RISC-V designs have been recognized for their high performance, modularity, flexibility, and open-source specifications and development tools.

Most multicore RISC-V processors have been implemented as a closed source which makes it difficult for further modification and enhancement. This paper discusses an implementation of an open multicore RISC-V processor.

While several multiprocessor RISC-V implementations exist [1–4], they are not open-source [1, 2]. Most of those open-source works in a machine mode only and don't support OS [3]. Those are highly designed as soft-processor such as the RISC-V Rocket processor [5]. It supports both in-order and out-of-order schedules. These designs have components off-chip such as a memory controller and a main memory and don't aim at FPGA platforms due to the extensive resource usage.

The novelty of our design is that the processor is designed on-chip as a dual-core processor and is extendable for more than two threads. It is optimized enough to keep high performance. The maximum operating frequency is so near that of the single-core Taiga. The processor is in-order but the compiler is optimized to minimize the processor's execution time. It is optimized to re-order and minimize instructions. The design targets different FPGA platforms. It is available as an open to provide opportunities for education and research, and academic students to develop their skills. The proposed processor is based on the open-source single-core RISC-V implementation by Matthews and Shannon, which includes a partial implementation of atomic instructions [6]. The proposed design comprises two Taiga cores, interconnecting, synchronization, and a synthesizable hardware memory [7]. Due to the small number of used cores in this design, a snoopy protocol was applied to achieve cache coherence [8]. This protocol will be optimized to reduce the amount of accessing memory. The management process that distributes tasks on the two threads, depends on an implemented simple peripheral unit [9].

The rest of this article is organized as follows. Section 2 offers a review and comparison between several state-of-the-art processors. Section 3 presents the proposed design, including an implemented peripheral device to achieve synchronization among operated cores, a UART unit for sending a specific data on serial, a pseudocode of programs run on our processor and steps of this process. Additionally, it explains an interconnect topology connecting all peripheral devices with working cores, and the design of the implemented memory. Section 4 discusses a noncoherent data problem and how we achieved cache coherence under the snoopy protocol. Section 5 introduces a benchmark resultof multicore operations. Section 6 concludes the article and discusses future work.

2 Comparison of RISC-V processors

Several RISC-V processors were reviewed and compared in terms of performance parameters and architecture level. The sweRV-EH2 processor was designed for small systems such as microcontrollers [10]. It has small tightly coupled memory in place of a cache. Additionally, the processor works in machine mode only. The Andes processor was designed as single-core but supports multicore functionality based on RISC-V ISA [11]. However, it does not contain a data cache (D-Cache) and fixed latency execution units. Moreover, the design is not open source.

The Taiga processor is a RISC-V single-core open-source processor [12]. Taiga is ready to support an operating system, as it was implemented with D-Cache and instruction cache (I-Cache). Each cache is 16 KB. The processor supports a variable latency unit, especially in the execution stage; making it available to add extra units with variable latency, and it also has atomic instructions. The design is modular with a variable-length pipeline, thereby enabling the addition of new functional units [6]. Additionally, it has an option of adding cache units, multiplication units, efficient division units, and two- or four-way set-associative (16 or 32 KB, respectively) caches. It supports three privilege modes: machine, supervised, and user levels [13]. The design can interface with various protocols, namely, AXI interface [14], Avalon interface [15], and Wishbone interface [16]. It was written in System Verilog HDL.

The below table shows the scores of the Taiga processor compared with those of other single-core RISC-V processors in two benchmarks. These benchmarks are standard for usages such as Dhrtsone and CoreMark. These scores are published results (Table 1).

The Lagorta project is an implementation of a superscalar multi-core platform based on a RISC-V processor [21]. Initially, developed as RV64I. It was later extended to a multi-core asymmetric processor. The project has implemented atomic instructions for the synchronization processes between multi-core processors and solved the problem of cache coherence in the P-Mesh system developed by OpenPiton, which is an open-source design. However, the Lagorta project has some limitations, such as being partially designed using Verilog HDL, which can be confusing for hardware developers. Additionally, it does not use standard HDLs such as Chiesl. The system is also not on-chip.

Table 1 Comparison among single RISC-V core processors on different benchmarks

Core name	Dhrystone (DMIPS/MHz)	CoreMark/MHz
Taiga	1.65	2.63
Rudolv [17]	0.736 … 1.815 (depending on Dhrystone implementation	1.354
PicoRV32 [18, 19]	0.516	–
Advpub, RISC-V 32-bit microcontroller on 65-nm silicon-on-thin-BOX (SOTB) [20]	1.27	2.4
Ariane	–	2.45

Taiga is the proposed single-core processor for the dual-core processor

PlackParrot [22] is an open multi-core RISC-V processor with four cores and six pipeline stages. It supports atomic instructions, single- and double-precision floating point operations, integer multiplication and division, compressed, fence instructions, and CSR instructions. It solves the problem of cache coherence based on the directory-controller protocol and has been designed using SystemVerilog HDL. It includes a built-in SoC design with accelerators and is capable of working on Linux systems. The project has achieved a CoreMark/MHz of 3.04 on an FPGA as published. SoC is mostly a target design for high performance but it is a challenge to save the dissipated power and not exceed the resource usage of an operated FPGA kit. We have achieved these factors in our design. The proposed design aims to address these challenges while providing an on-chip dual-core processor that can be extended to multiple threads while maintaining high performance. It is optimized to minimize execution time and supports cache coherence under the snoopy protocol. Additionally, it is available as open-source.

3 The proposed design

The proposed dual-core processor is built upon the single-core Taiga processor. Figure 1 depicts the block diagram of the Taiga processor.

The single-core design includes two types of memory: SIM-memory and local memory. SIM-memory is a 4 GB simulation memory that is used at the simulation level and is not synthesizable. It interfaces using the AXI protocol. On the other hand, local memory is a small-sized read/write (R/W) memory that is faster than the main memory for storing data and instructions.

The local memory is a HW synthesizable that is used on FPGA instead of the unsynthesizable main memory. (The design is configured to work on either the Local memory or the SIM-memory but the Local memory was only used in the single-core operation)

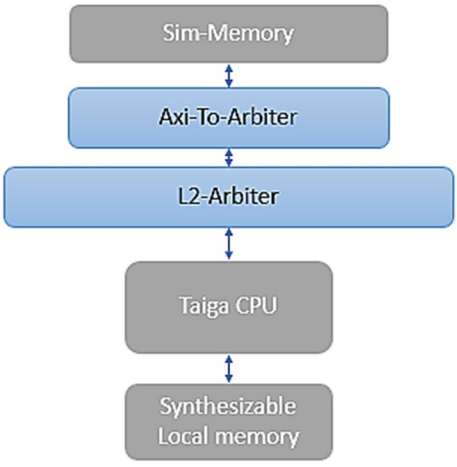

Fig. 1 Taiga open-source single-core block diagram

Taiga: is a single-core RISC-V processor that serves as a foundation for the proposed dual-core processor.

L2-arbiter: is an arbiter entity transferring requests from Taiga (more than one core) to the memory based on the round-robin arbitration method. A round-robin is a simple algorithm and a methodology for arbitration among multiple threads in cyclic order.

Finally, **Axi-To-Arbiter** converts arbiter request signals to AXI channels' signals enabling communication between the Taiga processor and memory.

Figure 2 depicts the block diagram of the proposed dual-core procressor.

3.1 Core management unit

The core management unit (CMU) is responsible for managing the operation of the cores in the proposed dual-core processor. The CMU includes a storage buffer with 32 locations in size, with each location index representing the core number. For example, location zero represents the state of core_1, location one represents the state of core_2, and so on. The other locations are reserved for future appended threads. The task of the CMU is to stop or run each core, and it does so use two output control signals halt1 and halt2. These signals are responsible for halting or unhalting core_1 and core_2, respectively. Control messages sent from software to the CMU update these signals.

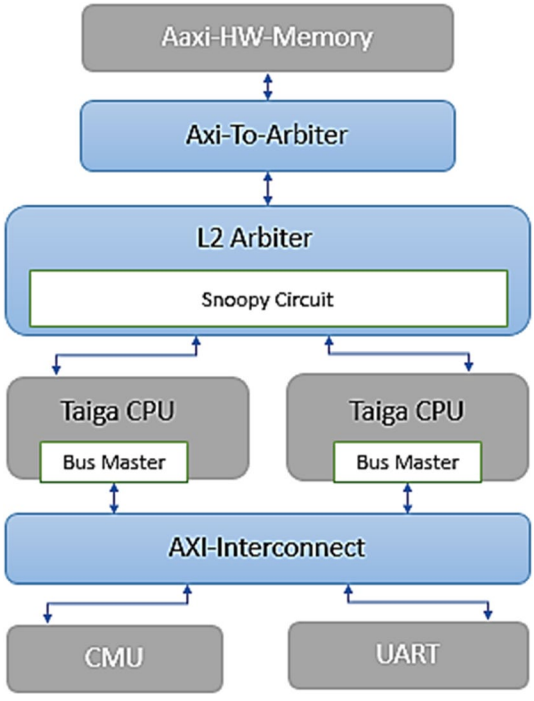

Fig. 2 Top design of dual Taiga RISC-V Core

In addition to the storage buffer, the CMU includes a control register called the core enable register. This register includes a status bit called "CTS" (core task select), which guides each thread for its current task.

Figure 3 shows the format of the CMU registers, providing a visual representation of the various components and their relationships to one another within the CMU.

The proposed dual-core processor includes several control bits that are used to manage the operation of the cores. These control bits are managed by a control circuit that coordinates their operation and ensures that the cores are executing instructions as intended.

The WRV (work at reset vector bit) is used to indicate that a core is in default mode.

WSA (work at a specific address bit) is used to indicate that a core is not in default and is executing instructions from a specific address.

Halt bit is a control bit that can be set to stop a core or cleared to run a core. When the Halt bit is set, the core stops executing instructions at the current PC and stops fetching the next instruction. Finally, the CTS bit (core task select) is a control bit that guides each thread for its current task. If it equals zero, the current task is related to Core_1. Otherwise, it is related to core_2.

The CMU control for read and write (R/W) operations is designed as a combinational circuit. For read operations, the available read addresses range from $0\times60{,}000{,}000$ to $0\times6000001f$, with each address representing the state of a specific thread as formatted in Fig. 3. However, address $0\times60{,}000{,}004$ represents the core enable register (CEReg).

Writing operations occur on a CMU control status register (CCSR) addressed by $0\times60{,}000{,}000$. The CCSR receives 8-bit data which is decoded to perform a specific action, such as halting or running a core. For example, if the software sends 0×00 to the CCSR, it means halt core2, and if it sends 0×07, it means run core1.

Core1 is controlled as the last core, as mentioned at the above decoding process, as it is assumed a default thread. The default thread is the first thread to be executed

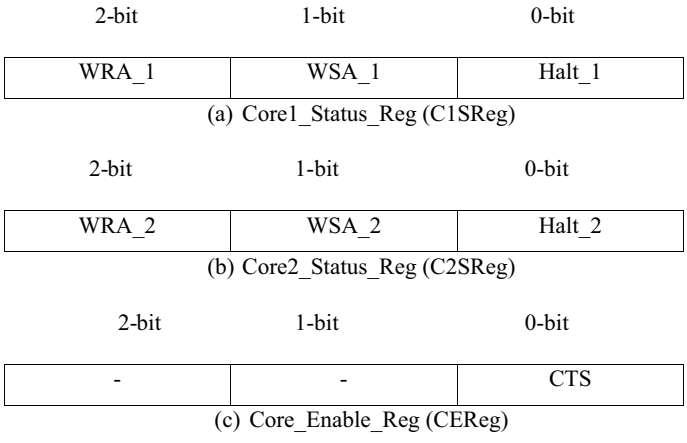

Fig. 3 Structure of CMU registers

and is never halted, but it waits at a specific address (addressed by the program counter) until other threads finish their tasks. Additionally, Core1 shares any other thread in distributed tasks (shared tasks).

Inkscape [23] has been used as a drawing tool, to develop the later diagrams.

Figure 4 provides a comparison of timing diagrams between read and write (R/W) operations in the proposed dual-core processor. The example shown in the figure involves unhalting core2 (a write operation) and reading the state of an arbitrary thread (a read operation).

3.2 UART

The proposed dual-core processor includes a transmitter/receiver circuits module as additional peripheral unit. This module is capable of sending and receiving one byte at a time and operates at standard baud rates such as 200, 2400, 4800, 9600, and 115,200 bps. The built-in baud rate for the module is 115200 bps, and it can only be modified by the hardware developer.

Figure 5 shows the pseudocode for the transmitter circuit, which is designed using the methodology generated on the Nandland site [24].

Overall, the transmitter circuit in the proposed dual-core processor is designed to efficiently transmit data on a byte-by-byte basis, using standard baud rates and a five-state machine to ensure reliable and accurate communication with external devices or circuits.

The transmitter circuit (Tx) is implemented as a five-state machine synchronized with the system clock. Firstly, an idle state indicates that Tx is valid. Then, if the UDR contains data, TX sends a start bit with a value of zero to indicate the start of transmission.

After sending the start bit, the Tx sends the 8-bit data, bit by bit, until all bits have been transmitted. Once the eighth bit of data has been sent, the Tx sends a stop

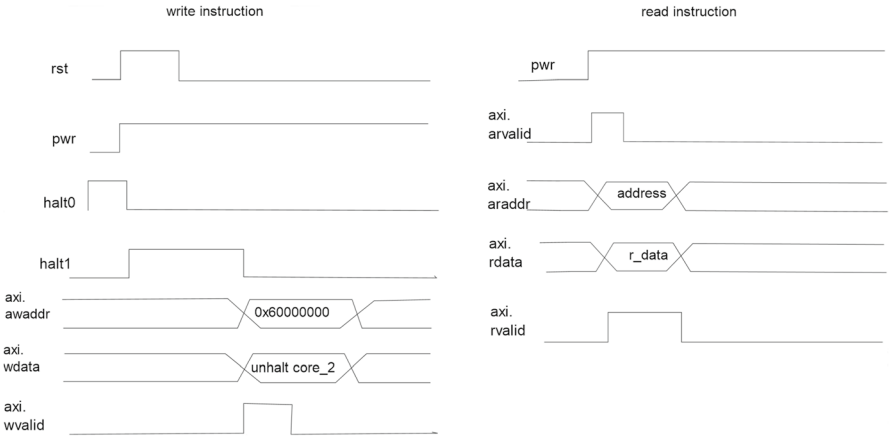

Fig. 4 Timing diagram of write and read operations on the CMU

Case (current state)

Begin

Idle: start to send 1 (referring to stop) on serial and jump to **start-bit state** in case of coming data

Start-bit: send 0 on serial (start-bit) and announce that UART is active and jump to **send-data state**

Send-data: start to send 8-bit data then jump to **stop-bit state**

Stop-bit: send 1 on serial and announce that UART isn't active. Then, jump to **clean-up** state

Clean-up: stay here only one clock cycle. Then, jump to the **idle** state

End

Endcase

Fig. 5 Pseudocode of UART transmitter module

bit with a value of one to indicate the end of transmission. The stop bit is configured to be one bit.

Finally, Tx takes one clock cycle to deactivate operated tasks that were performed during the transmission, such as setting a busy flag. This state is the clean-up state.

The receiver circuit is also implemented as a five-state machine, with the same states as the transmitter circuit (Idle, start-bit, receive 8-bit data, stop-bit, and clean-up state). The UART data register, which stores the one-byte data, is addressed by $0\times60{,}000{,}100$ and $0\times60{,}000{,}101$ in the case of transmitting and receiving, respectively.

3.3 AXI-interconnect unit

The AXI-interconnect unit, serves as a communication channel, delivering the requests of the two threads to the CMU and the UART. The choice of an interface protocol is essential because it simplifies the operation of a component that affects the hardware design area and delay. The AXI-interconnect unit is named as such because it depends on the AXI protocol for interfacing with the dual-processor.

AXI has a five-channels for interfacing, separating the read interface from the writing one that has simplified the unit architecture. It was designed based on multiplexer topology (MuxT) and is a soft on-FPGA network [25]. R/W operations based on a multiplexer topology (MuxT). The MuxT topology is used for both read and write operations, with the read operation being separate from the write operation.

Figure 6 shows the block diagram of the two peripheral devices, the CMU and the UART interfacing with the AXI interconnect unit. This figure also shows the writing Muxes of the AXI-interconnect.

The figure shows the interconnection between the processor and the two PDs. The black block is a controller that differentiates requests coming to the CMU from those coming to the UART. That depends on the address mapping of Table 2.

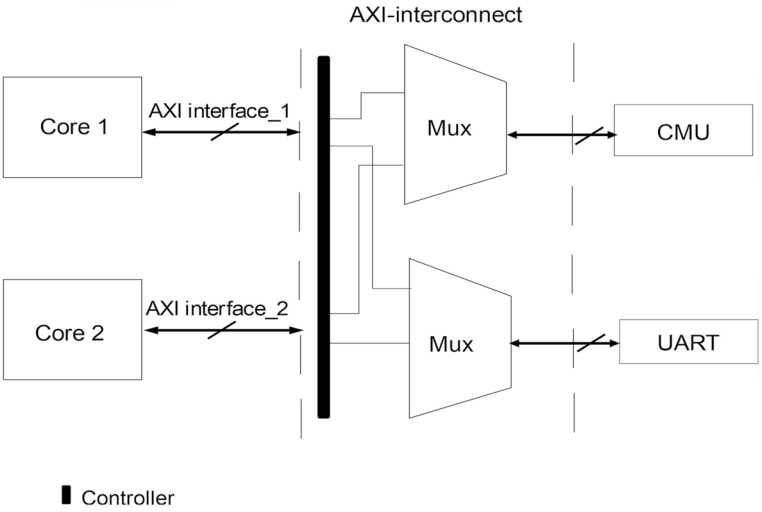

Fig. 6 Block diagram of the interconnection between the processor and peripheral devices

Table 2 Memory map of peripheral devices

Bus system	
Peripheral device	Address space
CMU	From 0×60,000,000 to 0×600000ff
UART (transmitter)	0×60,000,100 & 0×60,000,200 & 0×60,001,000
UART (receiver)	0×60,000,101 & 0×60,000,201

CMU and UART have specific I/O signals interfaced with the AXI interconnect. These signals are described as:

AXI interface_1: is an AXI interface connecting the core_1 to our AXI-interconnect.

AXI interface_2: is an AXI interface connecting the core_2 to our AXI-interconnect.

UART Interfaces:o_Tx_Byte: is 8-bit data to be transmitted on serial.o_Tx_DV: is the valid data signal that indicates new data existing on o_Tx_Byte.i_Tx_Done: is an input done signal. It points to that the UART has finished 8-bit data transmission.i_Tx_Active: in the case of being high (which equals one), it means that UART is busy.

S_machine: is a 3-bit signal indicating the current state of the UART.

CLKS_PER_BIT: is the number of clock cycles per bit that the UART takes to send one bit on serial.i_Clock: is the input system clock.i_Rx_DV: indicates that 8-bit of data exists in the receiving UDR.i_Rx_Byte: is the received 8-bit of data.

CMU Interfaces:

W_Address: is a 32-bit address line of the writing operation.

W_data: is 32-bit written data.
R_address: is a 32-bit address line of a read operation.
Arvalid: indicates that the read address bus has been updated by a valid address.
R_data: is 32-bit read data.
R_valid: indicates validating the read data bus.
W_valid: indicates validating the write data bus.
Aw_valid: indicates validating the write address bus.

Core_1 is prioritized to access the CMU over the other core, given its status as the default core. If both cores attempt to access the CMU simultaneously, Core_1 is granted access and the other core's request is queued for a maximum one clock cycle. However, parallel accesses to the CMU are infrequent as requests to halt/unhalt either core. They are typically made at the beginning and the end of an operated task.

On the other hand, either core can access the UART, with the first accessed core dominating the path to the transmitter (Tx) of the UART. If a core needs to send a specific sequence of 8-bit data, it can send to reserve the UART Tx by sending a request to lock it. This reservation prevents other threads from accessing the Tx until the transmission is complete. However, if the sender only has a single byte of data to transmit there is no need to reserve the Tx. Tx remains busy until it transmits the entire series of data has been transmitted.

The token flag, located at address $0\times60{,}001{,}000$, is used to reserve the UART Tx. The sender core requests to take the token flag to prevent other threads from accessing the UART transmitter. If either core sends a request (one) to this address before the other, it reserves the Tx. At the last byte of data transmission, the sender sends zero on the same address to unlock the Tx.

The busy status bit, located at address $0\times60{,}000{,}200$, is checked first before data transfer to ensure that the UART Tx is available. The data valid bit, located at address $0\times60{,}000{,}101$, indicates the presence of data in the receiver's UDR.

Figure 7 shows the timing diagram of the reading and writing transactions of the interconnect unit.

3.4 Main memory design

The proposed dual-port memory unit is designed to work synchronously for read and write operations, with portA designated for read operations and portB for write operations. The memory unit is synthesizable and has been designed to operate on-FPGA. It communicates with the two cores of the dual-core processor based on the AXI protocol interface. Figure 8 provides a dual-port synchronized memory Xilinx-supported.

The dual-port memory unit is instantiated at an outer control module that interfaces with the dual-core processor using the AXI protocol. This control module samples each AXI request first and then passes the required control, address, and data to the dual-port memory buses (as shown in Fig. 8). The dual-port memory unit has been configured with 8192 address lines, which can be configured by the

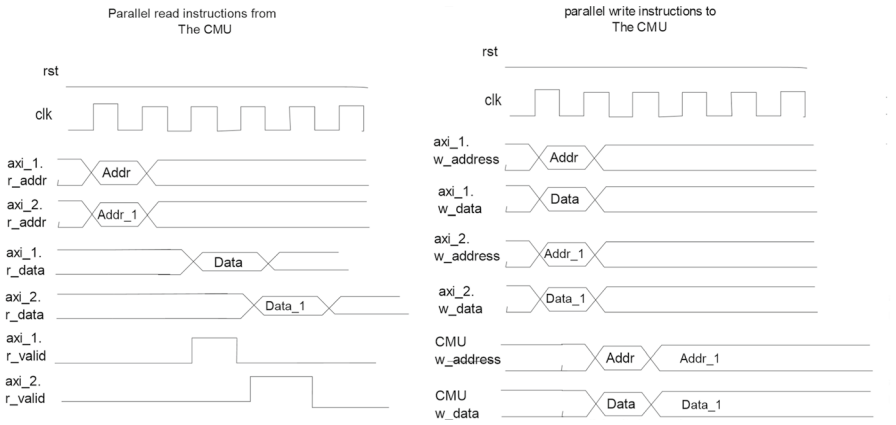

Fig. 7 Timing diagram of write and read operations for the CMU via AXI interconnect

hardware (HW) developer. When a read instruction is executed, the memory unit sends a block of memory that is four words in-depth.

3.5 Steps of parallelism

The program tasks in the proposed dual-core processor are manually distributed between the two cores. The software developer can divide the main task into dual-core or single-core tasks based on the task size or the developer's target. This approach can be particularly useful if the second core is implemented as a co-processor for a specific task. The software developer also has control over whether to operate the two threads in parallel. The CMU is the responsible unit for receiving and responding to control messages to operate or stop the two threads.

Algorithm 1 provides a structure for a distributed program on the dual-processor, where Core2 is the second core, and Core1 is the first and default core. The "C2Fin" variable indicates when Core2 has finished its task, and the CTS variable is a logic bit of the CMU's CEReg.

At the beginning of the program execution, both the CTS bit and the "C2Fin" variable are cleared. Core1 runs by default and begins fetching the first instructions as long as the CTS bit is zero. Once Core1 starts running, it sends a control message to the CMU to unhalt Core2. The CMU responds by unhalting Core2 and setting the CTS bit.

When the CTS bit becomes one, core2 begins executing its corresponding task. At the end of core2's task, it updates "C2Fin" by one. Core2 then sends another control message to halt itself. As soon as "C2Fin" is set, core1 continues to operate other tasks if they exist.

```
This module has:
    Input preload_file, Lines, Start_address
    Input clk, logic [$clog2(Lines)-1:0] addr_a, // portA is a read port
    Input en_a,
    Output logic[31:0] data_out_a,
     Input logic [$clog2(Lines)-1:0] addr_b,  // portB is write port
     Input logic [3:0] be_b, // it is a byte enable control signal
     Input logic [31:0] data_in_b
    logic [31:0] ram [Lines-1:0];
 initial
  begin
       $readmemh (preload_file,ram, 0, Lines-1);
  end
  always_ff @ (posedge clk) begin
            if (en_a)
             data_out_a <= ram[addr_a];
     end
  generate
  genvar i;
  for (i=0; i < 4; i++) begin
  always_ff @ (posedge clk) begin
          if (be_b[i]) begin
            ram[addr_b][8*i+:8] <= data_in_b[8*i+:8];
        end
     end
  end
  endgenerate
endmodule
```

Fig. 8 dual-port memory unit

Algorithm 1: Assumed pseudocode of uploaded program on the processor

```
Define Volatile variable C2Fin=0
Define a pointer to the address of CTS status bit addressed by 0x60000004
Define a pointer to the Control status register (CSR) register of messages sent to CMU
addressed by 0x60000000
int main (void) {
if (*CTS == 0)  // core_1 related task
{
    *CSR = 1; // Unhalt core_2
// corresponding core_1 task
wait until C2Fin equals to one ;    }
else                  // core_2 related task
 {
// corresponding core_2 task
C2Fin = 1
*CSR = 0; // Halt core2  }
// other statements    }
```

4 Cache coherence

Cache coherence is essential in ensuring that the memory system of a multiprocessor is coherent and consistent. To achieve cache coherence between these two cores in the proposed dual-core processor, a unit controller has been designed based on the snoopy protocol [26]. The snoopy protocol is suitable for this processor because it currently has only two threads, making it a low number.

The use of a simple protocol is preferable in a dual-core processor, and one such protocol is the snoopy protocol for cache coherence [27]. Figure 9 provides a visual representation of the snoopy circuit and its interface.

The snoop circuit, as shown in Fig. 9, is implemented inside the L2-arbiter and works in conjunction with other functions in the processor. The store variable is a logic bit that indicates whether the address on the address bus is related to a write or read instruction.

The snoopy circuit receives the 32-bit address of each core during a write instruction and passes it to an invalidation-response FIFO. This FIFO sends an invalidation request to each thread in case of an invalid address. The invalid address could be due to the snoopy circuit or other functions, such as synchronizing instructions.

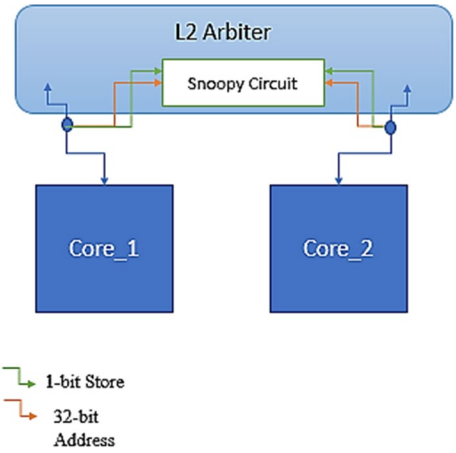

Fig. 9 Snoopy circuit interfaced with the two threads

Figure 10 shows the block diagram of implemented FIFOS related to the snoop circuit and other tasks of the L2 arbiter. The default task related to synchronizing instructions is given higher priority.

Figure 10 shows the data flow of snoop requests on snoop and invalidation_ response and input FIFOs. Input-FIFO is for invalidation addresses of tasks like synchronization. Snoop_FIFO's size depends on the expected number of invalidation requests of the input-FIFO. AT most, one request comes for each two clock cycles into the input_FIFO. The invalidation_response_FIFO takes one clock cycle to pop its input data. Therefore, the snoop-FIFO was sized as four in-depth rather than avoiding unexpected overflow issues. The pseudocode for the snoop control circuit implementation is shown in the figure below (Fig. 11).

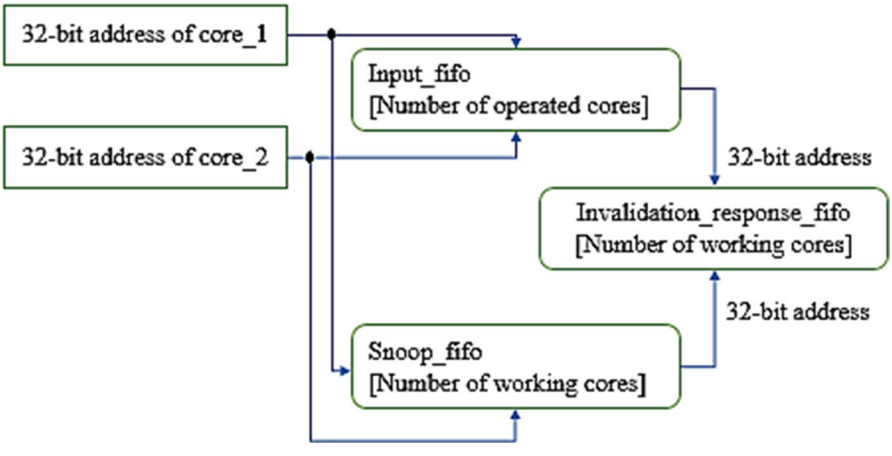

Fig. 10 L2-arbiter hierarchy FIFOS with the snoop-fifo

```
Define a FIFO of addresses to be snooped [Number of operated cores].
For (loop on the number of working cores by index i) begin
        For (loop on the number of working cores by index j)
                If (i !=j) begin
                        Snoop_fifo[j].push = request[i].writing // indicates writing transaction
                        Snoop_fifo[j].data_in = request[i].address
                End
End
For (loop on the number of working cores by index i) begin
        Invalidation_fifo[i]. pop = request[i].invalidation_ack
        Request[i]. invalidation_address = Invalidation_fifo[i].data_out
        Invalidation_fifo[i]. push = synchronizing_invalidation_request | size of snoop_fifo[i]
> 0
        Invalidation_fifo[i].data_in = (synchronizing _invalidation_request) ?
        input_fifo[i].data_out : snoop_fifo[i].data_out
        Snoop_fifo[i].pop = (not synchronizing _invalidation_request & size of snoop_fifo[i] >
        0)
End
```

Fig. 11 Pseudocode of appending snoop circuit inside L2 arbiter

5 Result and discussion

To verify the multi-processor, both standard and non-standard benchmarks have been used Fig. 12 illustrates the data flow of the top design for the test bench. The AXI-HW memory receives the program and each core fetches its related tasks based on Algorithm 1. If a halt/unhalt request (control message) reaches the AXI interconnect, it delivers it to the CMU based on the core highest priority. The CMU then updates the output halts signal to run/stop the operated threads. In each cache operation, especially during storing transactions, the owner core sends an invalidation request to the other thread via the snoop circuit implemented at the L2 arbiter. If address matching occurs, the listener invalidates the address's tagline. Each core can send on serial. If it requires sending a specific series of data, it can reserve the UART. Then, it should send a control message to release the UART.

Based on the RTL rules [28–30], the design was successfully implemented. Figure 13 shows the RTL blocks of the top design

Table 3 compares the resource usage of the proposed dual-core processor with other multi-processors and shows the stages of the optimization process. The Worst Negative Slack (WNS) is a positive value of 0.105 ns, and the Worst Hold Slack (WHS) is also positive with a value of 0.024 ns. The Total Negative Slack (TNS) and Total Hold Slack (THS) are both zero, indicating that the timing constraints have been met successfully.

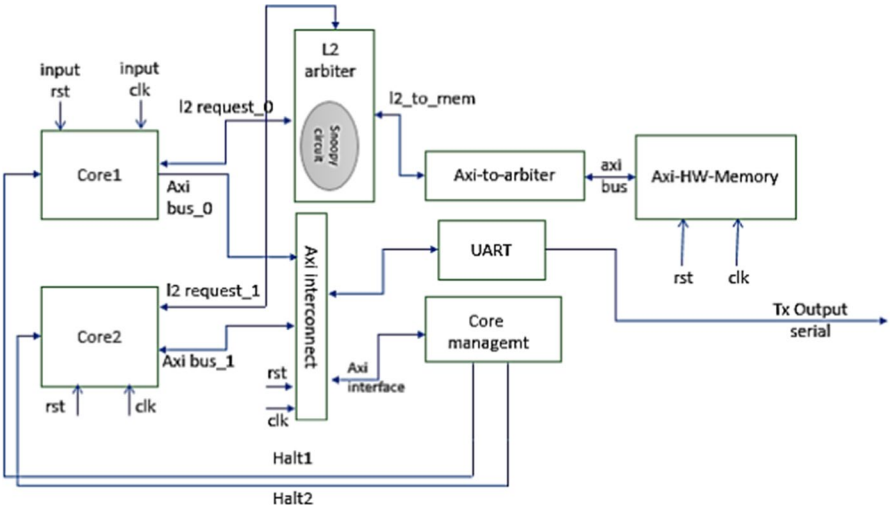

Fig. 12 Testbench module of system design

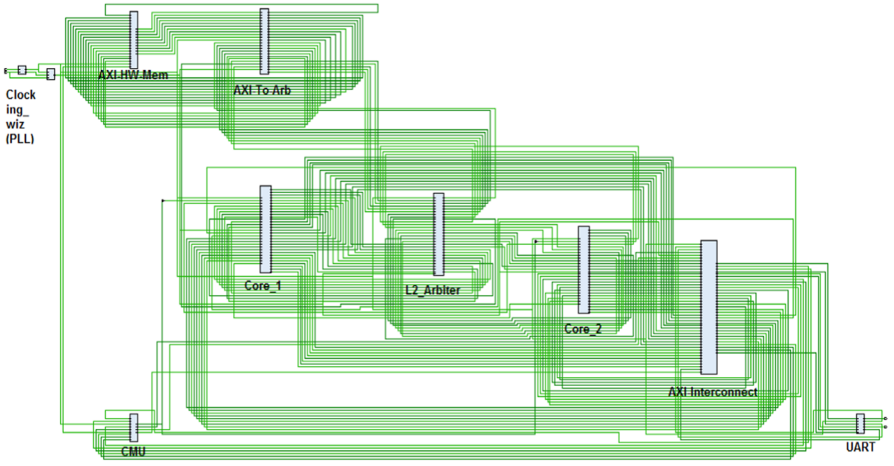

Fig. 13 RTL model of the processor

In Vivado, there are several strategies available for implementation, such as performance, placement, physical optimization, area optimization, and power optimization. Overall, the use of the performance-retiming strategy and the optimization process has resulted in a highly efficient and optimized dual-core processor with excellent performance and low power consumption. The power dissipation of the processor at a 98 MHz operating frequency is 0.325 watts.

CoreMark is widely used as a standard benchmark [32] for testing multiple-core processors. In this study, the CoreMark has been distributed to work on a dual-core processor depending on Algorithm 1. Each core receives its parameters

Table 3 Comparison in utilization resources among a set of RISC-V multiprocessors

Processor type	Utilization resources				Operating frequency (MHz)
Dual-Taiga top	LUT	FF	BRAM	DSP	
	6437	3161	32	8	98
	6394	3161	32	8	96
	6336	3185	32	8	90
	6608	3071	32	8	
	6617	3061	32	8	
	6766	3063	32	8	
Hydra (4-cores) Conventional system [31]	10,964	4181	–	–	
Hydra (4-cores) Composable system [31]	12,522	4517	–	–	
Rocket processor	17,144	9058	10	0	54

of addresses and begins operating its three modules. Table 4 provides a comparison of the proposed dual-core processor with other dual- and quad-core processors in terms of CoreMark/MHz score. The benchmark was performed on a Zedboard (Xilinx board) [33].

The design is also ready to support multi-core architecture for Operating Systems (OS), making it suitable for a range of computing applications. Furthermore, the processor is configured for academic research purposes, allowing researchers to use it for a variety of projects and experiments.

Table 4 Coremark result of quad- and dual-core processors

	CoreMark/MHz	Operating frequency (MHz)	Memory configuration	Vendor	Number of cores
Taiga dual-core processor	4.605	98	Stack	Xilinx	2
SunFire v210	2.39	1503	Heap	Sun microsystems	2
Esp32	4.13	160	Stack	Espressif	2
Sitara AM6442	6.525	1000	16-bit DDR4	Texas instruments	2
Ingenic × 2000	6.853	1200	DDR3L	Ingenic semiconductor	2
NXP	7.74	150	SRAM	NXP semiconductors	2
ZCU104	3.25	1200	DDR4	Xilinx	4
Raspberri pi 4B	22.67	2145	Heap	Raspberri pi	4

Underlined values refer to our implemented processor

The proposed dual-core processor design supports both heterogeneous symmetric/asymmetric computing architectures, making it versatile and adaptable to a range of computing needs. Heterogeneous architecture is becoming increasingly popular, especially in the Intel family of processors. Additionally, the processor is designed to be compatible with both Intel and Xilinx families of processors.

The processor is currently applicable for integer operations, but plans are in place to append a floating-point unit in the future.

6 Conclusion and future work

In this study, an open-source multi-processor (dual-RISC-V core) design has been discussed. Two peripheral devices have been presented. The CMU is for the management process between the two threads. The UART module is for sending on serial and is configurable to be reserved for one core for a while. Keeping on modularity, an AXI interconnect is designed, separated from the memory system path, for all peripheral devices to interface with the dual-processor. Some memory issues are discussed, for example, the noncoherent cache. Its solution has depended on the snoopy protocol. The design is ready for appending extra custom peripheral devices. There is a high range of address space that can cover other units. In addition, the resource usage is less compared with other multi-processors. The main memory depends on the AXI interface with the dual-processor. AXI protocol is FPGA sensed. The main memory is 8192 address lines and four words as a memory block for each reading operation. A standard benchmark tool (CoreMark) has been used to measure the performance. The processor has achieved a good result compared with the references. Also, several benchmarks have been developed for extensive testing. It shows how the performance of the design is consistent for complex programs.

In the future, we will modify the cache unit and snoopy circuit to make the listener core read data from the owner's L1 cache in case of matching instead of the main memory to reduce the number of clock cycles consumed for reading updated data from the main memory. We will append a floating unit in single and double precision. Additionally, we can append TC to the cache module [2]. Instead of AXI HW memory, we can use an off-chip DDR4 memory [34, 35] with an on-chip memory controller [36]. The design can be optimized more to keep the maximum frequency as it is in single-core Taiga (104 MHz).

Funding Open access funding provided by The Science, Technology & Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB).

Code availability All block design files that are developed by this work, are open and available on a Google drive link: https://drive.google.com/file/d/1lFkQGpzdTs8649PjTVWIRpyzekbg8GXJ/view?usp=share_link.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval Not applicable.

References

1. Li J, Zhang S, Bao C (2022) DuckCore: a fault-tolerant processor core architecture based on the RISC-V ISA. Electronics. https://doi.org/10.3390/electronics11010122
2. Semidynamics: High Bandwidth RISC-V IP Core (2020). https://semidynamics.com/products/atrevido
3. SCR1 RISC-V Core (2019) https://github.com/syntacore/scr1
4. RV12 RISC-V 32/64-bit CPU Core (2018). http://roalogic.github.io/RV12
5. Asanovi K, et al. (2016) The rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016–17
6. Matthews E, Shannon L (2017) TAIGA: a configurable RISC-V soft-processor framework for heterogeneous computing systems research. Semantic Scholar
7. Leyva-Santes NI et al (2019) Lagarto I RISC-V multi-core: research challenges to build and integrate a network-on-chip. In: Torres M, Klapp J (eds) Supercomputing. Springer, Cham. https://doi.org/10.1007/978-3-030-38043-4_20
8. Jang H et al (2021) Developing a multicore platform utilizing open RISC-V cores. IEEE Access 9:120010–120023. https://doi.org/10.1109/ACCESS.2021.3108475
9. Emil D, et al. (2022) Dual-RvCore32IMA: imlementation of a peripheral device to manage operations of two RvCores. In: 4th International Conference on Intelligent Computing, Information and Control Systems
10. EH2 SweRV RISC-V Core™ design RTL (2020) https://github.com/chipsalliance/Cores-SweRV-EH2
11. AndesCore™ AX45 (2018) http://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax45/
12. Taiga RISC-V processor (2019) https://gitlab.com/sfu-rcl/Taiga
13. Waterman A, Asanović K, Hauser J (2021) The RISC_V instruction set manual volume II: privileged architecture. University of California, Berkeley
14. Holdings ARM (2019) AMBA AXI and ACE protocol specification, ARM IHI 0022H.c
15. Intel: Avalon ® Interface specifications (2020)
16. Sharma M, Kumar D (2012) Design and synthesis of wishbone bus dataflow interface architecture for SoC integration. IEEE Xplore. https://doi.org/10.1109/INDCON.201206420729
17. RudolV by bobbl-RISC-V processor (2020) https://www.librecores.org/bobbl/rudolv
18. PicoRV32 -A Size-Optimized RISC-V CPU (2017) https://github.com/cliffordwolf/picorv32
19. PicoRV32 processor score. https://riscv.org/exchange/cores-socs/
20. Hoang TT, Duran C, Nguyen KD, Dang TK, Nguyen QNQ, Than PH, Tran XT, Le DH, Tsukamoto A, Suzaki K, Pham CK (2020) Low-power high-performance 32-bit RISC-V Microcontroller on 65-nm silicon-on-thin-BOX (SOTB). IEICE Electron Express 1:2. https://doi.org/10.1587/elex.17.20200282

21. Ramírez C, Hernández C, Morales CR, García GM, Villa LA, Ramírez MA (2017) Lagarto I—Una plataforma hardware/software de arquitectura de computadoras para la academia e investigación. Res Comput Sci 137:19–28
22. Petrisko D, Gilani F, Wyse M, Jung DC, Davidson S, Gao P, Zhao C, Azad Z, Canakci S, Veluri B, Guarino T, Joshi A, Oskin M, Taylor MB (2020) BlackParrot: an agile open-source RISC-V multicore for accelerator SoCs. IEEE Comput Sci 40:93–102
23. Moini et al.: INKSCAPE (2021) https://inkscape.org/release/1.1.1/windows/
24. Nandland. http://www.nandland.com
25. Dimitrakopoulos G, Kachris C, Kalligeros E (2011) Scalable arbiters and multiplexers for on-FGPA interconnection networks. In: 2011 21st International Conference on Field Programmable Logic and Applications, Chania, Greece, pp 90–96. https://doi.org/10.1109/FPL.2011.26
26. Alshehri M, Almakdi S, Alazeb A (2015) Cache coherence mechanisms. Int J Eng Innov Technol (IJEIT) 4:158–167
27. Ulfsnes R (2013) Design of a snoop filter for snoop based cache coherency protocols. Semantic Scholar
28. Gayathri S, Taranath TC (2017) RTL synthesis of case study using design compiler. In: International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT), pp 1–7. https://doi.org/10.1109/ICEECCOT.2017.8284603
29. Zhang Y, Ren H, Khailany B (2020) Opportunities for RTL and gate level simulation using GPUs (invited talk). In: IEEE/ACM International Conference on Computer Aided Design (ICCAD)
30. Alam SA, Gregg D, Gambardella G, Blott M, Preusser T (2022) On the RTL implementation of finn matrix vector compute unit
31. Marshall B, Page D, Pham T, Whale M (2022) HYDRA: a multicore RISC-V with cryptographically useful modes of operation
32. Gal-On S, Levy M Exploring CoreMarkAM a benchmark maximizing simplicity and efficacy
33. Zedboard Digilent. https://digilent.com/reference/_media/zedboard:zedboard_ug.pdf
34. DDR4 Memory Standard. https://www.kingston.com/en/memory/ddr4-overview
35. DDR4 SDRAM Specification (2014) Chapter 1 &2, Samsung Electronics
36. DDR4Memory controller. https://github.com/ananthbhat94/DDR4MemoryController

Authors and Affiliations

Demyana Emil[1] · Mohammed Hamdy[1] · Gihan Nagib[1]

Mohammed Hamdy
Mhm00@fayoum.edu.eg

Gihan Nagib
Gna00@fayoum.edu.eg

[1] Faculty of Engineering, Fayoum University, Fayoum City, Egypt