Open Design of Dual Core RISC-V Multicore processor

Demyana Emil, Mohammed Hamdy, Gihan Nagib, Islam Hussien

Faculty of Engineering, Fayoum University, Fayoum City, Egypt

✉ Demyana Emil
Dem11@fayoum.edu.eg

Abstract

RISC-V set architecture has a significant effect on processor technology due to open instructions that enable researchers to build and enhance computing systems. So, in this paper, an open-source multicore RISC-V processor is implemented. The design is based on an open-source single RvCore processor (Taiga). Two cores of Taiga are integrated considering cache coherence, interconnect, and memory synchronization problems. Therefore, a solution has been developed to achieve data coherence between implemented caches and the main memory; its architecture is mainly based on snoopy protocol. Additionally, a hardware customized peripheral unit has been developed to achieve the synchronization process among working cores. For more consistent and highly controlled memory storage, main memory unit has been designed in dual-port based on AXI protocol in interface, and 4096 lines and word addressable unit. The processor has been implemented in System Verilog HDL. Moreover, extensive testing of the system on various test benches was conducted to assure correct functionality. Hence, the performance of the system has been assessed using standard multicore benchmark (CoreMark). The processor performance has been shown to be comparable to state-of-the-art results.

Keywords: Interconnect unit; Cache coherence; RISC-V; multicore; Memory synchronization

1 Introduction

Single-core processors have reached their peak performance because of physical limitations. Higher performance can be attained by exploiting parallel processing techniques at the hardware and software levels. Presently, multicore processors are used for various applications, such as personal computers, supercomputers, and even smartphones. RISC-V is a new RISC instruction set architecture that is open source and well designed. It has been known for high performance since its inception in 2011 to date in addition to other advantages, such as modularity, flexibility, 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. Hence, this paper discusses implementation of multicore RISC-V processor. Its design is open to get to higher development.

Multicore processor design involves several issues that should be addressed to ensure correct operation and good performance. All modern processors contain cache memory to exploit spatial and temporal locality in memory accesses to enhance average memory access time. In a multicore processor, each core has its cache memory that must be synchronized to other caches and the main memory. Several synchronization paradigms exist, which are usually classified into snooping- and directory-based. Multicore processors and memory controllers must be connected with a well-designed interconnect topology to avoid contention for resources and memory wait cycles.

Several RISC-V implementations have been developed, from single-core to multicore implementations [1,2]. Open-source implementations also exist [3,4]. The proposed in this paper design is based on the work of Matthews and Shannon, which provided an open-source single-core implementation of RISC-V with partial implementation of atomic instructions [5]. The proposed design comprises two Taiga cores in addition to interconnect, synchronization, and memory interface hardware [6]. Due to the small number of used cores in this design, a snoopy protocol was applied to achieve cache coherence [7]. Synchronization process depends on an implemented peripheral unit [8].

The rest of this article is organized as follows. **Section 2** presents a review and comparison between several state-of-the-art processors. **Section 3** presents the proposed design. It discusses the pseudocode of programs run on our processor. It introduces an imlemented peripheral unit to achieve synchronization among working cores and shows the steps of this process. Additionally, it explains an interconnect topology connecting all peripheral devices with working cores. It also introduces the design of the implemented main memory. **Section 4** presents a noncoherent data problem and how to achieve cache coherence under the snoopy protocol. **Section 5** introduces a benchmark to test multicore operations and its result in addition to comparison among single RvCore processors. **Section 6** presents the conclusion and 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 microcontroller design [9]. It has small tightly coupled memory in place of 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 [10]. However, it does not contain D-Cache and fixed latency execution units. Moreover, the design is not open source. More processors in single-core architecture are discussed and introduced with Dhrystone and CoreMark scores in section of result and discussion.

The Taiga processor is a RISC-V single-core open-source processor [11]. Taiga is ready to support an operating system, as it was implemented with D-Cache and I-Cache. Each cache is 16 KB. The processor supports a variable latency uni t, 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 [5]. 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 [12]. The design can interface with various protocols, namely, Axi interface [13], Avalon interface [14], and Wishbone interface [15]. It was written in System Verilog HDL.

3 The proposed design

From the previous comparison, Taiga is the most recommended processor. Figure 1 shows Taiga processor structure which the proposed design is based on.

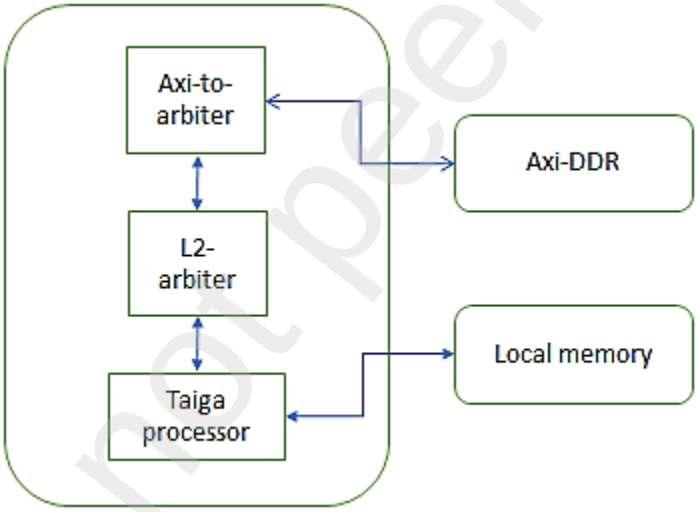


Fig. 1 Taiga open-source single-core block diagram

Taiga processor is a single-core processor. L2_arbiter is an arbiter that stores all requests of the processor to send them to DDR memory (main memory). Axi-to-arbiter is just a converter block that converts the general request interface coming from L2_arbiter to the Axi request interface, vice versa. DDR memory is implemented as simulation DDR with Axi interface. Local memory is a small-sized Read/Write (R/W) memory that is faster than DDR memory.

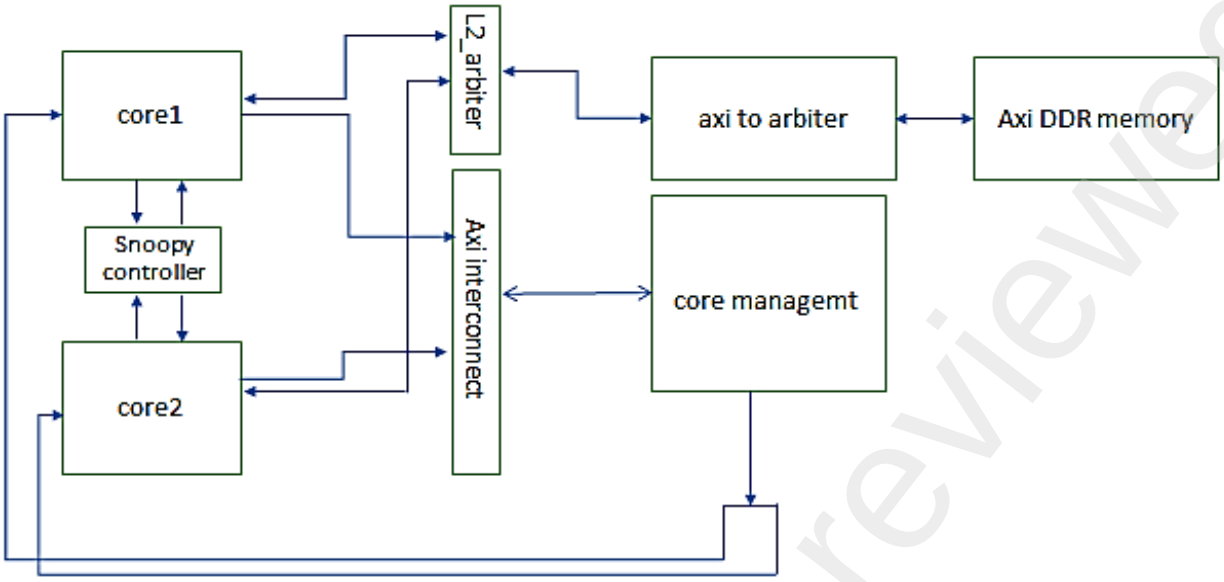


Fig. 2 Top design of dual Taiga RvCore

3.1 Steps of parallelism

Synchronization of working cores is the most important process in any multicore processor design. A core management unit is an external peripheral unit that was designed to organize the timing to run or stop each core. Before going on design details, we show an assumed pseudocode of available programs uploaded on the processor and the synchronization process steps. Algorithm 1 shows these steps, where core2 is the second core, core1 is the first core, and "C2Fin" indicates that core2 finished its task; core selection is a bit, inside the core management unit's memory, that carries the last working core id (zero or one); when it is cleared, it means that the last core is core1 but when it is set, it means that the last core is core2.

At the beginning, the core selection bit and "C2Fin" are cleared. Core1 runs by default and starts to fetch the first insrtuctions as long as core selection bit is cleared. Once core1 starts to run, it sends a control message to the core management unit to unhalt core2. The core management unit responds to unhalt core2 and set the core selection bit. When core selection bit is changed to one, core2 starts to execute its corresponding task. At the end of core2's task, it updates "C2Fin" by one. Afterward, it sends another control message to halt itself. As soon as "C2Fin" is set, core1 starts to execute the rest of the program until it reaches the end.

Algorithm 1 : Assumed pseudocode of uploaded program on the processor

```
Define C2Fin=0
Define a pointer to address of core_selection bit
Define a pointer to address of control messages
int main (void) {
If (core selection == 0) {
Unhalt core2
// corresponding core1 task
Wait until C2Fin is set      }
Else {
// corresponding core_2 task
C2Fin = 1
Halt core2   }
// other statements    }
```

3.2 Core management unit

On the design to implement the abovementioned parallelism process; an elementary block design was implemented (Fig. 3). Figure 3 shows all needed signals of the core management unit: the type of protocol used on its bus (Axi) to interface with each core. A power signal is an active-high input to control the unit to start/stop working, but it is written high by hardware to stop software from controlling it. Halt signals are to stop or run a corresponding core. The core management unit was designed to receive control messages from working cores. To store these messages, a small static memory was implemented inside it. This memory comprises 32 locations in size and 32 bit in width (Fig. 4). According to the written data, it generates halt signals for each core to stop or run the respective core. On halt, a core stops at its current instruction. Even if the core has started to execute the next instruction, it will flush it and stop.

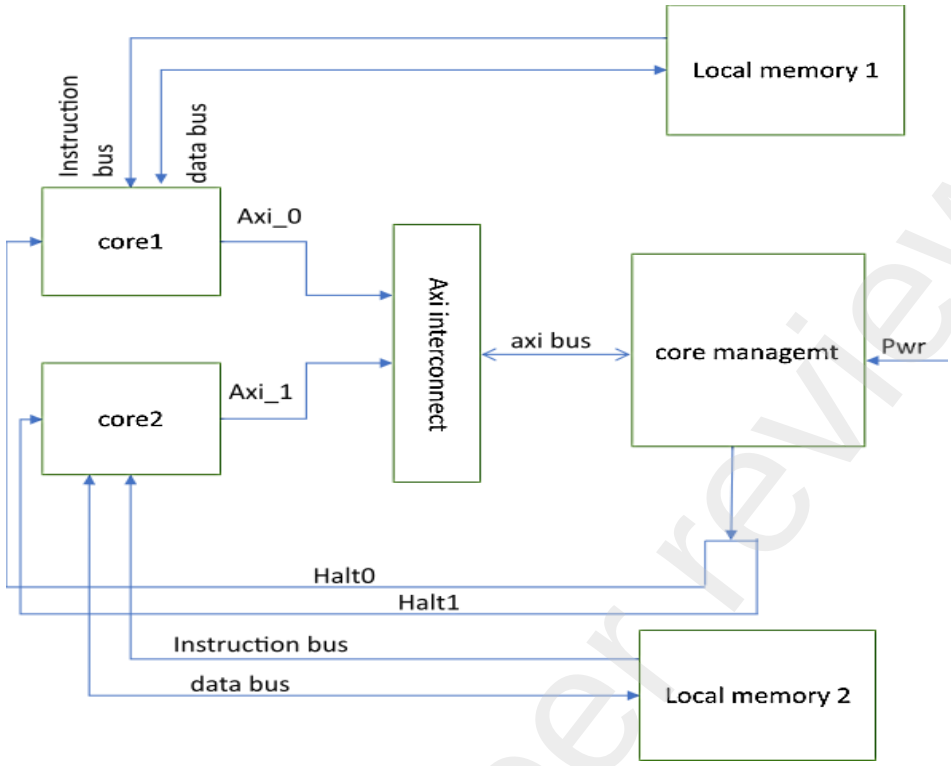


Fig. 3 Synchronized dual-core processor with Axi-interconnect, core management unit and local memories

	31	30	29	28:0
0	WRV	WSA	Halt	0x00000000
1	WRV	WSA	Halt	0x00000000
. .				
4	0	0	0	[28:1] =0, [0]=core selection
. .				
31				

Fig. 4 Core management's buffer format

WRV (work at reset vector bit) indicates that a core is in default.
WSA (work at specific address bit) indicates that a core is not in default.
Halt bit controls the dual-core processor to stop a core or to run a core if it equals 1; the core stops at the current PC and stops fetching the next instruction.
Core selection is used to define which core is running immediately.

As seen in Fig. 4, not all locations of the RAM are used, just three of them (two locations represent the case of two cores, but the third one contains an important variable called core selection), and the rest is for the future.

3.2.1 Control circuit

The control circuit of the core management unit for (R/W) operations is shown in flowchart format (Fig. 5 and 6) based on the received requests. The control of receiving and sending requests/data in this unit was implemented combinational. All the following flowcharts and timing diagrams were developed by Inkscape tool [16].

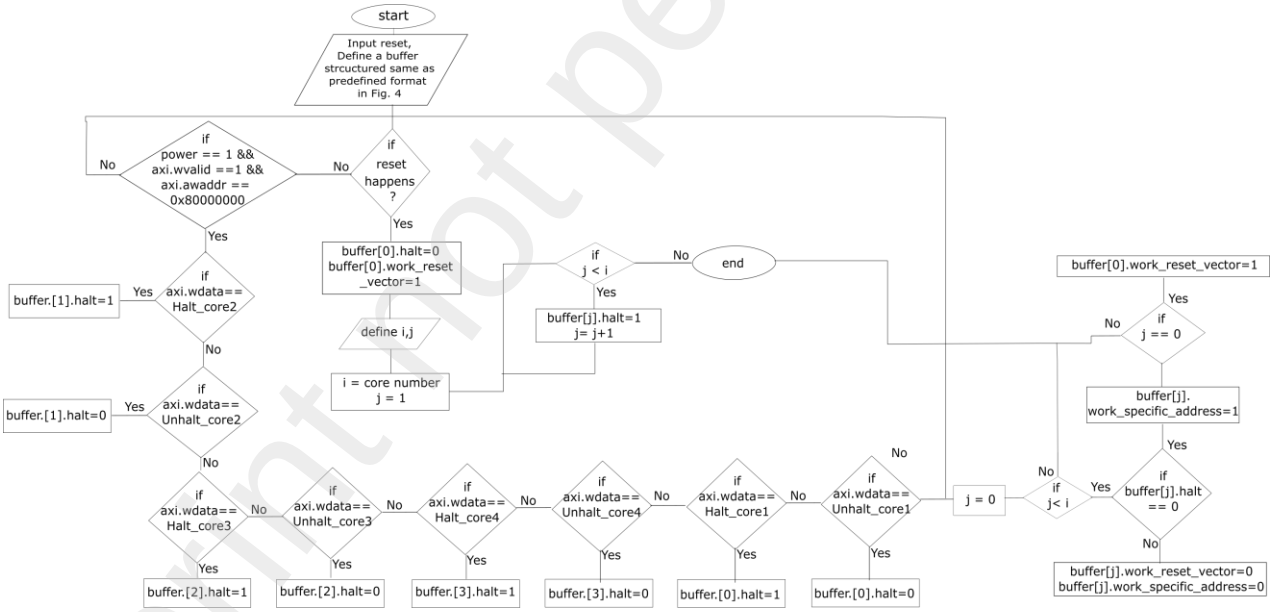


Fig. 5 Flowchart of core management's control circuit on write operation

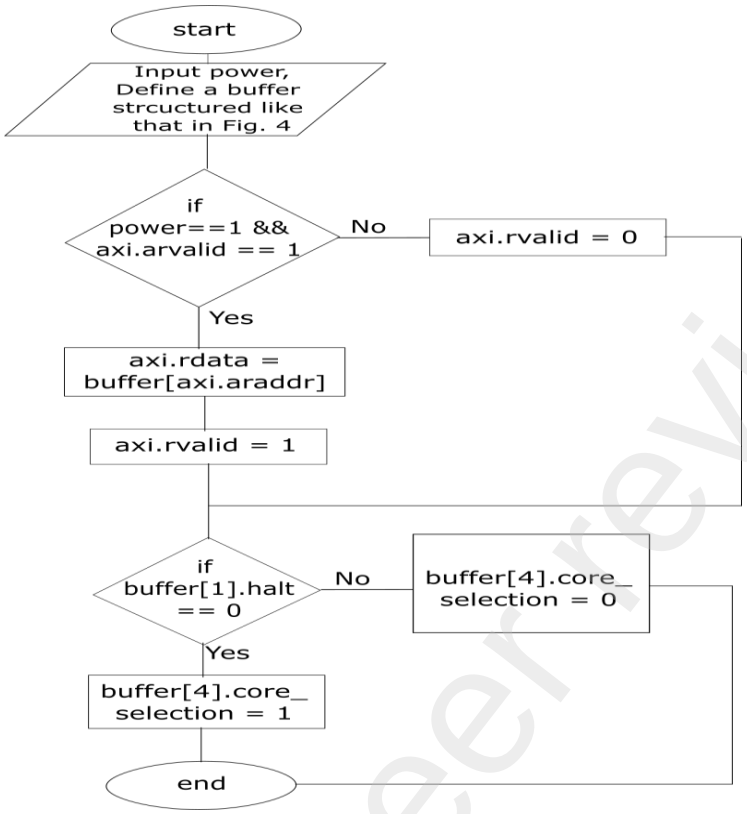


Fig. 6 Flowchart of core management's control circuit on read operation

Figure 7 shows a comparison in timing diagrams between the two operations (R/W). An example is shown to unhalt core2 for writing operation.

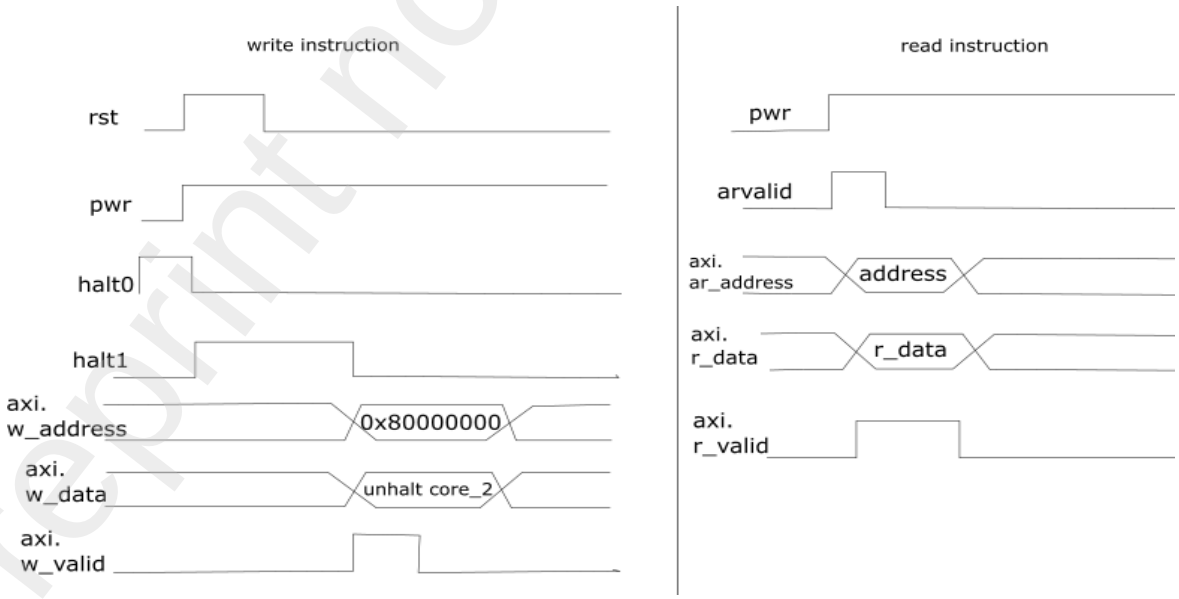


Fig. 7 Timing diagram of write and read operations on core management unit

3.2.2 Data control

On writing transactions to the core management, the written data entail just a control message. First, the message is decoded. Then, according to the message decoding, the core management unit decides to stop or run the on-chip cores. The decoded message must be sent to the address "0 × 80000000." For example, if the data are as follows.

- 0 × 00000000 means halt core1.
- 0 × 00000001 means run core1.
- 0 × 00000002 means halt core2. For the future
- 0 × 00000003 means run core2. For the future
- 0 × 00000004 means halt core3. For the future
- 0 × 00000005 means run core3. For the future
- 0 × 00000006 means halt core0.
- 0 × 00000007 means run core0.

On the reading transaction, possible read data on address lines (0 × 00:0 × 03) are as follows.

- 0 × 80000000 means that this core is running from the reset vector; the message is possible for a default core.
- 0 × 40000000 means that this core is running from the specific task; this message is almost for nondefault cores.
- 0 × 20000000 means halt this core.

3.3 Axi interconnect unit

Axi interconnect unit is designed to take all requests from the two cores to deliver them to the core management unit or to another corresponding peripheral unit (in the future). It was designed separated from the L2_arbiter module to make all interfaces for some peripheral units alone. Figure 8 shows the data flow between the management unit and interconnect unit.

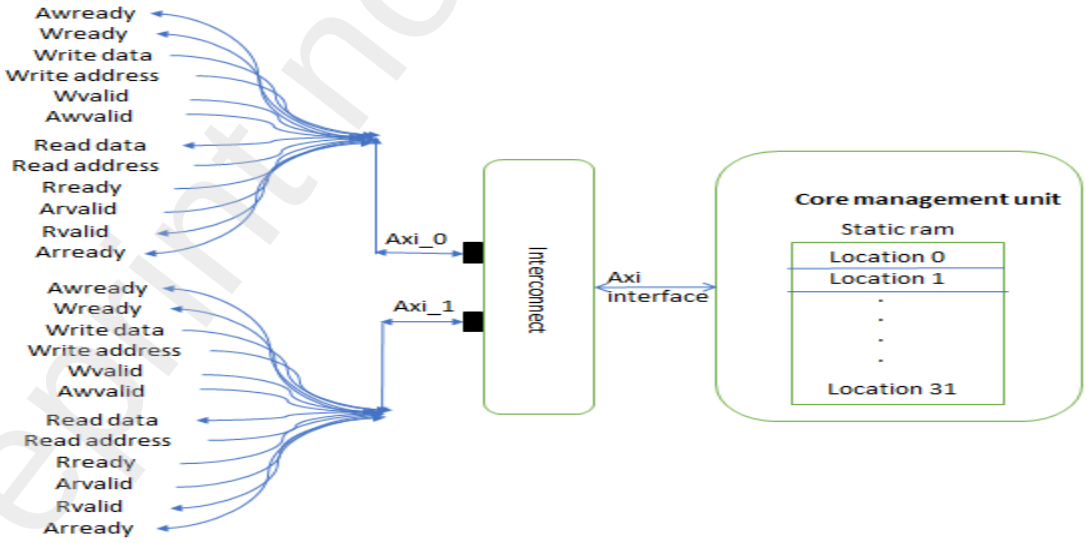


Fig. 8 Core management unit with interconnect configuration

The control circuit of the interconnect unit was implemented to receive and send data sequentially. It was to receive all requests from two cores on two Axi interfaces in parallel and store them until it releases them to specific peripheral device (core management unit) in order.

To store these requests of the two cores, it needs a three-state buffer comprising several locations as many as the number of working cores. It is sized as two locations to store the two requests. Each request is formatted such as Fig. 9.

Rvalid	Arvalid	Raddress	Rdata	Wvalid	Waddress	Wdata
1-bit	1-bit	32-bit	32-bit	1-bit	32-bit	32-bit

Fig. 9 Processor-to-core management request format

Figures (10 and 11), 12 show the control circuit and timing diagram respectively on the interconnect unit for the (R/W) operations.

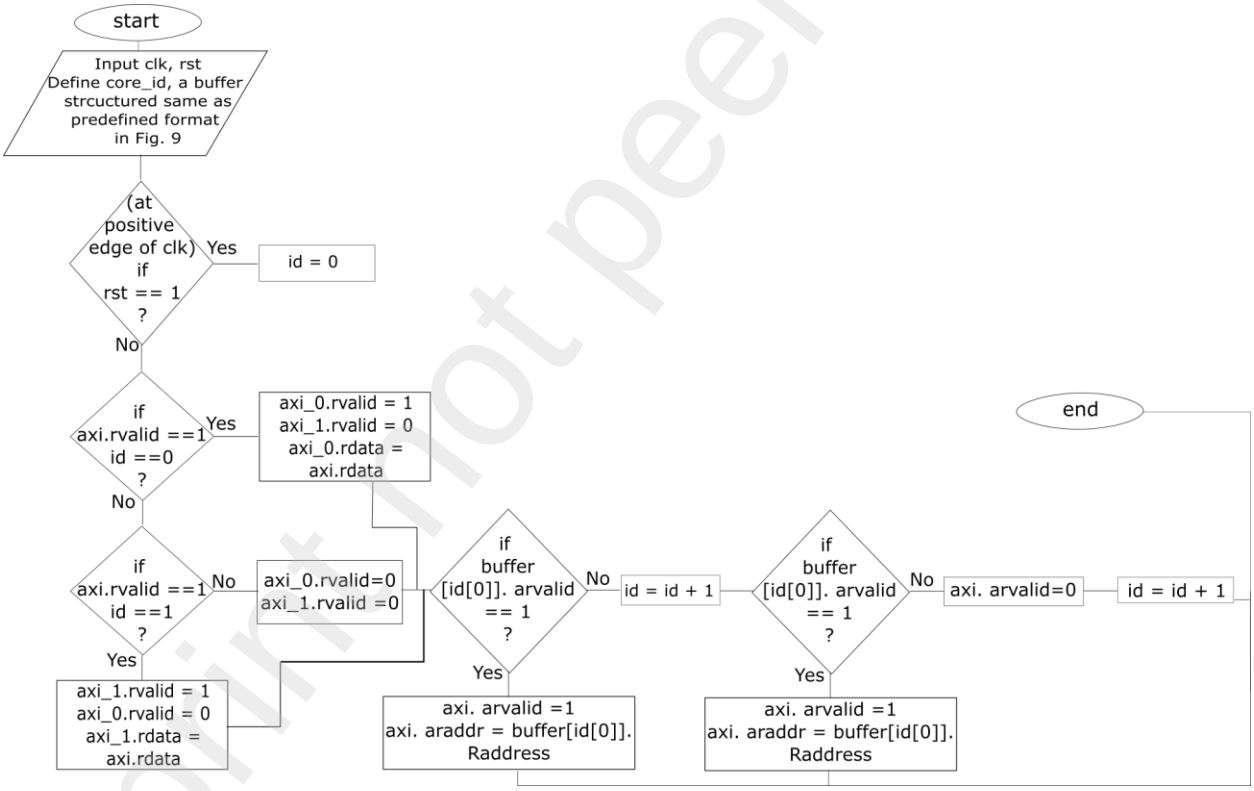


Fig. 10 Flowchart of interconnect module's control circuit on read transaction

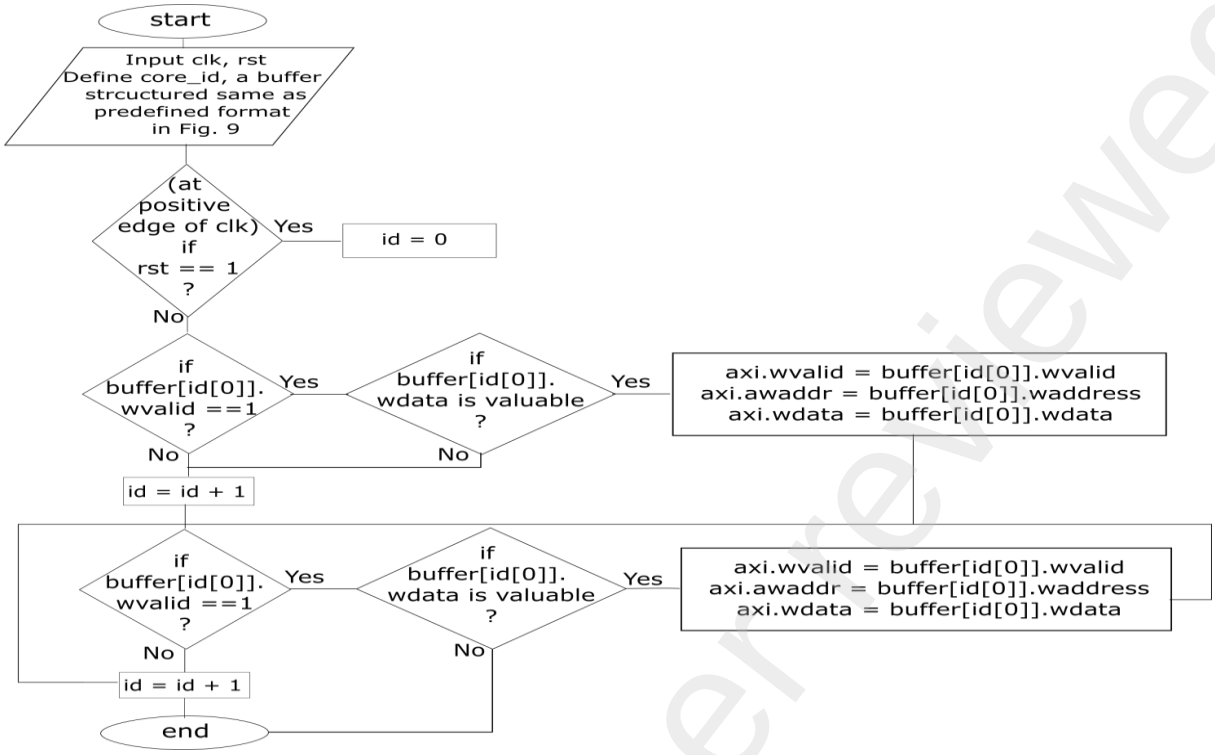


Fig. 11 Flowchart of interconnect module's control circuit on write transaction

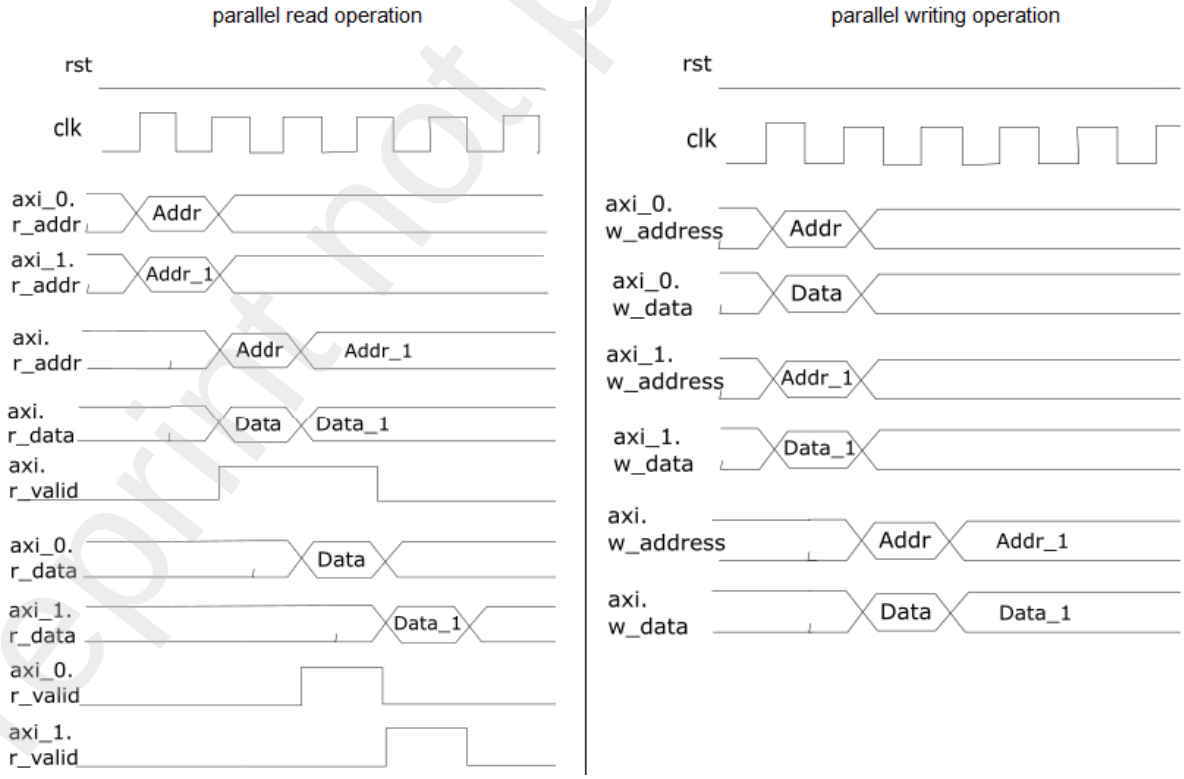


Fig. 12 Timing diagram of write and read operations on interconnect

3.4 Main memory design

It has been designed as dual-port, portA for read operations and portB for write operation. The memory unit has been designed to work synchronously in read and write opertions (Fig. 13). It works to communicate with the two cores basd on AXI protocol. Therefore AXI-HW-Memory module has been designed to control read/write operations based on this protocol as shown in Fig. 14. This module includes an instance of this dual-port memory (Fig. 13) and has been configured to be 4096 lines in size to be synthesizable in hardware verification. Its block of memory is 4-word at a time reading from memory.

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. 13 dual-port memory unit

This module has:

Input rst, **Output** clk

Slave port of AXI interface

Define an instance of dual port memory (portA, portB)// this memory is an instance of Fig. 13

Define FIFO_Read instance of FIFO module. // this FIFO is for storing read address channel signals of AXI protocol

define FIFO_Read_depth by 8

Define FIFO_Write instance if FIFO module. // this FIFO is for storing write address channel signals of AXI protocol

define FIFO_Write_depth by 4

Define int read_burst_count // this integer is for counting read data words for read burst transaction

Always_comb begin

If (read_ burst_count == FIFO_Read.arlen)

FIFO_Read.pop =1;

Else

FIFO_Read.pop = 0;

End

Assign AXI.aready = size of FIFO_Read < FIFO_Read depth;

Always_comb begin

If (AXI.aready & AXI.arvalid) **begin**

FIFO_Read.push = 1;

FIFO_Read.data_in = {Read_address_channel_signals of AXI};

End

Else

```
    FIFO_Read.push = 0;
  End
Assign AXI.rdata = data_out_a;

Always_ff @ (poesdge clk) begin
    If (rst) begin
      AXI.rvalid <= 0;
      AXI.rlast <= 0;
      read_burst_count <= 0;
    End
    Else begin
    If (size of FIFO_Read > 0) begin
      If (AXI.rready) begin
       addr_a <= FIFO_Read.data_out.araddr [15:2] + read_burst_count ;
      AXI.rvalid <= 1;
      AXI.rid <= FIFO_Read.data_out.arid;
      If (FIFO_Read.data_out..arlen == read_burst_count) begin
          AXI.rlast <= 1;
          read_burst_count <= 0;
        End
        Else begin
          read_burst_count <= read_burst_count + 1;
          AXI.rlast <= 0;
         End
        End
      Else begin
        AXI.rvalid <= 0;
        AXI.rlast <= 0;
```

```
        End
      End
      Else begin
          AXI.rvalid <= 0;
          AXI.rlast <= 0;
        End
    End
End
 Always_comb begin
    If (AXI.awvalid & AXI.awready) begin
    FIFO_Write.push = 1;
    FIFO_Write.data_in = {write_address_channel _signals of AXI};
    End
     Else begin
     FIFO_Write.push = 0;
     End
 Assign AXI.awready = size of FIFO_Write < FIFO_Write_depth;
Assign AXI.wready = 1;
Always_comb begin
    If (rst) begin
       AXI.bvalid <= 0
    End
    Else if (AXI.wvalid & AXI.wlast) begin
       AXI.bresp <= 0
       AXI.bvalid <=1
       AXI.bid <= FIFO_Write.data_out.awid
       FIFO_Write.pop <= 1
    End
```

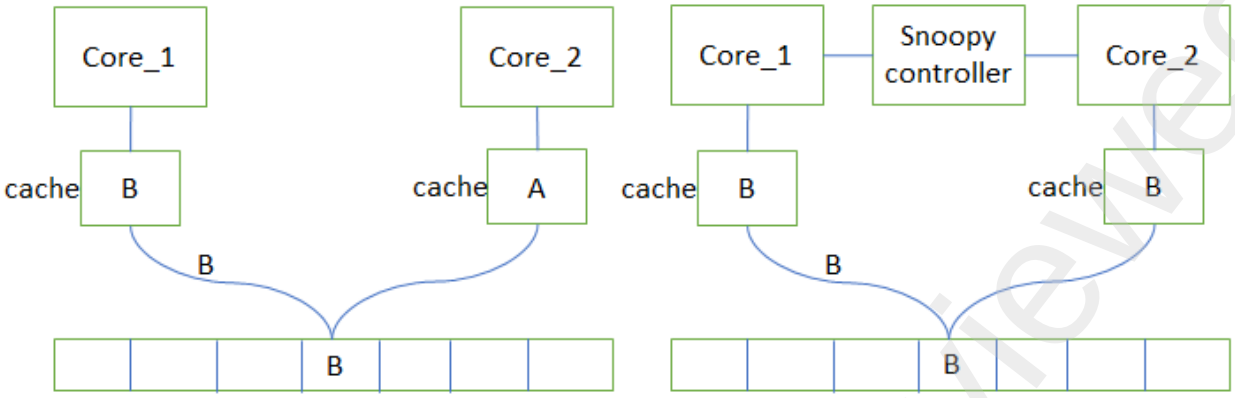
```
    Else if (AXI.bready) begin
        AXI.bvalid <=0;
        FIFO_Write.pop <= 0
    End
  End

 Always_comb begin
   If (AXI.wvalid) begin
        addr_b <= size of FIFO_Write > 0 ? FIFO_Write.data_out.awaddr [15:2] : AXI.awaddr [15:2];
        data_in_b <= size of FIFO_Write > 0 ? 32'h00000000 : AXI.wdata ;
        be_b <= size of FIFO_Write > 0? (4)'(0): AXI.wstrb;
    End
  End
Endmodule
```

4 Cache coherence

Cache coherence is a common problem that occurs at the first step for multicore processor development; it appears because of different loaded data into processor caches from the same main memory location.

Cache coherence problem occurs if one or more cores (core1) load some data from the main memory location and one of them (core0) wants to write different data at the same location. At that moment, the other cores (core1) do not know if there is an update at that location. Therefore, this problem affects the consistency of data coherent.

The design is based on two cores so that it is preferable to solve this problem using simple protocol. Snoopy protocol has been chosen for cache coherence process [17]. Figure 15 shows the result after implementing it in our deisgn.



(a) Before appending snoopy controller **(b)** After appending snoopy controller

Fig. 15 Comparison of cache and main memory coherence on processor before appending snoopy controller and after

Snoopy controller was designed to pass the invalidation request of any working core to the other cores on specific interfaces (Fig. 16). All interfaces contain the same signals. These signals are as follows.

- Wnr indicates that the request to cache is a write operation.
- Valid indicates existing address.
- 32 bit address indicates the snoopy address to which a respective core is writing. These signals are embedded in request invalidation and pass invalidation interfaces Fig. 18 on snoopy controller block.

This snoopy controller unit was designed to take one clock cycle to pass the request of invalidation of the owner core (Fig. 14). In this flow chart, consider that req_inv1indicates the invalidation request generated by core1 and req_inv2 indicates that is by core2 Pass1 indicates pass interface (input invalidation request) sent to core1 and pass2 indicates that is sent to core 2.

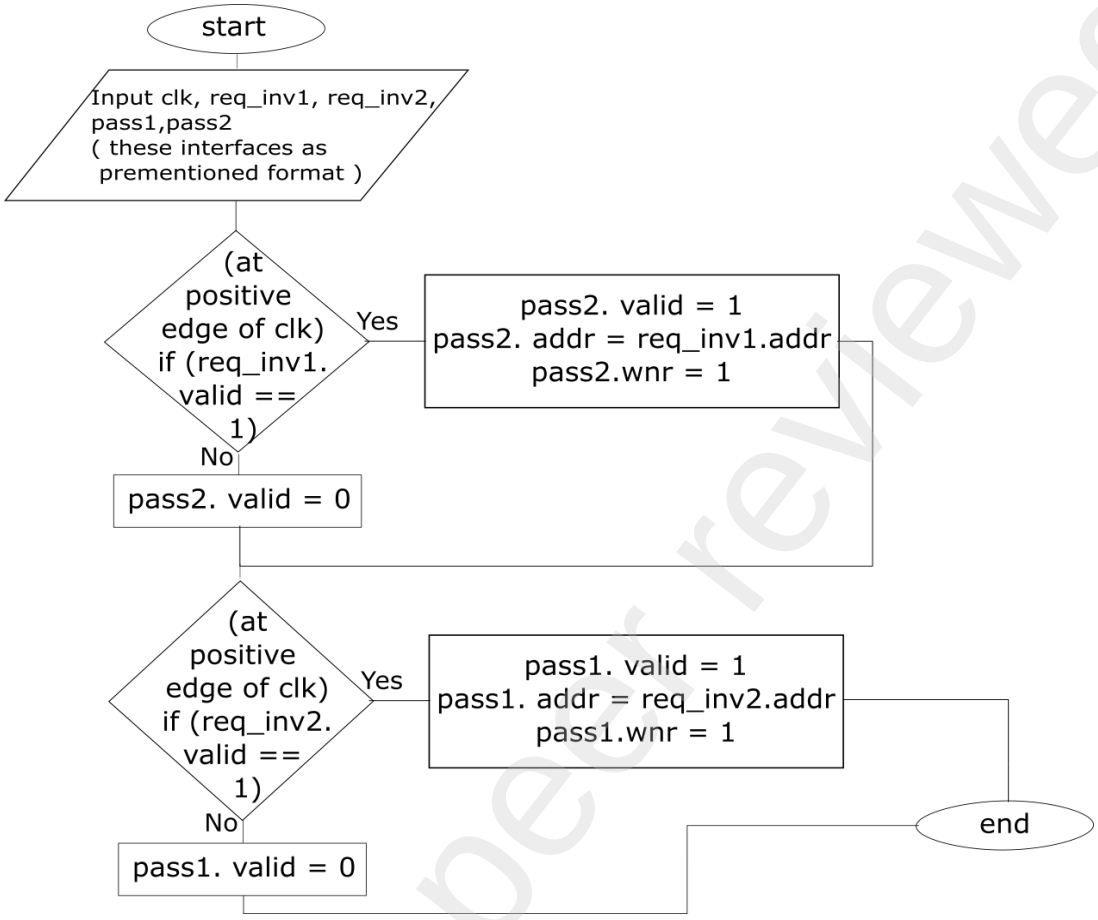


Fig. 16 Flowchart of snoopy controller

Snoopy control circuit has been implemented in the data cache. In case of listener core, it was designed to invalidate a corresponding tag according to the matching of the snoopy address and existing addresses. If this address tag matches with any stored line, this line will be invalidated by clearing the corresponding valid bit. In case of the owner core, another circuit was appended (Fig. 17). Consider in Fig. 17 that: in_instr indicates a new bus cycle instruction for the cache unit. St_instr indicates that this instruction is store. Addr indicates the current bus address. Inv_req indicates the invalidation request interface signals.

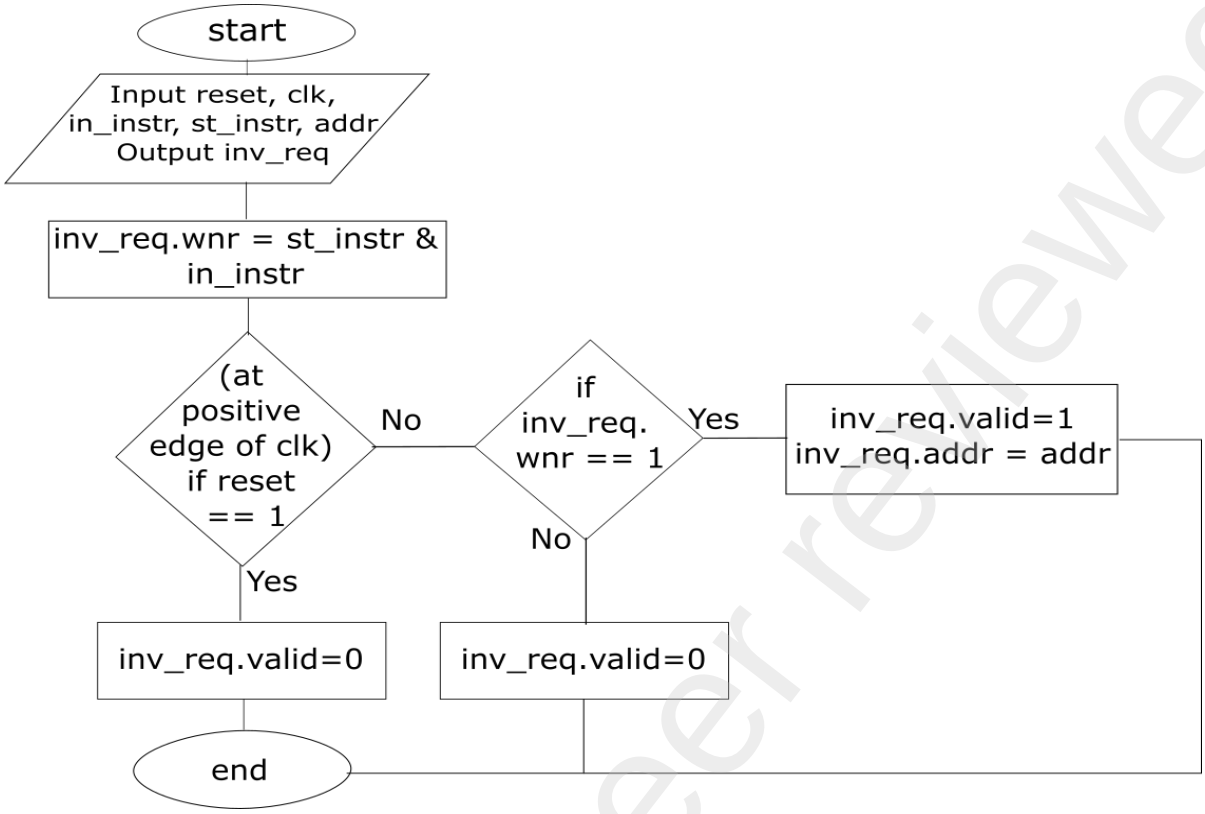


Fig. 17 Flowchart of cache control sub-circuit in case of the owner core

Figure 16 illustrates the data flow when a program is uploaded on the Axi DDR memory, it is fetched by the two cores according to shown steps in Algorithm 1. If any halt/unhalt requests reach the Axi interconnect, it will deliver them to the core management unit in order. In turn, the core management decides to update the output halts signal according to the function of input data. For cache request operations while storing transactions, the owner core always sends an invalidation request to the snoopy controller, which in turn delivers it to the listener. For address matching, the listener will invalidate this tag line.

5 Result and discussion

Dhrystone and CoreMark are the most popular tools for measuring performance of different systems. First, we introduce these tools' scores for several single RvCore processors, and then we show our multicore processor score. As shown at Table 1, Taiga processor's score in single-core architecture is from the biggest scores.

Table 1 Comparison among single RvCore processors on different benchmarks

Core Name	Dhrystone (DMIPS/MHz)	CoreMark/MHz
Taiga	1.65	3.28
Rudolv [18]	0.736 … 1.815 (depending on Dhrystone implementation	1.354
PicoRV32 [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
BlackParrot [21]	--	3.04

After extensive tests, the final stage of the system test is shown in Fig. 18, which represents the testbench unit.

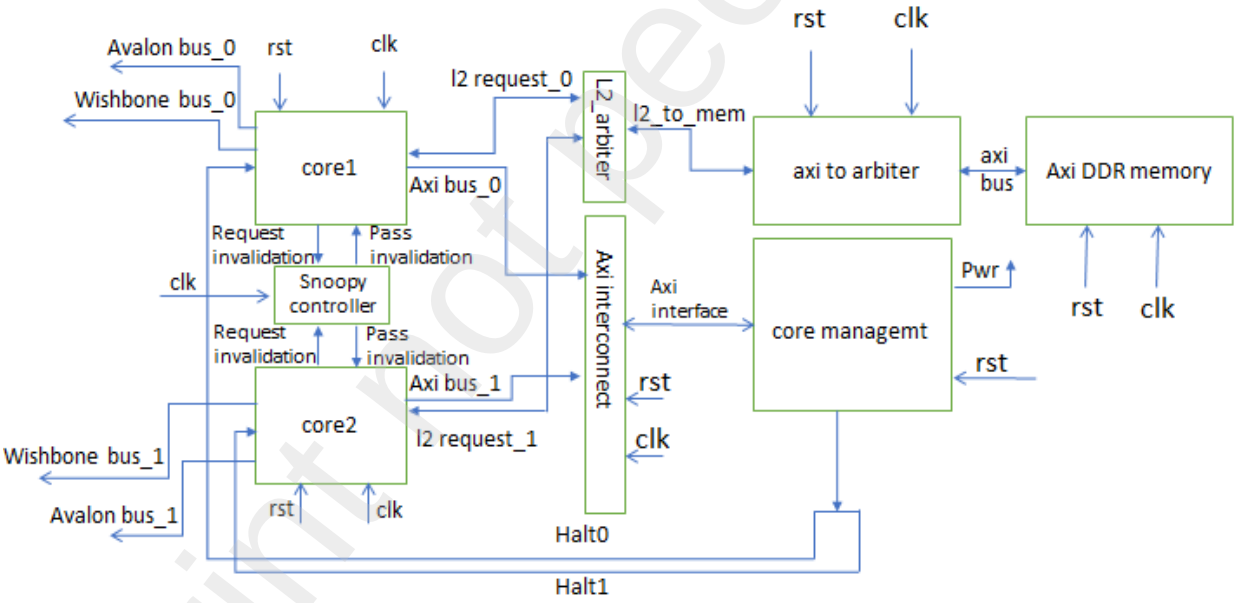


Fig. 18 Testbench module of system design

CoreMark, as a synthetic benchmark tool, is used for testing only single-core processors. In this study, we have modified a bit on the program to work on a dual-core processor on simulation. Vivado 2020.2 has been used to simulate the processor. The operating frequency was 500 MHz (on simulation). On hardware level, it is available to work up to 104 MHz. Table 2 shows a comparison between the results on single- and dual-core in number of clock cycles (k) with

different numbers of iterations. Additionally, Fig. 19 shows the output CoreMark waveforms of the dual-core processor.

Table 2 Coremark result of single and dual core processor

	Single-core	Dual-core	Single-core	Dual-core	Single-core	Dual-core	Single-core	Dual-core
Clock cycles(k)	451.332	-	880.343	800.988	1736.479	1562.066	3448.74	3103.9
Iterations number	1	-	2	2	4	4	8	8

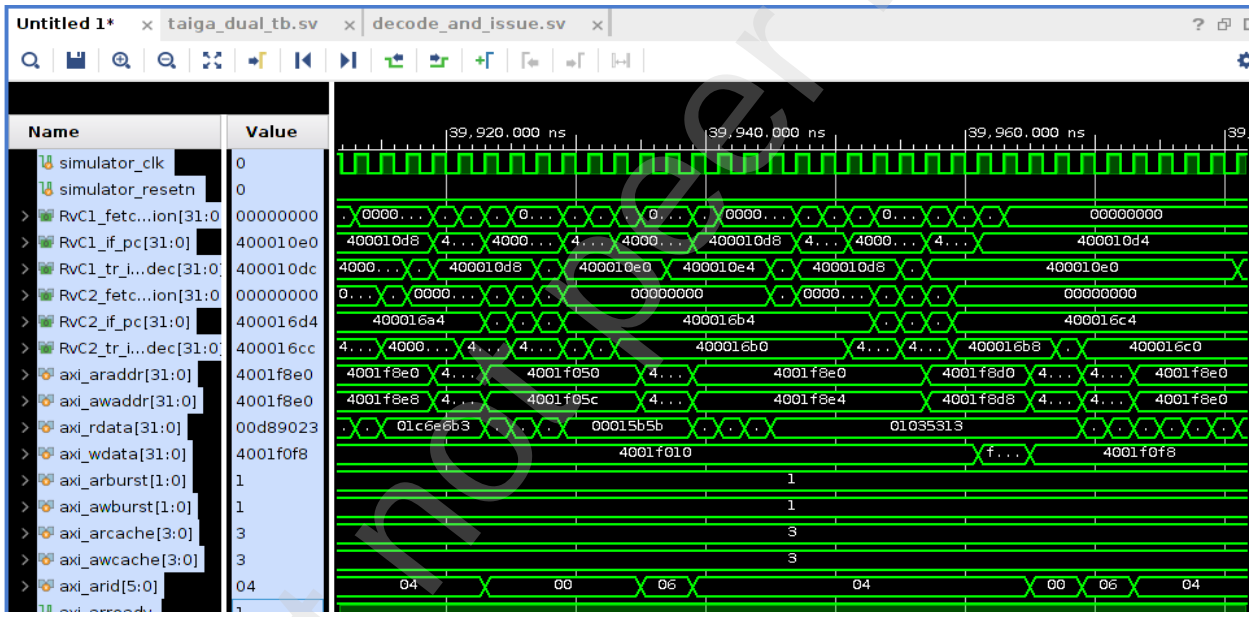


Fig. 19 CoreMark waveform of dual-core processor

According to the results in Table 2, the greater number of iterations, the more stable performance. In case of bigger and complex programs, it guarantees that the performance reaches a stable point since the increase in clock cycles number reduces gradually for bigger programs as shown above.

On zedboard, the operating frequency of dualprocessor that has been used for all benchmarks, is 10 MHz system clock. Several benchmarks have been used for hardware testing process, for example, matrix multiplication, different algorithms of sorting and infinite loop. Table 3 shows all benchmarks, operating frequency and improvement percentage in performance in dual-core processor rather than single-core processor. All resources that have been consumed by this processor are shown in Table 4.

Table 3 Benchmarks' result in clock cycle number

Benchmarks	Number of CLK cycles	Improvement percent	Frequency	Tested on Zedboard
CoreMark (8-iterations)	3104000	10%	500 MHz	
Sorting 100 Numbers (0 … 99)	1998	20%	10 MHz	Done
Summation 1000 Numbers (0 .. 999)	7781	-	50 MHz	Done
Summation 100 Numbers (0 .. 99)	979	21.05%	500 MHz	
	914	-	70 MHz	Done
Summation 10 Numbers (0 .. 9)	637	12.6%	500 MHz	
	277	15.9%	50 MHz	Done
Infinite loop	-	-	10 MHz	Done
Matrix Multiplication	-	-	10 MHz	Done

Table 4 hardware consumed resources

	LUT	FF	BRAM	DSP
Dual-Taiga	6328	3185	28	8

6 Conclusion and Future work

In this study, an open-source multiprocessor (dual-RvCore) design is discussed. Most important aspects in multicore are introduced such as synchronization process and how it is achieved efficiently. Keeping on modularity, an Axi interconnect is designed, separated from the memory system path, for all peripheral devices to interface with processor. Some memory issues are discussed such as cache noncoherency and how it is solved (as well as which protocol is recommended to solve it and how it is implemented). It is shown how the design is ready for

appending custom peripheral devices. The main memory design has been shown in details in pseudocode. To check the harmony of this design, a benchmark tool (CoreMark) is used to measure the performance. Also, several benchmarks have been developed for extensive testing. The performance has been measured in number of clock cycles. It shows how the performance of design is consistent for complex programs.

In the future, we will modify the cache unit and snoopy controller to make the listener core read data from the owner's L1 cache in case of matching not from the main memory to reduce the number of clock cycles consumed for reading updated data from the main memory and those that are for invalidating address tags. Additionally, we can append TC to the cache module [2].

Declarations

Ethical Approval (not applicable)

Code availability all block design files that are developed by this work, are open and available on a GitHub link: https://github.com/demyana123/DeMoJi/tree/main

Funding: No fund has been received

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

Authors' contributions: Demyana Emil and Mohammed Hamdy have worked on the blocks design. Demyana Emil has worked on simulation and testing. Gihan Nagib has worked on paper revision.

Reference

[1] Jiemin Li, Shancong Zhang, Chong Bao: DuckCore: A Fault-Tolerant Processor Core Architecture Based on the RISC-V ISA. Electronics (2022). 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] Eric Matthews and Lesley Shannon: TAIGA: A Configurable RISC-V Soft-Processor Framework for Heterogeneous Computing Systems Research. SEMANTIC SCHOLAR (2017)

[6] Leyva-Santes N.I. et al.: 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 (2019). https://doi.org/10.1007/978-3-030-38043-4_20

[7] H. Jang et al.: Developing a Multicore Platform Utilizing Open RISC-V Cores. In: IEEE Access, vol. 9, pp. 120010-120023. (2021). doi: 10.1109/ACCESS.2021.3108475

[8] Demyana Emil et al.: Dual-RvCore32IMA: Imlementation of a Peripheral device to Manage Operations of Two RvCores. (2022) 4th International Conference on Intelligent Computing, Information and Control Systems

[9] EH2 SweRV RISC-V Core™ design RTL(2020) https://github.com/chipsalliance/Cores-SweRV-EH2

[10] AndesCore™ AX45 (2018) http://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax45/

[11] Taiga RISC-V processor (2019) https://gitlab.com/sfu-rcl/Taiga

[12] Andrew Waterman, Krste Asanović, John Hauser: The RISC_V Instruction Set Manual Volume II: Privileged Architecture. University of California, Berkeley (2021)

[13] ARM Holdings: AMBA AXI and ACE Protocol Specification, ARM IHI 0022H.c (2019)

[14] Intel: Avalon ® Interface Specifications (2020)

[15] Mohandeep Sharma and Dilip Kumar: Design and Synthesis of Wishbone Bus Dataflow Interface Architecture for SoC Integration, IEEE Xplore, pp. 813-818. (2012). doi: 10.1109/INDCON.201206420729

[16] Moini et al.: INKSCAPE (2021) https://inkscape.org/release/1.1.1/windows/

[17] Rasmus Ulfsnes: Design of a Snoop Filter for Snoop Based Cache Coherency Protocols. SEMANTIC SCHOLAR (2013)

[18] RudolV by bobbl-RISC-V processor (2020) https://www.librecores.org/bobbl/rudolv

[19] PicoRV32 -A Size-Optimized RISC-V CPU (2017) https://github.com/cliffordwolf/picorv32

[20] Hoang, Trong Thuc and Duran, Ckristian and Nguyen, Khai Duy and Dang, Tuan Kiet and Nguyen, Quang Nhu Quynh and Than, Phuc Hong and Tran, Xuan Tu and Le, Duc Hung and Tsukamoto, Akira and Suzaki, Kuniyasu, Pham: Cong Kha Low-power High-performance 32-bit RISC-V Microcontroller on 65-nm Silicon-On-Thin-BOX (SOTB). IEICE Electronics Express. (2020) https://doi.org/10.1587/elex.17.20200282

[21] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark

Oskin and Michael Bedford Taylor: BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. IEEE Computer Science (2020)