Dual-RvCore32IMA: Implementation of a Peripheral device to Manage Operations of Two RvCores

[*] Demyana Emil [1], Mohammed Hamdy [2], and Jihan Nagib [3]

Electrical Engineering Department, Electronics and Communications Engineering Section, Faculty of Engineering, Fayoum University, Egypt

[*1]dem11@fayoum.edu.eg

[2]mhm00@fayoum.edu.eg

[3]gna00@fayoum.edu.eg

Abstract. This paper introduces an efficient and simple implementation of management methodology between two RvCores (32-bit) microprocessor to manage the parallel processing to support multi-core processor. In hardware level, a peripheral unit has been developed to manage timing of operations between the two cores without making any disturbance to the default working core in addition to some important future work that will be executed on this device for more enhancement. Under test techniques: unit, integration and system tests, extensive tests and testbenchs have been created for each type of test on the corresponding design to check the correctness of operations. On system test, a dual core task has been developed by C language to check the output correct result. Additionally, the performance of the processor has been shown in number of executed clock cycles.

Keywords: Multiprocessor, synchronization, RISC-V, open-source architecture, management peripheral device, dual-core operations organization, SystemVerilog HDL.

1 Introduction

RISC-V is a microprocessor which refers to “Reduced Instruction Set Computer Version Five”. Its instruction set architecture (ISA) has been open and well known for its high performance since its appearance in 2011 up till now in addition to other properties like: Modular, Flexible, open-source tools and license [1-2].

There are several open-source implementations of the RISC-V ISA [3-4]. Most of these implementations are single core [5-6]. Some implementations support synchronization instructions; hence can be extended to multi-core implementations.

In this paper, a multi-core RISC-V multiprocessor is developed. The available open source Taiga processor [7] has been extended to build a dual core multiprocessor. This processor is introduced as full design multiprocessor.

First, you should have a background on synchronization and parallel processing before going on reading. Synchronization and management between working cores are important to operate in parallel efficiently [8]. Management process is to divide the workloads on the two cores. It was designed based on hardware-scheme and software-scheme. In hardware development, a unit called core management has been developed as a peripheral device to keep the two cores working in parallel. Synchronization process is to keep the two cores updated with new data in case of processes sharing the same memory space.

The coming work is ordered as follows: **section 2** is a preview on Taiga and introduces a simple comparison between it and others, **section 3** discusses the core management unit which controls running or stopping the working cores, **section 4** introduces an example to test intended multi-core operations, **section 5** conclude what has been achieved, **section 6** presents future work.

2 Related Work

More processors under RISC-V ISA have been established in various features. A comparison among set of these RvCores is introduced below taking into consideration the hardware design and performance:

- SweRV-EH2 [9]: is designed for micro-controller which has small tightly coupled memory instead of cache. It is machine mode only.
    - Andes [10]: is multi-core based on RISC-V ISA but it doesn't have data cache neither supports variable latency units. Unfortunately, it isn't open-source.
    - RISC-V Processor [6]: this processor is a 32-bit, 5-stage and fixed-length pipelined processor. It supports Integer, Multiply, and Atomic instructions. It doesn't support Division operations. Wishbone B.3 bus protocol is the only standard bus on-chip. The design achieves peak frequency of 100MHZ on Virtex-7 (XC7VX485tffg1761-2) board.
    - Taiga Processor [11]: is a single RvCore 32-bit system. It
        - Provides atomic instruction.
        - Is ready to support operating system (OS).
        - Supports variable-length pipeline.

- Is optional D-Cache, I-Cache, Multiplication unit, Efficient Division unit.
- Is optional to use 2-way set associative (16kB) or 4-set associative (32kB).
- Has implementation of DMMU, IMMU.
- Provides ITLB and DTLB.
- Provides branch predictor unit.
- Supports privileges like machine, supervisor, and user level [12].
- Ready for interfaces: Axi interface [13], Avalon interface [14] as well as Wishbone interface [15] but the latter two interfaces aren't used.
- Can work up to 123MHZ as peak of frequency on a Xilinx Zynq Z7CZ020 FPGA on a zedboard.

So, from the above comparison and the features of Taiga, Taiga has been selected to be worked on for dual-core processor implementation. It can be extended and configured to support multi-core process. Also, this dual-core is about to be open.

Its open source design from which we have started, is in Fig. 1 and the intended design for dual-core processor is shown in Fig. 2. The two figures show the hardware implementation of each one.

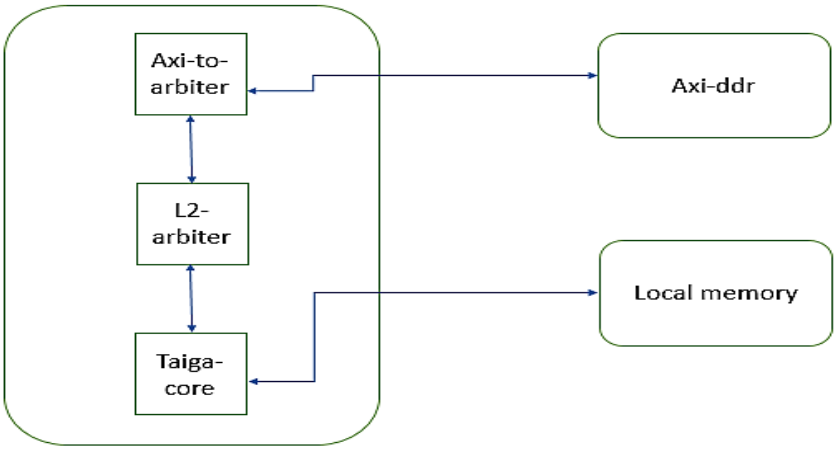

Fig. 1. Taiga Open-Source Single Core Block Diagram

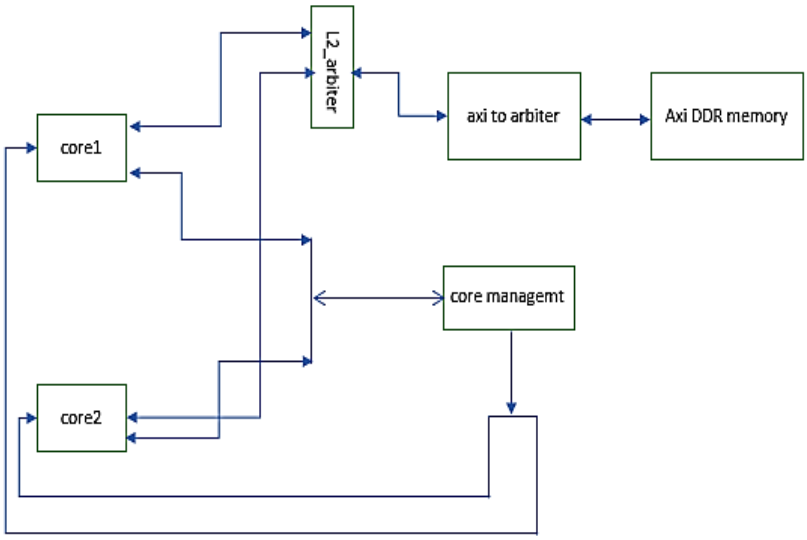

Fig. 2. The intended design of dual-core Taiga processor

As shown in Fig. 2, Core_1 & core_2: is the two Taiga cores. L2_arbiter: is a unit used to receive requests from the two cores to the memory system. Axi to arbiter: is used to convert the arbiter requests to Axi signals under the Axi protocol and vice versa. Core management unit: is a customized peripheral device to manage the two cores operations.

3 Core Management Unit

Core management unit is an external unit from the core. It was designed to organize the timing of concurrent workloads of each core. This unit design has been developed based on software and hardware schemes.

Its interfaces have been developed as shown in Fig. 3. Core management unit has been designed to receive control messages and send read data from/to the two cores sequentially at positive edge clock. According to the type of these messages (written data) it will generate halt signals to each core. Halt signal is used to stop its respective core if it is activated. In case of core halt, the core stops at its current instruction that has been fetched before the halt signal activation and the program counter (PC) keeps its current instruction address value.

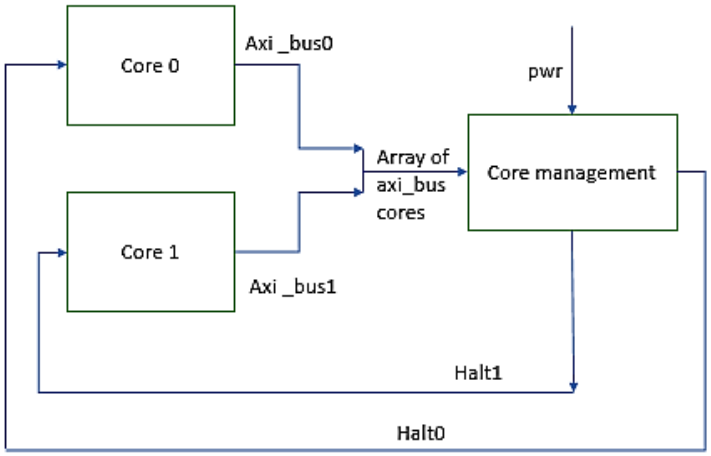

Fig. 3. Subsystem of core management unit interface with dual-core processor

Figure 3 shows the type of interface (Axi interface) between each core and the core management unit. Halt signals have been discussed before. Pwr is an active high input to control it to start/stop working but it was fixed written high by hardware to stop software from controlling it.

On designing this unit, a counter has been developed to make it option for the designer to increase or decrease its speed. This counter has been used to count the number of clocks, after which it becomes available to execute new transaction. Even if it may be available before but it has been designed with this methodology to be able to control its speed in the future i.e. you can use lower frequency clock on this unit to provide consumption power. So, it is available to make that after making some hardware changing on it.

The algorithm of management process is declared in some steps which are obligatory to be used on any software applied on the dual-processor. Before defining these steps, take into consideration some concepts are used in this algorithm:

1- Core2Finished: is a public variable and indicates that if core_2 has finished its related task or not. Its location in the main memory ("Axi DDR Memory" as shown in Fig. 2).
2- Core_selection: the core management unit is a peripheral device that has a small memory (Fig. 4). The "Core_selection"is a bit at an addressed location in the memory of this unit, this bit indicates the state of working cores. It is defined and explained more clearly in Table 1.

Table 1. Core_selection bit cases and their description

Core_selection bit case	Description
'0'	Core_1 is working & core_2 isn't working
'1'	Core_1 and Core_2 are both working

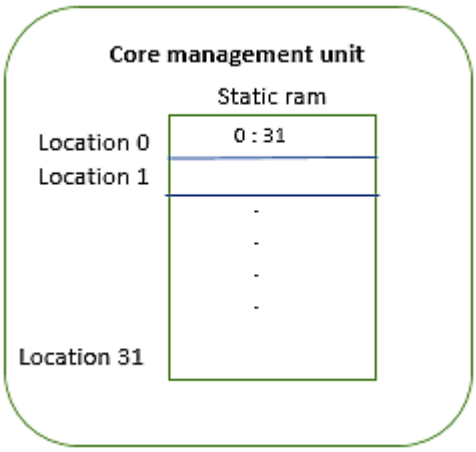

Fig. 4. Core management unit memory

The algorithm steps are:

1- Define Core2Finished as public variable which is initialized by zero.
2- Define a pointer to address of core_selection bit location which is initialized by zero.
3- Then, in main function, check if core_selection bit is zero, so unhalt core_2 and execute the task related by core_1 on core_1.
4- In hardware implementation of core management unit, if Core_2 is unhalted, it updates core_selection bit by one.
5- After Core_1 ends from executing its related task, it waits until core_2 executes all its related operations. It continuously checks the Core2Finished. If Core2Finished is zero, it waits unit it becomes one.
6- When core_selection is set by one, core_2 executes its related task.
7- After core_2 ends from executing its related task, it updates Core2Finished by one then halt itself.
8- When core_2 halt itself, core_selection bit is updated by zero.

9- As soon as Core2Finished is set by one, core_1 starts to work. If there are some instructions that aren't related to core_1 neither core_2, core_1 will execute them.

Core_selection bit changes by hardware according to the coming halt1 and halt2 signals. If halt2 is cleared, core_selection is cleared by zero. If halt2 is set by one, core_selection is set by one.

Each location in the static ram in core management unit as in Fig. 3 includes three variables representing the current case of each working core. They are 32 locations but three locations are only used. Two locations of them contain data for each core and the third one contains important bit called core selection. This core selection control bit is used to define which the last core is running now. Take a look clearly into Table 2.

Table 2. Core management buffer structure format

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

WRV (Work at reset vector bit): represents that this core is default.

WSA (Work at specific address bit)**:** represents that this core is not the default.

Halt bit: controls to stop this core or run if equals 1, the respective core stops at the current pc and stop fetching the next instruction.

3.1 Data Control

On writing transaction to the core management, the written data is only control data which control the current working core to stop or run. This data is sent on address 0x80000000. It is designed to receive control data at this location address.

- On writing operation, if this control data is:
    - 0x00000000: it means halt core1.
    - 0x00000001: run core1.
    - 0x00000002: halt core2. for the future
    - 0x00000003: run core2. for the future
    - 0x00000004: halt core3. for the future
    - 0x00000005: run core3. for the future
    - 0x00000006: halt core0.
    - 0x00000007: run core0.

3.2 Flow Control

All needed control circuits are characterized in write control circuit and read control circuit. It is shown below the write control circuit in Fig. 5 and the read control circuit in Fig. 6 in Pseudocode structure. The design has been established in Systemverilog.

```
For writing operation
Generate
always_comb begin
If (reset happens) begin
clear buffer [0]. halt;
set buffer [0]. work_reset_vector;
loop through the rest locations in the buffer up to the number of connected cores
set each location. halt;
end loop
end
else begin
if (pwr is set and there is a valid data on write data bus and the write address on write address bus equals to predefined address) begin
case (write_data)
  halt core_2: set buffer [1]. halt
  unhalt core_2: clear buffer [1]. halt
  halt core_3: set buffer [2]. Halt
  unhalt core_3: clear buffer [2]. Halt
  halt core_4: set buffer [3]. Halt
  unhalt core_4: clear buffer [3]. Halt
  halt core_1: set buffer [0]. Halt
  unhalt core_1: clear buffer [0]. Halt
endcase
end
Loop through each location in the buffer
    if (buffer [index]. halt is cleared) begin
     buffer [index]. Work_specific_address is set;
     if (index equals to zero)
       buffer [0]. Work_reset_vector is set;
                end
    else begin
    buffer [index]. work_reset_vector is cleared;
    buffer [index]. work_reset_vector is cleared;
      end
    end loop
end
end
  endgenerate
```

Fig. 5. Pseudocode of core management unit's circuit on write control messages

```
Reading operation and core_selection changing
Generate
always_comb begin
  if (pwr is set and there is a valid address on read address bus) begin
     update read data bus by data at buffer location addressed by the read address channel;
     set read valid signal;
end
     else
    clear   read valid signal;
loop through each location at buffer starting from position one up to the number of connected cores
    if (buffer[index]. halt is cleared) begin
    buffer [4] is set;
    end
    else begin
    buffer [4] is cleared;
    end
 end loop
end
endgenerate
```

Fig. 6. Pseudocode of core management on read operation

4 C Code Multi-Core Task for Testing

To check the work of multi-core top design, source C code has been developed to work on the two cores. To be able to upload it on memory, a python tool was used

to convert C code to RV32I[M][A] machine language. This tool was published on the open source GitHub link [1] to complete this task correctly. All these works, installed tools, simulation and result were on Linux 20.04 OS. The design has been written in Systemverilog and the simulation tools have been Verilator [16] and Vivado 2020.2.

An example C code has been developed to test the work on dual-core RISC-V processor. Its task was to add five numbers on core_1 and another five numbers on core_2 taking into consideration the addition result by core_1. The source C code is shown in Fig. 7 and the output result is shown in Fig. 8. The result is shown in Fig. 8 at location 0x1000000c (hexadecimal representation) in hexadecimal representation.

On simulation, the frequency that has been worked on is 500 MHz. The number of clock cycles that have been consumed to finish this operation is 233 clock cycles. This number of clock cycles includes the time required to store the result in the main memory. Each clock cycle is 2 nanoseconds so the total time invested by this program is 466 nanoseconds on simulation.

```c
#include <stdio.h>
unsigned char core1_finished = 0;
unsigned char sum_flag = 0;
int main (void) {

unsigned char *core_select = (unsigned char *) 0x00000004;
unsigned char *halt = (unsigned char*) 0x80000000;
unsigned char *arr = (unsigned char*) 0x40000000;
unsigned char *sum = (unsigned char*) 0x40000030;
unsigned char a,b;
a=0;
b=0;
if (*core_select == 0) {
*sum = 0x0;
for (unsigned char i =0; i< 10 ;i++)
   *(arr + (4*i))=i;
    *halt = 1;
     for (int i = 0; i<5 ;i ++)
     a=a + *(arr +(4*i));
     *sum = *sum +a;
 while (core1_finished == 0) ;
}
else {
for (int i = 5; i<10; i++)
    b=b + *(arr +(4*i));
    *sum = *sum +b;
    core1_finished = 1;
    *halt = 0;
}

if (sum_flag == 0)
sum_flag = 1;

return 0;
}
```

Fig. 7. C code dual core example

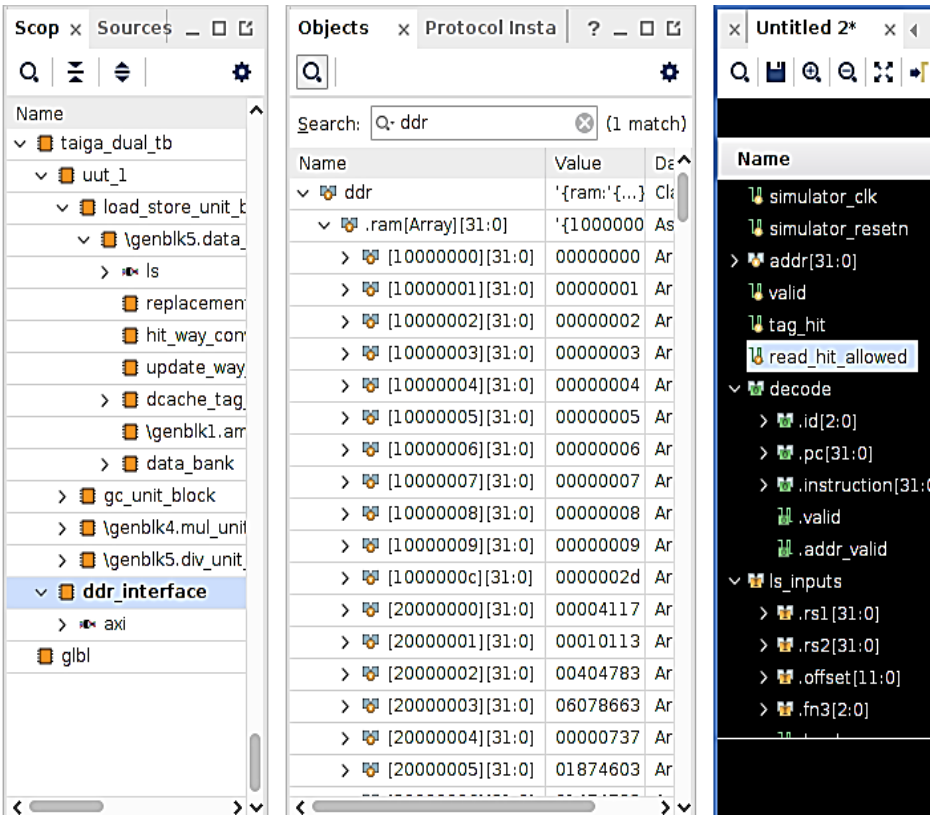

Fig. 8. Output result in L2 memory destination address

The above figure shows the data content and the result in main memory after the simulation ends. Figure 9 shows the waveform of written data bus of the main memory. Axi_wdata (Axi written data) bus in Fig. 9 shows the end result of the program stored in the main memory. The end result is 0000002d (hexadecimal). It has been performed by the two cores following the algorithm of management steps discussed in the core management unit section.

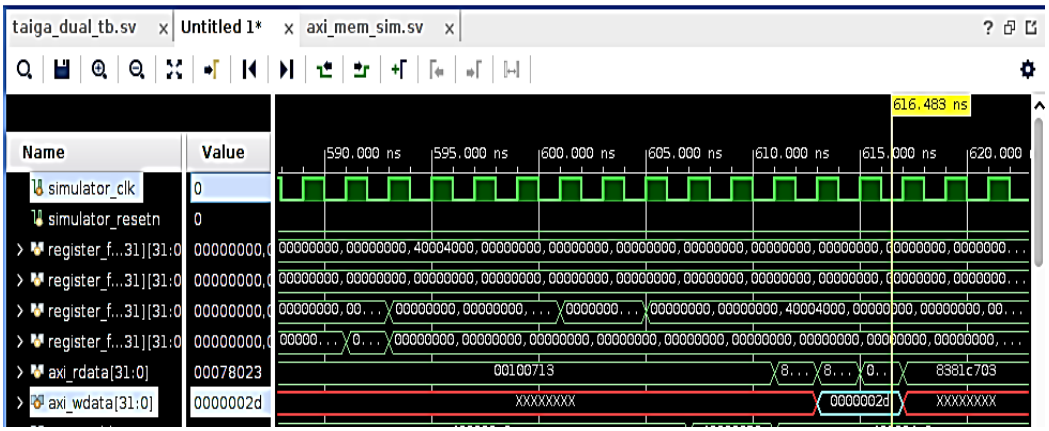

Fig. 9 Waveform of written data in the main memory

5 Conclusion

This work keeps on what others have stopped in multicore processors field (Taiga). All the points related to synchronization process discussed in first section have been achieved. Synchronization has been provided by synchronization peripheral unit which has controlled the operations between the two cores to keep them work synchronously keeping on its advantage of modularity. To check the harmony of these transmission signals, simple application has been developed and applied on the top module. This example was to sum ten numbers, five numbers on core_1 and five numbers on core_2, then core_2 after completing its task it loads the previous value of core_1 result and stores the new value again in the result location.

6 Future Work

Cache coherence problem is a general issue in multi-core processor so it will be taken into consideration to be solved in the futuristic implementation. It will be shown also type of recommended protocol to solve this problem.

Additionally, instead of direct connection between core management unit and dual-core processor, an interconnect network can be developed to be in between. This unit block is for peripheral devices only. It will be established separately from the memory path. Its main function is to deliver some information related to specific peripheral device to the correct device.

More tests also can be used for more enhancement and check if there is any bug in the design. More standard benchmarks can be used like CoreMark and others to check for any bugs in hardware design. Finally, Coremark as standard software benchmark works on single-core systems so it will be modified a bit to divide its task on more than one core to support multi-core systems.

References

1. V. A. Frolov, V. A. Galaktionov, V. V. Sanzharov: Investigation of RISC-V. Programming and Computer Software. Springer Nature (2021)
2. Alexander Dörflinger et al.: ECC Memory for Fault Tolerant RISC-V Processors, Architecture of Computing Systems – ARCS 2020, 2020. Lecture Notes in Computer Science (), vol. 12155. Springer, Cham. (2020) https://doi.org/10.1007/978-3-030-52794-5_4
3. Doerflinger at al.: A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations, Virtual Conference on Computing Frontiers, doi 10.1145/3457388.3458657. (2021)
4. R. Höller et al.: Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation, 8th Mediterranean Conference on Embedded Computing (MECO) IEEE, pp. 1-6, doi: 10.1109/MECO.2019.8760205. (2019)
5. D. K. Dennis et al.: Single cycle RISC-V micro architecture processor and its FPGA prototype, 7th International Symposium on Embedded Computing and System Design (ISED) IEEE, pp. 1-5, doi: 10.1109/ISED.2017.8303926. (2017)
6. A. Birari, P. Birla, K. Varghese and A. Bharadwaj: A RISC-V ISA Compatible Processor IP, 24th International Symposium on VLSI Design and Test (VDAT) IEEE, pp. 1-6, doi: 10.1109/VDAT50263.2020.9190558. (2020)
7. Eric Matthews and Lesley Shannon https://gitlab.com/sfu-rcl/Taiga
8. 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. ISUM 2019. Communications in Computer and Information Science, vol 1151. Springer, Cham. (2019) https://doi.org/10.1007/978-3-030-38043-4_20
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. Eric Matthews and Lesley Shannon: TAIGA: A Configurable RISC-V Soft-Processor Framework for Heterogeneous Computing Systems Research. SEMANTIC SCHOLAR (2017)
12. Andrew Waterman, Krste Asanović, John Hauser: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (2021)
13. ARM, “AMBA AXI and ACE Protocol Specification”, number “ARM IHI 0022H.c”
14. Intel: Avalon ® Interface Specifications (2020)
15. Mohandeep Sharma and Dilip Kumar: Design and Synthesis of Wishbone Bus Dataflow Interface Architecture for SoC Integration (2012)
16. Wilson Snyder: Verilator (2021)