AXI4 High Speed Communication for Microprocessors and RegEx Architectures

Marco La Barbera, Giulio Lotto

June 29, 2024

POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

Abstract

Communication between hardware has always been one of the challenges of systems design, particularly today when the limitation is no longer the architecture's kernel but the vast amounts of data that need to be managed. The aim of this project is to implement a high-speed communication protocol and a caching system for a general-purpose microprocessor, with particular attention to RegEx architectures. Specifically, an existing RegEx architecture named ALVEARE has been taken as a case study. To accomplish this goal, the AXI4 protocol has been initially implemented between ALVEARE and the DRAM, followed by the development of a double cache system as an architectural enhancement.

1 Introduction

Regular Expressions (REs or RegEx) are among the most pervasive yet challenging computational kernels to execute. Indeed, REs matching enables the identification of functional data patterns in heterogeneous fields.

One of the newest RegEx architectures is ALVEARE [1], a full-stack domain-specific framework able to overcome the current state-of-the-art limitations and target scenarios. ALVEARE 10-core is 34x faster than the state of the art and delivers energy efficiency improvements with peaks of 57.88× and 29× against the DPU and the A53. Despite this, ALVEARE's speed is limited by the protocol used to send data to the architecture. ALVEARE receives data through an AXI-Lite interface which is designed for non-complex transmission.

Based on this observation, we decided to equip ALVEARE with the AXI4 [2] (Full) interface. Adding such a protocol allows for the rapid and efficient sharing of large amounts of data in a memory-mapped manner. ALVEARE will enhance its capacity to transmit larger packets to the DRAM in a single burst. Additionally, the current state-of-the-art buffering system will be replaced by a more streamlined and engaging double-cache system.

Enhancing ALVEARE's communication performance expands the potential applicability of our final solution beyond RE architectures to encompass a more generalized scope of use cases. A simplified microprocessor, called ASH (ALU_sys_HDL), has been developed from the ground up to closely mirrors ALVEARE's functionality, wherein the operations serve as the pattern, and the input data serves as the data packet. This microprocessor has been equipped to execute a range of operations using diverse data obtained from the DRAM via both AXI4 and AXI-Lite protocols. ASH communication protocol is capable of reaching up to 250x speed-up compared with a simplified ALVEARE version.

2 Background and Challenges

The state-of-the-art communication between ALVEARE and the SoC processor is a SISD architecture, which sends a single data for a single instruction with an AXI-Lite.

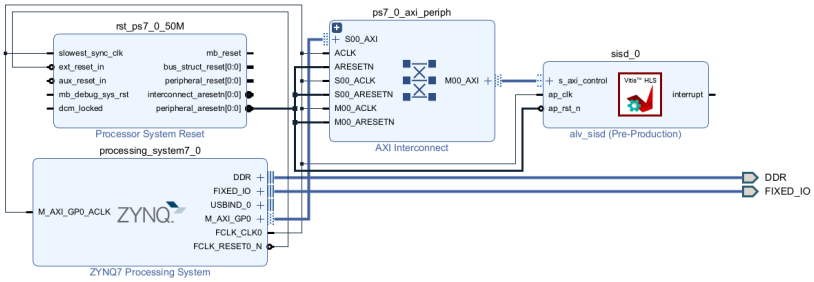

Figure 1: SISD Communication Infrastructure

2.1 ALVEARE working principles

The ALVEARE architecture is designed for Regular Expressions (REs), enabling the analysis of a specified data packet and pattern input to determine the presence and precise location of the identified pattern within the data packet. It has the capability to alter the input data packet while preserving the existing pattern, or it can simply store a new pattern. The choice of the operational mode for ALVEARE depends on a signal provided by the SoC processor, which we will refer to as *selec*. Therefore, the architecture will execute different tasks based on the *selec* signal:

1. Load a new pattern
2. Load new data packets, Execute and Write Back the results
3. Load a new pattern, new data packets, Execute and Write Back the results
4. Reset ALVEARE

In the subsequent section, the integration of the AXI4 interface into ALVEARE will be presented.

2.2 Challenges

The AXI4 and AXI-Lite protocols are both sophisticated communication protocols designed for the transmission and reception of data. In comparison, the AXI-Lite protocol has reduced complexity compared to the AXI4, but operates more slowly. The proposed approach involves utilizing AXI-Lite to transmit information about the DRAM position of the data and instruction. Data are then transmitted through the AXI4. The AXI-Lite will also provide ASH with the *select* signal, conveying the instructions on how it should operate.

ASH will be equipped with a dual cache system, named 'PING' and 'PONG', designed to handle smaller, decoupled data packets, resulting in a substantial increase in overall throughput.

3 Methodology

The first step of the project was the implementation of three AXI4 ports and one AXI-Lite that was initially used to transmit 50 couples of data, a single operation, and the result to an HLS-generated ALU.

The next step was to implement an architecture able to operate on a set of 50 couples of data and 50 different operators. So another AXI4 was used to transmit also the 50 different operations to the ALU. Moreover, it was introduced the so-called *selec* signal to inform the architecture if it has to store a new set of operations or load new data, execute and write back the results in the DRAM, or do both. Finally, we obtain the last infrastructure:

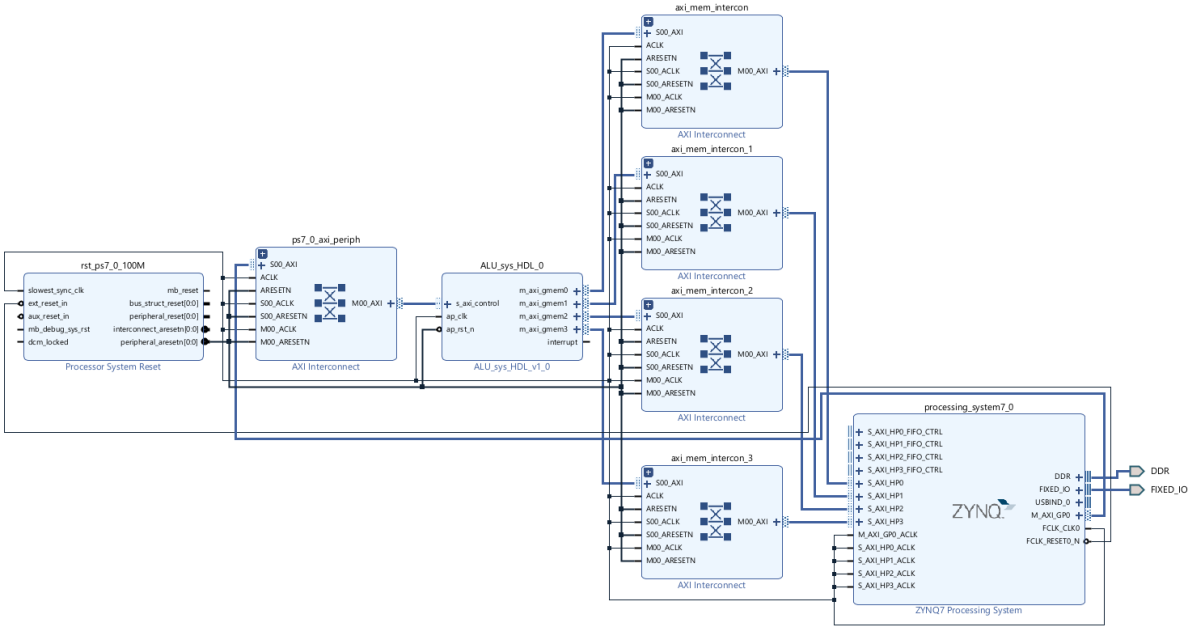

Figure 2: ALU_sys_HDL Communication Infrastructure

3.1 Base HLS Structure

A very simple ALU architecture has been firstly created using HLS to establish a diverse set of functions, in a data-flow way [3], capable of accomplishing a wide range of tasks.

Load and *Store* functions are responsible for transmitting data through the AXI4s into two FIFOs designated for data, and a RAM allocated for operations. A function called *Execution* uses data and operations to perform different calculations (sum, subtraction, multiplication and division). The final *Write-Back* function will write the results back to the DRAM using the AXI4. The specific instructions that will be executed depend on the value of the *selec* variable, which is passed to the top function through an AXI-Lite and evaluated inside it.

The Execute function is responsible for computing calculations and this is where HLS-generated ALU architecture has been replaced with our new and adaptable VHDL architecture. ALVEARE is a VHDL-generated architecture, so it is important to be able to substitute the HLS-generated VHDL with the one written for ALVEARE.

3.2 VHDL replacement

To replicate the behavior of the ALU in VHDL it is necessary to understand the signals generated by HLS for the Execute module.

Algorithm 1 HLS Execute generated Module

```
entity ALU_sys_HDL_op_data_exe_wb_Pipeline_exe is
port (
    ap_clk   : IN STD_LOGIC;
    ap_rst   : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done  : OUT STD_LOGIC;
    ap_idle  : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;

    data_a_dout     : IN STD_LOGIC_VECTOR (31 downto 0);
    data_a_empty_n  : IN STD_LOGIC;
    data_a_read     : OUT STD_LOGIC;

    data_b_dout     : IN STD_LOGIC_VECTOR (31 downto 0);
    data_b_empty_n  : IN STD_LOGIC;
    data_b_read     : OUT STD_LOGIC;

    data_result_din     : OUT STD_LOGIC_VECTOR (31 downto 0);
    data_result_full_n  : IN STD_LOGIC;
    data_result_write   : OUT STD_LOGIC;

    ALU_operation_MEM_address0  : OUT STD_LOGIC_VECTOR (5 downto 0);
    ALU_operation_MEM_ce0       : OUT STD_LOGIC;
    ALU_operation_MEM_q0        : IN STD_LOGIC_VECTOR (31 downto 0)
    );
end;
```

In reference to the AMD Block and Port Level Protocols guide [4], we have gained insight into the management of the *ap_signals*. This entails their regulation through a FSM to provide other state information concerning our upcoming sub-modules, which will be detailed in the subsequent paragraph. As it can be seen, two input FIFOs for the two input operands, one output FIFO for the result, and one RAM for operations have been generated. The realization of the new ALU (Figure 3) has involved its segmentation into three primary sub-modules: Ping Pong cache, Dispatcher, and Functional Units.

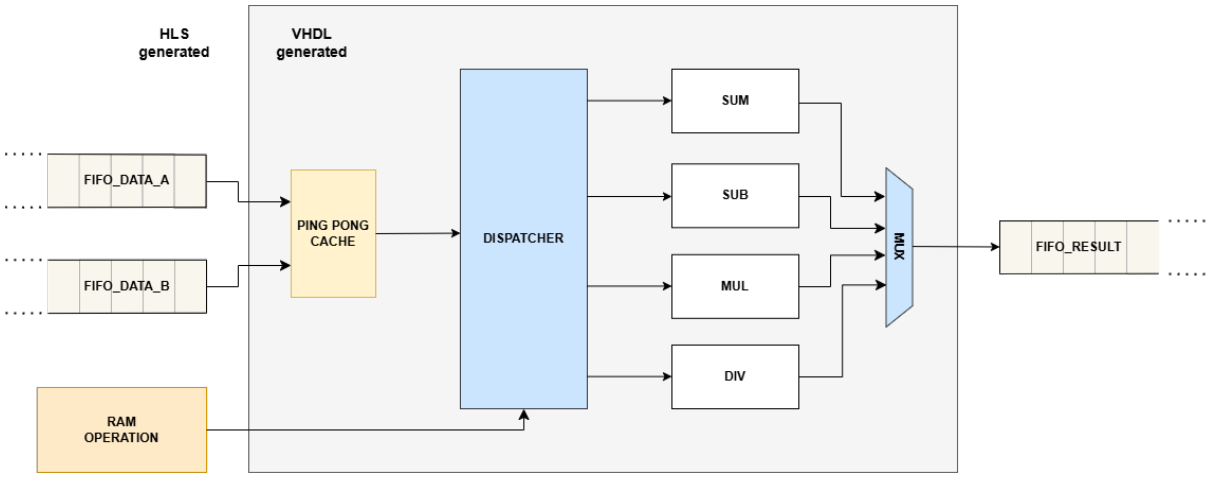

Figure 3: The New Execute Module

3.2.1 Ping Pong Cache

The PPC has been built with a pair of shift registers. At the first cycle, there are no data to send to the Dispatcher (Figure 4), so PONG does not output any data (similarly at the last cycle in Figure 6). When the PPC is at regime (Figure 5), one buffer can receive data while the other can send data. The receive buffer can store data independently if the other buffer is temporarily in the stall phase, this leads to a great improvement in terms of throughput. The PPC also associates the data sent to the Dispatcher with an index that will be linked with an instruction.

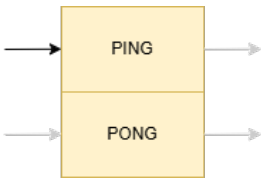

Figure 4: Step 1

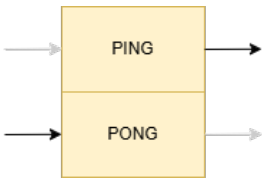

Figure 5: Step 2

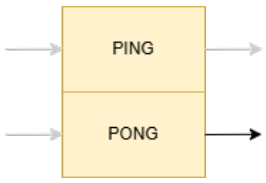

Figure 6: Step 3

3.2.2 Dispatcher

The Dispatcher's goal is to read the RAM value associated with the index of the operation and send its input data to the right FU to compute the right operation.

3.2.3 Functional Units

Four FUs compute four operations: sum, subtraction, multiplication, and division. While the sum and sub are very simple operations the *mul* and the div are more complex to handle in VHDL. This leads to a bigger amount of clock cycles needed to compute these operations. So it is possible that a multiplication which should be done before a sum, ends after it because it has a greater computational cost. To guarantee in-order write-back, the index of the operation is sent inside the FUs with the data and compared with an external counter. So even if the sum ends before the multiplication, result is not written in the output FIFO but it waits until the multiplication (that has a lower index) ends, and writes the result at the output.

3.2.4 Communications between modules

All these complex modules need to communicate efficiently with each other. To do so it was used an AXI-Stream protocol between the various modules. The AXI-Stream is the least complex among the AXI protocols and it is good this task for fast and efficient communication.

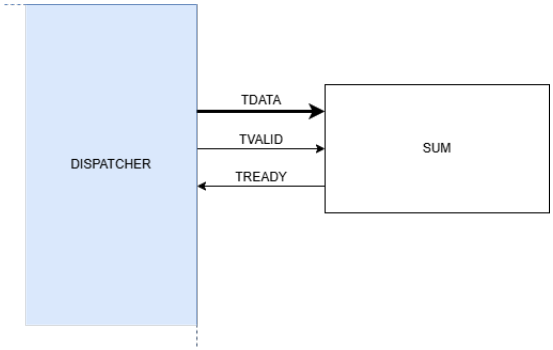

Figure 7: Used AXI Stream protocol

4 Evaluation and Results

We implement ASH in a PYNQ-Z2 board, generate the bitstream with Vivado 2023.2, run it at 100 MHz, and exploit the PYNQ-Z2 environment [5] for Host-DSA communications. ASH has been scaled out from a burst of 50 data to a burst of 400 data, as it is the maximum number fitting our FPGA resources.

As previously said the state-of-the-art for ALVEARE communications consists of an AXI-Lite transmission SISD architectures. The Speed-Up values have been computed for a burst of 50, 100, 200, and 400 data. The results show that by increasing the burst size, the Speed-Up becomes bigger: 6.75x, 8.01x, 11.34x, 13.25x with the Python but more importantly 109.65x, 151.76x, 239.26x, and 256.57x with the state-of-the-art ALVEARE's communication system.

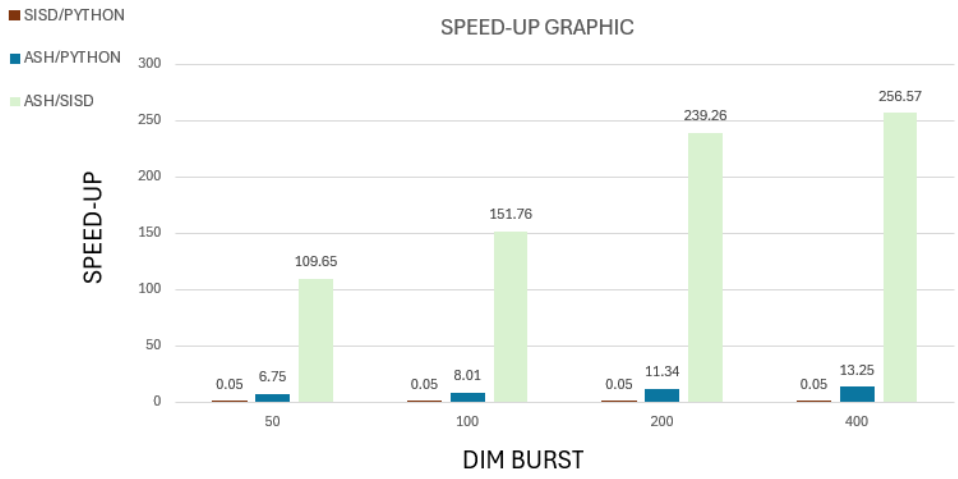

Figure 8: Graphic that shows speed-up improvements

On the other hand, to accommodate a larger volume of data within a single transmission, we are resorting to a substantially increased utilization of LUTs as LUT RAM. This has resulted in a consistent escalation of hardware usage (Fig. 9 to Fig. 12) and power consumption.

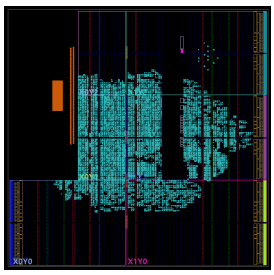

Figure 9: 50

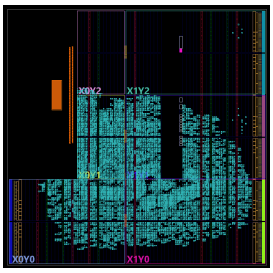

Figure 10: 100

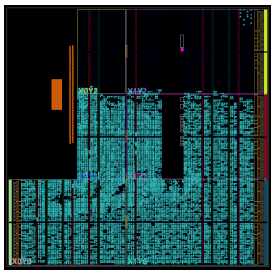

Figure 11: 200

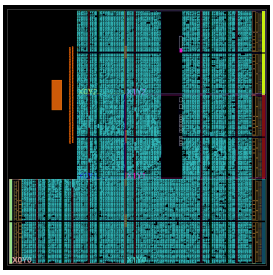

Figure 12: 400

5 Conclusions and Future Works

This paper presented a scenario where kernel limitations are no longer the architectures but the vast amount of data transmission that need to be managed. A high-speed communication protocol and a caching system for general-purpose microprocessors have been implemented, taking ALVEARE as a case study. For the next step of the project, we envision inserting a new component that allows a burst of a bigger set of data separated into more than two groups to speed-up the computations without filling the FPGA's LUT utilization.

References

[1] Filippo Carloni, Davide Conficconi, and Marco D Santambrogio. *ALVEARE: a Domain-Specific Framework for Regular Expressions*. 2024.

[2] arm. *AMBA AXI Protocol Specification*. 2023. URL: https://developer.arm.com/documentation/ihi0022/k/?lang=en.

[3] AMD. *Dataflow Debug and Optimization*. 2023. URL: https://docs.amd.com/r/en-US/Vitis-Tutorials-Hardware-Acceleration/Dataflow-Debug-and-Optimization.

[4] AMD. *Block and Port Level Protocols*. 2018. URL: https://home.mit.bme.hu/~szanto/education/vimima15/heterogen_vivado_hls_5.pdf.

[5] Xilinx. *PYNQ-Z2 Environment Setup Guide*. 2018. URL: https://pynq.readthedocs.io/en/v2.5.1/getting_started/pynq_z2_setup.html.