High-Speed System Communication and Caching Enhancement for RegEx Matching

Filippo Carloni, Marco La Barbera, Giulio Lotto

June 30, 2024

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. This architecture receives 16 kB of data in input through an AXI-Lite interface. To increase the transmission speed, the idea is to implement an AXI4 protocol between ALVEARE (the general-purpose microprocessor) 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 ranging from personalized medicine to computer security.

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 data chunks to the DRAM in a single burst. Additionally, a new streamlined and engaging **double-cache system** will be inserted.

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 as will be explained later, at the end of the 3.1. For now, it is sufficient to know that a simpler microprocessor, called ASH (ALU_sys_HDL), has been developed from the ground up to closely mirror ALVEARE's functionality, wherein the operations serve as the pattern, and the input data serves as the data chunk. This microprocessor has been equipped to execute a range of operations (just like an ALU) using diverse data obtained from the DRAM via both AXI4 and AXI-Lite. ASH communication protocol is capable of reaching up to **250x speed-up** compared with an ALU that uses a SISD architecture (just like the one used by ALVEARE) to transmit data (the speed-up consideration will be discussed in the 4).

2 Background and Challenges

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

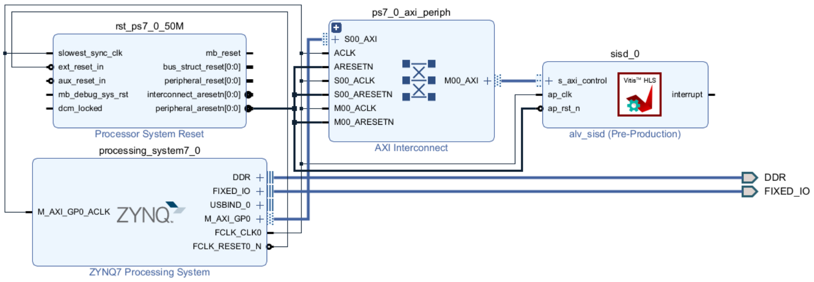


Figure 1: SISD Communication Infrastructure

2.1 ALVEARE working principles

The ALVEARE architecture is designed for Regular Expressions (REs), enabling the analysis of a specified data chunk and pattern input to determine the presence and precise location of the identified pattern. It can acquire the input data chunk while preserving the existing pattern, or it can simply store a new pattern. After one of these steps, it proceeds to the computation. 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 chunks, Execute, Write Back the results;
3. Load a new pattern, load new data chunks, Execute, Write Back the results;
4. Reset ALVEARE

ASH will propose again the operational mode of ALVEARE. The only difference (other than the faster communication protocol of ASH) will be the Execute since it will accomplish a different task(ALU for ASH, RE computation for 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 position of the data and instruction that are located inside the DRAM. 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 chunks, resulting in a substantial increase in overall throughput.

3 Methodology

The first step of the project for the creation of ASH was the implementation of three AXI4 ports, one AXI-Lite port, and an ALU. The idea was to transmit N couples of data using two AXI4 ports and a single operation using the AXI-Lite port to the ALU, compute the N results, and send them to the DRAM through an AXI4 port. We decided to take $N = 50$ in this case, just as we did for the others.

The next step was to implement an architecture able to operate on a set of 50 couples of data and 50 different operators. So the AXI-Lite was substituted with another AXI4 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. The *selec* signal was trasmitted through an AXI-Lite port. Finally, we obtain the last infrastructure:

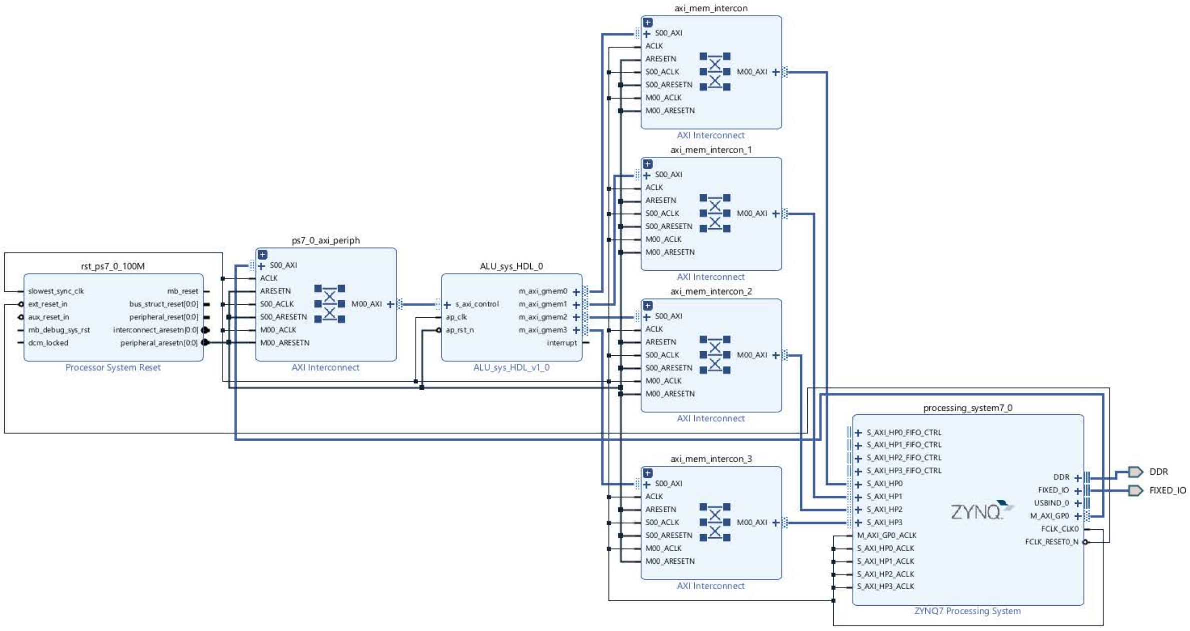


Figure 2: ALU_sys_HDL Communication Infrastructure

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 approach[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 *Execute* uses data in the FIFOs and operations in the RAM to perform different calculations (sum, subtraction, multiplication and division) and put the results in another FIFO;
- The final *Write-Back* function will write the results stored in the last FIFO in 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. In fact, there is a switch case construct that evaluates the value of *selec* and performs different functions depending on its value:

- if $selec = 0$ then the architecture uses a *Load* to load the values of the operations in a vector, and a *Store* to store them in the RAM;
- if $selec = 1$ then the architecture uses two *Load* to load the values of the couple of data in input in two vectors and two *Store* to store them in two FIFOs, then in proceed with the *Execute* and the $Write - Back$;
- if $selec = 2$ then the architecture do what it does with $selec = 0$ and the what it does with $selec = 1$;
- if $selec =$ ”Any other number” is the reset condition, so all the FIFOs, the RAM, and the vectors get cleared;

It was already explained that except for the communication protocol, the *Execute* function is the only difference in the operational mode between ASH and ALVEARE. So, since this project aims to speed-up ALVEARE (that is a Vivado VHDL-generated architecture) it is important to be able to substitute the HDL entity for *Execute* generated by HLS with an *Execute* HDL entity generated in Vivado. In this case, it will be shown that the entity generated in Vivado accomplishes the same task as the *Execute* HDL entity generated in HLS (in 3.2, some improvements will be shown). But it is important to underline that the *Execute* HDL entity generated in Vivado could be created to accomplish a different task (ALVEARE *Execute* or a general-purpose microprocessor).

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, HLS generated a VHDL entity for the *Execute* with two input FIFOs (lines 10 to 16) for the two input operands, one output FIFO (lines 18 to 20) for the result, and one RAM (lines 22 to 24) for operations. The realization of the new ALU (Figure 3) has involved its segmentation into three primary sub-modules: Ping Pong cache, Dispatcher, and Functional Units.

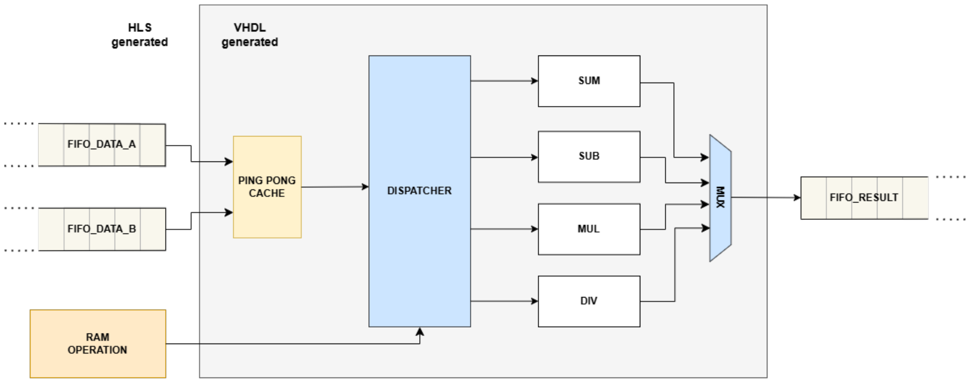


Figure 3: The New Execute Module

3.2.1 Ping Pong Cache

The PPC has been built with a pair of shift registers (if the total number of data couples that can be passed to the PPC is N, then each shift register can contain up to $N/2$ couples values). For simplicity let's call them PING and PONG.
The working principle of the PPC is easy to understand: when PING receives, Pong sends, and vice-versa. Let's suppose that at the beginning the PPC in empty. In the first cycle (Figure 4), PING receives the data from the input FIFOs. There are no data inside PONG (since as we just said at the beginning the PPC is empty), so PONG does not send any data to the Dispatcher. Then, when PING finishes receiving data, it starts to send them to the Dispatcher. In the meantime, PONG will start receiving data from the input FIFOs (Figure 5).

When PING finishes sending data and PONG finishing receive data, PONG will start to send the stored data at the Dispatcher (this time it has data to send) and PING will be ready to receive data from the input FIFOs but they will be empty since they already send all the data inside them in the two previous steps (shift registers length is half the length of THE FIFOs) (Figure 6).

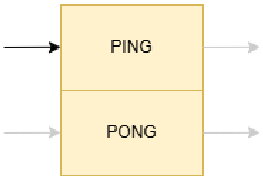


Figure 4: Step 1

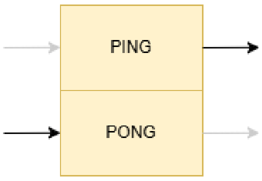


Figure 5: Step 2

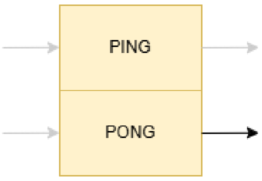


Figure 6: Step 3

The data sent to the dispatcher are associated with a counter that goes from 0 to $N-1$ depending on the sending order: the first couple will have an index equal to 0 and the last one index equal to 49. The reason of this index assignation will be explained in the following subsection.

3.2.2 Dispatcher

The Dispatcher's goal is to send the couple of data arriving from the PPC to the right FU. To do so, the dispatcher reads the RAM address corresponding to the index that the PPC sends with the data. Once the RAM is read the data can be sent to the right FU but only if it is ready to receive them: it may happen that a FU is still computing a result and it is not ready to accept new data in input, in this case, the Dispatcher stalls until the FU is ready. The Dispatcher will also provide the FU with the index used to read the RAM. The reason will be explained in the next subsection.

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 difficult to handle in VHDL since they are more computationally complex. This leads to a bigger amount of clock cycles needed to compute mul (10 cc) and div (50 cc). 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.

The external counter counts how many times a value has been written in the output FIFO and it is updated inside the ASH but outside the sub-entities (PPC, Dispatcher, FUs). Only if the counter inside the FU and the external counter have the same value (and the FU has finished its computation) the FU can write the result in the output FIFO and accept a new couple of data at the input. In all the other cases the result cannot be written at the output.

So even if the sum ends before the multiplication, the result is not written in the output FIFO but it waits until the multiplication (that has a lower index because it started before) ends, and writes the result at the output.

Essentially the operations are stored in the RAM with an order and even if the results of an operation with a greater index can be ready before the result of an operation with a smaller index, they will be written in the output FIFOs in order.

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 created inside the Execute entity. The AXI-Stream is the least complex among the AXI protocols and it is good this task for fast and efficient communication.

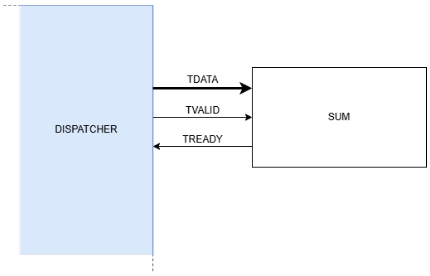
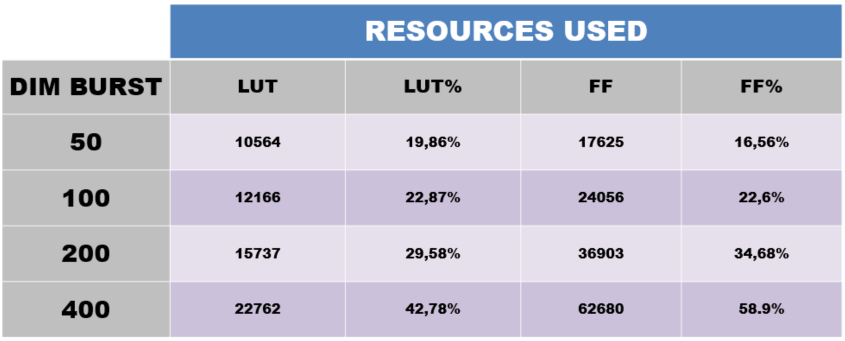


Figure 7: Used AXI Stream protocol

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 (the number of LUT for an 800 data burst is approximately 116.000 while the maximum number of LUT of the boar used is 106.000).

As previously said ALVEARE communication consists of an AXI-Lite transmission SISD architectures. The speed-up values with the new ASH communication system have been computed for a burst of 50, 100, 200, and 400 data. It has been calculated as the ratio between two values of time for each different burst size: the time that an ALU needs to receive the data, compute the results, and write them back in the DRAM with the old SISD architecture communication system over the time needed with the new ASH communication system. This calculation has been performed several times and an average has been taken. The results show that by increasing the burst size, the speed-up becomes bigger: 109.65x, 151.76x, 239.26x, and 256.57x with the current ALVEARE communication system.



	RESOURCES USED			
DIM BURST	LUT	LUT%	FF	FF%
50	10564	19,86%	17625	16,56%
100	12166	22,87%	24056	22,6%
200	15737	29,58%	36903	34,68%
400	22762	42,78%	62680	58.9%

Figure 8: Resources used in the PYNQZ2 for design implementation

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.

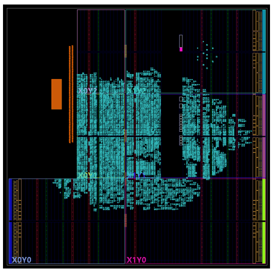


Figure 9: 50

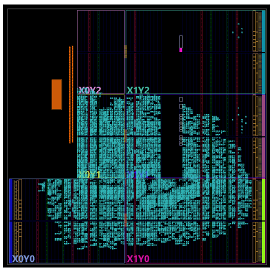


Figure 10: 100

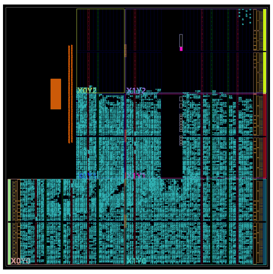


Figure 11: 200

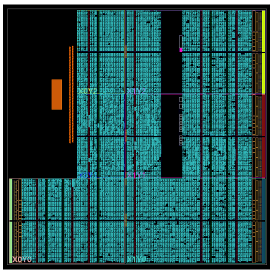


Figure 12: 400

5 Conclusions and Future Works

This report presented a scenario where kernel limitations are no longer the architectures but the vast amount of data transmission that needs 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 shift registers (so there will be an upgrade of the PPC) 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.