FPGA implementation for dynamically scheduled high-level synthesis generated circuits

Laboratoire d'Architecture des Processeurs - Report

Professor : Paolo Ienne Supervisor : Andrea Guerrieri Student : Antoine De Gendt

Winter semester 2020

1. Abstract

Laboratoire d'Architecture des Processeurs
IN-F Ecublens
CH-1015 Lausanne
http://lap.epfl.ch

Semester Project
Winter 2020

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

FPGA implementation for dynamically scheduled high-level synthesis generated circuits

De Gendt Antoine **IC**

Introduction

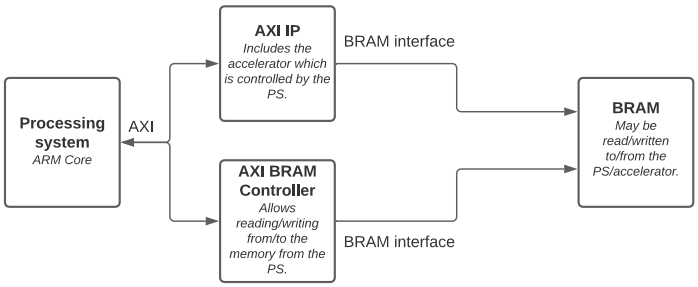
Dynamatic is an HLS tool developed at EPFL. HLS stands for high level synthesis and enables developers to transform high level code such as C or C++ into hardware description languages such as VHDL or Verilog. More precisely, these tools transform a C/C++ function. Dynamatic generates dynamically scheduled circuits whereas most available HLS tools generate statically scheduled circuits, which achieve a lower throughput. These pieces of software are particularly useful to create accelerators for boards with a processing system and programmable logic. Their goal is to offload some work an application has to perform to the programmable logic, which, when arranged specifically for one task, can greatly improve performance. The abbreviation PS will be used for processing system and PL will be used for programmable logic throughout the report.

Goals

VivadoHLS, even though it generates statically scheduled circuits, makes it very easy for anybody with little knowledge about FPGAs to create accelerators for a variety of applications. It integrates very well with Vivado which is an Electronic design automation tool and is used to implement the design on the FPGA. The goal of this project is to find out how to make a functioning accelerator out of the Dynamatic output, that is to say, how to integrate it in a design where it can communicate with the processing system. Once such a design is found, we need to automate its generation to make the use of Dynamatic as easy as possible, requiring minimum input from the user. In fewer words, matching the flow used when working with VivadoHLS. To be clear, one would use Dynamatic to generate the HDL and then the automation software to generate the design.

Summary

Firstly, an accelerator only has a purpose if it is able to communicate with a processing system, the place where the core of the application will run. So, the first major step of the project was to find a solution to establish communication. This is done using the AXI protocol. The IP created by Dynamatic from the C/C++ source is wrapped inside an AXI IP which enables the accelerator to communicate with the CPU on the board. It was also necessary to incorporate memories in the design which can interface with the accelerator and the processing system. Indeed, you may want to be able to initialize the memory from the processing system, start the computation and then get the results back by reading the memory.
Then, once a template for the design was studied and tested, a tool was developed to automate the process for any program that would work with Dynamatic.



Future considerations

Unfortunately, due to a lack of time, the design has not been tested with the Amazon F1.
Also, the automation tool only fully supports two FPGAs and one might want to extend the set of FPGAs supported.

Finally, even though the tool supports arrays of variable widths, arguments and return values can only be 32 bits for now.

Contents

2. The design

The design can be decomposed into two relatively distinct parts. First, the AXI IP itself, which wraps the accelerator and allows it to communicate with the processing system. Then, the rest of the design, with the memories and the processing system.

2.1 The AXI IP

The AXI IP is implemented with two VHDL files. The top file will instantiate the accelerator and connect it with the other file which actually implements the AXI protocol. It will be referred to as the AXI file.

2.1.1 The accelerator

We will now describe the ports of the accelerator we are interested in :

- rst (in) : resets the accelerator when '1'
- start_valid (in) : starts the accelerator when '1'
- start_ready (out) : when '1', indicates the accelerator is ready to start
- end_out (out) : return value
- end_valid (out) : when '1' indicates the value of end_out is valid
- end_ready (in) : set to '1' by the PS when ready to read the return value
- For each array :
  - {array_name}_address0 (out) : write address
  - {array_name}_ce0 (out) : enables the BRAM when the accelerator wants to write
  - {array_name}_we0 (out) : enables writing in the BRAM
  - {array_name}_dout0 (out) : write data
  - {array_name}_din0 (in) : read data (not used)
  - The same set of ports indexed with a 1 exists for reading where din is used and dout is not used.
- For each argument :
  - {argument_name}_din (in) : argument value written by the PS

2.1.2 The top file

This file instantiates the accelerator and creates signals for the control/status ports and the arguments. It also exposes the memory interface of the accelerator for future connections with the BRAMs. The signals created are connected to both the AXI file and the accelerator. The AXI file will allow the processing system to read the outputs of the accelerator and write to its inputs.

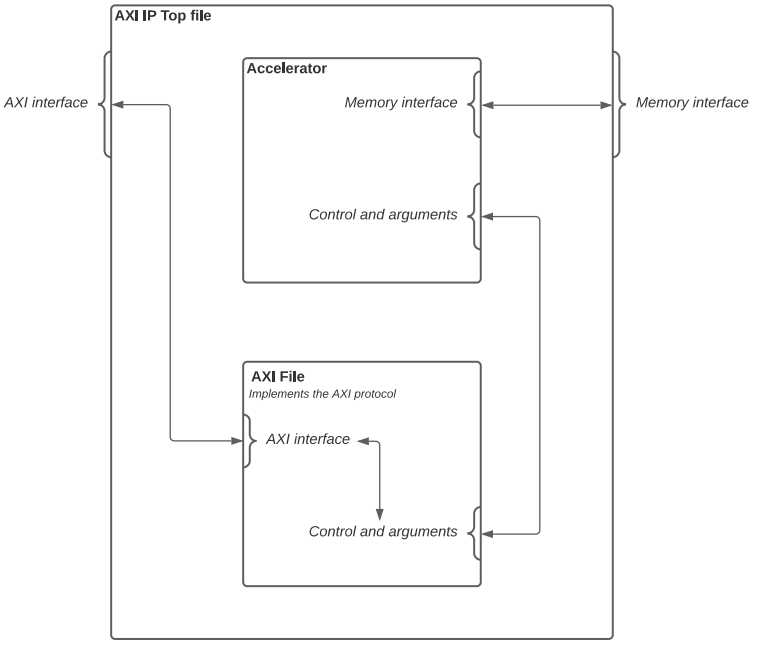


Figure 1. Block diagram of the AXI IP

As shown in Figure 1, the memory ports of the accelerator are only passed through the AXI IP. Also, the control/status and arguments signals will go back and forth between the processing system and the accelerator. The AXI file allows this communication through the AXI protocol.

2.1.3 The AXI file

This VDHL file contains slave registers through which we will communicate with the CPU. We will use these registers in two different ways. A set of register will be used for read-only signals and another one will be used for write-only signals. Here is the mapping between the accelerator ports and the slave registers :

	0	1	2	3...31
slvreg_0 (W)	start_valid	end_ready	rst	unused
slvreg_1 (R)	end_valid	start_ready	unused	
slvreg_2 (W)	arg_1			
⋮	⋮			
slvreg_N+1 (W)	arg_N			
slvreg_N+2	end_out			

Table 1. Mapping between the slave registers and the accelerator's ports | W : write-only | R : read-only | N : number of arguments

2.1.4 The floating point operators

The Dynamatic output uses black-box floating point operators components. Therefore, to be able to implement the design, one has to generate the floating point IPs specifically used by the accelerator. This is done through VivadoHLS. The original C/C++ program is fed into VivadoHLS which then apply synthesis to it. This creates tcl scripts and HDL files for the floating point operators, which we then respectively source and add in the AXI IP project. Finally, the component names in arithmetic_units.vhd need to updated to match the ones from VivadoHLS.

2.2 The block design

Once the IP is generated, it needs to be integrated inside the block design.

2.2.1 The BRAMs

The BRAMs are configured as True Dual Port BRAMs in standalone mode with no byte write enable and all output registers disabled. The two Ports are referred to as Port A and Port B. This means that they provide the following interface twice :

- clk (in) : clock input
- addr (in) : memory address
- din (in) : write data
- dout (out) : read data
- en (in) : enables operations
- we (in) : write enable

The processing system communicates with the BRAMs through AXI BRAM Controllers. However, this component addresses bytes and not words of the chosen width (like 32 or 64) and uses byte write enable. Two VHDL modules are used to make its interface compatible with the BRAMs in standalone mode. A write_enb_adapter converts the byte write enable into a simple write enable by outputting '1' when all bytes are written to and 0 otherwise. Another module, called address_adapter shifts the AXI BRAM Controller's address by the required number of bits (2 for 32 bits word, 3 for 64 bits words, etc.).

In the following section, the mention of write-only, read-only or read-and-write will be from the accelerator's point of view. Also, the solutions discussed will be per array solutions. Dynamatic arrays use two different set of ports, one exclusively for writing and the other exclusively for reading. Consequently, for a read-only or write-only array we add one BRAM and one AXI BRAM Controller to the design. The appropriate set of ports from the accelerator is connected to Port B of the BRAM and the AXI BRAM Controller is connected to Port A.

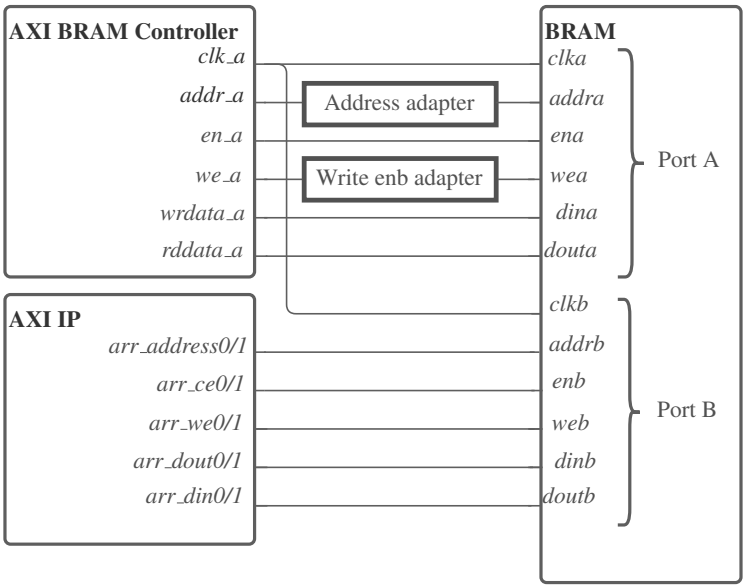


Figure 2. BRAM write/read-only case

An issue arises when the array is read and write. Indeed, it is not possible to connect the two set of ports of the accelerator to the BRAM because it also needs to be connected to the AXI BRAM Controller. Also, there is no three port BRAM available. Two solutions will be presented, the second one being an improvement on the first.

The first solution consists in using two BRAMs and two AXI BRAM Controllers for each W/R array. One BRAM is connected to the read ports of the array and the other BRAM is connected to the write ports of

the array. The AXI BRAM Controller connected to the ”write” BRAM will be used to read the values written in the array by the accelerator. And, the AXI BRAM Controller connected to the ”read” BRAM will be used to initialize the array for the accelerator.

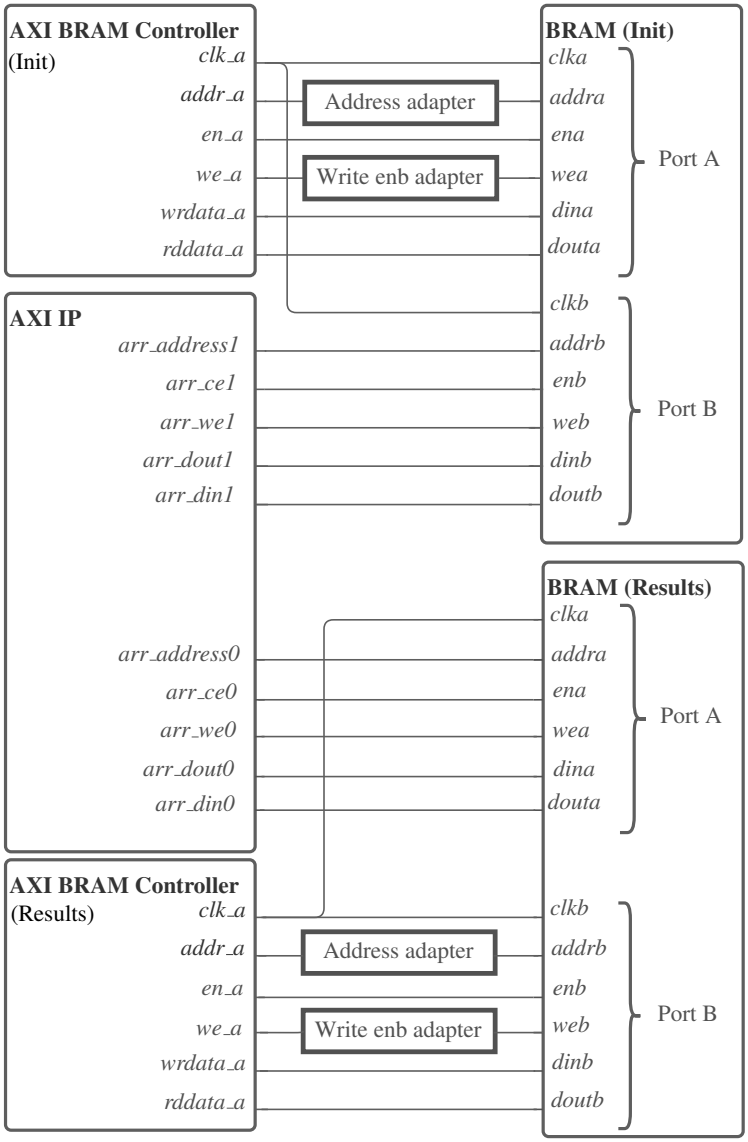


Figure 3. BRAM write and read solution 1

This solution has two major drawbacks. First, it multiplies by two memory use. Moreover, it prevents the design to work with any computation that depends on previously computed results. For instance :

arr[i] = 3 + arr[i - 1];

Let’s say to start the computation only arr[0] needs to be initialized. Since the accelerator only reads the memory initialized by the processing system and not the memory it writes to, as soon as the computation will depend on arr[i] with i $>$ 1 we enter undefined behavior. Indeed, we don’t know what is in the Init BRAM after the first value. Hence the need to find another solution. One realization is that the accelerator and the processing system don’t need to access memory at the same time. The processing system accesses the memory before the accelerator to initialize it and after to get the results. Therefore, a third port isn’t needed. There only needs to be a module that arbitrates the access to memory on one of the port of the BRAM. On the one hand, Port B of the BRAM will always be connected to the write ports of the accelerator for the array. On the other hand, Port A of the BRAM will either serve the AXI BRAM Controller’s requests or the requests of the read ports of the accelerator for the array. The rule for the arbitration is the following :

If the accelerator doesn’t want to access the memory or if the BRAM Controller wants to write, give access to the BRAM Controller, otherwise, give access to the accelerator.

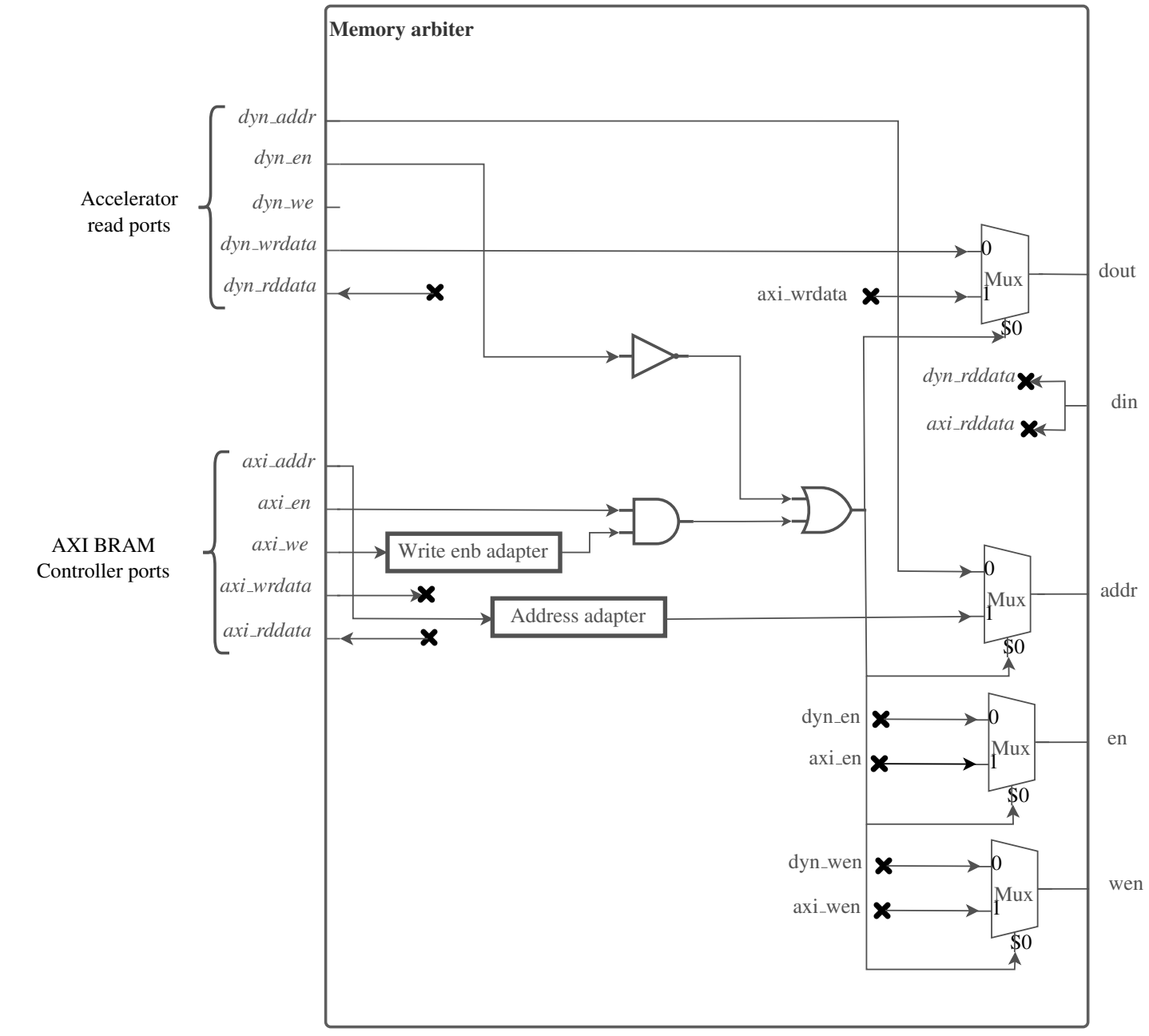


Figure 4. Memory arbiter

We have seen two iterations of the design for read and write arrays, here is the final design :

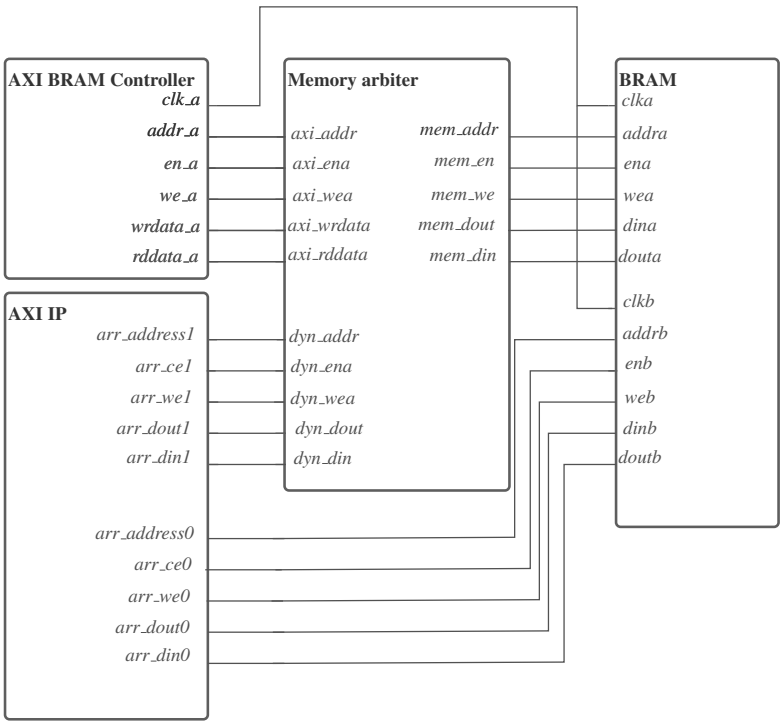


Figure 5. BRAM write and read solution 2

This solution fixes the two drawbacks of the first one. We no longer need to duplicate the BRAM. Moreover, since the accelerator can read the memory it has written, the problematic computations mentioned earlier are no longer an issue.

2.2.2 The complete block design

The complete design will feature the solutions found for the BRAMs for each array depending on whether it is read-only, write-only or read and write. It will also have the processing system. The following block design contains an accelerator which uses two arrays, one for the results which is read/write (out) and one only for initialization which is read-only (init). The writing set of ports for init is not connected because it will never be used. Also, the AXI SmartConnect allows one master peripheral to control several slaves peripheral. The control/status signals and arguments are passed-through the S01_AXI port of the AXI IP. Any request from the processing system to read or write init will go through the S00_AXI port and any request for out will go through the S02_AXI port.

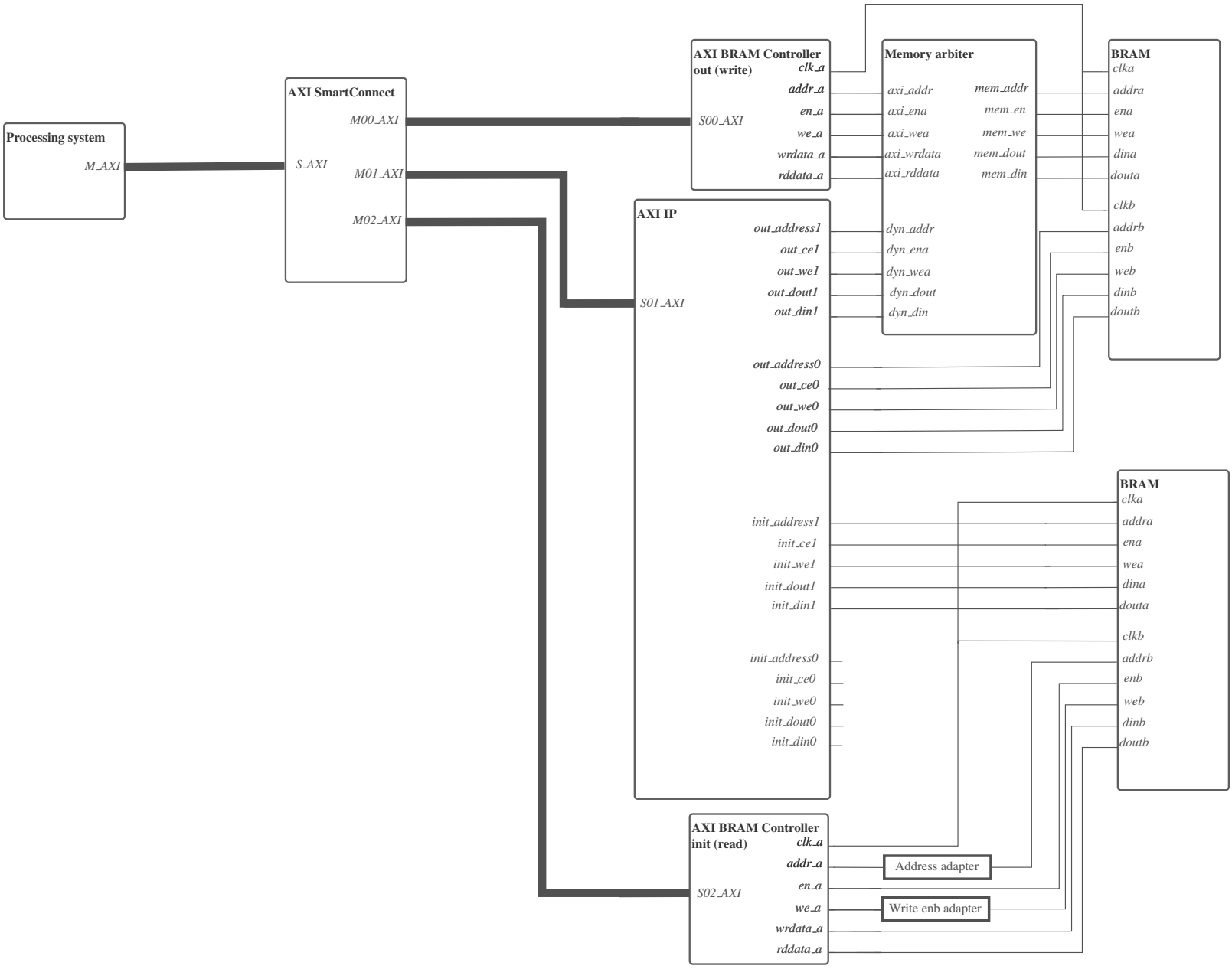


Figure 6. Block design where init is a read-only array and out is a read/write array

2.2.3 Latency issues

When tested with floats, this design produced strange results and behaviors. Indeed, the memory contained the correct values but at the wrong addresses. They seemed to be shifted. For instance, we would get the value that should have been at address 3 at address 0, and the value at address 4 we got at address 1.

Also, some output values were duplicated, meaning that two consecutive addresses stored the same value while we should have gotten two different values at these addresses. This was due to the latencies of the floating point components that were not properly tuned for the accelerator. To properly tune such a component, one has to look at the .dot file produced by Dynamatic and subtract 2 from its latency. This value should then be used to update the c_latency parameter of the tcl script for the floating point component. This has to be done for every floating point operator used by the accelerator.

3. Automation

Once the proper design template has been studied, it has to be automated to be practical. The automation tool is written in C and can be found in the references.

First, the tool asks the part number for which the design should be created. Then, it needs the hdl directory path, which is the output directory for the hdl produced by Dynamatic and the top hdl file name. Then, it can retrieve the entity name and the name of each array as well as their width. It does the same thing for the arguments. In the design section of the report several solutions for connecting arrays to memory were presented. Since they depend on whether the array is R, W or R/W, the tool parses the source C/C++ file to find this information. A VivadoHLS project is created with this source file in order to retrieve the floating point data for the accelerated function. This consists in a pair composed of a VHDL file and a tcl script for each floating point component. The program then read the dot file generated by Dynamatic which stores the latency parameter for each operator. Also, for comparison operators it has to read their names in the dot file because VivadoHLS only creates one fcmp HDL and tcl script pair. This component can implement several operations. The operation is selected using an opcode. The tool then asks the user for the path of the soon to be created Vivado project and for its name. A script for generating the AXI template files with the correct number of slave registers is created. This corresponds to the number of arguments plus three. Indeed, there is necessarily one register for reading the status signals, one for writing to the control/status signals and one for the return value. The script also imports the hdl files created by Dynamatic and configure them to be treated as VHDL 2008 files. Then, the script for creating the actual project is generated. Both these tcl files are then sourced by Vivado. The next step is to update the arithmetic_units.vhd file so that the component names it uses for floating point operators match the ones from VivadoHLS. The AXI files are also modified to instantiate the accelerator, expose its memory interface to the rest of the block design and map the slave registers to the correct signals. Finally, the program writes the last script for Vivado. This script first adds the floating point files from VivadoHLS to the AXI IP project, source the needed scripts and pack the IP. Then, for each array, it creates the appropriate memory layout and make the requisite connections. The IP is then added to the block design, and an AXI SmartConnect connects with the AXI BRAM Controllers and the AXI IP. If the part number is recognized by the program, the automation can go further. The compatible processing system is added to the design and properly configured. It is then coupled with the AXI SmartConnect.

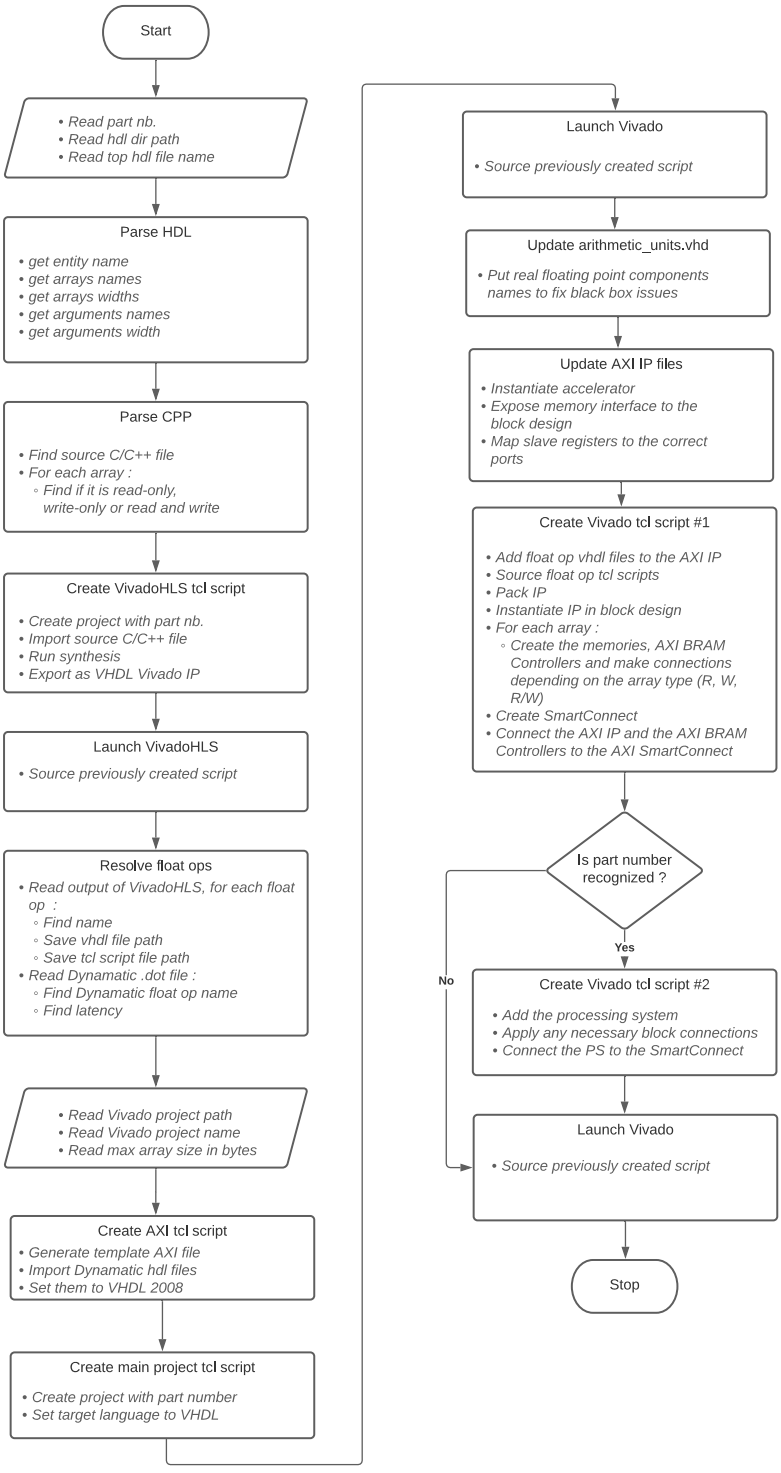


Figure 7. Automation flowchart

4. Conclusion

The goal was to make a functioning and practical accelerator out of the Dynamic output. Functioning meaning that it needs to be able to communicate with the processing system and practical meaning that it should be easy and quick to do so. The design and the automation tool have been both tested and provided correct results. The time it takes to create a proper block design for the accelerator generated by Dynamatic has been significantly reduced. However, the design and the program have their limitations. First, the arguments can only be 32 bits for now. Indeed, the configuration of the template for the AXI IP doesn't allow width modification. Even though a generic storing the width is provided in the two files constituting the AXI IP, its modification lead to incorrect results. This issue has to be further debugged. Also, due to a lack of time, the tool and the design have not been tested with the Amazon F1.

References

A. De Gendt. Github repository of the automation tool, 2020. URL `https://github.com/ADGLY/AutomateDynamatic`.

Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Invited tutorial : Dynamatic: From c/c++ to dynamically scheduled circuits. 2020. URL `https://dynamatic.epfl.ch/papers/FPGA_20_Dynamatic_From_C_C_to_Dynamically_Scheduled_Circuits.pdf`.

AXI Reference Guide. Xilinx, 2017. URL `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf`.

AXI Block RAM (BRAM) Controller v4.1. Xilinx, 2019a. URL `https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_1/pg078-axi-bram-ctrl.pdf`.

Block Memory Generator v8.4. Xilinx, 2019b. URL `https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf`.