Universidade do Minho
Escola de Engenharia

Afonso Oliveira, PG53599
Gonçalo Moreira, PG50802

Desenvolvimento de um CPU - VesPA

Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Adriano Tavares

janeiro de 2024

List of Figures

Contents

1 Introduction

This document presents an in-depth report on the VesPA Processor Design project. The primary objective of this endeavor is to develop and implement a specialized version of a VesPA microcontroller. The project's main objective is the deployment of this microcontroller within a Field-Programmable Gate Array (FPGA) system. Specifically, the implementation will utilize the Zybo Z7, a sophisticated FPGA board produced by Digilent.

In pursuing this ambitious project, we have utilized the extensive knowledge and skills accumulated during academic studies, particularly within the realms of Microcontrollers and Microprocessors. This academic foundation has been instrumental in guiding the project's direction and execution. Furthermore, we have incorporated advanced insights and methodologies learned through our masters in Embedded Systems. This course, expertly taught by Guru and Professor Adriano Tavares, has provided essential specialized knowledge that has significantly contributed to the project's development.

Through the VesPA Processor Design project, we aim to showcase the practical application of academic learning in a real-world, technical scenario. If after reading the report any questions still remain on the implementation, please visit github.com/AFOliveira/VesPA.

2 Methodology

To meet the established objectives set forth by our professor, a robust methodology was paramount. In the realm of engineering, the importance of a sound methodology cannot be overstressed. It stands as both a discipline and a strategic approach to minimize costs and boost process reliability by analyzing task execution. Moreover, such methods pinpoint optimal personnel utilization in a process, enabling efficient task completion.

Our project strategy was rooted in the adoption of the waterfall model. This model is a systematic and step-wise approach to software development, popular in software engineering and product development realms. It is characterized by its linear, sequential phases, resembling a cascading waterfall. Each stage in this model has definitive goals and endpoints, which, once completed, are not revisited.

The stages of the waterfall methodology are:

1. **Requirements**: The initial stage involves scrutinizing potential project requirements, deadlines, and guidelines, resulting in a detailed functional specification. It primarily focuses on the project's definition and initial planning, without delving into specific process details.

2. **Analysis**: Here, the system's specifications are examined to create product models and business logic that will steer the production process. This stage also encompasses the evaluation of financial and technical resource viability.

3. **Design**: The creation of a design specification document occurs here, specifying technical design elements like programming languages, hardware requirements, data sources, architecture, and services.

4. **Implementation**: Development of source code happens using the previously established models, logic, and requirements. The system is generally constructed in smaller segments or units, which are then combined.

5. **Verification**: This stage involves conducting various tests - quality assurance, unit, system, and beta testing - to detect and fix issues. It might necessitate revisiting the coding phase for debugging purposes. The project advances following successful testing.

6. **Maintenance**: In this ongoing stage, various forms of maintenance (corrective, adaptive, and perfective) are executed to refine, update, and enhance the final product, including the release of updates or new versions.

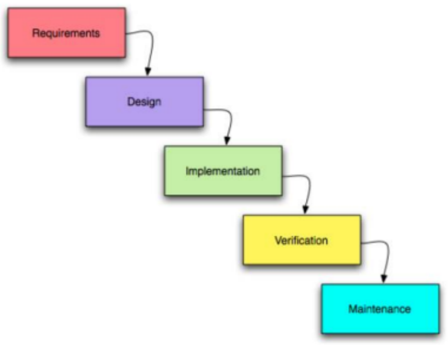


Figure 1: Waterfall methodology

The waterfall model is particularly suitable for projects with well-defined documentation, static requirements, ample resources, set timelines, and established technology. Its benefits include thorough initial documentation and planning, which ensure that teams, whether large or fluctuating, stay informed and united towards a common goal. The model's disciplined, simple-

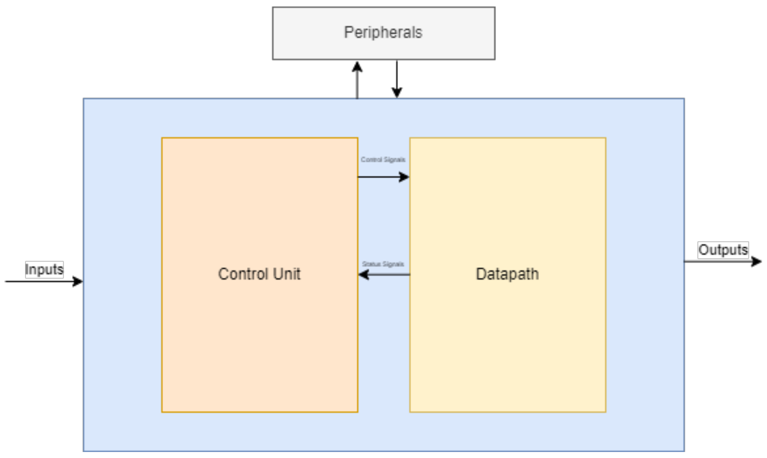


Figure 2: System overview

to-understand structure aids in task organization and supports managerial control based on schedules and deadlines.

In our project, we explored all the phases of the waterfall model, with the exception of the maintenance phase, given the project's limited scope.

3 Analysis

3.1 Requirements

In the context of product development and process enhancement, a 'requirement' can be understood as a documented need of a specific physical or functional characteristic that a particular design, product, or process must fulfill. Pertaining to this project, the requirements are formulated as follows:

- Enable the coding and deployment of a straightforward application onto an FPGA platform.
- Ensure the capability for expansion and scalability to accommodate future enhancements.
- Use a split control unit and datapath design.

3.2 Constraints

Conversely, constraints are elements that restrict project outcomes due to technical or non-technical considerations. For this project, the constraints are identified as:

- Incorporate instructions spanning all categories of the instruction set (arithmetic, logical, data transfer, and control flow instructions).
- Integrate necessary peripheral components.
- Restrict team composition to a maximum of two members.
- Operate within a limited temporal framework (the project is due by the semester's conclusion).
- Create an IP Memory and IP Register bank.
- Use Xilinx Vivado and Verilog.

3.3 Three address processor

A three-address processor is a type of computer architecture that specifies each operand of an instruction explicitly within the instruction itself. In this context, "address" refers to the operand, which could be a memory address or a register. Each instruction in a three-address processor can have up to three operands: two source operands and one destination operand.

3.4 Instruction Set Architecture

The VeSPA ISA (Instruction Set Architecture) encompasses a minimalist set of instructions. The ISA is categorized as follows:

Arithmetic Instructions

- **ADD**: Adds two operands and stores the result.

31 – 27	26 – 22	21 – 17	16	15 – 11	10 – 0	
00010	rdst	rs1	0	rs2	0000	0000 0000

31 – 27	26 – 22	21 – 17	16	15 – 0
00010	rdst	rs1	1	immed16

- **SUB**: Subtracts one operand from another and stores the result.

31 – 27	26 – 22	21 – 17	16	15 – 11	10 – 0		
00010	rdst	rs1	0	rs2	000	0000	0000

31 – 27	26 – 22	21 – 17	16	15 – 0
00010	rdst	rs1	1	immed16

Logical Operations

- **AND**: Performs a logical AND operation between two operands.

31 – 27	26 – 22	21 – 17	16	15 – 11	10 – 0	
00010	rdst	rs1	0	rs2	0000	0000 0000

31 – 27	26 – 22	21 – 17	16	15 – 0
00100	rdst	rs1	1	immed16

- **OR**: Executes a logical OR operation between two operands.

31 – 27	26 – 22	21 – 17	16	15 – 11	10 – 0	
00011	rdst	rs1	0	rs2	0000	0000 0000

31 – 27	26 – 22	21 – 17	16	15 – 0
00011	rdst	rs1	1	immed16

- **XOR**: Carries out an exclusive OR operation between two operands.

31 – 27	26 – 22	21 – 17	16	15 – 11	10 – 0	
00110	rdst	rs1	0	rs2	0000	0000 0000

31 – 27	26 – 22	21 – 17	16	15 – 0
00110	rdst	rs1	1	immed16

- **NOT**: Applies a logical NOT operation, inverting the bits of its operand.

31 – 27	26 – 22	21 – 17	16 – 0
00101	rdst	rs1	0 0000 0000 0000 0000

Control Instructions

- **CMP**: Compares two values and sets condition codes without storing the result.

31 – 27	26 – 22	21 – 17	16	15 – 11	10 – 0
00111	00000	rs1	0	rs2	0000 0000 0000

31 – 27	26 – 22	21 – 17	16	15 – 0
00111	00000	rs1	1	immed16

- **Bxx**: Conditional branch instructions that change program flow based on set conditions.

31 – 27	26 – 23	22 – 0
01000	cond	immed23

cond	**Assembly**	**Condition**
0000	BRA	branch always
1000	BNV	branch never
0001	BCC	branch on carry clear ($\overline{C}$)
0010	BCS	branch on carry set (C)
0010	BVC	branch on overflow clear ($\overline{V}$)
1010	BVS	branch on overflow set (V)
0011	BEQ	branch on equal (Z)
1011	BNE	branch on not equal ($\overline{Z}$)
0100	BGE	branch on greater than or equal to ($\overline{N}$ OR V)
1100	BLT	branch on less than (N AND $\overline{V}$)
0101	BGT	branch on greater than (Z AND ($\overline{N}$ OR V))
1101	BLE	branch on less than or equal to (Z OR (N AND $\overline{V}$))
0110	BPL	branch on plus (positive) ($\overline{N}$)
1110	BMI	branch on minus (negative) (N)

- **JMP**: Unconditionally changes the flow of the program to a specified address.

31 – 27	26 – 22	21 – 17	16	15 – 0
01001	00000	rs1	0	immed16

- **JMPL**: Similar to JMP, but also saves the return address.

31 – 27	26 – 22	21 – 17	16	15 – 0
01001	rdst	rs1	1	immed16

- **RETI**: Used to return after an interrupt.

Data Transfer Instructions

- **LD**: Loads a value from memory into a register.

31 . . . 27	26 . . . 22	21 . . . 0
01010	rdst	immed22

- **LDI**: Loads an immediate value into a register.

31 . . . 27	26 . . . 22	21 . . . 0
01011	rdst	immed22

- **LDX**: Loads a value from an indexed memory address into a register.

31 . . . 27	26 . . . 22	21 . . . 0	
01100	160	rdst	17

- **ST**: Stores a value from a register into memory.

31 . . . 27	26 . . . 22	21 . . . 0
01101	rdst	immed22

- **STX**: Stores a value from a register into an indexed memory address.

31 . . . 27	26 . . . 22	21 . . . 0	
01110	rst	rs1	immed17

Miscellaneous Instructions

- **NOP**: A no-operation instruction used for timing adjustments.

31 . . . 27	26 . . . 0
00000	000 0000 0000 0000 0000 0000

- **HLT**: Halts the processor's execution of instructions.

31 . . . 27	26 . . . 0
11111	000 0000 0000 0000 0000 0000

4 Design

4.1 Control Unit

The Control Unit (CU) is the component of the CPU that directs the operation of the processor by interpreting the computer program's instructions and generating the necessary signals to execute them. It orchestrates the timing and control of data flow within the CPU, coordinating the activities of the whole processor.

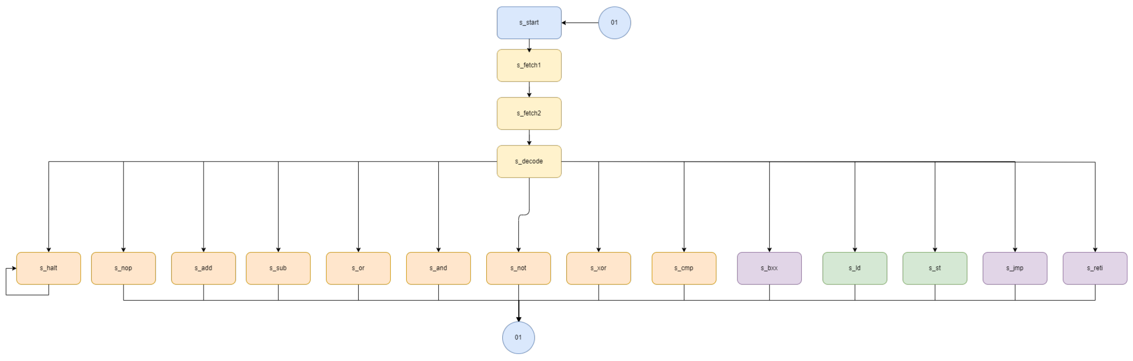


Figure 3: Control Unit FSM

The process designed in the FSM presented on figure 3 has 5 different stages:

1. **Start**: In the state start, there is a timing adjustment for some of the memory operations to be finished and there is the verification if any interrupt has to be serviced.
2. **Fetch 1**: Fetches the first part of the IR from the memory, that includes the opcode.
3. **Fetch 2**: Fetches the second part of the IR from the memory.
4. **Decode**: Decodes the instruction to be executed as well as the rest of the necessary data from the IR.
5. **Execute**: Executes the instruction.

4.2 DataPath

A data path is an integral part of a computer's CPU that performs operations on data, facilitating the execution of instructions from machine code. It comprises various interconnected hardware modules that process data and collectively execute the instructions.

In the depicted datapath design present on 4, the following modules are present:

1. **Program Counter (PC)**: A register that holds the address of the instruction to be executed next, crucial for the instruction sequencing process.

2. **Instruction Register (IR)**: It holds the currently executing instruction, acting as a staging area for CPU operations.

3. **Register File**: A set of registers for storing temporary data, providing the CPU with quick access to these values.

4. **Arithmetic Logic Unit (ALU)**: This module performs both arithmetic and logic operations, forming the computational heart of the CPU.

5. **Interrupt Control**: Manages interrupts that can pause or alter the execution flow, responding to both external and internal events.

These components are designed to work in concert, enabling the CPU to process instructions, handle data manipulation, and manage the flow of execution efficiently. Memory was designed outside the diagram, even though it is a part of the datapath for better understading of each module and in order to keep simplicity.

Moreover, there is a need to specify that the datapath does not take any decision by itself and always works with control unit signals.

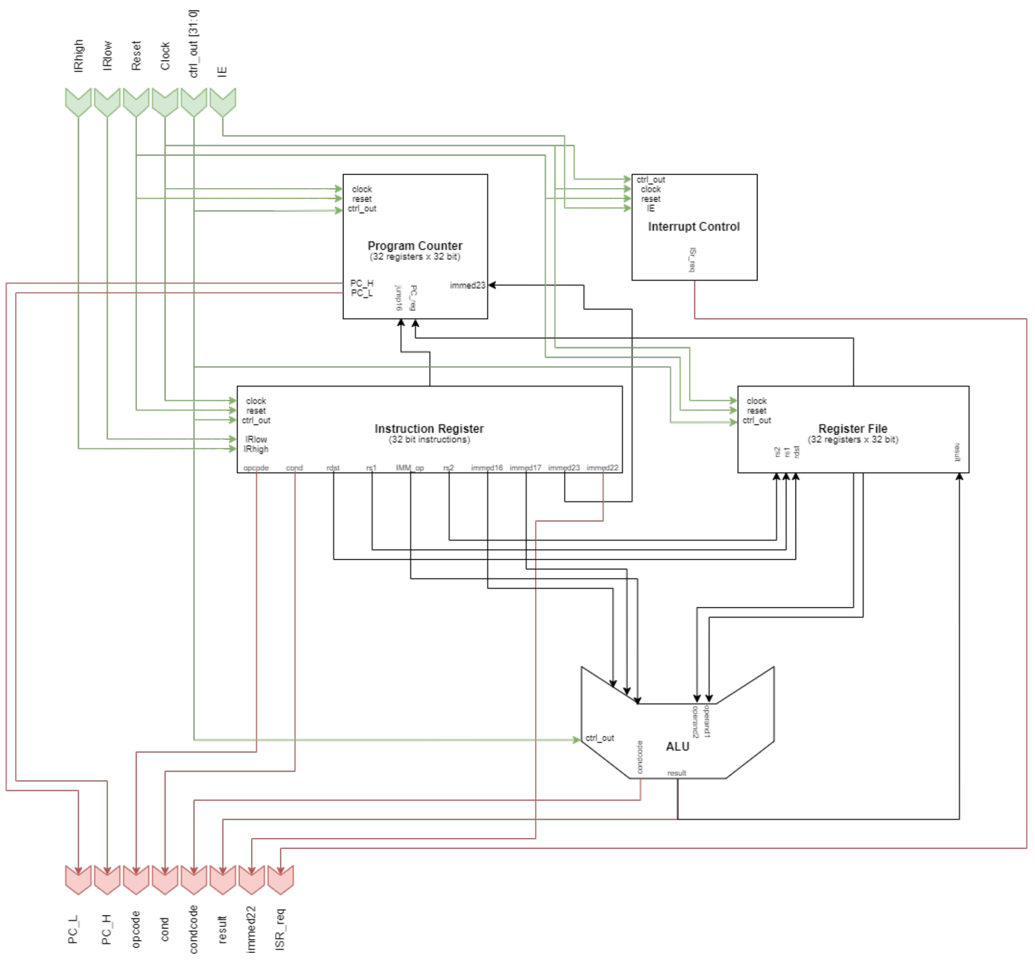


Figure 4: Datapath Design

4.2.1 Memory Design

In the VeSPA implementation, the Harvard architecture offers distinct advantages over the Von Neumann architecture, particularly in terms of performance and security. The key benefit lies in its separation of storage and pathways for instructions and data, which enables faster data fetching and instruction execution. This parallelism leads to enhanced processing speed, a critical factor for the efficiency-focused VeSPA.

Additionally, the Harvard architecture's segregation of memory spaces for instructions and data inherently boosts system stability and security. This separation prevents the accidental corruption of instruction sets by program data, ensuring more reliable operation.

In summary, the Harvard architecture enhances VeSPA by providing faster processing, improved security and reliability, and better resource optimization compared to the Von Neumann architecture.

In the design of our 32-bit CPU, particular attention has been given to ensuring that the memory is byte-addressable. Byte addressability is a crucial feature, allowing each byte within the memory to be accessed individually. This is especially important in a 32-bit architecture, where data is often handled in 32-bit (4-byte) chunks. To align with this requirement and to facilitate proper functioning, specific logic has been implemented within the CPU.

The best way we found to implement this memory was through a dual port BRam Block that is presented on figure 4.

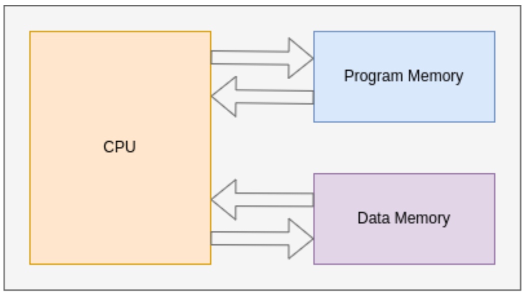


Figure 5: Harvard memory design

4.2.2 Data Memory

In our Von Neumann architecture, the data memory plays a pivotal role, designed as RAM (Random Access Memory) to facilitate efficient execution of Store and Load instructions. This design choice not only allows for dynamic data manipulation but also ensures seamless data retrieval and storage operations, crucial for the overall performance of the CPU. Additionally, a dedicated section of this memory is reserved explicitly for peripherals, optimizing the interaction between the CPU and external devices. This reserved space streamlines data exchange with peripherals, enhancing system functionality and ensuring a more integrated and efficient operational environment.
The best way we found to implement this memory was through a dual port BRam Block that is presented on figure 6.

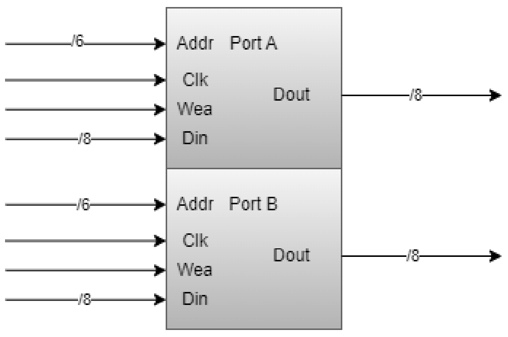


Figure 6: Data Memory Design

In figure 6:

- Addr, represents the adress to read/write images.
- Clk, represents the clock source.
- Wea, represents the pin that enables/disables writing on the addr.
- Din, represents the pin that inputs data to the memory.

This approach of a two port BRam allows for a byte-adressable Data memory, that needs two cycles to process the 32 bit memory that the CPU stores/loads. This approach is great to use with the Harvard architecture because, since the architecture decision makes the process one cycle faster, there is enough room to make adjustments in the specific memory implementation.

4.2.3 Instruction Memory

The instruction memory in our CPU serves as the repository for storing executable code. In designing this critical component, we opted for RAM (Random Access Memory) instead of the more traditional ROM (Read-Only Memory). This choice was driven by the flexibility that RAM offers, notably its ability to be reset or reprogrammed as needed.

The use of RAM for instruction memory brings several advantages. Firstly, it allows for dynamic updating of the code, which is particularly useful during the development phase or for applications requiring frequent software updates. The ability to rewrite the instruction set enables developers to modify and improve the CPU's functionality without the need for physical hardware changes.

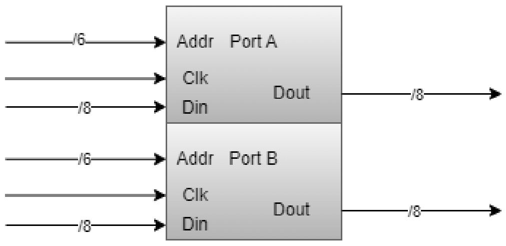


Figure 7: Instruction Memory Design

Moreover, the reprogrammable nature of RAM-based instruction memory aligns well with systems that might need to adapt to different tasks or environments. It allows the CPU to be more versatile and adaptable, a quality especially valuable in experimental or educational settings where the processor might need to run a variety of programs.

For the ip integration we followed the same principle as for the the Data Memory. The differences are that even though the memory is reprogramable, we did not implement the methods, and so there are still optimizations that can be made. With that in mind figure 7 represents our Instruction memory.

In figure 7:

- Addr, represents the adress to read/write images.
- Clk, represents the clock source.
- Wea, represents the pin that enables/disables writing on the addr.
- Din, represents the pin that inputs data to the memory.
- Dout, represents the ping that outputs data from the memory.

4.2.4 Register File

In our processor's design, we have implemented a three-address register file, as mandated by the ISA (Instruction Set Architecture). This register bank is crucial for facilitating the three-address instruction format, where most instructions explicitly specify two source operands and one destination operand. The presence of this register file allows for efficient execution of complex arithmetic and logical computations, as it provides quick access to operands and results. This implementation aligns with the processor's ISA requirements. For the design, we did it on a two single-ports BRam Blocks.

In figure 8:

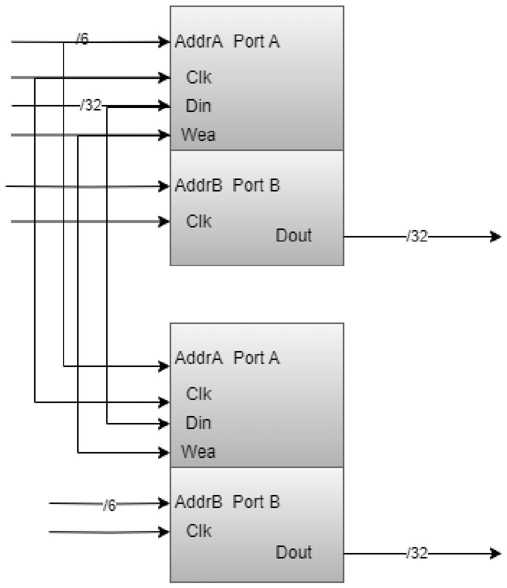


Figure 8: Register File Design

- AddrA, represents the adress to write data for both blocks.
- AddrB-1, represents the address to read data from Block 1.
- AddrB-2, represents the address to read data from Block 2.
- Clk, represents the clock source.
- Wea, represents the pin that enables/disables writing on the addrA.
- Din, represents the pins that input data to the Register File.
- Dout, represents the pins that output data from the Register File.

4.2.5 ALU

An Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU), designed as a combinational digital circuit. Its primary function is to perform arithmetic and bitwise operations on integer binary numbers. This includes basic calculations like addition and subtraction, as well as operations involving the manipulation of individual bits within a binary number, such as AND, OR, XOR, and shift operations. Furthermore, the ALU is used when using the conditional branches that need some logic to be verified.

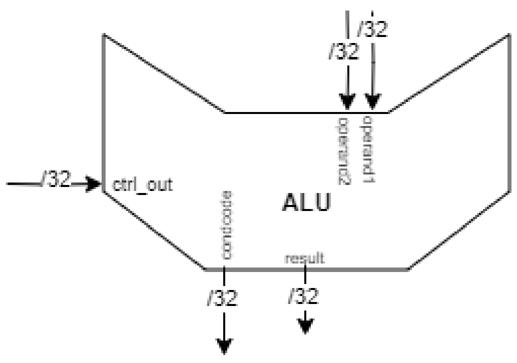


Figure 9: ALU Design

In figure 9:

- operand1 represents the input of th first opperand to be used.
- operand2 represents the input second opperand to be used.
- ctrl-out represents the input from CU control bits.
- condcode represents the output pin that of the Condition Flags.
- result represents the output pin of the ALU result.

In the VeSPA architecture, the ALU (Arithmetic Logic Unit) **condition code** bits, specifically **C**, **V**, **Z**, and **N**, are vital for its operation. These bits are set based on the results of ALU operations and influence conditional branch instructions:

- **C (Carry) Bit**: Set if a carry-out from the most significant bit position occurs, indicating an overflow in unsigned arithmetic.
- **V (Overflow) Bit**: Indicates a carry-out from the sign bit, signaling an overflow in arithmetic operations.
- **Z (Zero) Bit**: Set when the ALU operation result is zero, crucial for equality checks and comparisons.
- **N (Negative) Bit**: Set when the operation result is negative, as indicated by the sign bit being 1, used in signed comparisons.

These condition codes significantly influence the processor's decision-making process based on arithmetic and logical operation results.

4.3 Interrupt Controller

An interrupt is an event that causes a deviation from the normal execution flow of a processor, redirecting it to a specific code segment. There are various types of interrupts, including:

- **External Interrupts**: These are caused by the activation of an electrical signal generated by a device external to the processor.
- **Traps**: Traps are caused by the execution of special instructions from the processor's instruction set.
- **Exceptions**: Exceptions are triggered by attempts to execute illegal or unknown operations.

In our system architecture, the interrupt controller plays a pivotal role in managing and responding to both SPI (Serial Peripheral Interface) and external interrupts.

- **SPI Interrupts**: The controller efficiently handles interrupts generated through the Serial Peripheral Interface. This is critical for seamless communication with SPI devices.
- **External Interrupts**: Additionally, the controller is adept at managing external interrupts. These interrupts are triggered by external hardware devices.

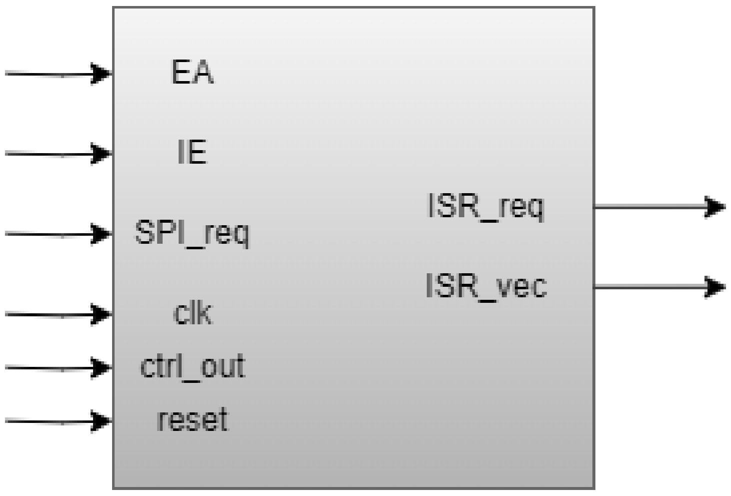


Figure 10: Interrupt Controller Design

- EA represents the input of the Enable All interruptions.
- IE represents the external interrupt, coming directly from the button debounce.
- SPI-req represents the SPI reponsde module requesting an interrupt to process data.
- clk represetn clock source.
- ISR-req represents the interrupt controller method to flag the need for an interrupt to the Control Unit.
- ISR-vec represents the interrupt controller method to flag which interruption needs to be handled.

4.4 SPI

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol designed for short-range communication, predominantly used in embedded systems. Originally developed by Motorola in the mid-1980s, SPI has since become an industry standard for its efficiency and reliability in facilitating communication between a central processor and various peripheral devices.

To proper implement SPI, we implemented Clock Domain Crossing, in order to metastabilize the Chip Select signal.

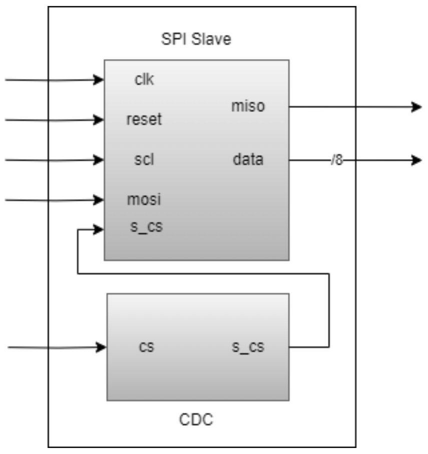


Figure 11: SPI Slave design

- clk represents clock source.

- reset represent reset source.
- scl represents masters source clock in spi protocol.
- mosi represents Master Out Slave In pin.
- s-cs represents Metastable Chip Select signal.
- CS represents Chip Select input from Masters'.

4.4.1 Clock Domain Crossing

In SPI communication, clock domain crossing (CDC) is essential when the master's serial clock (SCLK) and the slave's internal clock are asynchronous. The master controls the data flow with its SCLK, but the slave must synchronize this data to its own clock to prevent errors. This is typically done using synchronization flip-flops that align the data with the slave's clock edges, mitigating the risk of metastability—a condition where the flip-flop fails to resolve to a stable logical state. Careful design of these CDC mechanisms ensures reliable data transfer between the SPI master and slave devices, maintaining the integrity of the communication process. In this VesPA implementation, CDC was used to properly verify the CS signal.

5 Implementation

5.1 Opcode and Defines

Firstly, there was an opcode module implementation, where all the opcodes were defined to a macro. Later, we added other defines, such as branch codes and control unit flags array position, that would also be useful during VesPA implementation.

```
`ifndef _OPCODES_V_
`define _OPCODES_V_


//Arithemtic
`define s_add    5'b00001
`define s_sub    5'b00010
`define s_bxx    5'b01000



//Logical
 `define s_or     5'b00011
 `define s_and    5'b00100
 `define s_not    5'b00101
 `define s_xor    5'b00110
 `define s_cmp    5'b00111

//Bolean

//Program Branching

`define s_jmp    5'b01001
`define s_jmpl   5'b01010
`define s_reti   5'b11100


//Memory management
`define s_ld 5'b01011
`define s_ldi 5'b01100
`define s_ldx 5'b01101
`define s_st 5'b01110
`define s_st2 5'b11011
`define s_stx 5'b01111

`define s_nop      5'b00000
`define s_halt 5'b11111



//State machine help
`define s_start 5'b10001
`define s_start2 5'b11001
`define s_jextra2 5'b10010
`define s_fetch 5'b10011
`define s_fetch2 5'b10100
`define s_fetch3 5'b10110
`define s_decode 5'b10101
`define s_extra 5'b11110


//Branch Logic
`define BRA 4'b0000
`define BNV 4'b1000
`define BCC 4'b0001
`define BCS 4'b1001
`define BVC 4'b0010
`define BVS 4'b1010
`define BEQ 4'b0011
`define BNE 4'b1011
`define BGE 4'b0100
`define BLT 4'b1100
```

```
`define BGT 4'b0101
`define BLE 4'b1101
`define BPL 4'b0110
`define BMI 4'b1101

//ctrl array positions
`define p_add        5'b00000    //0
`define p_sub        5'b00001    //1
`define p_and        5'b00010    //2
`define p_or         5'b00011    //3
`define p_xor        5'b00100    //4
`define p_not        5'b00101    //5
`define p_cmp        5'b00110    //6
`define p_ld         5'b00111    //7
`define p_st         5'b01000    //8
`define p_outd1      5'b01001    //9
`define p_outd2      5'b01010    //10
`define p_outdr      5'b01011    //11
`define p_brxen      5'b01100    //12
`define p_IRL1       5'b01101    //13
`define p_IRL2       5'b01110    //14
`define p_code_en    5'b01111    //15
`define p_PCinc      5'b10000    //16
`define p_PCload     5'b10001    //17
`define p_PCisr      5'b10010    //18
`define p_ram_r_e    5'b10011    //19
`define p_ram_w_e    5'b10100    //20
`define p_w_data     5'b10101    //21
`define p_jmpen      5'b10110    //22
`define p_muxsel1    5'b10111    //23
`define p_muxselldi 5'b11000     //24
`define p_st2        5'b11001    //25
`define p_ld2        5'b11010    //26
`define p_reti       5'b11011    //27


`endif
```

Listing 1: Defines Module Implementation

5.2 CPU

To create an abstraction layer from the peripherals to the CPU, there was a definition of a CPU.v That has the instantiation of the Control Unit and the datapath. This way we also have the inputs and outputs of the VesPA.

```
    module CPU(
    input clk,
    input rst,
    input IE, //button interrupt enable alter future
    input EA,

    input [7:0] mem_outL,  // both memories outputs
    input [7:0] mem_outH,
    input [7:0] IRhigh,
    input [7:0] IRlow,
    input [31:0] spi_data,

    input SPI_req,

    output [31:0]immed22,
    output [31:0]PC,
    output [31:0]PChigh,
    output [31:0]PClow,
    output ram_write_en,


    output [7:0] restomem1,
    output [7:0] restomem2,
    output [31:0]result,
    output [7:0] d_addrH,
    output [7:0] d_addrL,

    output [31:0] ctrl_out,
    output reg [3:0] gr_result
    );
```

Listing 2: CPU interface definition

This way, we can see all the signals the CPU needs to send to the memory and peripherals. Just to note, the memory is inside the datapath, but for better representation and understading of a real and more advanced CPU, we kept it outside in the RTL design.

```
    control_unit ctrl_unit (
        .clk(clk),
        .rst(rst),
        .opcode(opcode),
        .cond(cond),
        .IMM_op (IMM_op),
        .C(C),
        .Z(Z),
        .N(N),
        .V(V),
        .ISR_req(ISR_req),
        .ram_write_en(ram_write_en),
        .ctrl_out(ctrl_out)
        );
```

```

datapath data_path  (
          .clk(clk),
          .rst(rst),
          .ISR_req(ISR_req),
          .IE(IE),
          .EA(EA),
          .mem_outL(mem_outL),
          .mem_outH(mem_outH),
          .IRhigh(IRhigh),
          .IRlow(IRlow),
          .ctrl_out(ctrl_out),
          .operandi(spi_data),
          .SPI_req(SPI_req),
          .opcode(opcode),
          .cond(cond),
          .IMM_op(IMM_op),
          .C(C),
          .Z(Z),
          .N(N),
          .V(V),
          .immed22(immed22),
          .PC(PC),
          .PChigh(PChigh),
          .PClow(PClow),
          .restomem1(restomem1),
          .restomem2(restomem2),
          .finresult(result),
          .d_addrH(d_addrH),
          .d_addrL(d_addrL)
          );
```

Listing 3: CU and DataPath instation on CPU.v

Additionaly, the RTL design of the cpu has the interface present on figure 12. This design is useful to check any errors on the interface as well as to give a more interactive understanding of the CPU module.

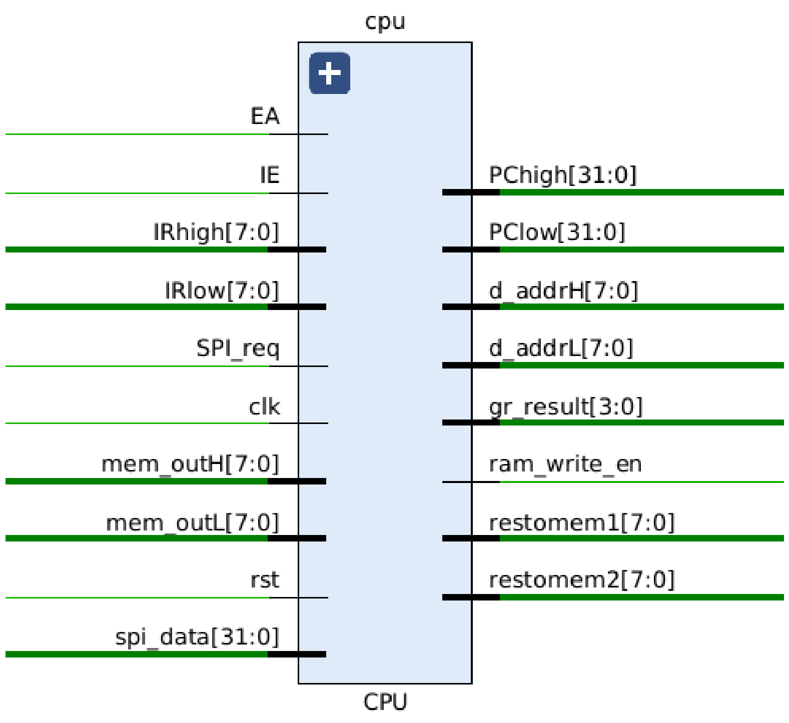


Figure 12: CPU RTL design

5.3 Control Unit

In the control Unit, we proceeded to translate the FSM in figure 3 to verilog code. Firstly, we defined all the inputs and outputs.

```verilog
    module control_unit(
    input clk,
    input rst,
    input [4:0]opcode,
    input [3:0]cond,
    input IMM_op,

    input C,
    input Z,
    input N,
    input V,

    input ISR_req,

    output ram_write_en,

    output [31:0]ctrl_out
    );
```

Listing 4: controlunit.v module defition

Then we proceeded to the implementation of the FSM in Verilog. The first part of the FSM is synchronous and used to assign the state and next state, and the second part is fully assynchronous.

```
        always @(posedge clk)
    begin

        if(rst == 1'b1)
        begin
            state <= `s_start;
        end else begin
            state <= next_state;
        end

    end

    always@(*)
    begin
        case (state)

              `s_start:
               begin
                  next_state = `s_fetch;
               end

              `s_fetch:
                    next_state = `s_fetch2;

              `s_fetch2:
                    next_state = `s_decode;
              `s_decode:
                    next_state = opcode;
              `s_nop:
                    next_state = `s_start;
              `s_add:
                    next_state = `s_start;
              `s_sub:
                    next_state = `s_start;
              `s_or:
                    next_state = `s_start;
              `s_and:
                    next_state = `s_start;
              `s_not:
                    next_state = `s_start;
              `s_xor:
                    next_state = `s_start;
              `s_cmp:
                    next_state = `s_start;
              `s_bxx:
                    next_state = `s_start;
              `s_jmp:
                    next_state = `s_start;
              `s_ld:
                    next_state = `s_start;
              `s_ldi:
                    next_state = `s_start;
              `s_ldx:
                    next_state = `s_start;
              `s_st:
                    next_state = `s_start;
              `s_stx:
                    next_state = `s_start;
              `s_sti:
                    next_state = `s_start;
              `s_halt:
                    next_state = `s_halt;
              `s_reti:
              begin
                    next_state = `s_start;
              end

           default:
            next_state = `s_start;
        endcase
```

Listing 5: controlunit.v FSM

In figure 13 there is he RTL auto designed by Vivado after the verilog implementation. There is a possiblity to check that all the inputs and outputs are correctly designed.

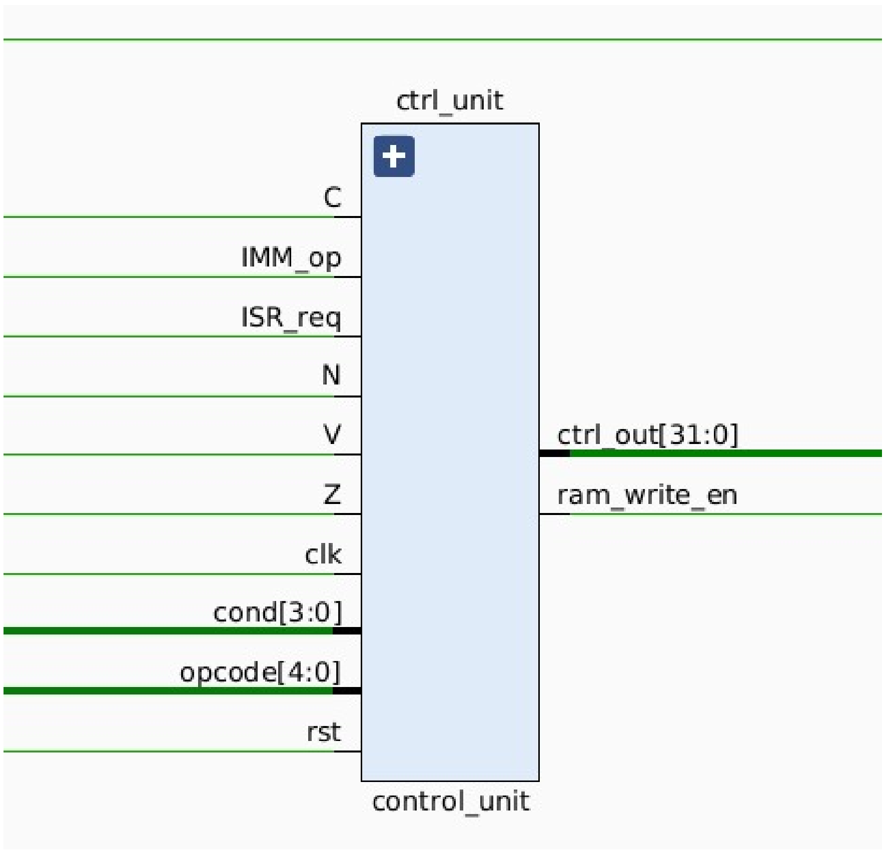


Figure 13: Control Unit RTL design

5.4 DataPath

In the datapath implementation we translated to code figure 4. Firstly, we defined the module, as well as its inputs and outputs.

```
    module datapath(
        input clk,
        input rst,
        input ISR_req,
        input IE, //external interrupt
        input EA, //enable inteterrupts

        input [7:0] mem_outL,  // both memories outputs
        input [7:0] mem_outH,
        input [7:0]IRhigh,
        input [7:0]IRlow,
        input [31:0]ctrl_out, // control unit flags
        input [31:0] operandi,

        input SPI_req,

        output [4:0]opcode,
        output [3:0]cond,
        output IMM_op,
        output C,
        output Z,
        output N,
        output V,

        output [31:0]immed22,
        output [31:0]PC,
        output [31:0]PChigh,
        output [31:0]PClow,

        output [7:0]restomem1,
        output [7:0]restomem2,
        output [31:0]finresult,
        output [7:0] d_addrH,
        output [7:0] d_addrL

        );
```

Listing 6: Datapath Verilog interface

The datapath, as explained in figure 4, is composed by different modules. This modules are all instatiated in the datapath.v module and can be seen in the following listing.

```
    ALU arith_logic_unit (
          .opcode(opcode),
          .rdst(rdst),
          .rs1(rs1),
          .IMM_op(IMM_op),
          .rs2(rs2),
          .immed22(immed22),
          .immed16(immed16),
          .operand1(operand1),
          .operandi(operandi),
          .operand2(operand2),
          .mem_operand(mem_result),
          .ctrl_out(ctrl_out),
          .result(result),
          .C(C),
          .Z(Z),
          .N(N),
          .V(V)
          );

register_bank register_bank (
            .clock(clk),
            .reset(rst),
            .rs1(dst),
            .rs2(rs2),
            .rdst(rdst),
            .in_data(finresult),
            .ctrl_out(ctrl_out),
            .pc_reg_val(pc_reg_val),
            .operand1(operand1),
            .operand2(operand2),
            .operandoutram(operandoutram)
            );



Program_Counter Program_Counter(
    .clk(clk),
    .rst(rst),
    .op_immed23(immed23),
    .jmp_16adrr(jmp_16adrr),
    .pc_reg_val(operand1),
    .ctrl_out(ctrl_out),
    .isr_vec(isr_vec),
    .PC(PC),
    .PClow(PClow),
    .PChigh(PChigh)
    );


Instruction_Register Instruction_Register(
   .rst(rst),
   .clk(clk),
   .ctrl_out(ctrl_out),
   .IRlow(IRlow),
   .IRhigh(IRhigh),
   .jmp_16adrr(jmp_16adrr),
   .opcode(opcode),
   .rdst(rdst),
   .rs1(rs1),
   .IMM_op(IMM_op),
   .rs2(rs2),
   .op_immed23(immed23),
   .op_immed22(immed22),
   .op_immed17(immed17),
   .op_immed16(immed16),
   .cond(cond)
);

interruptcontrol interrupt_control (
        .clock(clk),
        .reset(rst),
        .IE(IE),
        .EA(EA),
        .SPI_req(SPI_req),
        .ctrl_out(ctrl_out),
        .ISR_vec(isr_vec),
        .ISR_req(ISR_req)
        );

mux2_1 muxRegfile(
    .rs1(rs1),
    .rdst(rdst),
    .ctrl_out(ctrl_out),
    .dst(dst)
    );

mux2_1iRegFile mux2_1iRegFile(
    .result(result),
    .immed22(immed22),
    .ctrl_out(ctrl_out),
    .mem_result(mem_result),
    .in_data(finresult)
    );

 memoryHandler datamemoryHandler(
    .clock(clk),
    .finresult(finresult),
    .ctrl_out(ctrl_out),
    .immed22(immed22),
    .mem_outL(mem_outL),
    .mem_outH(mem_outH),
    .restomem1 (restomem1),
    .restomem2 (restomem2),
    .d_addrL(d_addrL),
    .d_addrH(d_addrH)z,
    .mem_result(mem_result)
    );
```

Listing 7: Datapath modules instantiation

As presented on lhe listing, there was the need of two multiplexers in order to keep each module *Comme il faut* and to have a nice autogenerated RTL design, as presented on figure 14.

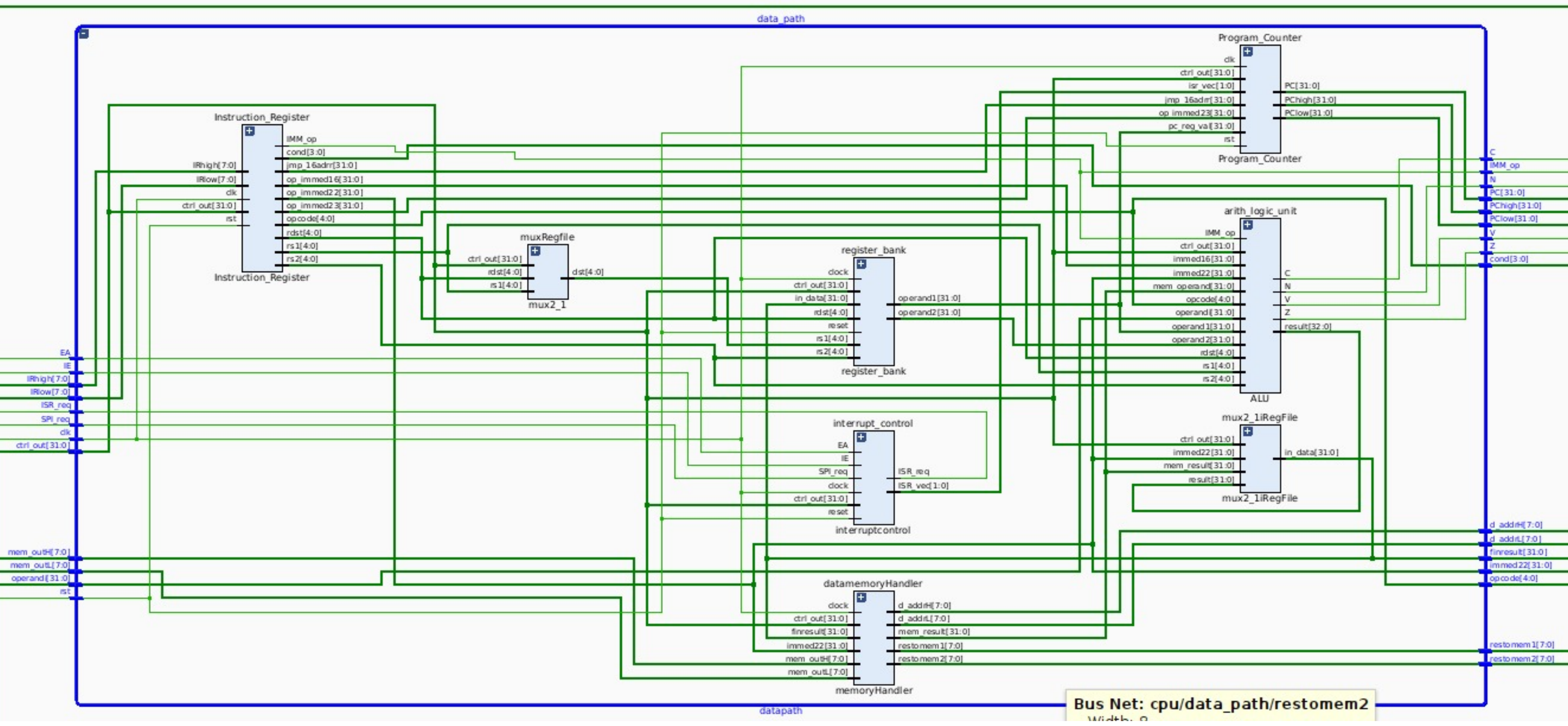


Figure 14: Datapath RTL design

5.4.1 Instruction Register

In this module, all the variables needed to proper functioning were created.

```
module Instruction_Register(
    input rst,
    input clk,
    input [31:0] ctrl_out,
    input [7:0] IRlow,
    input [7:0] IRhigh,
    output [31:0] jmp_16adrr,

    output [4:0] opcode,
    output [4:0] rdst,
    output [4:0] rs1,
    output IMM_op,
    output [4:0] rs2,
    output [31:0] op_immed23,
    output [31:0] op_immed22,
    output [31:0] op_immed17,
    output [31:0] op_immed16,
    output [3:0] cond
    );
```

Listing 8: Instruction Register Definition

After, there is a logic implementation needed to handle the byte adressable Instruction memory. This happens in a synchronous process and with the help of an auxiliary shadow register ir16.

```
    always @(negedge clk)
    begin
        if(rst)
        begin
            IR<=0;
        end
        else if(ctrl_out[`p_IRL1])
        begin
           ir_16 <= {IRlow, IRhigh};
        end
        else if (ctrl_out[`p_IRL2])
        begin
            IR <= {ir_16,IRlow, IRhigh};
        end
    end
```

Listing 9: Instruction Register Handler

After the logic is done and the IR is ready, it is segmented into all the parts that will go into other modules. Note that since this happens assynchronously, the opcode is always sent of the first fetch cycle, which optmizes the VesPA.

```
    assign opcode = IR[31:27];
    assign rdst = IR[26:22];
    assign rs1 = IR[21:17];
    assign IMM_op = IR[16];
    assign rs2 = IR[15:11];
    assign immed23 = IR[22:0];
    assign immed22 = IR[21:0];
    assign immed17 = IR[16:0];
    assign immed16 = IR[15:0];
    assign cond = IR [26:23];
```

Listing 10: Instruction Register Decoding

Lastly, the immediates are sign extended to properly match the VesPA specification.

```
    assign jmp_16adrr = {{16 {IR[15]}},IR[15:0]};
    assign op_immed16 = {{16{IR[15]}}, immed16};
    assign op_immed17 = {{15{IR[16]}}, immed17};
    assign op_immed22 = {{9{IR[21]}}, immed22};
    assign op_immed23 = {{8{IR[22]}}, immed23};
```

Listing 11: Instruction Register Immediate Sign Extensions

5.4.2 Program Counter

The first part of the programcounter.v is the defition of all the variables needed for the implementation.

```
module Program_Counter(
    input clk,
    input rst,
    input [31:0]op_immed23,
    input [31:0]jmp_16adrr,
    input [31:0]pc_reg_val,
    input [31:0]ctrl_out,
    input [1:0]isr_vec,

    output reg [31:0] PC,
    output reg [31:0] PClow,
    output reg [31:0] PChigh
    );
```

Listing 12: Program Counter Definition

In the following listing there is all the logic behind the program counter and the methods the program counter uses to answer interrupt requests from the CU.

```
    begin
        if (rst)
        begin
            PC = 0;
            PChigh = 0;
            PClow = 1;
        end
        else if(ctrl_out[`p_PCisr] == 1'b1)
        begin
            PC_isr_ret <= PC;     // defines the ISR return
                                 //value when RETI is used.
            if(isr_vec[0] == 1) begin
                PC <= 32'd68; //ISR value assignment
            end
            if (isr_vec[1] == 1)
            begin
                PC <= 32'd76; //ISR value assignment
            end
            PChigh <= PC;
            PClow <= PC +1;
        end
        else if (ctrl_out[`p_reti] == 1'b1)
            begin
                PC <= PC_isr_ret; // RETI is used.
                PChigh <= PC;
                PClow <=  PC +1;
        end
        else if (ctrl_out[16] == 1'b1)
            begin
                PC = PC + 2;
                PChigh = PC;
                PClow = PC +1;
            end
        else if (ctrl_out[`p_PCload] == 1'b1)
        begin
             if(ctrl_out[`p_brxen])
            begin
                 PC <= PC + op_immed23 - 3'b100;
                 PChigh <= PC;
                 PClow <= PC +1;
            end
            else if (ctrl_out[`p_jmpen] == 1'b1)
            begin
                PC <= pc_reg_val + jmp_16adrr;
                PChigh <= PC;
                PClow <= PC +1;
            end
        end
    end
```

Listing 13: PC Logic Implementation

5.4.3 Arithmetic and Logical Unit

In ALU, as in default verilog implementation, first the module and its variables was defined.The interface provided by the ALU module is represented in the following listing.

```
    module ALU(
    input clock,
    input reset,
    input [4:0]opcode,
    input [4:0]rdst,
    input [4:0] rs1,
    input IMM_op,
    input [4:0] rs2,
    input [31:0] immed23,
    input [31:0] immed22,
    input [31:0] immed17,
    input [31:0] immed16,
    input  [31:0]  operand1 ,
    input  [31:0]  operand2 ,
    input  [31:0]  mem_operand ,

    input  [31:0] ctrl_out ,

    output  [32:0]  result ,

    output reg  C ,
    output reg  Z ,
    output reg  N ,
    output reg  V
    );
```

Listing 14: ALU module interface

Then the ALU full assynchronous logic was implemented for better CPU optimization and velocity and instant reaction to changes in the inputs.

```
assign result = (ctrl_out[`p_add] && ~IMM_op) ?
(operand1 + operand2) : (ctrl_out[`p_add] &&
                 IMM_op) ? (operand1 + immed16) :
                 (ctrl_out[`p_sub] && ~IMM_op) ?
                 (operand1 - operand2) : (ctrl_out[`p_sub] &&
                 IMM_op) ? (operand1 - immed16) :
                 (ctrl_out[`p_and] && ~IMM_op) ? (operand1 &&
   operand2) : (ctrl_out[`p_and] &&
                  IMM_op) ? (operand1 & immed16) :
                  (ctrl_out[`p_or] && ~IMM_op) ? (operand1 |
  operand2) : (ctrl_out[`p_or] && IMM_op)
                   ? (operand1 | immed16) :
                  (ctrl_out[`p_xor] && ~IMM_op) ? (operand1 ^
   operand2) : (ctrl_out[`p_xor] &&
                  IMM_op) ? (operand1 ^ immed16) :
                  (ctrl_out[`p_not]) ? ~operand1 :
                  (ctrl_out[`p_cmp]) ? (operand1 - operand2)
  :
                  (ctrl_out[`p_ld2]) ? mem_operand :
                  (ctrl_out[`p_st]) ? operand1 :
                  (ctrl_out[`p_sti]) ? operandi : 32'hZZZ;
```

Listing 15: ALU Operation Logic

5.4.4 Register File IP integration

The register filed was implemented in two BRam blocks, following the design made in 8 and then was handled by a separate module inside the datapath. The register file was created in a Block Designed "*Regfile.bd*".

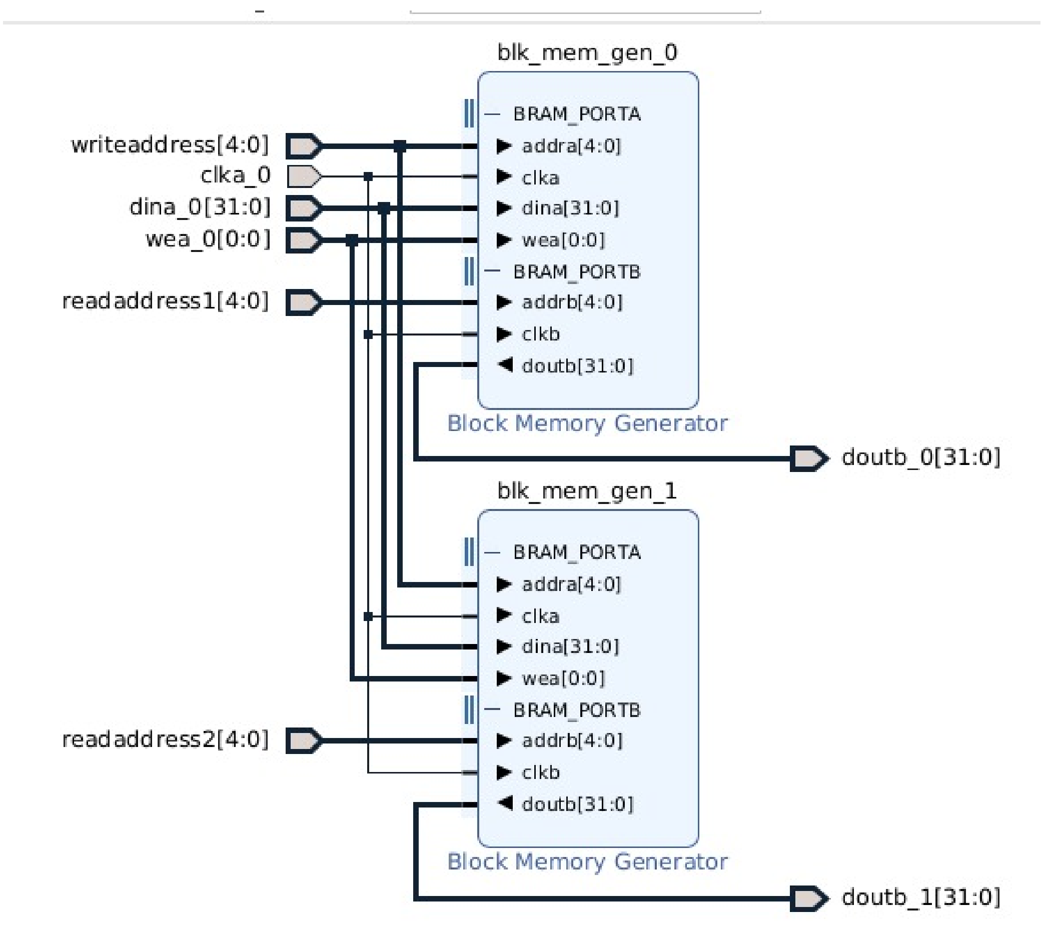


Figure 15: Register File Block Design

In order to connect the Register File with the datapath, it was created a *regfilehandler.v*. This module allowed for a seamless integration of the IP Brams inside Block Design with VesPA. The handler has it's interface define as presented on the following listing.

```
module register_bank(

    input clock,
    input reset,
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] rdst,
    input [31:0] in_data,
    input [31:0]ctrl_out,

    output [31:0] operand1,
    output [31:0] operand2
```

```
        );
```

Listing 16: Instruction Register Handler

5.5 Interrupt Controller

The interrupt controller was implemented following the design present on figure 10.First, it has the following interface.

```
    module interruptcontrol(
    input clock,
    input reset,
    input IE,
    input EA,
    input SPI_req,
    input [31:0]ctrl_out,
    output reg [1:0]ISR_vec,
    output reg ISR_req
    );
```

Listing 17: Interrupt Controller interface

The logic on the interrupt controller allows for the two interruptions it was designed to have, the SPI and the External Interrupt. This is done with synchronous logic on the negative edge of the clock.

```
always @(negedge clock)
begin
    if (reset)
    begin
      ISR_req <= 1'b0;
    end
    else if(ctrl_out[`p_PCisr] && ISR_vec[0])
    begin
        ISR_req <= 1'b0;
        ISR_vec[0] <= 1'b0;
    end
    else if (ctrl_out[`p_PCisr] && ISR_vec[1])
    begin
        ISR_req <= 1'b0;
        ISR_vec[1] <= 1'b0;
    end

    else if(IE == 1'b1 && (EA == 1'b1))
    begin
        ISR_vec[0] <= 1'b1;
        ISR_req <= 1'b1;
    end
    else if((SPI_req == 1'b1) && (EA == 1'b1))
    begin
        ISR_vec[1] <= 1'b1;
        ISR_req <= 1'b1;
    end
end
```

Listing 18: Interrupt Controller logic

5.6 Memory

Following the design on figure 5, the implementation proceeded to do an Harvard Memory on a Block Design.

5.6.1 Data Memory

According to figure 6, one True RAM Dual Port BRam blocks were generated in a Block Design as present of figure 16.

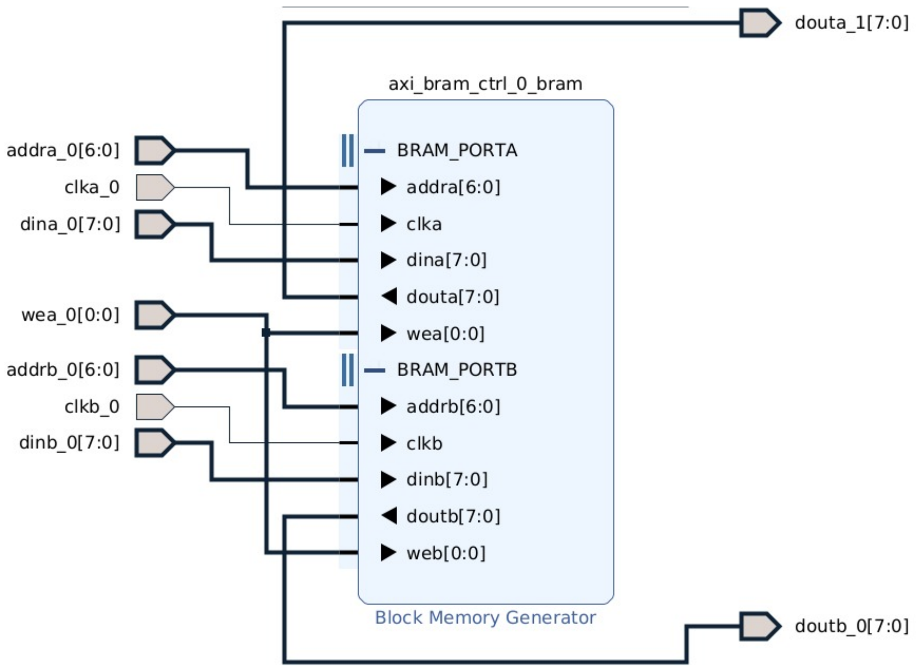
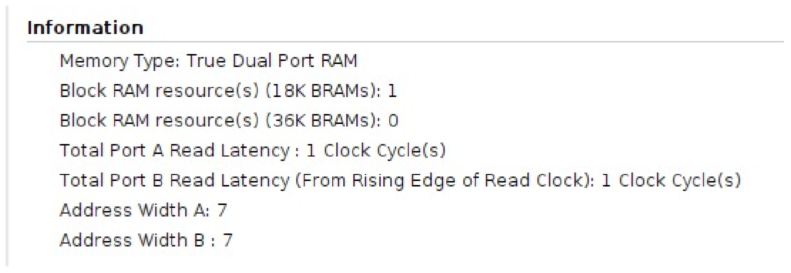


Figure 16: Data Memory Block Design

This IP memory block has some specific configurations to properly work as desired on a VesPA implementation. It is working as a True Dual Port RAM Block and has 1 cycle of latency as presented on figure 17.



Information

Memory Type: True Dual Port RAM
Block RAM resource(s) (18K BRAMs): 1
Block RAM resource(s) (36K BRAMs): 0
Total Port A Read Latency : 1 Clock Cycle(s)
Total Port B Read Latency (From Rising Edge of Read Clock): 1 Clock Cycle(s)
Address Width A: 7
Address Width B : 7

Figure 17: Data Memory Configuration

5.6.2 Instruction Memory

The instruction memory also followed the design of the figure 7, with a seamless integration of a True Dual Port RAM Block. This instruction memory was configured as a RAM block to enable a future upgrade that would allow software code memory reset.

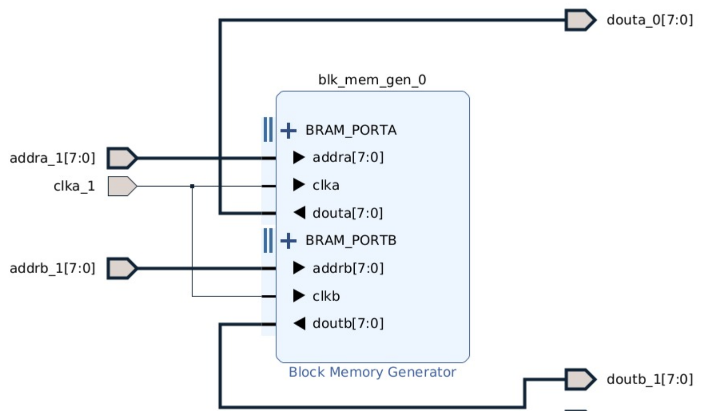
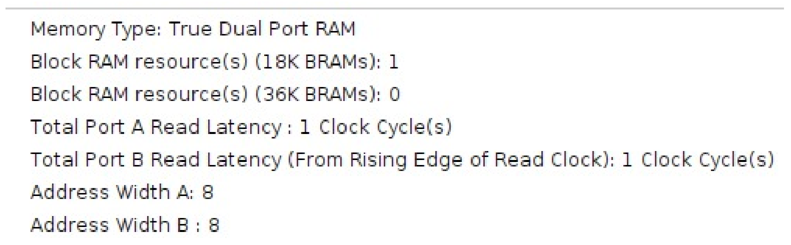


Figure 18: Instruction Memory Block Design

The ip memory configuration summary can be observed in the following figure 19.



Memory Type: True Dual Port RAM
Block RAM resource(s) (18K BRAMs): 1
Block RAM resource(s) (36K BRAMs): 0
Total Port A Read Latency : 1 Clock Cycle(s)
Total Port B Read Latency (From Rising Edge of Read Clock): 1 Clock Cycle(s)
Address Width A: 8
Address Width B : 8

Figure 19: Instruction Memory Configuration

5.7 SPI

The SPI implementation followed the design on figure 11.

```
    module spi_slave(
    input i_scl,          // Master Clock (arduino uno)
    input i_clk,
    input i_rst,
    input i_mosi,         //  (data output from master)
    input i_cs,           // Chip Slave Select
    output o_miso,        //  (data output from slave)
    output[31:0] o_data, // Output data
    output o_sync,         // Output CS synchronized
    output spi_req
    );
```

Listing 19: SPI slave interface

To implement the SPI there is a State machine functioning on master's source clock.

```
        // SPI State Machine
    always @(posedge i_scl or posedge i_rst)
    begin
        if (i_rst == 1'b1) begin
            data <= 8'h0;
        end
        else if (~o_sync) begin
            data <= {data[30:0], i_mosi};     // Left shift
        end
    end
```

Listing 20: SPI FSM

5.7.1 Clock Domain Crossing

To ensure a stable connection between master and slave there is Clock Domain Crossing. This implementation was focused on ensuring the Chip Select signal was properly received without metainstability.

```
    always @(posedge i_clk or posedge i_rst)
    begin
        if (i_rst) begin
            cdc <= 2'b00;
        end
        else begin
            cdc <= {cdc[0], i_signal};
        end
    end
```

Listing 21: CDC Code implementation

5.8 External Interrupt

The external interrupt was set to a button on the constraints.xml file and was declared as a CPU input as seen on CPU figure 12.

5.8.1 Debounce

Since that the VesPA is using a button for the EI there is a need for a debounce.v module.

```
    module Button_debounce(
    input clock,
    input reset,
    input button,
    output reg button_out
);
```

Listing 22: Debounce Verilog interface

Subsequently, the module implementation was a basic debounce control where it counts up to the minimum number that is equivalent to the desired time in seconds. Since the counter goes up to 655535, and VesPA's internal clock frequency is 100MHZ, that is approximately equivalent to 0.66 seconds.

```
  always @(posedge clock) begin
   if(reset) begin
        button_pressed <= 0;
        button_out <= 0;
        debounce_counter <= 0;
    end
    if (button && !button_pressed) begin
      debounce_counter <= debounce_counter + 1;

      if (debounce_counter == 32'h00FFFFFF) begin
        counter <= 1'b1;
        button_pressed <= 1;
        debounce_counter <= 0;
      end
    end
    else if (counter != 0) begin
      counter <= counter - 1;
    end
    else begin
      button_out <= 0;
      button_pressed <= 0;
      debounce_counter <= 0;
    end
    button_out <= (counter!=0);
  end
```

```
endmodule
```

Listing 23: Debounce Verilog implementation

5.9 Top Layer RTL Verification

To correctly assure all the connections between modules were correctly made, the RTL Top layer gives a general yet precise view of all external connections to the CPU.

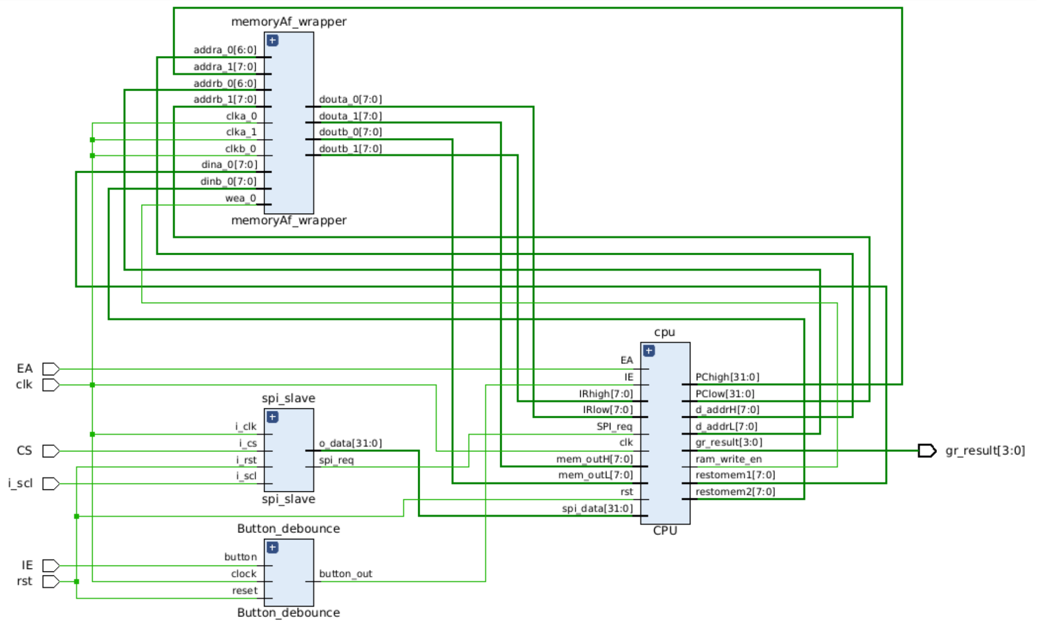


Figure 20: RTL Top layer

6 Verification

6.1 ALU operations

In order to verify the ALU several tests were done on both arithmetic and logical operations.

6.1.1 Arithmetic Operations

In order to test the arithmetic operations, we proceeded to do the following tests.

```
00001000 11000011 00000000 00001111
00001000 01000110 00011000 00000000
00010000 10001001 11111111 11111111
```

Listing 24: ALU test instructions.

The first instruction is an ADD between register 3 and the imeddiate 1111, with destination register being on register 1. The second is an ADD between register 3 and 0 with the destination register being register 3. The third is a sub between register 1 and the imeddiate 1111 1111 storing the destination value on register 2.

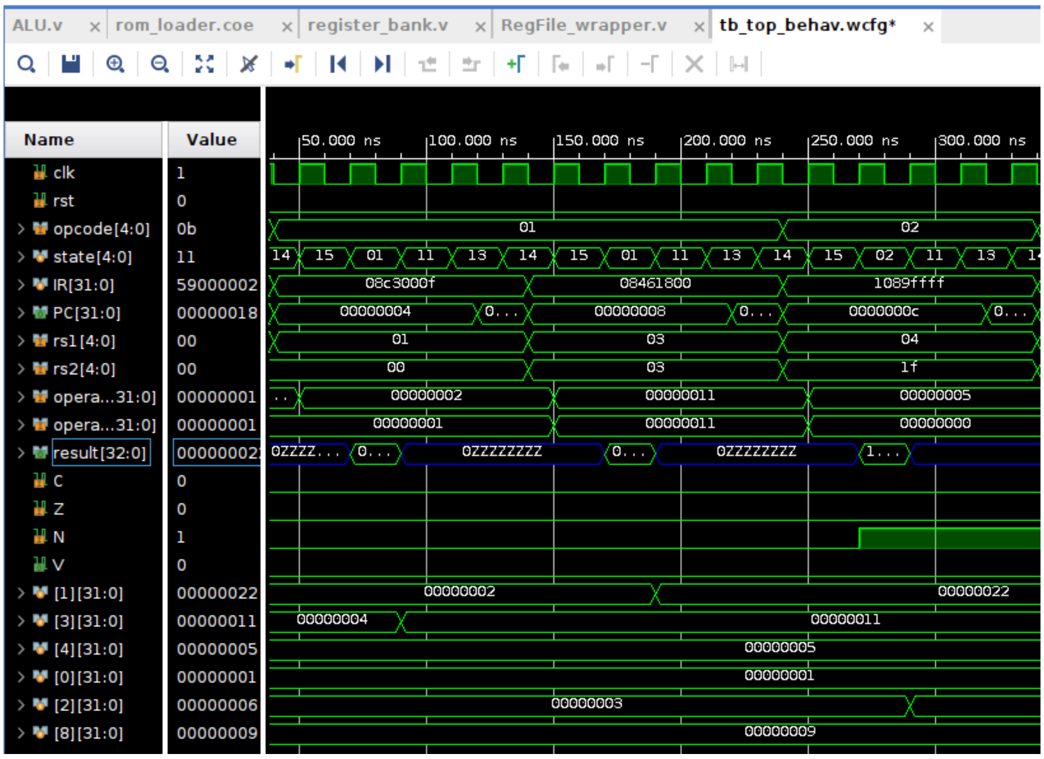


Figure 21: Logical Instruction Verification Test

As verified on figure 21, every instruction is working properly and the condition flags are working properly as seen on the sub instruction. Note that the register File as delay because of it's implementation with BRAM blocks.

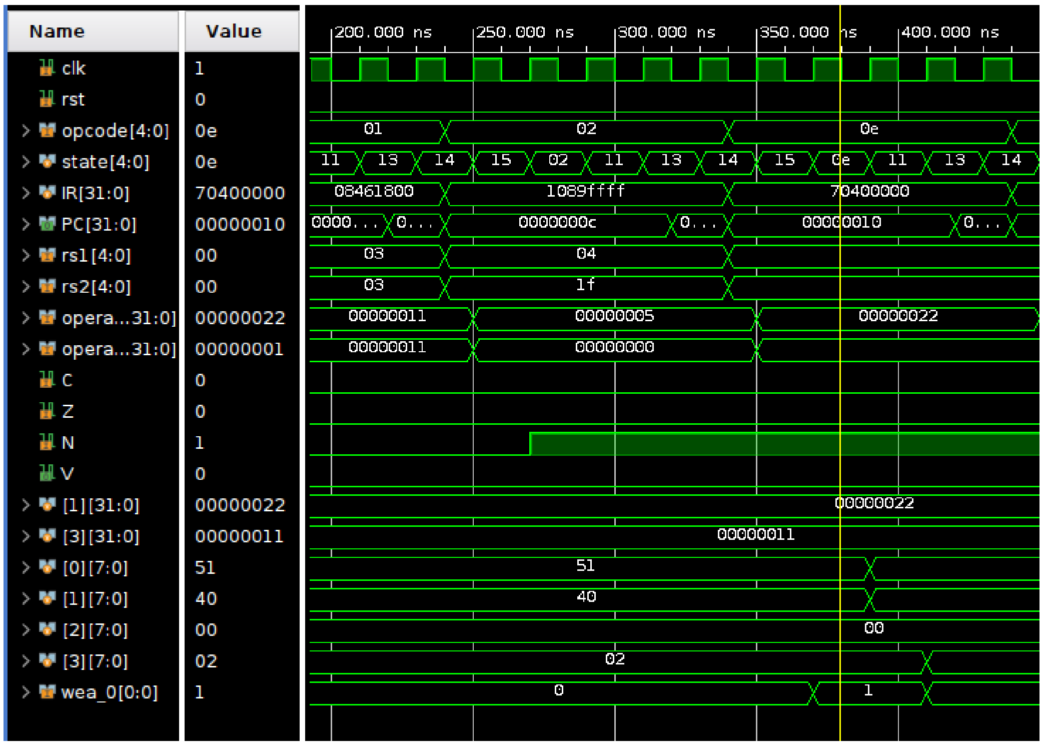
6.2 Data Memory transfer operations

In order to verify the memory operation several tests were done on both store and load instructions.

```
01110000 01000000 00000000 00000000
01011000 11000000 00000000 00000000
```

Listing 25: Memory Verification instruction code

The first instruction present on this verification is a store from register 1 to memory position 0. The second instruction is a load from memory position 0 to register 3.



Name	Value
clk	1
rst	0
opcode[4:0]	0e
state[4:0]	0e
IR[31:0]	70400000
PC[31:0]	00000010
rs1[4:0]	00
rs2[4:0]	00
opera...31:0]	00000022
opera...31:0]	00000001
C	0
Z	0
N	1
V	0
[1][31:0]	00000022
[3][31:0]	00000011
[0][7:0]	51
[1][7:0]	40
[2][7:0]	00
[3][7:0]	02
wea_0[0:0]	1

Figure 22: Memory operations verification

The final test is presented on figure 22 for a better understading of RegFile and memory delays, as well as write enable timings. The first values [1] and [3] represent register, and the following from [0] to [3] represent data memory.

6.3 Jump and Branch operations

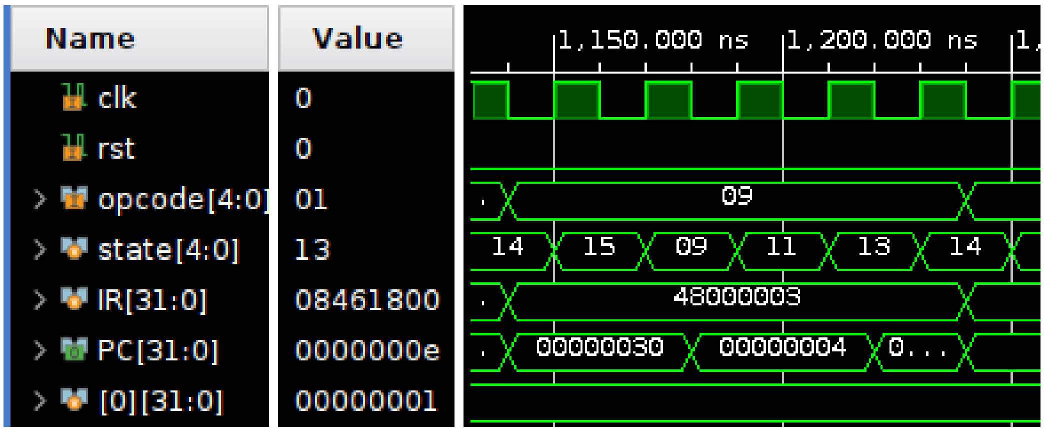
6.3.1 Jump

Several jump and branch operations were tested. For this matter the tests presented on figure 23 and ??erfectly exemplify the correct functioning of the VesPA CPU.

```
01001000 00000000 00000000 00000011
```

Listing 26: Jump Instruction Test

The Jump instruction used to verify simply jumped to reg[1]+immediate value 0011, which represents 3+1 and assigns value 4 to PC.



Name	Value
clk	0
rst	0
opcode[4:0]	01
state[4:0]	13
IR[31:0]	08461800
PC[31:0]	0000000e
[0][31:0]	00000001

Figure 23: Jump Verification

6.3.2 Branch

For the branch test, all the branches are tested and verified, both on positive and negative flags.

```
00001000 11000011 11111111 11111111
01000100 10000000 00000000 00000010
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001
00000000 00000000 00000000 00000011
```

Listing 27: Branch on Carry set test operation

The code on the listing presents a ADD operation to set the Carry, followed by a branch on carry set(BCS) operation, that on negative carry would simply follow for the NOP ending on 00 or, on set carry, would jump to the NOP instruction ending on 01.

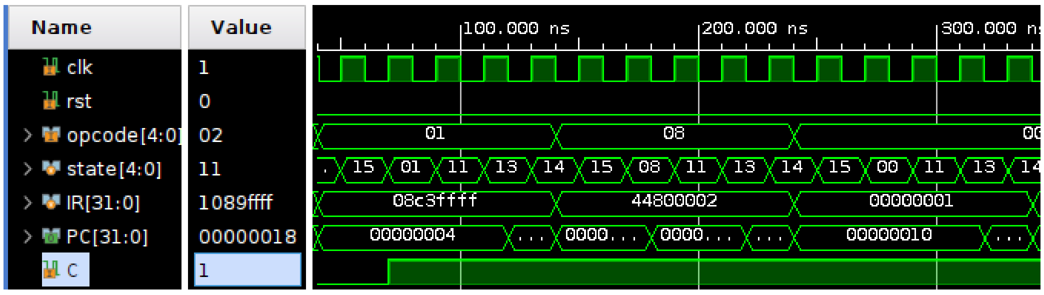


Figure 24: Branch Verification

As seen on 24, the Program Counter correctly updates during the branch state due to the Carry being set verifying the correct functioning of the VesPA processor.

6.4 External Interrupt

In order to verify the external interrupt, there is a instruction memory code responding to ISR finishing on a RETI instruction that returns to the initial PC value.

```
00001000 01000011 00000000 00000001
11011000 00000000 00000000 00000000
00000000 00000000 00000000 00000001
11100000 00000000 00000000 00000000
01001000 00000000 00000000 00000011
```

Listing 28: External Interrupt ISR

This way, there are 5 instructions happening and then the PC is returning to it's original value. Inside the ISR, there will be and ADD, a Store, a NOP and an indexed store.

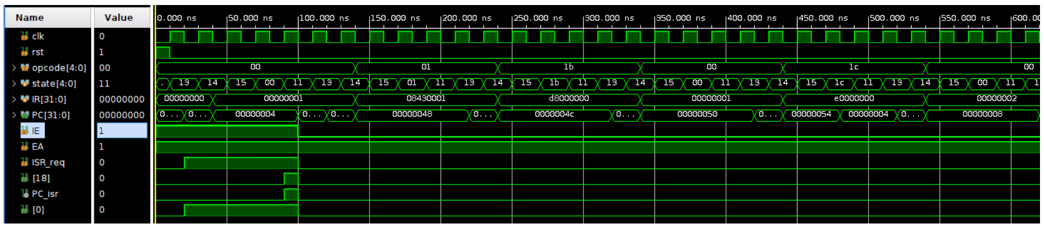


Figure 25: External interrupt verification

As visible on figure 25, all the instructions correctly execute, and when the RETI is called the code goes back to the initial PC value, which is decimal 4. It is also possible to see that the interrupt occurs on state Start and that the control signals all go to 0 after the ISR is being executed.

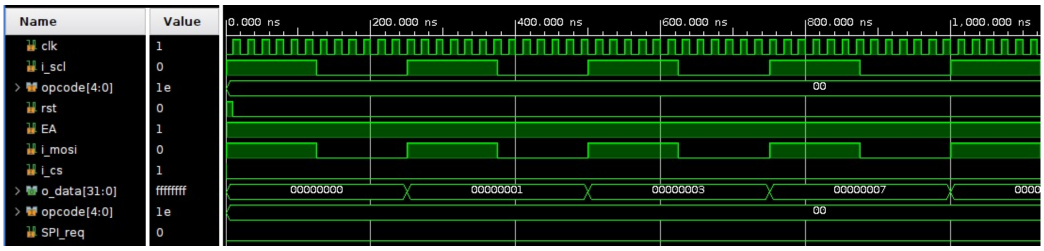


Figure 27: SPI begin transaction Verification

6.5 SPI test

To test SPI, firstly there is a definition on the Master Clock to be 4MHz and the Slave to be 100Mhz. Then, the CS is 0, and with this conditions, we can see that after the transference ocurred the SPI-req for an ISR, is active.

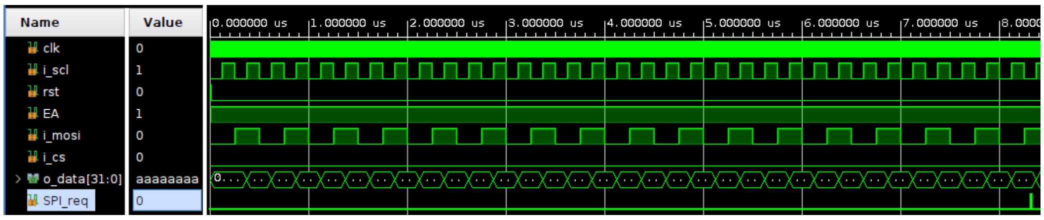


Figure 26: SPI ISR Verification

After these first validation on figure 26, the following test will be on all the variables envolved on the beginning of the transaction.

Finally, analysing the end of the transaction, it is possible to verify that whenever the transaction ends, the final value is assigned on memory position 124 and the variables SCLK and mosi are set to 0, and the Input Chip Select is set to 1, which verifies the operation is finished.

6.5.1 Clock Domain Crossing

The Clock Domain Crossing verification shows the Chip Select being metastabilized on the rising edge of the internal clock.

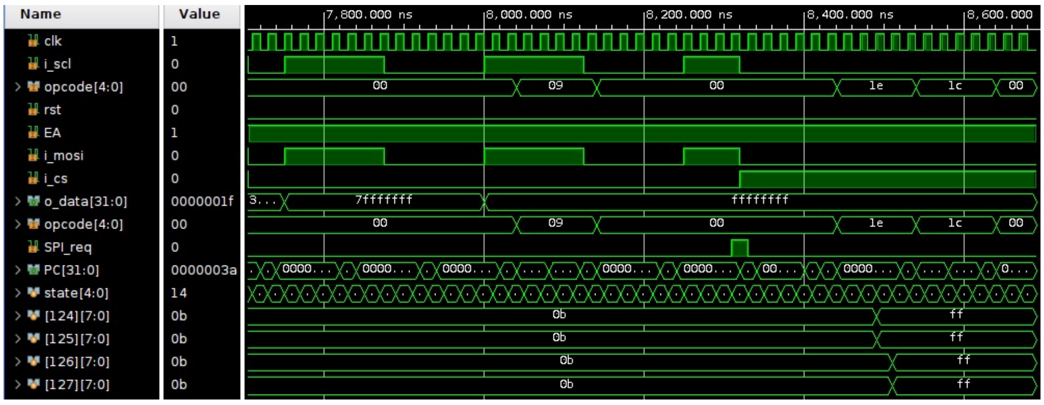


Figure 28: SPI transaction finished

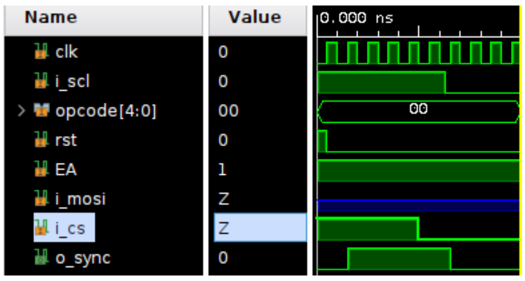


Figure 29: CDC verification

7 Conclusion

The VesPA project has been an illuminating journey that has significantly contributed to our understanding of microprocessor architecture. Throughout the design process, we have encountered various challenges and opportunities that have enriched our knowledge and skills in several key areas.

Firstly, the project emphasized the importance of thorough research and planning in the initial stages of design. Our comprehensive analysis of the Control Unit and Datapath laid a strong foundation for the subsequent phases of development. This approach not only streamlined our workflow but also helped us in making informed decisions, thereby enhancing the overall quality and efficiency of the project.

Secondly, the iterative nature of the design process in VesPA allowed us to continually refine our ideas and solutions. Through multiple cycles of development, testing, and optimizing, we were able to evolve our initial concepts into more robust and effective designs.

Moreover, collaboration and interdisciplinary teamwork played a pivotal role in the success of VesPA. The diverse perspectives and expertise within our small team brought innovative solutions. The experience has highlighted the significance of effective communication and teamwork in achieving complex design objectives.

In addition, the project provided us with practical insights into IP Blocks Designing and use, enhancing our capability to tackle similar challenges in future endeavors.

In conclusion, the VesPA project has not only achieved its intended goals but has also served as a valuable learning experience. The knowledge and skills acquired during this project will undoubtedly be beneficial in our future projects and contribute to our ongoing pursuit of excellence in Embedded Systems.