Marcelo Ribeiro PG54028

Pedro Pereira PG54155

VeSPA
(Very Simple Processor Architecture)

Master's in Industrial Eletronic and Computer Engineering

Professor:
Dr. Adriano José Conceição Tavares

Contents

List of Figures

1 Introduction

This project focuses on the design and implementation of a VeSPA (Very Simple Processor Architecture) on a FPGA(Field Programmable Gate Array), more specifically, the Zybo-z7. Our processor's core is designed to be single-cycle with 32-bit instructions and its based on a byte-addressable Harvard architecture, featuring distinct and independent memories for instructions and data. This architectural choice allows for simultaneous access to both instruction and data operations, improving the overall performance and efficiency of the processor, but also requires the use of more hardware.

1.1 Objectives

1.1.1 Design a robust datapath and control unit.

1.1.2 Implement all the instructions required by the VeSPA's Instruction Set.

1.1.3 Develop an interrupt controller.

1.1.4 Integrate a PS/2 keyboard controller for user input.

2 Methodology

2.1 Instruction Set Architecture (ISA)

There are 4 types of categories in terms of their instructions, arithmetic operations, logical operations, control operations and data transfer operations, and miscellaneous instructions, of which there are a total of 30 instructions. Of the 30, there are only 16 instructions that could be distinguished using the opcode field of only 4 bits, but 5 bits were still used in this field with a view to future expansion of the microprocessor.

Instruction	OPCODE(decimal)	OPCODE(binary)
NOP	0	00000
ADD	1	00001
SUB	2	00010
OR	3	00011
AND	4	00100
NOT	5	00101
XOR	6	00110
CMP	7	00111
BXX	8	01000
JMP	9	01001
JMPL	9	01001
LD	10	01010
LDI	11	01011
LDX	12	01100
ST	13	01101
STX	14	01110
HLT	31	11111

Figure 1: Opcode List

2.1.1 Instructions Format

This architecture features 32-bit instructions, comprising 30 for arithmetic/logical operations, 14 conditional jumps, 2 unconditional jumps, and 2 miscellaneous operations. Each instruction has a specific operation code, optimizing encoding. This design ensures efficiency in executing complex programs while maintaining simplicity in processor processing. In the table below, (table 1) it's possible to understand deeper the instruction format of all the types of instructions.

Instruction Format						
Arithmetic and Logic	31-27	26-22	21-17	16	15-11	10-0
	Opcode	Destiny Register (rdst)	Register 1 (rs1)	0	Register 2 (rs2)	_
				1	Immediate 16 bits	
Conditional Branchs	31-27	26-23	22-0			
	Opcode	Condition	Immediate 23 bits			
Jumps	31-27	26-22	21-17	16	15-0	
	Opcode	XXXX	Register 1 (rs1)	0	Immediate 16 bits	
				1		
Data Transfer	31-21	26-22	21-17	16-0		
	Opcode	Destiny Register (rdst)	Register 1 (rs1)	Immediate 17 bits		
			Immediate 22 bits			
		Store Register (rst)	Register 1 (rs1)	Immediate 17 bits		
			Immediate 22 bits			
Miscellaneous	31-27	XXXX	XXXX	XXXX	XXXX	XXXX
	Opcode					
		_	_	_	_	_

Figure 2: Instructions Format

2.2 General Overview

In figure 3 we can see the real Top layer of our system where we manage to see the the CPU connection with the RAM (which is only represented as external for a better visualization, although the RAM is still totally connected to the CPU's datapath) and the connection of this modules to the clock gating module, that way we improved the efficiency of our system, particularly in terms of power consumption. The primary purpose of clock gating is to selectively disable the clock signal to specific components or portions of the circuit when they are not actively processing data.

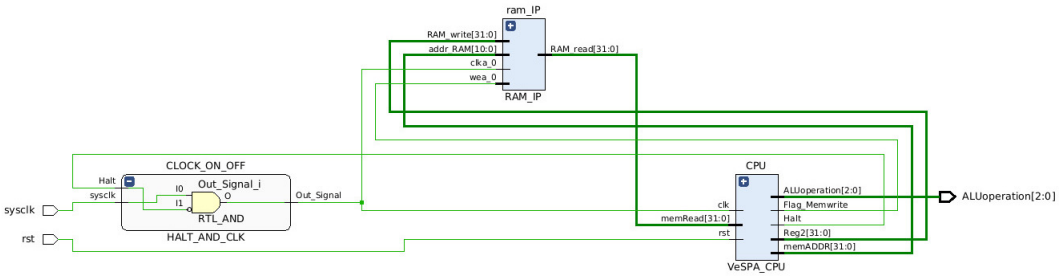


Figure 3: General Overview

2.3 Datapath Design

The Datapath is the component responsible for Fetching, Decoding and executing instructions. The architecture prioritizes simplicity and speed, enabling the processor to perform operations with minimal latency.

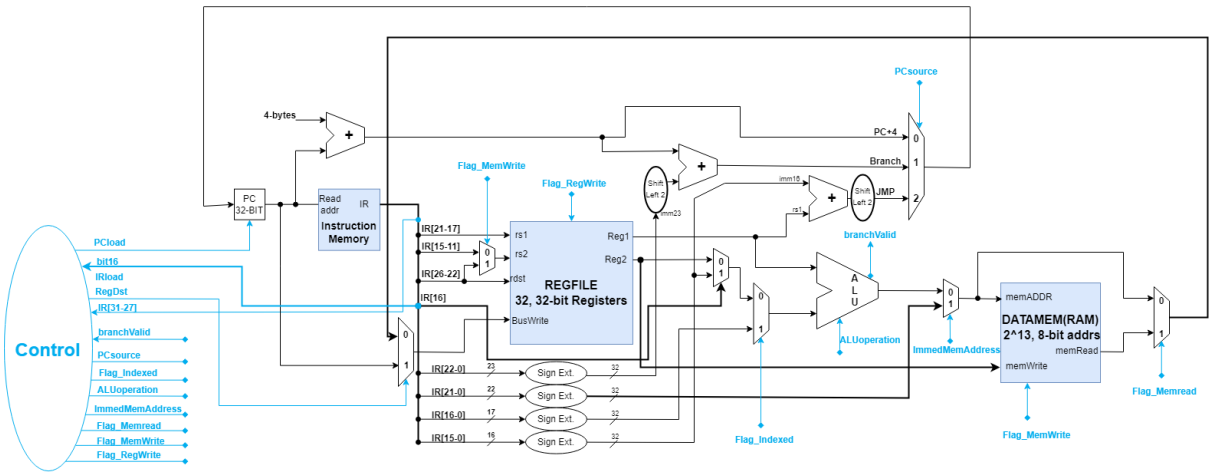


Figure 4: Datapath

In the figure 4 we can clearly see the Harvard architecture with separated code and data memories. For these we have used have used a depth of 4096(4Kb) bytes and 8192 bytes(8Kb), respectively. Further details on memory implementation will be provided.
It is also visible the processor's Register File that contain 32 registers with 32 bits each.

2.4 Control unit Design

The control unit coordinates, mainly, the Datapath's input signals such as multiplexers' or memory blocks control signlas and and ensures the proper execution of instructions.

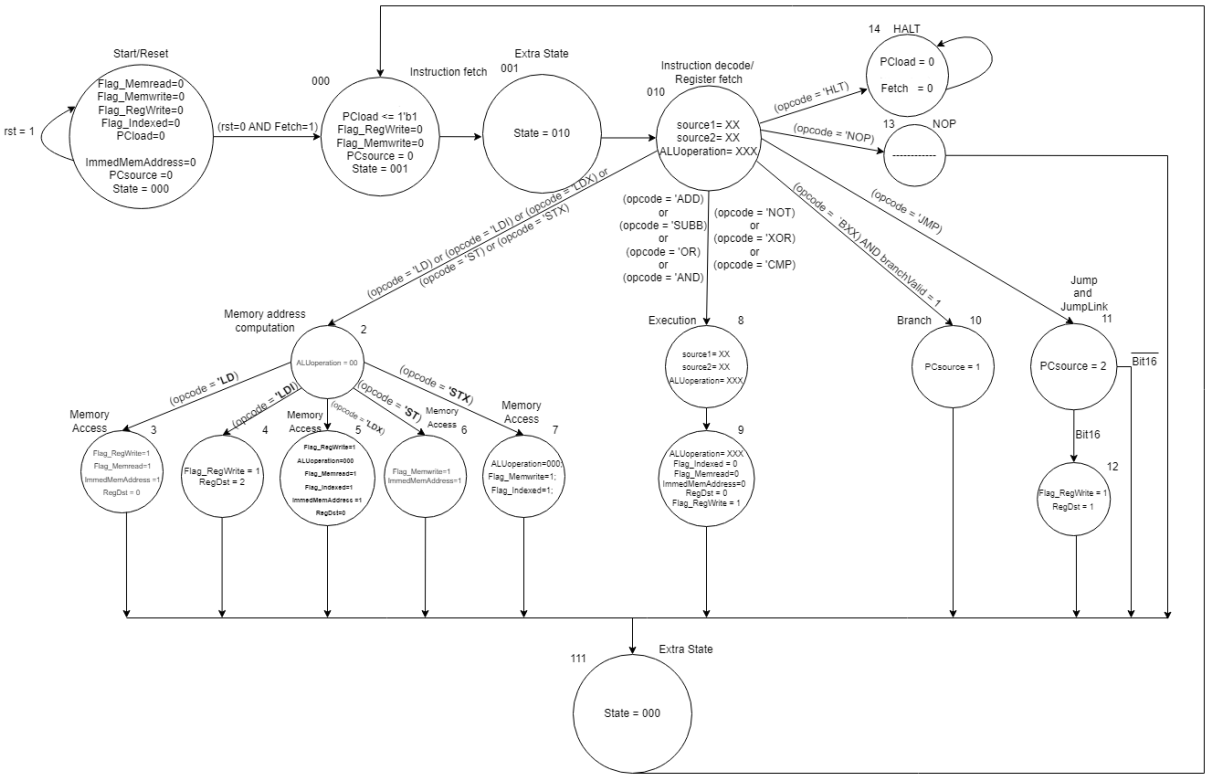


Figure 5: Control unit

2.5 Interrupt Controller

An interrupt controller has been integrated to manage at least two interrupts in order to test their priority and mask/enabling flags. Initially, they are triggered by pressing a specific embedded button on the FPGA, and subsequently, by pressing a specific key on the external keyboard.

2.6 PS/2 Keyboard Controller

A PS/2 keyboard controller has been implemented to facilitate user input. This enables a wider interaction with the processor, making it suitable for a broader range of applications, mainly if working together with the interrupt controller as mentioned before.

3 Implementation

3.1 VeSPA's Core

In order to implement the VeSPA's core we must implement 5 really important components.

3.1.1 The Instruction Memory

As we have a byte-addressable architecture but the word size is 32 bits, we have implemented our instruction memory using 4 BRAMs and initializing the instructions by adding a .coe file in the BRAM options. By receiving the Program Counter on the first one, we get the the first byte of the instruction in the address PC+0 and then we add "1" to the Program counter so the next BRAM receives the the address PC+1 and so on until getting to the 4th BRAM and concatenate all 4 bytes into the Instruction Register. As a picture is worth more than a thousand words, in the figure 6 is a visual explanation of our implementation.

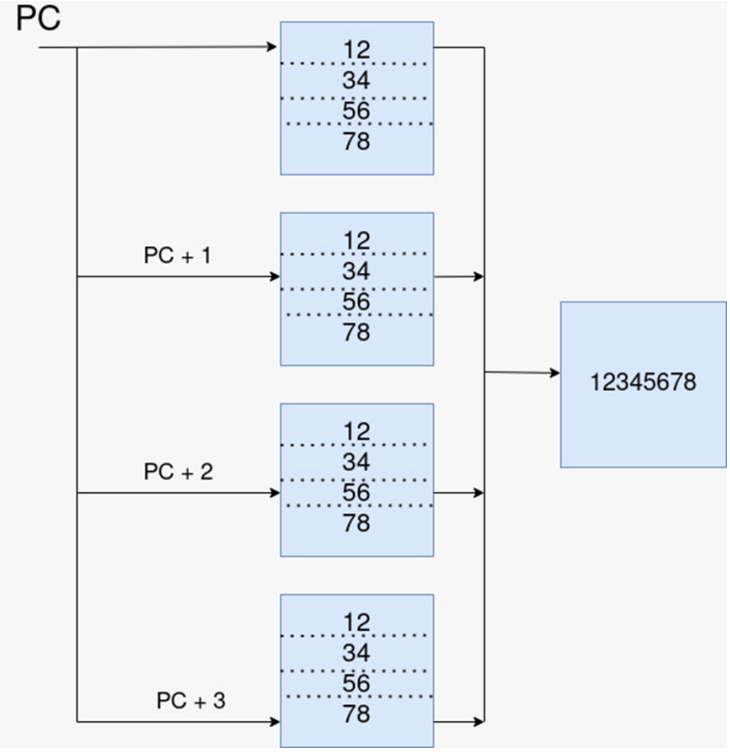


Figure 6: Instruction Memory Diagram

Using the Vivado's IP(Intellectual Propriety) "Block Memory Generator" (i.e. BRAM), we've managed to create a synthesizeable memory for Instruction memory, register file and also for the RAM. The figure 7 is our real implementation of the instruction memory using Vivado's IP:

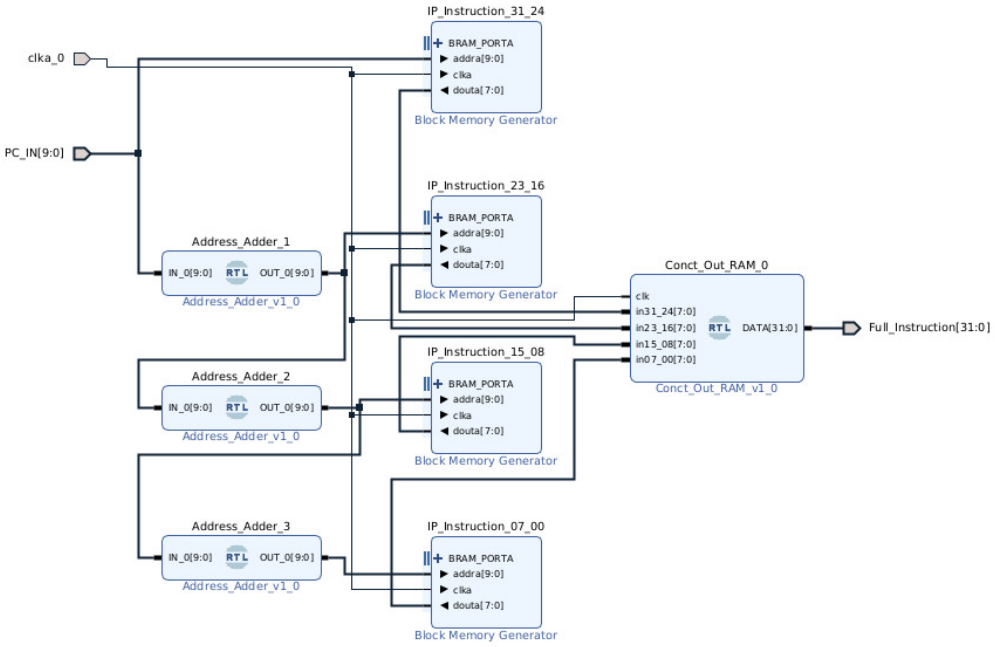


Figure 7: Instruction Memory IP

This process of reading the address and exporting the full instruction takes 1 clock cycle to be completed, so since we can't eliminate this latency we have added an extra cycle to the Fetch, Decode and Execute.

3.1.2 Register File

Our Register File was implemented with 2 BRAMs with exactly the same content at all times. We must have two because there are three addresses involved, "rs1" the address of register 1, "rs2" the address of register 2 and "rdst" the destiny address. Both IPs are "Simple Dual Port RAM" memory type since we only need one output(reading) in each one, since each block receives a different read/write address. The variable "RegWrite" is responsible for enabling or disabling the writing to the register file.

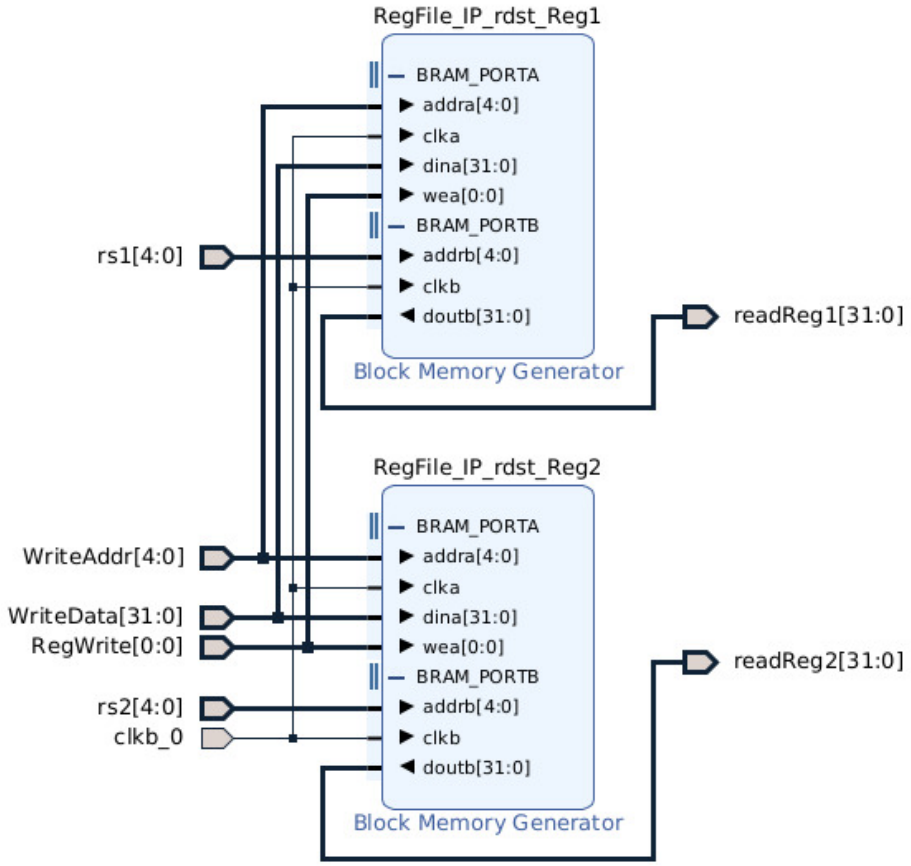


Figure 8: RegisterFile IP

3.1.3 Data Memory(RAM)

Our RAM is somewhat different compared to the instruction memory. In RAM, since we don't have to initialize a '.coe' (coefficient) file with the instructions, the 4 BRAMs don't need to be copied. So now, we split the word into 4 bytes, and on each BRAM, at every address, we store the respective byte of the full 32-bit data. Let's say we need to store 32-bit data at address '0'. The least significant byte will go to the BRAM that contains all the least significant bytes of data at address '0'. In the end, we concatenate the 4 bytes into a single 32-bit word.

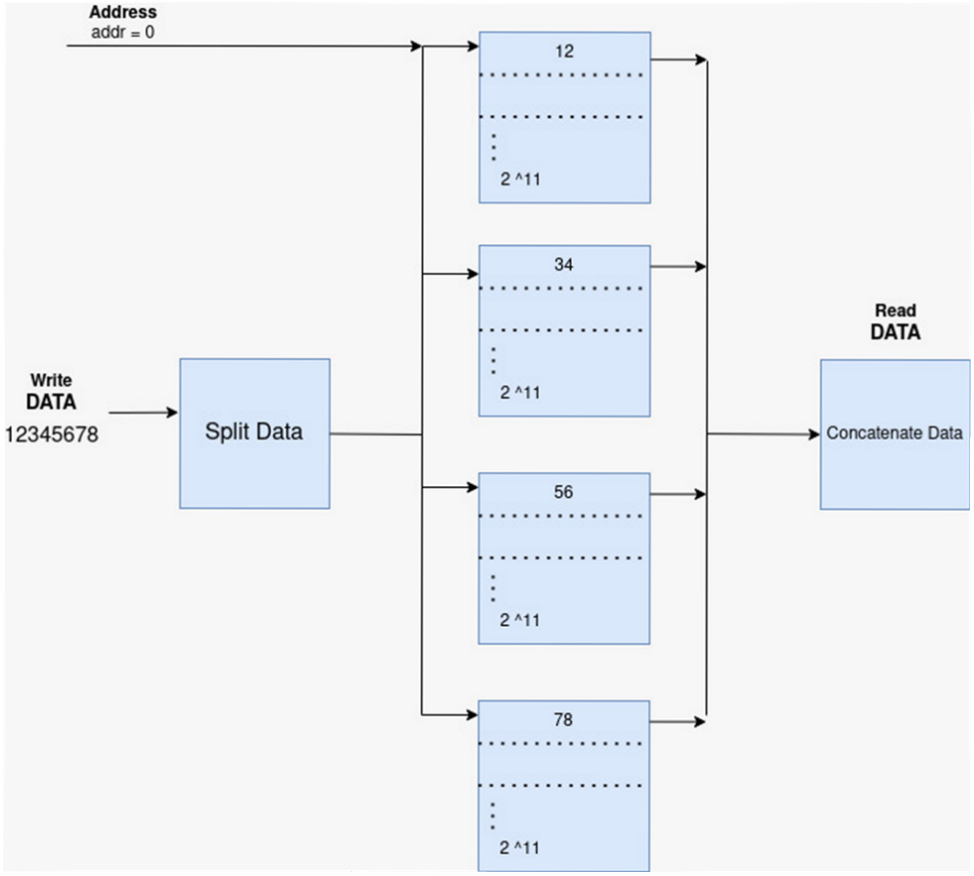


Figure 9: Ram diagram

As we can see in Figure 9 , if we wanted to store and, a while after, load it, in the store operation, the data would be split into 4 bytes at the same address in different blocks. To load it, we read from the same address in different blocks and then concatenate the 4 bytes into a word.

Our implementation of this concept in Vivado using IP's can be observed in figure 10.

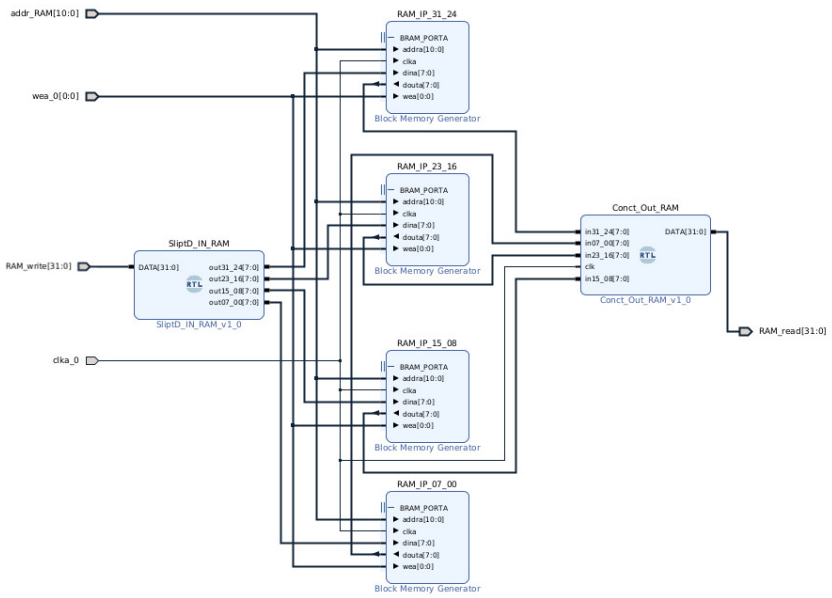


Figure 10: RAM IP

3.1.4 ALU (Arithmetic Logic Unit)

The ALU, at the heart of a CPU, is indispensable for executing mathematical calculations, logical comparisons, and bit-wise operations, playing a vital role in the datapath. Its implementation is fundamentally a selection control structure(switch-case), that exports the result and consequently 4 status flags, Carry, Negative, Zero and Overflow all based in the ALU's two input sources and its the ALU operation given by the instruction(based on the Opcode).

```
always@(*) begin
        case (ALUoperation)
            `ADD_alu: begin
                    result <= source1 + source2;    //ADD
                    subt <= 1'b0;
                    end
            `SUB_alu: begin
                    result <= source1 - source2;    //SUB
                    subt <= 1'b1;
                    end
            `OR_alu: begin
                    result <= source1 | source2;    // OR
                    subt <= 1'b0;
                    end

            `AND_alu: begin
                     result <= source1 & source2;   // AND
                     subt <= 1'b0;
                     end
            `NOT_alu: begin
                    result <= ~source1;             //NOT
                    subt <= 1'b0;
                    end
            `XOR_alu: begin
                    result <= source1 ^ source2;    //XOR
                    subt <= 1'b0;
                    end
            `CMP_alu: begin                         //CMP
                    subt <= 1'b0;
                    end
            `Disable_alu: begin
                    result<=0;
                    end
         endcase

  end

        assign Zero     = (`ARMT_OP) ?( (ALUoperation == `CMP_alu) ? ~((source1[WIDTH-1:0]-source2[WIDTH-1:0])) : ~(|result[WIDTH-1:0]) )
                                     :0;

        assign Carry    = (`ARMT_OP) ? result[WIDTH]
                                     :Carry;
        assign Negative = (`ARMT_OP) ? result[WIDTH-1]  //Verifica o MSB do resultado
                                     :Negative;
        assign Overflow = (`ARMT_OP) ?((source1[31] & source2[31] & ~result[31]) | (~source1[31] & ~source2[31] & result[31])) | ((source1[31] & source2[31] & result[31]) | (~source1[31] & ~source2[31] & ~result[31]))
                                     :Overflow;
```

Figure 11: ALU

3.1.5 Control Unit

And finally but not least, one of the most important peaces of this puzzle that is the CPU core, the control unit, the brain of the Central Processing Unit, generates control signals, coordinates data movement, and ensures proper sequencing, flow of instructions and handles branching. We have implemented it as state machine containing 5 states: Fetch, Fetch2, Decode, Execute, Execute2. The state Execute2 its another extra state added given to the need of an extra cycle when performing a Load type operation(it has to perform a read operation on RAM and a write operation on the register file) we've decided to enable this state to all possible instructions(not only the load) since in a larger scale project it would not be efficient to create a state only for every single instruction that needs it.

```
always@(posedge clk or posedge rst) begin
    if(rst) begin       //aqui só damos reset a estas flags?
        State  <= 3'b000;            //Fetch state
    end
    else begin
        case(State)

            3'b000:begin //Fetch
                State <= 3'b001;
            end
            3'b001:begin //Fetch 2
                State <= 3'b010;
            end

            3'b010:begin     //Decode

                State <= 3'b011;
            end

           3'b011:begin     //Execute
                State <= 3'b111;
            end

            3'b111:begin     //Execute2 due to Load ops

                State <= 3'b000;
            end

        endcase
    end

end

 assign PCload = (State == 3'b000) ? 1 : 0;

 assign ALUoperation = (opcode == `ADD && State == 3'b011) ? 3'b000  :
                       (opcode == `SUB && State == 3'b011) ? 3'b001  :
                       (opcode == `OR  && State == 3'b011) ? 3'b010  :
                       (opcode == `AND && State == 3'b011) ? 3'b011  :
                       (opcode == `NOT && State == 3'b011) ? 3'b100  :
                       (opcode == `XOR && State == 3'b011) ? 3'b101  :
                       (opcode == `CMP && State == 3'b011) ? 3'b110  :
                       (opcode == `JMP && State == 3'b011) ? 3'b111  :
                       (opcode == `BXX && State == 3'b011) ? 3'b111  :
                       (opcode == `LD  && State == 3'b011) ? 3'b111  :
                       (opcode == `LDI && State == 3'b011) ? 3'b111  :
                       (opcode == `LDX && (State == 3'b011 ||State == 3'b111) ) ? 3'b000  :
                       (opcode == `ST  && State == 3'b011) ? 3'b111  :
```

```
 assign Flag_RegWrite = (`RegWriteHIGH && (State == 3'b011)) ? 1 :
                        ((opcode == `JMP) && (State == 3'b011) && (bit16)) ? 1 :
                        ((opcode == `LD || opcode == `LDX) && (State == 3'b011 ||State == 3'b111)) ? 1:0;

 assign Flag_Memread = (`MemReadHIGH && (State == 3'b011 ||State == 3'b111) ) ? 1 : 0;

 assign Flag_Memwrite = (`MemWriteHIGH && (State == 3'b011 || State == 3'b111)) ? 1 : 0;

 assign Flag_Indexed = (`Flag_IndexedHIGH && (State == 3'b011||State == 3'b111)) ? 1 : 0;

 assign ImmedMemAddress = (`ImmedMemAddressHIGH && (State == 3'b011||State == 3'b111)) ? 1 : 0;

 assign PCsource = ((opcode == `BXX) && (branchValid))? 2'b01 :
                   ((opcode == `JMP)  && State == 3'b011) ?  2'b10 :
                   ((isr1 || isr2) && ~ISR_entered )? 2'b11 : 2'b00;

 assign RegDest = ((opcode == `JMP) && (State == 3'b011) && (bit16)) ? 1 : 0;

 assign Halt = (opcode == `HLT && State == 3'b011) ? 1 : 0;

 assign bit_changeCC = (State == 3'b011) ? 1 : 0;
```

Figure 12: Control Unit Code

3.2 Interrupt Controller

The interrupt controller within our processor operates on a sophisticated mechanism. When an interrupt is triggered, it checks whether both the interrupt mask and interrupts are enabled. If these conditions are met and a request is received, the interrupt controller proceeds with handling the interrupt. Initially, the program counter (PC) source is modified to save the previous PC value, and it is then updated to point to the address corresponding to the specific interrupt. Subsequently, the "ISR_Entered" flag is activated, causing the PC source to transition to 0. This facilitates the normal incrementing of the program counter within the Interrupt Service Routine (ISR). Crucially, the ISR concludes with a Return from Interrupt (RETI) instruction, allowing the program to seamlessly revert to its regular execution flow after the interrupt has been serviced. This intricate process ensures efficient and effective interrupt handling within the processor architecture. This process can be visualized in figure 13.

(If "a" condition is true PCsource = 3, if its false PCsource = 0)

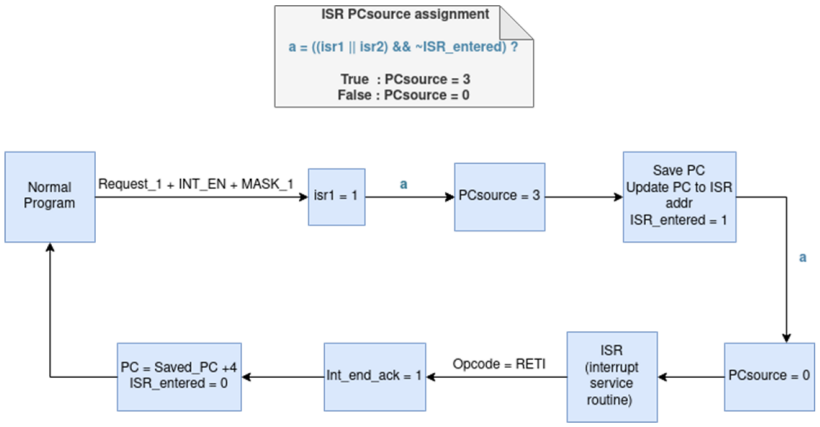


Figure 13: Interrupt controller diagram

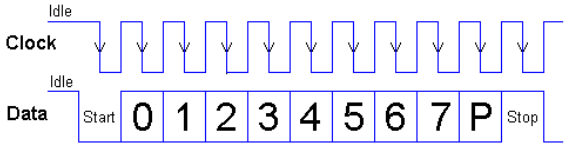
Our implementation of this mechanism in Verilog can be seen in figure 14. As we can see, in every positive edge of the clock it is verified if the interrupt enable flag is 1 before checking the mask and request. The priority control is automatically managed when entering on the "if" statement since the first conditions being checked are the ones that activate the higher priority interrupts. With the utilization of the "acknowledge" flag we can guarantee that an interrupt can not be interrupted by another one.

```
22  `define INTR_STATE1 2'b01
23  `define INTR_STATE2 2'b10
24
25
26  module intrController(
27      input clk,
28      input rst,
29      input i_intr_request1,   //Botao fpga 1
30      input i_intr_request2,   //Botao fpga 2
31      input [1:0]i_intr_enable_mask,  //smp a 1
32      input i_intr_enable,     //ISRs ativas //switch
33      input intr_ended_ack,        //ack cpu   //quando acaba a isr manda para CU
34      output reg [1:0]o_interrupt_vector, //
35      output reg o_interrupt_signal
36      );
37
38     reg acknowledged;    //Garante que uma interrupçao nao pode ser interrompida
39
40
41     always@(posedge clk)begin
42          if(rst)begin
43              o_interrupt_vector <= 2'b00;
44              o_interrupt_signal <= 0;
45              acknowledged <= 0;
46          end
47          else if(i_intr_enable) begin
48              if(i_intr_request1 && i_intr_enable_mask[0] && ~acknowledged) begin
49                 o_interrupt_vector<=`INTR_STATE1;
50                 o_interrupt_signal<=1;   //fica a 1 para caso haja um request da 2, nao possa entrar na isr2
51                 acknowledged<=1;
52               end
53              else if(i_intr_request2 && i_intr_enable_mask[1] && ~acknowledged) begin
54                 o_interrupt_vector<=`INTR_STATE2;
55                 o_interrupt_signal<=1;
56                 acknowledged<=1;
57               end
58              else if(intr_ended_ack)begin
59                  o_interrupt_vector<=2'b00;
60                  o_interrupt_signal<=0;
61                  acknowledged<=0;
62              end
63          end
64          else if (~i_intr_enable)begin
65              o_interrupt_vector<=2'b00;
66              o_interrupt_signal<=0;
67              acknowledged<=0;
68          end
69     end
70
71
```

Figure 14: Interrupt controller code

3.3 PS/2 Keyboard Controller

The PS/2 device interface was developed by IBM. Both the PS/2 keyboard and mouse interface implement a bidirectional synchronous serial protocol, but we will only consider the data-output from the PS/2 keyboard. The device writes a bit on the Data line when the Clock line is high, and it is read by the host when the Clock line is low. The PS/2 serial protocol uses 11-bit frames, which are 1 Start bit, 8-bit of data, one parity bit and 1 stop bit.



The module has four inputs (`clk`, `rst`, `PS2D`, and `PS2C`), and two outputs (`flag` and `key`). The `flag` signal indicates the end of a PS/2 transmission, while `key` represents the 8-bit data received, corresponding to the pressed key.

Internally, the module utilizes registers for various purposes, including storing the current data byte (`data_curr`), managing errors (`error`) and handling parity information (`keyboard_parity`).

Tests with an oscilloscope revealed the PS/2 keyboard's clock frequency to be approximately 13kHz(figure 15). Consequently, there is a need to downscale the FPGA clock frequency to at least 26kHz, following Nyquist's theorem.

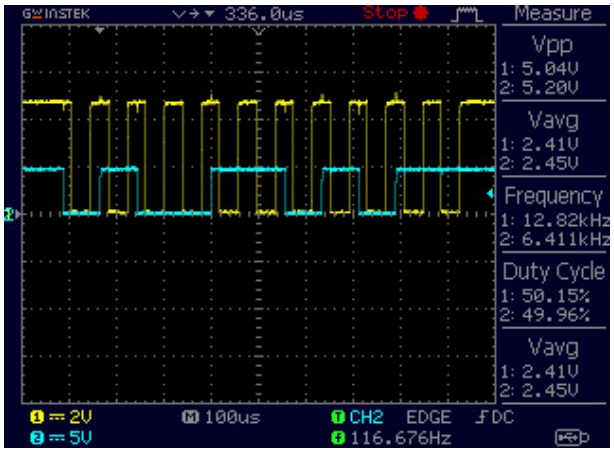


Figure 15: PS/2 clock and data

In our implementation a switch-case structure processes each incoming data bit, storing the 8-bit data in the (`data_curr`) register. In the tenth iteration, the module verifies the parity bit. During this iteration, each bit in (`data_curr`) is examined, and the count of ones is determined If the sum of this count and the (`keyboard_parity`) bit results in an odd total, the module sets the (`error`) flag to 1, signaling a parity error in the received data byte. The last iteration checks if the `PS2D` is high because the STOP bit is always high.

If the counter is one and the `PS2D` is low, that means that there is a new transmission (START bit always low), resulting in the reset of all registers. If the counter is between two and eleven, it is incremented by one to proceed to the next bit. If neither of these conditions is met, the counter is reset to one.

Finally, in the case of a negative transition of the `flag` register (indicating no error in the received data), the `key` output register takes on the value of the `data_curr` register.

```
22   always@(negedge clk)begin
23     if(rst) begin
24         counter<=4'b1;
25         data_curr<=8'b0;   //para verificar se retira efetivamente o data_curr
26         error<=1'b0;
27         pre_clk<=1'b0;
28         ticks<=12'b0;
29         flag<=0;
30         key<=0;
31
32         i<=0;       //iteracao
33         count<=0;   //contador de 1's
34     end
35     else begin
36
37           case(counter)
38                1: begin end
39                2: begin data_curr[0] <= PS2D; end
40                3: begin data_curr[1] <= PS2D; end
41                4: begin data_curr[2] <= PS2D; end
42                5: begin data_curr[3] <= PS2D; end
43                6: begin data_curr[4] <= PS2D; end
44                7: begin data_curr[5] <= PS2D; end
45                8: begin data_curr[6] <= PS2D; end
46                9: begin data_curr[7] <= PS2D; end
47
48                10: begin flag<=1;                  //FLAG VEM A 1 PARA INDICAR O FIM DA TRANSMISSAO
49                         keyboard_parity=PS2D;
50                         for(i=0; i<=7; i=i+1) begin
51                           if(data_curr[i]==1'b1) begin
52                                count = count + 1;
53                           end
54                          end
55                    if((count+keyboard_parity)%2!=1) begin
56                           error<=1;
57                    end
58                        else error<=0;
59                   end
60                //CASO NAO HAJA ERRO, A FLAG VOLTA A 0
61                11: begin if(PS2D!=1'b1) begin
62                           error<=1;
63                         end  else if(~error) begin
64                         flag<=0;
65                         end
66                    end
67                endcase
68
69            //NOVA TRANSMISSAO
70            if((counter==1) && (~PS2D)) begin
71                counter<=4'd2;
72                flag<=0;
73                error<=1'b0;
74                data_curr<=8'b0;
75                keyboard_parity<=0;
76                i<=0;
77                count<=0;
78            end
79            //PASSAR PARA O BIT SEGUINTE
80            else if ((counter>=2) && (counter<11)) begin
81                counter=counter+4'd1;
82            end
83            //CASO NENHUM DOS IFS, DA RESET AO COUNTER PARA 1
84            else  counter<=4'h1;
85          //end
86
87     end
88   end
89
90
91   always@(negedge flag) begin
92       key=data_curr;
93   end
94
95
96
97
98   endmodule
```

Figure 16: PS/2 Keyboard controller code

4 Tests

4.1 Clock Gating Test

As we can see on the simulation the clock signal stops when the "HALT" instruction is executed(Instruction - F8000000) this way we can use the control unit to control the clock flow and guarantee significant benefits for dynamic power reduction, as well as extend battery life and reduce energy consumption for portable and wireless devices.

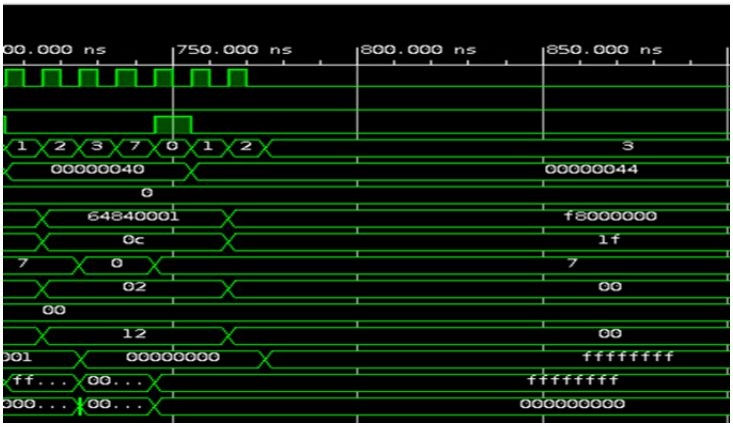


Figure 17: Clock Gating test

4.2 CPU core

For an easier interpretation of the CPU core simulation on figure 18 its present all the Opcodes corresponding to the possible CPU operations, we've tested them in order starting with the NOP (opcode:00000) and finishing with HALT(opcode:11111).

```
1  `define NOP  5'b00000
2  `define ADD  5'b00001
3  `define SUB  5'b00010
4  `define OR   5'b00011
5  `define AND  5'b00100
6  `define NOT  5'b00101
7  `define XOR  5'b00110
8  `define CMP  5'b00111
9  `define BXX  5'b01000
10 `define JMP  5'b01001
11 `define LD   5'b01010
12 `define LDI  5'b01011
13 `define LDX  5'b01100
14 `define ST   5'b01101
15 `define STX  5'b01110
16 `define HLT  5'b11111
17 `define RETI 5'b10000
```

Figure 18: Opcodes list

In this simulation we are able to see the correct functioning of the fetching, decoding and executing sequence of the instructions with of course the 2 extra states due to the latency mentioned previously. We can also see the result from the ALU only in the execute state and the control signals being enable/disabled on the respective instructions. To perform a PC change we've implemented a forward branch carry. This happens at 510ns from the beginning of execution (the yellow vertical line displayed).

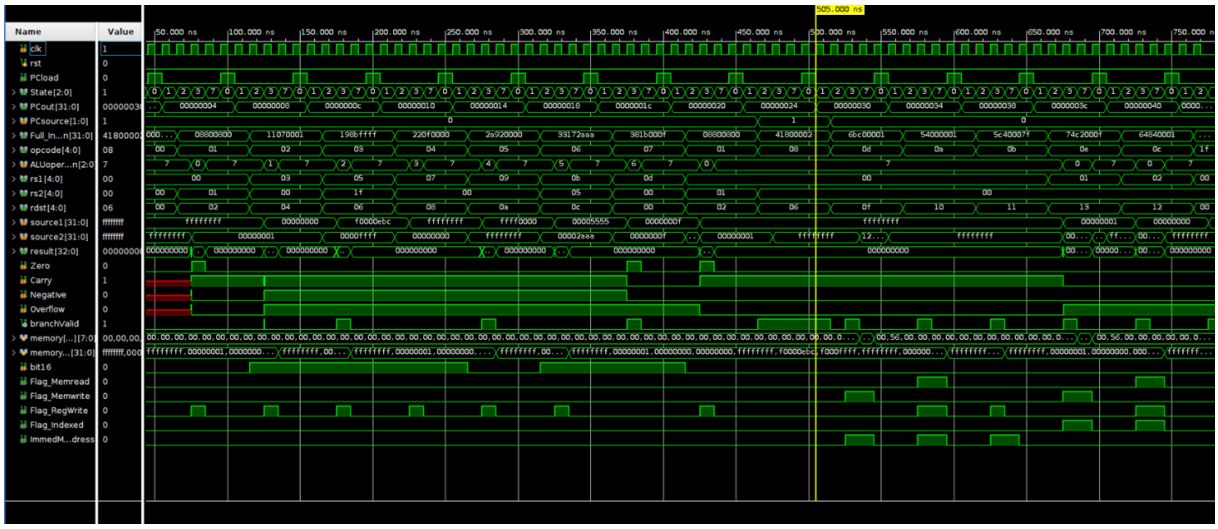


Figure 19: CPU core test

To test this synthesize this on the FPGA, the "ALUoperation" variable was displayed on the embedded FPGA's LEDs. In order to see them blink we had to divide the clock frequency so each instruction takes 1 second to execute since every instructions take 5 clock cycles(5 states) and the clock we are using as a frequency of 125MHz we've created a counter to count 15_600_000 times and switch the clock signal, other wise the led would blink a with a frequency to high to be visualized. This division was made on the clock gating module, before gating the clock as you can see in the following print screen.

```
23  module HALT_AND_CLK(
24      input sysclk,
25      input Halt,
26      output VeSPA_clk
27      );
28      reg OneHz_CLK;
29      reg [25:0]counter_vsp = 0;
30
31      always@(posedge sysclk)begin
32          counter_vsp <= counter_vsp +1;
33          if(counter_vsp == 15_600_000)begin      //SYSCLK = 125MHZ entao o clock troca com um  freq de 15.6Mhz para obter 1 ciclo de instruçao,
34              counter_vsp<=0;
35              OneHz_CLK = ~OneHz_CLK;
36          end
37      end
38
39      assign VeSPA_clk = (OneHz_CLK & ~Halt);     //If Halt = 1 Out_Signal = (1 & 0) = 0
40
41
42  endmodule
43
```

Figure 20: Clock division & Clock Gating(TopLyr.v/HALT_AND_CLK.v)

After generating the bitstream and programming the ZYBO, we can now see the ALU operation changing with the change of the executing instruction. Comparing "ALUoperation" in figure 19 and the LEDs on figure 21 we are able to see the perfect behavior of the CPU's core.

Figure 21: FPGA Core testing

4.3 Interrupt Controller test

When the Interrupt Controller was tested, the ISR1, ISR2, and the priority of the interrupts were examined (these three tests were successfully synthesized and will be presented physically). In Figure 22, both simulations occur simultaneously; the top one represents the interrupt controller's behavior, while the bottom one represents the CPU's behavior towards an interrupt 1 request. At the beginning of the simulation, a request is observed, although nothing happens because neither the interrupt enable nor the respective mask are enabled. When these signals are finally enabled, we can observe the program counter updating to 0x60, and the previous one is stored in the 'CurrentPC' signal. When the RETI instruction is reached, the program counter is updated back to the one stored previously.

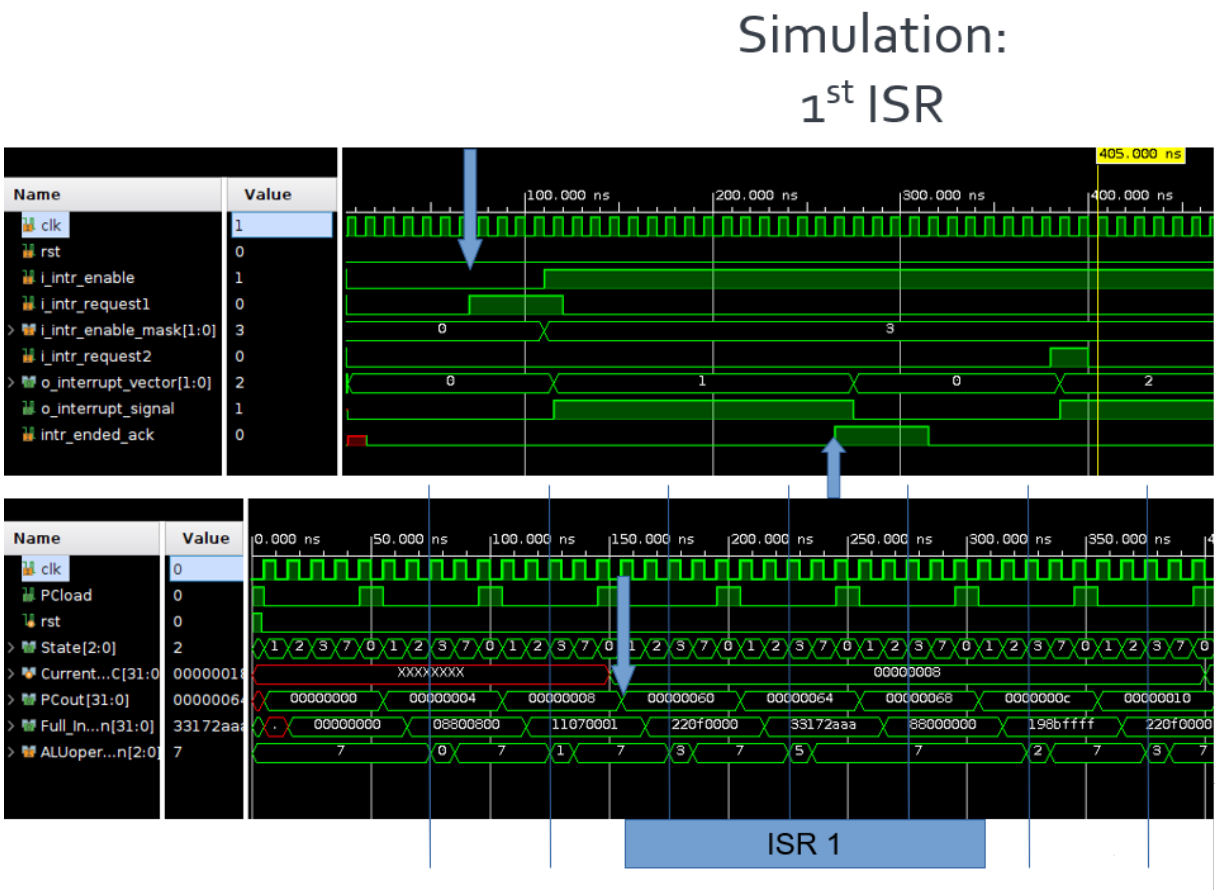


Figure 22: ISR 1 test

The same process happens with the isr2 test but in this case the PC is updated to 0x70:

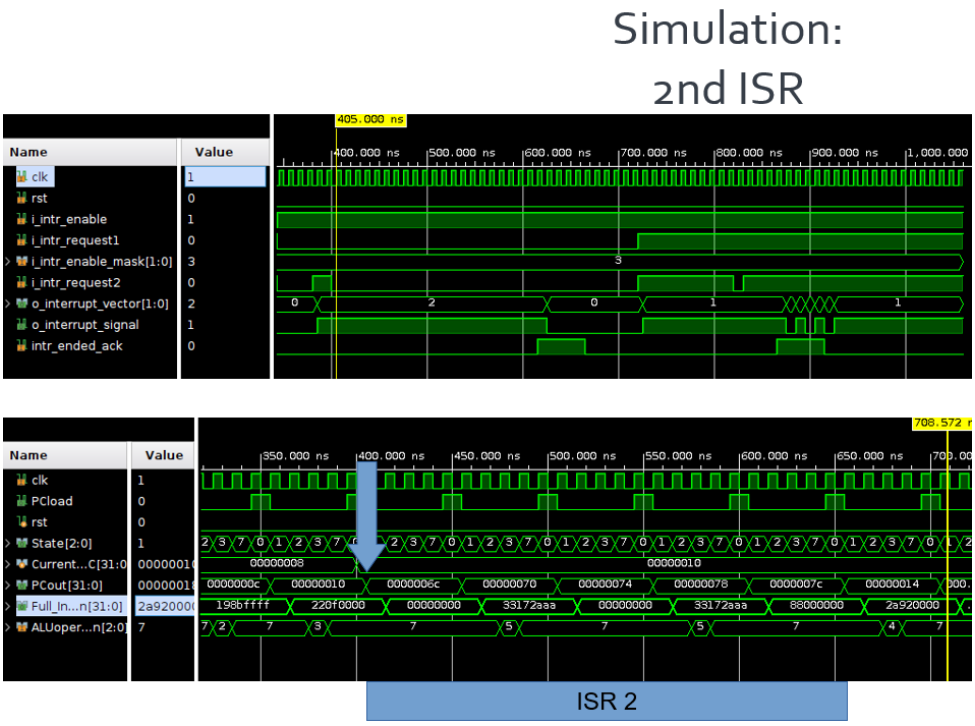


Figure 23: ISR 2 test

And finally has you can see on the priority test 2 request happened at 700 nano seconds but the isr1(the higher priority interrupt) is the one to execute.

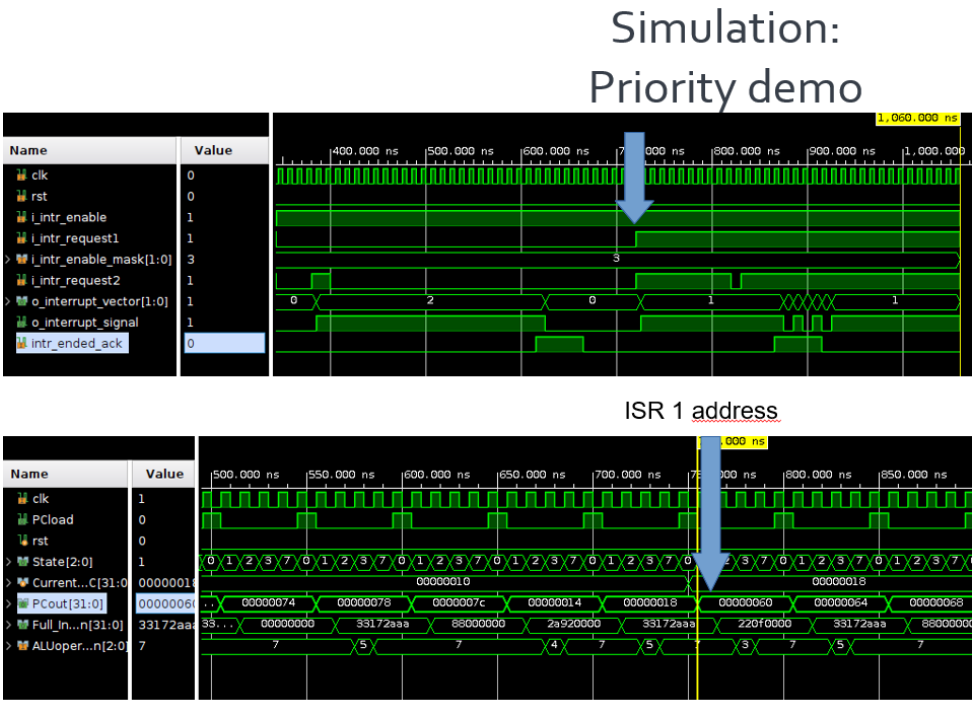


Figure 24: Interrupt Priority test

4.4 PS/2 Keyboard controller Test

First of all, we started testing the implemented keyboard controller separately from VeSPA CPU, so after creating a test bench(figure 26) simulating every single bit which the keyboard would transmit, we've ran our simulation.

```
//tecla A=0x1C = 00011100
initial begin
clk=0;
PS2C=1;
PS2D=1;

#5 PS2D=0;          //START BIT

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;  //BIT 0

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;   //BIT1

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=1;   //BIT2

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=1;   //BIT3

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=1;   //BIT4

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=0;   //BIT5

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=0;   //BIT6

#5 PS2C=0;

 #5
PS2C=1;
PS2D=0;   //BIT7

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=0;   //PARIDADE

 #5 PS2C=0;

 #5
PS2C=1;
PS2D=1;  //STOP

#5 PS2C=0;
#5

#5 PS2C=1;

#40 //NOVA TECLA**************
//tecla Z=0x1A = 00011010

PS2C=1;
PS2D=1;

#5 PS2D=0;          //START BIT

#5 PS2C=0;


#5
PS2C=1;
PS2D=0;    //BIT0

#5 PS2C=0;

#5
PS2C=1;
PS2D=1;    //BIT1

#5 PS2C=0;

#5
PS2C=1;
PS2D=1;    //BIT2

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT3

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT4

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT5

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT6

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT7

#5 PS2C=0;

#5
PS2C=1;
PS2D=1;    //PARIDADE

#5 PS2C=0;

#5
PS2C=1;
PS2D=1;    //STOP

#5 PS2C=0;

#5 PS2D=0;

#5 PS2C=1;

#100 //NOVA TECLA**************
//tecla SPACE=29 = 00101001

PS2C=1;
PS2D=1;

#5 PS2D=0;          //START BIT

#5 PS2C=0;


#5
PS2C=1;
PS2D=1;    //BIT0

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT1

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT2

#5 PS2C=0;

 #5
PS2C=1;
PS2D=1;    //BIT3

#5 PS2C=0;

 #5
PS2C=1;
PS2D=0;    //BIT4

#5 PS2C=0;

 #5
PS2C=1;
PS2D=1;    //BIT5

#5 PS2C=0;

 #5
PS2C=1;
PS2D=1;    //BIT6

#5 PS2C=0;

#5
PS2C=1;
PS2D=0;    //BIT7

#5 PS2C=0;

 #5
PS2C=1;
PS2D=0;    //PARIDADE

#5 PS2C=0;

 #5
PS2C=1;
PS2D=1;    //STOP

#5 PS2C=0;

#5 PS2D=0;

#5 PS2C=1;

end
```

Figure 25: PS/2 Test Bench

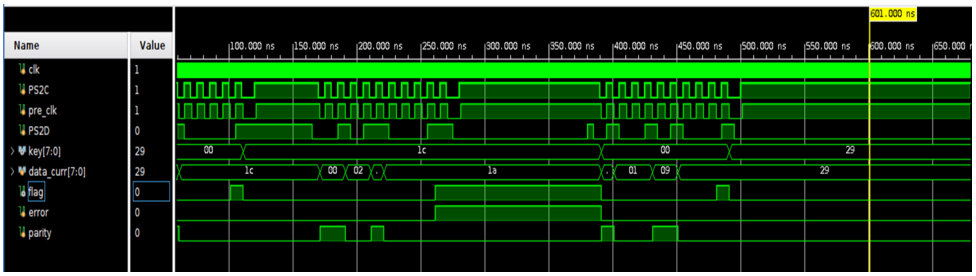


Figure 26: PS/2 Keyboard controller Individual simulation

When unifying this protocol with our CPU, we found a lot of problems with the PS/2 keyboard that we were using and we couldn't even turn it on, because of this we had to find another way to test our PS/2 keyboard controller. So, we have simulated all the signals that a real key (from a PS/2 keyboard) would send to the FPGA (the clock and the 11 bits) using an STM32.

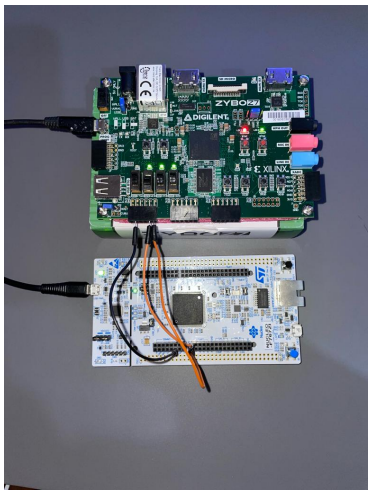


Figure 27: FPGA & STM32 setup

The Key that we have tested is the letter 'A' which corresponds to the number 0x1C, which by it self only contains 3 1's that by using an odd parity makes the parity bit equal to 0(because the quantity of 1's is already odd). After knowing the bit sequence a PS/2 keyboard(Figure 28) would send if letter 'A' was pressed, we've started implementing on the STM Cube IDE.

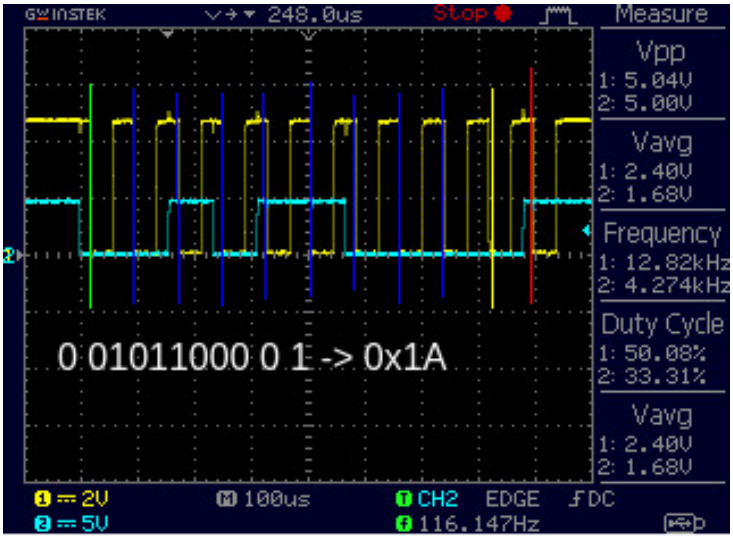


Figure 28: Osciloscope PS/2 signals

We firstly started by creating 2 GPIO outputs for the clock and for the data sequence, and also by activating Timer7 interrupt and making it overflow with a frequency of 13kHz(prescaler=0 ; ARR=4000 (around 13,7kHz))(the frequency measured on our PS/2 keyboard).

Inside the "HAL_TIM_PeriodElapsedCallback" (the callback function that is called every time the timer overflows) we toggled the clock pin to generate an output clock with the frequency of 13kHz and wrote on the data pin the bit sequence as you can see in figure 29

```
const unsigned short int A_Key_bit_sequence[12] = {0,0,0,1,1,1,0,0,0,0,1};
unsigned short int sent_data[12];
```

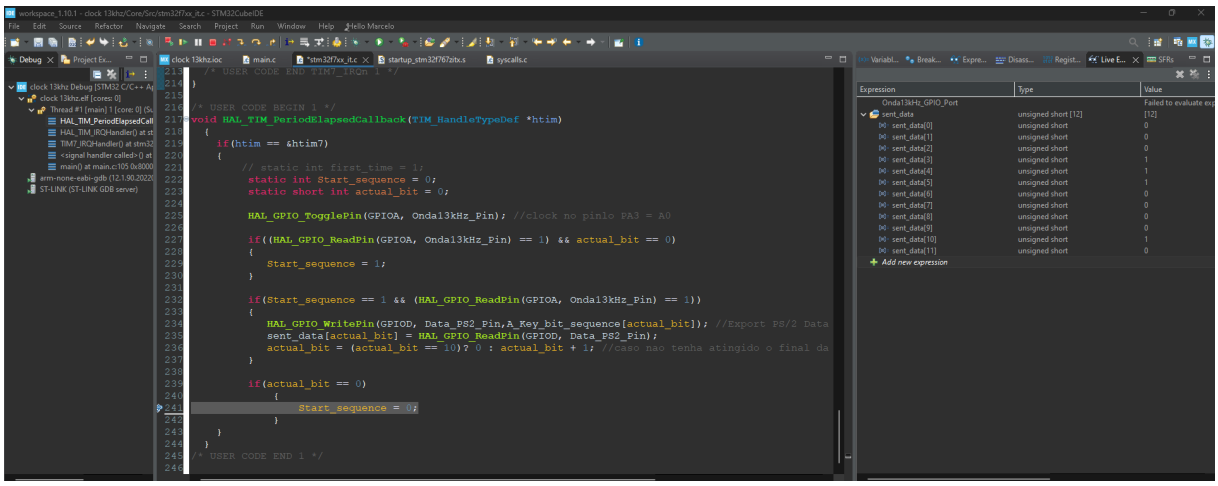


Figure 29: STM32 Letter 'A' generation

After debugging the code and placing a break point in the end of the callback, we are able to see on the "Live Expressions" the data sent to the FPGA, this breakpoint is also useful to simulate the click of the keyboard letter by resuming the execution after it stopped.
To Complete our project we've made one single test that is the three modules(CPU + IR+PS/2 controllers) together. By pressing the letter 'A', the ISR1 of the interrupt controller request is activated interrupting the main CPU program and jumping to the isr code.This verification was made in the control unit.(This test will also be demonstrated in the presentation).

The PCsource can be change if a branch instruction is executed, a jump instruction is executed or if a isr is able to execute and no interrupt is occurring. If none of these condition are true the PCsource is zero.

```
131  assign PCsource = ((opcode == `BXX) && (branchValid))? 2'b01 :
132                    ((opcode == `JMP)  && State == 3'b011) ?  2'b10 :
133                    ((isr1 || isr2) && ~ISR_entered )? 2'b11 : 2'b00;
```

Figure 30: PCsource assignment(ControlUnit.v)

The ISR will be able to execute if the conditions on figure 31 meet, in this particular example we don't use the "isr1" flag because we wanted the request 1 to be the PS/2 key.

```
152  wire [1:0] o_interrupt_vector;
153  wire o_interrupt_signal;
154
155  intrController InterruptCTRL(
156      .clk(clk),
157      .rst(rst),
158      .i_intr_request1(PS2_INTR),      //Letter 'A'
159      .i_intr_request2(INTR_BOT2),     //Botao fpga 2
160      .i_intr_enable_mask(Mask),       //smp a 1
161      .i_intr_enable(INTR_EN),         //ISRs ativas //switch
162      .intr_ended_ack(intr_end_ack),   //ack cpu
163      .o_interrupt_vector(o_interrupt_vector), //
164      .o_interrupt_signal(o_interrupt_signal)
165      );
166
167  assign intr_end_ack = (opcode == `RETI) ? 1 : 0;
168
169  assign isr1 = (o_interrupt_signal && o_interrupt_vector == 2'b01 && ~isr2)? 1 : 0;
170  assign isr2 = (o_interrupt_signal && o_interrupt_vector == 2'b10  && ~isr1)? 1 : 0;
171
172  `define A_key (PS2_key == 8'h1C)
173
174  assign PS2_INTR = (`A_key)? 1:0;
175
176  endmodule
```

Figure 31: PS/2 request and Interrupt(ControlUnit.v)

After the PCsource changes it will enter on the `switch-case` in figure 32 and depending on the isr it will change the program counter accordingly.

```
42   always@(negedge clk)begin
43
44      if(rst) begin       //por defeito
45        //PCoutMux <= PCin + 4;
46        PCoutMux <= 0;
47        ISR_entered <= 0;
48      end
49
50      else if(intr_end_ack)begin
51              PCoutMux<= Current_PC + 4;
52              ISR_entered <= 0;
53          end
54       else begin
55              case(PCsource)
56                  2'b00: begin        //PC+4bytes intruction memory as 32bit each address
57                      PCoutMux <= PCin + 4;
58                  end
59                  2'b01: begin        //BRANCH
60                      PCoutMux <= PCin + 4 + (sext23<<2);
61                  end
62                  2'b10: begin        //JUMP
63                      PCoutMux <= ((rs1 + sext16)<<2);
64                  end
65                  2'b11:begin         //ISR
66                      if(PCload == 1)begin
67                          Current_PC <= PCin; // saves the current pc before jumping
68                          ISR_entered = 1;
69                          if(isr1)begin
70                              PCoutMux<= 32'h60;
71                          end
72                          else if(isr2)begin
73                              PCoutMux<= 32'h6C;
74                          end
75                      end
76
77                  end
78              endcase
79
80      end
```

Figure 32: Program Counter Multiplexer(DataPath.v/PCAdder.v)

5 Conclusion

The VeSPA processor project has successfully achieved its objectives, resulting in a customized and efficient processor. The combination of a low latency datapath, a robust control unit, an interrupt controller, a PS/2 keyboard controller, and a byte-addressable Harvard architecture makes the processor suitable for a variety of embedded systems.

6 Future Work

To give continuity to the project we will explore optimizations for further performance enhancements with Pipelining, we could also integrate additional peripherals for expanded functionalities.

7 Acknowledgments

Last but no least, a special thanks to Professor Dr. Adriano Tavares for providing the necessary resources and support throughout the project.

References

[1] Programação de Microcontroladores(MCS-51) by Adriano T., Carlos L.,Carlos S., Jorge C., Paulo C., 2009.

[2] Computer Organization & Design, the Hardware/Software Interface by David A. Patterson, John L. Hennessy, 2nd Edition

[3] Memory Systems - Cache, DRAM, Disk by Bruce Jacob, Spencer W.NG, David T.Wang, 2008