Escola de Engenharia
Universidade do Minho

ESRG
EMBEDDED SYSTEMS RESEARCH GROUP

VeSPA

Industrial Electronics and Computers Engineering
Embedded Systems and Computers

Professor:
Adriano Tavares

Afonso Maior, PG53959
Mário Lima, PG54079

Contents

Acronyms

FPGA	Field-programmable Logic Array
IC	Interrupt Controller
HDL	Hardware Description Language
ISA	Instruction Set Architecture
CPU	Central Processing Unit
ALU	Arithmetic and Logic Unit
UART	Universal Asynchronous Receiver-Transmitter
IRQ	Interrupt Request Lines
ISR	Interrupt Service Routine
RISC	Reduced Instruction Set Computer

List of Figures

1 | Analysis

1.1 Problem statement

The project at hand involves the design and implementation of an the VeSPA processor, a RISC architecture with distinctive features such as Load-Store, 32-bit data processing, Big-Endian byte order, and a 3-address machine configuration. Additionally, the project encompasses the integration of an assigned peripheral and an interrupt controller for enhanced functionality and system interaction.

Furthermore, the whole implementation was to be performed using Verilog HDL and tested using an FPGA.

1.2 Requirements

In order to ensure proper functioning of the system, a set of functional and non-functional requirements has to be met.

1.2.1 Functional requirements:

- Implement an instruction set adhering to RISC architecture;
- Support a Load-Store architecture;
- Support a byte addressable memory;
- Handle interrupts through the use of an interrupt controller;
- Support Big-Endian byte order;

1.2.2 Non-functional requirements:

- Ensure an efficient performance of the system;
- Be reliable to ensure proper functioning over time.

1.3 Constraints

There is also a set of boundaries on how the project can be executed that is defined by its constraints, whether technical or non-technical.

1.3.1 Technical constraints:

- The implementation is performed on a Zybo Z7-10 board;
- Implementation performed using Verilog HDL.
- CPU implementation split into datapath and control unit;
- Utilize memory IPs to provide better system performance.

1.3.2 Non-technical constraints:

- Project developed by a group of two members;
- Project deadline at the end of the semester.

1.4 CPU

A processor is a special circuit that is typically composed by two main components: the control unit and the datapath. This approach promotes a modular design that may result in a greater ease of development and a higher level of flexibility, this way being more adaptable to different possible requirements. Not only that, but this approach also possibly promotes an increase in performance as it makes it simpler to implement a possible pipeline mechanism that allows for different instructions to be in different stages of processing simultaneously. This implementation of the VeSPA microprocessor's ISA was also performed according to this methodology.

VeSPA has a RISC architecture, embodying the principles of simplicity and efficiency in its design. As a RISC processor, it employs a streamlined set of instructions, each executing in a single clock cycle, to optimize performance and enhance instruction throughput.

This processor follows a Load-Store architecture, adhering to the philosophy that arithmetic and logic operations are exclusively performed on data residing in registers. Memory interactions are facilitated through dedicated load and store instructions, ensuring a clear delineation between computational tasks and data movement.

As referred in the requirements, VeSPA's memory model is byte-addressable. This addressing capability provides flexibility and precision in memory access, accommodating a diverse range of data structures and addressing requirements.

Another feature of the VeSPA processor is its adherence to Big-Endian byte order. In this byte order, the most significant byte of a data word is stored at the lowest memory address, followed by subsequent bytes in decreasing significance.

This processor is also a 32-bit machine, this way being capable of addressing a significant amount of memory and still maintaining a streamlined architecture for efficient instruction execution.

In addition to its RISC architecture and memory characteristics, VeSPA operates as a 3-address machine, enhancing its program versatility and execution efficiency.

1.5 Instruction formats and behaviour

Since VeSPA has a RISC architecture, its instruction set is made up of a reduced set of instructions, with a fixed size (32 bits) and simple decoding. Therefore, instructions are composed of different aligned fields, and for all instructions, the first 5 bits define the opcode and the rest are dedicated to the operands.

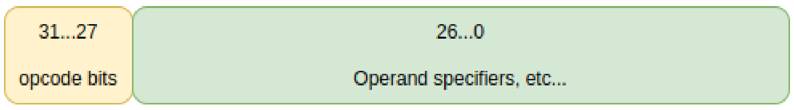


Figure 1.1: VeSPA basic instruction format

- **ALU operations**

  The instruction formats and behaviour for all the operations performed by the ALU are displayed in figures 1.2, 1.3 and 1.4 below.

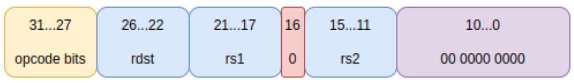


Figure 1.2: ALU instructions format 1

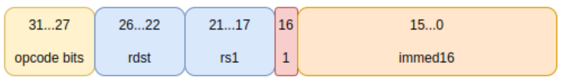


Figure 1.3: ALU instructions format 2

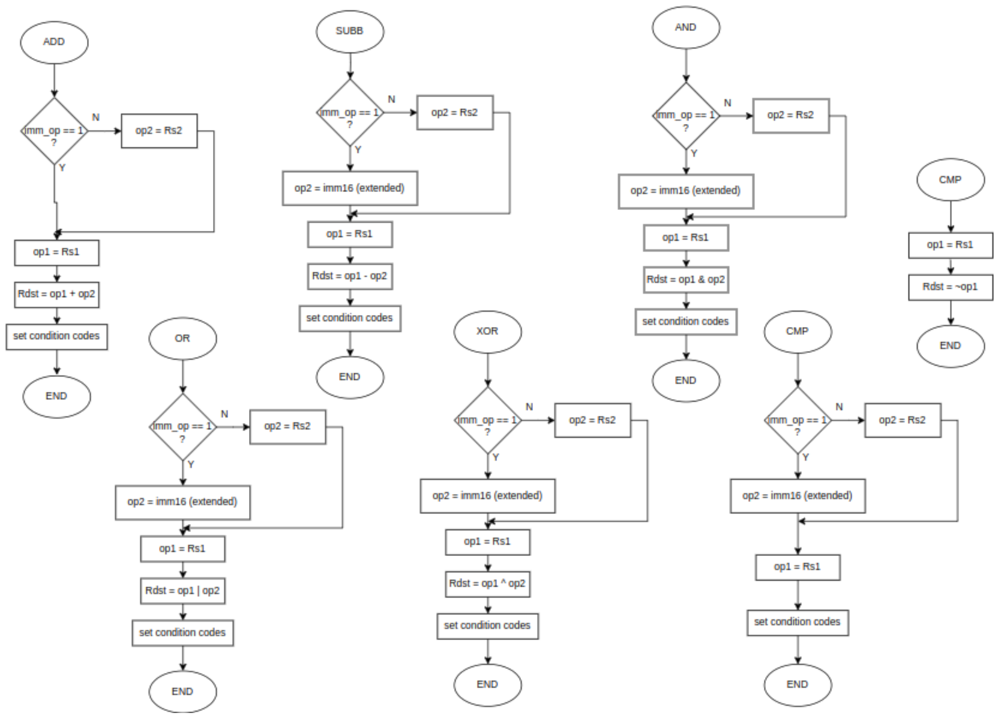


Figure 1.4: ALU instructions behaviour

- **BXX**

  The instruction format and algorithm for all the BXX operations are displayed in figures 1.5 and 1.6 below.

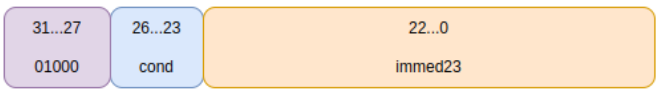


Figure 1.5: BXX instruction format

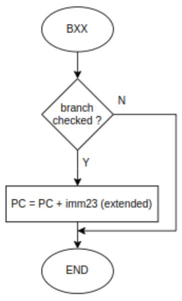


Figure 1.6: BXX instruction behaviour

- **JMP**

  The instruction format and algorithm for the JMP operation are displayed in figures 1.7 and 1.8 below.

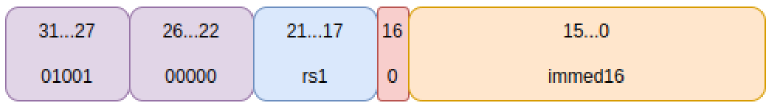


Figure 1.7: JMP instruction format

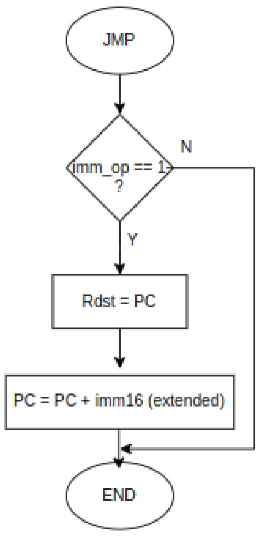


Figure 1.8: JMP instruction behaviour

- **JMPL**

  The instruction format and algorithm for the JMPL operation are displayed in figures 1.9 and 1.10 below.

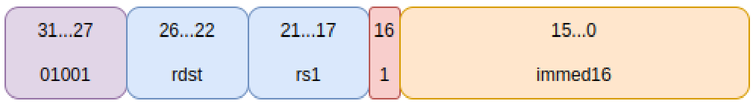


Figure 1.9: JMPL instruction format

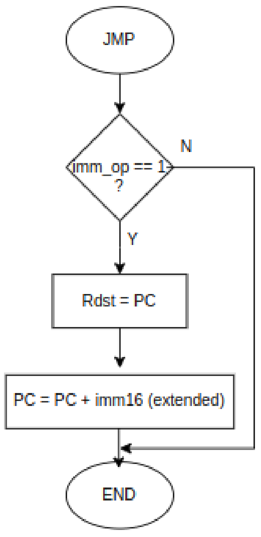


Figure 1.10: JMPL instruction behaviour

- **LD/LDI**

The instruction format and algorithm for the LD and LDI operations are displayed in figures 1.11 and 1.12 below.

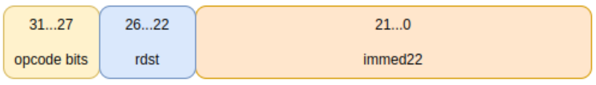


Figure 1.11: LD/LDI instructions format

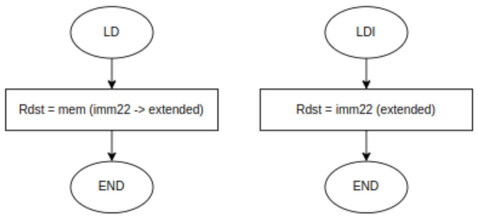


Figure 1.12: LD/LDI instruction behaviour

- **LDX**

  The instruction format and algorithm for the LDX operation are displayed in figures 1.13 and 1.14 below.

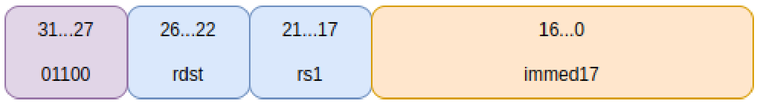


Figure 1.13: LDX instruction format

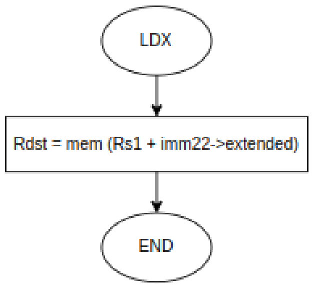


Figure 1.14: LDX instruction behaviour

- **ST**

  The instruction format and algorithm for all the ST operation are displayed in figures 1.15 and 1.16 below.

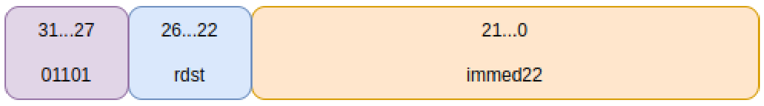


Figure 1.15: ST instruction format

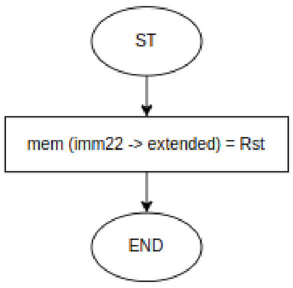


Figure 1.16: ST instruction behaviour

- **STX**

  The instruction format and algorithm for the STX operation are displayed in figures 1.17 and 1.18 below.

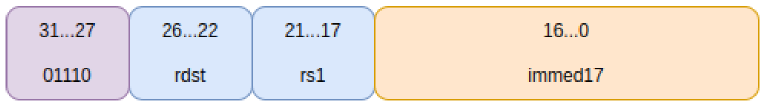


Figure 1.17: STX instruction format

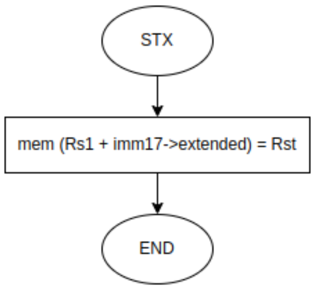


Figure 1.18: STX instruction behaviour

- **PUSH**

  Given that the processor developed contains a stack, a push operation was added with the intent to provide the user a simple way of using the stack, in this case, by "pushing" data into it. The instruction format and algorithm for the PUSH operation are displayed in figures 1.19 and 1.20 below.

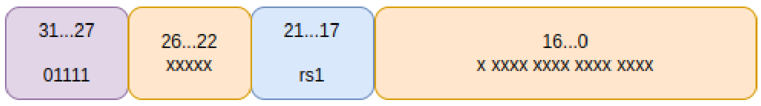


Figure 1.19: PUSH instruction format

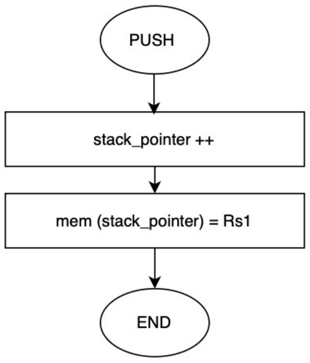


Figure 1.20: PUSH instruction behaviour

- **POP**

  Like the addition of the PUSH operation, a POP operation was also added to provide a simple way of retrieving data from the stack. The instruction format and algorithm for this operation are displayed in figures 1.21 and 1.22 below.

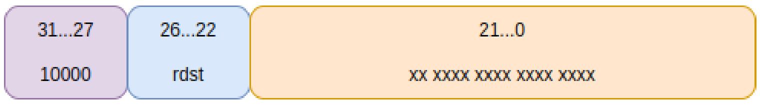


Figure 1.21: POP instruction format

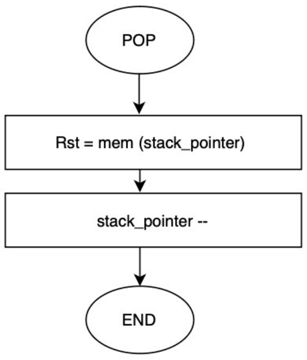


Figure 1.22: POP instruction behaviour

1.6 Interrupt controller

An interrupt controller is a hardware component in a computer system that manages and prioritizes interrupt signals from various devices and peripherals. The referred interrupt-generating devices are connected to the interrupt controller through specific interrupt request lines (IRQs). They are then interpreted by the component in question which then sends signals to the CPU that determine the flow of execution. This way, the interrupt controller ensures that the CPU attends said interrupts in an controlled and organized manner.

Typically, an interrupt controller also allows for the disabling of the interrupts as a whole and individually (interrupt masking).

An interrupt vector is an address pointing to the location of a particular interrupt's corresponding ISR. As referred before, after receiving and interpreting an IRQ, the interrupt controller then sends signals to the CPU. These signals allow for the fetching of the correct interrupt vector and, consequentially, for the execution of the corresponding ISR.

1.7 UART

The previously referred peripheral assigned to be implemented was the UART. The UART (Universal Asynchronous Receiver-Transmitter) is a technology that allows data transfer between devices. As the name implies, this protocol is asynchronous, which means there is no shared clock signal between devices. Communication between devices is done through two main pins: TX for data transmission and RX for data reception. This peripheral allows data transfer in a bidirectional manner, which means devices can transmit and receive information simultaneously. Data is transferred bit by bit, with a transmission rate called baudrate. Regarding the protocol, the line is normally set to a high level, and, when transmission begins, the line is set to a low level (start bit). The message is then sent, bit by bit. Finally, the stop bit is sent, which sets the line high again. This process is represented in the following figure 1.23.

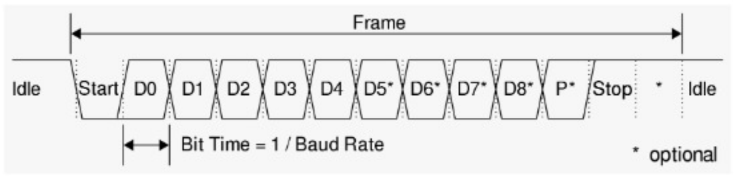


Figure 1.23: UART protocol representation

1.8 Tools

The project at hand makes use of Vivado as the main tool for the development of the project. Vivado is a design environment developed by Xilinx for the development of FPGA (Field-Programmable Gate Array) and SoC (System-on-Chip) designs. It provides a range of tools for designing, simulating, and implementing digital circuits on Xilinx FPGAs. Vivado supports a variety of design flows, including RTL (register-transfer level) design, high-level synthesis, and IP (intellectual property) integration. This tool also provides a user-friendly interface that makes it's utilization simple and intuitive.

2 | Design

2.1 Datapath

The CPU's datapath is responsible for the execution of all operations upon the received data. This component determines how the data is transferred between different functional units such as the ALU and the register file, for example. In figure 2.1 presented below is displayed the system's datapath design. It is worth mentioning that the variables represented in blue simply act as connections between the different functional units, the ones in green and red are inputs and outputs to the datapath, respectively.

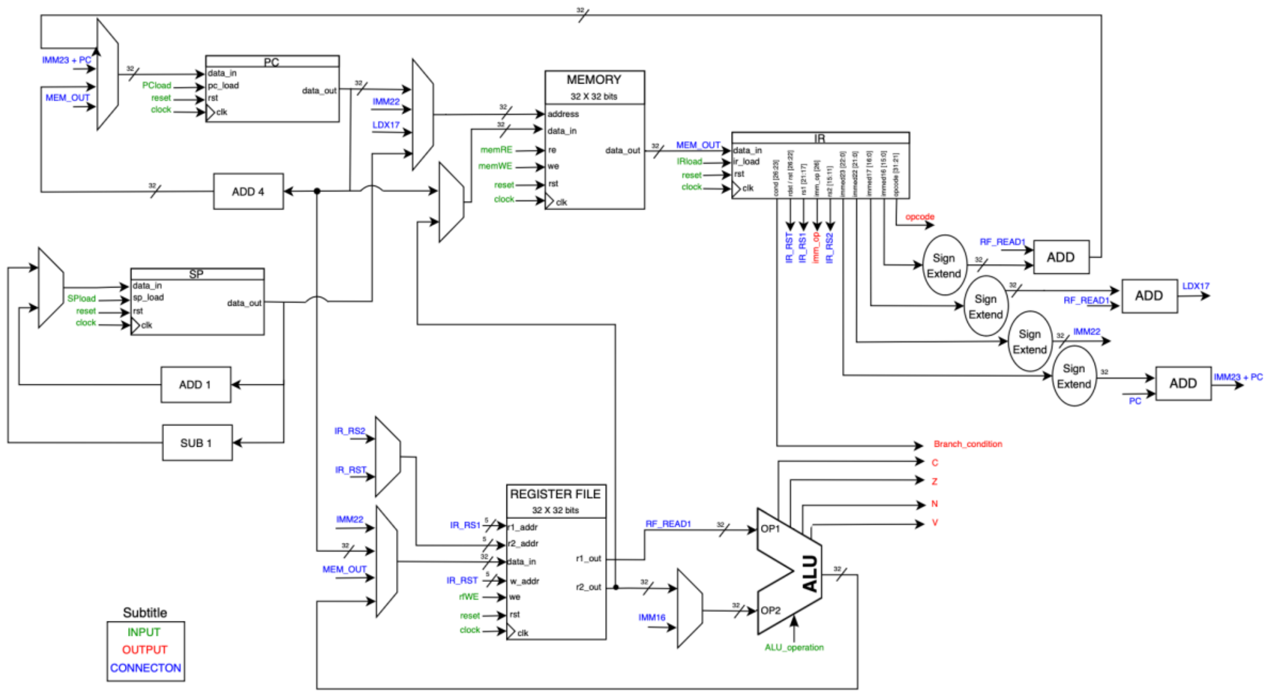


Figure 2.1: Datapath design

As depicted in the previous figure, the datapath receives and sends a number of inputs and outputs, respectively:

- **Inputs:**
  - reset - Reset signal;
  - clock - Clock signal;
  - PCload - Program counter load;
  - memRE - Memory read enable;
  - memWE - Memory write enable;
  - IRload - Intruction register load;
  - rfWE - Register file write enable;
  - SPload - Stack pointer load;
  - ALU_operation - Signals an ALU operation;
  - pc_sel - Multiplexer that mediates the source of the pc input;

  - mem_addr_sel - Multiplexer that mediates the memory address;
  - rf_read2_addr_sel - Multiplexer that mediates the register file's read2 address;
  - rf_write_sel - Multiplexer that mediates the source of the register file's data input;
  - mem_in_sel - Multiplexer that mediates the data input to the memory block;
  - sp_in_sel - Multiplexer that mediates the data input to the stack pointer block;
  - slu_op2_sel - Multiplexer that mediates the source of the second operand used by the ALU block (imm16 or from the register file);

- **Outputs:**
  - imm_op - immediate operand extracted from the instruction register;
  - opcode - Opcode extracted from the instruction register;
  - Branch_condition -Branch condition extracted from the instruction register;
  - C - Carry flag controlled by ALU;
  - Z - Zero flag controlled by ALU;
  - N - Negative flag controlled by ALU;
  - V - Overflow flag controlled by ALU;
  - irq_addr - Address of the IRQ's corresponding ISR;
  - int_ack_complete - Acknowledgement of the attendance of a ISR;
  - int_ack_complete - Acknowledgement of the attendance of a IRQ;

As referred above, the datapath contains different functional units. These will be described in the following subsections as well as their design for this specific system:

2.1.1 ALU

The ALU, as the name implies, is responsible for the handling of all of the arithmetic and logic operations (ADD, SUB, OR, AND, NOT, XOR, CMP). This functional unit has two inputs: one that comes from the register file "r1_out" and one that can also come from the register file, although in this case, from "r2_out", but also from the instruction register's extracted "imm_16", depending on the operation. It's output acts as an input to the register file's data input and it is also responsible for the control of the condition codes (C, N, Z, V). The figure 2.2 displayed below illustrates the referred functional unit.

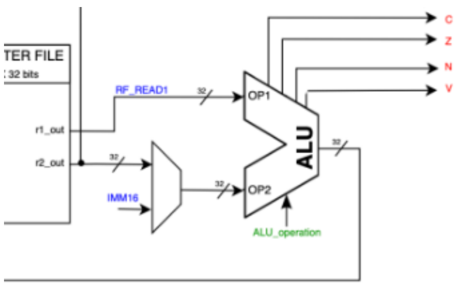


Figure 2.2: ALU functional unit design

2.1.2 Check Branch Condition

This functional unit handles the checking of the condition associated with operations of type branch (BRA, BNV, BCC, BCS, BVC, BVS, BEQ, BNE, BGE, BLT, BGT, BLE, BPL, BMI). To achieve this purpose, its inputs are the condition codes and the branch condition extracted from the instruction register, and it signals if the condition is checked or not.

2.1.3 Register File

As mentioned in the analysis phase, the Vespa is a 3-address machine. As such, to allow instructions like those in alu, in which the two operands come from the resgister file and the result goes to it, to execute in 1 clock cycle, it is necessary that the register file has 3 ports.

As the register file is a component that is located on a critical path of the datapath, it is necessary to implement memory using an IP, in order to make readings and writings more efficient.

However, after checking the Vivado documentation, it is not possible to configure a block RAM with 3 ports.

Therefore, two alternatives and respective trade-offs for implementing the register file will be presented below, in order to have 2 outputs and 1 data input.

Option 1

Implement the register file by split. To do this, it is necessary to instantiate two dual-port block RAMs with 16 addresses each, as you can see in figure x. To decide which of the two memories to access, the most significant bit of the address is used.

The advantage of this method is the reduction of resources spent, compared to the method presented below. The disadvantage is that a programmer can never read two registers from the same bank. However, this can be managed by the compiler. A diagram of this approach is displayed in figure 2.3 below.

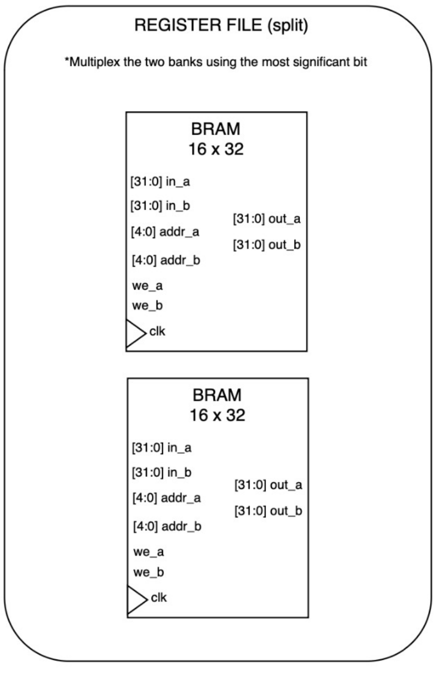


Figure 2.3: Split register file diagram

Option 2

Implement the register file by copying. To do this, instead of instantiating a single dual-port block RAM with 32 addresses, two are instantiated, as can be seen in figure x. By combining the data inputs and the write enable signal, the two memories always have the same content, allowing one of each output to be used.

An advantage of this method is that, unlike the previous one, the user can read 2 registers from the same bank. However, this method takes up twice as much resources.

After analyzing the advantages and disadvantages, it was decided to use the second alternative, since the size of the register file is not very large and the board used has sufficient resources. A diagram of this approach is displayed in figure 2.4 below.

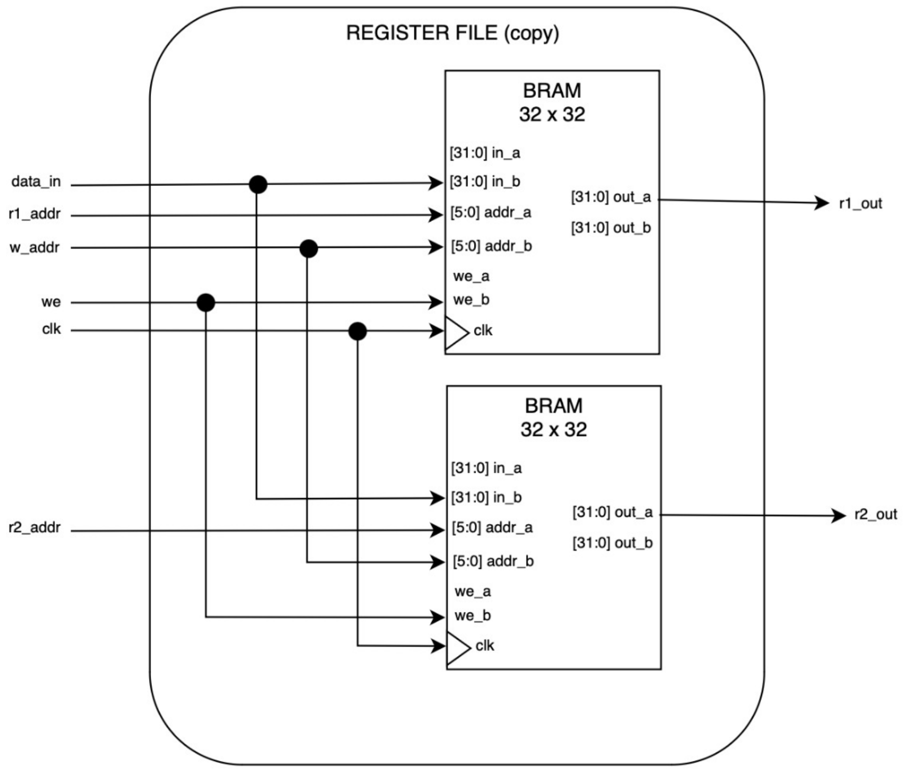


Figure 2.4: Copy register file diagram

2.2 Control Unit

A processor's control unit is responsible for managing the various operations of the CPU. The control unit works by activating and deactivating signals, attending to the desired operation to be performed, that will serve as inputs to the datapath. These two units work in unison to provide the desired functionality of the CPU. This component is implemented resorting to a state machine that is depicted in figure 2.5 displayed below.

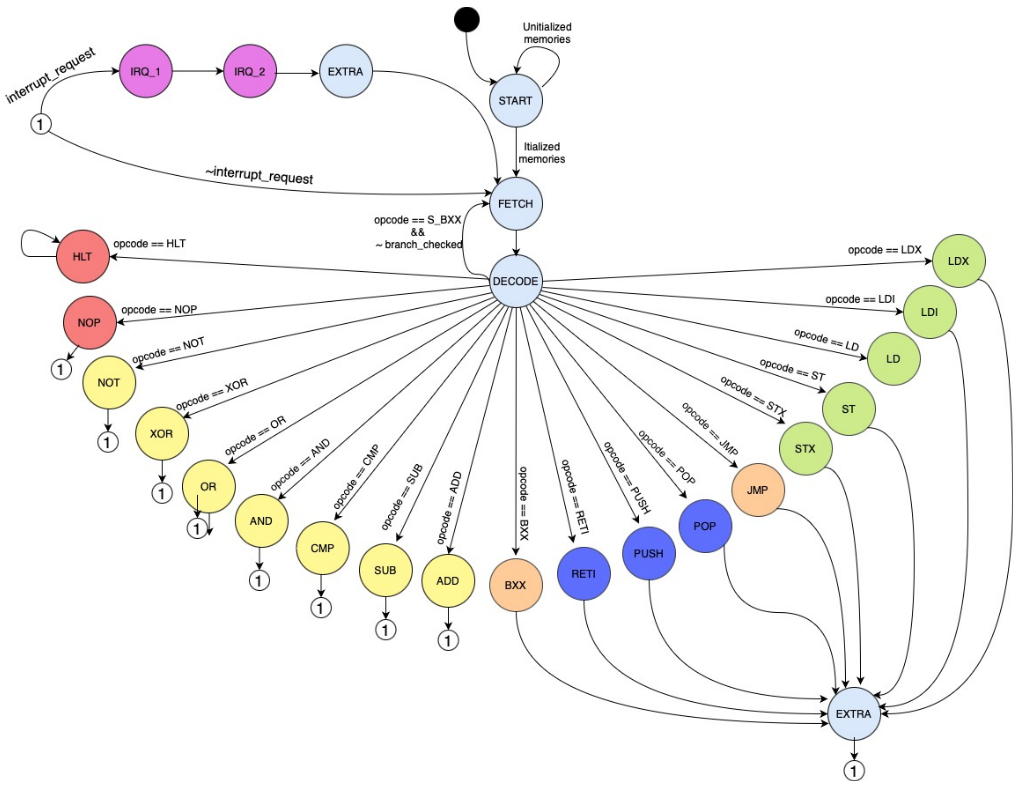


Figure 2.5: Control unit state machine

Each state of the system activates a given number of signals that will serve as inputs to the datapath circuit. The sequence of steps the CPU follows to execute an instruction is typically divided into three stages that form a continuous cycle: fetch, decode and execute.

2.2.1 Fetch

In the fetch state, signals will be sent to the datapath so that the CPU retrieves an instruction from the memory.

2.2.2 Decode

The decode state comes after the fetch state and precedes the execute state, as is verifiable in the previously presented state machine. In this state, the fetched instruction's opcode will be analysed (or, as the name implies, "decoded") in order to understand what operation needs to be performed.

2.2.3 Execute

The execute state is responsible for executing the correct operation based on the decoded instruction provided by the decode state.

There are some other considerations regarding the design of the control unit state machine that are worth noting. As referred in the register file design, due to it's latency when reading, there could be a need to add an extra cycle to cope with this condition. However, to avoid this delay in the execution, the decode estate was also used to read from the register file. With this being said, the only instructions that need an extra cycle are the ones that access the memory, so, an extra state follows those. This way, there is no wasted performance.

2.3 Memory

As mentioned in the analysis phase, following the Von Newman architecture, the code and data will be in the same memory. Therefore, it is important to define a good organization, as will be shown later.

According to Vespa specifications, the memory is byte addressable and the address and data bus is 32 bits.

The memory is single-port, which means that it has only one address input, one data input and one data output. Furthermore, it contains a write enable pin that allows writing and it also contains an output that indicates whether the memory has finished its initialization process after the reset.

In this project, an aligned memory will be implemented, which means that, as the memory is byte addressable, whenever there is an attempt to write or read at an address that is not divisible by 4, the 'error' signal is set to 1, alerting what happened. Later, this pin will be connected to the interrupt controller, thus allowing an exception to be raised.

Despite the address bus being 32 bits, it was decided to allocate 8192 bytes (2^{13}) of memory, half for code and the other half for data, as will be shown later.

Since the memory is located in a critical area of the datapath, it is necessary to implement the memory using a memory IP. According to the Vivado documentation, all available IPs have at least one cycle of latency when reading. Given this, and to maintain synchronization with the rest of the processor, it is necessary to overcome the effect of latency. Two alternatives and their respective advantages and disadvantages will be presented below.

Since it is byte addressable and the data is 32 bits, it would be necessary to write/read 4 bytes simultaneously. To do this, an IP RAM block with 4 ports was needed, as can be seen in figure 2.6. However, according to the software documentation, IPs have a maximum of 2 ports.

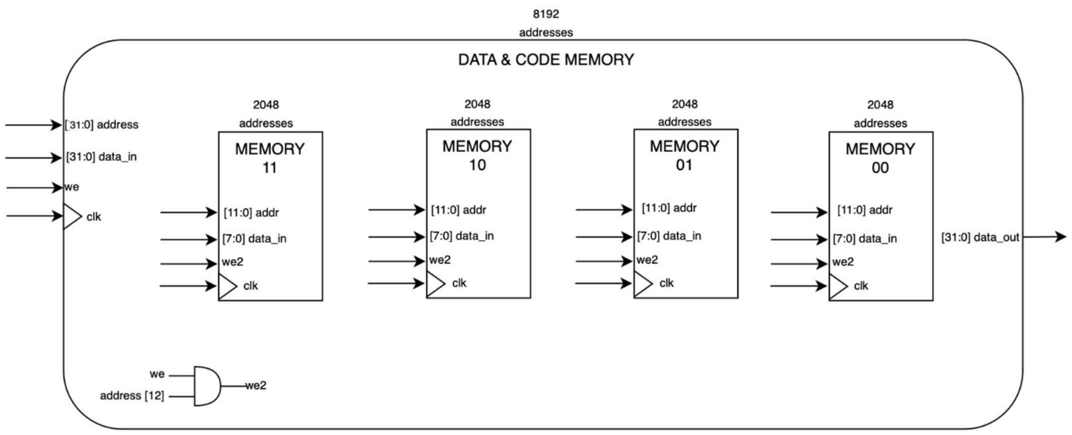


Figure 2.6: Memory design

This way, so that it is possible to write/read 4 bytes simultaneously, the memory will be composed of 4 instances of a single-port block RAM.

Thus, the last 2 bits of the address serve to identify which of the 4 instances should be accessed and the remaining 11 bits will be used to provide the address to all 4 block RAMs.

A disadvantage of this method is that each instruction has to be divided into 4 to be placed in each memory block. Then it will be necessary for the compiler to generate 4 different files with the divided instructions.

Option 1

Use a PLL (Phase-locked loop) or a MMCM (Mixed-Mode Clock Manager) to double the clock frequency for memory. An advantage of this method is that it will not be necessary to add extra states in the control unit for instructions that read from memory. However, this has a disadvantage. Normally, the instructions that take the longest to execute are those that access memory, so these will force instructions that take less time to operate at half the frequency established for the memory.

Option 2

Use an extra cycle only for instructions that read from memory. This alternative proves to be the most suitable, as it allows all instructions to operate at a higher frequency.

2.3.1 Memory organization

As referred before, the VeSPA follows a Von Newmann architecture, which makes a proper memory organization even more important. The design of the system's memory organization is displayed in figure 2.7 below. The upper half of the memory is a RAM and is dedicated to the data, and the lower half is a ROM and is dedicated to the code memory. This division is achieved by the interpretation of the address most significant bit. If the most significant bit is 1, the system can write to it. As displayed, the stack is mapped in the top of the data memory and it grows downwards according to the current needs of the system. There is also an area where the developed peripheral is mapped, as shown in the figure as well.

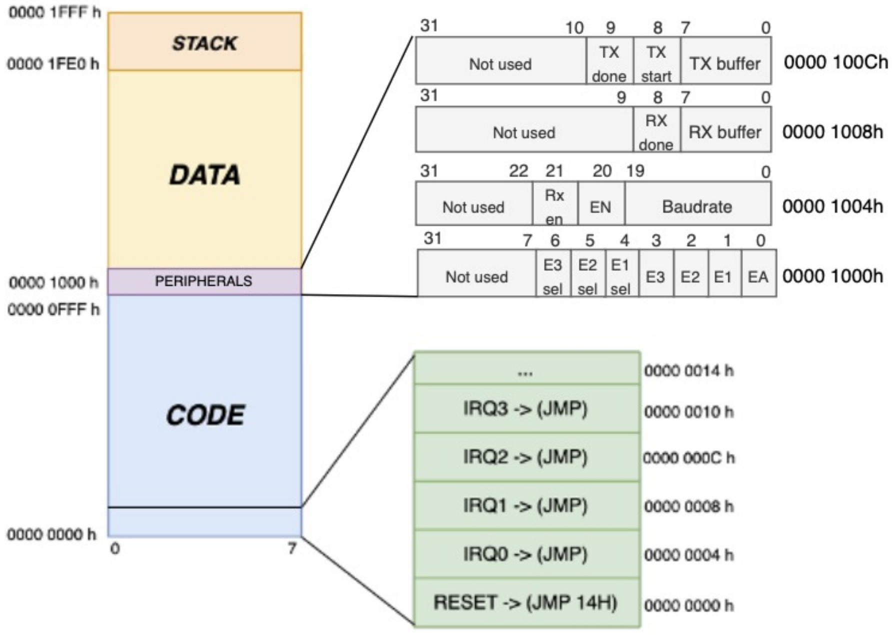


Figure 2.7: Memory organization design

2.4 Interrupt Controller

The interrupt controller to be developed has some particular features that should be specified to provide a better insight of it's inner-working:

- Capable of handling interruptions from up to 4 different sources;

- The interrupts have a fixed priority, where the interruption with the lowest index has the highest priority;
- Capable of handling the nesting of different interrupt sources.
- Capable of enabling and disabling the interrupts as a whole and also individually.

In order to implement the component in a way that it can support all of the referred features, the connection signals were established as displayed in figure 2.8 below.

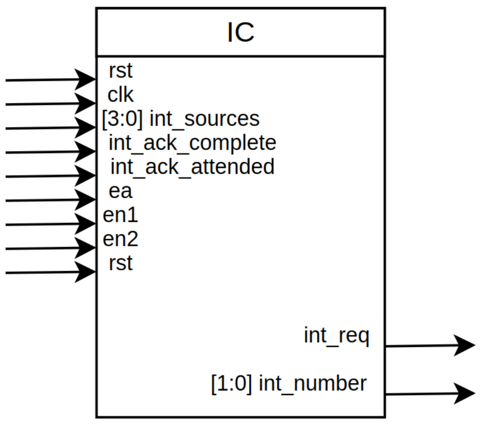


Figure 2.8: Interrupt controller connections diagram

2.4.1 Interrupt request handling

Considering the component's features and its design, an important trade-off regarding how the it's outputs are handled took place. When an interrupt request is received, there are three hypothesis on how to handle it:

Option 1:

An input sent directly from the control unit could be added to the multiplexer that mediates the data input of the PC, as displayed in figure 2.9, acting as the address of the correct IRQ on the interrupt vector. This option proves to be useful as it promotes a simpler implementation, however, there is a need to use an additional control bit for the multiplexer because of only one extra input.

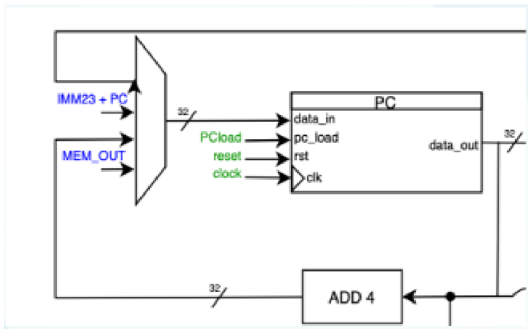


Figure 2.9: Additional input to PC data

Option 2:

Perform no changes to the current datapath and control the signals sent from the control unit in order to use the JMP (PC = Rx + imm16) intruction. This option has the downside of the need to use and extra clock cycle to load the Rx value and store the Rx on a stack.

Option 3:

Use the instruction JMP with link. This option proves to be useful as there is no need for a stack implementation and for a change in the datapath, however, there is the need of a dedicated register on the register file for interrupt handling, which would imply additional resource expenditure and a less flexible system.

Out of the three possible solutions, the chosen one was the first, adding an input sent directly from the control unit to the multiplexer that mediates the data input of the pc, given that it is a solution that implies a minimal change to the datapath and very little extra resource expenditure.

2.4.2 Stack

As referred throughout the presentation of this subject, when an interrupt is received, it typically, as the name implies, "interrupts" the current execution flow of the processor and executes the contents of its ISR, as long as it is enabled. The nesting of the interrupts consists in providing the capability of interrupting the execution of an interrupt's corresponding ISR, when an interrupt of a higher priority is received. A stack memory was implemented given that this feature implied that more than one program counter can possibly need to be stored, so that after the ISR's execution, it can go to where it had left off.

The behaviour of the designed interrupt controller is displayed in the following figure 2.10.

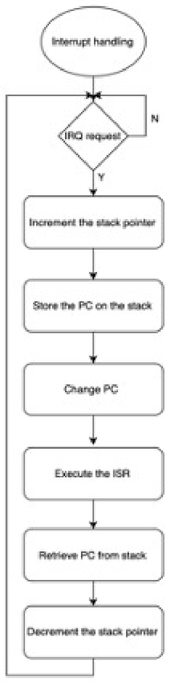

Interrupt handling
IRQ request
N
Y
Increment the stack pointer
Store the PC on the stack
Change PC
Execute the ISR
Retrieve PC from stack
Decrement the stack pointer

Figure 2.10: Interrupt controller behaviour

2.5 UART

The UART peripheral, which protocol was explained in the analysis, can be divided into 3 sub-modules:

- Baudrate generator;
- UART TX;
- UART RX.

The figure 2.11 presented below represents the connections between these referred modules.

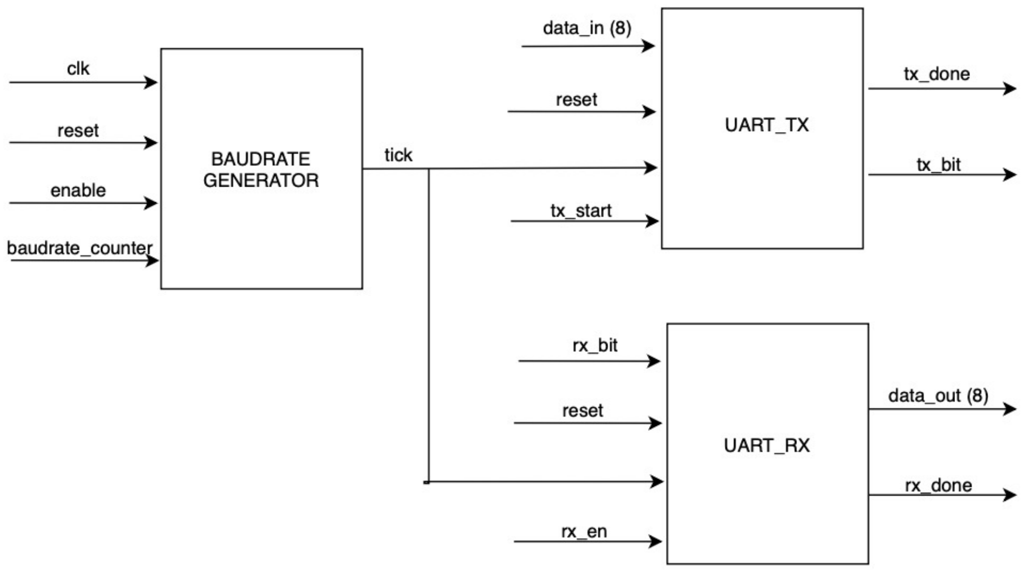


Figure 2.11: Connections between UART modules

In this project, a UART with configurable baudrate was implemented through a memory-mapped register (address 4100), however the data size is fixed (8 bits) and does not contain a parity bit.

2.5.1 Baudrate generator

As can be seen in the figure 2.12 below, the baudrate generator module receives as input a value that defines the baudrate (baudrate_counter). This value can be calculated with the following expression:

baudrate_counter = (clk->125Mhz) / (desired baudrate)

Using this value and the clock, the baudrate tick is generated.

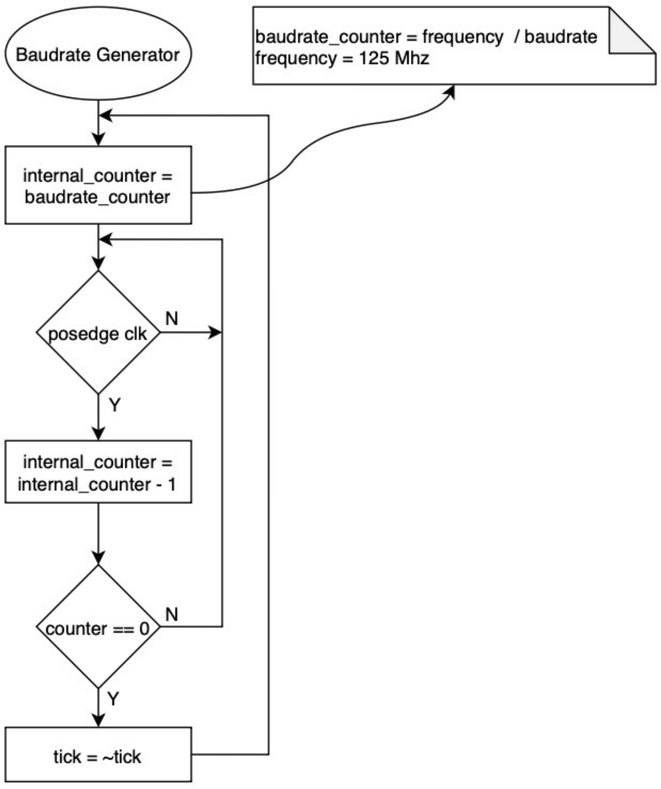

Baudrate Generator
baudrate_counter = frequency / baudrate
frequency = 125 Mhz
internal_counter = baudrate_counter
posedge clk
N
Y
internal_counter = internal_counter - 1
counter == 0
N
Y
tick = ~tick

Figure 2.12: Baudrate generator design

2.5.2 UART TX

The UART transmission module can be represented by the state machine in figure 2.13.

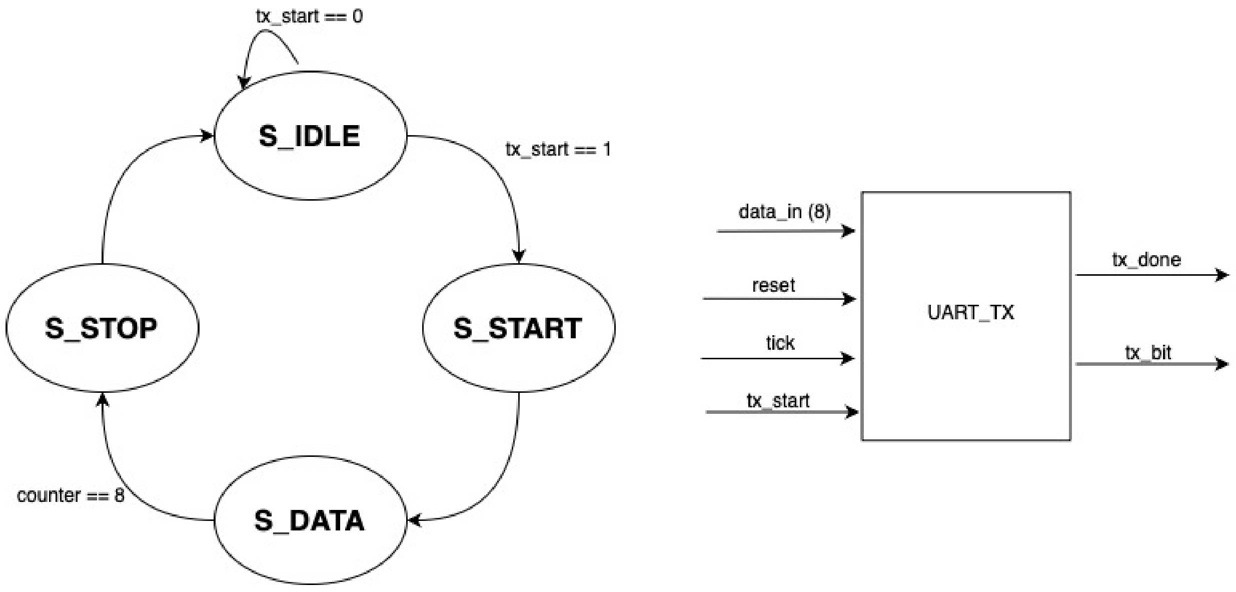


Figure 2.13: UART transmission module state machine

Initially, in the "IDLE" state, the system waits for the start bit to reach 1. After that, it switches to the

”START” state, where the output ”tx_bit” is set to 0 and where the data to be sent is placed in the buffer internal. It then enters the ”DATA” state where data is sent bit by bit. To do this, the internal buffer shifts right at each upward transition of the clock tick.

Finally, when the 8 bits are sent, it advances to the ”STOP” state where the ”tx_bit” is set to 1, advancing again to the ”IDLE” state.

2.5.3 UART RX

The UART reception module is represented by the state machine displayed in figure 2.14 below.

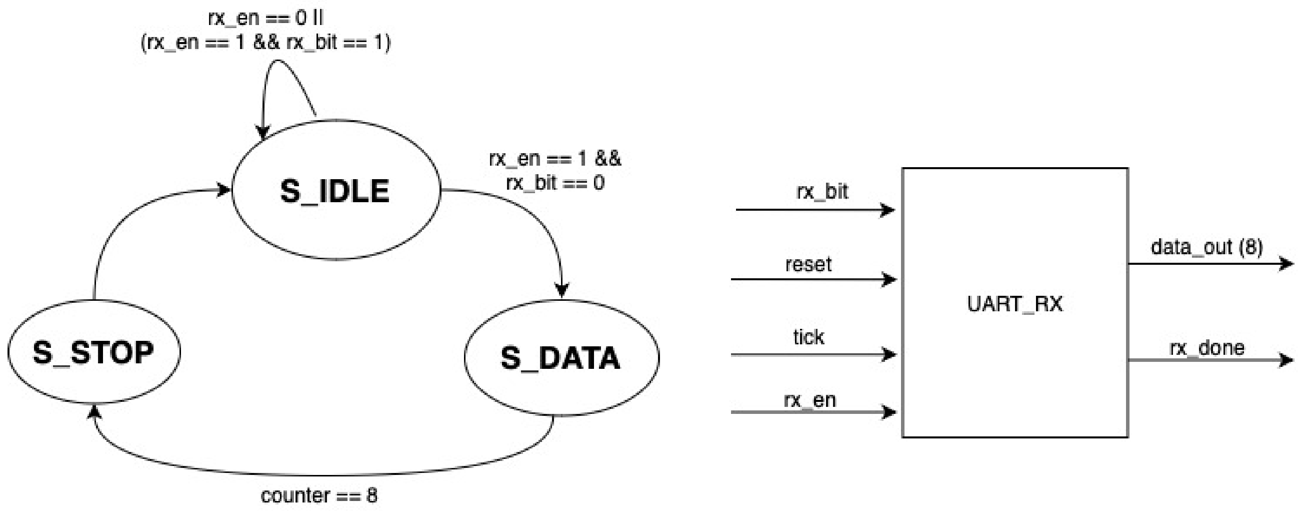


Figure 2.14: UART reception module state machine

In the beginning, the state ”IDLE” waits for the ”rx_bit” input to be set, which is the start bit. After that, if the reception is enabled, the state is then ”DATA” where the value of the ”rx_bit” input is stored at every positive edge of the clock in the most significant position of an internal buffer, shifting it’s content to the right.

After receiving 8 bits, the state turns to ”STOP”, where the buffer value is inserted in the output, before returning to the state ”IDLE”

2.5.4 System integration

Having specified the design for each of the modules that compose the whole system, it becomes relevant to specify how they are connected between each other and how they communicate. Figure 2.15 displays how all of the referred modules are connected as well as the connections of their previously displayed respective wires.

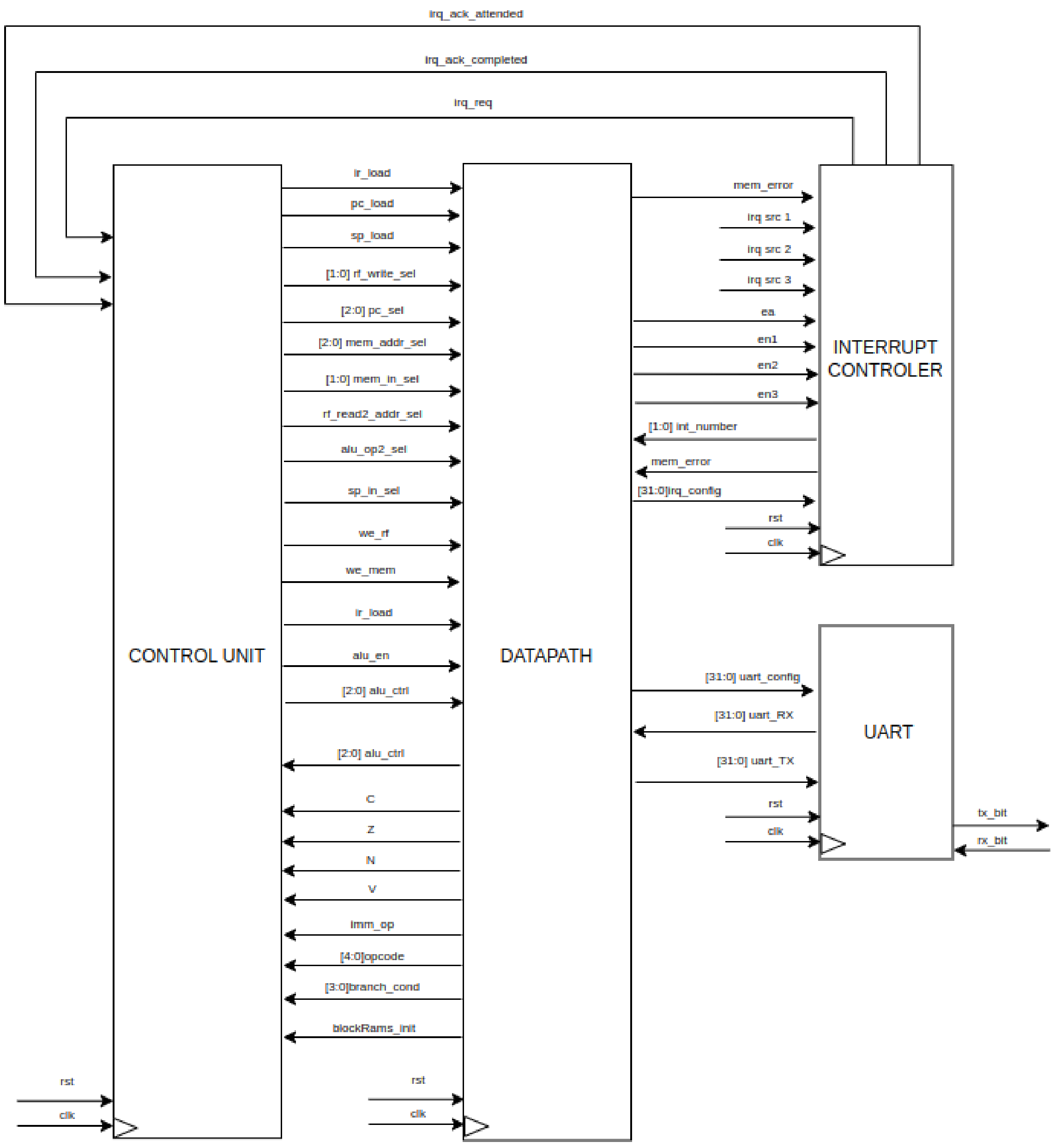


Figure 2.15: Integrated system modules

3 | Implementation and Verification

3.1 Datapath

The datapath is a very important part of the processor. It is responsible for executing all instructions through signals coming from the control unit. The fetch process is executed there, as illustrated in the code below.

```
//update instruction register when ir_load = 1
always @(posedge clk) begin
    if(reset == 1'b1) begin
        IR = 0;
    end
    else if(ir_load == 1'b1) begin
        IR = mem_out;
    end
end


//update PC value
always @(posedge clk) begin
    if(reset == 1'b1) begin
        PC = `PC_INIT;
    end
    else if(pc_load == 1'b1) begin
        PC = pc_in;
    end
end
```

Algoritmo 3.1: Program counter and instruction register

As shown in the design phase, it is also composed of a series of multiplexers important for selecting some inputs, for example. Some of these multiplexers are represented in the code below and are also available in the attachment.

```
//alu operator 2 mux
assign alu_op2 = alu_op2_sel ? imm16_extend : rf_out2;

//alu data input mux
assign rf_write_data = (rf_write_sel == 2'b00) ? imm22_extend :
                       (rf_write_sel == 2'b01) ? PC :
                       (rf_write_sel == 2'b10) ? mem_out : alu_res;


//jump address (rs1 + imm16)
wire [`REG_MSB:0]jmp_addr;
assign jmp_addr = rf_out1 + imm16_extend;


//program counter input mux
wire [`REG_MSB:0]pc_in;
assign pc_in = (pc_sel == 3'b000) ? (jmp_addr) :
               (pc_sel == 3'b001) ? (PC + imm23_extend) :
               (pc_sel == 3'b010) ? (PC + 4) :
               (pc_sel == 3'b011) ? (int_number << 2) + 4 : mem_out;


//memory address mux
wire [`REG_MSB:0]mem_addr;
assign mem_addr = (mem_addr_sel == 3'b000) ? SP :
                  (mem_addr_sel == 3'b001) ? imm22_extend :
                  (mem_addr_sel == 3'b010) ? (imm17_extend + rf_out1) :
                  (mem_addr_sel == 3'b011) ? 32'd4104 : PC;


//memory data input mux
wire [`REG_MSB:0]mem_in;
assign mem_in = (mem_in_sel == 2'b00) ? PC :
                (mem_in_sel == 2'b01) ? uartRX_data: rf_out2;
```

Algoritmo 3.2: Datapath multiplexers

Furthermore, the ALU, register file and memory module are instantiated in it.

3.1.1 ALU

As specified in the design the ALU has 3 inputs: OP1 (operand 1), OP2 (operand 2) and alu_operation. These are interpreted by the ALU, the result is sent through the output and the condition codes are updated accordingly. The implementation of this module is presented in the listing below.

```verilog
//alu module
module alu(
    input reset,
    input [`REG_MSB:0] op1,
    input [`REG_MSB:0] op2,
    input [2:0] operation,
    input enable,
    output reg C,
    output reg Z,
    output reg N,
    output reg V,
    output reg [`REG_MSB:0] res
    );


    //ALU INSTRUCTION CODE DEFINE
    `define ADD_ALU 3'd0
    `define SUB_ALU 3'd1
    `define OR_ALU  3'd2
    `define AND_ALU 3'd3
    `define NOT_ALU 3'd4
    `define XOR_ALU 3'd5


    //check if is a sub operation
    //used to update ovrflow flag
    wire sub;
    assign sub = (operation == `SUB_ALU) ? 1'b1 : 1'b0;

    //clear all condition flags
    initial begin
        C <= 0;
        Z <= 0;
        N <= 0;
        V <= 0;
        res <= 0;
    end


    always@(*) begin
        if (reset == 1'b1) begin
            C <= 0;
            Z <= 0;
            N <= 0;
            V <= 0;
            res <= 0;
        end
        else if(enable == 1'b1) begin
            case(operation)
                //sum operation
                 `ADD_ALU:
                    {C, res} = op1 + op2;

                //subtraction operation
                 `SUB_ALU:
                    {C, res} = op1 - op2;

                //logic OR operation
                 `OR_ALU:
                     res = op1 | op2;

                //logic AND operation
                 `AND_ALU:
                     res = op1 & op2;

                //logic NOT operation
                 `NOT_ALU:
                     res = ~op1;

                //logic XOR operation
                 `XOR_ALU:
                     res = op1 ^ op2;

             endcase

             //update zero flag
             Z <= ~(|res);

             //update negative flag
             N <= res[`REG_MSB];

             //update overflow flag if is ADD OR SUB operation
             if(operation == `ADD_ALU || operation == `SUB_ALU) begin
```

```
                V <= (res[`REG_MSB] & ~op1[`REG_MSB] & ~(sub ^ op2[`REG_MSB]))
                    |(~res[`REG_MSB] & ~op1[`REG_MSB] & (sub ^ op2[`REG_MSB]));
             end
         end
    end
endmodule
```

Algoritmo 3.3: ALU implementation

To test this implementation, the simulation displayed in figure 3.2 was performed. From the simulation it is possible to verify that the module is working correctly as the condition codes are being set and reset as they should according to the operations performed.

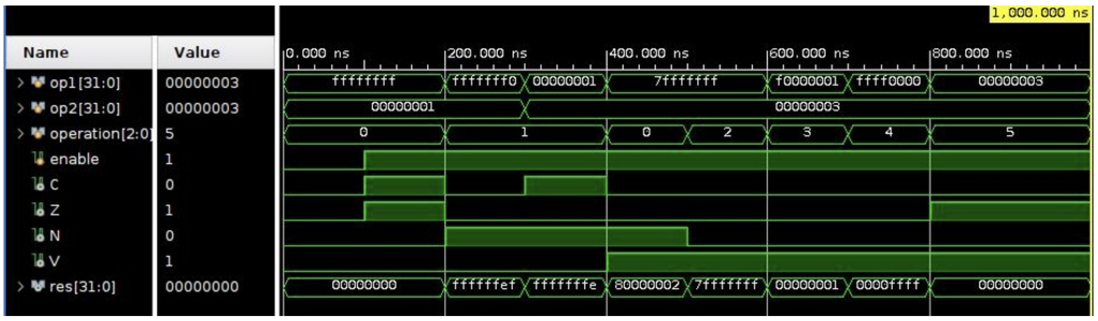


Figure 3.1: ALU simulation

3.1.2 Check Branch Condition

As referred in the design phase, this functional unit has the branch condition that comes from the instruction register and the condition codes as inputs and a single output that signals if the condition has been checked or not, which ultimately will result in the execution of the branch or not. The implementation of this component is displayed in the listing below.

```
//check branch condition module
module check_cond(
    input [3:0]cond,
    input c,
    input z,
    input n,
    input v,
    output reg out
    );

     always @(*) begin
        case (cond)
            `BRA:   //branch always
                out <= 1;

            `BNV:   //branch never
                out <= 0;

            `BCC:   //branch if not carry
                out <= ~c;

            `BCS:   //branch if carry
                out <= c;

            `BVC:   //branch if not overflow
                out <= ~v;

            `BVS:   //branch if overflow
                out <= v;

            `BEQ:   //branch if equal
                out <= z;

            `BNE:   //branch if not equal
                out <= ~z;

            `BGE:   //branch if greater than or equal
                out <= (~n & ~v) | (n & v);

            `BLT:   //branch if less than
```

```
                out <= (n & ~v) | (~n & v);

            `BGT:   //branch if greater than
                out <= ~z & ((~n & ~v) | (n & v));

            `BLE:   //branch if less than or equal
                out <= z | ((n & ~v) | (~n & v));

            `BPL:   //branch if plus
                out <= ~n;

            `BMI:   //branch if minus
                out <= n;
        endcase
    end
endmodule
```

Algoritmo 3.4: Check branch condition implementation

To test this implementation, the simulation displayed in figure 3.2 was performed. From the simulation it is possible to verify that the module is working correctly as the "out" signal is set when the branch condition and the condition codes match.

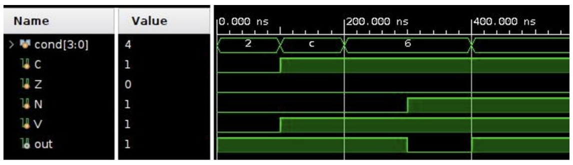
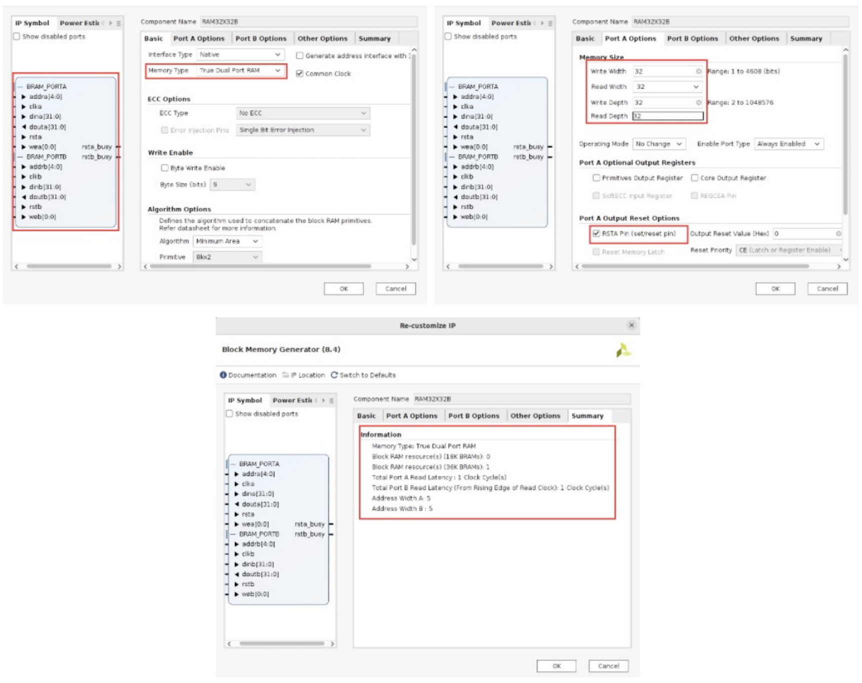


Figure 3.2: Check condition simulation

3.1.3 Register file

To implement the register file, the first thing was using Vivado's memory configurator, configuring the settings as displayed in figure 3.3. We started by defining the memory as 'True Dual Port Ram' and changed the size and number of registers to 32. The 'RSTA Pin' option was activated so that it was possible to reset the memory and have a output that indicates when the memory has finished its initialization process.



Component Name RAM32X32B

Interface Type: Native
Memory Type: True Dual Port RAM
Generate address interface with
Common Clock ✓

ECC Options
ECC Type: No ECC
Error Injection Pins: Single Bit Error Injection

Write Enable
Byte Write Enable
Byte Size (bits): 9

Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm: Minimum Area
Primitive: 8kx2

Memory Size
Write Width: 32 — Range: 1 to 4608 (bits)
Read Width: 32
Write Depth: 32 — Range: 2 to 1048576
Read Depth: 32

Operating Mode: No Change
Enable Port Type: Always Enabled

Port A Optional Output Registers
Primitives Output Register
Core Output Register
SoftECC Input Register
REGCEA Pin

Port A Output Reset Options
RSTA Pin (set/reset pin) ✓
Output Reset Value (Hex): 0
Reset Memory Latch
Reset Priority: CE (Latch or Register Enable)

Information
Memory Type: True Dual Port RAM
Block RAM resource(s) (18K BRAMs): 0
Block RAM resource(s) (36K BRAMs): 1
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: 5
Address Width B : 5

Figure 3.3: Register file IP configurations

The following module was then developed where 2 blocks of memory were instantiated and connected in order to create the register file by copy as explained in the design.

```verilog

`define REG_MSB     31      //register most significant bit
`define RF_SIZE     32      //register file size


//register file module
module registerFile(
    input clk,
    input rst,
    input we,
    input [4:0]w_addr,
    input [4:0]r1_addr,
    input [4:0]r2_addr,
    input [`REG_MSB:0]data_in,
    output [`REG_MSB:0]read1_out,
    output [`REG_MSB:0]read2_out,
    output initialized
    );


    wire rsta_busy_rf1;
    wire rsta_busy_rf2;
    wire rstb_busy_rf1;
    wire rstb_busy_rf2;

    assign initialized = (rsta_busy_rf1 & rstb_busy_rf1 & rsta_busy_rf2 & rstb_busy_rf2);

   RAM32X32B rf1(.clkb(clk),
                 .rsta(rst),
                 .rstb(rst),
                 .web(we),
                 .addrb(w_addr),
                 .dinb(data_in),
                 .clka(clk),
                 .wea(1'b0),
                 .addra(r1_addr),
                 .douta(read1_out),
                 .dina(0),
                 .rsta_busy(rsta_busy_rf1),
                 .rstb_busy(rstb_busy_rf1)
                 );

   RAM32X32B rf2(.clkb(clk),
                 .rsta(rst),
                 .rstb(rst),
                 .web(we),
                 .addrb(w_addr),
                 .dinb(data_in),
                 .clka(clk),
                 .wea(1'b0),
                 .addra(r2_addr),
                 .douta(read2_out),
                 .dina(0),
                 .rsta_busy(rsta_busy_rf2),
                 .rstb_busy(rstb_busy_rf2)
                );

endmodule
```

Algoritmo 3.5: UART TX implementation

As can be seen in the summary of figure 3.3, the reading latency on both ports is 1 clock cycle, however on port B it indicates that it is from the rising edge of the reading clock. So, for testing purposes, the register file without IP was developed, so that latency could be checked.

Then, a test bench was carried out and simulated with the register file without IP, obtaining the result shown in figure 3.4.

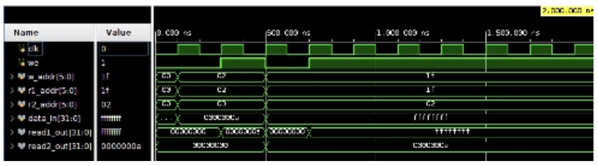


Figure 3.4: Register file without memory IP simulation

Then, using the same test bench, but changing the register file with IP, the result in figure 3.5 was obtained. It can be seen that in fact the reading is not done correctly and that there is in fact a delay cycle.

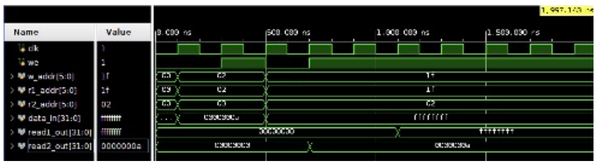


Figure 3.5: Register file with memory IP simulation

Then, the clock frequency for the memory was doubled and the reading was done from port A and the result in figure 3.6 was obtained. In this way, it can be seen that the result is now the same as in memory without IP.

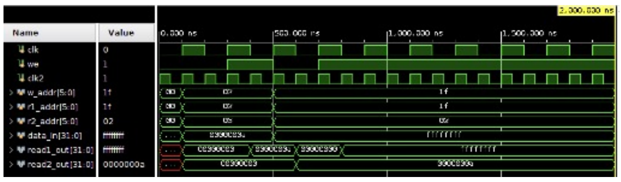


Figure 3.6: Register file with increased frequency and reading from port A

However, to verify the effect of latency on port B from the rising transition of the reading clock cycle, the reading was moved to port B, verifying that there is a greater delay, as displayed in figure 3.7.

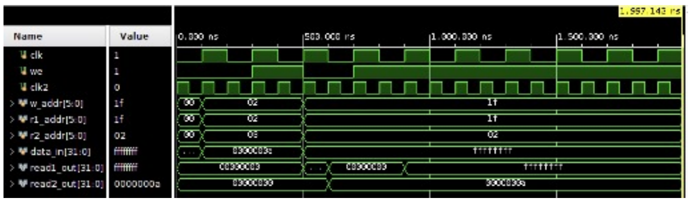


Figure 3.7: Register file with increased frequency and reading from port B

Taking these simulations in account, it was decided to place reading from port A and writing from port B.

3.2 Control Unit

As presented in the design phase, the control unit is implemented through the use of a state machine. Said state machine's implementation is presented in the listing displayed below.

```
    always @(posedge clk) begin
        if(rst == 1'b1) begin
            state <= s_START;
        end

        else begin
            case(state)
                s_START: begin
                    if(blockRams_init == 1'b0)
                        state <= s_FETCH;
                    else
                        state <= s_START;
                end

                s_FETCH: begin
                    state <= s_DECODE;
                end

                s_DECODE: begin
                    if(opcode == s_BXX && branch_checked == 1'b0)
                        state <= s_FETCH;
                    else
                        state <= opcode;
                end

                 s_HLT:
                    state <= s_HLT;

                 s_IRQ1: begin
                    state <= s_IRQ2;
                 end

                default: begin
                    if(state == s_LD || state == s_LDX || state == s_ST || state == s_STX || state == s_PUSH ||
                       state == s_POP || state == s_JMP || state == s_RETI || state == s_IRQ2 || state == s_BXX) begin

                        state <= s_EXTRA;
                    end

                    else if(int_req) begin
                        state <= s_IRQ1;
                    end

                    else
                        state <= s_FETCH;
                 end
            endcase
        end
    end
```

Algoritmo 3.6: UART TX implementation

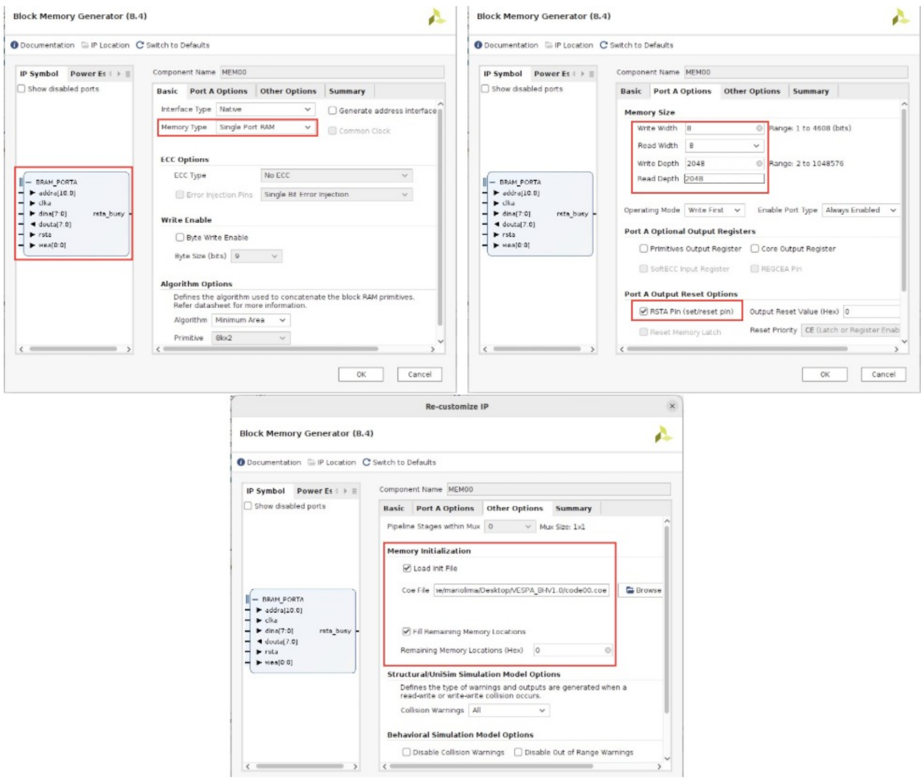
As referred in a previous chapter of this document, the control unit sends specific signals to the datapath according to the current state of the system. The implementation of this mechanism is displayed in the listing presented below.

```
    assign ir_load = (state == s_FETCH) ? 1'b1 : 1'b0;

    assign pc_load = (state == s_JMP || state == s_FETCH || state == s_BXX || state == s_IRQ2 || state == s_RETI) ? 1'b1 : 1'b0;

    assign pc_sel = (state == s_JMP) ? 3'b000 :
                    (state == s_BXX) ? 3'b001 :
                    (state == s_FETCH) ? 3'b010 :
                    (state == s_IRQ2) ? 3'b011 : 3'b100;

    assign rf_read2_addr_sel = (~(state == s_FETCH) && ~(state == s_EXTRA) && (opcode == s_ST || opcode == s_STX)) ? 1'b1 : 1'b0;

    assign rf_write_sel = (state == s_JMP && imm_op == 1'b1) ? 2'b01 :
                          (state == s_LD || state == s_LDX)  ? 2'b10 :
                          (state == s_LDI) ? 2'b00 : 2'b11;

    assign we_mem = (state == s_ST || state == s_STX || state == s_IRQ2 || state == s_MAP) ? 1'b1 : 1'b0;

    assign we_rf = (state == s_ADD || state == s_SUB || state == s_OR  ||
                    state == s_AND || state == s_NOT || state == s_XOR ||
                    state == s_LD  || state == s_LDI || state == s_LDX ||
                    state == s_JMP && imm_op == 1'b1) ? 1'b1 : 1'b0;

    assign alu_ctrl = (state == s_SUB || state == s_CMP) ? 3'b001 :
                      (state == s_OR)  ? 3'b010 :
                      (state == s_AND) ? 3'b011 :
                      (state == s_NOT) ? 3'b100 :
                      (state == s_XOR) ? 3'b101 : 3'b000;

    assign alu_op2_sel = (imm_op == 1'b1) ? 1'b1 : 1'b0;

    assign alu_en = (state == s_ADD || state == s_SUB || state == s_OR  ||
                     state == s_AND || state == s_NOT || state == s_XOR ||
                     state == s_CMP) ? 1'b1 : 1'b0;

    assign sp_load = (state == s_IRQ1 || state == s_POP ||
                      state == s_PUSH || state == s_RETI) ? 1'b1 : 1'b0;

    assign sp_in_sel = (state == s_IRQ1 || state == s_PUSH) ? 1'b0 : 1'b1;

    assign mem_addr_sel = ((state == s_DECODE && (opcode == s_RETI || opcode == s_POP)) || state == s_RETI ||
                            state == s_POP || state == s_IRQ2) ? 3'b000 :
                          ((state == s_DECODE && opcode == s_LD) || state == s_LD || state == s_ST)   ? 3'b001 :
                          ((state == s_DECODE && opcode == s_LDX) || state == s_LDX ||state == s_STX) ? 3'b010 :
                          (state == s_MAP) ? 3'b011 : 3'b100;

    assign mem_in_sel = (state == s_IRQ2 || state == s_IRQ1) ? 2'b00 :
                        (state == s_MAP) ? 2'b01 : 2'b10;

    assign int_ack_complete = (state == s_RETI) ? 1'b1 : 1'b0;

    assign int_ack_attended = (state == s_IRQ2) ? 1'b1 : 1'b0;
```

Algoritmo 3.7: UART TX implementation

3.3 Memory

As mentioned in the design, to implement byte-addressable memory, it was necessary to instantiate 4 block RAMs with 2048 addresses each. In figure 3.8 below you can see the settings made in the Vivado memory configurator.


Block Memory Generator (8.4)

Component Name MEM00

Basic

Interface Type: Native
Memory Type: Single Port RAM
☐ Generate address interface
☐ Common Clock

ECC Options
ECC Type: No ECC
☐ Error Injection Pins: Single Bit Error Injection

Write Enable
☐ Byte Write Enable
Byte Size (bits): 9

Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives. Refer datasheet for more information.
Algorithm: Minimum Area
Primitive: 8kx2

Port A Options

Memory Size
Write Width: 8 — Range: 1 to 4608 (bits)
Read Width: 8
Write Depth: 2048 — Range: 2 to 1048576
Read Depth: 2048

Operating Mode: Write First
Enable Port Type: Always Enabled

Port A Optional Output Registers
☐ Primitives Output Register ☐ Core Output Register
☐ SoftECC Input Register ☐ REGCEA Pin

Port A Output Reset Options
☑ RSTA Pin (set/reset pin)
Output Reset Value (Hex): 0
☐ Reset Memory Latch
Reset Priority: CE (Latch or Register Enab

Other Options

Pipeline Stages within Mux: 0 — Mux Size: 1x1

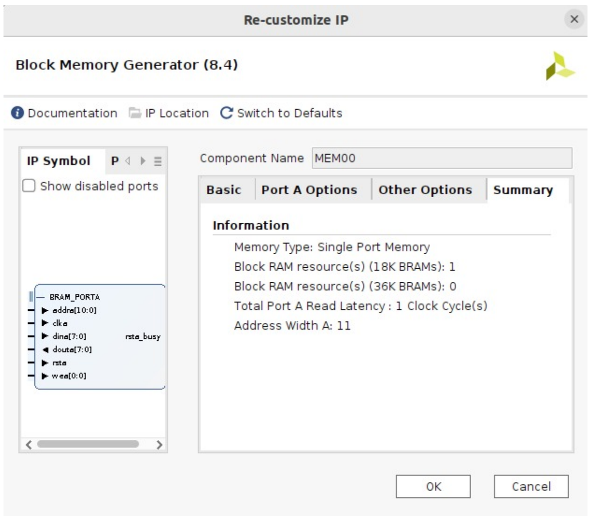
Memory Initialization
☑ Load Init File
Coe File: ıe/mariolima/Desktop/VESPA_BHV1.0/code00.coe
☑ Fill Remaining Memory Locations
Remaining Memory Locations (Hex): 0

Structural/UniSim Simulation Model Options
Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.
Collision Warnings: All

Behavioral Simulation Model Options
☐ Disable Collision Warnings ☐ Disable Out of Range Warnings

Figure 3.8: Memory IPs configurations

After these configurations, it is possible to check all the characteristics of the generated IP in the memory configurator summary, as displayed in figure 3.9 below.


Block Memory Generator (8.4)

Component Name MEM00

Summary

Information
Memory Type: Single Port Memory
Block RAM resource(s) (18K BRAMs): 1
Block RAM resource(s) (36K BRAMs): 0
Total Port A Read Latency : 1 Clock Cycle(s)
Address Width A: 11

Figure 3.9: Configured memory IPs characteristics

Below is the code created, where you can check the four block RAM instances. A check has also been added to see if an attempt is made to write code memory and if an unaligned access attempt occurs.

```
module memorySynthesis(
    input clk,
    input rst,
    input we,
    input [`REG_MSB:0]addr,
    input [`REG_MSB:0]data_in,
    output [`REG_MSB:0]data_out,
    output initialized,
    output error
    );

    assign error = (addr[1:0] != 2'b00 || (we && addr[`USED_MEM_MSBT] == 1'b0)) ? 1'b1 : 1'b0;

    wire [10:0]addr_aux;
    assign addr_aux = addr[`USED_MEM_MSBT:2];


    wire we_aux;
    assign we_aux = we;


    wire rsta_busy00;
    wire rsta_busy01;
    wire rsta_busy10;
    wire rsta_busy11;

    assign initialized = (rsta_busy00 & rsta_busy01 & rsta_busy10 & rsta_busy11);  //memory init

    //..................BIG-ENDIAN..........

    //first 8 most significant bits of data
    MEM00 mem0(.clka(clk),
               .rsta(rst),
               .wea(we_aux),
               .addra(addr_aux),
               .dina(data_in[31:24]),
               .douta(data_out[31:24]),
               .rsta_busy(rsta_busy00)
               );

    //second 8 most significant bits of data
    MEM01 mem1(.clka(clk),
               .rsta(rst),
               .wea(we_aux),
               .addra(addr_aux),
               .dina(data_in[23:16]),
               .douta(data_out[23:16]),
               .rsta_busy(rsta_busy01)
               );

    //third 8 most significant bits of data
    MEM10 mem2(.clka(clk),
               .rsta(rst),
               .wea(we_aux),
               .addra(addr_aux),
               .dina(data_in[15:8]),
               .douta(data_out[15:8]),
               .rsta_busy(rsta_busy10)
               );

    //8 less significant bits of data
    MEM11 mem3(.clka(clk),
               .rsta(rst),
               .wea(we_aux),
               .addra(addr_aux),
               .dina(data_in[7:0]),
               .douta(data_out[7:0]),
               .rsta_busy(rsta_busy11)
               );
endmodule
```

Algoritmo 3.8: Memory implementation using IP

For the purpose of testing the need for an extra cycle when reading memory by IP, the memory was also implemented without using an IP, as shown in the code below.

```
`define MEM_REG_MSB     7       //memory register most significant bit
`define REG_MSB         31      //register most significant bit
`define MEM_SIZE        (1<<8)  //memory size
`define USED_MEM_MSB    7      // address most significant bit


module memory(
    input clk,
    input we,
    input rst,
    input [`REG_MSB:0]addr,
    input [`REG_MSB:0]data_in,
    output [`REG_MSB:0]data_out
    );

    reg [`MEM_REG_MSB:0] MEM [`MEM_SIZE - 1:0];

    always @(posedge clk) begin
        if(we==1'b1 && addr[`USED_MEM_MSB]==1'b1 && addr<`MEM_SIZE) begin
            $display(addr);
            {MEM[addr], MEM[addr+1], MEM[addr+2], MEM[addr+3]} = data_in;   //big-endian
        end
    end

    always @(posedge clk) begin
       if (rst == 1'b1) begin
           {MEM[0], MEM[1], MEM[2], MEM[3]} = {5'd11, 5'd1, 22'd4};                          //LDI: R1 = 4
           {MEM[4], MEM[5], MEM[6], MEM[7]} = {5'd9, 5'd7, 5'd1, 1'd1, 16'd12};              //JMPL 12 + r1(4) -- L:R1
           {MEM[16], MEM[17], MEM[18], MEM[19]} = {5'd7, 5'd0, 5'd1, 1'd1, 16'd4};           //CMP:
           {MEM[20], MEM[21], MEM[22], MEM[23]} = {5'd8, 4'd3, 23'd16};                      //BEQ
           {MEM[40], MEM[41], MEM[42], MEM[43]} = {5'd0, 27'd0};                             //NOP
           {MEM[44], MEM[45], MEM[46], MEM[47]} = {5'd11, 5'd2, 22'd3};                      //LDI: R2 = 3
           {MEM[48], MEM[49], MEM[50], MEM[51]} = {5'd1, 5'd3, 5'd1, 1'd0, 5'd2, 11'd0};     //ADD: R3 = R1 + R2
           {MEM[52], MEM[53], MEM[54], MEM[55]} = {5'd31, 27'd0};                            //HALT
       end
    end

    assign data_out = {MEM[addr], MEM[addr+1], MEM[addr+2], MEM[addr+3]};


    initial begin
        $readmemb("/home/mariolima/Desktop/embedded_systems/VESPA_BHV/code.txt", MEM);
    end
endmodule
```

Algoritmo 3.9: Memory implementation for testing

After simulating, the following results were obtained (figure 3.10).

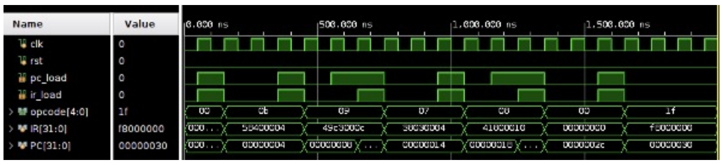


Figure 3.10: Initial memory simulation results

Then, using the same test bench, it was changed to memory with IP and simulated. As seen in figure 3.11, the instructions are not being read correctly.

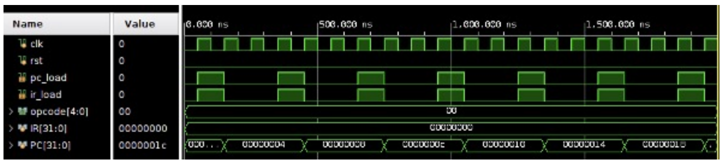


Figure 3.11: Memory simulation results using IP

Then an extra cycle was added and, in this way, the same result was obtained as when simulating memory without IP (figure 3.12).

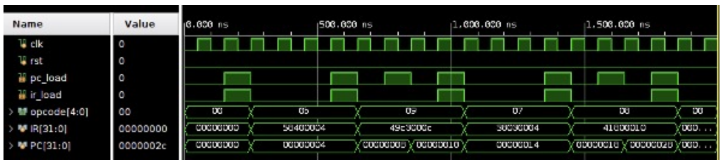


Figure 3.12: Memory simulation results using IP and extra cycle

As mentioned previously, one of the disadvantages of the memory being made up of 4 different blocks is that writing code in code memory can be a complicated task, as it is necessary to write the instruction in binary, and then divide it into 4 to place in each .coe file. To overcome this problem, a Python script was created so that it was possible to write the desired code in assembly and then automatically convert it to binary and divide it into different .coe files. The following code shows this script that can later be used by the compiler.

```python

#function that makes the following steps:
#  - read code.txt file
#  - parsing the instruction
#  - convert to binary and and write in binarycode.txt
#  - divide the binary code and write in each .coe file
def parse_and_generate_coe(input_filename, output_filenames):
    files = []
    first = True

    with open(input_filename, 'r') as input_file:

        for file_name in output_filenames:
            id = open(file_name, 'w')
            id.write("memory_initialization_radix=2;\n")        #coe header -> data format
            id.write("memory_initialization_vector= \n")
            files.append(id)


        for line in input_file:
            words = line.split()

            instruction = words[0]
            operands = words[1:]

            print(f"parse: {instruction}, {operands}")

            binary_representation = convert_instruction_to_binary(instruction, operands)

            files[4].write(binary_representation + "\n")

            binary = int(binary_representation, 2)

            print(f"Binary Representation: {binary_representation}")
            print(f"Binary : {binary}")

            divided_bytes = divide_instruction(binary)

            print(f"Divided Bytes: {divided_bytes}")

            if first:
                first = False

            else:
                files[0].write(",\n")
                files[1].write(",\n")
                files[2].write(",\n")
                files[3].write(",\n")

            files[3].write(format(int(divided_bytes[0]), '08b'))
            files[2].write(format(int(divided_bytes[1]), '08b'))
            files[1].write(format(int(divided_bytes[2]), '08b'))
            files[0].write(format(int(divided_bytes[3]), '08b'))


        files[0].write(";")
        files[1].write(";")
        files[2].write(";")
        files[3].write(";")

        files[0].close()
        files[1].close()
        files[2].close()
        files[3].close()

if __name__ == "__main__":
    print("START")

    input_filename = "code.txt"
    output_filenames = ["code00.coe", "code01.coe", "code10.coe", "code11.coe", "binaryCode.txt"]
    parse_and_generate_coe(input_filename, output_filenames)
```

Algoritmo 3.10: Developed python script

This way, it is possible to write the code as follows and then generate the .coe files.

```
    JMP 0 20            //reset
    JMP 0 208           //irq source 1
    JMP 0 160           //irq source 2
    JMP 0 88            //irq source 3
    JMP 0 208
    LDI 0 0
    LDI 30 3158749
    ST 30 4100
    LDI 30 95
    ST 30 4096
    LDI 4 0
    LDI 2 0
    LDI 3 0
    LDI 1 0
    XORI 1 1 1
    SUBI 31 4 1
    BEQ 12
    NOP
    NOP
    JMP 0 48
    HALT
    NOP
    NOP
    NOP
    ADDI 2 2 1
    SUBI 31 2 10
    BEQ 16
    ADDI 3 2 48
    ORI 3 3 256
    ST 3 4108
    RETI
    NOP
```

Algoritmo 3.11: Python script utilization

3.4 Interrupt Controller

For the nested interrupt controller, the following module was developed. Briefly, whenever there is an upward transition from one of the sources, the pending interruptions variable is updated. Furthermore, according to the pending interruptions and their priority, the variable 'nextIrq' is updated with the number of the interruption to be serviced next. If there are no interruptions to be attended to, the variable has the value of 4 so that it is a value greater than the number of interruptions. It is then checked whether the interrupt being serviced has a lower priority than the next one. If this happens, a request is sent to the control unit.

Then, when the control unit responds to the interrupt, the controller receives an ack and sets the completed bit to 0 to indicate that the interrupt has started to be serviced.

If, however, a higher priority interrupt is received, the process repeats itself, sending a new request.

```
module interrupt_controller(
    input rst,
    input clk,
    input [3:0] int_sources,
    input int_ack_complete,
    input int_ack_attended,
    output reg int_req,
    output [1:0]int_number,
    input ea,
    input en1,
    input en2,
    input en3
);


wire [3:0] int_sources_2;
wire [2:0]nextIrq;
reg [2:0]currentIrq;
reg [3:0]pending;
reg [3:0]completed;
reg [3:0]int_sources_prev;


//sources activation
assign int_sources_2[0] = int_sources[0] && ea;
assign int_sources_2[1] = int_sources[1] && en1 && ea;
assign int_sources_2[2] = int_sources[2] && en2 && ea;
assign int_sources_2[3] = int_sources[3] && en3 && ea;


//checks the next interrupt to be attended according to priorities
assign nextIrq = (pending[3] && ~pending[2] && ~pending[1] && ~pending[0]) ? 3'b011 :
                 (pending[2] && ~pending[1] && ~pending[0]) ? 3'b010 :
```

```
                (pending[1] && ~pending[0]) ? 3'b001 :
                (pending[0]) ? 3'b000 : 3'b100;


//asynchronous source checking
always@(*) begin
    if(rst == 1'b1) begin
        pending <= 4'b0000;
        int_sources_prev <= 4'b0000;
    end
    else if(int_sources_2 != int_sources_prev) begin      //if any of the sources changed state
        pending <= pending | int_sources_2;               //when the source reaches zero it does not change
        int_sources_prev <= int_sources_2;                //so changes only on rising edge
    end
    else if(int_ack_complete == 1'b1) begin             //if receive ack, clear pending
        pending <= pending & ~(4'b0001 << currentIrq);
    end
end



always @(posedge clk) begin
    if(rst == 1'b1) begin
        int_req <= 1'b0;
        currentIrq <= 3'b100;
        completed <= 4'b1111;
    end
    else begin
        if(int_ack_complete == 1'b1) begin                  //ehen receive an ack, update completed variable
            completed <= completed | (4'b0001 << currentIrq);
            currentIrq <= nextIrq;

            //if one ends and the one that is about to enter hasn't started yet
            if(((completed >> nextIrq) & 4'b0001 == 4'b0001) && int_sources_2 != 4'b0000) begin
                int_req = 1'b1;
            end
        end
        else if(currentIrq > nextIrq) begin         //if the next irq has higher priority
            currentIrq <= nextIrq;
            int_req <= 1'b1;                        //send new request
        end
        else if(int_ack_attended == 1'b1) begin     //if it was attended
            completed <= completed & ~(4'b0001 << currentIrq);      //clear completed bit
            int_req <= 1'b0;                                        //clear request
        end
    end
end

assign int_number = currentIrq[1:0];

endmodule
```

Algoritmo 3.12: Interrupt controller implementation

As can be seen in the simulation, the upward transition from source 2 begins and from that moment on, an interrupt request is sent from the interrupt controller to the control unit. When the interrupt starts to be answered, the control unit sends an ack to the interrupt controller. After this, the upward transition of source 2 takes place. Since this has higher priority, the service routine for interrupt 3 is interrupted and the service routine for interrupt 2 starts to be executed. When the RETI of ISR2 is executed, the control unit sends another ack to inform that it was complete, after which the ISR3 continues its execution until it reaches the RETI instruction. During this entire process, you can check how the stack works correctly.

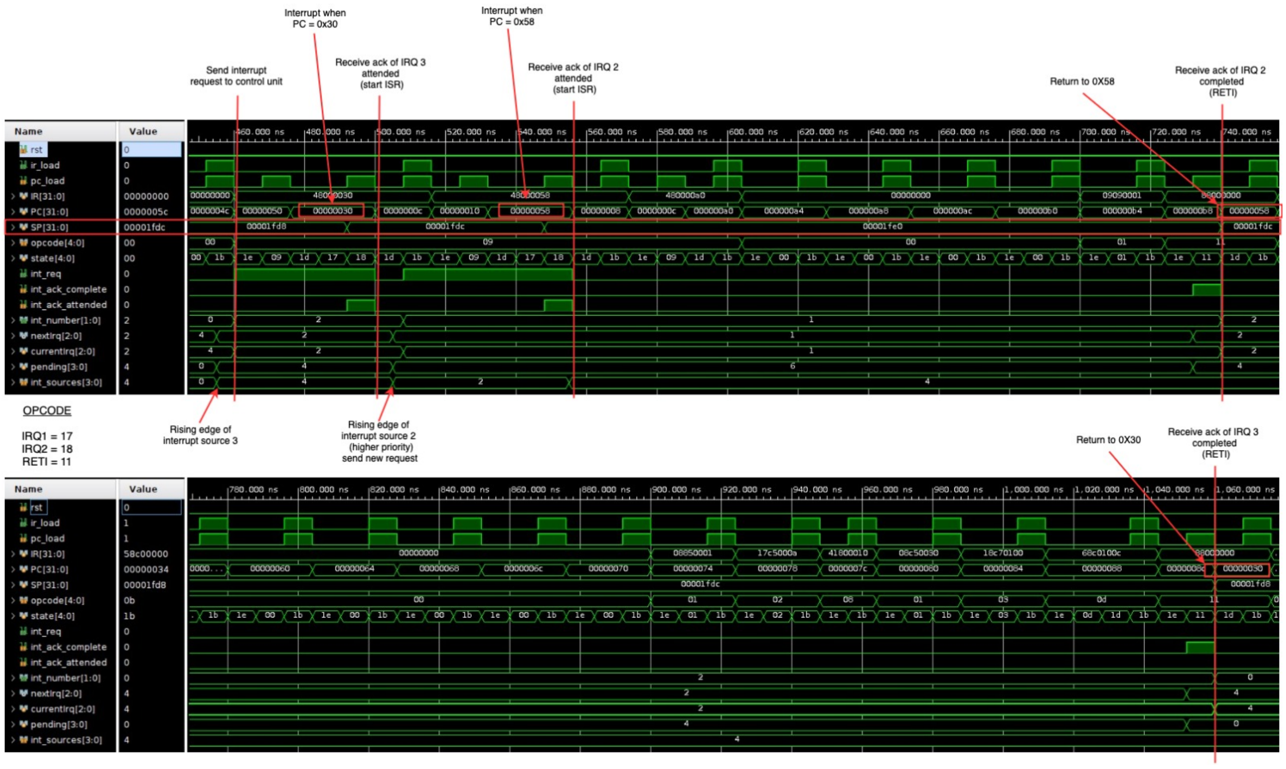

Interrupt when PC = 0x30
Interrupt when PC = 0x58
Send interrupt request to control unit
Receive ack of IRQ 3 attended (start ISR)
Receive ack of IRQ 2 attended (start ISR)
Return to 0X58
Receive ack of IRQ 2 completed (RETI)
OPCODE
IRQ1 = 17
IRQ2 = 18
RETI = 11
Rising edge of interrupt source 3
Rising edge of interrupt source 2 (higher priority) send new request
Return to 0X30
Receive ack of IRQ 3 completed (RETI)
Name
Value
rst
ir_load
pc_load
IR[31:0]
PC[31:0]
SP[31:0]
opcode[4:0]
state[4:0]
int_req
int_ack_complete
int_ack_attended
int_number[1:0]
nextirq[2:0]
currentirq[2:0]
pending[3:0]
int_sources[3:0]

Figure 3.13: Tables implemented in database

3.5 UART

As mentioned in the design phase, the UART peripheral can be divided into 3 modules:

- Baudrate generator;
- UART TX;
- UART RX;

3.5.1 Baudrate generator

To implement the baudrate generator, a cycle counter from the main clock was used (125 Mhz). This module receives as input the value of clock ticks to be counted to generate the correct baudrate (counter = 125M / baudrate).

However, in this module, this value is divided by 2 in order to reduce this counter by half and thus, when the counter overflows, the baudrate tick changes state, thus generating a signal with 50% duty-cycle, as represented the following code.

```
module uart_baudrate_generator(
input rst,
input clk,
input enable,
input [19:0] baudrate_counter,
output reg tick
);

wire clk2;
assign clk2 = clk & enable;          //if disabled, stop the internal clock

reg [19:0] internal_counter;

```

```

    always @(posedge clk) begin
        if(rst) begin
            tick <= 0;
            internal_counter <= 20'd1;
        end
        else if (internal_counter == 1) begin
            tick <= ~tick;                                  //toggle tick bit
            internal_counter <= {1'b0, baudrate_counter[19:1]}; //divide the counter by 2 and update internal counter
        end
        else begin
            internal_counter <= internal_counter - 20'd1;       //decrement counter
        end
    end
endmodule
```

Algoritmo 3.13: Baudrate generator implementation

3.5.2 UART TX

The UART TX module is implemented according to the state machine shown in the design phase.The implementation of this module is displayed in the listing presented below.

```
`define DATA_SIZE   8

module uart_tx(
    input tick,
    input tx_start,
    input [`DATA_SIZE-1:0] data_in,
    input rst,
    output reg tx_bit,
    output reg tx_done
    );

    `define S_IDLE1      2'b00
    `define S_START1     2'b01
    `define S_DATA1      2'b10
    `define S_STOP1      2'b11

    reg [1:0]state;
    reg [`DATA_SIZE-1:0] buffer;
    reg [2:0]counter;


    always@(posedge tick) begin
        if(rst) begin
            state = `S_IDLE1;
            tx_bit <= 1'b1;
        end
        else begin
            case(state)
              `S_IDLE1: begin            //wait for start command
                    if(tx_start == 1'b1)
                       state = `S_START1;
               end

              `S_START1: begin
                    tx_done <= 0;
                    tx_bit <= 0;
                    buffer <= data_in;      //loads the data to be sent to the buffer
                    counter <= 7;           //reset counter
                    state = `S_DATA1;
               end

               `S_DATA1: begin
                    tx_bit <= buffer[0];                //invit the less significant bit
                    counter <= counter - 1;             //decrement counter
                    buffer = {1'b0, buffer[7:1]};       //shift right buffer

                   if(counter == 0)
                        state = `S_STOP1;
               end

               `S_STOP1: begin
                   tx_done <= 1;
                   tx_bit <= 1;     //stop bit
                   state = `S_IDLE1;
               end

               default:
                    state = `S_IDLE1;
            endcase
        end
    end
endmodule
```

Algoritmo 3.14: UART TX implementation

In figure 3.14 presented below, it is possible to check the correct functioning of this module, where the value 0x4C was transmitted. In the buffer, it is also visible that the value is shifted to the right (divided by two). Furthermore, it is verified that the protocol is complied with, starting by setting the bit to 0 and ending with the bit to 1. At the end of the transmission it is also possible to verify that the "tx_done" signal is at a high level.

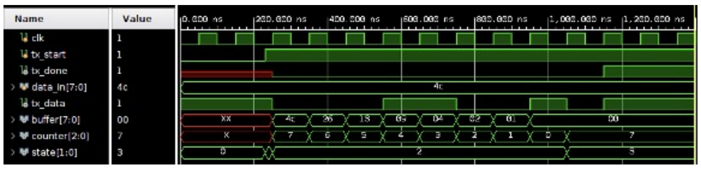


Figure 3.14: UART TX simulation

3.5.3 UART RX

Like the UART TX, the UART RX was implemented using a state machine, as demonstrated in the design phase. The listing presented below shows the code developed to implement this module.

```verilog
`define DATA_SIZE 8

module uart_rx(
    input tick,
    input rst,
    input rx_en,
    input rx_bit,
    output reg [`DATA_SIZE-1:0]data_out,
    output reg rx_done
    );

    `define S_IDLE      2'b00
    `define S_DATA      2'b01
    `define S_STOP      2'b10

    reg [3:0] counter;
    reg [`DATA_SIZE-1:0] buffer;
    reg [1:0] state;

    initial begin
        state <= `S_IDLE;
        rx_done <= 0;
    end

    always@(posedge tick) begin
        if(rst) begin
            state <= `S_IDLE;
            rx_done <= 0;
        end
        else if(rx_en == 1'b1) begin
            case(state)
               `S_IDLE: begin
                    rx_done <= 0;

                    if(rx_bit == 1'b0) begin    //wait for start bit
                        counter <= 0;           //reset counter
                        buffer <= 0;            //clear internal buffer
                        state <= `S_DATA;
                    end
               end
               `S_DATA: begin
                    buffer <= {rx_bit, buffer[7:1]};    //put received bit in the most significant buffer bit and shift right
                    counter =  counter + 1;             //increment counter

                    if(counter[3])
                        state <= `S_STOP;               //if the counter overflows
               end
               `S_STOP: begin
                    data_out <= buffer;                 //put received data in output
                    rx_done <= 1;
                    state <= `S_IDLE;
               end
               default:
                    state <= `S_IDLE;
            endcase
        end
    end
endmodule
```

Algoritmo 3.15: UART RX implementation

Through the simulation presented below in figure 3.15, it is possible to verify the correct functioning of this module. Initially, it waits for 'rx_bit' (din) to reach 0. From that moment on, with each upward transition of the baudrate tick, the value of 'rx_bit' is stored in the buffer, shifting it to the left. After receiving 8 bits and 'rx_bit' reaches 1, the value stored in the buffer is placed in the output.

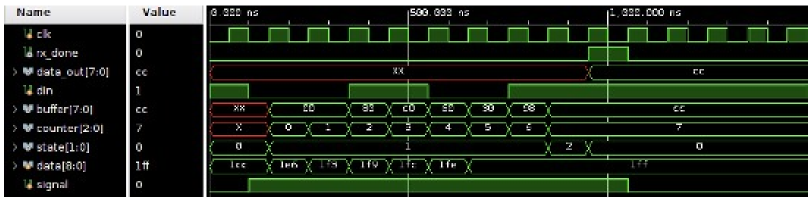


Figure 3.15: UART RX simulation

3.6 Top level

With all off the modules correctly implemented and verified the top level was formed through the instantiation and connection of all the other developed blocks. The implementation of said top level is displayed in the listing below.

```
module vespa_toplevel(input clk,
                      input rst,
                      input rx_bit,
                      output tx_bit,

                      input  [2:0]buttons,   //only for test in leds
                      output  [4:0]leds,     //only for test in leds
                      output  [7:0]result    //only for test in leds
                      );

    //wires to make connection are not represented here for space reasons


uart UART(.clk(clk),
          .rst(rst),
          .uart_en(uart_config[20]),
          .rx_bit(rx_bit),
          .rx_en(uart_config[21]),
          .tx_start(tx_data[8]),
          .tx_data(tx_data[7:0]), //8 bits
          .baudrate(uart_config[19:0]),
          .tx_done(tx_done),
          .tx_bit(tx_bit),
          .rx_done(rx_done),
          .rx_data(rx_data)
        );


controlUnit CONTROLUNIT(.clk(clk),
                        .rst(rst),
                        .alu_en(alu_en),
                        .alu_ctrl(alu_ctrl),
                        .C(C),
                        .Z(Z),
                        .N(N),
                        .V(V),
                        .opcode(opcode),
                        .branch_cond(branch_cond),
                        .imm_op(imm_op),
                        .blockRams_init(blockRams_init),
                        .ir_load(IR_load),
                        .pc_load(PC_load),
                        .sp_load(sp_load),
                        .we_mem(we_mem),
                        .we_rf(we_rf),
                        .pc_sel(pc_sel),
                        .mem_addr_sel(mem_addr_sel),
                        .rf_read2_addr_sel(rf_read2_addr_sel),
                        .rf_write_sel(rf_write_sel),
                        .mem_in_sel(mem_in_sel),
                        .sp_in_sel(sp_in_sel),
```

```verilog
                        .alu_op2_sel(alu_op2_sel),
                        .int_req(int_req),
                        .irq_addr(irq_addr),
                        .int_ack_complete(int_ack_complete),
                        .int_ack_attended(int_ack_attended),
                        .map_req(0),

                        .leds(leds) //only for test in leds
                       );


datapath DATAPATH(.clk(clk),
                  .reset(rst),
                  .we_rf(we_rf),
                  .we_mem(we_mem),
                  .pc_load(PC_load),
                  .ir_load(IR_load),
                  .sp_load(sp_load),
                  .irq_addr(irq_addr),
                  .alu_ctrl(alu_ctrl),
                  .alu_en(alu_en),
                  .C(C),
                  .Z(Z),
                  .N(N),
                  .V(V),
                  .opcode(opcode),
                  .imm_op(imm_op),
                  .branch_cond(branch_cond),
                  .rf_write_sel(rf_write_sel),
                  .pc_sel(pc_sel),
                  .mem_addr_sel(mem_addr_sel),
                  .rf_read2_addr_sel(rf_read2_addr_sel),
                  .alu_op2_sel(alu_op2_sel),
                  .mem_in_sel(mem_in_sel),
                  .sp_in_sel(sp_in_sel),
                  .blockRams_init(blockRams_init),
                  .error(error),
                  .uartRX_data({24'b0,rx_data}),
                  .uartTX_data(tx_data),
                  .uart_config(uart_config),
                  .irq_config(irq_config),
                  .int_number(int_number),

                  .result2(result2)         //only for test in leds
                 );

assign irq_src1 = buttons[0];
assign irq_src2 = buttons[1];
assign irq_src3 = buttons[2];

interrupt_controller IRQ(.clk(clk),
                         .rst(rst),
                         .int_sources({irq_src3, irq_src2, irq_src1, error}),
                         .int_ack_complete(int_ack_complete),
                         .int_ack_attended(int_ack_attended),
                         .int_req(int_req),
                         .int_number(int_number),
                         .ea(uart_config[0]),
                         .en1(uart_config[1]),
                         .en2(uart_config[2]),
                         .en3(uart_config[3])
                        );


assign result = {int_req, int_number, result2};     //only for test in leds
endmodule
```

Algoritmo 3.16: Top level implementation

The following image shows a diagram of blocks generated by Vivado's Linter and comparing this with what was presented in the design phase it can be seen that the blocks and connections correspond.

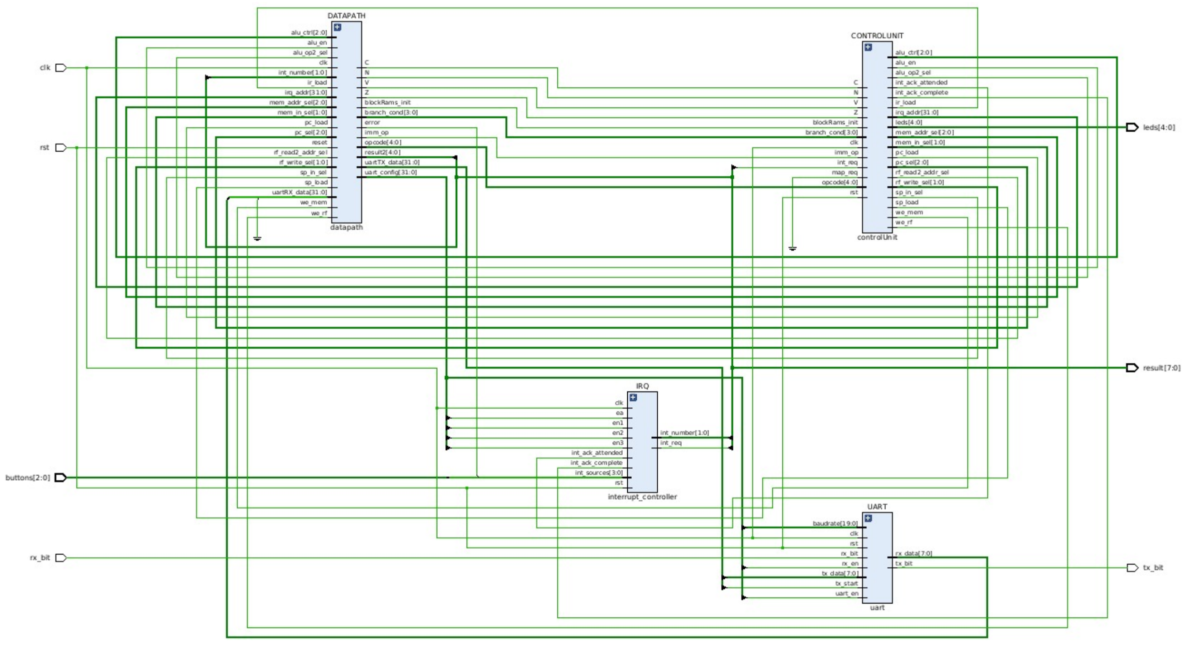


Figure 3.16: Top level synthesis

4 | Conclusion

The development of the VeSPA processor, the interrupt controller and the assigned peripheral were an opportunity to the group to solidify the concepts acquired over the course of the semester and to learn in a way that wouldn't´t be possible without "getting our hands dirty". This allowed us to gain experience in a way that wouldn't be possible without this project. This, without a doubt, made this curricular unit taught by professor Adriano Tavares much more rich and it's completion leaves us better prepared for the next semester and, of course, for all future work. In the future, our intellectual curiosity will continue to be fuelled and this project will be continued over the course of the second semester, where, among other features, a pipeline will be implemented which will provide the processor a great increase in performance and leaves the group excited for the future and new challenges.

5 | References

https://support.xilinx.com/s/knowledge-base?language=en_UStabset-0c7e1=2
https://support.xilinx.com/s/topiccatalog?language=en_US
https://www.xilinx.com/support.htmldocumentation