Universidade do Minho
Escola de Engenharia

Master degree in Industrial Electronic Engineering and Computers
Specialization in Embedded Systems and Computers

VeSPA SOC - A Pipeline Approach

Class Project

Professor:
Adriano Tavares

June 2024

Contents

List of Figures

Chapter 1

Introduction

This report explores the development of a System on Chip (SoC) for VeSPA, highlighting the project's evolution from the initial development of the VeSPA microcontroller to the current version, which incorporates a pipelined architecture. Previously, a basic VeSPA microcontroller was developed, serving as a proof of concept for its functionality. In this new phase of the project, the aim is to enhance the efficiency and performance of the microcontroller by introducing a pipeline, a fundamental technique in modern processor design that allows multiple instructions to be processed simultaneously in different stages.

Building on the previous implementation, this project takes advantage of several peripherals that were already developed for the initial VeSPA microcontroller. Additionally, a VGA driver was developed to enhance the system's capabilities and demonstrate its versatility in managing various input and output devices.

The document details each step of the implementation, from the initial design to the final testing, emphasizing the challenges faced and the solutions applied. Utilizing a Field-Programmable Gate Array (FPGA), specifically the Zybo Z7 board, provides the flexibility and adaptability required to test and validate the new pipelined design. This report also examines the benefits of this approach in terms of performance and efficiency, and how the practical application of theoretical concepts in embedded systems can lead to significant technological advancements.

The implementation of the pipeline in the VeSPA microcontroller aims not only to improve the system's throughput but also to demonstrate the practical applicability of advanced processor architecture techniques. This report offers a comprehensive overview of the development of the VeSPA SoC, providing a solid foundation for future developments and optimizations in the field of embedded systems.

Chapter 2

Methodology

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

The stages of the waterfall methodology are:

1. **Analysis**: The initial stage involves learning and documenting the objectives of the system, as well as its constraints and requirements. Furthermore, the system's specifications are examined to create product models and business logic that will steer the production process. This stage also encompasses the evaluation of financial and technical resource viability.

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

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

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

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

The waterfall model is particularly suitable for projects with well-defined documentation, static requirements, ample resources, set timelines, and established technology. Its benefits include thorough initial documentation and planning, which ensure that teams, whether large or fluctuating, stay informed and united towards a common goal. The model's disciplined, simple-to-understand structure aids in task organization and supports managerial control based on schedules and deadlines.

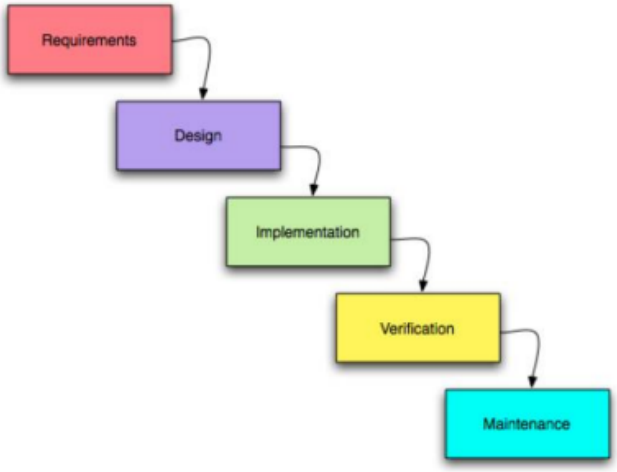

Figure 2.1: Waterfall methodology

In this project, the class explored all the phases of the waterfall model (figure 2.1), with the exception of the maintenance phase, given the project's limited scope.

Chapter 3

Analysis

3.1 Problem Statement

The project aims to demonstrate the performance improvements that can be achieved by moving to a pipelined microarchitecture and to provide insights into the design considerations and trade-offs involved in constructing such a microarchitecture. Therefore, the objective is to develop a pipelined microarchitecture in Verilog for the VeSPA microcontroller. The project will involve analyzing the original architecture of the VeSPA microcontroller, identifying its dependencies and hazard risks within the instruction set, and designing a pipelined microarchitecture to improve its performance. Processor design features must be considered to achieve the best possible performance, while various hazards must be resolved to ensure correct operation.

3.2 Requirements

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

- Ensure a functionally similar architecture to the base VeSPA architecture.
- Ensure the capability for expansion and scalability to accommodate future enhancements.
- Optimize the micro-architecture for the least possible pipeline stalls and maximize performance and efficiency.
- Develop a VGA and PS/2 peripheral.
- Add some peripherals implemented in the previous semester.

3.3 Constraints

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

- Restricted team composition.

- Operate within a limited time frame (the project is due by the semester's conclusion).
- Use Xilinx Vivado.
- Use Verilog.
- Use the ZYBO Z7 board.

3.4 CPU

3.4.1 Instruction Set Architecture

The VeSPA ISA (Instruction Set Architecture) encompasses a minimalist set of instructions. For the arithmetic and logic instructions, it is important to note that these instructions can be executed with either another register (bit 16 set to '0') or an immediate value (bit 16 set to '1').

The ISA is categorized as follows:

Arithmetic Instructions

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

ADD

31-27	26-22	21-17	16	15-11	10-0
00001	rdst	rs1	0	rs2	00000000000

31-27	26-22	21-17	16	15-0
00001	rdst	rs1	1	immed16

Figure 3.1: ADD format instruction

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

SUB

31-27	26-22	21-17	16	15-11	10-0
00010	rdst	rs1	0	rs2	00000000000

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

Figure 3.2: SUB format instruction

Logical Instructions

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

OR

31-27	26-22	21-17	16	15-11	10-0
00011	rdst	rs1	0	rs2	00000000000

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

Figure 3.3: OR format instruction

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

AND

31-27	26-22	21-17	16	15-11	10-0
00100	rdst	rs1	0	rs2	00000000000

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

Figure 3.4: AND format instruction

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

NOT

31-27	26-22	21-17	16-0
00101	rdst	rs1	00000000000000

Figure 3.5: NOT format instruction

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

XOR

31-27	26-22	21-17	16	15-11	10-0
00110	rdst	rs1	0	rs2	00000000000

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

Figure 3.6: XOR format instruction

Control Instructions

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

CMP

31-27	26-22	21-17	16	15-11	10-0
00111	00000	rs1	0	rs2	00000000000

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

Figure 3.7: CMP format instruction

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

BXX

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

Figure 3.8: BXX format instruction

Cond	Assembly	Condition
0000	BRA	Branch always
0001	BCC	Branch on carry clear ($\overline{C}$)
0010	BVC	Branch on overflow set ($\overline{V}$)
0011	BEQ	Branch on equal (Z)
0100	BGE	Branch on greater than or equal to ($\overline{N}$ OR V)
0101	BGT	Branch on greater than (Z AND ($\overline{N}$ OR V))
0110	BPL	Branch on plus (positive) ($\overline{N}$)
1000	BNV	Branch never
1001	BCS	Branch on carry set (C)
1010	BVS	Branch on overflow set (V)
1011	BNE	Branch on not equal ($\overline{Z}$)
1100	BLT	Branch on less than (N AND $\overline{V}$)
1101	BLE	Branch on less than or equal to (Z OR (N AND $\overline{V}$))
1110	BMI	Branch on minus (negative) (N)

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

JMP

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

Figure 3.9: JMP format instruction

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

JMPL

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

Figure 3.10: JMPL format instruction

Data Transfer Instructions

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

LD

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

Figure 3.11: LD format instruction

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

LDI

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

Figure 3.12: LDI format instruction

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

LDX

31-27	26-22	21-17	16-0
01100	rdst	rs1	immed17

Figure 3.13: LDX format instruction

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

ST

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

Figure 3.14: ST format instruction

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

STX

31-27	26-22	21-17	16-0
01110	rdst	rs1	immed17

Figure 3.15: STX format instruction

Miscellaneous Instructions

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

NOP

31-27	26-0
00000	000 0000 0000 0000 0000 0000

Figure 3.16: NOP format instruction

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

HALT

31-27	26-0
11111	000 0000 0000 0000 0000 0000

Figure 3.17: HALT format instruction

Additional Instructions

- **RETI:** used to handle interruptions, which the original VeSPA does not account for.

RETI

31-27	26-0
10001	000 0000 0000 0000 0000 0000

Figure 3.18: RETI format instruction

- **SHIFT:** created for supporting the shift operations, freeing the compiler from this task.

RIGHT SHIFT

31-27	26-22	21-17	16	15-11	10-0
01111	rdst	rs1	0	rs2	00000000000

31-27	26-22	21-17	16	15-0
00001	rdst	rs1	1	immed16

LEFT SHIFT

31-27	26-22	21-17	16	15-11	10-0
10000	rdst	rs1	0	rs2	00000000000

31-27	26-22	21-17	16	15-0
00001	rdst	rs1	1	immed16

Figure 3.19: RIGHT and LEFT SHIFT format instruction

3.4.2 Pipeline Approach

The non-pipelined microarchitecture depicted in figure 3.20 showcases a sequential process flow, with each symbol representing a stage in the execution of a task. The timeline indicates that only one stage is active at any given time, with each stage needing to be completed before the next begins. The gaps between active stages imply idle time for certain components, highlighting the inefficiencies of this approach components are underutilized, awaiting the completion of previous tasks. This sequential execution model results in longer overall process times and demonstrates how a non-pipelined system handles tasks one at a time without overlap.

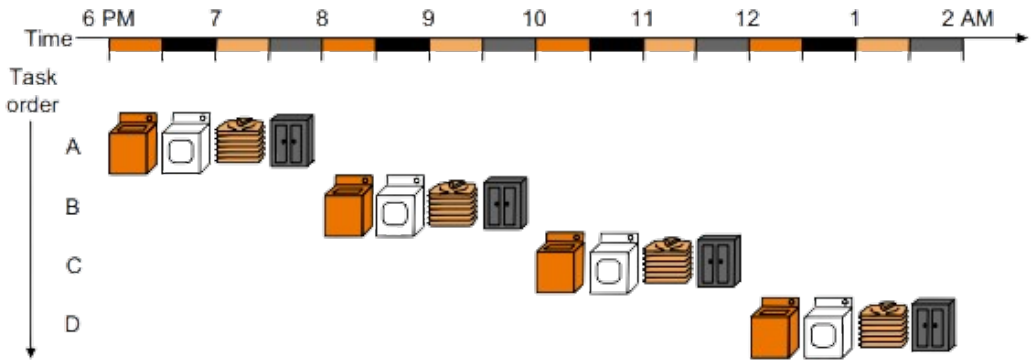

Figure 3.20: Non-Pipeline Approach

Figure 3.21 depicts a pipelined approach to task management, where tasks A through D are processed in a staggered fashion across a timeline from 6 PM to 10PM. The key characteristic of this pipelined process is that different stages of multiple tasks are executed in parallel, unlike a non-pipelined system where tasks are completed one after the other without overlap. The pipeline allows the initiation of a new task (for example, the drying machine) while the first (in this case, the washing machine) is starting again, effectively utilizing the time gaps that would be present in a non-pipelined approach. This results in multiple tasks being in different stages of completion simultaneously. It's apparent from the layout that, translating to architecture design, while one task is in the execution phase, another can be in the decode phase, and yet another in the fetch stage.

The overlapping of these stages leads to a more efficient use of time and system resources, reducing overall latency and increasing throughput. The image exemplifies how a pipelined approach streamlines processes, maximizes efficiency,

and minimizes idle times, demonstrating the concurrent progression of tasks through the system.

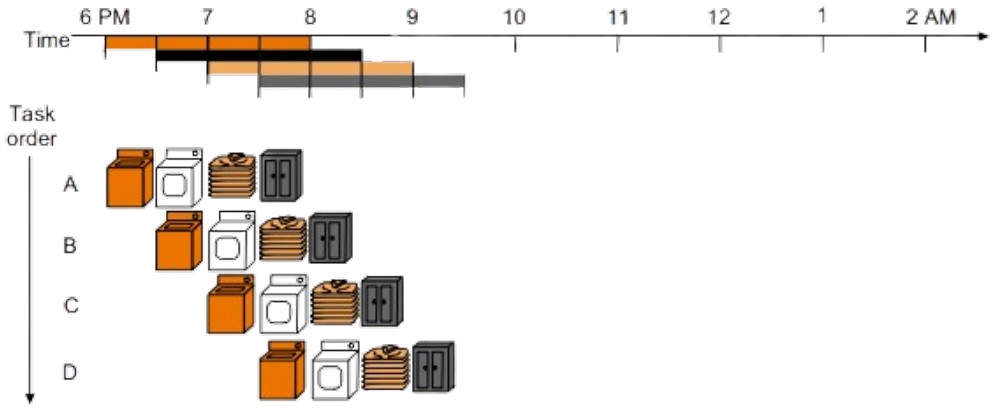

Figure 3.21: Pipeline Approach

3.4.3 Pipeline Hazards

One of the biggest obstacles in the pipeline process are hazards, where the continuous process of executing instructions is affected by dependencies between them. The hazards that exist can be categorised into three different types: data hazards, structural hazards and control or branch hazards.

Structural Hazard

Structural hazards occur when two or more instructions that are already in the pipeline need to access the same resource in the same execution cycle, thus generating a conflict between the instructions. A possible case in which this scenario occurs would be two instructions that need to access the memory in the same cycle.

A practical example would be a write-back and a fetch executed at the same time on the same pipe in a Von Neumann architecture in which code memory would be accessed by the fetch and data memory accessed by the write-back, thus generating a structural hazard.

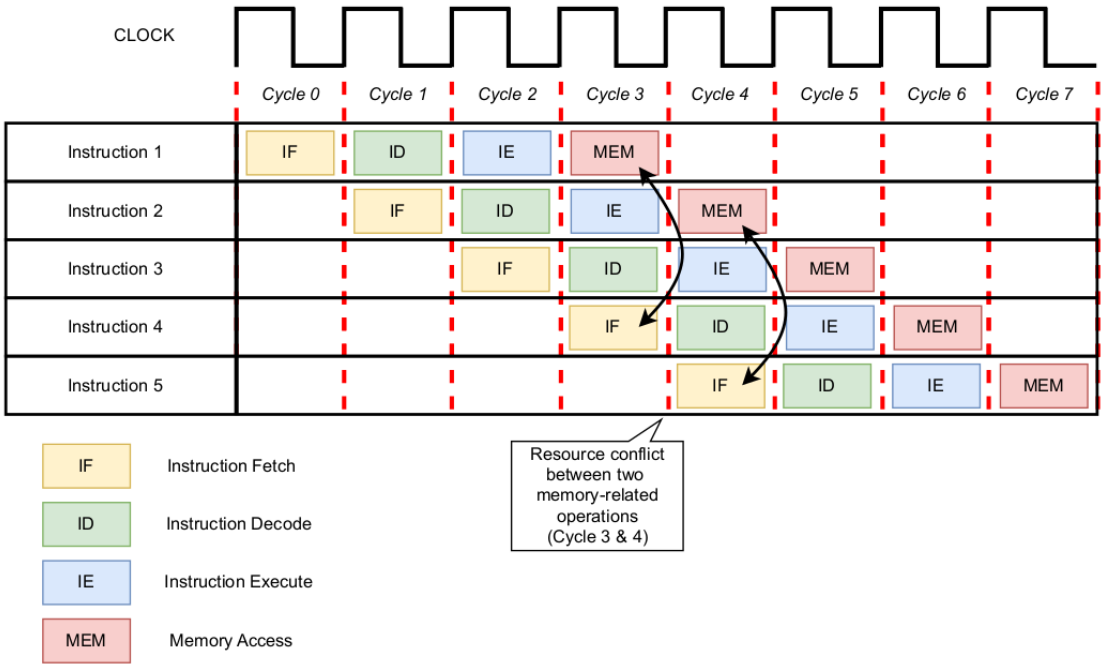

Figure 3.22: Structural Hazard representation

Data Hazard

Data hazards occur when an instruction relies on the outcome of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. Assume instruction i occurs in program order before instruction j and both instructions use register x, then there are three different types of hazards that can occur between i and j:

- **Read after write (RAW):** Occurs when j tries to access a source before it has been written to it, therefore j mistakenly retrieves the old value. This hazard, known as a true data dependence, is the most frequent type.
- **Write after write (WAW):** Occurs when j attempts to update an operand before it's modified by i, the writing occurs in an incorrect sequence. Consequently, the value authored by i supersedes the one authored by j. This situation characterizes an output dependence hazard. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.
- **Write after read (WAR):** This situation arises when j attempts to update a destination before it's accessed by i, causing i to mistakenly retrieve the

new value. This hazard arises from an anti-dependence (or name dependence). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline and other instructions that read a source late in the pipeline, or when instructions are reordered.

The WAW and WAR hazards are also dependent on name, and not on value as in the case of the RAW dependence. Meaning that in the RAW dependence, the correct execution of the read instruction depends on the correct value transmission from the write instruction. In the WAW and WAR there is no need for value communication, therefore those do not contitute a true dependency.

As said above, the WAW and WAR hazards only occur in architectures that either have instructions defined in ISA that write to the register file in different stages of the pipeline, or the pipeline itself has two or more different stages that write to the register file. So if the architecture only allows to write to destination in the last stage and the writing occurs in the order of the program, then the WAR and WAW dependencies are already being handled.

Control or Branch Hazards

Control hazards, also known as branch hazards, occur in the presence of conditional branches or jumps in the instruction stream of a processor. These hazards occur when the outcome of a branch instruction is determined late in the pipeline, after several subsequent instructions have already been fetched and partially executed.

The pipeline works by fetching, decoding, executing, and committing instructions in sequential order. However, when a branch instruction is encountered, the processor must decide whether to take the branch or continue executing sequentially. Until the branch condition is evaluated and the target address is determined, subsequent instructions are fetched and partially executed based on the assumption that the branch will not be taken.

Hazard Mitigation

Pipeline hazards can significantly impact pipeline efficiency. To address this problem, several mitigation methods can be employed.

(1) Forwarding (Bypassing):

Forwarding, also known as bypassing, involves directly transferring data from the output of one pipeline stage to the input of another to resolve data hazards.

By forwarding data instead of waiting for it to be written to a register, subsequent instructions can proceed without stalling, improving pipeline efficiency.

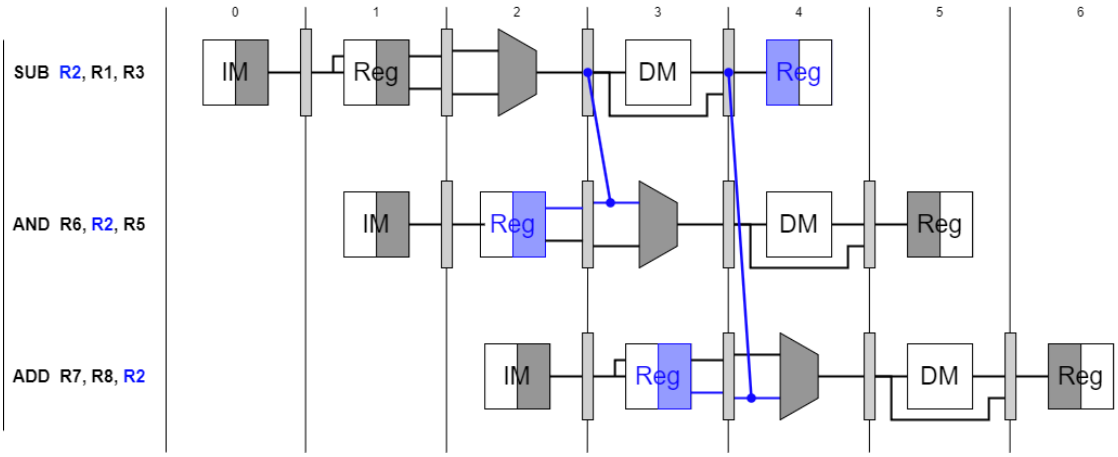

Figure 3.23: Pipeline Bypass representation

(2) Stall:

Stall is a method to mitigate hazards by pausing the pipeline's progress until the hazard is resolved. During a stall, the instruction encountering the hazard is paused in its current stage of the pipeline, while the subsequent stages remain idle until the hazard is resolved.

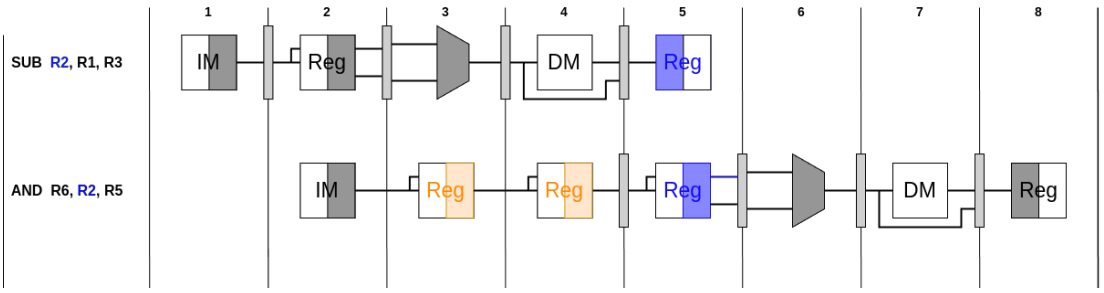

Figure 3.24: Pipeline Stall representation

(3) Flush:

Pipeline flush is a method used to discard all instructions in the pipeline and reset it to an empty state. This is typically employed in response to control hazards, such as branch mispredictions, to ensure correct program execution. When a control hazard occurs, the pipeline is flushed to prevent incorrect instructions from being executed.

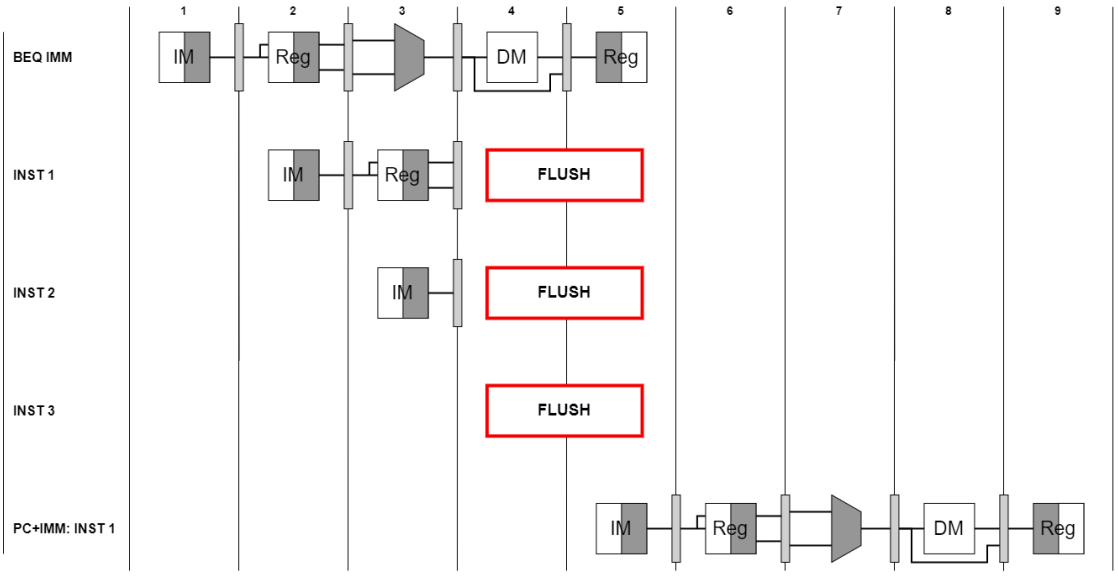

Figure 3.25: Pipeline Flush representation

(4) Branch Prediction:

Branch prediction techniques aim to predict the outcome of branch instructions before they are executed, reducing the impact of control hazards. Predictions are based on historical data or patterns in program behavior. If a prediction is correct, the pipeline continues executing instructions from the predicted branch path, minimizing stalls.

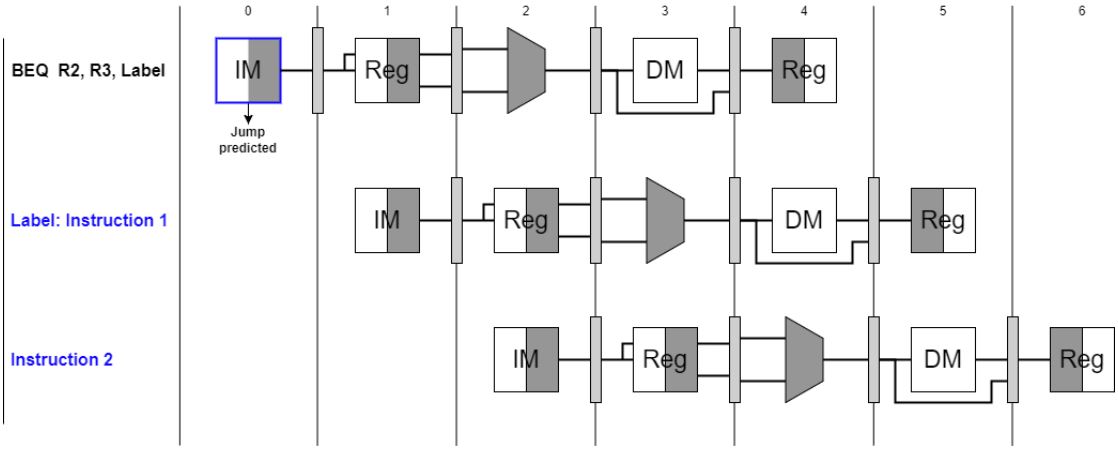

Figure 3.26: Pipeline Branch Prediction representation (branch always taken)

(5) Scoreboarding:

Scoreboarding, consists in having a validation bit in the register file for each register. Each time an instruction, that will later (on write-back) access the register file, reaches the decode stage, the valid bit is reset. And, when that instruction reaches write-back, the register becomes valid again. If any instruction accesses an invalid register, then the pipeline will stall, as shown in Figure 3.24, until the register becomes valid.

The advantage of this solution is that it is simple to implement because the logic of the hazard detection is very simple. The main drawback is that this solution would stall the pipeline not only for the RAW hazards but also for the WAR and WAW ones, which is not ideal since those don't need to be handled. It also implies the modification of the current register file implemented on VeSPA.

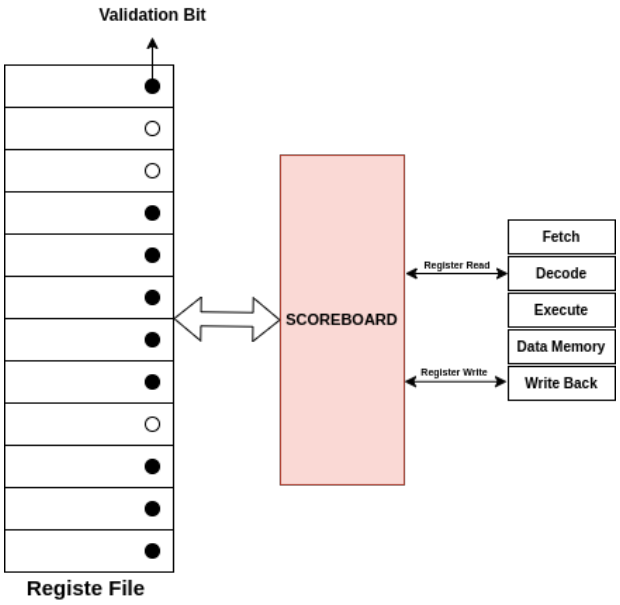

Figure 3.27: Scoreboarding method

3.5 Bus

A bus is a communication system within a computer system that connects the CPU (Central Processing Unit) to various peripherals. It allows the CPU to send and receive data from devices such as memory and input/output devices. This component ensures efficient data transfer and communication between the CPU and other components.

One of the most widely used buses is AXI (Advanced eXtensible Interface) because it provides high-performance, high-frequency data communication. AXI is part of the ARM AMBA (Advanced Microcontroller Bus Architecture) family and is known for its ability to efficiently handle multiple data streams simultaneously, support high-speed data transfers, and offer flexibility in connecting various components within the system.

The handshaking mechanism of the AXI protocol is crucial because it guarantees dependable and effective data transfer between the slave and master devices. The Figure 3.28 bellow, is a temporal diagram to a AXI write process, four clock cycles are necessary to conclude the communication.

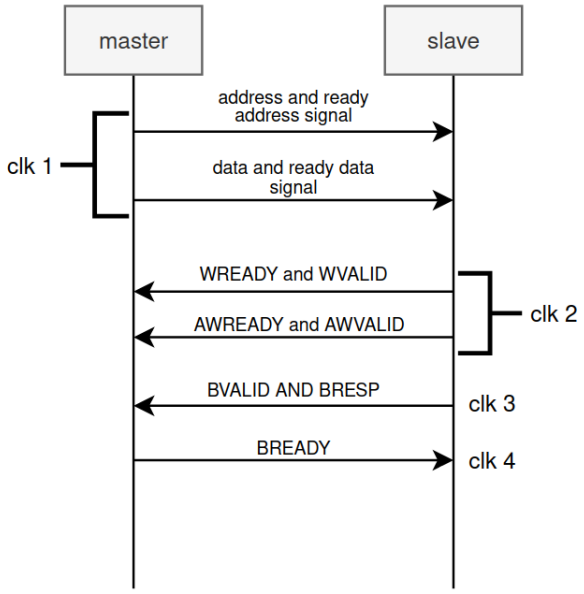

Figure 3.28: AXI Sequential Diagram

Fast connections with various peripherals are fundamental for this microprocessor because they ensure efficient data transfer and quick response times, which are crucial for high-performance computing tasks. Although the AXI protocol offers many positive features, such as guaranteeing that information is delivered correctly, its four-phase transaction process (address phase, data phase, response phase, and handshaking) and the multiple data streams simultaneously introduce latency. This latency might be considered excessive for applications that require ultra-fast communication.

Therefore, a custom bus protocol will be developed specifically for this case. This custom protocol eliminates the confirmation phase that verifies successful data delivery, significantly speeding up communication. While this approach sacrifices

the built-in error-checking and confirmation features of AXI, it provides much faster data transfer, which is essential for the performance needs of this processor.

In the Figure 3.29, it is shown that this custom bus does not include a confirmation of the successful information arrival, but it is much quicker lasting only one clock cycle.

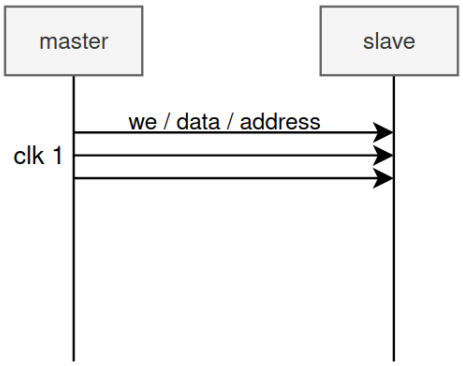

Figure 3.29: Custom Bus Sequential Diagram

3.6 Peripherals

3.6.1 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. They are then interpreted by the component in question which then sends signals to the CPU that determines the flow of execution. This way, the interrupt controller ensures that the CPU attends said interrupts in a 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.

In summary, the interrupt controller must be able to:

- Manage multiple interrupt sources and their respective signals.
- Prioritize interrupts to ensure the highest-priority interrupt is managed first.
- Enable or disable interrupts globally or individually (interrupt masking).
- Send signals to the CPU to fetch the correct interrupt vector for the corresponding ISR.
- Notify the CPU if there are more pending interrupt requests.

3.6.2 GPIO

GPIO (General Purpose Input/Output) is a set of pins on a microcontroller that can be programmed to function as digital inputs or outputs, allowing interaction with external components such as sensors, LEDs, buttons, and more. These pins provide a flexible interface for connecting various hardware devices, enabling the microcontroller to receive input signals and send output signals to control and communicate with other electronic components in a system.

In more advanced applications, GPIOs can be used for communication protocols like I2C, SPI, or UART, providing the microcontroller with the ability to communicate with other microcontrollers or peripheral devices. This versatility makes GPIOs a fundamental aspect of embedded system design, offering a simple yet powerful means of expanding the microcontroller's interaction with the external environment.

3.6.3 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, for this project, only the data-output from the PS/2 keyboard will be considered.

Data Transmission Protocol:

The transmission protocol ensures that data integrity is maintained during the communication process. Understanding the data transmission process is crucial for developing interfaces that communicate effectively with the PS/2 keyboard.

This process consists of two phases:

- The PS/2 keyboard transmits a bit on the Data line when the clock line is high.
- The host device reads the transmitted bit when the clock line goes low.

Frame Structure:

The frame structure of the PS/2 protocol includes specific bits that help in identifying and validating the data being sent from the keyboard. These bits ensure that the communication is synchronized and error-free.

Proper implementation of the frame structure is necessary for accurate data interpretation by the host device.

- 1 Start bit, always low.
- 8 Data bits, unique key code (except special keys).
- 1 Parity bit for error detection (odd parity).
- 1 Stop bit, always high.

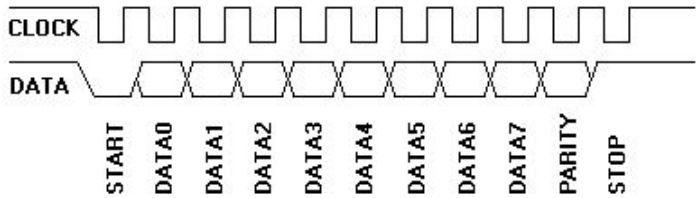

Figure 3.30: PS/2 Waveform

The PS/2 keyboard protocol uses scancodes to represent each key press and release. Each key on a PS/2 keyboard is associated with a unique code, which is sent to the computer when the key is pressed or released. For example, the 'A' key generates the scancode 0x1C when pressed and 0xF0 when released. These scancodes are used by the keyboard controller to interpret which key was pressed and to send this information to the computer's operating system.

This way the protocol ensures reliable communication by embedding additional bits for start, parity, and stop in the data packets transmitted between the keyboard and the computer as explain before.

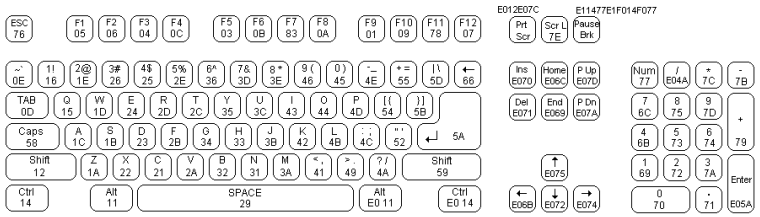

Figure 3.31: PS/2 Keyboard

3.6.4 Timer

A timer is a specialized hardware component within a computer system that precisely generates timing signals. These signals are essential for accurately measuring the duration of events and facilitating the execution of tasks at specific intervals. This analysis explores the key components and operational considerations for implementing an effective timer.

- **Clock Source:** The clock source is fundamental to the timer's operation, providing a consistent reference timing signal. The accuracy and stability of the clock source directly impact the precision of the timer. Typical choices for the clock source include oscillators and crystal clocks, selected based on the desired frequency and accuracy.
- **Enable Signal:** The enable signal is crucial for initiating the timer's operation. When activated, it allows the timer to start counting clock cycles. This signal can be controlled by software or hardware triggers, providing flexibility in how and when the timer is activated.
- **Counter:** The counter is the core component responsible for accumulating clock cycles. With each clock cycle, the counter increments by one. The width of the counter (e.g., 8-bit, 16-bit, 32-bit) determines the maximum count value before an overflow occurs. A wider counter can count more cycles before overflowing, allowing for longer timing intervals.
- **Overflow Signal:** The overflow signal is generated when the counter reaches its predefined maximum count value. This signal can be used to trigger interrupts, indicating that the specified timing interval has elapsed. Handling the overflow signal appropriately is critical for tasks that require periodic execution.

3.6.5 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.

Protocol Overview

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 Figure 3.32.

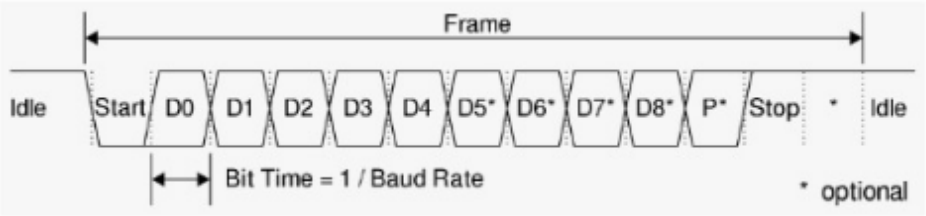

Figure 3.32: UART Protocol

3.6.6 VGA

VGA stands for Video Graphics Array. Initially, it refers specifically to the display hardware first introduced with IBM® PS2 computer in 1987. With the widespread adoption, it now usually refers to the analog computer display standards (defined by VESA®), the DE-15 Connector (commonly known as VGA connector), or the 640×480 resolution itself.

A DE-15 connector, commonly known as a VGA connector, is a three row 15-pin D-sub miniature Connector (named after their D-shaped metal shield). Only 5 signals out of 15 pins will be handled in this project. These signals are Red, Green, Blue, HS, and VS. Red, Green, and Blue are three analog signals that specify the color of a point on the screen, while HS and VS provide a positional reference of where the point should be displayed on the screen.

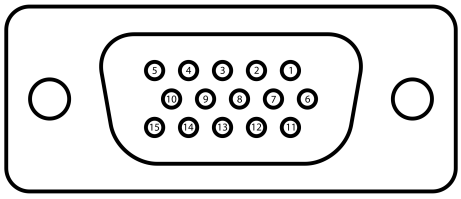

Figure 3.33: VGA Connector

By properly driving these five signals according to the VGA timing specification, we can display everything we want on any monitors. To understand how these signals should be driven, we need to take a look at how our monitors actually work.

CRT Monitor Functionality

Cathode Ray Tube (CRT) displays use electron beams to light up a phosphor-coated screen. These beams, controlled by electron guns, create images by moving across the screen in a pattern. The screen lights up where the beams hit, and colors are created by using three beams for red, blue, and green.

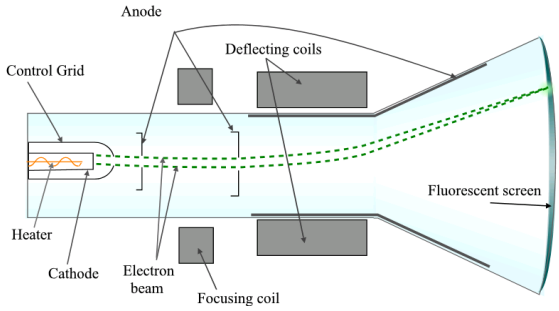

Figure 3.34: CRT Displays diagram

The beams are directed by electromagnetic fields, allowing them to move horizontally and vertically. The display only shows images when the beams move forward; the time taken for the beams to reset is not used for displaying information.

The resolution of the display depends on the beam size, how fast the beams move, and how quickly they can be modulated.

Modern Displays

Modern VGA displays offer various resolutions controlled by a VGA controller circuit. This circuit generates timing signals to control the display patterns. Raster video displays are defined by rows and columns, determining the pixel size. Video data, stored in video refresh memory, is applied to the display as the electron beams move across each pixel.

VGA Protocol and signals

The VGA protocol, managed by the VGA controller, coordinates timing signals like HS (horizontal sync) and VS (vertical sync) with video data delivery based on the pixel clock. VS sets the display refresh frequency (typically 50Hz to 120Hz), while HS ensures correct line transitions. Together, they define the horizontal retrace frequency. Additionally, the protocol handles border generation, delineating the area outside the active display, and manages RGB values within the active area to determine the color composition of each pixel, generation the whole frame.

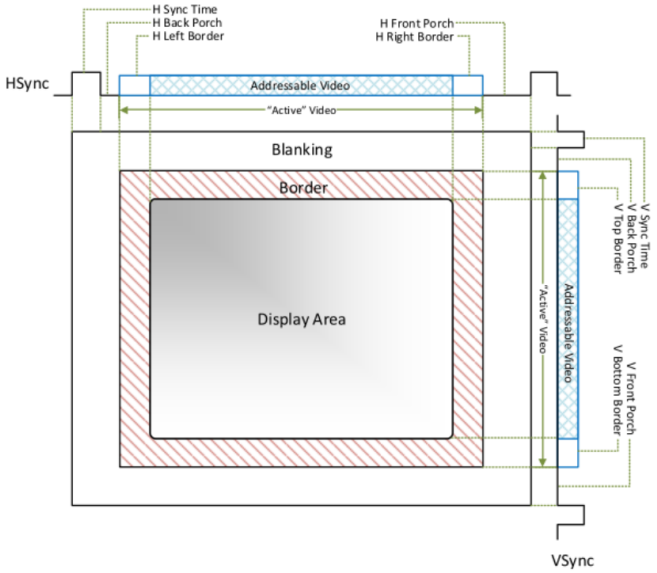

Figure 3.35: Display Area

Chapter 4

Design

4.1 CPU

4.1.1 Pipelined Datapath and Control

The division of an instruction into five stages means a five-stage pipeline, which in turn, means that up to five instructions will be in execution during each clock cycle. Thus, the datapath must be separated into five sections, with each section named corresponding to a stage of instruction execution:

- **IF**: Instruction Fetch
- **ID**: Instruction Decode
- **EX**: Execution or Address Calculation
- **MEM**: Data Memory Access
- **WB**: Write Back

In a pipeline with 5 stages, 4 intermediate registers are required to facilitate the flow of values from the first stage to the last one:

- Fetch to Decode Register File
- Decode to Execute Register File
- Execute to Memory Register File
- Memory to Write-Back Register File

To incorporate intermediate registers effectively, it is required to first define their interface, detailing the inputs and outputs for each register and stage. The following diagram on Figure 4.11 portrays an overview of the datapath implementation with the incorporation of the new buffers:

Fetch stage

The fetch stage is the initial phase in the pipeline process of the CPU, where the processor retrieves an instruction from memory. This crucial stage involves accessing the PC, which holds the address of the next instruction to be executed.

During this stage, the CPU uses the address from the PC to access the instruction memory and retrieve the respective instruction. Once retrieved, the instruction is loaded into the IR of the CPU, which will hold the instruction to further being decoded and executed in subsequent stages of the pipeline.

While in this stage, the PC is updated with the address of the next instruction incrementing by four in this 32-bit instruction set architecture. The fetched instruction in the IR is passed on to the next stage of the pipeline, the decode stage, where it will be analyzed and prepared for execution.

The efficiency of this stage is crucial for overall CPU performance, as any delays can impact the entire pipeline. Note that the IR is always delayed one cycle when compared to the PC pointing position due to IP BRAM delays.

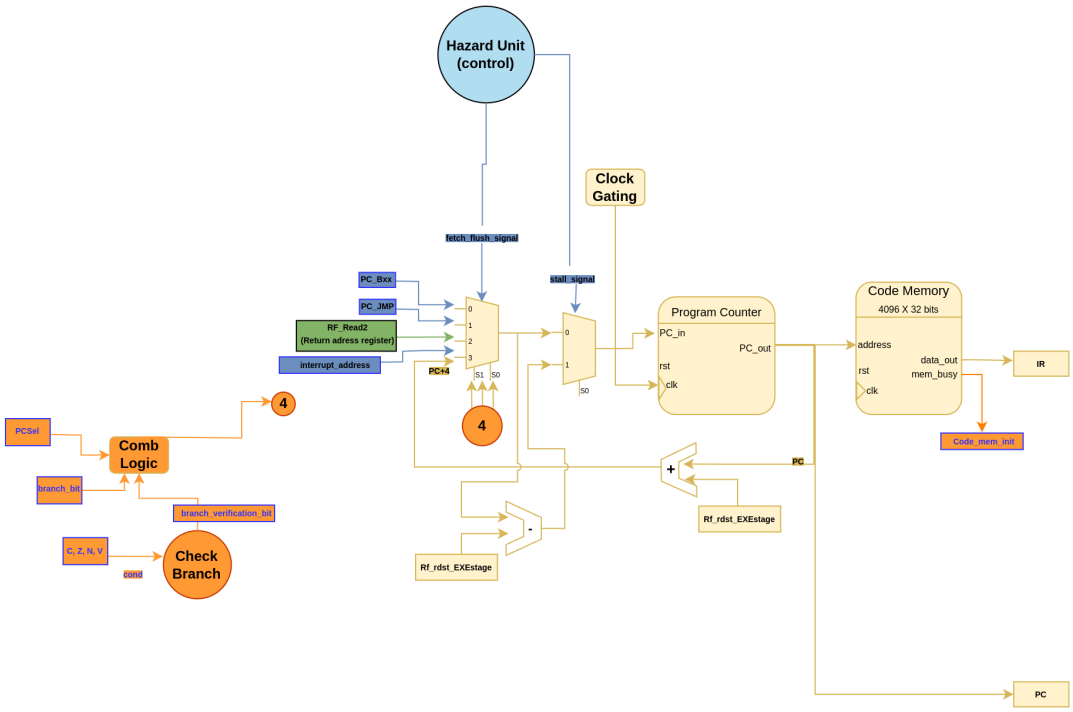

Figure 4.1: Fetch Stage Design

Fetch to Decode Register File

The Fetch to Decode Register File captures and holds data necessary for the transition and as they move from one stage to the other. The PC and IR signals transition through every register file, the Flush signal instructs the register file to clear its contents when hazards are detected and Stall signal pauses data transfer to the execute stage when hazards are detected.

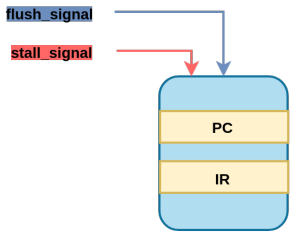

Figure 4.2: Fetch to Decode Register File Design

The **Stall** signal is required because if the decode stage is stalled due to a dependency on a previous instruction, the fetch stage must also be stalled to prevent new instructions from entering the pipeline. The **Flush** signal is required because when the execution flow of the program is changed, due to a branch misprediction or an interrupt, any instructions in the pipeline are invalid and need to be removed to avoid incorrect execution.

Decode stage

Sequentially, there is the Decode stage. The Decode stage in the CPU's pipeline is, perhaps, the most critical as it interprets and prepares the fetched instruction for execution. This stage involves several key components and processes that work together to ensure the instruction is accurately understood and set up for subsequent execution stages. Here's a detailed explanation of the decode stage, its components, and their functions:

- **Instruction Register Decoding:** Once the instruction is fetched from memory and stored in the IR, the decode stage begins by interpreting this instruction. This involves parsing the binary instruction into its opcode and operand parts. The opcode dictates the type of operation to be performed, while the operands specify the registers, constants, or memory addresses involved in the operation.

- **Control Unit:** The Control Unit is responsible for generating control signals based on the opcode of the instruction. These signals are used throughout the processor to guide the movement of data and the operation of various functional units like the ALU or the hazard unit.

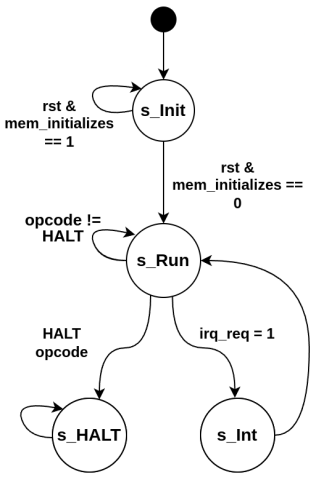

Figure 4.3: Control Unit state machine

As seen in the Figure 4.3, after the initialization, the control unit will stay in the run state, and only transitions from that if an interrupt comes up or an HALT instruction. If an interrupt request surges, the Control Unit will slide to the interrupt state for one cycle to handle that and return to the run state. In the other hand if a HALT instruction happens, the code will jump to the HALT stage and stay there.

- **Hazard Unit:** Integrated into or associated with the decode stage is the Hazard Unit, which plays a crucial role in identifying and managing hazards that could affect the processing of the instruction pipeline. The hazard unit monitors for data hazards and control hazards. As will be explained in the hazard unit section, the only data hazards the hazard unit will need to monitor are RAW hazards. This unit sends signals such as stalls and flushes to ensure that these hazards do not lead to incorrect program executions.

- **Register File:** One of the key resources during the decode stage is the RF. The implemented RF is a pseudo triple-port design, with two read ports and one write port that can be used simultaneously. This setup allows for efficient reading and writing operations within a single clock cycle. The register file it's composed by 32 registers, each 32 bits wide. The latency of reading the register file is one clock cycle.

- **Signal Extension:** In addition, the decode stage extends the necessary signals to ensure proper handling of different instruction types. This extension is crucial for ensuring that when the execution stage is reached, all necessary data and configurations are ready for immediate processing.

The decode stage is pivotal for setting the groundwork for its execution. It integrates closely with other components of the CPU to manage data flow, address hazards, and prepare operations, ensuring that each instruction progresses through the pipeline efficiently and correctly.

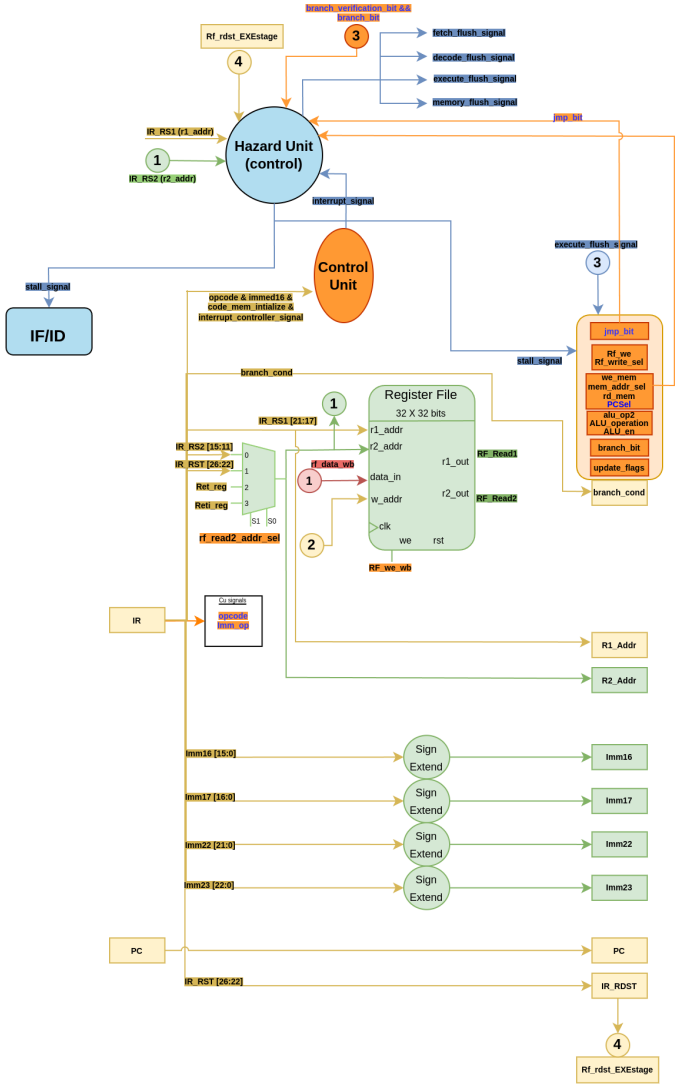

Figure 4.4: Decode Stage Design

Decode to Execute Register File

The Decode to Execute Register file facilitates the transition of data between the decode and execute stages. Similarly to the previous intermediate register file, it uses the Flush and Stall signals to manage its contents. The full signals are demonstrated in Figure 4.5.

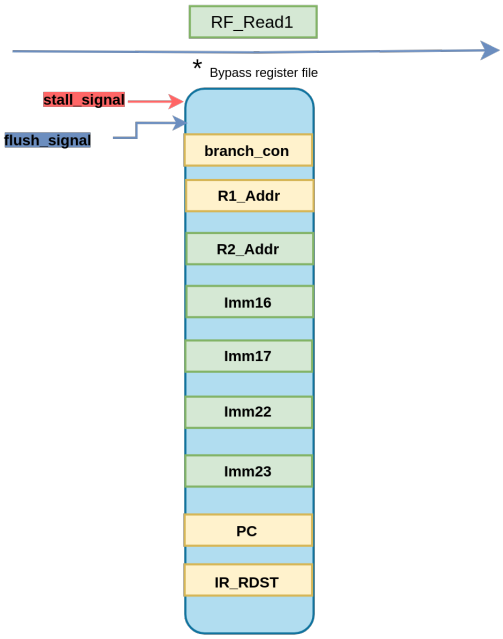

Figure 4.5: Decode to Execute Register File Design

The **Stall** signal is required because the Decode stage can be stalled due to a dependency on a previous instruction. The **Flush** signal is required for the same reason as the previous intermediate register.

Note: The RF output values bypasses this register file due to the latency cycle of the register file.

Execute stage

In the Execute stage of the CPU pipeline, several crucial operations take place, central to which is the function of the ALU. The ALU is responsible for performing all arithmetic and logical operations dictated by the current instruction. Inputs to the ALU are meticulously chosen based on the specific requirements of the instruction being executed. Furthermore, the execute stage is key in handling data hazards through a mechanism known as forwarding.

When there are data dependencies between closely sequenced instructions-where a subsequent instruction requires data from an instruction that is yet to complete forwarding allows the data to be used directly without waiting for it to cycle back through the RF. This is depicted in the stage diagram, figure 4.6 with multiple pathways feeding into the ALU, indicating where data is forwarded to meet dependency requirements.

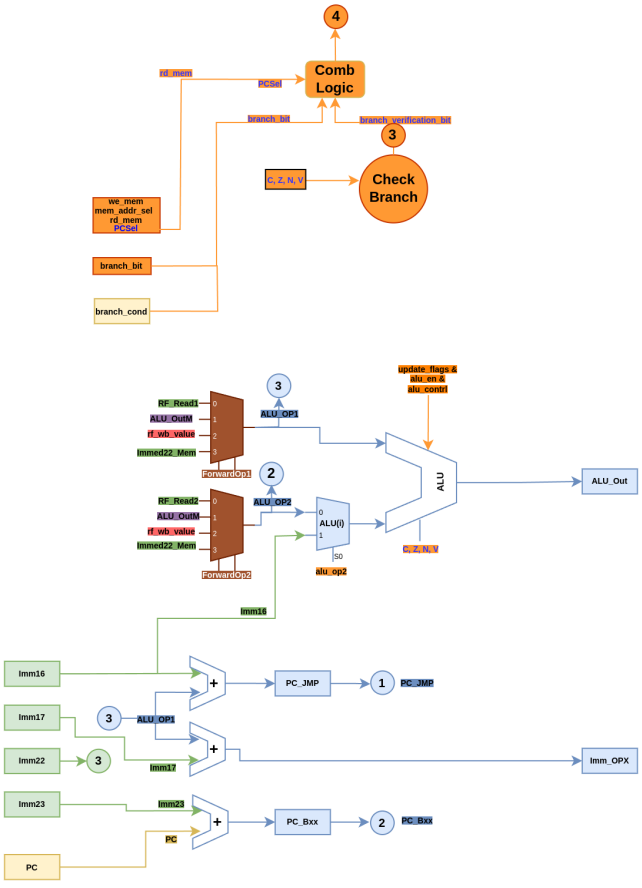

Figure 4.6: Execute Stage Design

Another vital role of the execute stage is branch checking. This function involves the assessment of branch conditions to determine the execution path of the program. Components within the stage such as "Check Branch" are instrumental in this process. They evaluate whether the conditions for branching are met, such as comparing ALU flags to see if they are equal in the case of a BEQ (Branch if Equal) instruction. Depending on the outcome of these checks, the flow of program execution might change, leading to an update in the PC.

In addition the execute stage also calculates effective addresses for memory or control operations.

Execute to Memory Register File

The Execute-to-Memory Register file serves as a bridge between the Execute and Memory stages. It uses the Flush signal but does not require the Stall signal. The rest of its signals are illustrated in Figure 4.7.

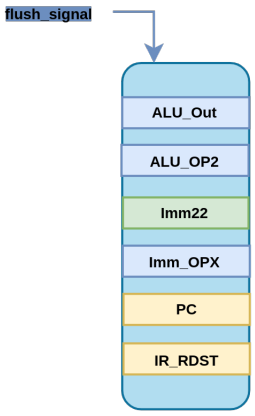

Figure 4.7: Execute to Memory Register File Design

Memory stage

The Memory Access stage of the CPU's pipeline handles reading from or writing to data memory depending on the operation specified by the current instruction. A key component is the memory address selector, which determines the address calculation mode based on the operation type: 0 for operations involving load (LD) and store (ST) instructions, and 1 for indexed operations such as STX and LDX.

Data Memory, depicted as a 1024 x 32 bits block, performs the read and write operations. For reads, data from the calculated address is fetched and sent out for use in later stages, while for writes, incoming data (data-in) is written to

the specified memory address. Control signals including Write Enable (WE) and the clock (clk) ensure these operations are precisely timed and executed, with a multiplexer directing the selected address to the memory, ensuring the correct memory location is accessed according to the instruction requirements. This memory stage is vital for the CPU's functionality, efficiently managing the data flow necessary for executing various types of memory-interacting instructions.

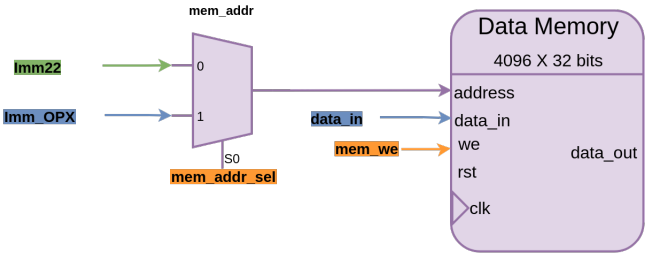

Figure 4.8: Memory Stage Design

Memory to Write-Back Register File

The Memory to Write-Back Register file is the final intermediate register. At this point in the pipeline, the instruction has already been executed and the memory access has been completed. There are no more pipeline stages dependent on this instruction, so flushes or stalls are not required.

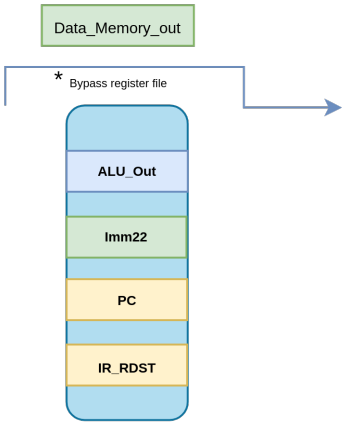

Figure 4.9: Memory to Write-Back Register File Design

Note: The output value from the Data Memory bypasses this register file because of the latency of reading from the register file.

Write-Back stage

The Write-Back stage of the CPU pipeline where the results of computations or memory operations are finalized and recorded within the processor. Central to this stage is a multiplexer that selects the source of data to be written back based on the instruction type. The options include outputting a memory value typically for load instructions, storing the program counter in a register as in jump-and-link operations, writing back ALU results, and loading an immediate value into a register.

The WB stage is crucial for updating the processor's state, facilitated by control logic that manages data writes to the register file or specific system registers. This special register is pivotal during interrupts or special operations, saving the current PC for later restoration, ensuring the system can resume correctly post-interruption.

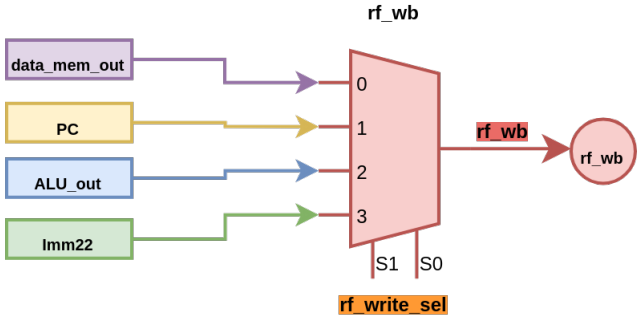

Figure 4.10: Write-Back Stage Design

Considering the previous explanation, in Figure 4.11 it's represented the full design of the Datapath pipelined.

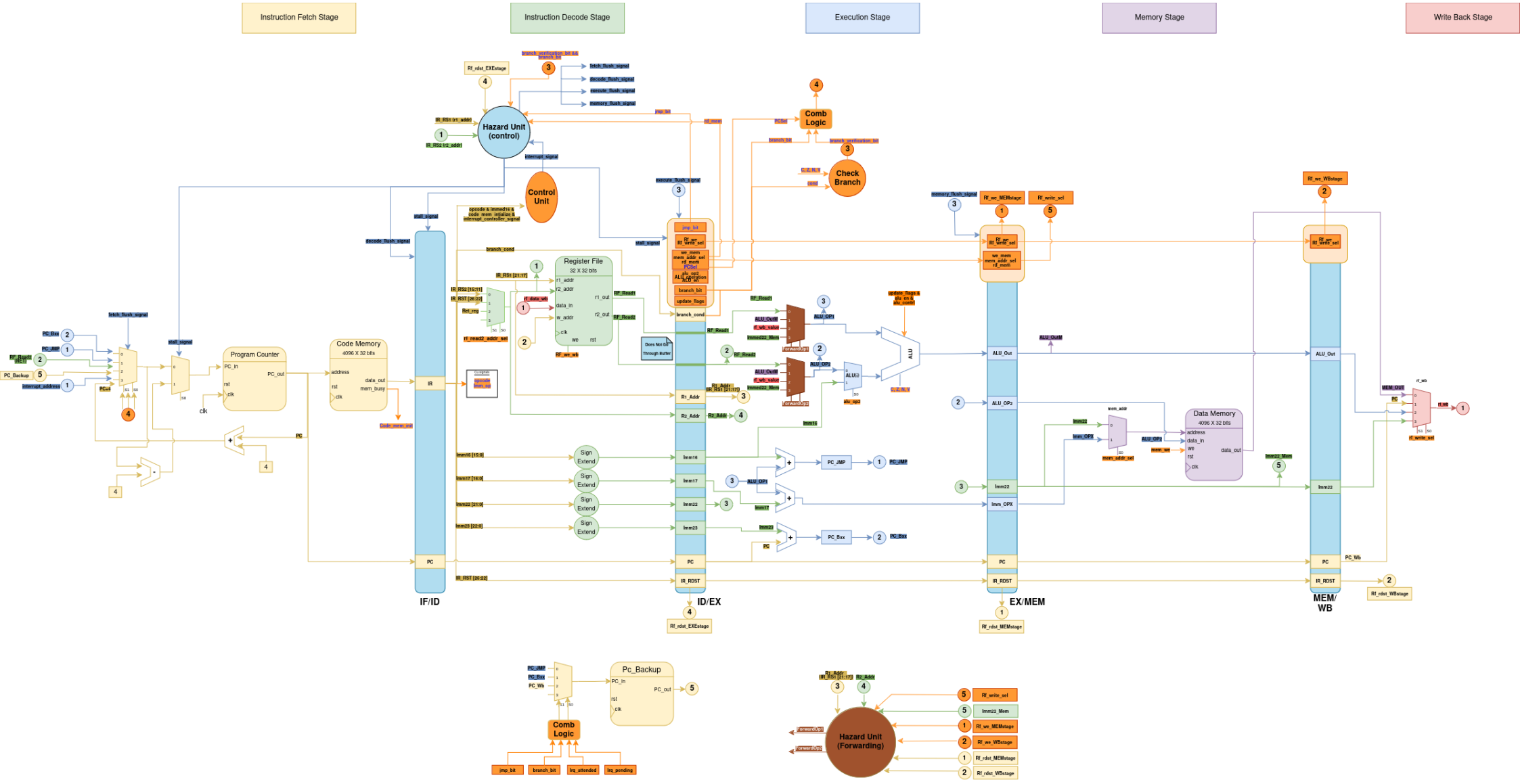

Figure 4.11: Datapath pipelined overview

4.1.2 Hazard Unit

Structural Hazard Mitigation

The analysis showed that structural hazards arise when multiple instructions attempt to access the same hardware resource. To mitigate this issue a Harvard architecture was chosen, where code and data memory are separated, effectively eliminating the possibility of structural hazards by multiple access.

Another possible structural hazard could occur if the register file is accessed during the cycle in which the pipe executes the Write-Back and Instruction Decode operation. However, this is not an issue because the register file is configured in 'write first' mode. This means that if two instructions access the resource, the write operation will always be performed first, ensuring that the value is always up to date. To address this issue, some architectures use half clock cycles to access the register file. Specifically, the RF is accessed for the write-back operation on the rising clock transition, and by the instruction Decode stage during the falling transition. This approach helps to improve efficiency and reduce delays.

Stalling a cycle to avoid shared accesses on the same pipe is the most time-consuming solution. Although it is easier to implement, it is not the most advantageous in terms of performance.

Figure 4.12 illustrates the elimination of potential structural hazards by transitioning to a Harvard architecture. This architecture allows for simultaneous access to memory resources, including data and code memory, as they are separated.

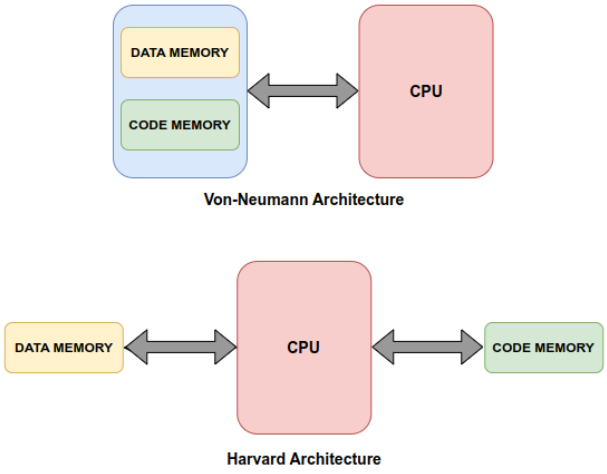

Figure 4.12: Von-Neumann vs Harvard Architecture

Data Hazard Mitigation

As seen before, the VeSPA processor's pipeline will have 5 stages, where only one stage will write to the register file, which is the write back stage. As said in the analysis,only the RAW data hazards need to be solved and some of the possible solutions will be presented in this section, as well as their trade-offs.

Instead of stalling the pipeline by hardware, it is also possible to insert NOP's by software or re-order the instructions in a way that they become less dependent. Although this solution reduces the cost of hardware and the length of the datapath, it overloads the compiler and also if the pipeline architecture changes, there would be more need to re-design the backend of the compiler.

Other possible solution is to detect the hazard with a dedicated unit using combinational logic and then also stall the pipeline until the value becomes available in the register file. This solution, as opposed to the previous one, would only stall the pipeline for the RAW hazards, since the logic can be done to only detect those kind of hazards. Because of that, this one would significantly increase the performance when compared to the last one. However, stalling the pipeline every time a RAW dependency occurs might still not be the best solution in terms of performance.

The other solution is the data forwarding or bypassing, which is represented in the Figure 4.13. Taking the example of that figure, the second instruction needs the value of R2 in the execute stage, but the value was not yet written to the register file, which only happens in the fourth cycle. The same happens for the third instruction, because it needs the value R2 in the fourth cycle and it has not yet been written. However, the value of R2 is ready at the end of the second cycle, so the result can be forwarded with direct value communication to the execute stages of the second and third instructions. The benefits of this solution is that there would be less stalling when executing instructions, improving the performance at the cost of adding more hardware to the datapath for direct communication of the forwarding value.

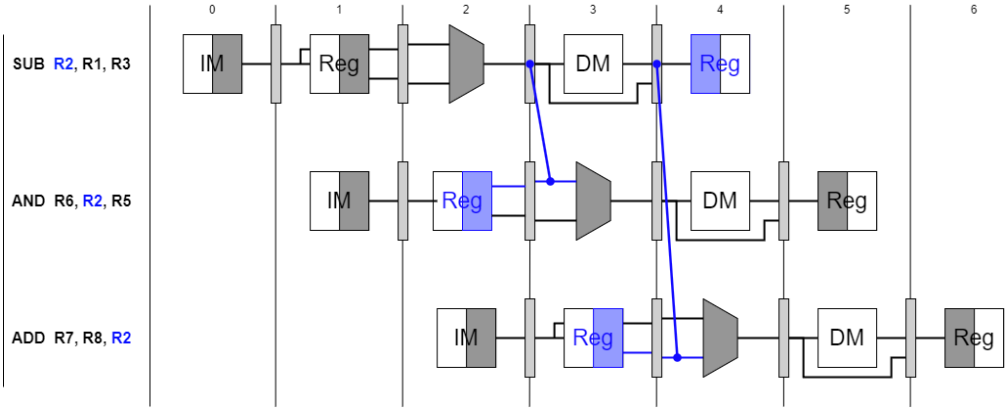

Figure 4.13: Pipeline Bypass representation

The data forwarding solution covers most of the RAW dependencies, but if the instruction that writes to the register before others source it is a load instruction, the situation would be different. The load instruction accesses the data memory to get the value that will be written in the last stage, so the value is not ready to be forwarded at the end of the load execute stage, but only on the end of the data memory stage. A solution for this is to stall the pipeline for one cycle and then data forward the result, as shown in Figure 4.14.

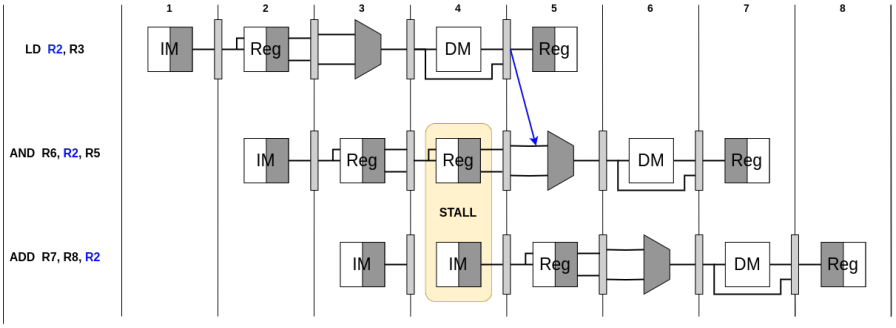

Figure 4.14: Dataforward with load dependency

Given the trade-offs above, the group decided to go for the data forward solution and to stall the pipeline only when the data hazard is a load dependency. The flowchart for the data hazard resolution is represented in Figure 4.15.

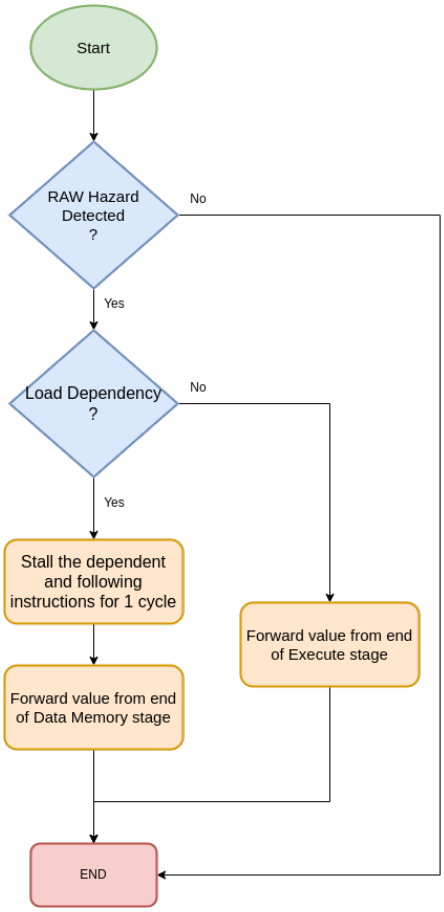

Figure 4.15: Data Hazard Flowchart

Control Hazard Mitigation

To address the control hazard inherent in VeSPA, particularly during Branch or Jump instructions, a systematic approach is implemented. Recognizing the distinct requirements of Branch and Jump implementations, distinct approaches are necessary.

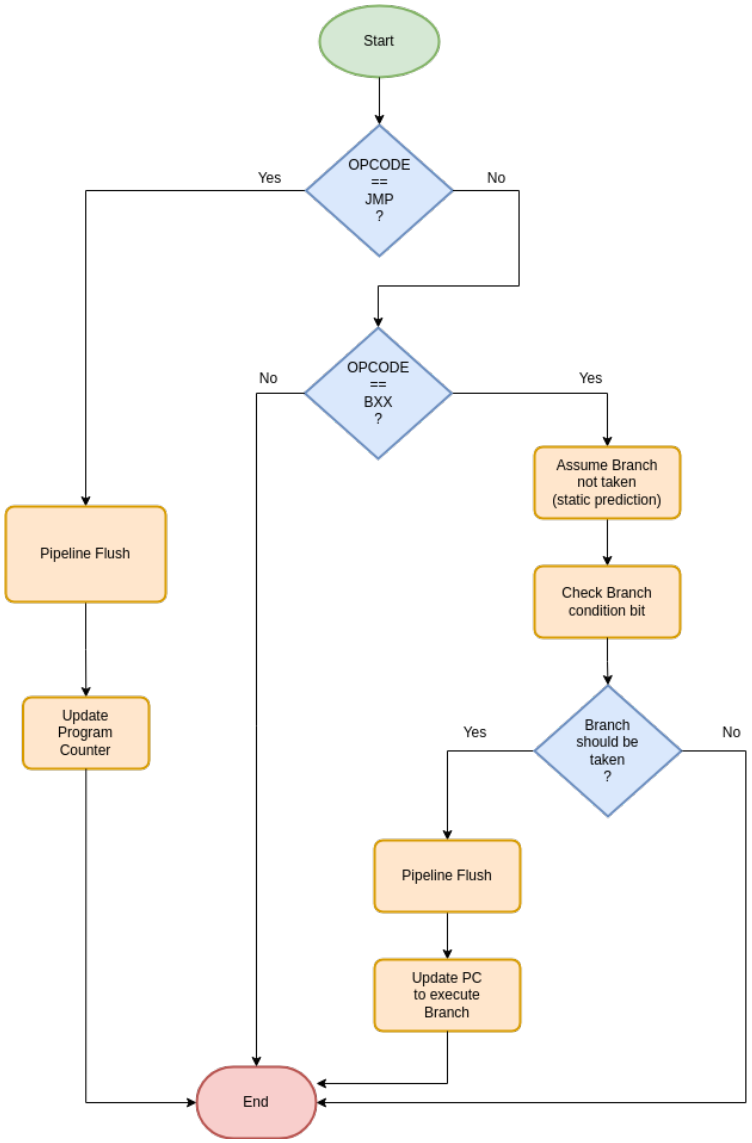

Figure 4.16: Control Hazard Flowchart

Jump instructions mandate an immediate value and a register value, entailing passage through the decode stage for accessing the RF value. To prevent erroneous instruction execution resulting from the PC value, a flush of the pipeline stages is executed and the PC is updated accordingly.

Branch instructions on the other hand, require only an immediate value present in the IR, which is subsequently added to the current PC. A static branch prediction is employed, presuming the branch will not be taken. In the event of a misprediction,

a pipeline flush ensues. This approach was selected due to its simpler branch prediction logic and reduced misprediction penalty for balanced or biased branches.

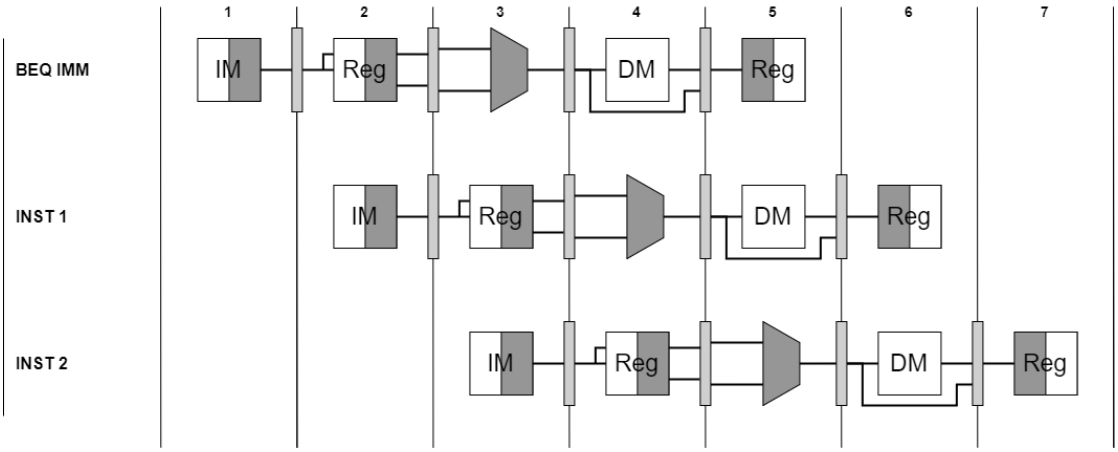

Figure 4.17: Branch not taken Prediction Success

The following figure illustrates a scenario where a misprediction occurs, leading to a pipeline flush.

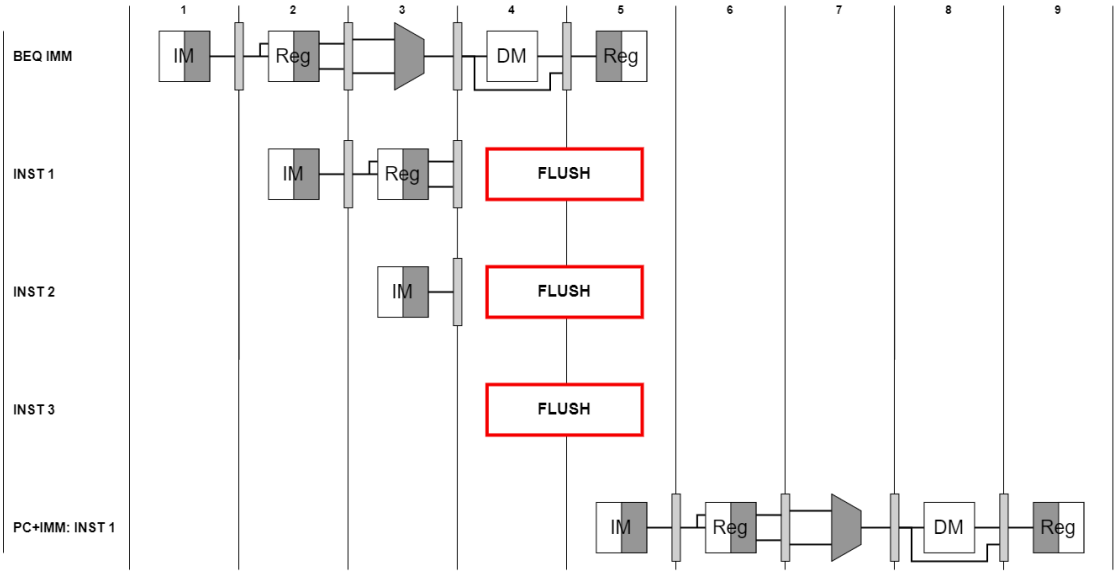

Figure 4.18: Branch not taken Prediction Failure

The following picture shows an interface that will be utilized for both the Control Hazard and the Forward Unit.

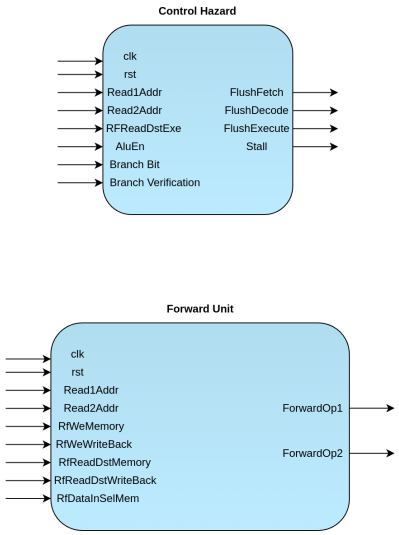

Figure 4.19: Control Hazard and Forward Unit

4.1.3 Barrel Shifter

As previously mentioned, the shift instruction (left and right) was added. To achieve this, a Barrel Shifter module will be created and instantiated within the ALU to execute the instructions. The Barrel Shifter is implemented as a cascade of multiplexers, each performing a shift of 1, 2, 4, 8, and 16 bits.

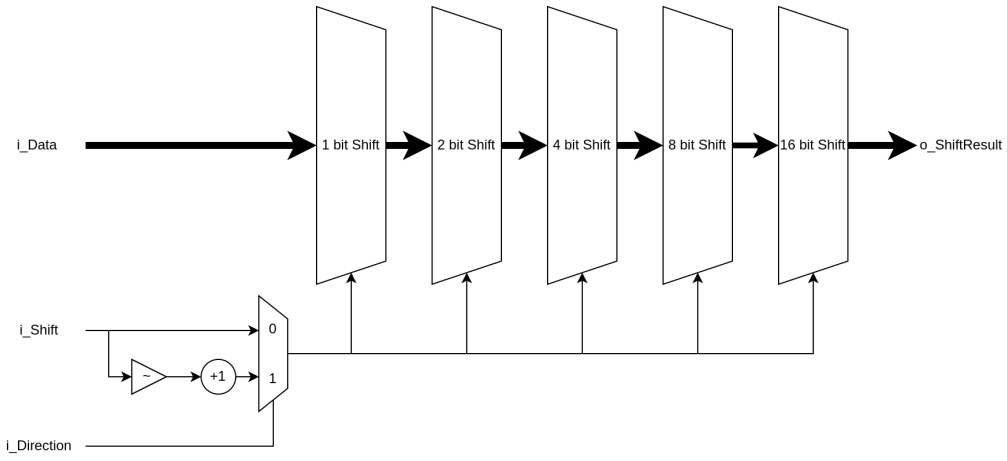

Figure 4.20: Barrel Shifter Design

4.2 Bus

For the specified bus, an Interconnect was developed, which will be the intermediary between the CPU and the peripherals, this will be purely asynchronous so as not to introduce latency in the write and read processes, as seen in Figure 4.21. The main function of the interconnect bus is to guide the signals in function of the memory addresses sent by the CPU.

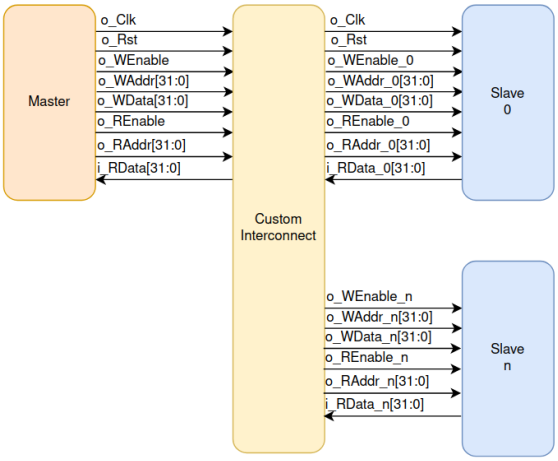

Figure 4.21: General Interconnect representation

According to the address, the interconnect will divert the signals to the peripheral to which the CPU wants to communicate.

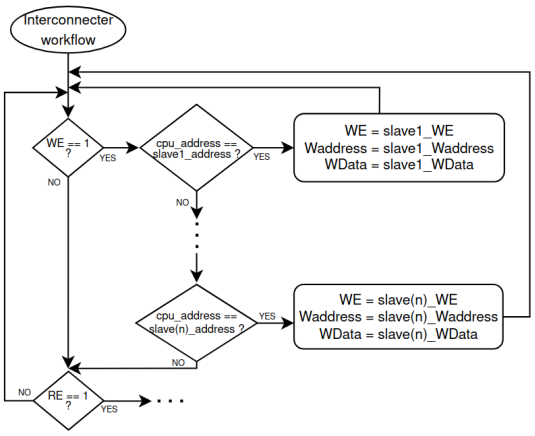

Figure 4.22: Interconnect write process

As shown in figure 4.22 above, when the interconnect detects the "write enable" signal, it determines which slave the address corresponds to. After that, the write signals are sent to the slave. The process is exactly the same for reading, but with the read signals.

The writing and reading process would therefore be generic between all the slaves and the CPU because of that, on the slaves' side, in order to accept the data in the format sent by the CPU, a wrapper has been developed which is applied to each of the peripherals.

The following diagram shows how this wrapper will be connected:

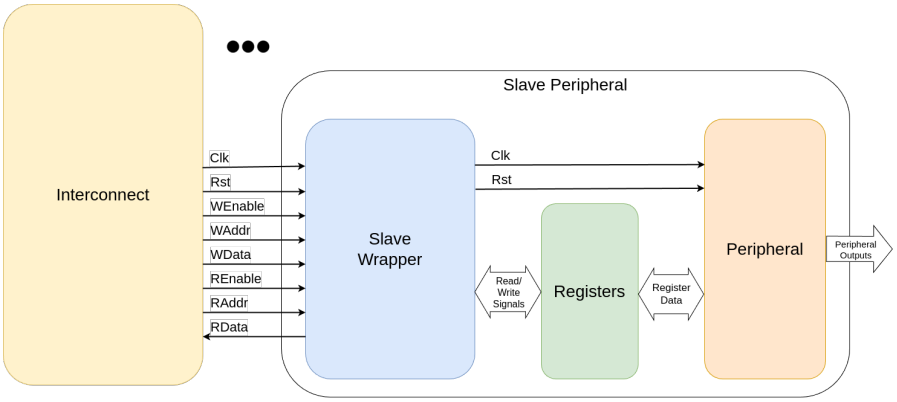

Figure 4.23: Interconnect Peripheral Wrapper

4.3 Peripherals

Various peripherals have been developed which will then be connected to the CPU via a bus. To do this, the various peripherals were mapped so that they could interact with the CPU. The peripherals are the Interrupt Controller, GPIO, PS/2, Timer, UART and VGA. VGA will not be mapped due to the lack of FPGA resources. Each peripheral will have four internal registers for its configuration.

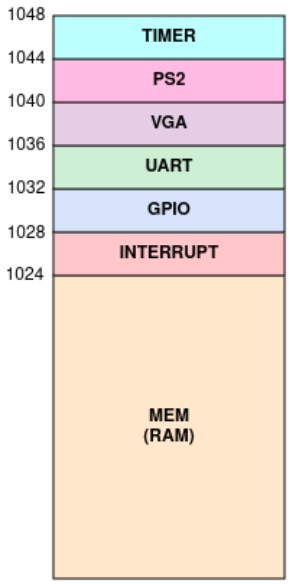

Figure 4.24: Peripheral Mapping

4.3.1 Interrupt Controller

The interrupt controller to be developed has some particular features that should be specified to provide a better insight on its inner-working:

- Capable of handling interruptions from up to 4 different sources;
- Interrupts have a fixed priority.
- Capable of managing interrupts priority, where the interruption with the lowest index has the highest priority;
- Capable of notifying the CPU of pending interrupts.

In order to implement a component capable of all of the referred features, the connection signals were established as displayed in figure below.

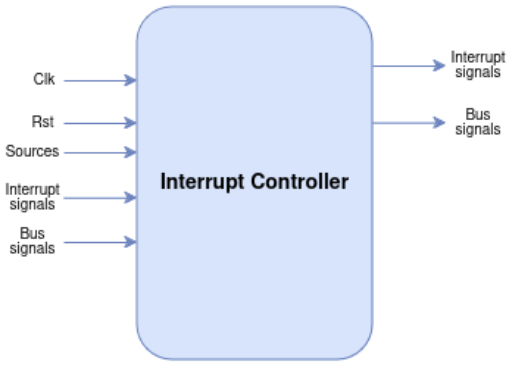

Figure 4.25: Interrupt Controller Overview

VeSPA utilized a custom communication bus to connect the CPU with various attached peripherals. The same bus will be used to interface with the interrupt controller.

The interrupt controller will have four sources, and four registers will be used control them, as shown in figure 4.26. The registers are "EA" to enable all interrupts, "EN1", "EN2", and "EN3" to enable/disable each interrupt individually.

Interrupt number zero is always active because it functions as the system's hardware fault detector. This interrupt triggers for events like out-of-bounds memory access, eliminating the need for a specific enable register.

The specified registers control the interrupt functionality on the custom bus. These registers, named "i_WAddr" for writing and "i_RAddr" for reading, are detailed below.

IE : 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	EN3	EN2	EN1	EA
-	-	-	-	-	-	-	-	-	-	-	-	RW	RW	RW	RW

Bit 0 : **EA**
0 -> Fully disables interrupts from all sources
1 -> Enables interrupts from all sources

Bit 1 : **EN1**
0 -> Fully disables interrupt from source 1
1 -> Enables interrupt from source 1

Bit 2 : **EN2**
0 -> Fully disables interrupt from source 2
1 -> Enables interrupt from source 2

Bit 3 : **EN3**
0 -> Fully disables interrupt from source 3
1 -> Enables interrupt from source 3

Figure 4.26: Interrupt Controller Registers

The figure 4.27 highlights the connections to the interconnect. Green signals handle reading and writing to registers, while red signals are specific to interrupts, receiving information from sources and notifying the CPU directly. Each interrupt has a fixed position in the interrupt vector, located at the beginning of the code memory.

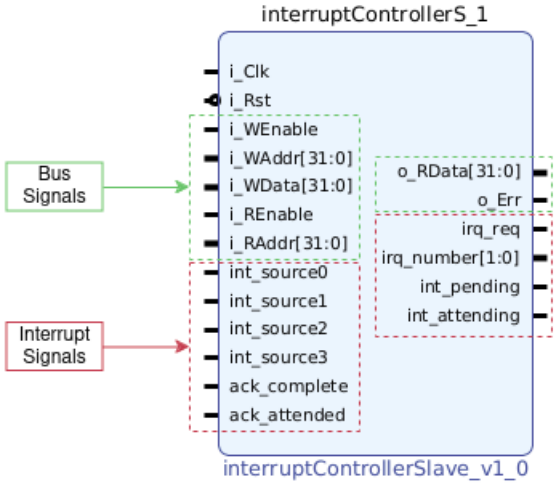

Figure 4.27: Interrupt Controller interface

The following table, contains the interrupt signals and their description.

Signals	I/O	Meaning
i_Clk	I	Clock signal
i_Rst	I	Reset signal
int_source0	I	Interrupt source 0
int_source1	I	Interrupt source 1
int_source2	I	Interrupt source 2
int_source3	I	Interrupt source 3
ack_complete	I	Inform IC that interrupt as finished
ack_attended	I	Inform IC that interrupt was attended
irq_req	O	Inform CPU of interrupt request
irq_number	O	Inform CPU which interrupt source is in execution
int_pending	O	Informs CPU of a pending interrupt
int_attending	O	Bit set between the attended and complete signals

Table 4.1: Interrupt Controller signals

In this pipeline system, the **hazard unit (control)** is responsible to control the attending process of a interrupt. This unit controls the flush of the pipeline on interrupt. When the control unit receives the irq_req signal indicating a pending interrupt, the current PC value is stored in the shadow register Pc_Backup. The address from the interrupt vector is loaded into the PC, and all pipeline stages are cleared except for the write-back stage.

After the interrupt is serviced, upon encountering a RETI instruction, the ack_complete signal is sent from the CPU to the interrupt controller, and the PC is restored from the shadow register.

Pending Interrupt

When an interrupt occurs while another is being serviced, the RETI instruction for the first interrupt is ignored. Instead, the interrupt vector is jumped to directly, and the interrupt code is executed immediately.

CPU modifications

In order to correctly integrate the interrupt controller, small modifications to some stages of the CPU pipeline were required.

The first change involved adding a shadow register to store the PC value of the last instruction executed when an interrupt occurs. During the Interrupt Service Routine execution, the PC value stored in the shadow register remains unchanged

to preserve the pipeline context. Since the PC value is not always linear, additional signals and logic are required to manage this complexity, as illustrated in figure 4.28.

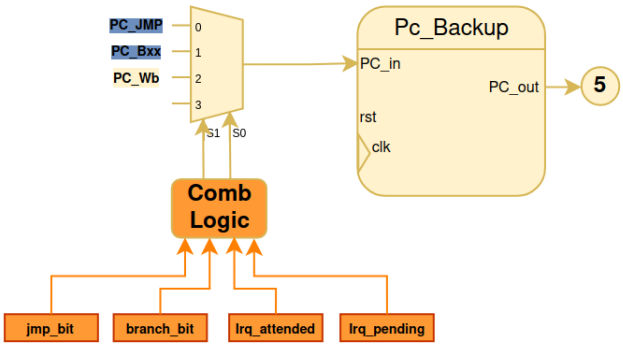

Figure 4.28: Modifications to the shadow register (PcBackup)

The second change, prompted by the first, involved adding two new entries to the multiplexer controlling the PC value in the fetch stage: one for the interrupt value, referencing the interrupt vector, and another for the RETI. This ensures that the pipeline context is not lost, as shown in figure 4.29.

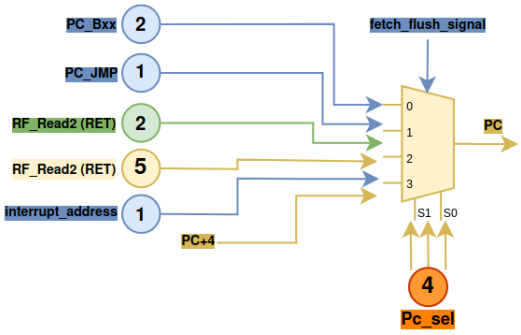

Figure 4.29: Modifications done in fetch stage

Three key scenarios must be studied when an interrupt occurs:

- When there is no control instruction in the current pipeline;
- When a control instruction is in the execute stage, and the jump is taken at that moment;
- A few clock cycles after a control instruction occurs;

Example 1:

Assuming the simple case, when there is no control instruction in the current pipeline. In the first clock cycle, after initialization, the pipeline will be cleared, and no instructions will be running, as shown in figure 4.30.

Given this instruction sequence:

```
.CODE

@0x00    SUB R1, R2, R3
@0x04    AND R4, R5, R6
@0x08    OR R7,R8, R9
@0x12    ADD R3, R3, R1
@0x16    ST R5, #100
...
```

Listing 4.1: Test Code 2

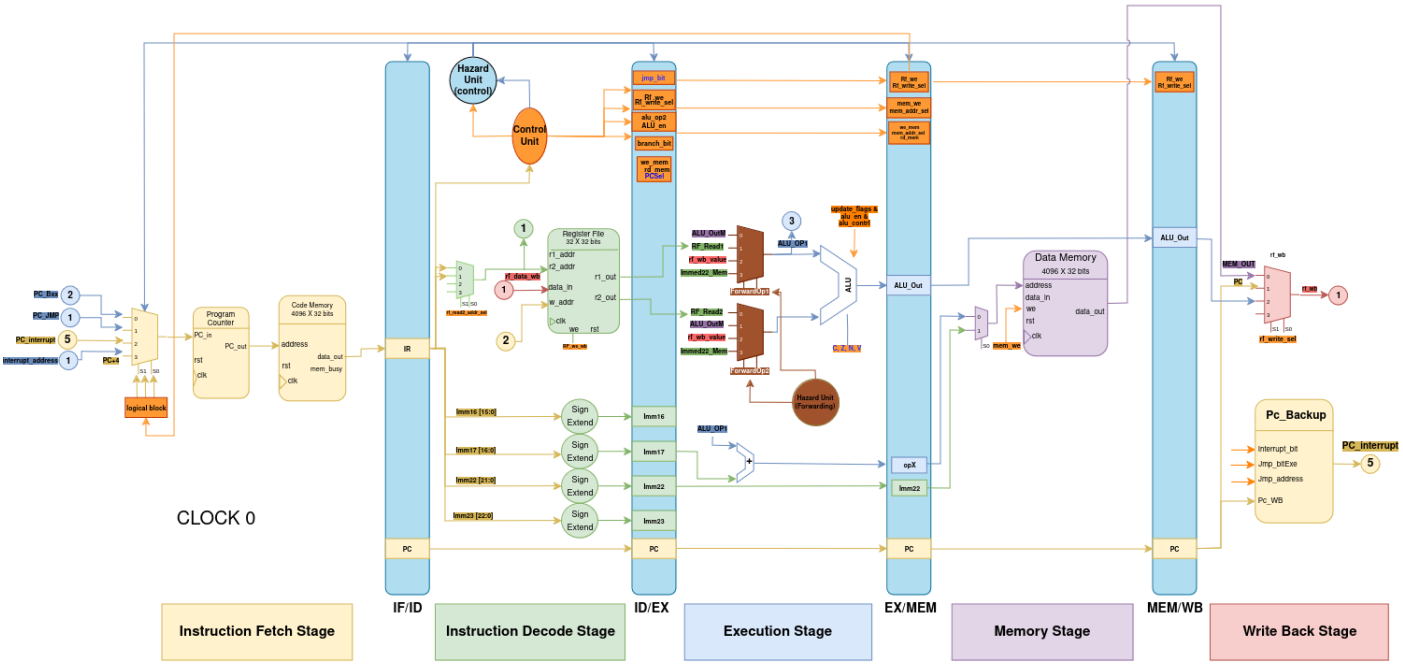

Figure 4.30: Interrupt Controller scheme for clock 0

Six clock cycles later, assuming the running code is as described above, the pipeline will be as illustrated in figure 4.31. If an interrupt is generated by source 1 at this moment, the sequence will be interrupted. After the control unit receives the interrupt, two instructions will be executed: the "sub" instruction in the WB

stage and the "and" instruction in the memory stage. Since the "and" instruction is at index "@0x04", its PC value is "@0x08" (the address of the next instruction). This value will be stored in the shadow register.

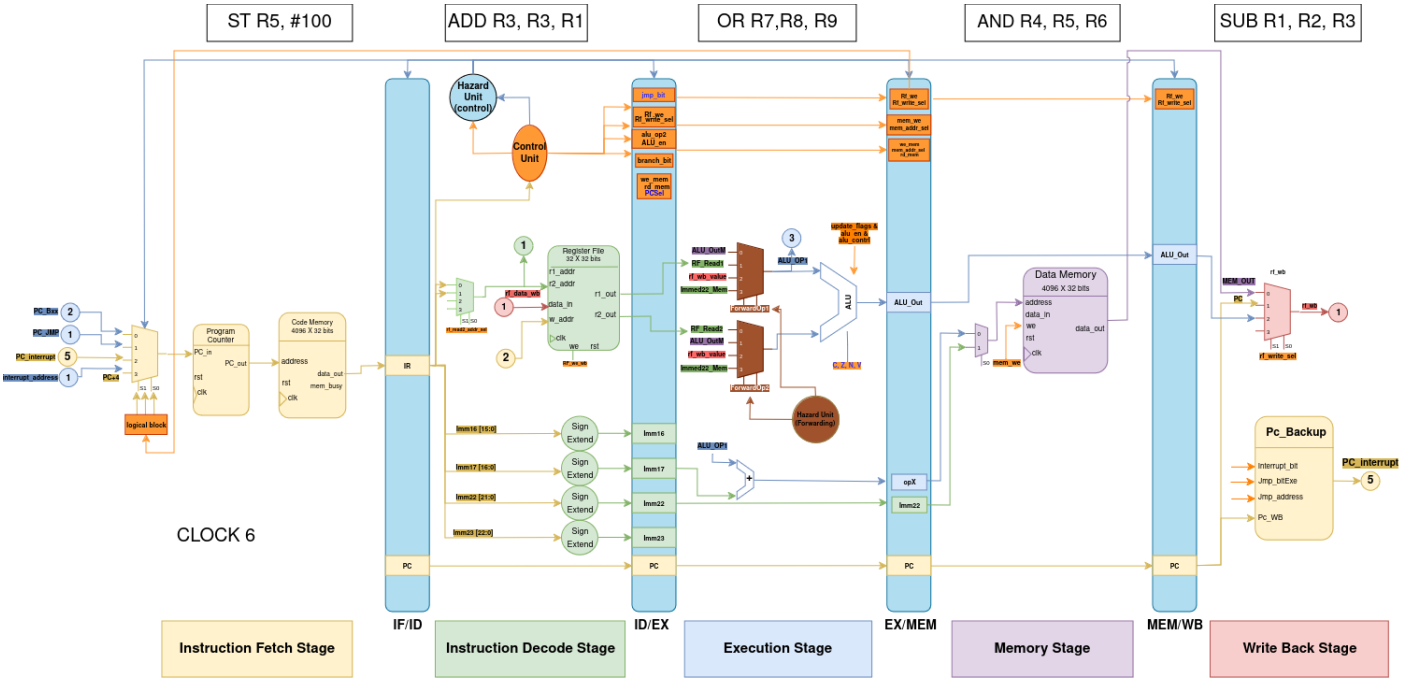

Figure 4.31: Interrupt Controller scheme for clock 6

After the pipeline is flushed, the subsequent instruction will be a jump to the corresponding code section determined by the interrupt vector, depending on the source of the interruption. Assuming the code below represents the section of code for the source one interrupt:

```
1 .CODE
2
3 @0x200   AND R4, R5, R6
4 @0x204   LDI R12, #100
5 @0x208   RETI
6 @0x212   ADD R9, R0, R1
7 @0x216   SUB R2, R3, R7
8 ...
```

Listing 4.2: Test Code 2

In this scenario, when the RETI instruction reaches the execute stage at the next positive clock edge, the memory and write-back stages are completed, while all other stages are flushed. The value stored in the shadow register, "@0x08", representing the last instruction not executed, is then loaded into the PC.

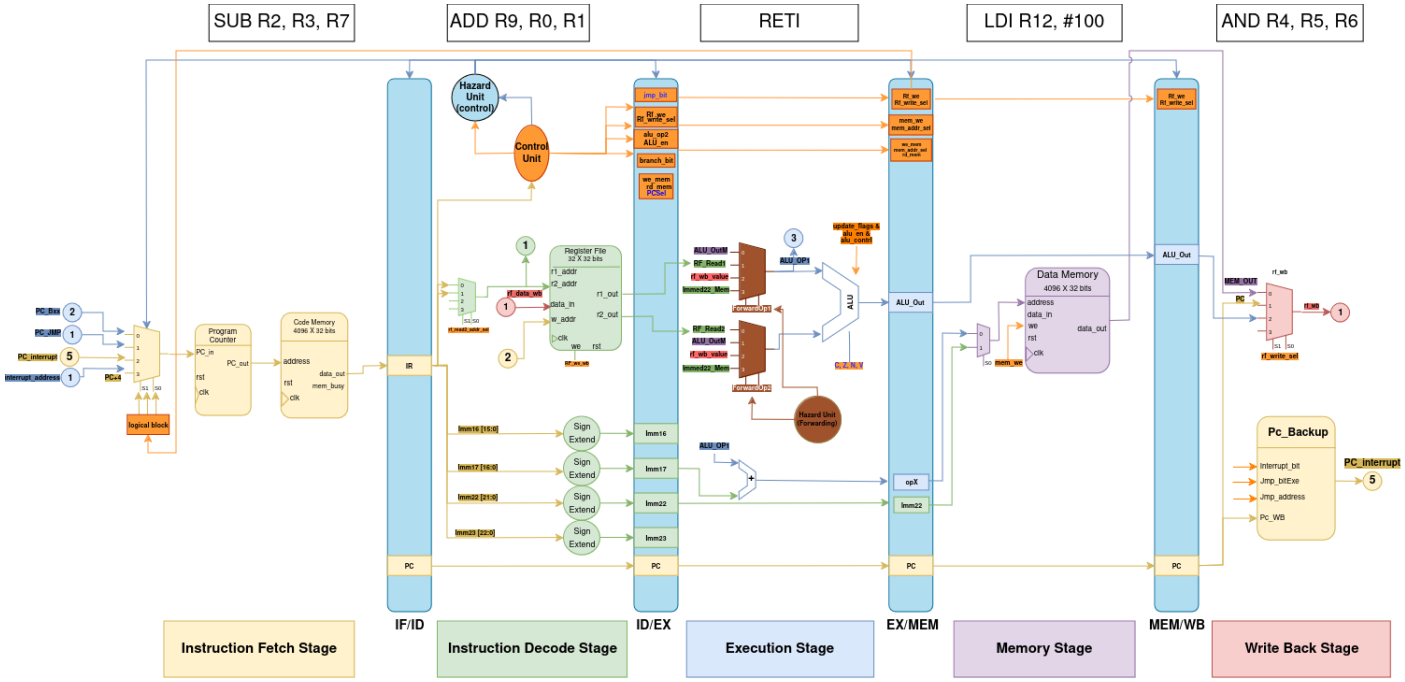

Figure 4.32: Pipeline when interrupt is finished

Example 2:

The next scenario is a bit more intricate, such as studying the case when an interrupt occurs while a jump is in the execute stage, and the jump is taken at that moment.

The example provided is the following code:

```
.CODE

@0x40    ST R5, #100
@0x44    ADD R3, R3, R1
@0x48    JMP R30, 0
@0x52    ADD R9, R0, R1
@0x56    SUB R2, R3, R7
...
```

Listing 4.3: Test Code 2

Disregarding the initial phase, where the pipeline stage remains the same as in the previous example, as depicted in figure 4.30 with no instructions executing, it is important to analyze what occurs a few clocks later.

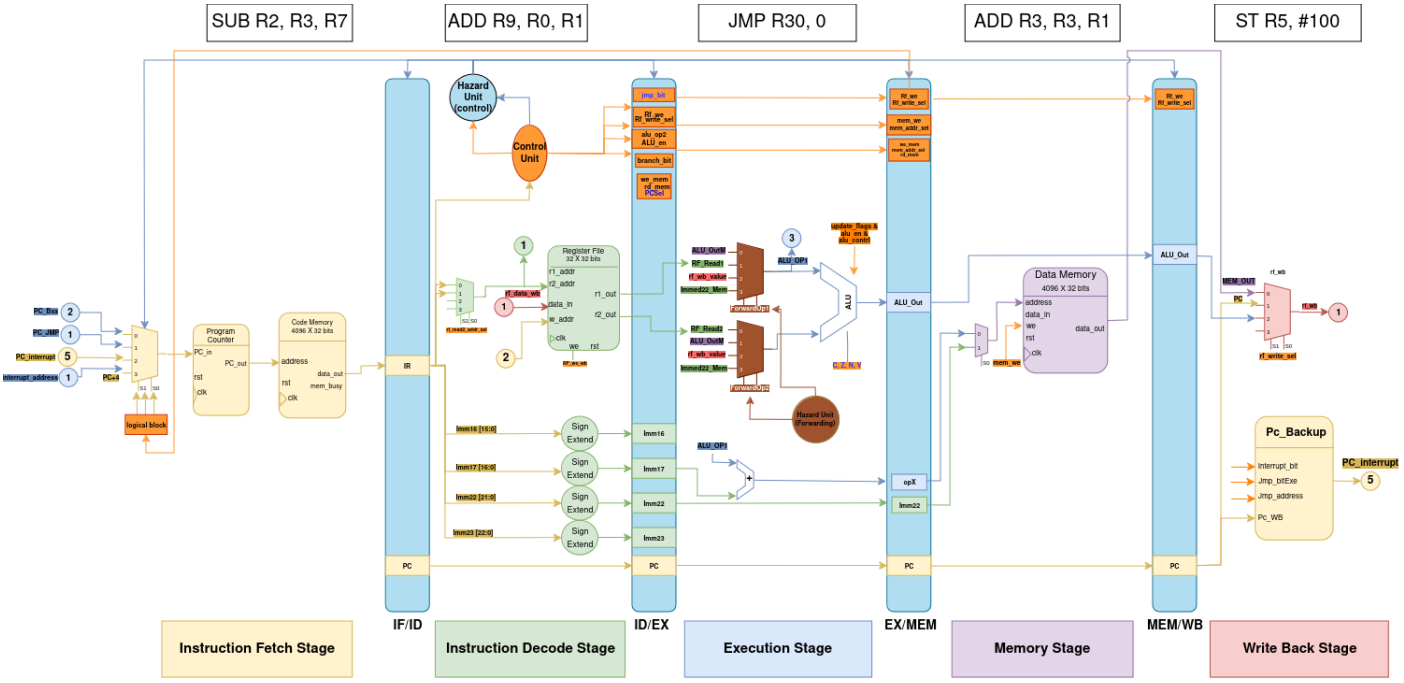

Figure 4.33: Pipeline when interrupt is finished

The moment depicted in figure 4.33 above represents a special case, where the value stored in the shadow register will not be the PC from the WB stage as seen previously.

Upon receiving an interrupt, only in the subsequent cycle will the flush occur. Consequently, the two instructions in the memory and write-back stages will be executed. Due to this, the jump destination value will be the value stored in the shadow register, rather than the PC value from the write-back stage.

After the interrupt is handled, the pipeline will resume from the stored jump value.

Example 3:

This last case, when an interrupt occurs a few clocks after a control instruction, although it is not easy to detect, it is crucial for the correct functionality of interrupts in the pipeline. Let's consider the following code:

```
.CODE

@0x04    SUB R2, R3, R7
@0x08    AND R9, R0, R1
@0x12    JMP R30, 80
@0x16    ADD R3, R3, R1
@0x20    ST R5, #100
...

@0x80    LDI R27, #200
@0x84    ST R19, #24
```

Listing 4.4: Test Code 2

As observed previously, after a jump occurs, the instructions in the memory and write-back stages will be executed, the pipeline will be flushed, and it will start afresh.

The issue arises if an interrupt arises before one of those instructions reaches the WB stage. Before this occurs, if an interrupt is triggered, the stored value will be zero because the write-back stage is still empty.

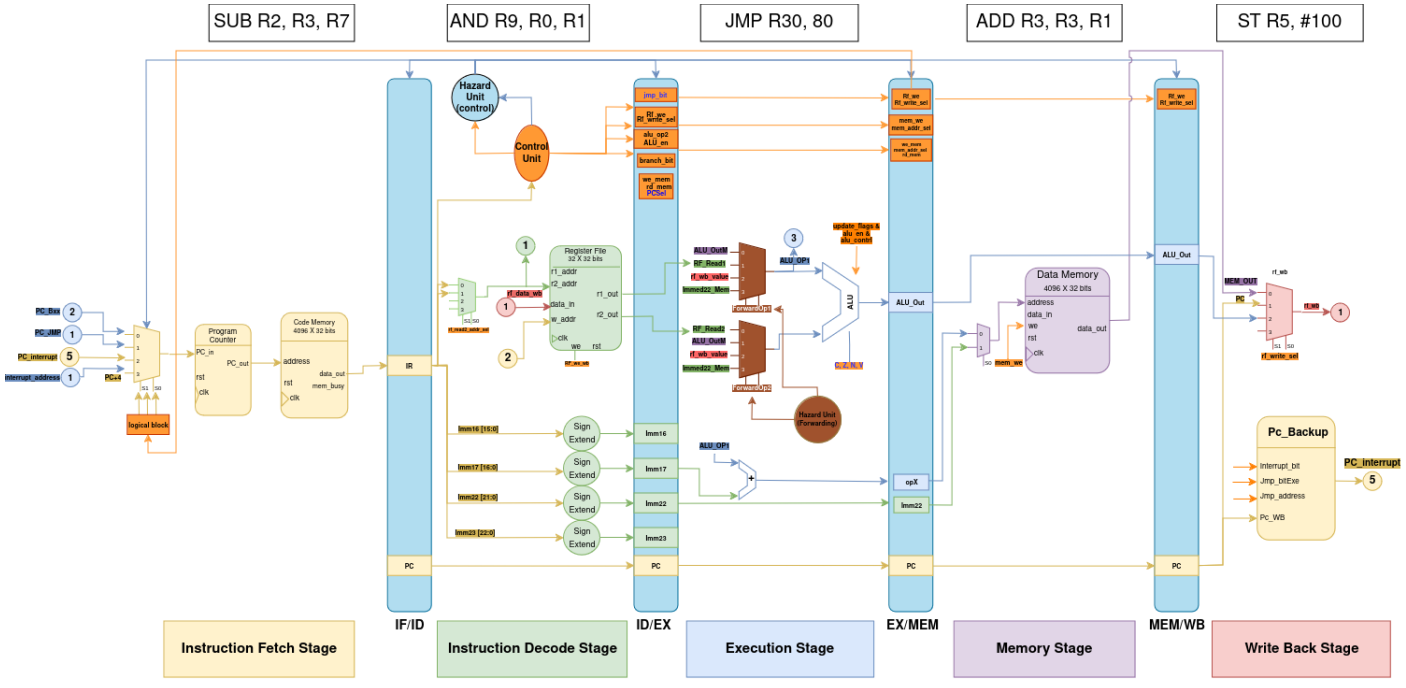

Figure 4.34: Pipeline during JMP execute stage

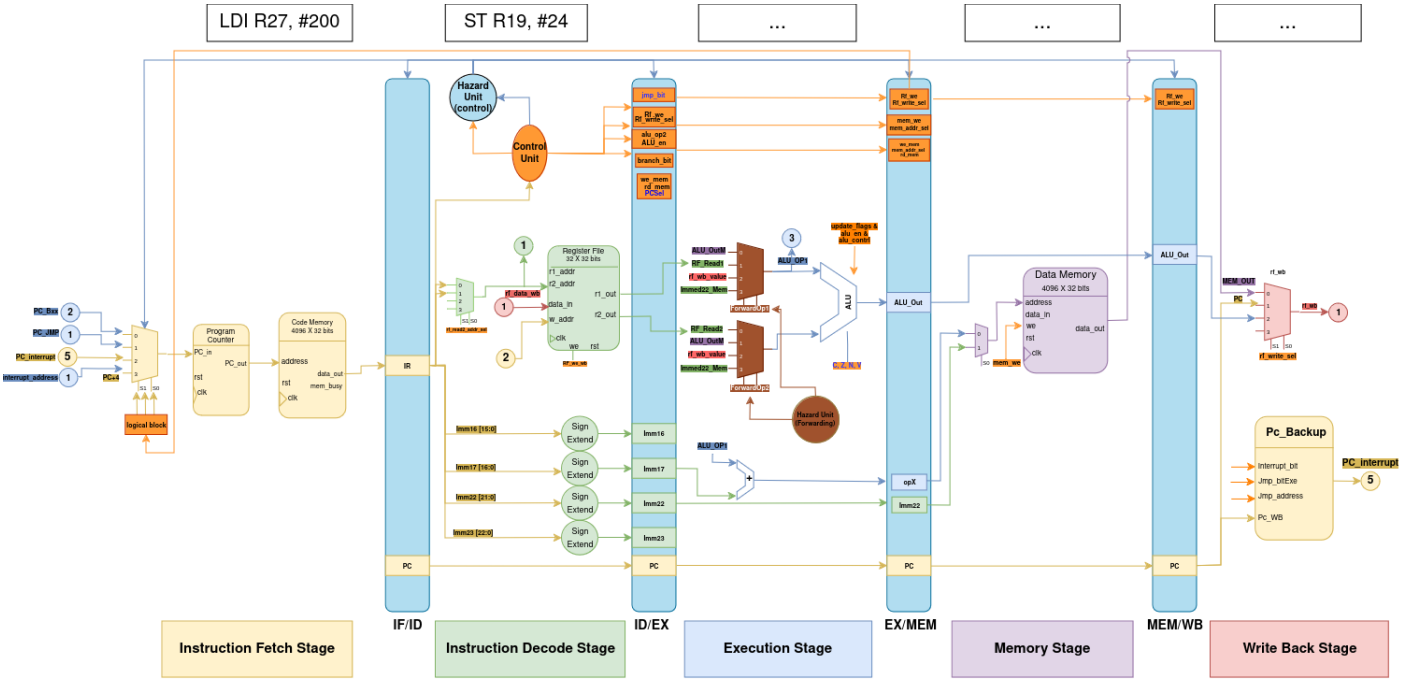

Figure 4.35: Pipeline when JMP is done

In the initial moment, as depicted in figure 4.34, you can observe the current state of the pipeline before the jump occurs. Following that, in figure 4.35, you can see the state of the pipeline after the jump.

If an interrupt occurs at this juncture, the PC value in the WB stage will be null. Therefore, it becomes necessary to store the PC address of the jump in the shadow register for the next 6 cycles after the jump. This duration accounts for the time needed for the first instruction after the jump to reach the write-back stage.

Consequently, if an interrupt occurs in the scenario depicted in figure 4.35, the value stored in the shadow register will still be "@0x80". Following the handling of the interrupt, the pipeline will restart from this stored value.

4.3.2 GPIO

The GPIO peripheral is a simple module, and its structure is shown in the figure below.

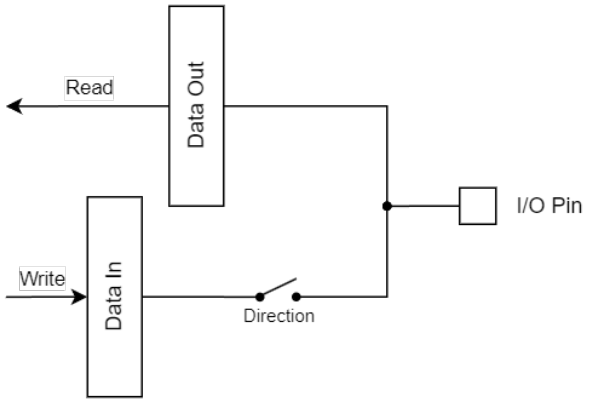

Figure 4.36: GPIO directions

In this module, data intended for output are written to the output register, while incoming data are read from the input register. The operational mode is determined by the specified direction setting. A direction value of 0 indicates that any externally provided input through the I/O port will be written to the input register. Conversely, if the direction is set to 1, the output register will directly control the I/O port.

The GPIO pins work in high impedance mode (also known as high-Z or tri-state) mode, which means it does not drive the signal actively to either a high or low voltage level. Instead, it becomes receptive to external signals, allowing it to detect changes in voltage or receive input from other devices without causing interference.

Below, detailed figures of each register are provided to enhance comprehension of their structures.

- **GPIO_DATA:** The GPIO_DATA register is used to read or write 8-bit data to a specific pin. It allows the user to interact with the pin's state by either retrieving its current value or setting a new value. Upon reset, all pins are initialized to zero, ensuring a known and consistent starting state for all pin operations. This initialization helps maintain system stability and predictability.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	DATA7	DATA6	DATA5	DATA4	DATA3	DATA	DATA	DATA
								rw	rw	rw	rw	rw	rw	rw	rw

Bit 31-8 : **RESERVED**

Bit 7-0 : **GPIO_DATA**

Figure 4.37: GPIO data

- **GPIO_DIR:** The GPIO_DIR register is utilized to configure each pin as either an output or an input. Setting a pin to 1 designates it as an output, allowing it to send signals, while setting it to 0 designates it as an input, allowing it to receive signals. By default, upon reset, all pins are configured as inputs, ensuring they are in a safe state until explicitly reconfigured by the user.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Dir
															rw

Bit 31-1 : **RESERVED**

Bit 0 : **GPIO_DATA**

0 -> Input mode

1 -> Output mode

Figure 4.38: GPIO directions

4.3.3 PS/2 keyboard controller

Module Description

The PS/2 keyboard controller module has the following inputs and outputs.

Inputs: The inputs include clk, reset, PS2_enable, which is the PS/2 module enable flag, PS2C, which is the PS2 keyboard clock signal, and PS2D, which is the bit read from the PS/2 peripheral.

Outputs: The outputs consist of Error, a flag indicating an error in the PS2 transmission, and Key, an 8-bit register used for reading the pressed key.

Internal Functionality: The module uses various internal registers to manage the PS2 communication: state, a 2-bit register that contains the current FSM state, data_read, an 11-bit register that stores the currently received data, counter, a 4-bit register used to track the received data bits, parity, which tracks the parity bit for error checking, previous_PS2C, which stores the previous state of the PS/2 clock signal, and ticks, which generates a down-clock signal.

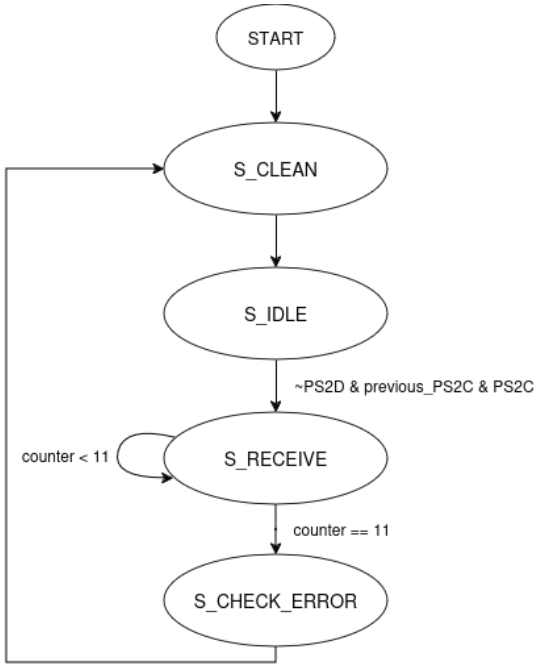

Figure 4.39: PS/2 Flowchart

In the S_IDLE state, it waits for a low transition of the PS2C to enter the S_RECEIVE state, where it will read 1 bit from the value of the PS2D on each falling edge of PS2C. After receiving 11 bits it transitions to the S_CHECK_ERROR state. In this state, the start and stop bits are verified, as long as the parity bit (the number of 1's including the parity bit must always be odd). If no errors are detected the key is updated with the received data stored in data_read, if errors are detected the error flag is set. In the end, it transition back to the S_CLEAN state to prepare for the next data reception.

PS/2 Registers

The PS/2 peripheral includes the following registers to configure its functionality:

- **PS2_CR - PS/2 Control Register**: This register is used to enable or disable the peripheral. This state can be read too.

PS2_CR: 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	**PS2_EN**
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**RW**

Bit 0 : PS2_EN
0 -> PS2 is disabled
1 -> PS2 is enabled

- **PS2_READ - PS/2 Received Data**: This register is used to read the PS/2 key pressed on the keyboard.

PS2_READ : 0x01

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	**PS2_KEY**							
-	-	-	-	-	-	-	-	R							

Bit 0-7 : PS2_KEY
PS/2 Keyboard Key scan code

All other values are **RFU**

4.3.4 Timer

The following block diagram illustrates the high-level architecture of the timer peripheral.

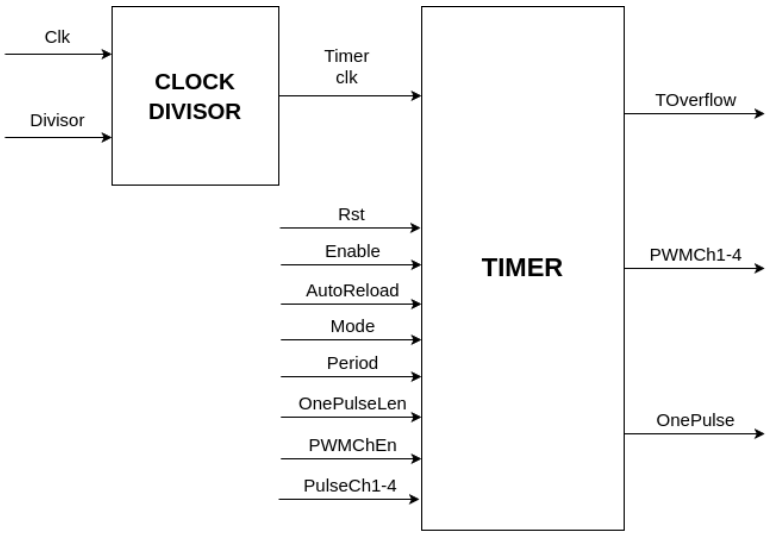

Figure 4.40: Timer Block Diagram

The Timer module consists of two primary components:

- **Clock Divisor:** allows for input clock division by factors of 1, 2, 4, and 8.
- **Timer:** the core component, processes the configuration register values and performs operations accordingly.

As illustrated in figure above, the Clock Divisor receives the input clock and the division factor to generate a divided clock signal (Timer clk). This signal is fed into the Timer, which operates based on various configuration inputs such as Reset, Enable, AutoReload, Mode, Period, OnePulse Length, PWM Channel Enable, and Pulse Channel 1-4. The Timer outputs include Timer Overflow, PWM Channels 1-4, and One Pulse.

Clock Divisor

The following block diagram depicts the clock divisor, which allows dividing the input clock frequency. The divisor value is programmed in a dedicated register. Here's a breakdown of the divisor settings:

- Divisor = 0: Actual divisor used = 1 (no division);

- Divisor = 1: Actual divisor used = 2 (divide by 2);
- Divisor = 2: Actual divisor used = 4 (divide by 4);
- Divisor = 3: Actual divisor used = 8 (divide by 8);

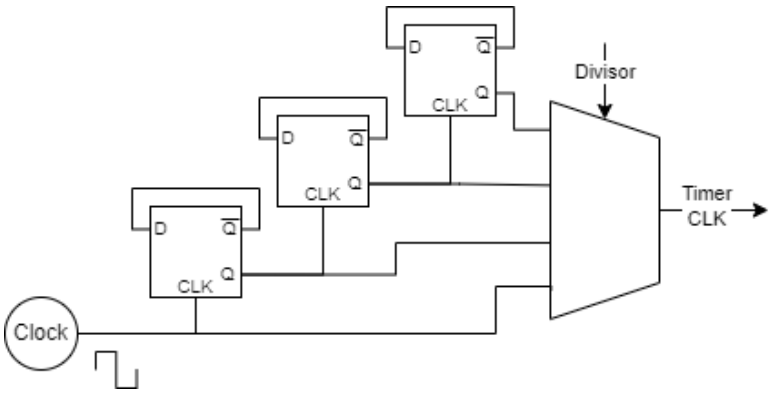

Figure 4.41: Clock Divisor Block Diagram

In total the Timer will have 4 operation modes:

Timer Mode:

In timer mode, the Timer increments a counter until it reaches the specified Period value. When this occurs, the Overflow flag is set. If the auto-reload flag is also set, the counter resets, and the process begins anew. This sequence is contingent on the Enable flag being set.

The overflow time can be calculated using the following equation:

$$Toverflow = \frac{Divisor}{Clk(Hz)} \times Period(s) \tag{4.1}$$

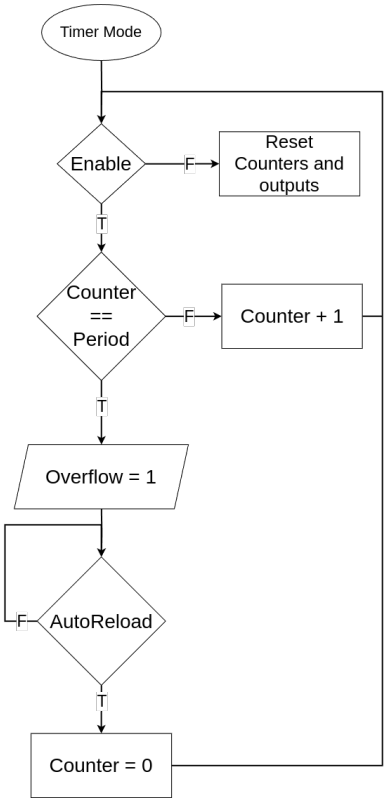

Figure 4.42: Timer Mode Flowchart

Counter Mode:

In counter mode, once enabled, the Timer increments a counter with each clock cycle. The counter value can be read as needed. When the Enable flag is reset, the counter is also reset.

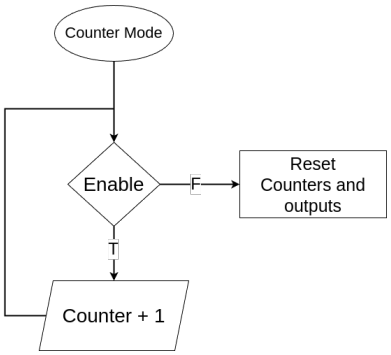

Figure 4.43: Counter Mode Flowchart

PWM Mode:

In PWM mode, when the Enable flag is set, the Timer increments the counter on each clock edge. If the counter reaches the Period value, it resets to 0. The PWM output channels are set to 1 if the counter is less than the respective PulseChannel values (and reset to 0 if otherwise), provided the corresponding channel is enabled. This process continues as long as the Enable flag remains set.

Given the duty-cycle value, the Pulse Width, can be obtained using the following expression:

$$PulseWidth = \frac{Period \times Duty - Cycle(0 - 100)}{100} \tag{4.2}$$

And the PWM frequency by the equation:

$$PWMfrequency = \frac{Clk(Hz)}{Divisor \times Period}(Hz) \tag{4.3}$$

Note: The counter in PWM mode is 8-bit, so the Period value can't exceed 0xFF.

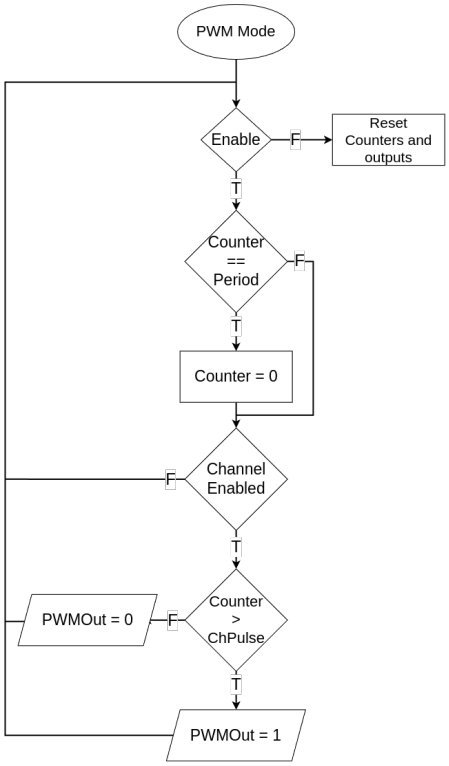

Figure 4.44: PWM Mode Flowchart

One Pulse Mode:

In One Pulse mode, the Timer increments a counter until it reaches the Period value. When this occurs, the One Pulse output is set, and the counter is reset. The counter then increments again until it equals the OnePulseLength value, at which point the One Pulse output is reset. This process will only repeat if the Enable flag is reset and then set again.

The time until pulse can be calculated using:

$$T_{overflow} = \frac{Divisor}{Clk(Hz)} \times Period(s) \tag{4.4}$$

And the pulse duration using the equation:

$$T_{overflow} = \frac{Divisor}{Clk(Hz)} \times OnePulseLen(s) \tag{4.5}$$

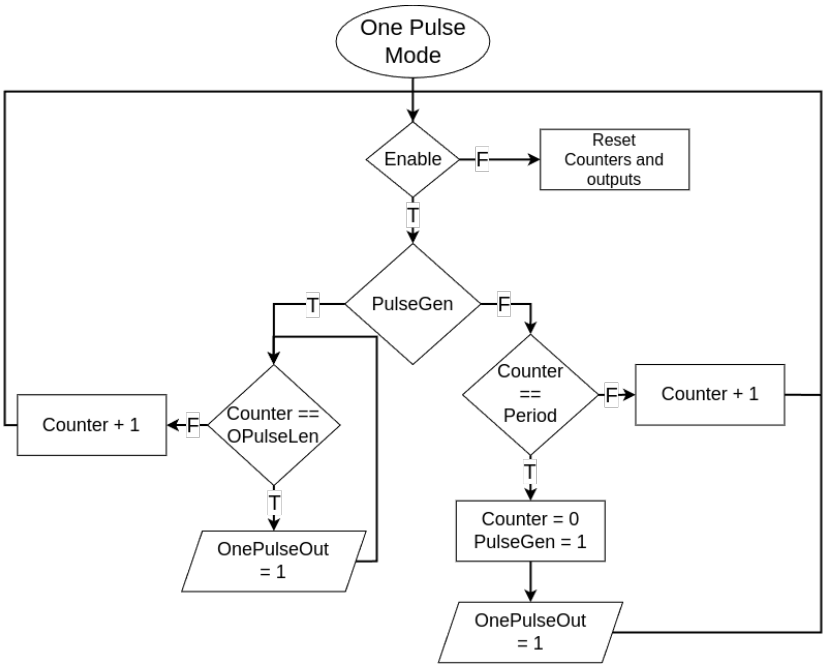

Figure 4.45: One Pulse Mode Flowchart

Timer Registers

The Timer peripheral includes the following registers to configure its functionality:

- **TCR - Timer Configuration Register**: This register is used to configure most of the timer functionalities, such as: divisor, mode, auto-reload and enable.

TCR : 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	TIMDIV		TIMMODE		TIMAR	TIMEN
-	-	-	-	-	-	-	-	-	-	RW		RW		RW	RW

Bit 0 : **TIMEN**
0 -> TIMER is disabled
1 -> TIMER is enabled

Bit 1 : **TIMAR**
0 -> Auto Reload is disabled
1 -> Auto Reload is enabled

Bit 2:3 : **TIMMODE**
00 -> Timer mode
01 -> Counter mode
10 -> PWM mode
11 -> One pulse mode

Bit 4:5 : **TIMDIV**
00 -> Timer frequency divided by 1 of clock frequency
01 -> Timer frequency divided by 2 of clock frequency
10 -> Timer frequency divided by 4 of clock frequency
11 -> Timer frequency divided by 8 of clock frequency

All other values are RFU

- **TPR - Timer Period**: This register is used to configure the timer Period value.

TPR : 0x01

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PERIOD															
RW															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PERIOD															
RW															

Bit 0-32 : **PERIOD**
Timer Period.

- **TOF - Timer Overflow Flag**: This register contains the overflow flag.

TOF : 0x02

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	**TIMOF**
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	RW

Bit 0 : **TIMOF**
0 -> TIMER Overflow is disabled
1 -> TIMER Overflow is enabled

- **TCE - Timer Channel Enable**: This register is used to enable the PWM channels.

TCE : 0x03

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	PWMCHEN			
-	-	-	-	-	-	-	-	-	-	-	-	RW			

Bit 0-3 : **PWMCHEN**
Each bit represents a PWM channel output state, set to 1/0 to enable/disable PWN channel output

All other values are RFU

- **PCP1/PCP2 - PWM Channel Pulse**: These registers are used to configure the PWM channels pulse, 1 through 4.

PCP1 : 0x04

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH2P								CH1P							
RW								RW							

Bit 0-7 : **CH1P**
PWM channel 1 pulse

Bit 8-15 : **CH2P**
PWM channel 2 pulse

All other values are RFU

PCP2 : 0x05

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH4P								CH3P							
RW								RW							

Bit 0-7 : **CH3P**
PWM channel 3 pulse

Bit 8-15 : **CH4P**
PWM channel 4 pulse

All other values are RFU

- **OPL - One Pulse Length**: This register is used to configure the One Pulse Length.

OPL : 0x06

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPL															
RW															

Bit 0-15 : **OPL**
One Pulse Length

All other values are RFU

- **CV - Counter Value**: This register is used to read the Timer Counter value.

CV : 0x1C

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CV															
R															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CV															
R															

Bit 0-31 : **CV**
Counter Value

4.3.5 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 4.46 presented below represents the connections between these referred modules.

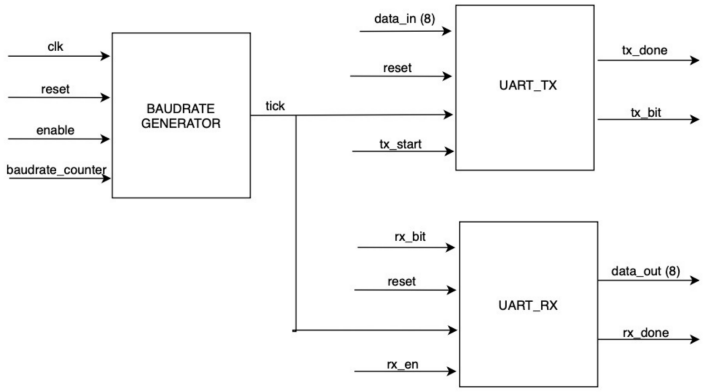

Figure 4.46: UART Connection Diagram

Baudrate Generator

The baudrate generator is responsible to generate the correct clock source for Tx module and Rx module, accordingly desired baudrate.

As can be seen in figure 4.47, the baudrate generator module receives as input a value that defines the baudrate (baudrate_counter). This value can be calculated with the following expression:

$$baudrate\_counter = \frac{(clk -> 125MHz)}{(desired\_baudrate)} \tag{4.6}$$

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

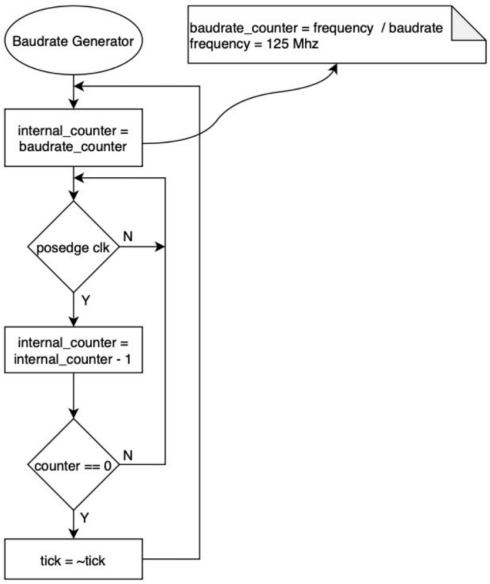

Figure 4.47: Baudrate Generator Design

Transmission module

The UART transmission module can be represented by the state machine in figure4.48. 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 internal buffer. 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 clocktick. 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.

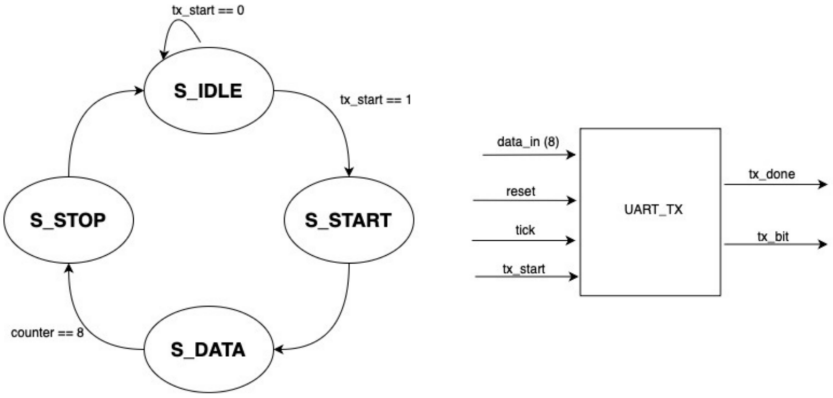

Figure 4.48: UART Tx state machine

Reception module

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

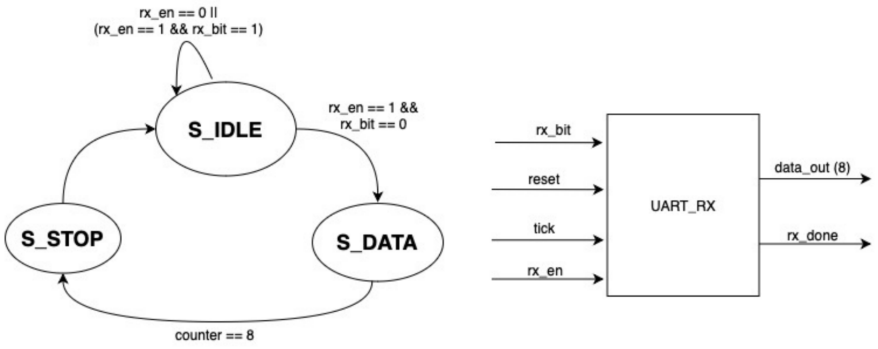

Figure 4.49: UART Rx state machine

Registers description

For the UART Registers, 4 registers of 32 bits were defined. The registers created were the UART_CR (control register), the UART_BRR (baudrate register), the UART_RXBUFF and UART_TXBUFF, which were mapped in 4 addresses.

Below, detailed figures of each register to provide a more comprehensive understanding of their structures.

- **UART_CR:** This register is used to control the peripheral by reading the "RxDONE" and "TxDONE" flags that signalize when the data as been receive and transmitted, respectively. Also is possible to read/write from the "TxEN" which enables the Tx module, "RxEN" which enables the Rx module and the "UARTEN" that enables the UART peripheral.

UART_CR : 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RXDONE	TXDONE	TXEN	RXEN	UARTEN
-	-	-	-	-	-	-	-	-	-	-	R	R	RW	RW	RW

Bit 0 : **UARTEN**
0 -> UART is disabled
1 -> UART is enabled

Bit 1 : **RXEN**
0 -> Receiver is disabled
1 -> Receiver is enabled

Bit 2 : **TXEN**
0 -> Transmitter is disabled
1 -> Transmitter is enabled

Bit 3 : **TXDONE**
0 -> UART Transmission idle
1 -> UART Transmission was completed. This flag is cleared by hardware upon a successful transmission

Bit 4 : **RXDONE**
0 -> Disables the UART Peripheral
1 -> Enable the UART Peripheral

All other values are RFU

Figure 4.50: UART Control Register

- **UART_BRR:** This register is used to change or read the UART's baudrate value.

UART_BRR : 0x01

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BRR															
RW															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRR															
RW															

Bit 0-31 : BRR
Baudrate Value

Figure 4.51: UART Baudrate Register

- **UART_RXBUFF:** This register is used to store an 8-bit buffer containing the data received by the UART peripheral. The register access is bi-directional.

UART_RXBUFF : 0x02

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	**RX_BUFF**							
-	-	-	-	-	-	-	-	RW							

Bit 0-7 : RX_BUFF
Data received by the UART peripheral

All other values are **RFU**

Figure 4.52: UART RxBuff Register

- **UART_TXBUFF:** This register is used to store an 8-bit buffer containing the data to be transmited by the UART peripheral. The register access is also bi-directional.

UART_TXBUFF : 0x03

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	RFU	RFU	RFU	RFU	RFU	RFU	RFU	**TX_BUFF**							
-	-	-	-	-	-	-	-	**RW**							

Bit 0-7 : TX_BUFF
Data to be transmitted by the UART peripheral

All other values are **RFU**

Figure 4.53: UART TxBuff Register

4.3.6 VGA

Timing Specification for 640x480(60Hz)

To create the pixel clock, a clock divider is necessary, offering timing guidance to HS and VS signals. In the 640x480 specification, the pixel clock runs at 25.175MHz, though 25MHz with an accuracy of ±0.5% is also suitable. This frequency is easily attainable on an FPGA board.

Signal and Frame generation

Two counters are needed: a horizontal counter to count pixels per line and a vertical counter to count lines in a frame. The horizontal counter resets when reaching the end of a line (799 in this case) and provides a Terminal Count signal to the Enable input of the vertical counter to increment when a new line begins. Similarly, the vertical counter resets at the end of a frame.

Comparing the counter values to constants defined in the specification generates HS and VS signals. Red, Green, and Blue signals are driven outside the display area. Refer to Figure 4.54 for HS and VS generation based on counter values.

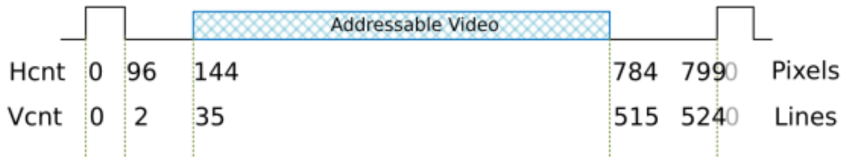

Figure 4.54: Horizontal and Vertical counters

Description	Time	Width/Frequency
Pixel clock	39.7 ns	25.175 MHz
H Sync Time	3.813 us	96 Pixels
H Back Porch	1.907 us	48 Pixels
H Front Porch	0.636 us	16 Pixels
H Addr Video Time	25.422 us	640 Pixels
H Left/Right Time	-	-
V Sync Time	0.064 ms	2 Lines
V Back Porch	1.048 ms	33 Lines
V Front Porch	0.318 ms	10 Lines
V Addr Video Time	15.253 ms	480 Lines
V Top/Bottom Border	-	-

Table 4.2: Timing and Dimension Specifications

Frame Generation FSM

Based on the detailed information above about how the VGA peripheral generates images and handles signals, the following FSM was developed to manage this process.

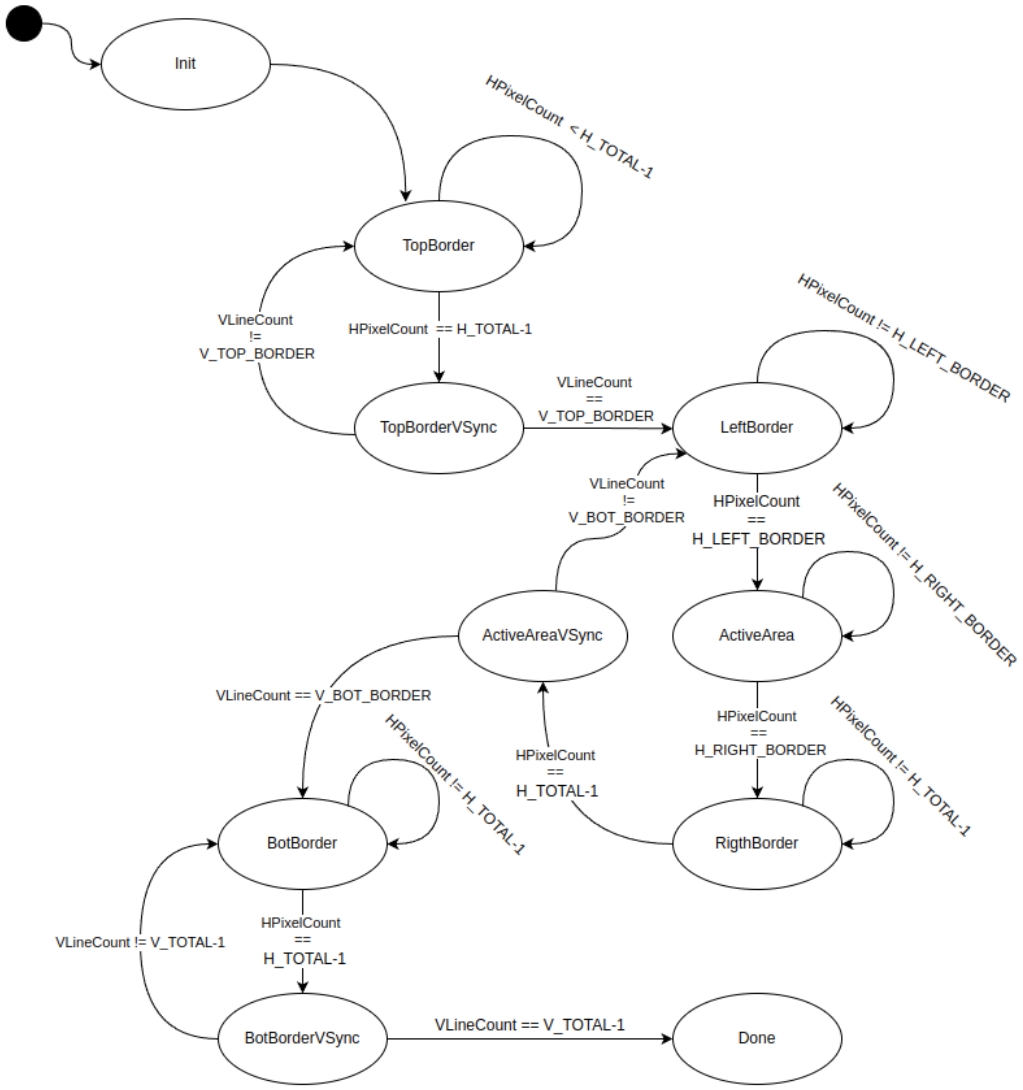

Figure 4.55: Frame Generation FSM

Note: The Zybo lacks the resources to integrate the VGA peripheral with the rest of the SoC. It cannot handle the memory required for a single frame at 640x480 resolution. Therefore, the stored frame size will be reduced to 320x240, half of the desired resolution. This reduction will require additional logic to extend the memory values to cover 2 pixels horizontally and 2 pixels vertically.

Chapter 5

Implementation

5.1 CPU

5.1.1 Control Unit

As said in the design phase, the control unit is implemented through a state machine, represented in figure 5.1.

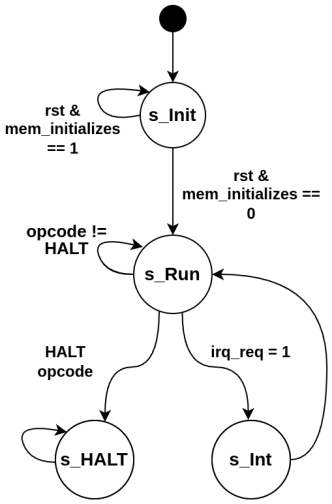

Figure 5.1: Control unit state machine

This state machine has been transcribed into Verilog, as can be seen in the code below. In this code, you can check the different state changes according to the respective conditions.

```
...
always @(posedge i_Clk) begin
    if (i_Rst) begin
        r_CurrentState <= ST_INIT;
    end
    else begin
        case (r_CurrentState)
            ST_INIT: begin
                if (w_PeripheralsRdy == 1'b1) begin
                    r_CurrentState <= ST_RUN;
                end
                else begin
                    r_CurrentState <= ST_INIT;
```

```
14                  end
15              end
16
17              ST_RUN: begin
18                  if(i_IntRequest) begin
19                      r_CurrentState <= ST_INT;
20                  end
21                  else begin
22                      r_CurrentState <= ST_RUN;
23                  end
24              end
25
26              ST_INT: begin
27                  r_CurrentState <= ST_RUN;
28              end
29
30              ST_HALT: begin
31                 r_CurrentState <= ST_HALT;
32              end
33
34              default:
35                  r_CurrentState <= ST_HALT;
36          endcase
37      end
38  end
39  ...
```

Listing 5.1: FSM of Control Unit

The control unit is also responsible for deciding whether a jump is taken or not according to the ALU flags and the jump condition. Therefore a module has been instantiated which is responsible for carrying out this check.

5.1.2 Pipelined Datapath

The design chapter, presented the foundation for the pipelined architecture of the CPU, emphasizing the need for a structured approach to handle multiple instruction stages concurrently. This section explores the specifics of each pipeline stage, detailing the Verilog code that implements the logic and interactions within these stages.

Fetch Stage

The code in the listing below shows how the program counter is updated based on the current control signals. If a stall signal is asserted, the program counter and control signals remain unchanged. Otherwise, the program counter is updated based

on the selected control path, which can include branch, jump, return, interrupt, or default increment operations.

```
//(...)

if (i_Stall) begin
    o_ProgramCounter <= o_ProgramCounter;
end
else begin
    case (i_PcSel)
        `PC_SEL_BXX: begin
            o_ProgramCounter<= i_PcBxx;
        end

        `PC_SEL_JMP: begin
            o_ProgramCounter<= i_PcJmp;
        end

        `PC_SEL_RET: begin
            o_ProgramCounter<= i_PcRet;
        end

        `PC_SEL_RETI: begin
            o_ProgramCounter<= i_PcReti;
        end

        `PC_SEL_INT: begin
            o_ProgramCounter<= i_PcInt;
        end

        default: begin
            o_ProgramCounter <= o_ProgramCounter + `PC_INC;
        end
    endcase
end

//(...)
```

Listing 5.2: Instruction fetch stage implementation

Decode Stage

The decode stage decodes the instruction to extract the opcode, operand addresses, and immediate values.

The code on the listing displayed below shows how these fields are extracted from the instruction register following the VeSPA ISA and how the read addresses for the RF are selected based on control signals.

```
...
assign o_IrRs1 = w_IrRs1;
assign o_IrRs2 = w_IrRs2_in;
assign o_IrRst = w_IrRst;

//Decode IR Register
assign w_IrRs1 = i_InstructionRegister[21:17];
assign w_IrRs2 = i_InstructionRegister[15:11];
assign w_IrRst = i_InstructionRegister[26:22];

assign w_Imm16 = i_InstructionRegister[15:0];
assign w_Imm17 = i_InstructionRegister[16:0];
assign w_Imm22 = i_InstructionRegister[21:0];
assign w_Imm23 = i_InstructionRegister[22:0];

assign o_BranchCond = i_InstructionRegister[26:23];
assign o_ImmOp = i_InstructionRegister[16];
assign o_Opcode = i_InstructionRegister[31:27];

assign w_IrRs2_in = (i_Read2AddrSel == 2'b00) ? w_IrRs2 :
                    (i_Read2AddrSel == 2'b01) ? w_IrRst :
                    `RET_REG;

...
```

Listing 5.3: Instruction Decode stage implementation

Execute Stage

The Instruction Execute module performs arithmetic and logic operations based on the decoded instruction. It's also responsible for calculating the JMP/BXX address.

The code showcased in the listing below details the selection of operands for the ALU and the computation of jump and branch targets. This stage also handles forwarding of data to resolve hazards.

```
...
assign o_PcSelExe2Fetch =
    (i_BranchVerification == 1'b0 && i_BranchBit == 1'b1) ?
    `PC_SEL_ADD4 : i_PcSel;

assign o_PcJmp = i_Imm16 + w_AluOp1;

assign o_PcBranch = i_ProgramCounter + i_Imm23;

assign o_ImmOpX = w_AluOp1 + i_Imm17;

assign w_AluOp1 = (i_ForwardOp1 == 2'b00) ? i_R1Out     :
```

```
                    (i_ForwardOp1 == 2'b01) ? i_AluOutMem :
                    (i_ForwardOp1 == 2'b11) ? i_Immed22Mem:
                    i_RfOutValue;

assign o_AluOp2 = (i_ForwardOp2 == 2'b00) ? i_R2Out     :
                    (i_ForwardOp2 == 2'b01) ? i_AluOutMem :
                    (i_ForwardOp2 == 2'b11) ? i_Immed22Mem:
                    i_RfOutValue;

assign w_AluIn2 = (i_AluOp2Sel == 1'b0) ? o_AluOp2 : i_Imm16;
...
```

Listing 5.4: Instruction Execute stage implementation

Memory Stage

The memory module is responsible for accessing data memory.

The code in the listing below shows how the memory address is selected based on the control signal (immediate access or indexed access).

```
...
assign o_dataMemAddress = (i_MemAddrSel == 1'b0) ? i_Imm22 :
                          i_ImmOpX;
```

Listing 5.5: Memory stage implementation

Write Back Stage

The Write Back module writes the results from the execute or memory stage back to the RF.

The code in the listing below demonstrates how control signals select the data to be written back, determining whether it comes from the ALU result, data memory output, immediate values, or the PC.

```
...
assign o_RfData = (i_RfDataInSel == 2'b00) ? i_DataMem :
                  (i_RfDataInSel == 2'b01) ? i_ProgramCounter :
                  (i_RfDataInSel == 2'b10) ? i_AluOut : i_Imm22;
...
```

Listing 5.6: Write Back stage implementation

5.1.3 Intermediate Registers

Since all the Intermediate Registers are very similar in the way they were implemented, listing 5.7 demonstrates the way these registers were implemented.

```
module Intermidiate_Register_Example
(
    input Clock,
    input Reset,

    input Example_In,
    input Flush,
    input Stall,

    output Example_Out
);

    always @(posedge Clock) begin
        if(Reset) begin
            Example_Out <= 0;
        end
        else begin
            if(Flush) begin
                Example_Out <= 0;
            end
            else if(Stall) begin
                Example_Out <= Example_Out;
            end
            else begin
                Example_Out <= Example_In;
            end
        end
    end
endmodule
```

Listing 5.7: Intermediate Register general implementation

5.1.4 Hazard Unit

The Hazard Unit, as mentioned previously in the design chapter, is critical for managing both control and data hazards. The inputs to the Hazard Unit include various signals such as register addresses from the decode and execute stages, branch verification signals, jump and return-from-interrupt bits, memory read and write enables, ALU enable signals, and an interrupt signal. These inputs allow the Hazard Unit to detect potential hazards that might occur due to dependencies between instructions. The outputs of the Hazard Unit include signals to flush different stages of the pipeline (fetch, decode, execute, memory) and a stall signal.

Within the Hazard Unit, two critical modules are instantiated: the Control Hazards Unit and the Forwarding Unit. These modules work together to manage the various types of hazards that can occur in a pipelined architecture.

Control Hazards Unit

The Control Hazards Unit is responsible for detecting and managing control hazards, which occur due to branches, jumps, and interrupts. This unit monitors inputs such as branch verification signals, jump bits, and interrupt signals to determine when a pipeline flush is necessary. When a branch is taken or a jump occurs, the Control Hazards Unit generates flush signals for the fetch, decode, execute, and memory stages, ensuring that any instructions fetched or decoded after the branch or jump do not proceed. Additionally, the unit handles interrupts by flushing the pipeline and ensuring that the processor correctly handles the interrupt routine.

```
...
always @(posedge i_Clk) begin
    if(i_Rst == 1'b1) begin
        o_FlushDecode <= 0;
        o_FlushExecute <= 0;
        o_FlushFetch <= 0;
    end
    else begin
        if((i_BranchVerification == 1'b1 && i_BranchBit == 1'b1)
            || i_JmpBit == 1'b1
            || i_RetiBit == 1'b1
            || i_InterruptSignal == 1'b1
           )
        begin
            o_FlushDecode <= 1;
            o_FlushExecute <= 1;
            o_FlushFetch <= 1;
        end
        else begin
            o_FlushDecode <= 0;
            o_FlushExecute <= 0;
            o_FlushFetch <= 0;
        end
    end
end

assign o_StallSignal =
    ((i_RdMemExe && i_AluEnDec) && (i_RfReadDstExec ==
    i_IrRead1AddrDec)) ? 1'b1 :

    ((i_RdMemExe && i_AluEnDec) && (i_RfReadDstExec ==
```

```
32      i_IrRead2AddrDec)) ? 1'b1 :
33
34      ((i_RdMemExe && i_WeMemEnable) && (i_RfReadDstExec ==
35      i_IrRead2AddrDec)) ? 1'b1 : 1'b0;
36  endmodule
```

Listing 5.8: Implementation of control hazard unit

Key Responsibilities

- As explained before, the flush is utilized when a jump occurs, a branch is verified and taken, a return from interrupt (RETI instruction) occurs, or when an interruption occurs.
- The stall signal is enabled when there is a memory read instruction and an ALU instruction in the decode stage, and the destination register of the memory read in the execution stage matches either of the read addresses in the decode stage. Additionally, the stall signal is enabled if there is a memory read instruction and a memory write instruction, and the destination register of the memory read in the execution stage matches the read address in the decode stage.

Forwarding Unit

The Forwarding Unit handles data hazards, which occur when instructions depend on the results of previous instructions that have not yet completed their execution.

The forward multiplexer can be seen in the Figure 6.2.

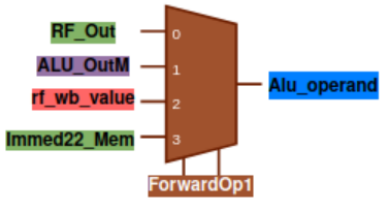

Figure 5.2: Forward Unit multiplexer

This unit mitigates such hazards by forwarding data from later stages of the pipeline back to the earlier stages where they are needed. For instance, if an instruction in the execute stage needs the result of an ALU operation or an immediate value, currently in the memory stage, the forwarding unit provides the necessary data to the execute stage, avoiding the insertion of stalls until the instruction has the necessary data.

```
...
assign o_ForwardOp1 =
    //alu result in memory stage
    ((i_IrRead1AddrDecodeExec == i_RfReadDstMemory) &&
    i_RfWeMemory && (i_RfDataInSelMem == 2'b10)) ? 2'b01 :

    //immediate value (ldi) in memory stage
    ((i_IrRead1AddrDecodeExec == i_RfReadDstMemory) &&
    i_RfWeMemory && (i_RfDataInSelMem == 2'b11)) ? 2'b11 :

    //alu result or immediate value in write-back stage
    ((i_IrRead1AddrDecodeExec == i_RfReadDstWriteBack) &&
    i_RfWeWriteBack) ? 2'b10 : 2'b00;
...
```

Listing 5.9: Implementation of forward unit

What is demonstrated in the listing above, is that the forwarded operand 1 can be an ALU output forwarded from the Memory stage, or an immediate value from a LDI instruction forwarded from the Memory stage, or either an ALU output or LDI value forwarded from the WB stage. In default, the output value of RF is used. The same logic applies to forward operand 2.

5.1.5 Barrel Shifter

The Barrel Shifter takes a 32-bit input data (‘i_Data‘), a 5-bit shift amount (‘i_Shift‘), and a shift direction (‘i_Direction‘), where ‘0‘ indicates a right shift and ‘1‘ indicates a left shift. The shifting process is handled in stages, each stage performing shifts by 1, 2, 4, 8, and 16 bits respectively.

The ‘w_ShiftControl‘ signal determines the actual shift amount by considering the direction of the shift. Intermediate wires (‘w_Stage1‘, ‘w_Stage2‘, ‘w_Stage3‘, and ‘w_Stage4‘) store the results of each stage. The final shifted result is assigned to ‘o_ShiftResult‘.

The Verilog implementation of the barrel shifter its presented in the following code listing:

```
module Barrel_Shifter(
    input [31:0] i_Data,            // input data
    input [4:0] i_Shift,            // shift amount
    input i_Direction,              // shift direction
                                    //0 - right or 1 - left
    output [31:0] o_ShiftResult     // result
);

wire [4:0] w_ShiftControl;
```

```
10 wire [32:0] w_Stage1, w_Stage2, w_Stage3, w_Stage4;
11
12 assign w_ShiftControl = i_Direction ? ~i_Shift + 1
13                                      : i_Shift;
14 // 1 bit shift
15 assign w_Stage1=w_ShiftControl[0]?{i_Data[0], i_Data[31:1]}
16                                      : i_Data;
17 // 2 bit shift
18 assign w_Stage2=w_ShiftControl[1]?{w_Stage1[1:0], w_Stage1[31:2]}
19                                      : w_Stage1;
20 // 4 bit shift
21 assign w_Stage3=w_ShiftControl[2]?{w_Stage2[3:0], w_Stage2[31:4]}
22                                      : w_Stage2;
23 // 8 bit shift
24 assign w_Stage4=w_ShiftControl[3]?{w_Stage3[7:0], w_Stage3[31:8]}
25                                      : w_Stage3;
26 // 16 bit shift
27 assign o_ShiftResult=w_ShiftControl[4]?{w_Stage4[15:0],
28                                        w_Stage4[31:16]}
29                                        : w_Stage4;
30
31 endmodule
```

CPU Modifications

In order to successfully integrate the barrel shifter in VeSPA, some alterations must be done for the correct behaviour of these new shift instructions. Such modifications happen in the Control Unit, the ALU and in the opcode list.

The following new opcode values have been defined for the right and left shifts:

```
1 `define OP_RR 5'd15
2 `define OP_RL 5'd16
```

Control Unit

The Control Unit has been updated to include new conditions for several signals based on the new opcodes:

The ‘o_WrEnRf‘ signal has the following conditions added to its assignment statement.

```
1 assign o_WrEnRf = <...>
2                   (i_OpCode == `OP_RR || i_OpCode == `OP_RL ||
3                   (i_OpCode == `OP_JMP && i_ImmOp == 1'b1))? 1'b1
4                                                            : 1'b0;
```

The ‘o_AluEn‘ signal has the following conditions added to its assignment statement.

```
assign o_AluEn = <...>
                 (i_OpCode == `OP_CMP || i_OpCode == `OP_RR
                 || i_OpCode == `OP_RL) ? 1'b1 : 1'b0;
```

The ‘o_AluCtrl‘ signal has the following conditions added to its assignment statement.

```
assign o_AluCtrl = <...>
                   (i_OpCode == `OP_RR) ? 5'b01111 :
                   (i_OpCode == `OP_RL) ? 5'b10000 : 5'b00000;
```

ALU

The ALU has been updated with new conditions for the ‘w_Output‘ signal, a new assignment statement and the ’Barrel_Shifter’ module instantiation:

The ‘w_Output‘ signal has the following condition added to its assignment statement:

```
assign w_Output = <...>
                  (i_OpCtrl == `OP_RR || i_OpCtrl == `OP_RL)
                                      ? w_ShiftResult : 0;
```

A new assignment statement for the shift direction has been created:

```
assign w_ShiftDirection = (i_OpCtrl == `OP_RL) ? 1'b1 : 1'b0;
```

The following wires and module instantiation have been added to support the new functionalities inside the ALU module:

```
wire w_ShiftDirection;
wire [31:0] w_ShiftResult;

Barrel_Shifter shifter(
    .i_Data(i_LeftOp),
    .i_Shift(i_RigthOp),
    .i_Direction(w_ShiftDirection),
    .o_ShiftResult(w_ShiftResult)
);
```

5.2 Bus

The Interconnect Bus was implemented as follows in the next listing and basically it diverts the data according to the address indicated by the CPU.

```
module CustomInterconnect(
    input i_Clk,
    input i_Rst,

    //CPU connection
    input i_WEnable,
    input [`BUS_WIDTH-1:0]i_WAddr,
    input [`BUS_WIDTH-1:0]i_WData,
    input i_REnable,
    input [`BUS_WIDTH-1:0]i_RAddr,
    output [`BUS_WIDTH-1:0]o_RData,

    //peripheral 0 connection
    output o_WEnable_0,
    output [`BUS_WIDTH-1:0]o_WAddr_0,
    output [`BUS_WIDTH-1:0]o_WData_0,
    output o_REnable_0,
    output [`BUS_WIDTH-1:0]o_RAddr_0,
    input [`BUS_WIDTH-1:0]i_RData_0,
    //(..... Equal to the other peripherals .....)
    );

    //slave 0 interface -> data memory 1 kByte
    assign o_WEnable_0=(i_WAddr[10] == 1'b0) ? i_WEnable : 1'bZ;
    assign o_WAddr_0  =(i_WAddr[10] == 1'b0) ? i_WAddr   : 32'bZ;
    assign o_WData_0  =(i_WAddr[10] == 1'b0) ? i_WData   : 32'bZ;
    assign o_REnable_0=(i_RAddr[10] == 1'b0) ? i_REnable : 1'bZ;
    assign o_RAddr_0  =(i_RAddr[10] == 1'b0) ? i_RAddr   : 32'bZ;
    //(... Same logic for the peripherals ...)

    //Read data
    assign o_RData =
          (r_RAddr[10]==1'b0) ? i_RData_0 :
          (r_RAddr[10]==1'b1 && r_RAddr[4:2]==3'b000) ? i_RData_1:
          (r_RAddr[10]==1'b1 && r_RAddr[4:2]==3'b001) ? i_RData_2:
          (r_RAddr[10]==1'b1 && r_RAddr[4:2]==3'b010) ? i_RData_3:
          (r_RAddr[10]==1'b1 && r_RAddr[4:2]==3'b011) ? i_RData_4:
          (r_RAddr[10]==1'b1 && r_RAddr[4:2]==3'b100) ? i_RData_5:
          (r_RAddr[10]==1'b1 && r_RAddr[4:2]==3'b101) ? i_RData_6:
          (r_RAddr[10]==1'b1 &&(r_RAddr[4:2]==3'b110) ? i_RData_7:
          32'bZ;
```

Listing 5.10: Interconnect Bus Implementation

Peripheral Interconnect Wrapper

The code below shows the implementation of a generic wrapper used in one of the slaves. If the CPU wants to write in order to configure the peripheral, then its registers will be changed.

```
1
2  reg [31:0] slave_reg1;
3  reg [31:0] slave_reg2;
4  ...
5
6  if (i_WEnable) begin
7      case (i_WAddr)
8          2'b00:
9              slave_reg1 <= cpu_dataOut;
10
11         2'b01:
12             slave_reg2 <= cpu_dataOut;
13
14         default:
15             invalidAddr <= 1;
16     endcase
17 end
18 else if (i_REnable) begin
19      case (i_RAddr)
20         2'b00:
21             cpu_dataIn <= slave_reg1;
22
23         2'b01:
24             cpu_dataIn <= slave_reg2;
25
26         default:
27             invalidAddr <= 1;
28     endcase
29 end
```

Listing 5.11: Slave wrapper implementation

5.3 Peripherals

5.3.1 Interrupt Controller

The interrupt controller implemented, comprises of two main modules: the 'interruptController_wrapper' and the 'interruptController'. The wrapper module interfaces with the custom bus and manages the RF, while the core module handles interrupt prioritization and acknowledgment.

As stated in the Design chapter the interrupt controller receives inputs from four interrupt sources ('int_source0' to 'int_source3') and processes these through a priority encoder to determine which interrupt should be handled first.

The code in the listing below, demonstrates how the interrupt sources are enabled. As can be seen, each int_sources bit is set only if the corresponding interrupt source is active and enabled.

```
...
wire [3:0] int_sources_2;
reg [3:0] pending;
reg [3:0] int_sources_prev;

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;
...
```

Listing 5.12: Interrupt Source Activation

To demonstrate how the interrupt priority logic is managed the code below is presented. The pending register tracks active interrupt requests. As can be seen, the value of nextIrq is assigned based on the status of the pending register. The highest-priority pending interrupt is selected, with pending[0] having the highest priority and pending[3] the lowest.

```
...
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;
...
```

Listing 5.13: Interrupt Priority Determination

To understand the behaviour of the interrupt controller and how it manages interrupts, interrupt priorities and informs the CPU of additional pending interrupts, a brief code explanation is presented, followed by the code it self.

First when a reset occurs all the signals are reset to their initial states. When not in a reset condition, three possible situations can occur:

- **An interrupt has been completed:** Meaning the signal int_ack_complete is asserted. Then, there are two possible outcomes:
  - If there is no following interrupt requests, this means that nextIrq[2] is asserted, so currIrq is changed to its default value.
  - If there is a following interrupt to be attended, this means that nextIrq[2] is de-asserted, so to indicate there is an interrupt request int_req is set to 1. Then to determine the priority of the interrupt nextIrq[1:0] is passed on to int_number.
- **An interrupt has been attended:** The signal int_ack_attended is asserted, indicating that an interrupt has been attended and is now being serviced. So int_req is set to 0, current interrupt request (currIrq) takes the value of next interrupt to be serviced (nextIrq), and int_pending is updated to indicate if there are still pending interrupts.
- **A new interrupt request needs to be initiated:** Meaning the value of currIrq is set to default, and the value of nextIrq is not. Therefore int_req is asserted and int_number takes the value of nextIrq[1:0].

```
...
always @(posedge clk) begin
    if(rst == 1'b1) begin
        int_req <= 1'b0;
        int_number <= 2'b0;
        currIrq <= 3'b100;
        int_pending <= 1'b0;
    end
    else begin
        if(int_ack_complete) begin
            if(!nextIrq[2]) begin
                int_req <= 1'b1;
                int_number <= nextIrq[1:0];
            end
            else begin
                currIrq <= 3'b100;
            end
        end
        else if(int_ack_attended) begin
            int_req <= 1'b0;
            currIrq <= nextIrq;
            int_pending<=((pending & ~(4'b0001 << nextIrq))!= 0);
```

```
23          end
24          else if(currIrq[2] && !nextIrq[2]) begin
25              int_req <= 1'b1;
26              int_number <= nextIrq[1:0];
27          end
28          else begin
29              //Do nothing
30          end
31      end
32  ...
```

Listing 5.14: Interrupt Controller

Below follows an explanation on how the interrupt controller manages the state of servicing interrupts using the signal int_attending, followed by the code implementation.

When a reset occurs the signal int_attending is set to its initial state which is zero. When not in a reset condition, three possible situations can occur:

- An interrupt is attended and there are no pending interrupts. This means the signal int_ack_atended is asserted and the signal int_pending is de-asserted. When this occurs, it means an interrupt service routine will be executed, so the signal int_attending is asserted to indicate this.
- An interrupt service routine has been completed and there are no pending interrupts. This means the signal int_ack_complete is asserted and the signal int_pending is de-asserted. When this occurs, it means there are no interrupt service routines being executed, so the signal int_attending is de-asserted.
- None of the above conditions is true. In this situation, the signal maintains its current state.

```
1   ...
2   always @(posedge clk) begin
3       if(rst == 1'b1) begin
4           int_attending <= 0;
5       end
6       else if(int_ack_attended && !int_pending) begin
7           int_attending <= 1;
8       end
9       else if(int_ack_complete && !int_pending) begin
10          int_attending <= 0;
11      end
12  end
13  ...
```

Listing 5.15: Interrupt Controller

5.3.2 GPIO

The GPIO module essentially implements a three-state buffer controlled by the direction variable. If the direction is set to 0, the buffer output will be in a high-impedance state. In this case, if there is a signal on the pins, it will drive the data output of the peripheral. If the direction is set to 1, the signal present at data_in will be outputted through the pins.

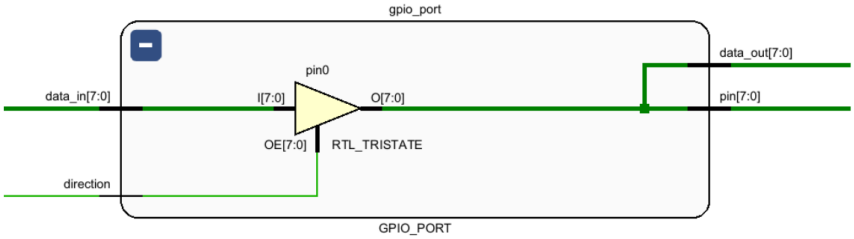

Figure 5.3: GPIO Port

To implement the GPIO Port the following code was developed:

```
module GPIO_PORT(
    input [7:0] direction,

    input [7:0] data_in,
    output [7:0] data_out,

    inout [7:0] pin
);

    generate
    genvar i;
    for (i = 0; i < 8; i = i + 1) begin
        assign pin[i] = direction[i] ? data_in[i] : 1'bZ;
    end
    endgenerate

    assign data_out = pin;
endmodule
```

Listing 5.16: GPIO Port Implementation

5.3.3 PS/2 keyboard controller

The PS/2 keyboard controller module in Verilog has specific inputs and outputs essential for its operation. The inputs include a system clock signal (clk), an

active-low reset signal (reset), a PS/2 module enable flag (PS2_enable), a PS/2 keyboard data signal (PS2D), and a PS/2 keyboard clock signal (PS2C).

On the output side, the module features an active-high signal (error) that indicates an error in the PS/2 transmission. Additionally, it includes an 8-bit register (key) that stores the received data corresponding to the key code. This description can be visualized in figure 5.4.

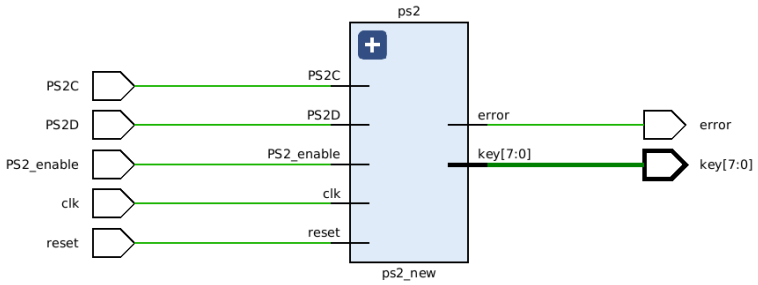

Figure 5.4: PS2 Schematic

The PS/2 implementation comprises of two main modules: the custom_ps2_interface which serves as a wrapper and the actual PS/2 keyboard controller. The wrapper module handles the custom bus interfacing and configuration, while the controller module manages the PS/2 protocol for reading data from a PS/2 keyboard.

Bellow it is listed the PS/2 keyboard controller implementation:

```
(....)
always @(posedge clk) begin
    if (reset) begin
        (...)
    end
    else begin
        ticks <= ticks + 1'b1;
        if(PS2_enable == 1'b1 && ticks >= 12'd4000) begin
            ticks <= 1'b0;
            case (state)
                S_CLEAN: begin
                    data_read <= 11'b0;
                    counter <= 4'b0;
                    previous_PS2C <= 1'b1;
                    parity <= 1'b1;
                    state <= S_IDLE;
                    ticks <= 12'd4000;
                end

                S_IDLE: begin
                     if ((~PS2C) && previous_PS2C) begin
                        error <= 1'b0;
```

```
                        state <= S_RECEIVE;
                        ticks <= 12'd4000;
                    end
                end

                S_RECEIVE: begin
                    // falling edge on PS2C -> PS2D can be read
                    if ((~PS2C) && previous_PS2C) begin
                        //concatenate PS2D in the LSB of data_read
                        data_read <= {data_read[9:0], PS2D};
                        counter <= counter + 4'b1;
                    end
                    if (counter > 4'd1 && counter < 4'd10) begin
                        parity <= parity ^ PS2D;
                    end
                    // all 11 bits were read
                    else if (counter == 4'd11) begin
                        state <= S_CHECK_ERROR;
                    end
                    previous_PS2C <= PS2C;
                end

                S_CHECK_ERROR: begin
                    // verify start and stop
                    if ((data_read[10] != 1'b0)
                        || (data_read[0] != 1'b1))
                    begin
                        error <= 1'b1;
                    end
                    else if (parity != data_read[1]) begin
                        error <= 1'b1;
                    end
                    else begin
                        key <= {data_read[2], data_read[3],
                                data_read[4], data_read[5],
                                data_read[6], data_read[7],
                                data_read[8], data_read[9]};
                    end
                    state <= S_CLEAN;
                end
                default: begin
                    state <= S_IDLE;
                end
            endcase
        end
    end
end
```

Listing 5.17: PS/2 Keyboard controller implementation

5.3.4 Timer

After the comprehension of the designed timer the following module was developed.

Timer Module

In this section, it will be explained the interface of the developed timer. In the figure below it's represented the timer module diagram.

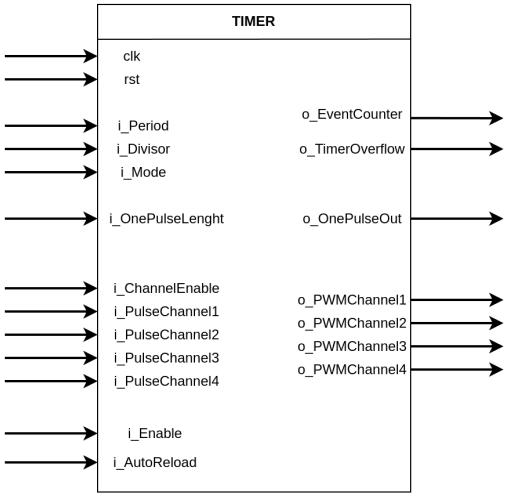

Figure 5.5: Timer Diagram

Timer Configuration

The timer configuration options, accessible through dedicated registers, provide control over its behavior.

- **Divisor (2-bit):** Divides the input clock (1x, 2x, 4x, 8x);
- **Mode:** Modifies the Timer mode;
- **ChannelEnable: (4-bit):** This register bit controls individual output activation for PWM channels 1 to 4;
- **Auto-reload (1 bit):** When enabled, the counter automatically resets to 0 upon reaching the configured Period value (only available for Timer mode);
- **Period: (32-bit):** Defines the maximum counter value in most modes (exact function might varies with mode);

- **PulseChannel1-4 (8-bit):** Sets PWM duty-cycle;
- **OnePulseLength (16-bit):** Sets One Pulse Mode pulse length;
- **Enable (1-bit):** Starts Timer;

Important: To ensure proper operation, it's recommended to configure timer settings while the timer is disabled (Enable flag reset). Only after configuration is complete should the timer be enabled.

Clock Divisor

The clock divisor operates by utilizing three additional registers, each corresponding to a divisor of 2, 4, and 8. The implementation daisy chains the input clocks: the first register uses the input clock, the second register uses the output of the first register, and the third register uses the output of the second register. This setup ensures that each subsequent register toggles its value on the positive edge of its input clock, doubling the clock period for each stage. Consequently, this chain produces clock signals with periods that are 1, 2, 4, and 8 times longer than the original input clock.

The following Verilog code demonstrates the implementation of this clock divisor:

```
// Divisor CLocks
reg r_DivisorClk2;
reg r_DivisorClk4;
reg r_DivisorClk8;
wire w_TimerClk;

always @(posedge i_clk)
    r_DivisorClk2  <= ~r_DivisorClk2;

always @(posedge r_DivisorClk2)
    r_DivisorClk4  <= ~r_DivisorClk4;

always @(posedge r_DivisorClk4)
    r_DivisorClk8  <= ~r_DivisorClk8;

assign w_TimerClk = (i_Divisor == `TIM_DIVISOR_2) ? r_DivisorClk2:
                    (i_Divisor == `TIM_DIVISOR_4) ? r_DivisorClk4:
                    (i_Divisor == `TIM_DIVISOR_8) ? r_DivisorClk8:
                    i_clk;
...
```

Listing 5.18: Divisor Implementation

As mentioned before, the developed timer has 4 types of modes: timer mode, counter mode, PWM mode and One Pulse mode. To control the desired type of timer, an input (i_mode) was developed for that purpose.

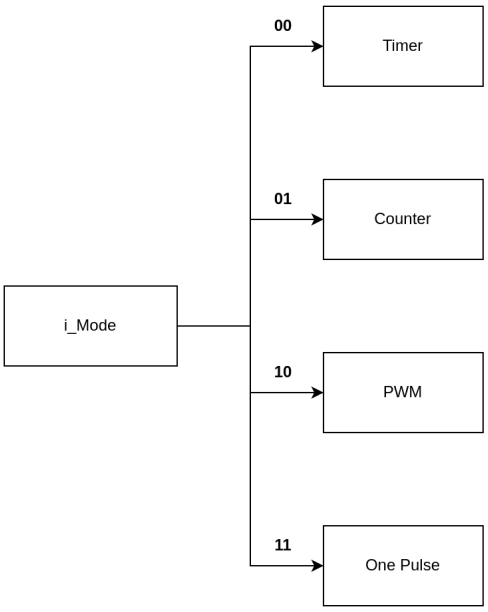

Figure 5.6: Timer Modes

- **Timer mode:**

  In Timer mode, a 32-bit counter increments from 0 to the desired period specified by the i_Period input.

  When the counter reaches the desired value, the o_TimerOverflow flag is set (the user must reset this flag). If i_AutoReload is set, the timer automatically reloads to 0 after overflow. Otherwise, the counter continues incrementing but the o_TimerOverflow flag won't be set on subsequent overflows.

```
1 //Counters
2 reg [`TIM_COUNTER_DEPTH-1:0]     r_Counter;  //Timer counter
3 ...
4
5 //Flags
6 reg r_TimeOverflow;                          //Overflow flag
7 ...
8
```

```
always @(posedge w_TimerClk) begin
    if (i_rst) begin
        r_Counter <= `COUNTER_RESET_VALUE;
        r_TimerOverflow <= 1'b0;
        ...
    end
    else if (i_Enable) begin
    case (i_Mode)
        `TIM_TIMER_MODE:
        begin
            if (r_Counter == i_Period) begin
                r_TimerOverflow <= 1'b1;

                if(i_AutoReload)
                    r_Counter <= `COUNTER_RESET_VALUE;
            end
            else begin
                r_Counter <= r_Counter + 1;
                r_TimerOverflow <= 0;
            end
        end
        ...
        else begin
            case (i_Mode)
                `TIM_TIMER_MODE:
                begin
                    r_Counter <= `COUNTER_RESET_VALUE;
                    r_TimerOverflow <= 0;
                end
        ...
```

Listing 5.19: Timer Implementation

- **Counter Mode:**

  In the Counter mode, a 32-bit event counter increments from 0 whenever the i_Enable flag is set. The o_EventCounter register continuously reflects the current counter value. Resetting the i_Enable flag will stop the counting and reset the counter value to 0.

```
// Counters
reg [`TIM_COUNTER_DEPTH-1:0]     r_Counter;
...
always @(posedge w_TimerClk) begin
    if (i_rst) begin
        r_EventCounter <= `COUNTER_RESET_VALUE;
        ...
    end
```

```
    else if (i_Enable) begin
        case (i_Mode)
            `TIM_COUNTER_MODE:
            begin
                r_EventCounter <= r_EventCounter + 1;
            end
            ...
        endcase
    end
    else begin
        case (i_Mode)
            `TIM_COUNTER_MODE:
            begin
                r_EventCounter <= `COUNTER_RESET_VALUE;
            end
        ...

assign o_EventCounter  = r_EventCounter;
```

Listing 5.20: Counter Implementation

- **PWM Mode**

  In PWM mode, the timer operates as a 4-channel PWM generator. Each channel can be independently enabled using the i_ChannelEnable register. An 8-bit counter forms the basis for PWM generation. The i_Period register defines the counter's maximum value, similar to timer mode. After reaching this value, the counter resets and the PWM cycle recommences. Duty cycle for each channel (1-4) is programmed through dedicated 8-bit registers, i_PulseChannel1-4. Setting the i_Enable flag initiates PWM generation on the designated output pins (o_PWMChannel1-4).

```
// PWM dedicated counter
reg [`TIM_PWM_COUNTER_DEPTH-1:0] r_PWMCounter;
...

always @(posedge w_TimerClk) begin
if (i_rst) begin
    <...>
end
else if (i_Enable) begin
    case (i_Mode)
        `TIM_PWM_MODE:
    begin
    if (r_PWMCounter == i_Period)
        r_PWMCounter = `COUNTER_RESET_VALUE;

```

```
16          r_PWMCounter  = r_PWMCounter + 1;
17
18          o_PWMChannel1 = ((r_PWMCounter < i_PulseChannel1
19              || r_PWMCounter == i_PulseChannel1) &&
20              i_ChannelEnable[`PWM_CHANNEL_1_INDEX])? 1'b1:1'b0;
21
22          o_PWMChannel2 = ((r_PWMCounter < i_PulseChannel2
23              || r_PWMCounter == i_PulseChannel2) &&
24              i_ChannelEnable[`PWM_CHANNEL_2_INDEX])? 1'b1:1'b0;
25
26          o_PWMChannel3 = ((r_PWMCounter < i_PulseChannel3
27              || r_PWMCounter == i_PulseChannel3) &&
28              i_ChannelEnable[`PWM_CHANNEL_3_INDEX])? 1'b1:1'b0;
29
30          o_PWMChannel4 = ((r_PWMCounter < i_PulseChannel4
31              || r_PWMCounter == i_PulseChannel4) &&
32              i_ChannelEnable[`PWM_CHANNEL_4_INDEX])? 1'b1:1'b0;
33          end
34          ...
35      endcase
36      end
37      else begin
38          case(i_Mode)
39          default:
40              begin
41                  r_PWMCounter    <= `COUNTER_RESET_VALUE;
42
43                  o_PWMChannel1   <= 1'b0;
44                  o_PWMChannel2   <= 1'b0;
45                  o_PWMChannel3   <= 1'b0;
46                  o_PWMChannel4   <= 1'b0;
47              end
```

Listing 5.21: PWM Implementation

- **One Pulse Mode:**
  In One-Pulse mode, the timer functions as a single-pulse generator. When enabled (i_Enable set), it produces a single pulse with a programmable duration defined by the 16-bit i_OnePulseLength register. The 32-bit register i_Period register, in this mode, configures the time delay before the pulse generation upon receiving a trigger.

```
1 //Counters
2 reg [`TIM_COUNTER_DEPTH-1:0]     r_Counter;
3 ...
4 // Pulse generation started flag
5 reg r_PulseGeneration;
6 ...
```

```
7  always @(posedge w_TimerClk) begin
8      if (i_rst) begin
9          r_Counter <= `COUNTER_RESET_VALUE;
10         r_PulseGeneration <= 1'b0;
11
12         // Reset One Pulse output
13         o_OnePulseOut    <= 1'b0;
14
15     end
16     else if (i_Enable) begin
17         case (i_Mode)
18             `TIM_ONE_PULSE_MODE:
19             begin
20                 if(!r_PulseGeneration) begin
21                 // Time until pulse is generated
22                     if (r_Counter == i_Period) begin
23                         r_PulseGeneration <= 1'b1;
24                         r_Counter <= `COUNTER_RESET_VALUE;
25
26                         o_OnePulseOut <= 1'b1;
27                     end
28                     else
29                         r_Counter <= r_Counter + 1;
30                 end
31                 else begin
32                 // Pulse generation
33                     if (r_Counter == i_OnePulseLength)
34                         o_OnePulseOut <= 1'b0;
35                     else
36                         r_Counter <= r_Counter + 1;
37                 end
38             end
39         endcase
40     end
41     else begin
42         case (i_Mode)
43             `TIM_ONE_PULSE_MODE:
44             begin
45                 r_Counter <= `COUNTER_RESET_VALUE;
46
47                 o_OnePulseOut <= 1'b0;
48             end
```

Listing 5.22: One Pulse Implementation

5.3.5 UART

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

- Baudrate generator;
- UART Tx;
- UART Rx;

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 = 125MHz / 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.

```
always @(posedge clk2) begin
    if(rst) begin
        (...)
    end
    else if (internal_counter == 1) begin
        //toggle tick bit
        tick <= ~tick;
        //divide the counter by 2 and update internal counter
        internal_counter <= {1'b0, baudrate_counter[31:1]};
    end
    else begin
        //decrement counter
        internal_counter <= internal_counter - 1;
    end
end
```

Listing 5.23: Baudrate generator implementation

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.

```
always @(posedge tick or posedge rst) begin
    if(rst) begin
        (...)
    end
    else begin
        case(state)
            S_IDLE1: begin                //wait for start command
                if((tx_start == 1'b1)) begin
                   tx_done <= 0;
                   state = S_START1;
                end
            end

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

            S_DATA1: begin
                //send the less significant bit
                tx_bit <= buffer[0];
                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
```

Listing 5.24: UART Tx implementation

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.

```
...
    always @(posedge tick) begin
        if(rst) begin
            (...)
        end
        else if(rx_en == 1'b1) begin
            case(state)
                `S_IDLE: begin
                    rx_done <= 0;
                    //wait for start bit
                    if(rx_bit == 1'b0) begin
                        //reset counter
                        counter <= 0;
                        //clear internal buffer
                        buffer <= 0;
                        state <= `S_DATA;
                    end
                end
                `S_DATA: begin
                    //put received bit in the most significant ...
                    //...buffer bit and shift right
                    buffer <= {rx_bit, buffer[7:1]};

                    counter =  counter + 1;

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

Listing 5.25: UART Rx implementation

5.3.6 VGA

In this section, the implemented VGA peripheral will be presented.

The following image provides a high-level schematic of the peripheral system, detailing component interconnections and signals.

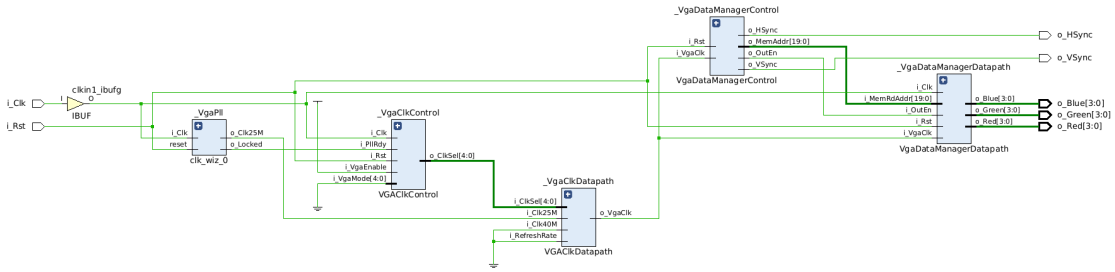

Figure 5.7: Full Schematic

The peripheral receives clock and reset inputs and outputs the necessary signals for VGA protocol (red, green, blue, vsync and hsync). It operates through the coordinated functionality of two core domains: **clock management** and **data management**, and those same domains are divided into control unit and datapath.

The clock management unit is responsible for generating the vga clock signals. Given the selected resolution, it generates the correspondent clock.

The schematic contains a Clocking Wizard block (VgaPLL), and a control block (VgaClkControl). The VgaPLL generates a stable 25 MHz clock for the VGA. The VgaClkControl manages the clock signals, reset, and lock states. The Clocking Wizard block generates the necessary clocks.

The VgaClkControl determines which clock should the VGA peripheral use based on the settings. Inputs include system clock, reset, PLL ready signal, VGA enable, and VGA mode setting. Outputs are a clock error flag and a register storing the selected clock divider/multiplier.

The VgaClkDataPath outputs the VGA clock based on the selected by the VgaClkControl.

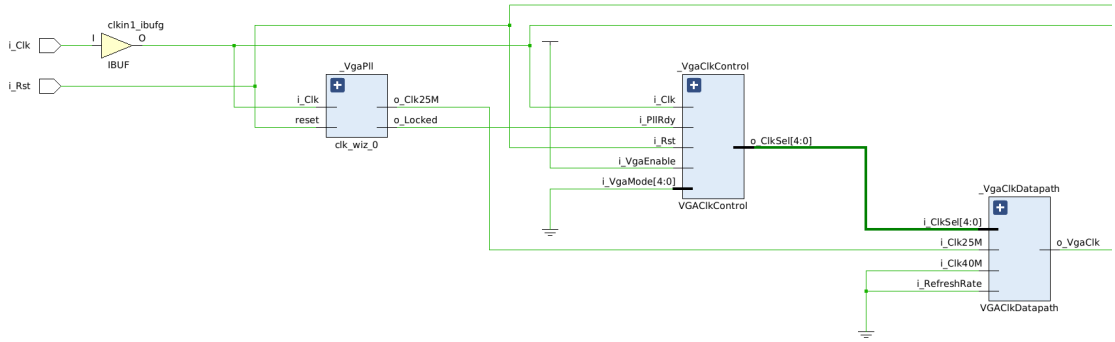

Figure 5.8: Schematic Clock Management

The data management unit within the peripheral handles the critical task of video signal generation. It processes color information, synchronizing signals, and border creation. Additionally, it accesses and traverses the video memory to retrieve the appropriate color values for each pixel on the display.

The schematic represents a VGA data manager circuit, featuring two primary blocks: the control block (VgaDataManagerControl) and the datapath block (VgaDataManagerDatapath). The control block handles the management of horizontal sync (HSync), vertical sync (VSync), memory read address and vga output enable signals. The datapath block processes the memory read address and receives the VGA clock (VgaClk), reset, and output enable signals. It outputs the RGB data signals (o_Blue, o_Green, o_Red) corresponding to the VGA display requirements.

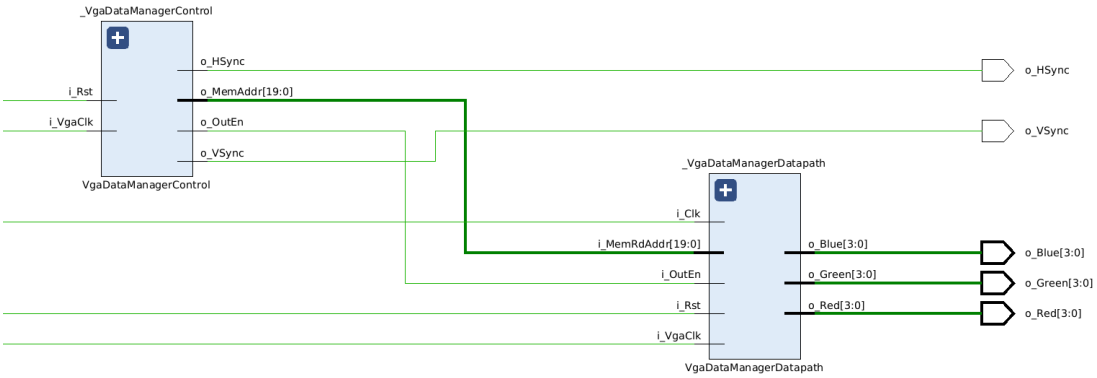

Figure 5.9: Schematic Data Management

VgaClkControl

The VGA clock controller was implement using a finite state machine. In the Initialization (ST_INIT) state, it waits for the PLL to be ready. The Idle (ST_IDLE) state is active when VGA is enabled and a valid resolution mode is selected. The Running (ST_RUN) state maintains operation as long as a valid mode is selected and VGA remains enabled. The Error (ST_ERR) state is triggered when an invalid resolution mode is detected. The code ensures the error flag is set for invalid modes and the register reflects the chosen mode only during valid operation.

```
...
always @(*) begin
    r_NextState = r_CurrentState;
    case (r_CurrentState)
        ST_INIT: begin
            if (i_PllRdy) begin
                r_NextState = ST_IDLE;
            end
        end
        ST_IDLE: begin
            if (i_VgaEnable) begin
                if (i_VgaMode > VGA_MODE_MAX) begin
                    r_NextState = ST_ERR;
                end else begin
                    r_NextState = ST_RUN;
                end
            end
        end
        ST_RUN: begin
            if (i_VgaMode > VGA_MODE_MAX) begin
                r_NextState = ST_ERR;
            end else if (!i_VgaEnable) begin
                r_NextState = ST_IDLE;
            end
        end
        ST_ERR: begin
            if (i_VgaMode <= VGA_MODE_MAX) begin
                r_NextState = ST_IDLE;
            end
        end
    endcase
end
...
```

Listing 5.26: VgaClkControl Implementation, FSM

Afterwards, based on the current state of the Clk Controller the following code outputs correct ClkSel value for the VgaClkControl. Upon a reset, the ClkSel and ClkErr outputs are cleared to zero. During normal operation, in the ST_RUN state, ClkSel is set to VgaMode + 1 and ClkErr is cleared. In the ST_ERR state, ClkSel is set to zero and ClkErr is set to one. In all other states, both ClkSel and ClkErr remain zero.

```
...
always @(posedge i_Clk or posedge i_Rst) begin
    if (i_Rst) begin
        o_ClkSel <= 5'b0;
        o_ClkErr <= 1'b0;
    end else begin
        case (r_CurrentState)
            ST_RUN: begin
                o_ClkSel <= (i_VgaMode + 1);
                o_ClkErr <= 0;
            end
            ST_ERR: begin
                o_ClkSel <= 0;
                o_ClkErr <= 1;
            end
            default: begin
                o_ClkSel <= 0;
                o_ClkErr <= 0;
            end
        endcase
    end
end
...
```

Listing 5.27: VgaClkControl Implementation, Ouput Decode

VgaClkDatapath

The VGA Clock Datapath module operates straightforwardly. It receives multiple clock signals as inputs and selects the appropriate one based on the values provided by the VgaClockControl module, then outputs the selected clock signal to the peripheral components.

```
...
parameter VGA_MODE_640_400 = 5'b00001;
parameter VGA_MODE_800_600 = 5'b00010;
// Output Assignments
assign o_VgaClk = (i_ClkSel == VGA_MODE_640_400) ? i_Clk25M :
                  (i_ClkSel == VGA_MODE_800_600) ? i_Clk40M : 0;
...
```

Listing 5.28: VgaClkDatapath Implementation

VgaDataControl

The VGA Data Control module serves as the primary controller for managing output signals. It orchestrates the timing of borders, active areas, and synchronization signals through counters.

The horizontal pixel counter increments with each clock cycle until it reaches the configured horizontal resolution, at which point it resets. Concurrently, the vertical line counter increments until it reaches the configured vertical resolution, resetting thereafter.

```
...
always @(posedge i_VgaClk or posedge i_Rst) begin
    if (i_Rst) begin
        r_HPixelCount <= 0;
        r_VLineCount <= 0;
        o_HSync <= 1;
        o_VSync <= 1;
        o_OutEn <= 0;
        o_MemAddr <= 0;
    end
    else begin
        // Horizontal Counter
        if (r_HPixelCount < H_TOTAL - 1) begin
            r_HPixelCount <= r_HPixelCount + 1;
        end
        else begin
            r_HPixelCount <= 0;

            // Vertical Counter
            if(r_HPixelCount == H_TOTAL - 1) begin
                if (r_VLineCount < V_TOTAL - 1) begin
                    r_VLineCount <= r_VLineCount + 1;
                end
                else begin
                    r_VLineCount <= 0;
                end
            end
        end
...
```

Listing 5.29: VgaDataControl Implementation, Horizontal and vertical counters

For synchronization signals, the horizontal sync signal transitions high when the horizontal pixel count falls within the horizontal sync pulse range, and low otherwise. Similarly, the vertical sync signal transitions high when the vertical line count falls within the vertical sync pulse range, and low otherwise.

```
...
    if (r_HPixelCount >= 0 && r_HPixelCount < H_SYNC_PULSE) begin
        o_HSync <= 1;
    end
    else begin
        o_HSync <= 0;
    end

    if (r_VLineCount >= 0 && r_VLineCount < V_SYNC_PULSE) begin
        o_VSync <= 1;
    end
    else begin
        o_VSync <= 0;
    end
...
```

Listing 5.30: VgaDataControl Implementation, Generate horizontal and vetical signals

Lastly the output enable and memory address outputs. The output enable signal is set high, and the memory address is updated when the horizontal pixel count and vertical line count fall within the active area. If not within this active area, output enable signal is set low, ensuring black borders.

Furthermore, due to resource limitations on the Zybo-10, it proved infeasible to accommodate all necessary values for the minimum VGA resolution within a single memory. To address this, a workaround involves extending the RGB value in each memory address by 2 pixels (both horizontally and vertically), thus covering the entire resolution. This necessitates the utilization of the following equation:

$$MemAddr = \frac{VLineCounter - (V\_SYNC\_PULSE + V\_BACK\_PORCH)}{2} \times \frac{X\_RES}{2} + \frac{HPixelCounter - (H\_SYNC\_PULSE + H\_BACK\_PORCH)}{2} \tag{5.1}$$

Doing so, its ensured that the memory address to only increment when the index increments 2 times (working both horizontally and vertically).

```
...
if (r_HPixelCount >= (H_SYNC_PULSE + H_BACK_PORCH) &&
    r_HPixelCount < (H_SYNC_PULSE +H_BACK_PORCH +X_RES)&&
    r_VLineCount >= (V_SYNC_PULSE + V_BACK_PORCH) &&
    r_VLineCount < (V_SYNC_PULSE + V_BACK_PORCH + Y_RES)) begin
    o_OutEn <= 1;
    o_MemAddr <= (((r_VLineCount -
    (V_SYNC_PULSE + V_BACK_PORCH))/2) *
```

```
9       (X_RES/2)) + ((r_HPixelCount -
10      (H_SYNC_PULSE + H_BACK_PORCH))/2);
11  end else begin
12      o_OutEn <= 0;
13  end
14  ...
```

Listing 5.31: VgaDataControl Implementation, Generate OutEn signal and Memory Address

VgaDataDatapath

The VGA Data Datapath module handles the generation of Red, Green, and Blue signals for output. It retrieves the memory address value from the VgaDataControl module and accesses the corresponding value in the Video Memory. When the output is enabled (Active Area), the module extracts the RGB values from memory. Conversely, if the output is disabled (Borders), it consistently outputs black.

```
1   module VgaDataManagerDatapath
2   (
3       input i_Clk,
4       input i_VgaClk,
5       input i_Rst,
6       input i_OutEn,
7       input [19:0] i_MemRdAddr,
8       output [3:0] o_Red,
9       output [3:0] o_Green,
10      output [3:0] o_Blue
11  );
12
13  wire [11:0] w_MemOut;
14
15  assign o_Red = i_OutEn ? w_MemOut[11:8] : 4'b0;
16  assign o_Green = i_OutEn ? w_MemOut[7:4] : 4'b0;
17  assign o_Blue = i_OutEn ? w_MemOut[3:0] : 4'b0;
18
19  VideoMemory _VgaVideoMemory
20  (
21    .clka(i_VgaClk),
22    .rsta(i_Rst),
23    .wea(1'b0),
24    .addra(i_MemRdAddr[17:0]),
25    .dina(16'b0),
26    .douta(w_MemOut)
27  );
28  endmodule
```

Listing 5.32: VgaDataDatapath Implementation

Video Memory

Finally, for the Video Memory, Block Memory Generators were used. They were configured 16-bit wide to fit in each single place the values of Red, Green and Blue (each 4-bit wide) of a single pixel. As previously mentioned, due to resource constraints on the Zybo-10, the Video Memory is half the size of the VGA resolution (320x240), which makes a total memory depth of 76800.

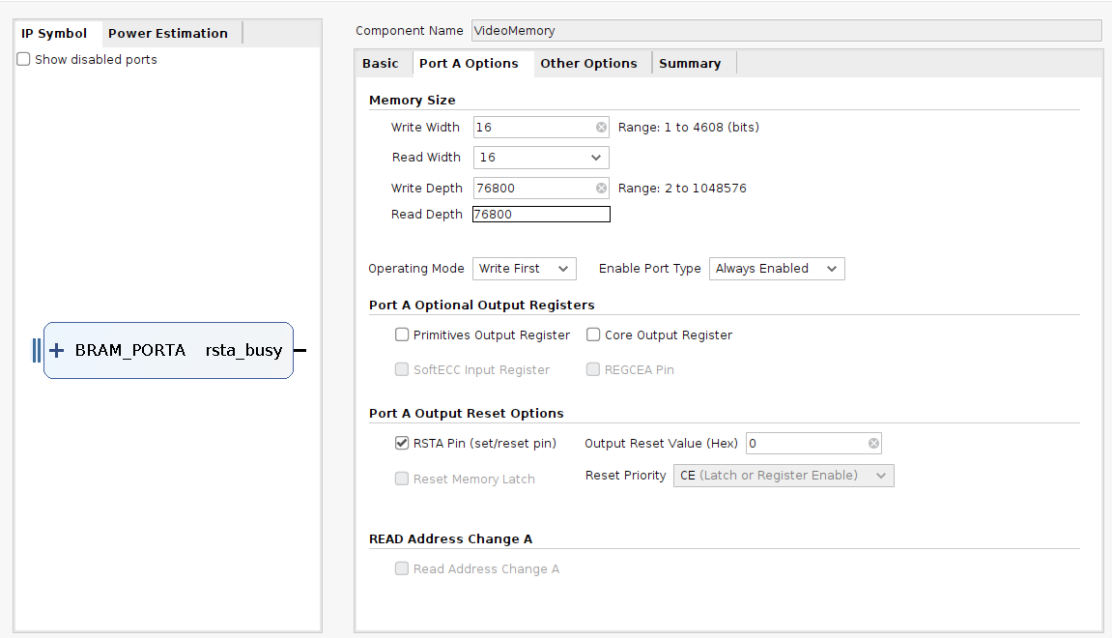

Figure 5.10: Video Memory Configuration

For testing purposes, a COE file was utilized. This file contains image values generated by a Python script specifically developed to convert images into the desired data format, subsequently stored within the COE file.

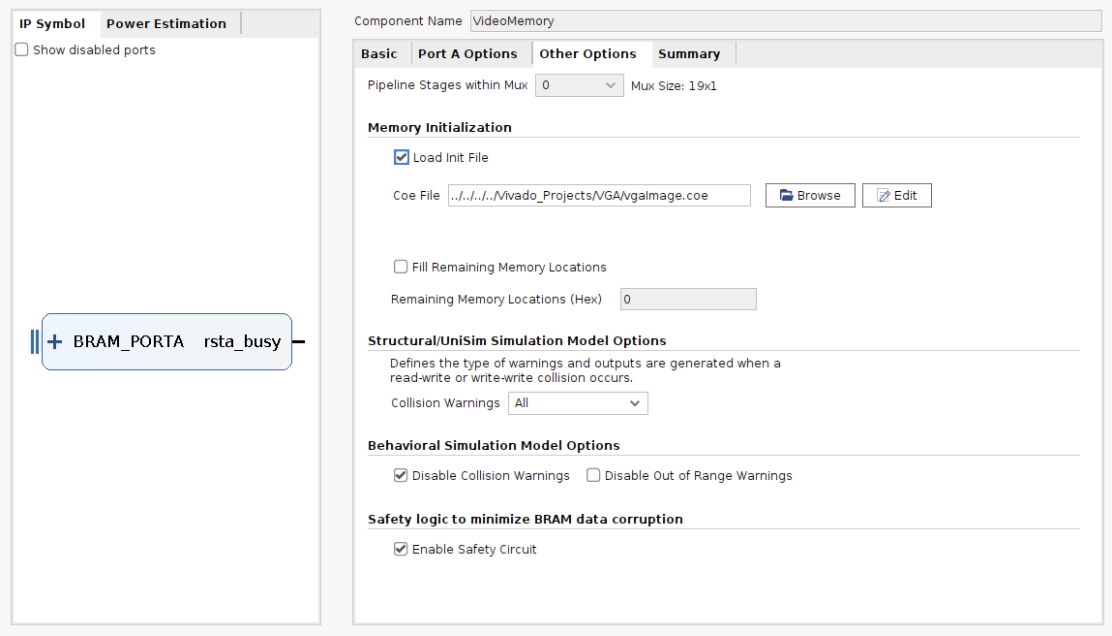

Figure 5.11: Video Memory Coe

Below is the summary of the created memory block, providing additional information.

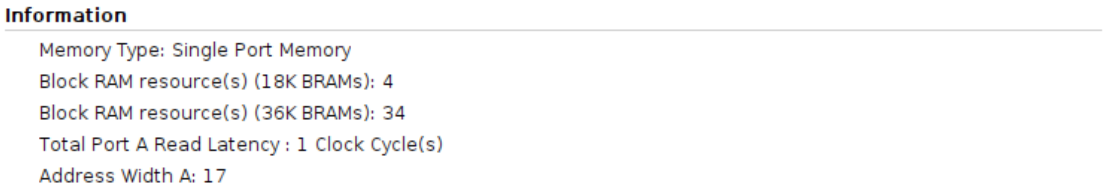

Figure 5.12: Video Memory Summary

Chapter 6

Verification

In the verification chapter, we detail the processes and methodologies employed to ensure the correct functionality and performance of our pipelined CPU design. For testing purposes, various Vivado directives were utilized with the synthesis tool, including `(* keep *)` and `(* dont_touch = "true" *)`.

- `(* keep *)` **directive** : Preserves specific signals or registers, preventing the synthesis tool from optimizing or removing them.
- `(* dont_touch = "true" *)`**directive** : Similar to keep, but can be applied to entire modules.

The use of these directives offers several advantages such as: improved debug visibility, preservation of essential signals, and maintenance of design integrity throughout the synthesis process. This facilitates a more thorough analysis and quicker identification of issues during simulation and helps maintain the validity of assumptions made during design through the verification phase. Overall leading to a more reliable and accurate final hardware implementation.

6.1 CPU

6.1.1 Barrel Shifter

The Barrel Shifter was verifiyed by simulating the module with a simple instruction that left-shifts the number five, three times.

As can be seen in figure 6.1, in the red square when the ALU left operand is '5'(value to be shifted), the right operand '3' (number of shifts) and the shift direction is '1' (left shift), we then get the correct output, '28' as expected.

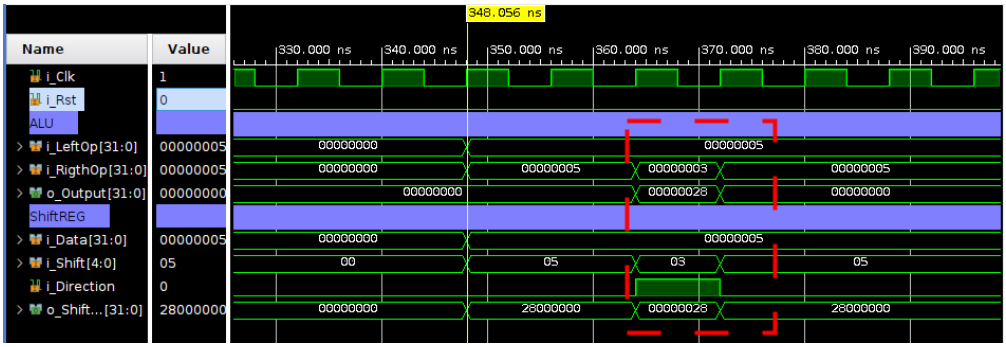

Figure 6.1: Barrel Shifter Simulation

6.1.2 Hazard Unit Integration

In the following tests the objective was to verify the behaviour of the hazard unit in mitigating data and control hazards.

Forward Mechanism

This example provides an in-depth examination of the code shown in the listing below.

```
1 LDI R2, #10
2 LDI R3, #20
3 SUB R4, R3, R2
4 OR R5, R4, R3
5 HALT
```

Listing 6.1: Forward mechanism test code

Two load immediate (LDI) instructions were used to load values to register 2 and 3. Then, the following two ALU operations SUB and OR utilize values of the previous instructions. To be completed successfully they require the forward mechanism.

The forward mechanism utilizes the multiplexer represented in the figure 6.2.

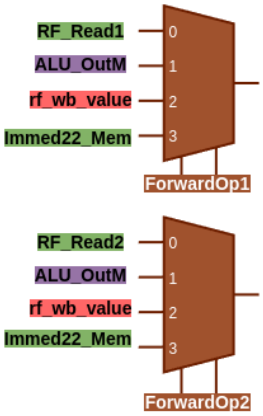

Figure 6.2: Forward Unit multiplexer

As stated in the design, there are 3 types of forwards possible:

- The forward of a value that is in the write-back stage, that was going to be written to the register file from a previous ALU operation (point 2). In this case when the instruction SUB is executed the value of its second operand is

currently in the write-back stage of first LDI instruction so it is forwarded. The same happens when the instruction OR is being executed for its second operand.

- The forward of an immediate value that is in the Memory stage (point 3). In this case, when the instruction SUB is executed the value of its first operand is an immediate value currently in the memory stage, so it is forwarded.
- The forward of an ALU output value that is in the Memory stage (point 4). In this case when the instruction OR is executed the value of its first operand is an ALU output currently in the memory stage, so it is forwarded.

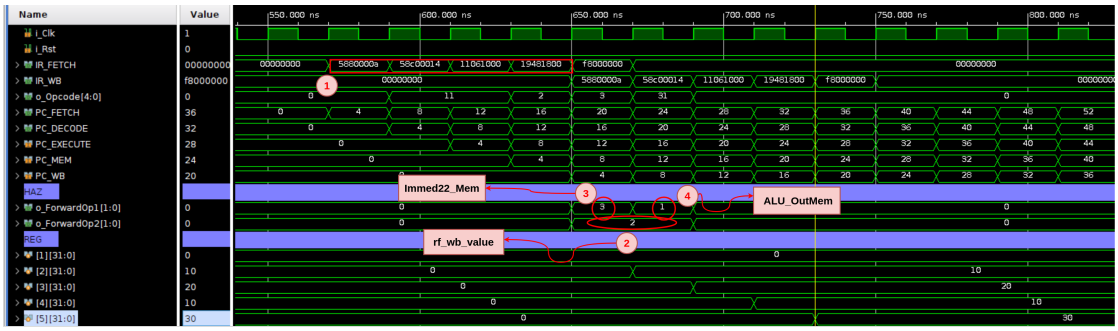

Figure 6.3: Behavioural Simulation of Forward Unit

After performing synthesis, the data forwarding unit maintains its execution perfectly. As can be seen in figure 6.3 the registers are updated with the correct corresponding values. For example, after the subtraction, the register has the value 10, which is correct even though at that moment the registers do not yet contain the written values.

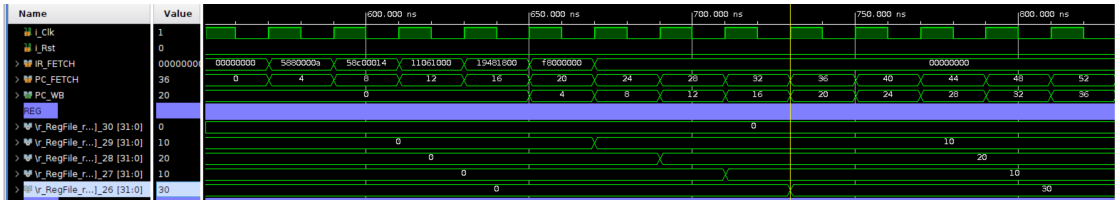

Figure 6.4: Post-Synthesis Simulation of Data Forwarding mechanism

Stall Mechanism

This example provides an in-depth examination of the code shown in the listing below.

```
1 LDI R2, #10
2 ST R2, #100
3 LD R5, #100
4 ADD R1, R3, R5
5 HALT
```

Listing 6.2: Stall mechanism test code

One load immediate (LDI) was used to load the value 10 to register 2. Then a store instruction (ST), stores the value present in register 2, in the memory address 100. Following this a load instruction (LD) stores the value present in the memory address 100, in register 5. Finally, an add (ADD) operation is performed utilizing register 5. To be completed successfully this operation requires the stall mechanism.

A load instruction accesses the data memory to retrieve a value, this is done in the Memory stage, as there is a latency cycle when obtaining the value, only when the Load is in the WB does it have access to the data. Because of this, when the Load instruction has the desired value and is in the write back stage it means that the Add instruction will be in the memory stage, as the add operation is done in the ALU in the execute state this creates an incompatibility.

To solve this, the stall signal is activated by the hazard unit (point 1). This introduces a NOP between the Load and Add instructions. Since it is not possible for the Load instruction to forward the correct value to the add, the stall mechanism is used for one cycle (point 2). At the end of the Load instruction's Memory stage (point 3), the value can be forwarded correctly (point 4). Then, the Add instruction is performed.

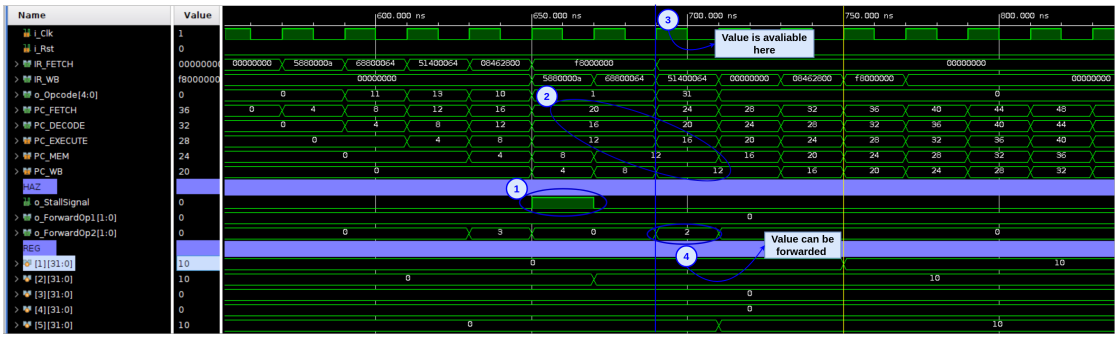

Figure 6.5: Behavioural Simulation of Stall mechanism

After synthesis we can observe in Figure 6.6 that the stall mechanism still effectively handles such dependencies, maintaining the integrity of instruction execution without compromising performance.

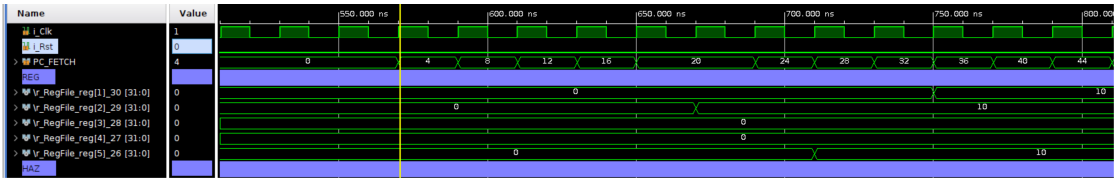

Figure 6.6: Post-Synthesis Simulation of Stall mechanism

Flush Mechanism

This example provides an in-depth examination of the code shown in the listing below.

```
LDI R2, #10
LDI R3 #10
SUB R2, R3, R2
BEQ #8
NOP
NOP
ADD R4, R2, R3
JMP R0 #0
HALT
```

Listing 6.3: Flush mechanism test code

Two Load Immediate (LDI) instructions were used to load values to register 2 and 3. The following instruction, SUB, subtracts the value in register 3 from register 2, storing the result back in register 2. Then a branch instruction (BEQ) is used, since the condition is activated the branch is executed. In the end, a jump (JMP) is executed to address 0. To be completed successfully this operations require the flush mechanism.

A branch instruction might change the flow of execution of the program. The jump instruction always does. When a change of flow is executed and the PC is altered there will always be previous executions present in the pipeline. In order to change the flow of the program correctly, these previous instructions will need to be flushed.

- For **branch instructions**, in the Decode stage the branch conditions are evaluated (point 1). If verified, at the end of the Execute stage of a branch

instruction the hazard unit sends flush signals (point 2). After this the PC of the Fetch stage will be updated correctly and all the other PC's will be flushed, resulting in the previous two instructions in the pipeline not being executed (point 3). The add instruction, where the jump was performed to, is the next instruction being executed (point 4).

- For **jump instructions**, in the Decode stage it is already known the jump is going to be executed (point 5). So at the end of the Decode stage the hazard unit sends flush signals (point 6). Then at the end of the Execute stage the PC of the Fetch stage is updated correctly (point 7), and the previous instructions in the pipeline are flushed. The HALT instruction is never executed (point 8).

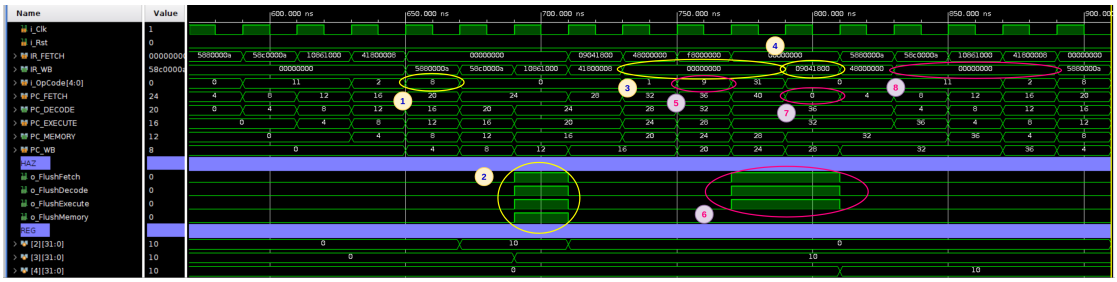

Figure 6.7: Behavioural Simulation of Flush mechanism

After performing synthesis, the data forwarding unit maintains its execution perfectly. As can be seen in Figure 6.8, the forwarding unit works correctly, and the registers are updated with the correct corresponding values. For example, after the subtraction, the register has the value 10, which is correct even though at that moment the registers do not yet contain the written values.

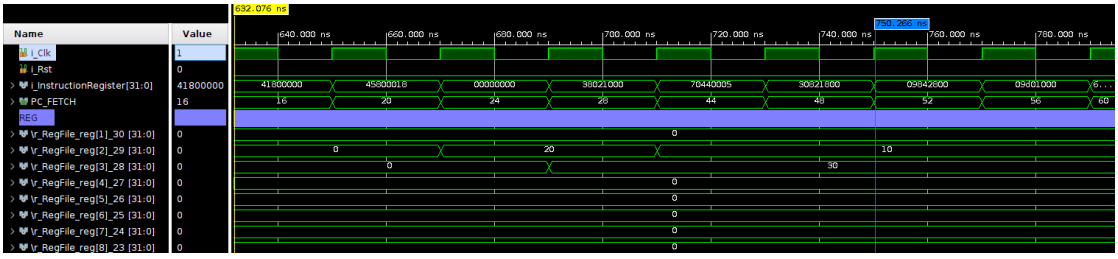

Figure 6.8: Post-Synthesis Simulation of Flush mechanism

Integration Test

This example provides the integration of every test. The code shown in the listing below combines several control and data hazards and the simulations demonstrate how the CPU responds.

```
LDI R2, #20
LDI R3, #30
SUB R2, R3, R2
BEQ #0
BNE #24
NOP
CMP R1, R2
STX R1, R2, #5
LDI R20, #4
OR R6, R3, R1
LDI R1, #2
XOR R2, R1, R3
ADD R6, R2, R5
ADD R7, R8, R2
ST R2, #100
LD R5, #100
ADD R1, R3, R5
ADD R8, R8, R5
JMP R0, #24
```

Listing 6.4: Test Code

The resulting behavioural simulation is demonstrated in Figure 6.9.

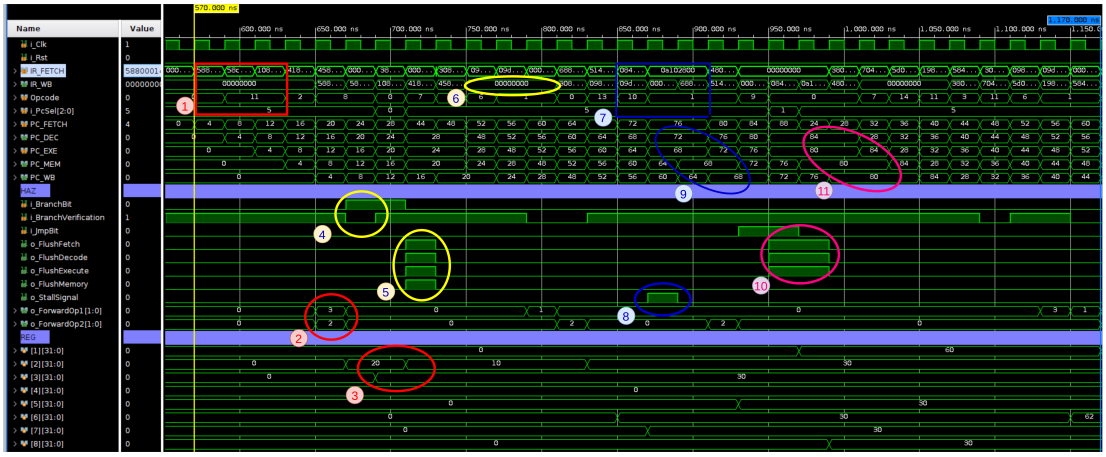

Figure 6.9: Behavioural Simulation of Hazard Unit test

As can be seen in Figure 6.9, the first two instructions are executed loading values 20 and 30 into registers 2 and 3 (point 1). The following instruction, SUB, subtracts the value in register 3 from register 2, storing the result back in register 2. Its possible to confirm the values placed in the registers (point 3). This instruction is performed without delays thanks to the forward mechanism implemented in the Data Hazard Unit (point 2). This mechanism ensures the previous result is available for the next operation.

Following this, two Branches are executed, where the BNE condition is met because of the result of the previous operation. Branch execution involves evaluating the condition in the execute stage and updating the program counter correctly. When a branch is confirmed during the decode stage, a BranchBit is set to 1 (point 4). This signals the Hazard Unit to flush the instructions already in the pipeline. The Hazard Unit responds by setting the Flush Signals to 1 (point 5). This ensures the IR of the previous instructions in the pipeline are reset to zero (point 6), guaranteeing the correct execution of the pipeline.

Since the BNE condition is met, the PC is updated with its current value plus the immediate value in the instruction which in this case is 24, therefore a jump to the instruction XOR is executed. The program continues from there, and after a couple of ADDs and a ST, the next sequence which consists of a LD and two ADDS is executed (point 7). This sequence demonstrates the system's ability to manage RAW dependencies involving load instructions. In this scenario, the load instruction accesses the data memory to retrieve a value that will be written in the last stage of the pipeline. Since the value is not available for forwarding at the end of the load execute stage but only at the end of the data memory stage, the pipeline must stall for one cycle. So when the hazard is detected the Stall signal is activated (point 8). The effect of the stall mechanism pausing the pipeline can be seen (point 9), allowing the correct data to be forwarded in the following cycle, as can be verified in the registers 1 and 8.

The sequence concludes with a JMP to address 24, flushing the Fetch, Decode, and Execute stages (point 10) before jumping to the correct PC value (point 11), thereby changing the flow of execution correctly.

6.1.3 Timings

Timings Verification of a Pipelined CPU

In modern computer architectures, especially those involving pipelined CPUs, timing verification is a critical aspect of CPU verification.

This subsection focus on the necessity to analyze the times at each stage which is crucial in a pipelined processor due to:

- **Performance Optimization:**By verifying the stage timings, the pipeline stages can be balanced, maximizing the CPU throughput, also understanding the timing of various instructions through the pipeline stages helps in identifying bottlenecks. Optimizing these bottlenecks can significantly enhance overall CPU performance.
- **Predictability:** Ensuring that each stage of the pipeline operates within the specified time constraints is crucial for the correct functioning of the CPU, this way timing analysis helps in verifying that instructions are executed within the expected time frames.
- **Measure the maximum clock:** Verifying the maximum CPU clock, by measuring the instruction that takes the most time, engineers can ensure that the CPU operates at optimal performance levels while maintaining the system reliability.
- **Power Efficiency:** Timing analysis can lead to better power management by allowing the CPU to enter low-power states when certain stages are idle, thereby reducing overall power consumption.

In order to get this timings we must measure, in a Post-Synthesis simulation, the time elapsed between the clock rise of the instruction in the desired stage and the most time-critical event in that stage.

Fetch Stage timing measurements

In the fetch stage, the event that takes the most time is the instruction register output from the code memory. This time is practically the same for all types of instructions, since it only depends on the latency of reading the code memory. There may be some variations in the time depending on how long the output of the address multiplexer takes to stabilize, but these variations can be disregarded.

In figure 6.10 it is possible check the time between the rising edge of clock and the stabilization of the code memory output.

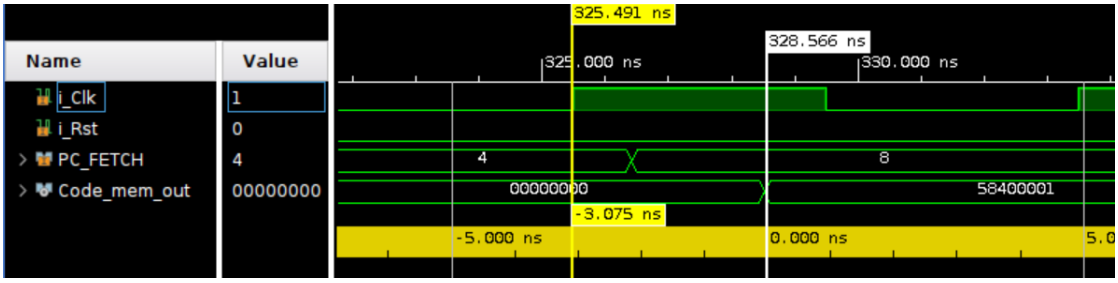

Figure 6.10: Timing on fetching instruction

Decode Stage timing measurements

In order to measure the time required in the decode stage, it is necessary to measure the time between the rising edge of clock and the stabilization of the signals coming from the control unit (signals relevant to the instruction). In figure 6.11 can be seen the decode timing for an ADD instruction.

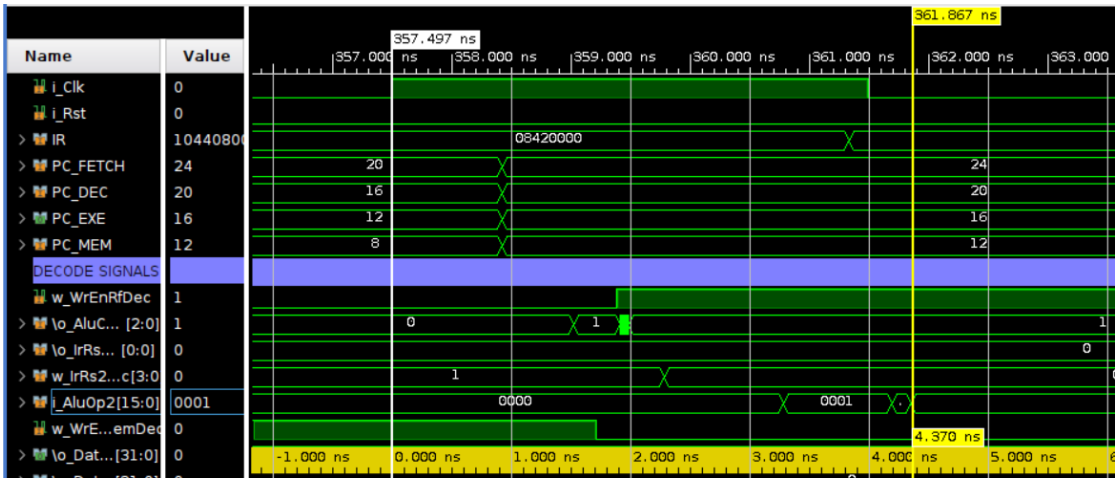

Figure 6.11: Decode timing of an ADD instruction

Execute Stage timing measurements

Regarding execute stage, instructions can be divided into two categories: ALU operations and non-ALU operations.

To measure the execute time of non-ALU instructions, the time from rising edge of clock to the stabilization of the result of the address calculation.

To measure the execute time of ALU instructions, the time from the rising edge of the clock to the stabilization of the ALU output is measured. This event can be

seen in figure 6.12 on the highlighted signal in the bottom (ALU's output). At this cycle is executed an OR instruction.

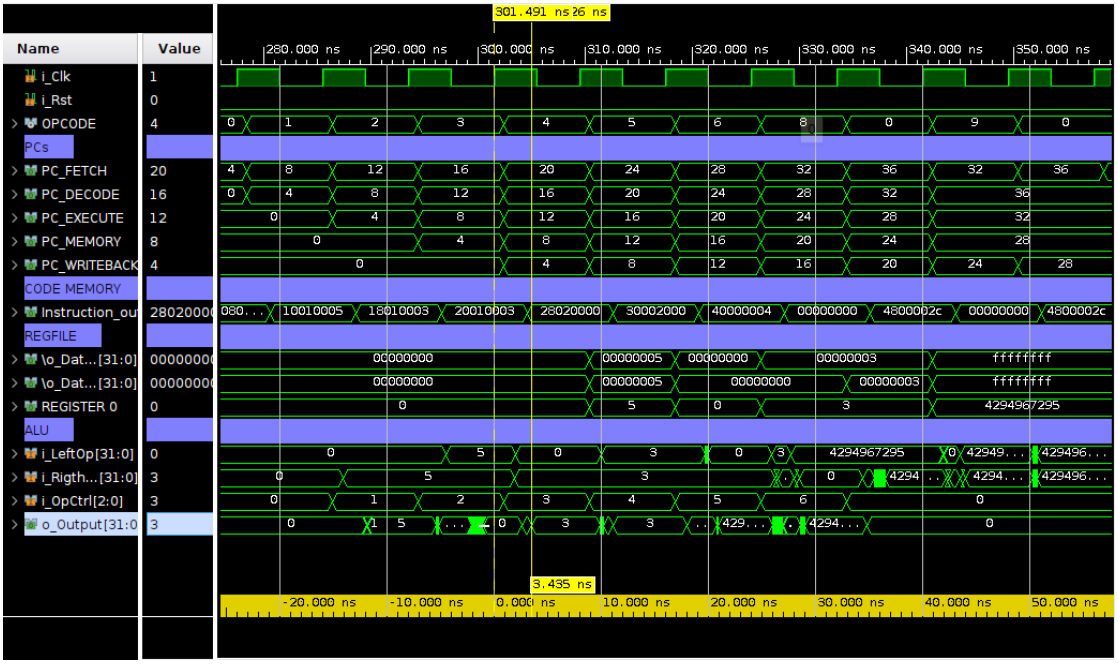

Figure 6.12: Execute time of an OR instruction

Memory Stage timing measurements

In the Memory stage, only four instructions consume time: LD, ST, LDX, and STX. As expected, the critical event is data availability at the memory output for a read instruction. BRAMs have a read latency cycle, but this cycle is compensated in the pipeline design (as explained above) since the memory output bypasses the register file between the memory and write-back stages.

Other critical event is a write instruction, where is necessary wait for data availability at the desired memory address.

As the memory cannot be accessed in the post-synthesis simulation, the figure was not available.

Write Back Stage timing measurements

In the Write back stage, as expected the time-critical event is the waiting related to the data being written in the desired register in the register file.

Timing results

After analysing in post-synthesis simulation all the instruction in every stage, the table 6.1 was constructed.

Instruction	Time (ns)				
	Fetch	Decode *	Execute	Memory	Write-back
ADD	3.038	4.37	4.046	-	1.96
SUB	3.038	3.86	6.930	-	1.96
OR	3.038	2.76	3.435	-	1.96
AND	3.038	2.023	3.051	-	1.96
NOT	3.038	1.95	2.025	-	1.96
XOR	3.038	2.54	3.017	-	1.96
CMP	3.038	3.86	6.93	-	-
BXX	3.038	3.23	4.36	-	-
JMP	3.038	3.150	4.36	-	-
LD	3.038	2.65	-	3.95 *	1.96
LDI	3.038	1.69	-	-	1.96
ST	3.038	1.89	-	2.93	-

***NOTE:** In these cases, the memory has a latency of 1 clock cycle, but this has been attenuated by not passing this signal through the register file between stages.

Table 6.1: Timing results

As can be seen in the previous table, the maximum time is in execute state of a SUB instruction. This result is due to the fact that, in hardware, subtraction is done using two adders connected in cascade which extends the critical path, making the execute stage the bottleneck in the pipeline.

Clock frequency calculation

Finally, after obtaining all the timings needed (figure 6.1) the maximum clock frequency can be calculated using the following the expression:

$$f = \frac{1}{T} = \frac{1}{6.93\,\text{ns}} = 144,3MHz \tag{6.1}$$

6.2 Peripherals

6.2.1 Interrupt Controller

In order to verify the correct operation of the interrupt controller in different scenarios, the following code was written in assembly. This code begins with a JMP on reset to bypass the interrupt vector. In the interrupt vector, there is a JMP to the respective interrupt service routine. After that, there is a set of instructions to enable the interrupts. In the main loop, some random operations are performed. In the interrupt service routines, only two LDIs and one RETI are executed.

```
1  JMP R0, #20      ;reset jump
2  JMP R0, #72      ;interrupt vector
3  JMP R0, #84
4  JMP R0, #96
5  JMP R0, #108
6  LDI R1, #15      ;enable interrupts (ea, en1, en2, en3)
7  ST R1, #1024
8  LDI R5, #1
9  ST R5, #100      ;random code
10 LDI R11, #11
11 LD R5, #100
12 ADD R5, R5, #1
13 ST R5, #100
14 LDI R20, #20
15 LDI R30, #30
16 LDI R25, #25
17 JMP R0, #20
18 LDI R2, #1       ;ISR 0
19 LDI R3, #1
20 RETI
21 LDI R4, #1       ;ISR 1
22 LDI R5, #1
23 RETI
24 LDI R6, #1       ;ISR 2
25 LDI R7, #1
26 RETI
27 LDI R8, #1       ;ISR 3
28 LDI R9, #1
29 RETI
```

Listing 6.5: Test Code 2

Simulation 1 - Interrupt request during non-control instructions

The figure below shows the result of simulating the interrupt controller when there are no jump instructions in the pipeline stages. Two interrupt sources (1 and 2) were triggered to verify if the one with higher priority is handled first.

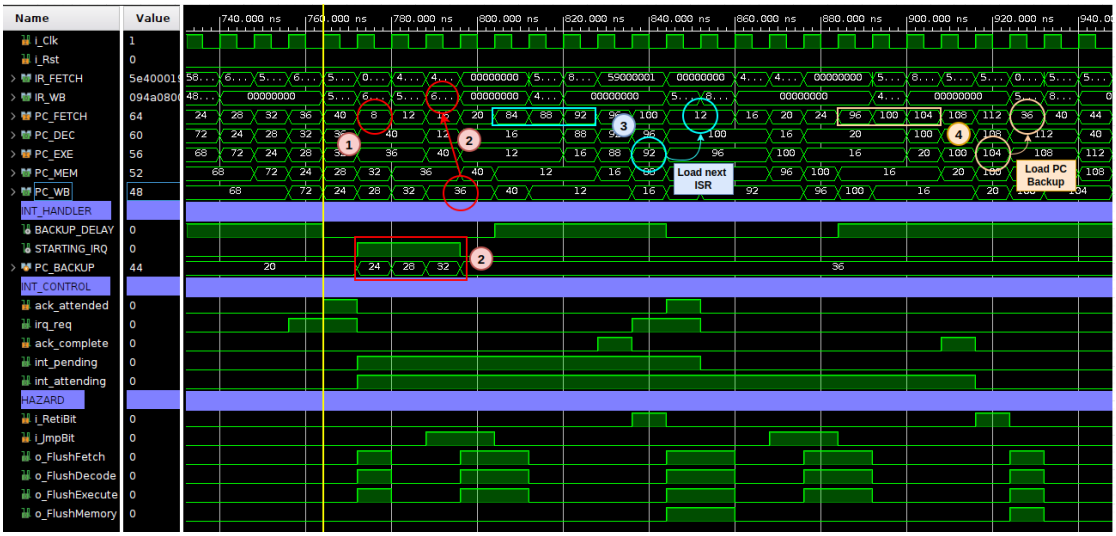

Figure 6.13: Interrupt request during non-control instructions

As demonstrated in the simulation, when an interrupt occurs (yellow line), it is processed in the next clock cycle by updating the program counter (point 1) in the fetch stage to the address of the interrupt vector for source 1, which has a higher priority than source 2. Additionally, it flushes the fetch, decode, and execute stages, as can be seen at the bottom of the figure.

With the exception of jump instructions, which are treated in a special way in interrupt control, an instruction is considered complete when it reaches the write-back stage. Therefore, to easily detect whether an instruction has been completed, you can look at the instruction register of the write-back stage and check that it corresponds to the respective instruction. As you can see in point 2 of the simulation, when an interrupt starts to be serviced, 2 instructions are still being executed, since the execute and write-back stages are not flushed. Therefore, PC_backup has to be updated for another 2 clock cycles.

As can be seen in point 3, the RETI of the first interrupt is ignored because another interrupt is pending. Instead of placing the address in the PC of the fetch stage, the address of the next interrupt to be handled is used.

Finally, can be seen that at the end of the two interrupts the program counter of the fetch stage is changed to the value in PC_Backup.

Simulation 2 - Interrupt request when JMP/BXX is in execute stage

The figure below shows the result of simulating the interrupt controller when there are jump or branch instructions in the execute stage.

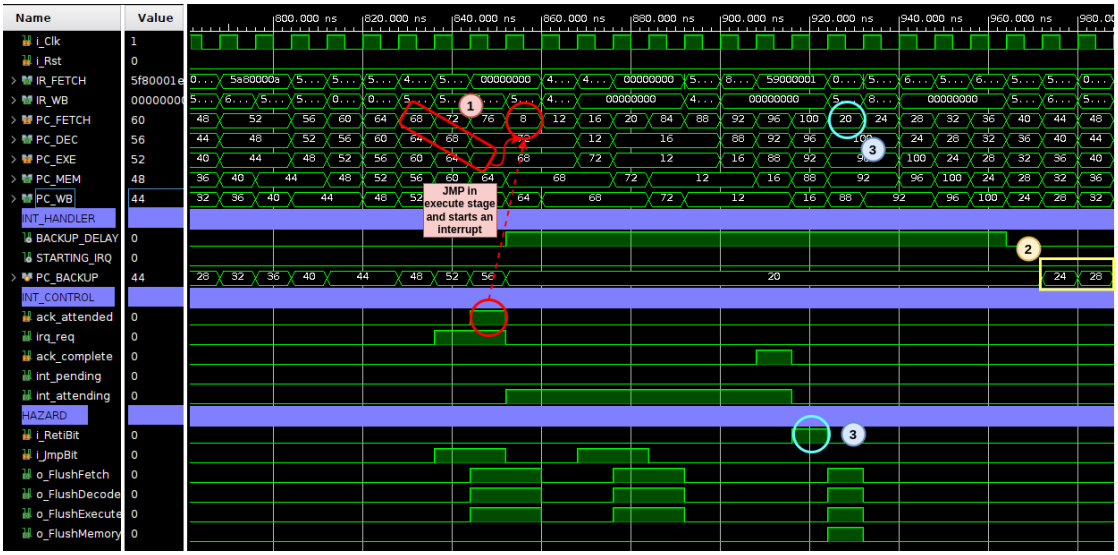

Figure 6.14: Interrupt request when JMP/BXX is in execute stage

As demonstrated in the simulation, when an interrupt occurs, it checks if the jump instruction is in the execute stage and instead of fetching the jump address, the PC fetches the ISR address and stores the jump address in the PC backup. This way, at the end of the ISR, the jump can be completed by placing the jump address back into the PC (point 3).

Simulation 3 - Interrupt request during the first 6 cycles after a JMP instruction

The figure below illustrates the outcome of simulating the interrupt controller when interrupt requests occur within the first 6 cycles following a JMP instruction.

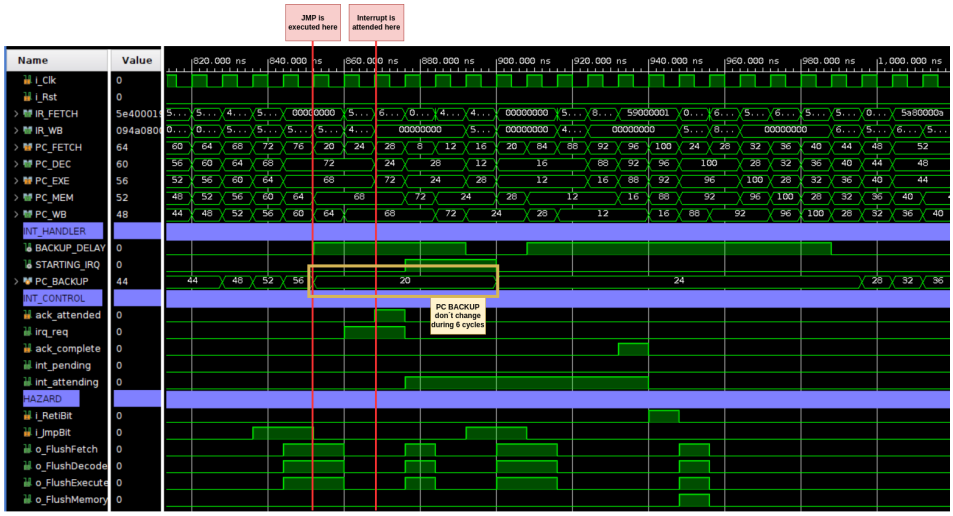

Figure 6.15: Interrupt request during the first 6 cycles after a JMP instruction

When a jump instruction is executed, some pipeline are flushed. So, this way, the program in the write-back will be empty until a new valid instruction reaches the write-back stage, i.e. during 6 cycles. So the PC_Backup cannot be changed during this time.

6.2.2 GPIO

In this test, the operating mode was set to input. A signal of 0xF1 was connected to the pins, and 0x20 was written to the GPIO input. Then, the operating mode was changed to output, and 0x55 was written to the input.

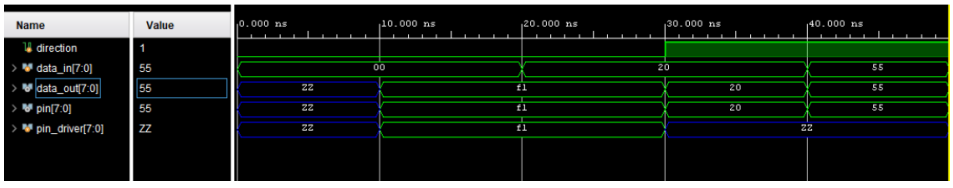

Figure 6.16: GPIO Behavioral Simulation

As observed, the peripheral operated as intended, outputting the correct value corresponding to the input data.

6.2.3 PS/2 keyboard controller

With the use of a logic analyzer, the PS/2 keyboard's clock frequency measured was, approximately, 13kHz, as can be seen in the figure 6.17. Consequently, there is a need to downscale the FPGA clock frequency to at least 26kHz, following Nyquist's theorem. The used clock source was the one available on the ZYBO Z7-10 which as a frequency of 125MHz. Dividing this by 4000 results in approximately 32kHz.

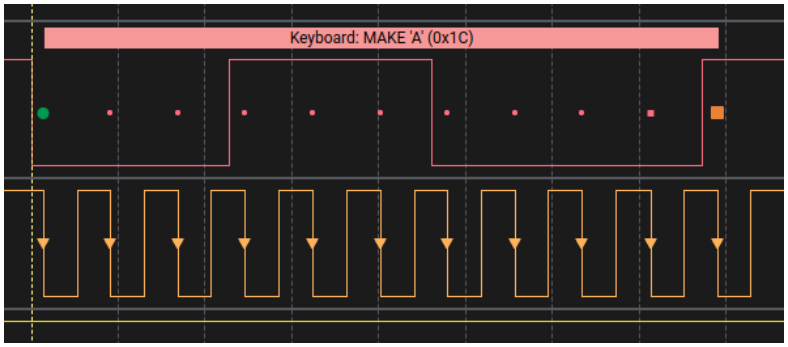

Figure 6.17: PS/2 data output

In the flowchart of the next page can be seen the algorithm the CPU is running in this PS/2 verification. First of all the PS/2 and UART are initialized and the PS/2 key is read from the peripheral. After that, it is ensured that the process does not continue if the received key is repeated or matches the value emitted by the keyboard when the key is released (0xF0). Then, the UART communication is set in order to showcase the pressed key on a serial terminal. The code doesn´t continue until the full buffer is transmitted.

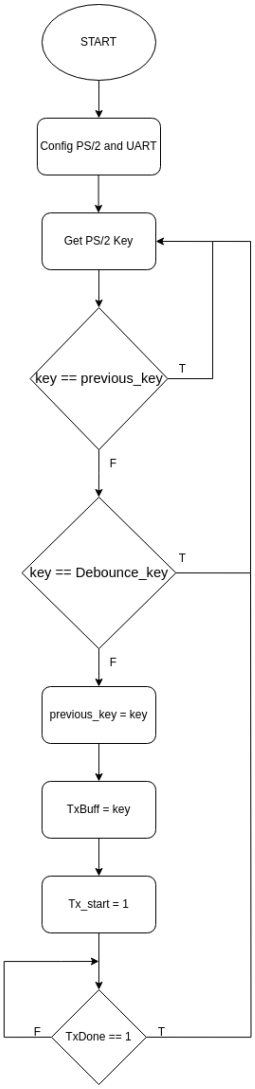

Figure 6.18: PS/2 echo flowchart

In the figure 6.19 can be seen a Post-synthesis timing simulation with the behaviour of the previously displayed algorithm.

As expected, the PS/2 is correctly enabled by writing the value one to the first address of the peripheral(highlighted in the left in the "value" column). As well the baudrate is correctly set to 13021 which is the decimal value that must be received to set the UART baudrate to 9600 bps. Afterwords is presented the value 5 being written enabling then the UART successfully.

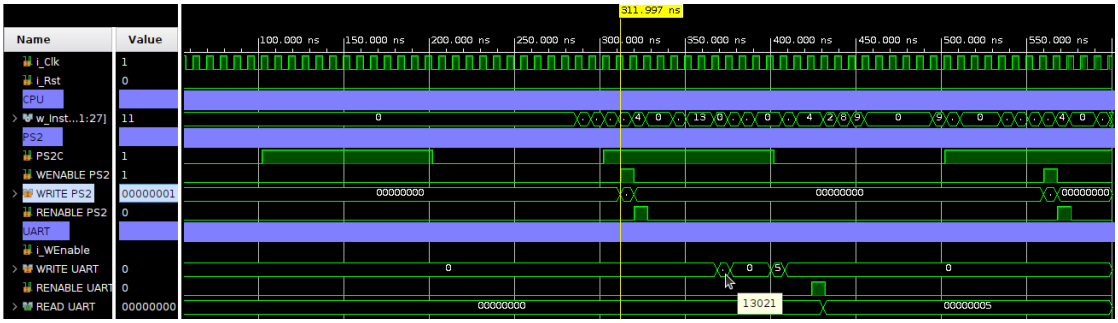

Figure 6.19: PS/2 Post-Synthesis simulation

Finally, was wired the PS/2 keyboard to a level-shifter to lower the keyboard output voltage of 5 volts to 3.3 volts since the FPGA only supports signals with that voltage. The FTDI module was connected to the FPGA and to a computer leading the data to a serial terminal.

The keyboard was tested by pressing the key "B" which, in the PS/2 protocol scan codes corresponds to the hexadecimal 0x32, and it is presented in figure 6.20 which showcases the perfect functioning of the PS/2 (and UART Tx module).

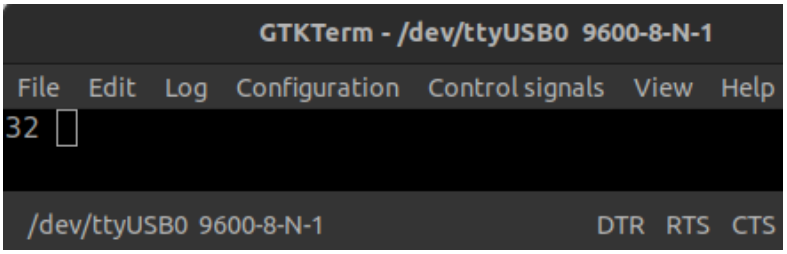

Figure 6.20: PS/2 FPGA test

6.2.4 Timer

Simulation 1 - PWM Mode Verification

This example sets the timer to PWM mode and configures the pulse widths for four channels. The goal is to verify the correct generation of PWM signals with specified pulse widths and periods.

For reference the memory addresses of the timer's registers are listed in the table 6.2:

Register Name	Description	Memory Address
TCR (Timer Configuration Register)	Timer mode and configurations	1048
TPR (Timer Period)	Timer's clock period	1049
TCE (Timer Channel Enable)	Enable PWM channels	1051
PCP1 (Channel 1 and 2 PWM Pulse)	PWM pulse width for Channel 1 and 2	1052
PCP2 (Channel 3 and 4 PWM Pulse)	PWM pulse width for Channel 3 and 4	1053

Table 6.2: Timer Registers and Memory Addresses

In this example the load (LD) and store (ST) instructions were used to load immediate values to the timer's registers.

```
NOP
LDI R1, #20
ST R1, #1049       ; TPR
LDI R1, #2565
ST R1, #1052     ; PCP1
LDI R1, #5135
ST R1, #1053       ; PCP2
LDI R1, #15
ST R1, #1051     ; TCE
LDI R1, #11
ST R1, #1048       ; TCR
HALT
```

Listing 6.6: Test Code 1

First the timer's clock period was set to 20, in the TPR. Then the PWM pulse width for Channel 1 and Channel 2 was set to 5 and 10 respectively, in the PCP1. And the PWM pulse width for Channel 3 and Channel 4 was set to 15 and 20 respectively in PCP2. Following this all PWM channels were enabled in the TCE. Finnaly, the timer was set to PWM mode, and the timer set to run using the TCR.

In the simulation in Figure 6.21 it can be seen the PWM pulse of 5, 10, 15 and 20 for each channel, and the respective duty-cycles of 25%, 50%, 75% and 100% successfully testing the behaviour of the PWM mode.

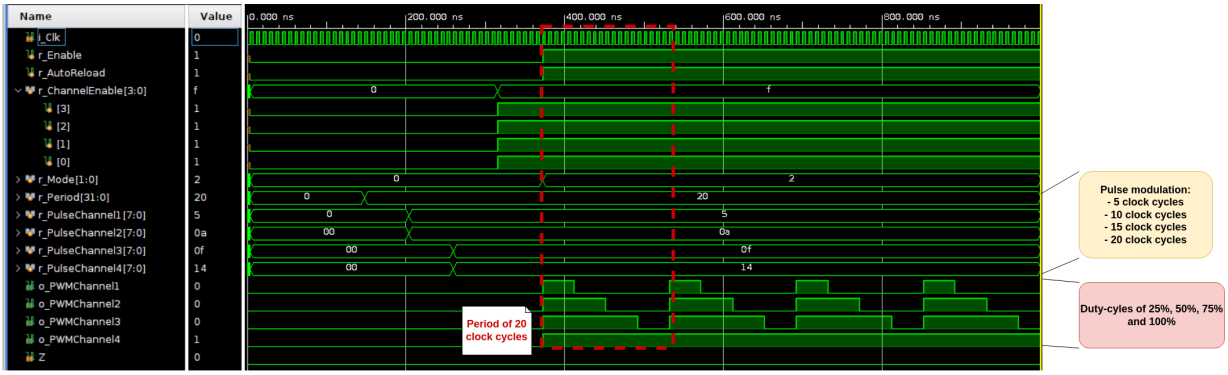

Figure 6.21: PWM Mode Behavioural Simulation

The post-synthesis simulation in Figure 6.22 verifies the correct generation of PWM signals post-synthesis as can be seen in the respective PWM Channels duty-cycles.

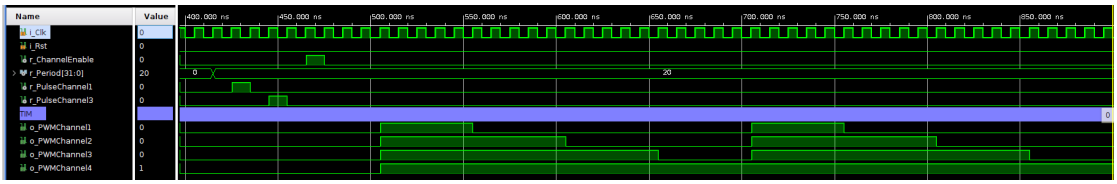

Figure 6.22: Post-Synthesis Simulation of PWM Mode

Simulation 2 - PWM Mode Verification

This example consist on a loop that overtime increases the PWM Pulse. Below is the code used.

```
NOP
LDI R1, #10
ST R1, #1049 ; TPR
LDI R2, #1
ST R2, #1051 ; TCE
LDI R3, #0
ST R3, #1052 ; PCP1
LDI R4, #11
ST R4, #1048 ; TCR
NOP
ADD R6, R5, #1
ST R6, #1052 ; PCP1
ADD R5, R6, #0
SUB R10, R5, R1
BEQ #4
```

```
16 JMP R31, #36
17 LDI R5, #0
18 ST R5, #1052 ; PCP1
19 JMP R31, #36
20 HALT
```

Listing 6.7: Test Code 2

First, the timer period was set to 10 clock cycles in the TPR. Then Channel 1 was enabled, by setting the value of TCE to 1. Following this the pulse width for Channel 1 was initialized to 0 in the PCP1. Finally, the timer was set to PWM mode, and the timer set to run using the TCR.

During execution, the program performs an addition operation, adding 1 to the value of register 5, and storing the result in register 6, which represents the pulse width for PWM Channel 1. The new pulse width value is then set in the PCP1. Subsequently, the program compares the pulse width value with the previously set period value. If the pulse width is equal to the period, the program halts. Otherwise, it continues execution. If the pulse width exceeds the period, it is reset to 0, ensuring that it remains within the specified range.

In the simulation in Figure 6.23, it can be seen that the PWM pulse width of PWM Channel 1 is being incremented over time. This successfully tests the behaviour of the PWM mode.

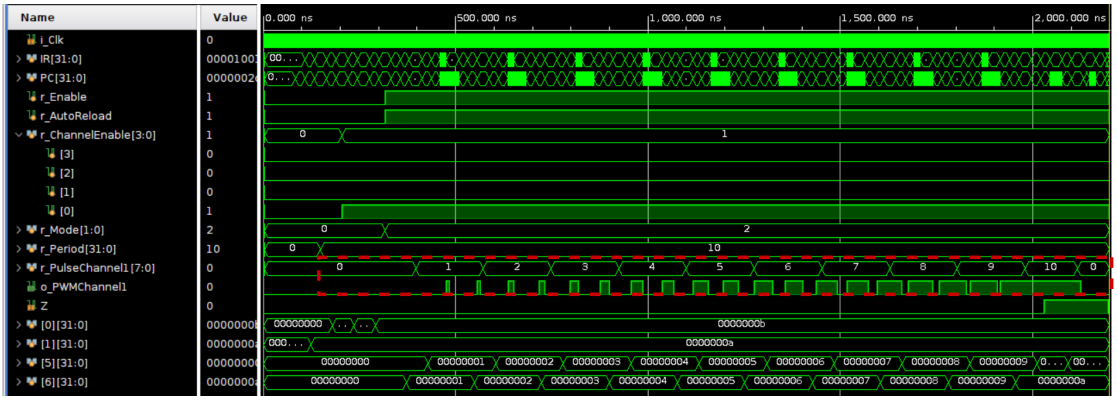

Figure 6.23: PWM Mode Behavioural Simulation

The post-synthesis simulation in Figure 6.24 verifies the same behaviour as can been seen by the increase of the pulse width of PWM Channel 1.

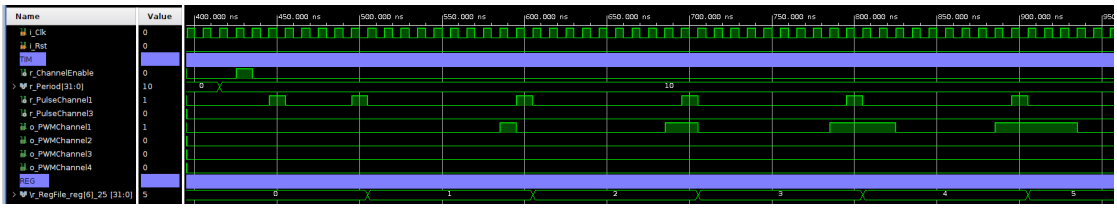

Figure 6.24: Post-Synthesis Simulation of PWM Mode

6.2.5 UART

First of all a unitary test was performed in the UART, in order to verify the correct individual component behaviour

UART Tx simulation

In figure 6.25 presented below, it is possible to check the correct functioning of the Transmission module (Tx), 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 followed, 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.

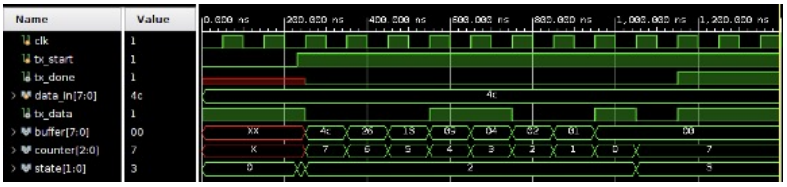

Figure 6.25: UART Tx behavioural simulation

UART Rx simulation

Through the simulation presented below in figure 6.26 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.

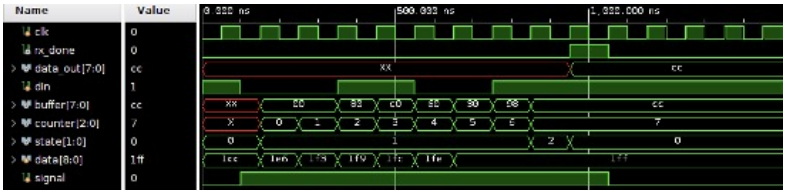

Figure 6.26: UART Rx behavioural simulation

After the unitary test, the UART peripheral was integrated in the SoC and tested with an echo algorithm as presented in the code listing bellow:

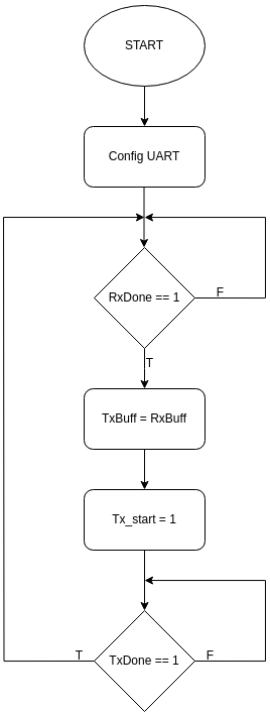

Figure 6.27: UART echo fluxogram

```
LDI R0, #0
LDI R1, #13021    ;Set Baudrate 9600
ST R1, #1033
NOP
LDI R1, #3        ;Set Rxenable e R_Enable to 1
ST R1, #1032
NOP
NOP
LD R2, #1032 ;{w_RxDone, w_TxDone, r_TxStart, r_RxEnable,r_Enable}
ANDI R2, R2, #16 ;Mask of 10000
SUBI R5, R2, #16 ;If RxDone = 1
```

```
12 BEQ #16           ;Jump to line 15
13 NOP
14 NOP
15 JMP R0, #28       ;Else jump to line 8
16 NOP
17 NOP
18 LD R3, #1035      ;Read received char
19 ST R3, #1034      ;Place char on TxBuff
20 NOP
21 LDI R4, #5        ;Set Tx_start to 1
22 ST R4, #1032
23 NOP
24 LD R2, #1032   ;{w_RxDone, w_TxDone, r_TxStart, r_RxEnable,r_Enable}
25 ANDI R2, R2, #8   ;Mask 01000
26 SUBI R5, R2, #8   ;If TxDone = 1
27 BEQ #16           ;Jump forward
28 NOP
29 NOP
30 JMP R0, #84       ;Else Keep waiting for TxDone
31 NOP
32 NOP
33 ST R0, #1032      ;Reset flags
34 ST R0, #1034      ;Reset Txbuff
35 JMP R0, #0        ;Jump to beginning
36 HALT
```

Listing 6.8: Test UART

Finally, the UART peripheral was tested on the FPGA by connecting a FTDI module from the FPGA PMOD to a desktop (ready to receive serial communication on a GTK terminal) and can be observed that the data was received by the VeSPA and transmitted to the desktop correctly.

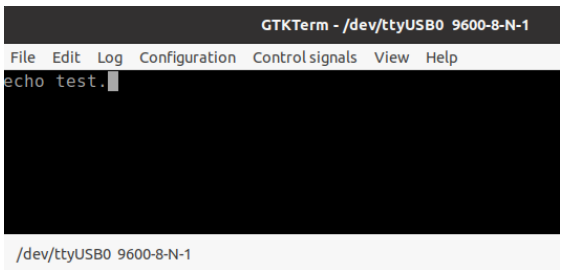

Figure 6.28: UART FPGA test

6.2.6 VGA

This section covers the conducted tests to validate the implemented VGA driver.

Using the previously referred script, a pattern was loaded to the COE file to verify if the output image is being generated correctly.

The pattern consists of values alternating between 0xF00, 0x0F0 and 0x00F to test if the RGB values would change correctly. These were the results:

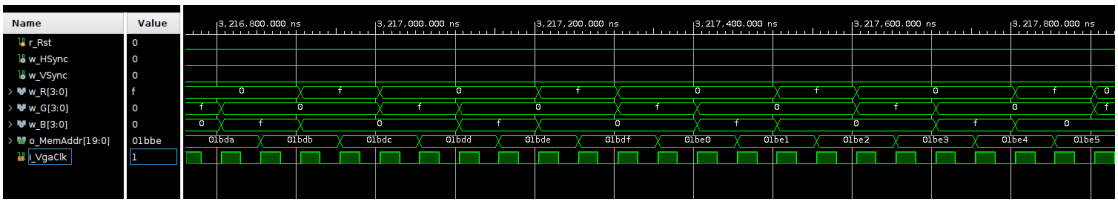

Figure 6.29: Behavioral Simulation, RGB color switching

As observed in the image above, the RGB values correctly alternate according to the loaded pattern. Additionally, the address changes only after 2 VGA clock cycles, confirming that each RGB value is used twice.

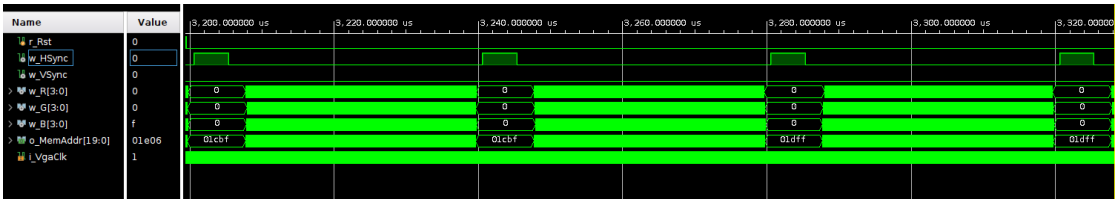

Figure 6.30: Behavioral Simulation, RGB color swicthing

Furthermore, as presented in the image above, the RGB values for each row are being repeated once.

Therefore, as pretended, each RGB value stored in a single memory address is being used for 2 pixels both horizontally and vertically.

The same simulation was run until the entire image was generated. From the resulting images, we verified that all borders (top, bottom, left, and right) are being generated correctly, and the horizontal and vertical sync signals are being triggered as intended.

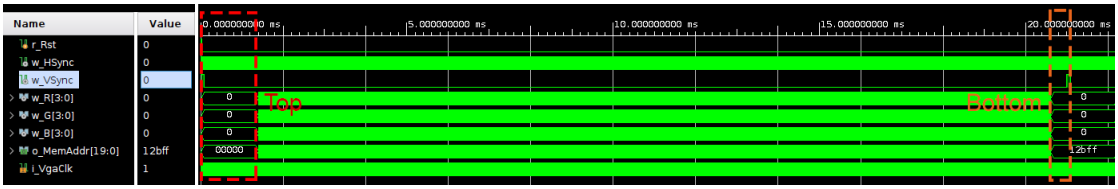

Figure 6.31: Behavioral Simulation, Top and Bottom borders generation and vsync signal

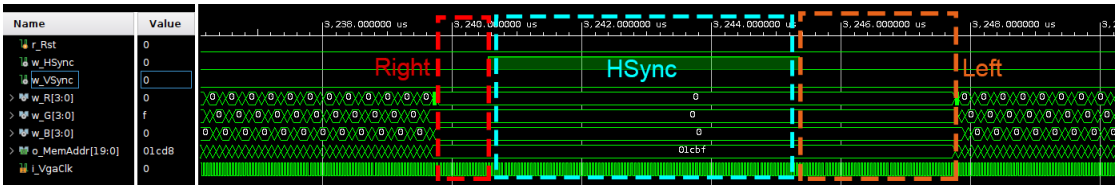

Figure 6.32: Behavioral Simulation, Left and Right borders generation and hsync signal

Afterward, to ensure the proper functioning of the peripheral, the same tests were conducted in Post-Synthesis Simulation.

First the RGB values were verified, and as the images below show, the address value still doesn't change for 2 VGA clock cycles, both horizontally and vertically.

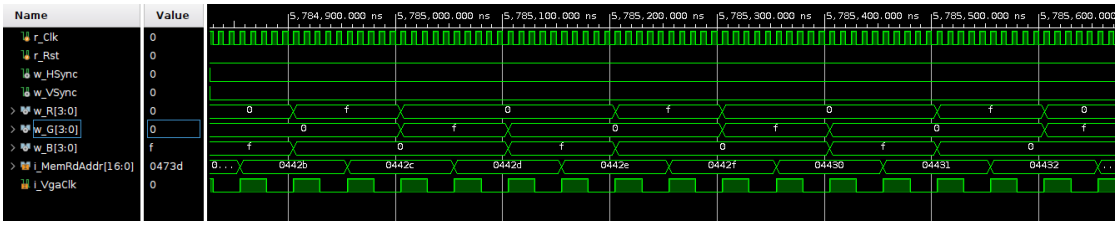

Figure 6.33: Post-Synthesis Simulation, Top and Bottom borders generation and vsync signal

Figure 6.34: Post-Synthesis Simulation, Left and Right borders generation and hsync signal

Then the border generation was verified and once more, as the images bellow show, the top, bottom, left and right borders, and signals hsync and vsync, are being correctly generated/triggered.

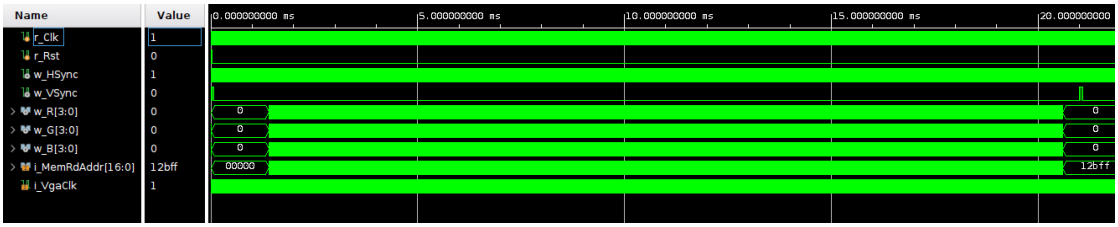

Figure 6.35: Post-Synthesis Simulation, Top and Bottom borders generation and vsync signal

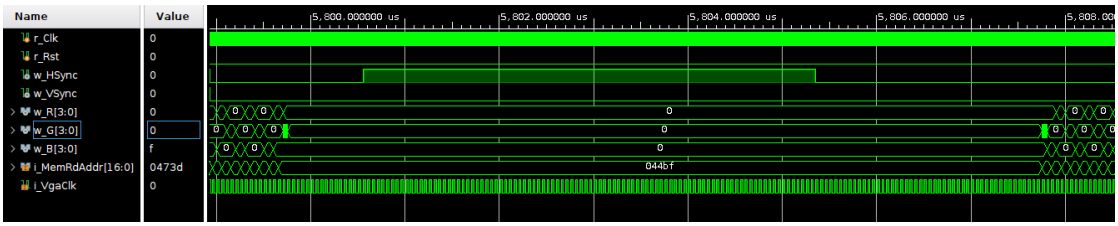

Figure 6.36: Post-Synthesis Simulation, Left and Right borders generation and hsync signal

Chapter 7

Conclusion

To conclude this report, the project's objectives were met, with the pipelined architecture significantly improving performance. The class effectively addressed the constraints and requirements, including integrating various instruction sets and peripheral components within a limited timeframe and resources. This achievement underscores the class capability to navigate complex design considerations and trade-offs in processor design.

The SoC underwent thorough optimization procedures aimed at minimizing the critical path timings, which resulted on it operating at the maximum clock frequency of 125MHz, this concludes by seeing that the longest time of a stage need is 6.930ns. All peripherals were implemented and correctly connected to the CPU. The implemented interconnect did not induce latency, and jumps and branches have a latency of 3 cycles to be executed. One of the major improvements would be to perform jumps during the decode phase and, for branches, to replace the "always not taken" approach with a prediction unit using a suitable algorithm.

Interrupts, despite being one of the biggest challenges, have been successfully implemented and are attended in the clock cycle in which they occur, at any time, without any restrictions. Another significant challenge encountered during the project was addressing the post-synthesis issues, which required careful troubleshooting and modifications.

In summary, the VeSPA project exemplifies how a solid educational foundation, when effectively applied, can lead to the development of sophisticated and functional technology.

Bibliography

[1] David A. Patterson John L. Hennessy. Computer architecture: A quantitative approach. Computer Architecture: A Quantitative Approach - 6th Edition. Accessed on June 8th 2024.

[2] David A. Patterson John L. Hennessy. Computer organization and design. Computer Organization and Design - RISC-V Edition. Accessed on June 8th 2024.

[3] Prof. Dr. Adriano Tavares. Slide:'Processador Dedicado versus Genérico'. Accessed on June 8th 2024.

[4] Prof. Dr. Adriano Tavares. Modelação comportamental (vespa). Slide:'Modelação Comportamental (VeSPA)'. Accessed on June 8th 2024.

[5] Prof. Dr. Adriano Tavares. Pipelining: Fundamentos. Slide:'Pipelining: Fundamentos'. Accessed on June 8th 2024.