**POLITECNICO DI TORINO**

**Master's Degree in Electronic Engineering**

**Master's Degree Thesis**

# Efficient Multi-Processor Interfacing in RISC-V Systems Using Interrupt-Driven Communication and Shared Memory

**Supervisor**

**Prof. Daniele Jahier PAGLIARI**

**Candidate**

**Mustafa Bin TAHIR**

**July 2024**

## Abstract

This thesis investigates the interfacing of two RISC-V processors using interrupt handlers to provide efficient communication and data transfer. The study focuses on developing a method that uses interrupt-driven systems to optimize performance, particularly for IoT applications that require energy efficiency while providing relatively high performance. The thesis begins with an in-depth examination of the RISC-V architecture, including a detailed discussion of the RV32I subset, and progresses to the development of an interfacing methodology that employs shared memory and synchronization mechanisms.

It then shows how a primary low-power RISC-V processor can communicate with a secondary high-power accelerator processor using interrupt signals to manage data transfer through shared memory. The provided SystemVerilog and C code demonstrate the implementation of this methodology, highlighting the roles of interrupt handlers and memory management techniques.

Experimental validation was conducted entirely in a software simulation environment, employing tools such as the Xcelium Logic Simulator. Tests focused on verifying interrupt handling, data transfer accuracy, synchronization, and overall performance metrics. The experimentation, while preliminary, indicated the feasibility and efficiency of the proposed system, demonstrating the potential for reliable communication and high-performance data exchange between the processors.

Key contributions of this thesis include a robust framework for multi-processor interfacing using interrupt-driven communication and shared memory, along with practical insights into memory utilization and data synchronization. The research findings suggest a lot of potential for enhancing the performance and scalability of RISC-V based systems, particularly in resource-constrained environments.

Future work could extend these findings through hardware-based implementations and explore asynchronous processing techniques to further reduce power consumption and improve system responsiveness. This thesis lays a solid foundation for advancing the design and implementation of efficient multi-processor systems in various application domains.

# Acknowledgements

I would like to express my sincere gratitude to my unversity supervisor, Professor Daniele Jahier Pagliari, for his assistance and support in various aspects of my thesis. His contributions have been invaluable and has helped me to complete this thesis successfully.

I would like to extend my heartfelt thanks to my family for their endless support, love, and encouragement. Their constant support and motivation have kept me going during the challenging times.

Finally, I would like to thank all the participants who took part in my thesis and shared their valuable insights and experiences. Without their cooperation, this thesis would not have been possible.

# Table of Contents

# Acronyms

| | |
|---|---|
| **IOT** | Internet of Things |
| **RISC** | Reduced Instruction Set Computing |
| **CPU** | Central Processing Unit |
| **GPU** | Graphics Processing Unit |
| **RV32I** | RISC-V 32 Bit Integer |
| **ISA** | Instruction Set Architecture |
| **CISC** | Complex Instruction Set Computing |
| **IRQ** | Interrupt Request |
| **i/o** | Input/Output |
| **ISR** | Interrupt Service Routine |

**IVT**

Interrupt Vector Table

**CSR**

Control Status Register

**NMI**

Non-Maskable Interrupt

**mepc**

MAchine Exception Program Counter

**MIE**

Machine Interrupt Enable

**MPIE**

Machine Pending Interrupt Enable

**MPP**

Machine Previous Privilege

**mie**

Machine Interrupt Enable

**mip**

Machine Interrupt Pending

**mtvec**

Machine Trap Vector

**WARL**

Write Any Values, Reads Legal Values

**DMA**

Direct Memory Access

**PVT**

Process Voltage Temperature

**REQ**

Request

**ACK**

Acknowledge

**EMI**

Electromagnetic Interference

**PCI**

Peripheral Component Interconnect

**PCIe**

Peripheral Component Interconnect Express

**USB**

Universal Serial Bus

**SATA**

Serial Advanced Technology Attachment

**PATA**

Parallel Advanced Technology Attachment

**CAN**

Controller Area Network

**AMBA**

ARM Advanced Microcontroller Bus Architecture

**SoC**

System on Chip

**AXI**

Advanced eXtensible Interface

# Chapter 1

# Introduction

## 1.1 Motivation

The contemporary computing landscape is marked by a relentless pursuit of efficiency and performance optimization. With the advent of the Internet of Things (IoT), computing systems are confronted with the challenge of operating within strict constraints of power consumption, area utilization, and performance. As IoT applications continue to be adapted across diverse domains, there is an increasing demand for processing power, driving the need for innovative solutions to address these evolving requirements.

At the forefront of this technological evolution is the RISC-V architecture, celebrated for its simplicity, scalability, and open-source nature. Within the IoT context, RISC-V processors offer an attractive proposition, providing a customizable and energy-efficient foundation for embedded systems. However, standalone RISC-V processors may prove insufficient to meet the diverse computational demands of IoT applications.

This thesis delves into the exploration of synchronous RISC-V processors' interfacing, particularly within heterogeneous multi-CPU configurations. The primary processor employed in this study is a straightforward RV32I RISC-V CPU [1], chosen for its simplicity and energy efficiency. Despite its modest capabilities, the RV32I CPU serves as the cornerstone of the system, efficiently handling routine tasks while conserving energy.

In conjunction with the primary CPU, a secondary RV32I RISC-V processor with higher processing capabilities [2] is integrated as an accelerator. This accelerator processor is intended to execute complex and computationally intensive operations that surpass the capabilities of the primary CPU. By delegating such tasks to the accelerator, the overall system performance is improved, facilitating efficient utilization of computational resources.

The term "accelerator" in this context refers to a specialized processing unit engineered to offload specific tasks from the main CPU, thereby enhancing overall system performance. Accelerators are finely tuned for particular workloads, encompassing domains such as signal processing, machine learning inference, or cryptographic operations. By entrusting such tasks to dedicated hardware accelerators, the main CPU can concentrate on its primary functions, leading to heightened efficiency and responsiveness.

Additionally, this thesis delves into the emerging frontier of asynchronous RISC-V processing as a potential avenue for future advancements. In contrast to traditional synchronous designs, asynchronous circuits eschew a global clock signal, relying instead on local timing signals to synchronize individual components. This asynchronous approach offers several advantages, including reduced power consumption, improved noise immunity, and enhanced scalability. However, it also poses challenges such as heightened design complexity and potential timing hazards.

In summary, the thesis aims to develop efficient and adaptable computing systems tailored for IoT applications. By delving into the interfacing of synchronous RISC-V processors and investigating the prospects of asynchronous designs, this thesis aspires to contribute to the ongoing evolution of energy-efficient and high-performance computing solutions.

## 1.2 Thesis Structure

This thesis is structured as follows:

- **Chapter 1** introduces the motivation behind the research and outlines the structure of the thesis.
- **Chapter 2** presents the technical background that will become the basis for this thesis, which includes RISC, while also briefly shedding light on the potential method of communication to be used.
- **Chapter 3** elaborates on the methodology used to interface the two synchronous RISC-V processors, while also detailing the techniques utilized for data transfer and memory sharing.
- **Chapter 4** presents the experimentation and testing approach used to verify the methodology used in the previous chapter, while shedding light on the challenges and further avenues of improvement.
- **Chapter 5** discusses the implications of the findings, including the potential applications of asynchronous RISC-V and future directions for research in this area.
- **Chapter 6** provides a conclusion summarizing the key findings of the thesis and suggesting avenues for further research.

Through this structured approach, this thesis aims to provide valuable insights into the design and implementation of efficient and scalable computing systems based on the RISC-V architecture.

# Chapter 2

# Technical Background

## 2.1 RISC-V Architecture

### 2.1.1 ISA: Instruction Set Architecture

The Instruction Set Architecture (ISA) is defined as the interface between the hardware and software of a computer system, which gives the set of instructions that a processor can execute, how these instructions are encoded, the format of instructions, and the behavior of the processor in response to these instructions. It is basically a bridge between the developers of software, and the digital designers of the hardware, in order to come to terms on a common ground to understand how the processor works and which instructions it can take.

Depending on the complexity of the instructions, data types, number of operands etc., ISA can be of various types Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC).

### 2.1.2 RISC (Reduced Instruction Set Computing)

RISC defines a family of simple ISAs focused on speed. Processors based on this architecture usually feature a small set of instructions, which all performs a single, well-defined task. This reduces the complexity of instructions, which allows them to improve performance, simplify hardware design, and provide efficient pipelining and optimization techniques.

Key features of RISC architectures include fixed-length instruction formats, a large number of general-purpose registers, simple addressing modes, and a load/store architecture where arithmetic and logical operations operate only on data in registers. Examples of RISC architectures include RISC-V, MIPS, SPARC, ARM etc.

### 2.1.3 CISC (Complex Instruction Set Computing)

CISC architecture has a rich instruction set that includes complex instructions that can perform multiple tasks. Therefore, CISC processors normally feature a larger set of instructions. The goal of CISC architectures is minimize the number of instructions during writing complex tasks.

CISC architectures often include instructions that perform multiple operations in a single instruction, such as memory accesses, arithmetic operations, and control flow operations. For the execution, these complex instructions are broken down into smaller and simpler instructions, which are then passed through the execution unit. Examples of CISC architectures include Intel x86, Motorola 68k, and DEC VAX.

### 2.1.4 RISC vs CISC

The comparison between RISC and CISC architectures has been debated for a long time in computer architecture. While RISC architectures prioritize simplicity and efficiency, CISC architectures focus a rich set of complex instructions. Choosing between the two completely depends on factors such as performance requirements, power consumption, instruction set design philosophy, and application domain.

RISC architectures are noramlly preferred in embedded systems, mobile devices, and other resource-constrained environments where power efficiency and performance are critical. In contrast, CISC architectures are used more in desktop and server environments, where compatibility with existing software and legacy systems is important.

Despite their differences, modern processors often have a combination of RISC and CISC techniques, blurring the lines between the two architectures. Many modern processors use a RISC-like core with an instruction decoder that translates the complex CISC instructions into simpler ones for execution.

### 2.1.5 RISC-V Overview

The history of RISC-V traces back to research conducted at the University of California, Berkeley, by Prof. David Patterson, Prof. Krste Asanović, and their students. Beginning in 2010, the RISC-V project aimed to create an open-source ISA to support research and education in computer architecture. The project gained momentum with the formation of the RISC-V Foundation in 2015, further supported by the collaboration among industry and academic institutions to develop and promote the RISC-V ISA.

RISC-V was designed to address the shortcomings of existing ISAs, including proprietary licenses, limited extensibility, and lack of standardization. Since its beginning, RISC-V has witnessed a widespread adoption across various domains, including academia, and industry. Key features of the RISC-V architecture include:

- **Open Standard:** RISC-V is an open standard ISA maintained by the RISC-V Foundation, allowing for widespread adoption and collaboration in the development of hardware and software tools.
- **Modularity:** The architecture is designed in a modular fashion, enabling customization and extension through the addition of optional instruction set extensions (e.g., RV32I, RV64I, RV128I).
- **Simplicity:** RISC-V ISA offers a minimalistic instruction set, comprising a small set of basic instructions optimized for performance and efficiency.
- **Scalability:** RISC-V supports different register widths (e.g., 32-bit, 64-bit, 128-bit) and instruction set variants, allowing for scalability across a wide range of applications and performance requirements.

### 2.1.6 RISC-V RV32I

RISC-V RV32I [3] is a subset of the RISC-V ISA, characterized by its 32-bit register width and integer arithmetic and logic operations. It serves as a foundational instruction set for many RISC-V-based systems, offering a balance between simplicity and performance. Key features of RV32I include:

- **Instruction Set:** RV32I includes a set of basic instructions for integer arithmetic, logical operations, control flow, and memory access.
- **Registers:** RV32I architecture comprises 32 general-purpose registers (GPRs), each 32 bits wide, providing storage for data and intermediate computation results.

- **Load and Store Instructions:** RV32I supports load and store instructions for transferring data between memory and registers.

- **Arithmetic and Logic Instructions:** Basic arithmetic and logical operations, such as addition, subtraction, bitwise AND/OR, and shift operations, are supported.

- **Control Transfer Instructions:** RV32I includes instructions for control flow operations, such as branching and jumping.

- **Immediate Instructions:** Immediate instructions allow for arithmetic and logical operations with immediate values.

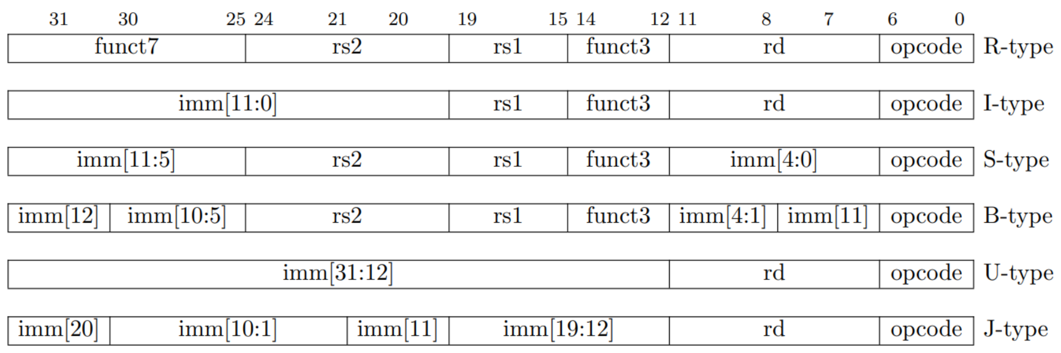

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

**Figure 2.1:** RISC-V base instruction formats showing immediate variants

Figure 2.1 shows the RISC-V base instruction formats. The simplicity of the RV32I instruction set facilitates efficient implementation and optimization, making it a popular choice for embedded systems and IoT devices where resource utilization is critical.

## 2.2 Communication Protocols

A method of back and forth communication is required between the two RISC-V CPUs for the purpose of data and commands transfer. Following is the overview of some possible protocols for this purpose, which will be discussed later in detail in the thesis.

### 2.2.1 Interrupts

Interrupts are essential mechanisms in computer systems for handling asynchronous events and prioritizing tasks. In RISC-V processors, interrupts are managed through a dedicated interrupt controller, which handles interrupt requests (IRQs) from various sources. Key aspects of interrupts in RISC-V systems include:

- **Interrupt Sources:** Interrupts can originate from external devices (e.g., timers, I/O devices), internal events (e.g., exceptions, software interrupts), or other processor cores in multi-core systems.
- **Interrupt Handling:** When an interrupt occurs, the processor suspends its current execution and transfers control to an interrupt handler routine specified by the interrupt vector table. Once the handler completes execution, the processor resumes its previous task.
- **Interrupt Prioritization:** Interrupts may be prioritized based on their urgency and importance, ensuring critical tasks are handled promptly.

Interrupts are used in various applications, including real-time systems, embedded systems, and multitasking operating systems, to handle events such as user inputs, hardware errors, and communication between hardware components.

In our case, these will be the primary method for communication between the two processors, utilized in the form of handshakes. During the Write Operation, when CPU1 wants to send data to CPU2, it will write the data to a designated location in shared memory. CPU1 sets the memory address and data, and then signals the write operation. Once the data is written to the shared memory, CPU1 triggers an interrupt to CPU2.

Similarly during the read Operation, when CPU2 receives the interrupt, CPU2 reads the data from the shared memory. CPU2 sets the memory address and signals the read operation. The data is then read from the shared memory and processed by CPU2.

To summarize, interrupts will be used not to transfer data, but to transfer the control between the two processors, which will then enable transfer of data through a data bus, which will also be discussed.

### 2.2.2 Buses

Buses are communication pathways that enable data transfer between the CPU, memory, peripherals, and other system components. They serve as the backbone of a computer system, aiding in efficient communication and coordination among different components. They can be categorized based on their protocols and characteristics. One major way is through System bus protocols or Peripheral protocols.

System bus protocols are used for communication between the CPU, memory, and other major components within a computer system. These protocols usually offer high bandwidth and low latency to support the performance requirements of the system. For example, AXI (Advanced eXtensible Interface) supports high-bandwidth, low-latency data transfers, making it a good choice for interfacing multiple processors in high-performance SoCs (System on Chip). Another example is AHB (Advanced High-performance Bus), which is used for high-performance modules with a multi-layer shared bus structure.

Peripheral protocols are used to connect external devices such as keyboards, mice, storage devices, and network cards to the motherboard. For example APB (Advanced Peripheral Bus) provides a simplified, low-power protocol for low-bandwidth devices. Peripheral Component Interconnect (PCI) and its advanced version, PCIe, are high-speed interfaces for peripherals, with PCIe using a point-to-point architecture to reduce latency and increase bandwidth. USB (Universal Serial Bus) is a widely used interface for peripherals, supporting plug-and-play and hot-swapping, with versions offering higher data rates and power delivery. SATA (Serial Advanced Technology Attachment) is primarily used for connecting storage devices, offering better performance with a point-to-point architecture.

Buses can also be categorized based on their communication types. Parallel protocols transfer multiple bits simultaneously and are used in older technologies like PATA (Parallel Advanced Technology Attachment) and some earlier versions of PCI, while serial protocols transfer data one bit at a time and are used in modern interfaces like USB, SATA, and PCIe. Additionally, buses can be synchronous or asynchronous. Synchronous protocols, such as AXI and AHB, use a global clock signal, ensuring coordinated operations, whereas asynchronous protocols use handshaking signals, providing greater flexibility and power efficiency.

Furthermore, buses can be categorized based on whether the system is single-master or multi-master. Single-master buses have one master device controlling the communication, which simplifies design but may limit performance. On the other hand, multi-master configurations can improve system robustness and flexibility by allowing multiple masters to connect to the bus, but they may require more complex arbitration mechanisms to avoid conflicts. An example of a multi-master bus is the Controller Area Network (CAN) bus, widely used in automotive and industrial applications.

### 2.2.3 Networks on Chip

Networks on Chip (NoCs) is another method, which can be used for interconnecting components within a chip. However, instead of traditional bus architectures, it uses a network-based approach, which are shared communication pathways, NoCs use multiple interconnected routers and switches to form a network that facilitates communication between components.

In our proposed multi-CPU configuration, the primary RISC-V processor is a low-power CPU, while the secondary processor acts as a high-power accelerator. Incorporating a NoC architecture can facilitate high-speed, low-latency communication between these processors, enhancing the overall system performance and efficiency.

### 2.2.4 Conclusion

Methods like buses and NoCs play a important role in the internal communication of computer systems, allowing various components to exchange data efficiently, the choice of which would impact the overall system performance, scalability, and compatibility with peripheral devices.

While AXI was a compelling option for the communication between two processors, we proceeded to implement a shared memory configuration where the two CPUs are connected to a common memory space in which data can be written by one CPU and read by the other. Hence, making it the actual medium for data transfer between the two processors. For the two-way interfacing, an interrupt handler is utilized for the purpose of handshakes. A data bus is used as the primary data communication method between the CPU and the shared memory. This setup ensures efficient data transfer between the two RISC-V processors by enabling direct access to shared memory.

# Chapter 3

# Methodology for Interfacing RISC-V Processors

## 3.1 Introduction

In this chapter, we will take a deep dive into the methodology used to interface two RISC-V processors. Our configuration, demonstrated in figure 3.1, includes two RISC-V CPUs: a primary low-power processor and a secondary high-power accelerator. Both of these processors are connected to a common memory space where data can be written by one CPU and read by the other. Therefore, this will be the actual medium for the data transfer. For the two-way interfacing, an interrupt handler will be utilized for the purpose of handshakes. After CPU1 sets the memory address and data and signals the write operation, it will trigger an interrupt to CPU2. Similarly, Once the CPU2 receives the interrupt, it will read the data from the shared memory by setting the memory address and signaling the read operation. After the calculations/process is performed, the same process is done except CPU2 and CPU1 replace each other's role. The primary focus is on utilizing an interrupt handler to establish a communication link between these two processors. The approach of offloading complex tasks to the secondary processor results in an efficient configuration from which IoT applications can potentially benefit.

In order to implement the interrupt mechanism in our configuration, we will first study the interrupt structure present in the IBEX processor, which is a 32-bit RISC-V CPU designed for low-power, embedded applications and implements the RV32IMC instruction set. It will be discussed shortly in this chapter before shedding light on the interrupt configuration of our own, tailored to our needs for the linking for the two RV32I CPUs.

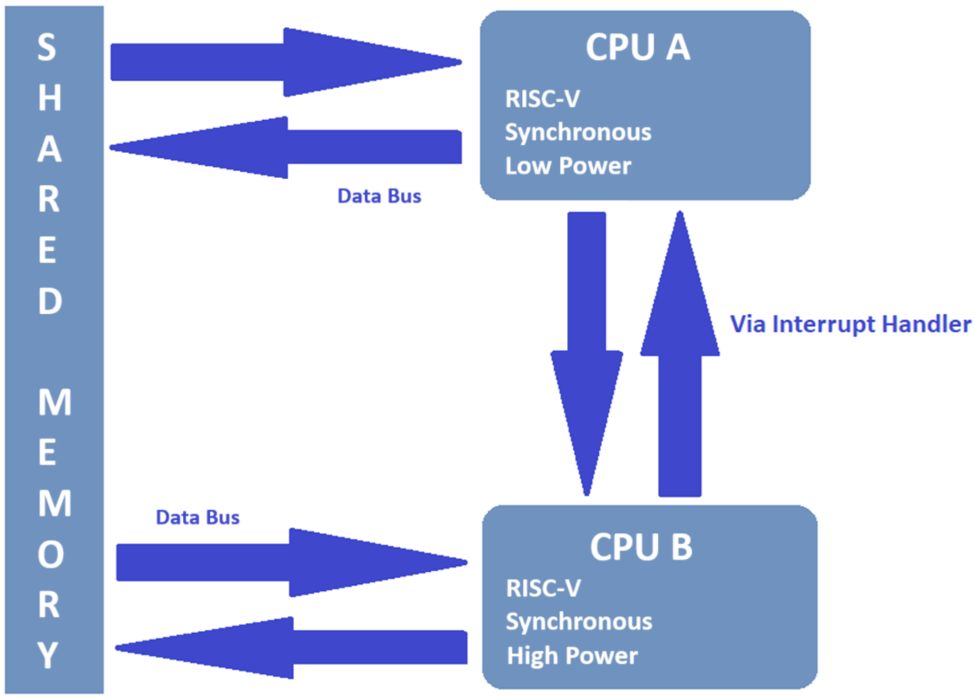


**Figure 3.1:** General Structure

## 3.2 Overview of Interrupts

Interrupts [4] are mechanisms used in computer systems to manage and respond to asynchronous events, allowing processors to handle various tasks efficiently without continuous polling. Generally, an interrupt is basically a signal that is either sent by either hardware or software to the CPU in order to point that there is an event that requires immediate action, which is defined as something that needs attention at the expense of pausing whatever the processor was currently working on.

When such an event occurs, the processor stops working on everything in it's normal routine and directs control to the interrupt handler, also known as Interrupt Service Routine (ISR). The actions performed at this point depend on the instructions that are written inside the Interrupt Handler. Moreover, there is a list of functions that the processor has linked corresponding to handle various operations like exceptions, faults, requests from devices etc. This list is known as the Interrupt Vector Table (IVT). Overall, the role of an interrupt can be summarized into these three parts:

- **Save Current Progress:** Before the control is actually transferred to the interrupt handler, the current progress of whatever the CPU was working on has to be saved. For this purpose, program counter, registers and some other important data is saved somewhere, due to which the CPU can return to this state after performing the operations in the handler.
- **Perform the Task:** The control is transferred to the Interrupt Handler and the instructions in it are followed by the processor. These instructions range from ordinary tasks to operations that are performed in case of an emergency/fatal errors, depending on a case-by-case basis.
- **Load the Saved Progress:** After the interrupt has been processed, the handler restores the state of the processor to the point where the interrupt occurred, based on the information it saved earlier (program counter, register information etc).

In the context of RISC-V processors, interrupt handlers play a crucial role in facilitating inter-processor communication and coordination.

### 3.2.1 Types of Interrupts

There are generally two types of interrupts: Software and Hardware Interrupts, which are discussed as follows:

#### Software Interrupts

When an interrupt is produced by a software (for example an Operating System). Primary examples include traps and exceptions. For example, an exception is normally called by the system when a certain function has to be called to respond to a fatal error. Normally, a software interrupt can be triggered by utilizing an instruction known as “interrupt instructions”, which results in processor stopping it’s current process and switching control to the line of code present in the interrupt handler for general tasks such as error handling.

#### Hardware Interrupts

The other type of interrupt, known as Hardware Interrupt, is the one which is more relevant to our example. It occurs when the trigger is from a hardware device instead of software. The devices in question are connected through an Interrupt Request Line and then, the interrupts can be configured to trigger on either logic 0 or 1. However the more common method is to configure the interrupts for rising/falling edge instead. The overall interrupt is the OR of all the devices in question that are connected to the Line. In our case, the Line will be a simple path between the two RV32I CPUs.

These Interrupts can further be classified into the two types defined below:

- **Maskable Interrupt:** The types of interrupts that can be activated/deactivated by choice, normally because of a mask register in the processor where each bit corresponds to a specific interrupt signal. When a bit in the mask register is enabled, the corresponding interrupt is set. Similarly, the interrupt is disabled when the bit in the mask register changes to 0. With the help of this framework, it can be decided when to ignore or consider an interrupt.
- **Non Maskable Interrupt (NMI):** The type of interrupt that cannot be ignored or masked by the processor. Instead, the NMI overrides the masks and always interrupts the current process as they have the highest priority among all interrupts to ensure they are handled immediately. NMIs are used for critical events that require immediate attention, such as hardware failures, power failures, or other emergency situations.

### 3.2.2 Interrupt Triggers

For an interrupt to be triggered, its input signal needs to be either level or edge sensitive. This depends on the continuity of the request in question, with level-sensitive inputs making constant requests depending on the specific logic level (0 or 1). However, edge-sensitive inputs respond to the rising or falling of a signal edge. Hence, the two types of triggers can be defined as follows:

- **Level Trigger:** In order to request this type of interrupt, the input signal needs to be held at the required logic level (0 or 1). A level-triggered interrupt is triggered when the device gives the input signal and keeps maintains it at the active level.
- **Edge Trigger:** In order to request this type of interrupt, the interrupt signal must undergo a level change (raising or lowering edge). An edge-triggered interrupt is triggered when a signal transitions from low to high (rising edge) or high to low (falling edge). However, dedicated hardware may be needed to

detect the pulses if the I/O is unable to originally do due to the pulse's short duration.

### 3.2.3 Benefits of Using Interrupts

Following are some of the benefits for using interrupts:

- **Real Time Processing:** They allow for real time processing, since the system can respond to the signals or events outside of it's working.
- **Efficiency:** Since interrupts allow the processors to keep performing their normal tasks (until the interrupt is triggered), it allows them to also remain in an idle state if no task is being performed. Without implementing an interrupt mechanism, the processor would likely be required to perform a routinely check for the external condition, making it less energy efficient.
- **Multitasking:** Interrupts allow the processor to perform multiple tasks concurrently, potentially resulting in better performance.
- **Throughput:** Since interrupts are asynchronous, they allow the device to overlap computation with other operations instead of waiting for their turn, which further increases the overall performance of the system.

### 3.2.4 IBEX Processor Overview

The IBEX processor [5], developed by the LowRISC project, is a 32-bit RISC-V CPU designed for low-power, embedded applications. It implements the RV32IMC instruction set, which makes it very suitable for a variety of lightweight computing tasks. The IBEX processor consists of a simple, two-stage pipeline and supports various power-saving modes, making it highly efficient in resource-constrained environments.

#### Interrupt Handling in IBEX

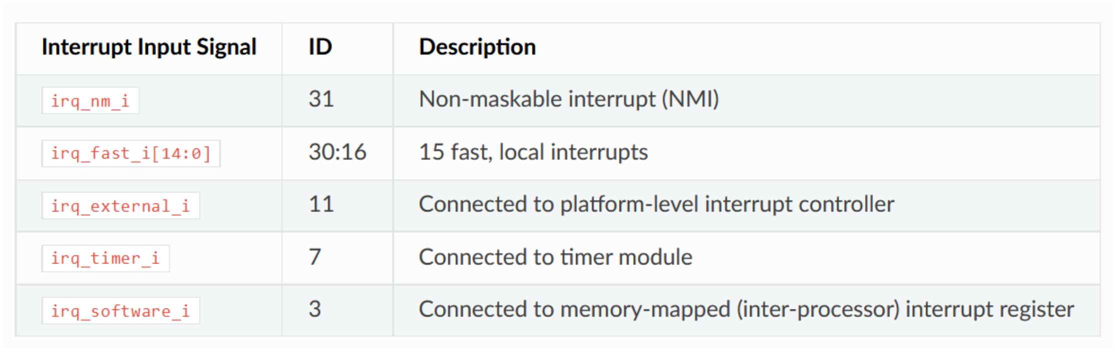
Interrupt handling in the IBEX processor is managed through the RISC-V standard interrupt architecture. The IBEX processor includes a Machine-Level Interrupt Controller (MLIC) that handles multiple interrupt sources and prioritizes them based on predefined criteria. The reason we decided to use the IBEX interrupt as an inspiration was because it follows the same general rules of functioning that we have discussed above. However, it is more complex due to having multiple functionalities that will not be required in our case, which include the variety of control status registers, from which the relevant ones will be discussed shortly. Moreover, we will only use interrupt as handshakes for reading or writing data onto the shared memory, so the interrupt requests for non-maskable interrupts, fast interrupts, timer module etc. will not be required. Some of the features of IBEX interrupt are as follows:

- **Various Interrupt Sources:** The IBEX processor can receive interrupts from external devices, internal timers, and software-generated events.
- **Interrupt Vector Table:** The processor uses an interrupt vector table to map interrupt requests to their corresponding interrupt service routines (ISRs).
- **Machine-Level Interrupts:** The IBEX handles interrupts at the machine level, providing precise control over interrupt priorities which allows for efficient context switching.

When an interrupt occurs, the IBEX processor suspends its current execution and transfers control to the ISR specified in the interrupt vector table. The ISR executes the required tasks and then returns control back to the main program, in order to make sure that minimal disruption to ongoing processes.

### Types of Supported Interrupts

In order to understand the main working of an IBEX interrupt, it is important to mention the types of interrupts that are supported by this processor, which are mentioned in figure 3.2. In case of multiple pending interrupts, they are handled in a pre-defined priority order.



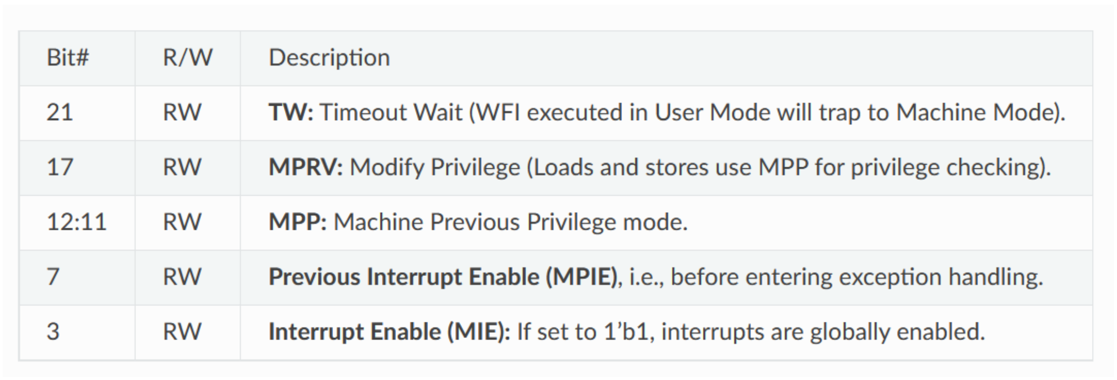
| Interrupt Input Signal | ID | Description |
|---|---|---|
| `irq_nm_i` | 31 | Non-maskable interrupt (NMI) |
| `irq_fast_i[14:0]` | 30:16 | 15 fast, local interrupts |
| `irq_external_i` | 11 | Connected to platform-level interrupt controller |
| `irq_timer_i` | 7 | Connected to timer module |
| `irq_software_i` | 3 | Connected to memory-mapped (inter-processor) interrupt register |

**Figure 3.2:** Supported Interrupt types by IBEX

Fast interrupts have priority over all other types, and among themselves, the interrupt with the lowest ID gets the highest priority. All interrupts except for the non-maskable interrupt (NMI) are controlled via the mstatus, mie and mip Control Status Registers (CSR). After reset, all interrupts are disabled.

### Interrupt Procedure

The CSRs are utilized in the functionality of interrupts. In particular, these are mepc, mstatus, mie, mip, mtvec. Their use-case will be discussed shortly.



| Bit# | R/W | Description |
|---|---|---|
| 21 | RW | **TW:** Timeout Wait (WFI executed in User Mode will trap to Machine Mode). |
| 17 | RW | **MPRV:** Modify Privilege (Loads and stores use MPP for privilege checking). |
| 12:11 | RW | **MPP:** Machine Previous Privilege mode. |
| 7 | RW | **Previous Interrupt Enable (MPIE)**, i.e., before entering exception handling. |
| 3 | RW | **Interrupt Enable (MIE):** If set to 1'b1, interrupts are globally enabled. |

**Figure 3.3:** Bits of the mstatus CSR

The procedure for an interrupt taking place in IBEX is as follows:

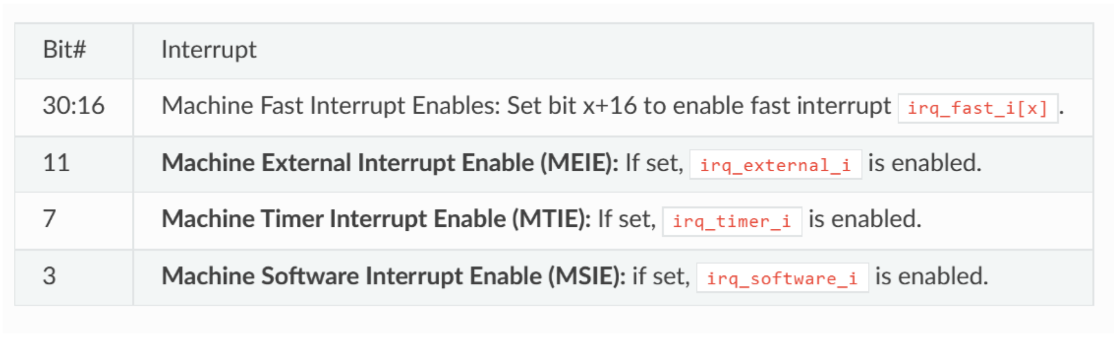
- When an interrupt or an exception is called, the current program counter is storred in a CSR named **mepc (Machine Exception Program Counter)**. This CSR has a label of 0x341.
- The bits of the CSR named **mstatus** (which has a label of 0x300) are adjusted in the following way:

$$mstatus.MPIE = mstatus.MIE$$

  Here, mstatus.MPIE is the status for the previous interrupt enable i.e., Before entering the interrupt. It is the R/W bit 7 of the mstatus CSR. Meanwhile, mstatus.MIE is the current interrupt enable. For example, if it is set to 1'b1, the interrupts become globally enabled. This is the R/W bit 3 of the mstatus CSR. In general, the MIE has to be 1 for interrupts to be allowed, and when the actual interrupt happens, MPIE is then set to 1, which signifies that an interrupt is pending, hence setting MPIE to MIE, which has the value 1 currently.

- More bits of the **mstatus** register are adjusted, which involve the setting of the required privilege mode which are of two types: Machine Mode (M-Mode) and User Mode (U-Mode). The core resets into M-Mode and will jump to M-Mode on any interrupt or exception. The R/W 12:11 bits of the mstatus register, called **mstatus.MPP** (Machine Previous Privilege Mode) will be set to the current privilege mode. On execution of an MRET instruction, the core will return to the Privilege Mode stored in mstatus.MPP.

- Next, the CSR register **mie (Machine Interrupt Enable)** (which has the label x304) is adjusted where the corresponding interrupt enable bit in this CSR needs to be set. This CSR is WARL register that allows to individually enable/disable local interrupts, all of which get disabled after a reset (0x0000_0000). Figure 3.4 shows the types of interrupts supported by mie, which is everything except NMI (Non Maskable Interrupt).

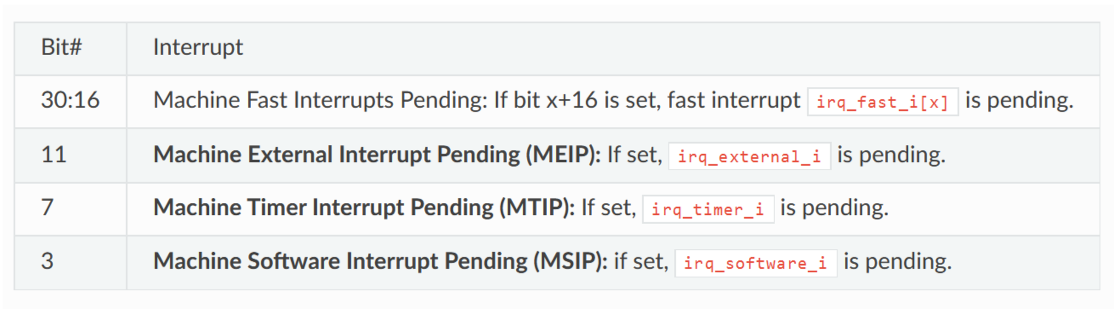


| Bit# | Interrupt |
|---|---|
| 30:16 | Machine Fast Interrupt Enables: Set bit x+16 to enable fast interrupt `irq_fast_i[x]`. |
| 11 | **Machine External Interrupt Enable (MEIE):** If set, `irq_external_i` is enabled. |
| 7 | **Machine Timer Interrupt Enable (MTIE):** If set, `irq_timer_i` is enabled. |
| 3 | **Machine Software Interrupt Enable (MSIE):** if set, `irq_software_i` is enabled. |

**Figure 3.4:** Bits of the mie CSR

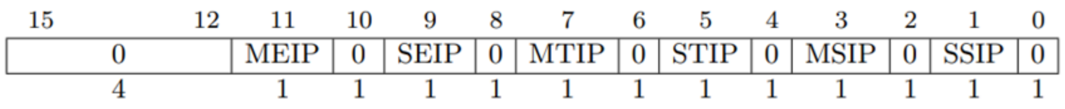
- Adjustments are made to the CSR named **mip (Machine Interrupt Pending)**, which has a label of 0x344. This is a read-only register indicating pending interrupt requests. A particular bit in the register reads as one if the corresponding interrupt input signal is high and if the interrupt is enabled in the mie CSR. The bits of mip CSR are shown in Figure 3.5 and a comparison of bits of mip and mie CSR is given in Figure 3.6.

- Lastly, the core jumps to the base address specified in the mtvec (Machine Trap-Vector Base Address, 0x305) CSR, the bits of which are shown in Figure 3.7. Initially, the base address of the vector table is set to the boot address during the core's booting process. This address must be aligned to 256 bytes,



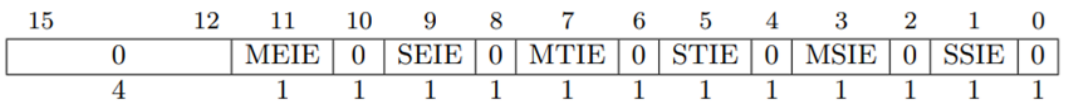
| Bit# | Interrupt |
|---|---|
| 30:16 | Machine Fast Interrupts Pending: If bit x+16 is set, fast interrupt `irq_fast_i[x]` is pending. |
| 11 | **Machine External Interrupt Pending (MEIP):** If set, `irq_external_i` is pending. |
| 7 | **Machine Timer Interrupt Pending (MTIP):** If set, `irq_timer_i` is pending. |
| 3 | **Machine Software Interrupt Pending (MSIP):** if set, `irq_software_i` is pending. |

**Figure 3.5:** Bits of the mip CSR



| 15 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | MEIP | 0 | SEIP | 0 | MTIP | 0 | STIP | 0 | MSIP | 0 | SSIP | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Standard portion (bits 15:0) of **mip**.



| 15 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | MEIE | 0 | SEIE | 0 | MTIE | 0 | STIE | 0 | MSIE | 0 | SSIE | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Standard portion (bits 15:0) of **mie**.

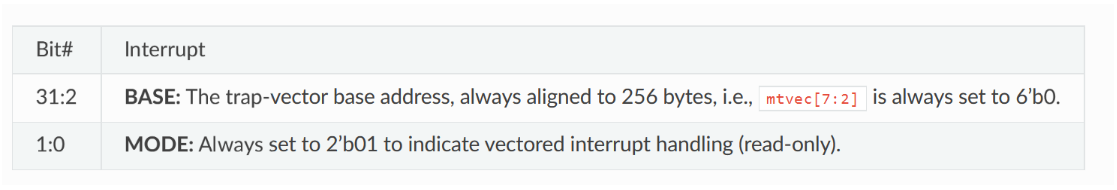
**Figure 3.6:** Comparison of the standard portions for both mie and mip

meaning its least significant byte should be 0x00. The base address can be modified after bootup by writing to the mtvec CSR.

Upon reset, the core starts fetching instructions from an address constructed by concatenating the most significant 3 bytes of the boot address with the reset value (0x80) as the least significant byte. This design assumes that the boot address is provided via a register to minimize signal path lengths to the instruction fetch unit.

The core starts fetching at the address made by concatenating the most significant 3 bytes of the boot address and the reset value (0x80) as the least significant byte. It is assumed that the boot address is supplied via a register to avoid long paths to the instruction fetch unit.

The interrupts are handled in Vector mode, so when an interrupt occurs, the



| Bit# | Interrupt |
|---|---|
| 31:2 | **BASE:** The trap-vector base address, always aligned to 256 bytes, i.e., `mtvec[7:2]` is always set to 6'b0. |
| 1:0 | **MODE:** Always set to 2'b01 to indicate vectored interrupt handling (read-only). |

**Figure 3.7:** Bits of the mtvec CSA

core calculates the address for the interrupt service routine (ISR) using the formula:

$$ISRAddress = mtvec + 4 * InterruptID$$

At this calculated ISR address, there is typically a jump instruction that directs the processor to the actual ISR. This ensures that ach interrupt has a unique and specific address for its ISR. For example, if the base address is 0x80000000 and the interrupt ID is 5, the ISR address will be 0x80000014. The entry at this address will contain a jump instruction to the actual ISR code, such as jmp ISR_5, which could be located at 0x80001000. This process is explained in figure 3.8.

- When the CPU has to return to its normal routine i.e., by executing the MRET instruction, the core jumps to the program counter that was previously saved in the mepc CSR. The mstatus CSR has the following change to it:

  $$mstatus.MIE = mstatus.MPIE$$

  Finally, the privilege mode is restored from the bit **mstatus.MPP**.

- It is to be noted that the **NMI** is enabled independent of the values in the mstatus and mie CSRs, and it is not visible through the **mip** CSR. It has interrupt **ID 31**, i.e., it has the highest priority of all interrupts and the core jumps to the trap-handler base address (in **mtvec**) **plus 0x7C** to handle the NMI. During which all interrupts including the NMI are ignored. Nested NMIs are not supported.

## 3.3 Using Interrupt Handlers for Interfacing

Now, using the above-mentioned IBEX interrupt handling method as reference, we will now Interface two RISC-V CPUs via an interrupt handler in which we set up a communication protocol where the primary processor can signal the secondary

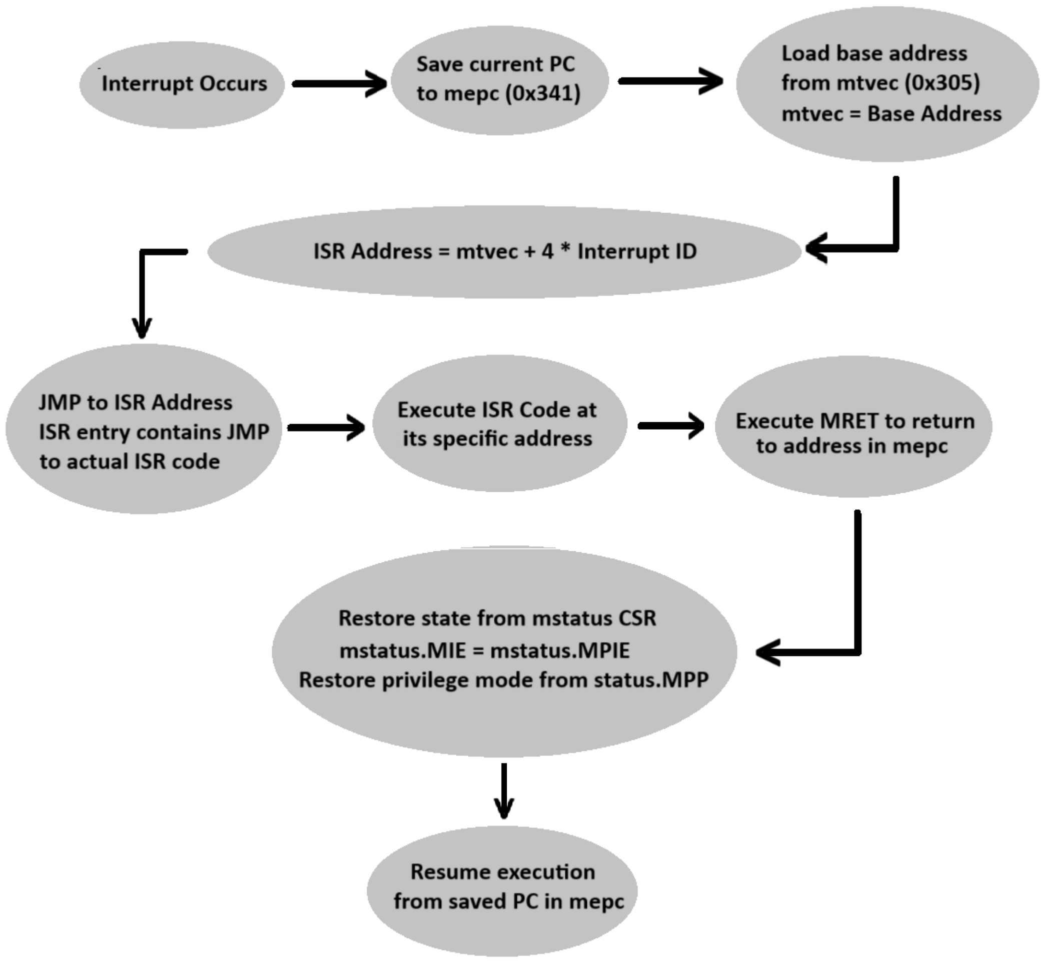

Interrupt Occurs
Save current PC to mepc (0x341)
Load base address from mtvec (0x305) mtvec = Base Address
ISR Address = mtvec + 4 * Interrupt ID
JMP to ISR Address ISR entry contains JMP to actual ISR code
Execute ISR Code at its specific address
Execute MRET to return to address in mepc
Restore state from mstatus CSR mstatus.MIE = mstatus.MPIE Restore privilege mode from status.MPP
Resume execution from saved PC in mepc

**Figure 3.8:** Process flow for handling interrupts in IBEX

processor to perform specific tasks and vice versa. This section outlines the steps and components involved in this process. The hardware aspects will be modeled by the module in SystemVerilog, whereas the Interrupt Service Routine and Main Program Logic for both the CPUs will be handled by the module written in C Language. The steps will be detailed as follows.

### 3.3.1 Setting Up Interrupt Sources

The first step in interfacing the two processors is to configure interrupt sources. These can be hardware interrupts triggered by specific events or software-generated interrupts initiated by the processors. The interrupt sources must be defined in both the primary and secondary processors to enable communication.

### Primary Processor Configuration

- The primary processor is configured to handle and generate interrupts that signal the secondary processor when data is available in the shared memory.
- Each CPU has an interrupt vector table, which maps interrupt IDs to their corresponding interrupt service routines (ISRs). This setup is initialized in the system initialization phase (system_init_cpu1). This is demonstrated by the following code snippet for CPU1. The primary processor's interrupt vector table is initialized during system startup to map interrupt IDs to their corresponding ISRs.

```c
#include <stdint.h>
// Define the addresses of the interrupt controller and shared memory
#define INTERRUPT_PENDING_REG_CPU1  (*(volatile uint32_t *)0x40000000
    )
#define INTERRUPT_CLEAR_REG_CPU1    (*(volatile uint32_t *)0x40000004
    )
#define INTERRUPT_ID_REG_CPU1       (*(volatile uint32_t *)0x40000008
    )
#define SHARED_MEMORY_ADDR          ((volatile uint32_t *)0x50000000)

// Function prototype for the ISR
void ISR_Handler_CPU1(void);

// Interrupt vector table for CPU1
void (*interrupt_vector_table_cpu1[2])(void);

// Function to initialize the system for CPU1
void system_init_cpu1(void) {
    // Initialize the interrupt vector table
    interrupt_vector_table_cpu1[0] = ISR_Handler_CPU1;
    interrupt_vector_table_cpu1[1] = ISR_Handler_CPU1;
}
```

### Secondary Processor Configuration

- The secondary processor is configured similarly to handle interrupts and read data from the shared memory.

- Similarly to the primary processor, the secondary processor's interrupt vector table is initialized during system initialization phase (system_init_cpu2), which maps the interrupt IDs to their corresponding ISRs. This is demonstrated by the following code snippet.

```c
#include <stdint.h>
// Define the addresses of the interrupt controller and shared memory
#define INTERRUPT_PENDING_REG_CPU1  (*(volatile uint32_t *)0x40000000
    )
#define INTERRUPT_CLEAR_REG_CPU1    (*(volatile uint32_t *)0x40000004
    )
#define INTERRUPT_ID_REG_CPU1       (*(volatile uint32_t *)0x40000008
    )
#define SHARED_MEMORY_ADDR          ((volatile uint32_t *)0x50000000)

// Function prototype for the ISR
void ISR_Handler_CPU1(void);

// Interrupt vector table for CPU1
void (*interrupt_vector_table_cpu1[2])(void);

// Function to initialize the system for CPU1
void system_init_cpu1(void) {
    // Initialize the interrupt vector table
    interrupt_vector_table_cpu1[0] = ISR_Handler_CPU1;
    interrupt_vector_table_cpu1[1] = ISR_Handler_CPU1;

}
```

In addition to this, a hardware module has to be defined which facilitates all the above mentioned aspects i.e. internal signals, shared memory interface, flags and the necessary parameters required to support the read and write operations for both CPUs, which will be discussed shortly in the Memory Utilization and Data Transfer Section. The following code snippet demonstrates the general structure of this module.

```systemverilog
module processor_interface (
    input logic clk,
    input logic reset,

    // Interrupt signals
    output logic irq_cpu1_to_cpu2,
    output logic irq_cpu2_to_cpu1,

    // Shared memory interface
    input logic [31:0] cpu1_wdata,
    input logic [31:0] cpu2_wdata,
    output logic [31:0] cpu1_rdata,
    output logic [31:0] cpu2_rdata,
    input logic cpu1_write,
    input logic cpu2_write,
    input logic cpu1_read,
    input logic cpu2_read
);

    // Internal shared memory
    logic [31:0] shared_memory [1:0];
    logic data_ready_flag_cpu1_to_cpu2;
    logic data_ready_flag_cpu2_to_cpu1;

    // Reset logic
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            irq_cpu1_to_cpu2 <= 0;
            irq_cpu2_to_cpu1 <= 0;
            data_ready_flag_cpu1_to_cpu2 <= 0;
            data_ready_flag_cpu2_to_cpu1 <= 0;
        end
    end
endmodule
```

### 3.3.2 Interrupt Service Routine (ISR)

Now, we need to configure the routine that needs to be followed once the interrupt is triggered. In our case, it must perform the necessary actions that are required to manage communication between the two processors. These are described as follows.

#### Primary Processor ISR

When an interrupt occurs, the corresponding ISR is executed based on the interrupt ID. The ISR performs specific tasks, for example reading from or writing to the shared memory. The following code snippet demonstrates a simple example in which read and write operations are performed in the ISR via a case statement.

```
void ISR_Handler_CPU1(void) {
    uint32_t interrupt_id = INTERRUPT_ID_REG_CPU1;

    switch (interrupt_id) {
        case 0:
            // Write data to shared memory atomically
            __atomic_store_n(&SHARED_MEMORY_ADDR[1], 0xCAFEBABE,
    __ATOMIC_SEQ_CST);
            break;
        case 1:
            // Read data from shared memory atomically
            uint32_t data = __atomic_load_n(&SHARED_MEMORY_ADDR[0],
    __ATOMIC_SEQ_CST);
            // Process the data
            break;
        default:
            // Unknown interrupt
            break;
    }

    // Clear the interrupt
    __atomic_store_n(&INTERRUPT_CLEAR_REG_CPU2, 1, __ATOMIC_SEQ_CST);
}
```

### Secondary Processor ISR

Similarly to the primary Processor, we perform the same steps for the CPU2. i.e., When an interrupt occurs, the corresponding ISR is executed based on the interrupt ID. The following code snippet demonstrates a simple example in which read and write operations are performed in the ISR of CPU2.

```
void ISR_Handler_CPU2(void) {
    uint32_t interrupt_id = INTERRUPT_ID_REG_CPU2;

    switch (interrupt_id) {
        case 0:
            // Write data to shared memory atomically
            __atomic_store_n(&SHARED_MEMORY_ADDR[1], 0xCAFEBABE,
    __ATOMIC_SEQ_CST);
            break;
        case 1:
            // Read data from shared memory atomically
            uint32_t data = __atomic_load_n(&SHARED_MEMORY_ADDR[0],
    __ATOMIC_SEQ_CST);
            // Process the data
            break;
        default:
            // Unknown interrupt
            break;
    }

    // Clear the interrupt
    __atomic_store_n(&INTERRUPT_CLEAR_REG_CPU2, 1, __ATOMIC_SEQ_CST);
}
```

### Simulating Hardware Interrupts (ISR)

Furthermore, we also need to configure both CPUs in such a way that they keep on checking for the pending interrupts.

### Primary Processor

In the main loop of each CPU, a function call_isr_cpu1 checks for pending interrupts and invokes the appropriate ISR from the vector table. This is shown in the code snippet below for CPU1 where INTERRUPT_PENDING_REG_CPU1 is the address to check pending interrupts for CPU1 and INTERRUPT_ID_REG_CPU1 is the address to read the interrupt ID for CPU1.

```
// This function simulates the hardware's interrupt call mechanism
void call_isr_cpu1(void) {
    // Check for pending interrupts
    uint32_t pending_interrupt = INTERRUPT_PENDING_REG_CPU1;
    if (pending_interrupt) {
        // Get the interrupt ID
        uint32_t interrupt_id = INTERRUPT_ID_REG_CPU1;
        // Call the corresponding ISR from the vector table
        interrupt_vector_table_cpu1[interrupt_id]();
    }
}
```

### Secondary Processor

Similarly to CPU1, the call_isr_cpu2 function checks for pending interrupts and triggers and invokes the appropriate ISR based on the vector table defined. Here, INTERRUPT_PENDING_REG_CPU2 is the address to check for pending interrupts for CPU2 and INTERRUPT_ID_REG_CPU2 is the address to read the interrupt ID for CPU2. This process is demonstrated by the following code snippet.

```c
// This function simulates the hardware's interrupt call mechanism
void call_isr_cpu2(void) {
    // Check for pending interrupts
    uint32_t pending_interrupt = INTERRUPT_PENDING_REG_CPU2;
    if (pending_interrupt) {
        // Get the interrupt ID
        uint32_t interrupt_id = INTERRUPT_ID_REG_CPU2;
        // Call the corresponding ISR from the vector table
        interrupt_vector_table_cpu2[interrupt_id]();
    }
}
```

## 3.4 Memory Utilization

Memory utilization plays an important role in the interfacing two RISC-V processors via interrupt handlers. Efficient memory management is required so that data is accurately transferred, stored, and retrieved without conflicts or corruption. In this section, we will delve into the key considerations for memory utilization, including memory mapping, synchronization, and access control.

### 3.4.1 Memory Mapping

Memory mapping ensures both CPUs have a common understanding of where the shared memory resides and how to access it. It involves defining specific memory regions for different types of data and operations. In a system with two RISC-V processors, the shared memory must be strategically mapped to ensure efficient data exchange. Considering the above-mentioned code snippets for both CPUs, Following are some of the parameters that contribute to this process.

#### Shared Memory Location

Defined in the SystemVerilog module to keep track of the direction of data flow, given as follows:

- **shared_memory[0]:** Used for data written by CPU1 and read by CPU2.
- **shared_memory[1]:** Used for data written by CPU2 and read by CPU1.

#### Control Registers

Defined in the Initialization of the two CPUs, these registers are used to read, clear and check for pending interrupts. They are listed as follows:

- **INTERRUPT_PENDING_REG_CPU1:** Address to check for pending interrupts for CPU1.
- **INTERRUPT_CLEAR_REG_CPU1:** Address to clear interrupts for CPU1
- **INTERRUPT_ID_REG_CPU1:** Address to read the interrupt ID for CPU1.
- **INTERRUPT_PENDING_REG_CPU2:** Address to check for pending interrupts for CPU2.
- **INTERRUPT_CLEAR_REG_CPU2:** Address to clear interrupts for CPU2
- **INTERRUPT_ID_REG_CPU2:** Address to read the interrupt ID for CPU2.

### 3.4.2 Synchronization

Synchronization is essential to prevent data corruption when both CPUs access shared memory. In our case, we use flags for reading and writing data onto the shared memory. As writing data to shared memory and setting the interrupt flag are two separate operations, it could lead to race conditions where the interrupt is raised before the data is fully written. Similarly, clearing the interrupt flag and reading the data are also separate operations, which can lead to similar issues.

In order to avoid this problem, we use atomic operations to ensure that writing data to shared memory and setting the interrupt flag are performed atomically. Similarly, we ensure that clearing the interrupt flag and reading the data are also atomic operations. This is demonstrated in the code snippets for the ISR handlers of both CPU1 and CPU2. Moreover, In the SystemVerilog processor interface module, we ensure that the setting and clearing of flags along with the data transfer are in the same always_ff block. The blocks for the read/write operations for both CPUs will be discussed in the Data Transfer section.

### 3.4.3 Read/Write Flags

For the purpose of Synchronization, CPU1 sets a flag and generates an interrupt to notify CPU2 when data is written to shared memory. On the other hand, CPU2 reads the data, clears the flag, and then generates an interrupt back to CPU1. These flags are specified as follows:

- **data_ready_flag_cpu1_to_cpu2:** Indicates when CPU1 has written data to shared memory for CPU2.
- **data_ready_flag_cpu2_to_cpu1:** Indicates when CPU2 has written data to shared memory for CPU1.

This infrastructure will also ensure that that only one CPU accesses a specific memory region at a time, preventing race conditions and ensures data integrity.

## 3.5 Data Transfer Mechanism

In a dual-processor system, efficient data transfer mechanisms are critical to ensure smooth communication between the two processors. Various methods can be employed to achieve this, such as direct memory access (DMA), shared memory with flag synchronization, and message passing. This section will focus on these methods, In our case, shared memory with flag synchronization is utilized for the purpose of communicating between CPU1 and CPU2.

A CPU writes data to shared memory, sets a flag to indicate data availability, and generates an interrupt to notify the other CPU. Then, the receiving CPU checks the flag, reads the data from shared memory, processes it, clears the flag, and may generate an interrupt back to signal completion.

### 3.5.1 Direct Memory Access (DMA)

DMA is a technique that allows peripherals or processors to directly read from or write to memory without involving the CPU for each transaction. This method offloads the data transfer work from the CPU, allowing it to perform other tasks while the data transfer is in progress. DMA is particularly useful for large data transfers. DMA is not used utilized in our thesis, but the concept can be applied for more efficient data transfer without involving the CPU.

### 3.5.2 Shared Memory with Flag Synchronization

As discussed earlier, this is the method we implement in our case. Shared memory with flag synchronization involves both processors accessing a common memory space for data exchange. Flags are used to indicate the status of the data (e.g., ready to be read, processed, or written). This method ensures that both processors are synchronized and data integrity is maintained.

### 3.5.3 Memory Passing

Message passing involves sending data encapsulated in messages between processors. This method can be implemented using interrupts to notify the receiving processor of a new message. Message passing is useful for smaller, discrete data transfers and commands.

### 3.5.4 Data Transfer Process

Taking shared memory with flag synchronization and the above mentioned code snippets from interrupt and shared memory into account, the data transfer in our configuration follows the following steps:

#### CPU1 Writing Data

- Writes data to **shared_memory[0]**.
- Sets the **data_ready_flag_cpu1_to_cpu2** flag.
- Generates the interrupt signal **irq_cpu1_to_cpu2**.
- CPU1 writing data onto the shared memory is demonstrated by the following code snippet.

```
    // CPU1 Write Operation
    always_ff @(posedge clk) begin
        if (cpu1_write) begin
            shared_memory[0] <= cpu1_wdata;
            data_ready_flag_cpu1_to_cpu2 <= 1;
            irq_cpu1_to_cpu2 <= 1;
        end
    end

```

### CPU2 Reading Data

- Checks the **data_ready_flag_cpu1_to_cpu2** flag.
- Reads data from **shared_memory[0]**.
- Clears the **data_ready_flag_cpu1_to_cpu2** flag after reading the data.
- Generates the interrupt signal **irq_cpu2_to_cpu1** for starting the next transfer from CPU2 back to CPU1.
- CPU2 Reading data from the shared memory us demonstrated by the following code snippet.

```
    // CPU2 Read Operation
    always_ff @(posedge clk) begin
        if (cpu2_read && data_ready_flag_cpu1_to_cpu2) begin
            cpu2_rdata <= shared_memory[0];
            data_ready_flag_cpu1_to_cpu2 <= 0;
            irq_cpu2_to_cpu1 <= 1;
        end
    end

```

For the communication from CPU2 back to CPU1, similar steps are implemented as above, except shared_memory[1] is utilized instead of shared_memory[0].

## 3.6 Challenges and Considerations

While using interrupt handlers provides numerous benefits, there are also challenges and considerations to address:

- **Complexity:** Designing and debugging interrupt-driven systems can be complex, requiring careful synchronization and coordination.
- **Latency:** Interrupt handling introduces some latency due to context switching and ISR execution, which must be minimized for high-performance applications.
- **Resource Management:** It is important to efficiently manage shared resources and prevent contention so that system stability and performance becomes possible to maintain.

## 3.7 Conclusion

Interfacing RISC-V processors via an interrupt handler provides a powerful and efficient means of managing communication and task offloading. By leveraging the strengths of interrupt-driven systems and efficient memory utilization, it is possible to achieve responsive and scalable multi-processor systems capable of handling complex workloads in a coordinated manner.

In our case, through the above-mentioned sections on configuring interrupt sources, defining ISRs, memory mapping, synchronization, and data transfer mechanisms, along with the provided SystemVerilog and C code, we are able to demonstrate how shared memory with flag synchronization and interrupts can be used for efficient data transfer between two RISC-V processors. By carefully managing memory access and employing interrupt-driven communication, the system ensures reliable and high-performance data exchange.

# Chapter 4

# Experimentation and Testing

## 4.1 Introduction

This chapter outlines the experimentation and testing methodologies employed to validate the interfacing mechanism between two RISC-V CPUs using interrupt handlers, as described in Chapter 3. Here, we aim to demonstrate the feasibility and effectiveness of the proposed approach through a series of carefully designed tests and simulations conducted entirely in a software-based environment.

## 4.2 Experimental Setup

### 4.2.1 Software Environment

The experiments were carried out in a simulation environment to model the behavior of the two IBEX RISC-V processors. The key components of the software environment included:

- **C Language:** Used for initializing the vector tables for both CPUs (for mapping interrupt IDs to corresponding ISRs) and implementing the interrupt service routines executed by the processors.
- **SystemVerilog:** used to define a hardware module which facilitates the internal signals, shared memory interface, flags and the necessary parameters required to support the read and write operations for both CPUs.
- **Simulation Tools:** Xcelium Logic Simulator was used for simulating the hardware design and verifying the behavior of the system.

- **Visual Studio Code** was used for the purpose of writing code and modifying the files.

- **Virtual Platforms:** Cadence Virtual System Platform was utilized to emulate the behavior of the hardware components and processors.

## 4.3 Testing Methodology

### 4.3.1 Objectives

We organize a test plan to cover various scenarios to verify the correctness and efficiency of the interrupt-based interfacing mechanism. The key aims and objectives of this plan are as follows:

- **Interrupt Generation and Handling:** Verify that interrupts are correctly generated by one processor and handled by the other in the simulated environment. Before this step, we first tested primarily the working on CPU1 by arbitrarily generating an interrupt and observing the response to see if it's working as intended. After this, it should be ensured that CPU1 is capable of generating an interrupt and CPU2 is able to handle it.

- **Data Transfer Verification:** Verify that the data transfer through shared memory in the simulation is accurate and without any problems. This involves writing data in CPU1 and trying to read it from CPU2, observing whether the data in both cases is the same or not.

- **Synchronization Mechanism:** Verify that the synchronization via flags is working as intended within the simulation environment. This involves checking the status of each of the flags/registers that were described in the previous chapter. They should all be correctly set and reset during the data transfer process while being in the simulation.

- **Performance Metrics:** After ensuring that everything in the system is working as intended, the next step is to measure latency and throughput of the data transfer process in the software simulation and observe the overall performance of the system. For this purpose, we use some test scripts written in C and passed to CPU1. They are described in more detail in the next section.

### 4.3.2 Test Script

For the Purpose of testing the performance of our configuration, we have organized a test script in which a multiplication loop has to be performed where the CPU2 is used as an accelerator to perform intensive computations while CPU1 handles the coordination and less demanding tasks. CPU1 initiates the computation by signaling CPU2, which performs the entire task as an accelerator. CPU2 handles the computationally intensive workload and signals CPU1 upon completion.

In order to avoid racing conditions, we utilize functions like **atomic_store, atomic_load, and atomic_fetch_add** to perform atomic operations, so that that read-modify-write sequences are not interrupted.

#### Script for CPU1

In the script, we define the number of iterations for the multiplication loop. A flag variable is declared to indicate the status of the interrupt signal, and a pointer to **shared_memory** is declared to facilitate data sharing between CPUs.

The ISR is defined to handle the interrupt signal **SIGUSR1**. When the ISR is triggered, it sets the flag to 1, indicating that the interrupt has been received.

The **CPU1_task** function is the main function for CPU1. Here, the interrupt handler is registered using signal **( SIGUSR1, ISR_handler)**. The Shared memory is allocated using **mmap** to allow both CPUs to access the result.

CPU1 signals CPU2 to start its task using kill **( getpid() + 1, SIGUSR1 )** and then waits for CPU2 to complete the remaining part of the task by checking the flag. Once the flag is set by CPU2, CPU1 reads the final result from the shared memory and prints it. The main function simply calls the **CPU1_task** function to start the process. These steps are demonstrated in the following code snippet.

```c

#define ITERATIONS 10000000
volatile atomic_int flag = 0;
volatile atomic_int *shared_memory;

// Interrupt Service Routine (ISR) for handling interrupt
void ISR_handler(int signum) {
    atomic_store(&flag, 1);
}

// Function to initiate the task and wait for CPU2 to complete
void CPU1_task() {
    // Register the interrupt handler
    signal(SIGUSR1, ISR_handler);

    shared_memory = mmap(NULL, sizeof(atomic_int), PROT_READ |
   PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, −1, 0);
    atomic_store(shared_memory, 0);

    // Signal CPU2 to start the entire task
    kill(getpid() + 1, SIGUSR1);

    // Wait for CPU2 to complete its task
    while (!atomic_load(&flag));

    // Read the result from shared memory
    printf("Final Result: %d\n", atomic_load(shared_memory));
}

int main() {
    CPU1_task();
    return 0;
}
```

**Script for CPU2**

Similar to CPU1, we define the number of iterations for the multiplication loop. Moreover, the flag variable and pointer to **shared_memory** are declared and the ISR is defined to handle the interrupt signal **SIGUSR1**, which sets the flag to 1 when the ISR is triggered.

The CPU2_task function is the main function for CPU2 where shared memory is allocated using **nmap**. The interrupt handler is registered using signal **( SIGUSR1, ISR_handler)**. CPU2 waits for the signal from CPU1 to start the task by checking the flag. Once the flag is set, CPU2 performs the multiplication loop, stated in the workload function i.e., **( workload, shared_memory))**. After its completion, CPU2 signals CPU1 that it is done using kill **( getppid(), SIGUSR1 )**. The main function simply calls the **CPU2_task** function to start the process. These steps are demonstrated in the following code snippet.

```
#define ITERATIONS 10000000
volatile atomic_int flag = 0;
volatile atomic_int *shared_memory;

// Interrupt Service Routine (ISR) for handling interrupt
void ISR_handler(int signum) {
    atomic_store(&flag, 1);
}

// Function to perform the entire workload
void workload(volatile atomic_int *result) {
    for (int i = 0; i < ITERATIONS; ++i) {
        atomic_fetch_add(result, i * i);
    }
}

void CPU2_task() {
    shared_memory = mmap(NULL, sizeof(atomic_int), PROT_READ |
   PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    // Register the interrupt handler
    signal(SIGUSR1, ISR_handler);

    // Wait for the signal from CPU1 to start the task
    while (!atomic_load(&flag));

    // Perform the entire workload
    workload(shared_memory);

    // Signal CPU1 that CPU2 is done
    kill(getppid(), SIGUSR1);
}

int main() {
    CPU2_task();
    return 0;
}
```

## 4.4 Challenges and Observations

### 4.4.1 Challenges

There were some challenges that were encountered while in the experimentation phase, which were primarily related to the testing of the Interrupt handler. Because of the difficulties faced in this stage, the rest of the procedure was unable to be followed. Some of the challenges encountered are as follows:

- **Simulation Accuracy:** Ensuring the simulation accurately models the behavior of the actual hardware components and processors.
- **Timing Issues:** Managing timing issues in the simulation to prevent race conditions and ensure proper timing for interrupt signals.
- **Debugging:** Identifying and resolving issues in the SystemVerilog code and ensuring correct operation of the interrupt handlers within the simulation.

### 4.4.2 Observations

Despite the challenges, some valuable observations were made:

- **Preliminary Validation:** Initial tests in the simulation showed that interrupts were correctly generated and handled, though some timing issues needed resolution.
- **Synchronization:** While not being able to be properly tested, the flag mechanism showed potential for effective synchronization. With additional testing, concurrent scenarios within the virtual environment could also be verified, indicating this method as a viable solution for data transfer between the two processors via shared memory.

## 4.5 Conclusion

This chapter detailed the experimental setup and testing methodology used to validate the interfacing mechanism between two RISC-V CPUs in a software-based simulation environment. While the experiments faced some challenges and the steps were not fully executed, the observations indicated that the proposed approach is feasible. With Further testing and optimization, the desired performance and reliability could be achieved, making the configuration viable for applications where relatively high performance is required while being lower power.

# Chapter 5

# Asynchronous Architecture and its Potential

## 5.1 Introduction to Asynchronous Processors

The majority of devices nowadays utilize synchronous processors at their core. But with time, some development has been made in the asynchronous sector, which represents a paradigm shift from the normal tradition. These processors, also known as clockless or self-timed processors have 1 fundamental key difference: While synchronous processors rely on a global clock signal to coordinate the timing of all operations across the processor, asynchronous processors do not depend on a central clock. Instead, their operations are driven by the completion of preceding tasks. This means that each part of an asynchronous processor proceeds independently based on local conditions and the readiness of data, leading to potentially more efficient and adaptive processing. This key difference can lead to various potential advantages like power efficiency, speed as well as robustness against the Process, Voltage, Temperature (PVT) variations.

In this chapter, we will look into the key components that define an asynchronous processor [6], how it works, and how these factors can be advantageous or disadvantageous compared to synchronous processors.

## 5.2 Key Concepts in Asynchronous Processing

### 5.2.1 Handshake Circuits

Handshake circuits manage communication between different parts of the processor. They ensure that data is transferred only when the sender and receiver are both ready, eliminating the need for a global clock to synchronize these operations.

In order to explain how handshake works, let us first consider a synchronous circuit as shown in figure 5.1. During the process of designing digital circuits of this type, the designers usually focus on the data processing and simply just assume a global clock for their operations. For example, in this figure the data clocked in R3 is obtained through the function CL3, performed at the data from R2 occurring at the previous clock.

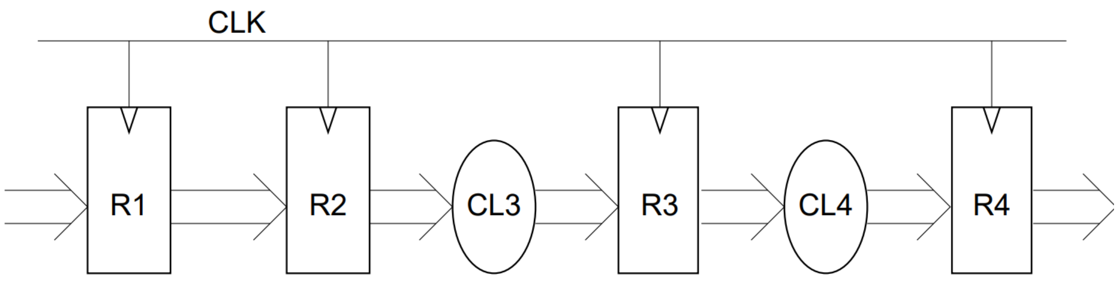


**Figure 5.1:** A Synchronous circuit

Whereas in an asynchronous circuit, a handshaking protocol is used between two registers instead of a clock signal. An example of such a protocol is given in figure 5.2 where a sender has to send data to a receiver but before the actual data transfer, the sender must inform the receiver of their availability (**REQ**), as well as confirming that the information has been received before changing or removing the data. The receiver on the other hand, receives the data and informs the sender of its confirmation (**ACK**).

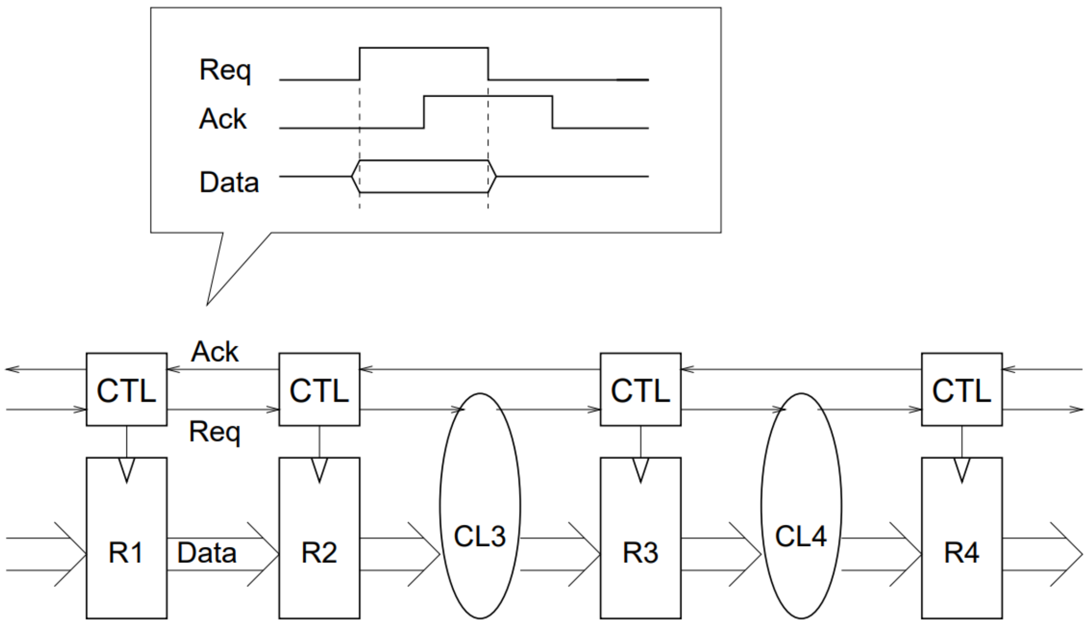


**Figure 5.2:** An Asynchronous circuit

### 5.2.2 Bundled Data Protocol

This is an expansion on the handshake protocol discussed previously, in which the data signals are encoded using normal Boolean levels and separate request and acknowledge wires are bundled alongside the main data path. Figure 5.3 represents a bundled data path.

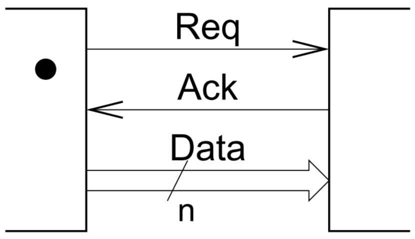


**Figure 5.3:** Bundled Data Path

This bundled data path may be categorized into two further types, which are as follows.

### 4-Phase Bundled Protocol

This protocol consists of 4 distinct phases, which are described as follows:

- When the data is ready, the sender sends the data and issues a request signal (REQ = 1), which means that the availability of signals is conveyed.
- The receiver receives the data and issues an acknowledge signal (ACK = 1), which means that the data has been acquired and now can be safely discarded.
- The sender responds by cancelling the request signal (REQ = 0), which means that the data may no longer be valid.
- The receiver responds by cancelling the acknowledge signal (ACK = 0), which means that the sender can now start a new communication cycle.

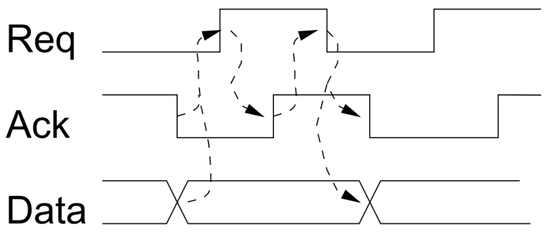


**Figure 5.4:** 4-Phase Bundled Protocol

This 4-step procedure, illustrated in figure 5.4, has a disadvantage which ultimately may result in more consumption of energy and time. This is due to the return-to-0 states which lack any meaningful information.

### 2-Phase Bundled Protocol

The disadvantage of 4-Phased protocol may be solved by the 2-Phased Bundled Protocol, shown in figure 5.5. It is similar to the previously discussed method except that the information on the request and acknowledge paths is now encoded as a signal transition, which results in no difference between (0 to 1) and (1 to 0)

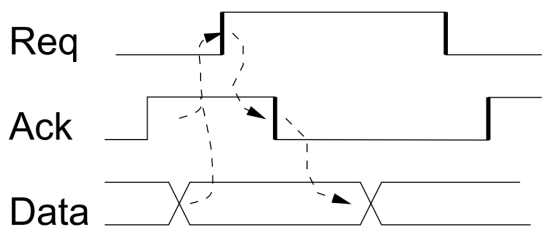


**Figure 5.5:** 2-Phase Bundled Protocol

change. Now, they both represent a "signal transition".

In theory, this solves the issue of time and energy consumption caused by the 4-Phased bundle, but practical implementation has shown that these circuits often end up being more complex, leading to no right or wrong choice between the two protocol and instead depending on a case-by-case basis.

### 5.2.3 The Muller C Element

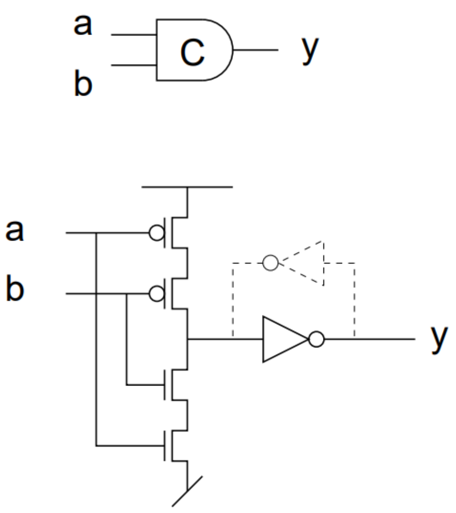
In order to explain the principle of C element, we need to first consider the concept of "indication" or "acknowledgement". For example, we have a simple OR gate with it's truth table (as shown in figure 5.6). Looking at the output of the gate from the truth table, we can extract some useful information. If there is a (1 to 0) transition, we can conclude that both the inputs are 0. Similarly for a (0 to 1) transition, we can conclude that one of the inputs is 1 and the other is 0, although we cannot identify them individually.

Not having the signal transitions indicated or acknowledged in other transitions can lead to hazards. In synchronous circuits, as long as the signal is stable at the rising or falling edge of the clock, the hazards won't functionally cause any problems. However in asynchronous circuits, these hazards should be avoided at all times as there is no global clock and each transition produces an output.

This is the reason why Muller C element (shown in figure 5.7) is critical to the function of such circuits. It is a state holding element and can be described as the asynchronous equivalent of a Set-Reset latch. The output is 0 when both the inputs are 0, and the output is 1 if both the inputs are 1 as well. If there are other input combinations, the output does not change. In terms of the concept of

| a | b | y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Figure 5.6:** Truth Table for OR Gate



Some specifications:

1: if $a = b$ then $y := a$

2: $a = b \mapsto y := a$

3: $y = ab + y(a + b)$

4:

| a | b | y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | no change |
| 1 | 0 | no change |
| 1 | 1 | 1 |

**Figure 5.7:** Muller C Element

“acknowledgement”, the (0 to 1) transition will conclude that both the inputs are 1 and a (1 to 0) transition would mean that both the inputs are 0.

### 5.2.4 The Muller Pipeline

When a bundled protocol utilizes Muller’s C elements in it, it becomes a Muller Pipeline. Each stage of this pipeline consists of a C element and an inverter, in which the input is the **Request (REQ) from the previous stage and an Acknowledge (ACK) from the next stage**. The output of the current stage is **REQ for the next stage and ACK for the previous stage**. A demonstration of this transition is given in figure 5.8.

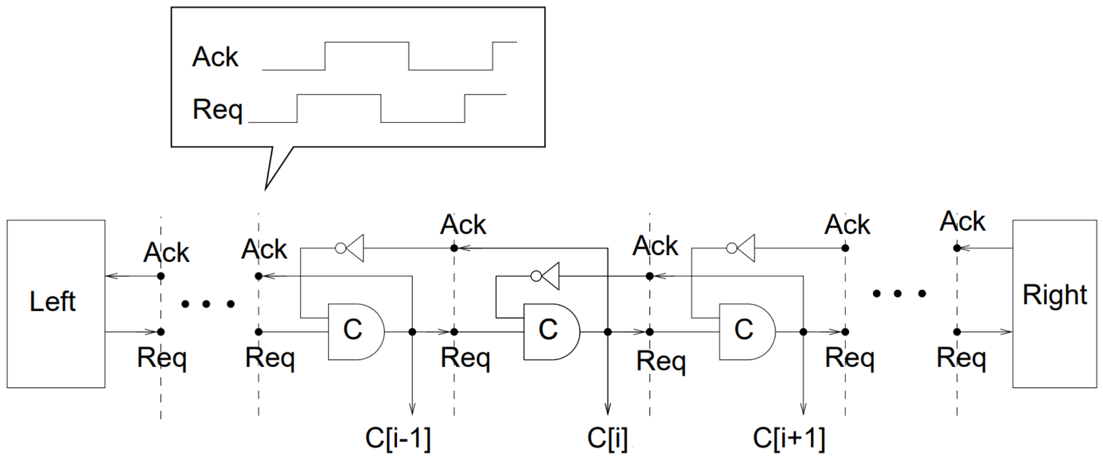


**Figure 5.8:** Muller Pipeline

The direction of propagation of this pipeline is from left to right. If all the C elements are initialized to 0, the request is taken from the left, while the handshake is down to the right, from where they receive the acknowledge signal. It is also worth noting that every C element propagates 0 only if the next state is 1, and similarly propagates 1 if the next state is 0. Therefore, the integrity remains, and no two back-to-back signals can be merged with each other, resulting in the aversion of potential hazards discussed previously.

### The Muller Pipeline (4-phase bundled data)

The 4-phase bundled data pipeline resembles the synchronous model the most as local clock pulses are generated, where the pulses from one stage overlap with the pulse generated in the surrounding stages resulting in a carefully interlocked pipleline. As an example, a FIFO has been considered and demonstrated in figure 5.9 in which combination circuits are added between the latches. Each combination circuit (function block) comes with its own delay. Therefore, in order to retain the correct behavior and timing across the circuit, matching delays also have to be placed on the request signal paths.

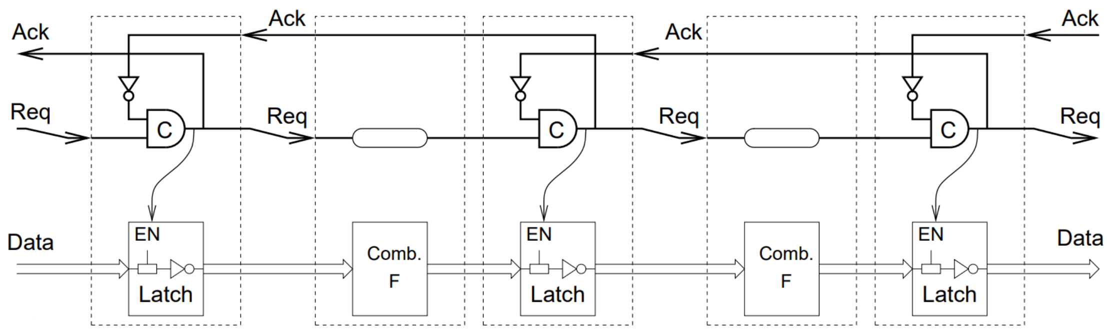


**Figure 5.9:** 4-phase bundled pipeline with function blocks and matching delays

This configuration could be viewed as similar to traditional synchronous data path in the sense that it consist of latches and combinational circuits clocked by a distributed clock-driver. Or it could be also viewed as an asynchronous data path consisting of latches and function blocks as it's handshake components. Either way, this similarity also results in some drawbacks, one of which occurs when the C elements store alternate states consecutively (0, 1, 0, 1,. . . ). This means that it is no different than a traditional master-slave flip-flop with only ever next latch storing meaningful data. Although this condition could be avoided on a case-by-case basis with some custom design.

### The Muller Pipeline (2-phase bundled data)

In order to understand the functioning of the 2-phase bundled data pipeline, the working of a Capture-Pass latch has to be considered first. It is an element used for data storage and is utilized in the pipeline here by being activated alternatively. The switches connected to the input of the latch change according to the capture line while the switches connected to the output of the latch change according to the Pass line.

The working of this latch is demonstrated in figure 5.10. At time t0, the latch is transparent (pass mode) and signals C and P are both 0. The Latch is turned into capture mode when an event on the C line happens. Similarly, an event on the Pass line means that the content of the latch has been used, therefore the latch being transparent again to receive new data.

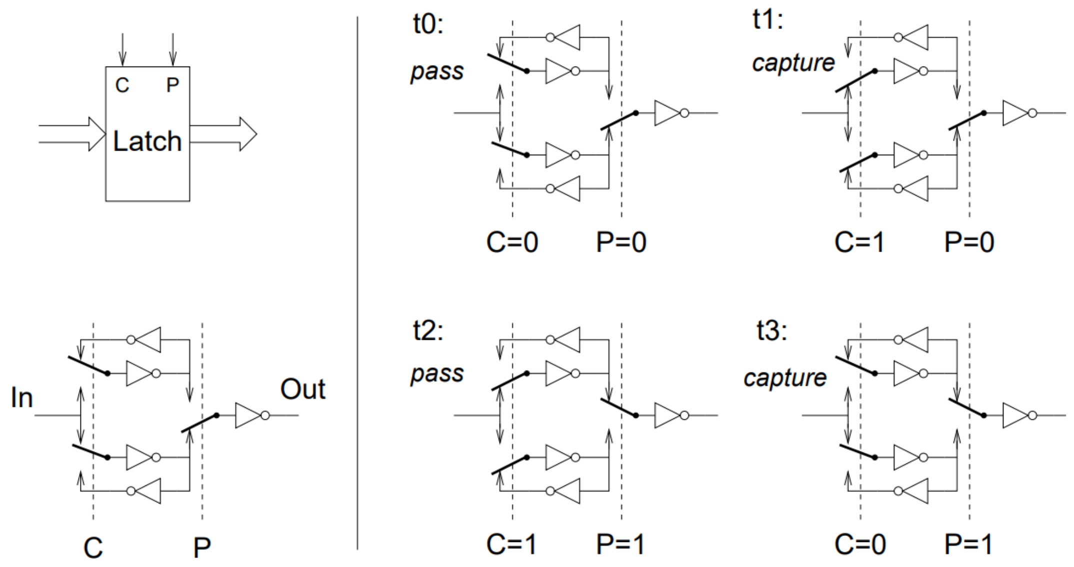


**Figure 5.10:** Working of a Capture Pass Latch

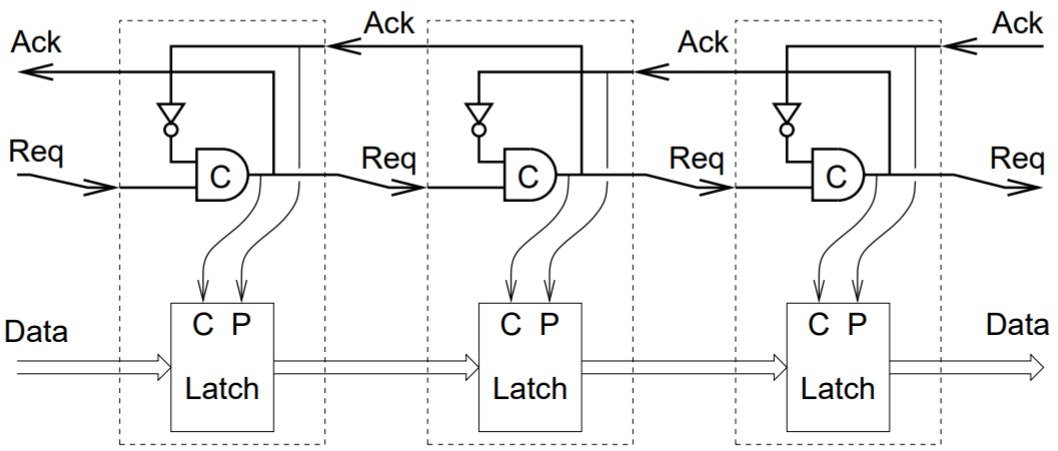


**Figure 5.11:** 2-phase bundled pipeline

Based on this latch, the configuration of a 2-phase bundled pipeline can be formed, as shown in figure 5.11. In theory, this method is more efficient and organized compared to the 4-phase bundled approach as it avoids the power and performance loss caused by the return-to-0 part of the handshaking as discussed previously. However, practical implementation has shown that these circuits often end up being more complex [7], leading to no right or wrong choice between the two protocol and instead depending on a case-by-case basis.

### 5.2.5 Classification of Asynchronous Circuits

In order to explain this, let us consider figure 5.12, which includes 3 gates A, B and C and the output of A is connected to both B and C through a fork, with each of them having their corresponding delay elements. Asynchronous circuits and be classified as follows:

- **Speed independent:** If we assume that components A, B and C have positive but unknown delay (dA, dB, dC), but the wires have 0 delay (d1=d2=d3=0), then this circuit becomes speed independent and is working correctly. Practically speaking, a wire cannot have 0 delay, so if we work under the assumption that d1, d2 and d3 have arbitrary lags and the two branched wires have equal delay (d2=d3), the condition for speed-independence still holds true.
- **Delay-insensitive:** A circuit will become delay-insensitive when both the components and the wires have arbitrary delays, which makes it a lot more robust than a speed independent circuit. However, there are only two circuits that follow up on this condition, which are C elements and inverters. Circuits, where the fork delays of wires d2 and d3 are the same, are known as **Quasi Delay Insensitive**, which are more common. A branch which has equal delays for all it's wires is called **isochronic**.
- **Self-timed:** These are the circuits in which the operation do not follow the basic delay assumptions described above and are instead based on more elaborate timing assumptions.

## 5.3 Use of Asynchronous Processors in the Studied Dual-Processor System

Based on the characteristics and principles of asynchronous processors which have been discussed above, there is room for a lot of development in this aspect. Further research and practical experimentation in this area for the implementation of our multi-processor configuration is highly recommended, for reasons described in relation to the key differences between asynchronous and synchronous processors, as well as the advantages the former has over the latter.

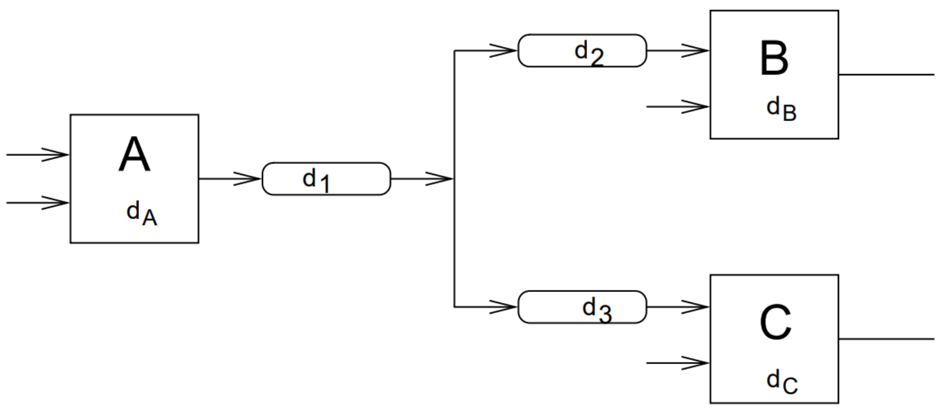


**Figure 5.12:** A circuit consisting of gates and delay elements

### 5.3.1 Differences Between Asynchronous and Synchronous Processors

Understanding the distinctions between asynchronous and synchronous processors is crucial for appreciating the unique benefits they present:

- **Clock Dependency:** Synchronous circuits operate based on a global clock that is used to coordinate all the operations. Which means that all operations, regardless of being major or minor, will have to operate on a fixed timing. Instead, asynchronous processors operate without a global clock while using handshakes and completion signals. This allows for designs that are much more suited and optimized for specific use-cases. The lack of a global clock also reduces the regular switching noise, leading to Reduced Electromagnetic Interference (EMI).
- **Power Consumption:** Synchronous processors consume more power due to a common clock signal which is constant, even when no meaningful work is being done. On the other hand, asynchronous processors can potentially result in improved power efficiency, as they only consume energy when an operation is active.
- **Speed and Performance:** In synchronous processors, the performance may be limited by the slowest operation in the clock cycle. However in case of asynchronous processors, higher performance can be achieved without relying on the wait for slower operations bottlenecking the faster operations.

- **Robustness:** A traditional system with a common clock is more prone to changes from the variations in process, voltage and temperature. As asynchronous processors lack a clock, they are more tolerant to these changes and become much more suitable in areas with fluctuating conditions.

As IoT applications continue to be adapted across diverse domains, there is an increasing demand for processing power, driving the need for innovative solutions to address these evolving requirements. Our proposed multi-CPU configuration aims to deliver on these requirements. However, utilizing asynchronous architecture in the same configuration instead of a traditional synchronous combination could further improve not only the performance, but also the power consumption and robustness of the system allowing for expanded use cases in various environments.

In this context, an asynchronous processor could be integrated either as the main processor or as an accelerator, depending on the specific application requirements. If used as the main processor, the asynchronous architecture would handle general-purpose tasks with increased efficiency and reduced power consumption, particularly benefiting scenarios with variable workloads and dynamic performance needs. On the other hand, if used as an accelerator, the asynchronous processor could be dedicated to handling computationally intensive tasks, such as signal processing, machine learning inference, or cryptographic computations, where its ability to operate independently of a global clock could lead to significant performance gains and energy savings.

By utilizing the strengths of asynchronous processors in these roles, we can enhance the overall system performance, making it more adaptable to the fluctuating demands of IoT applications. This flexibility allows for more robust and energy-efficient solutions, especially in environments where power efficiency and responsiveness are critical.

### 5.3.2 Disadvantages of Asynchronous Processors

Despite their benefits, asynchronous processors are not without its challenges and drawbacks, some of which are mentioned as follows:

- **Design Complexity:** The design and verification of asynchronous circuits are more complex due to the lack of a global clock, requiring specialized knowledge and tools.
- **Limited Tool Support:** Fewer design tools and methodologies are available for asynchronous processors compared to synchronous ones.

- **Compatibility Issues:** Integration with existing synchronous systems and standards can be challenging.
- **Market Adoption:** Lower market adoption and support due to the dominance of synchronous designs in the industry.

### 5.3.3 Summary

For the above mentioned factors, asynchronous architecture is a promising aspect in the application of Internet of Things (IoT). While they have some drawbacks which require further work and support, the potential benefits when it comes to performance, energy efficiency and robustness make them a worthwhile topic for research and further development.

# Chapter 6

# Conclusion

This thesis explored the interfacing of two RISC-V processors, specifically focusing on utilizing interrupt handlers for effective communication and data transfer. The primary aim was to develop a method that uses the strengths of interrupt-driven systems to improve performance, especially in IoT applications requiring energy efficiency and high processing power.

## 6.1 Methodology

The methodology described in Chapter 3 described the methodology to interface a primary low-power RISC-V processor with a secondary high-power accelerator processor using interrupt handlers. The primary processor, a simple RV32I RISC-V CPU, handled routine tasks, while the secondary processor took on more complex operations. The use of shared memory and interrupt-driven communication ensured efficient data transfer and processing synchronization between the two CPUs.

## 6.2 Memory Utilization

A critical aspect of the interfacing methodology was memory utilization. Shared memory was used as the primary medium for data transfer, managed through a series of flags to indicate read/write operations. This setup minimized memory contention and ensured efficient access by both processors. The SystemVerilog and C code provided detailed implementations of the memory management and synchronization mechanisms, showing how shared memory could be effectively utilized in a multi-processor environment.

## 6.3 Data Transfer Mechanism

The data transfer mechanism relied on interrupt signals to initiate the read/write operations in the shared memory space. Once an interrupt was received, the target processor accessed the shared memory, performed the required operations, and updated the synchronization flags. This approach was used to make sure that the data transfer was both timely and accurate, hence reducing the overhead typically associated with continuous polling mechanisms.

## 6.4 Experimentation

Chapter 4 detailed the software-based experimentation and testing of the proposed methodology. The experiments were conducted in a simulation environment, utilizing tools like Xcelium Logic Simulator and SystemVerilog for hardware design and verification. The tests focused on:

- **Interrupt Generation and Handling:** Ensuring that interrupts generated by one processor were correctly handled by the other.
- **Data Transfer Verification:** Validating the accuracy of data transfer through shared memory.
- **Synchronization Mechanism:** Confirming that the synchronization via flags worked correctly within the simulation environment.
- **Performance Metrics:** Measuring the latency and throughput of data transfer, ensuring the system's efficiency.

While results for all the steps were unable to be obtained, the preliminary findings did indicate the feasibility of the interrupt-based interfacing mechanism, and their potential to demonstrate reliable and high-performance data exchange between the processors.

## 6.5 Conclusion and Future Work

Interfacing RISC-V processors via interrupt handlers has proven to be a powerful and efficient method for managing communication and task offloading. The research demonstrated that with careful management of memory access and interrupt-driven communication, it is possible to achieve a responsive and scalable multi-processor system capable of handling complex workloads in a coordinated manner.

Future research could explore hardware-based implementations to further verify these findings. Moreover, investigating asynchronous RISC-V processing could offer new insights into reducing power consumption and improving performance. Developing more sophisticated synchronization mechanisms and exploring other communication protocols could also enhance the robustness and efficiency of multi-processor systems.

In conclusion, this thesis provides insights into the design and implementation of efficient and scalable computing systems based on the RISC-V architecture. The methodologies and findings presented here provide the groundwork for future advancements in the field, particularly in the context of IoT applications and beyond.

# Bibliography

[1] «riscv-simple-sv». In: *URL: https://github.com/tilk/riscv-simple-sv* () (cit. on p. 1).

[2] «Pipelined RISC-V Processor». In: *URL: github.com/estufa-cin-ufpe/RISC-V-Pipeline* () (cit. on p. 1).

[3] A Waterman and K Asanovic. «The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2. May 2017». In: *URL: https://content. riscv. org/wpcontent/uploads/2017/05/riscv-spec-v2* 2 () (cit. on p. 6).

[4] Pedro Mejia-Alvarez, Luis Eduardo Leyva-del-Foyo, and Arnaldo Diaz-Ramirez. *Interrupt Handling Schemes in Operating Systems*. Springer, 2018 (cit. on p. 12).

[5] «Ibex Reference Guide». In: *URL: ibex-core.readthedocs.io/en/latest/index.html* () (cit. on p. 15).

[6] Jens Sparsø. *Introduction to Asynchronous Circuit Design*. DTU Compute, Technical University of Denmark, 2020 (cit. on p. 39).

[7] Jens SparsO, Christian D Nielsen, Lars S Nielsen, Jcrgen Staunstrup, S Furber, and M Edwards. «Design of self-timed multipliers: A comparison». In: *Asynchronous Design Methodologies* 28 (1993), pp. 165–179 (cit. on p. 47).