EDRICO - Educational DHBW RISC-V Core

Semester Paper

from the Course of Studies Electrical Engineering

at the Cooperative State University Baden-Württemberg Ravensburg

by

Levi-Pascal Bohnacker, Noah Wölki

June 2021

Time of Project	31 Weeks
Student ID, Course	6818486, 5040009, TEN18
Reviewer	Prof. Dr. Ralf Gessler

Author's declaration

According to section 1.1.13 of appendix 1 of §§ 3, 4 and 5 in the "Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg" (Version 25.07.2018). We hereby certify that the thesis with the title:

EDRICO - Educational DHBW RISC-V Core

has been composed by us and is based on our own work, unless stated otherwise. No other person's work has been used without due acknowledgment in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged. Furthermore, we assure that the submitted digital version matches the printed version of this thesis.

Friedrichshafen, June 2021

Levi-Pascal Bohnacker, Noah Wölki

Abstract

EDRICO is a project executed by two electrical engineering students at the DHBW Ravensburg. In the very fast evolving technical world, it is important to keep up with new and upcoming technology. RISC-V is a modern open source ISA gaining a lot of popularity not only from the open-source community but also from international companies.

This project combines theoretical knowledge from lectures and research that has to be done during the course of the project with practical work that has to be performed in order to implement the design. The tasks and challenges in this project will result in a deep understanding of FPGA, HDL and RISC-V as well as general computer theory, hardware design and processor technology. Tools to be used are the Vivado design suit for HDL coding and simulation, Github for version control and the V-Modell to provide a project management framework.

A full implementation of the RV32I ISA is performed and fully tested on RTL level. The Zicsr instruction set extension is implemented in order to provide access to necessary control and status registers. EDRICO runs in one of the three available privilege modes: machine-mode. No additional memory ordering models are applied, since no cache, write buffer or out-of-order execution is performed. Memory accesses are secured to a minimum by implementing dedicated physical memory protection registers. Once locked these can only be unlocked by a system reset. EDRICO is shown to execute instructions with an average CPI of 12,07.

Future versions of the proposed design may focus on different improvements, such as security, execution speed or low power consumption.

Contents

Acronyms

AXI	Advanced Extensible Interface
ALU	Arithmetic Logical Unit
CISC	Complex Instruction Set Computer
CPI	Clock Cycles per Instruction
CSR	Control and Status Registers
CPI	Cycles per Instruction
CSR	Control and Status Register
CU	Control Unit
CPI	Cycles per Instructions
PaR	Place and Route
EEMBC	Embedded Microprocessor Benchmark Consortium
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPR	General Purpose Register
HDL	Hardware Description Language
IP	Intellectual Property
ISA	Instruction Set Architecture
KLOC	Kilo Lines of Code
ISR	Interrupt Service Routine
PC	Programm Counter
PMP	Physical Memory Protection
PMA	Physical Memory Attributes
RF	Register File
RISC	Reduced Instruction Set Computer
RV32I	RISC-V 32-Bit Integer
SISD	Single Instruction Single Data
UUT	Unit Under Test
VHDL	Very High Speed Integrated Circuit Hardware Description Language
IP	Interlectual Property
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
IR	Instruction Register
JALR	Jump and Link Register

EDRICO	Educational DHBW RISC-V Core
EDVAC	Electornic Discrete Variable Automatic Computer
ENIAC	Electronic Numerical Integrator and Computer
MIPS	Microprocessor without Interlocked Pipelined Stages
PCIe	Peripheral Component Interconnect Express
CAN	Controller Area Network
ARM	Advanced RISC Machines
AMBA	Advanced Microcontrol Bus Architecture
AXI4-Lite	Advanced eXtensible Interface 4 Lite
AXI	Advanced eXtensible Interface
MPIE	Machine Prior Interrupt Enable
MIE	Machine Interrupt Enable
MPP	Machine Previous Privilege
SPP	Supervisor Previous Privilege
PMP	Physical Memory Protection
PMA	Physical Memory Attributes
TOR	Top of Range
NA4	Natural aligned four-byte region
NAPOT	Naturally aligned power-of-two region
LUT	Look up Table
AMO	Atomic Memory Operation
BRAM	Block RAM
CLINT	Core Local Interrupt
JTAG	Joint Test Action Group
DSP	Digital Signal Processor
LSB	Least Significant Bit
CPU	Central Processing Unit
SoC	System-on-a-Chip

List of Figures

List of Tables

Listings

1 Introduction

Reduced Instruction Set Computer (RISC) based processing units are currently gaining more and more influence on future technologies. Nowadays, Advanced RISC Machines (ARM) cores are found in nearly every embedded device such as smartphones and System-on-a-Chips (SoCs). Since a few years even the long time undisputed x86 architecture is challenged by advances to run personal computers based on RISC devices.
One of the biggest moves towards this trend was carried out by apple when the announced a complete change in their product line-up. The first ARM based macbook was released in 2020, with more to follow. Apple plans to transition their whole product portfolio to in house developed ARM based Central Processing Units (CPUs) in a time frame of about two years [**apple**]. Many other companies like Intel or NVIDIA are showing increased interest in RISC. This can be seen by NVIDIA buying ARM and Intel making offers to buy one of the biggest RISC-V companies, SiFive.
Using ARM cores has multiple downsides, they are extremely cost expensive and could possibly be restricted by export bans. One alternative of ARM is presented in RISC-V, an open source RISC architecture. The microarchitecture is increasingly gaining influence, since its launch in 2010. SiFive started shipping desktop level RISC-V based solutions just recently, displaying the future potential of the RISC-V Instruction Set Architecture (ISA) [**hifive**].
The aim of this project is to develop a 32-bit RISC-V core from scratch. A main objective is for students to gain a deeper understanding of processor and digital design as well as to get familiar with one of the most promising upcoming ISAs of future processors. The project is performed over the course of six months. Its goals are to completely develop, implement and verify the core named Educational DHBW RISC-V Core (EDRICO).

1.1 Motivation

RISC-V was first proposed at Berkeley University in 2010. The architecture is therefore relatively new in comparison to others like x86, ARM or SPARC. Even though its young age is already very promising, every year new breakthroughs are achieved in the field of RISC-V based cores. Micro Magic for example announced a chip with a total CoreMark score of 2500 Coremarks running at 1 GHz and an incredible 250000 Coremarks/Watt in 2021. This poses a significant development, ARMs cortex A 9 processor achieves a maximum Coremarks/Watt score of approximately 1112 [**arma9**]. Many other companies like Alibaba, NVIDIA and SiFive are currently increasing research on RISC-V based cores. The Motivation behind this project was to gain experience in processor and Field Programmable Gate Array (FPGA) design and verification. Furthermore it poses an interesting opportunity for students to work on a new and upcoming processor architecture.

1.2 Project Planning

In order to control the flow of the project, the V-Model (figure 1.1) approach was taken. The project is therefore divided into Requirements, System Design, Architecture Design, Module Design and Implementation. After Implementation the corresponding verification phases are ready to be executed, starting from the lowest level (Unit Verification) to Integration Verification, System Verification and last but not least Acceptance Verification.

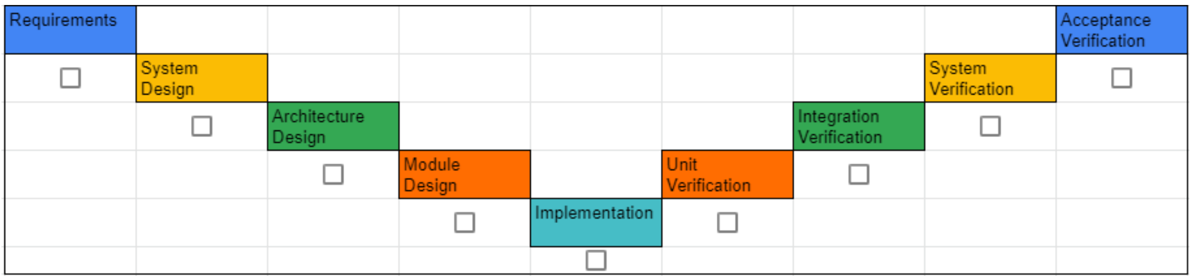


Figure 1.1: V-Model

At the beginning of every project phase, workloads were defined e.g. the definition of the Control Unit Entity. The target of Requirements Engineering was to define everything that is expected from the core and gather information about RISC-V. The data path as well as the entities of the Control Unit, Arithmetic Unit, Register Files, Exception Control, PMP & PMA Checker and AXI4-Lite Interfaces as well as a short summary of their function were defined during System Design. The next step, Architecture Design, aimed to further specify the entities mentioned above and sub-divide them into several entities. Module Design will be executed to define every single architecture, after that implementation and testing may start.

1.3 History

The history of computing is a very complex topic, it goes back to ancient times where mathematicians used simple mechanical devices such as the abacus. An abacus is usually comprised of a frame and multiple rods with pearls mounted on them. Unlike modern computation devices, it does not calculate on its own but is used to provide an overview of the calculations (e.g. the sum and carry) [**ORegan:2021**]. The following section will give a short overview of the history of computational devices.
The Zuse Z1, often referred to as the first computer, was invented and build by Conrad Zuse in 1938. Despite the fact that it was a mechanical computer, the Zuse Z1 was able to perform several arithmetic operations including floating point arithmetic. Later developments of the Z1 are the Z2 and Z3. The Z3 relied on relays and was therefore not fully digital. As the first programmable computer the Z3 marks a milestone in the history of computational devices[**ORegan:2021**].
Developed by the University of Pennsylvania in cooperation with the Ballistics Research Laboratory of the US Army, Electronic Numerical Integrator and Computer (ENIAC) can be seen as one of the first large scale computers (completed in 1946). It was fully digital, using vacuum tubes. Complex computations such as thirty five 10-digit divisions can be carried out in one second using ENIAC. Reprogramming the machine needed a great deal of effort, since it did not employ the stored-program concept. Its successor the Electornic Discrete Variable Automatic Computer (EDVAC) added this functionality to the design [**ORegan:2021**]. A major milestone is marked by the invention of the transistor in 1947 (Bardeen and Brattan, point-contact transistor) and the junction-based transistor in 1951 by Shockley [**ORegan:2021**]. These inventions allowed computers to decrease drastically in size, power and meantime to failure.
With the Intel 4004 the first ever microprocessor was invented in 1971. It was able to run at a speed of 60000 operations per second, providing roughly the same computational power as the ENIAC in 1946 [**ORegan:2021**]. Its successors (the 8008 and 8086) are the base of Intel's current x86 architecture. As one of the first personal computers, the Apple I and its successor the Apple II played a big role in providing more people with access to computational devices. The Apple I was released in 1976, it had to be assembled by the customer[**ORegan:2021**]. From this point on computing power increased drastically, this is achieved by higher clock frequencies, pipelining, parallel processing etc.. Most computer architectures like the x86 instruction set are Complex Instruction Set Computer (CISC) based. One of the first architectures to take a different approach towards simpler but

faster instructions is the Microprocessor without Interlocked Pipelined Stages (MIPS) ISA, more information on this architecture can be found in [**patterson:2020**].
One of the latest developments of RISC based computers is the developement of the RISC-V ISA in 2010.

2 Basics

2.1 Processing Units

Processing Units are the central and most important part of a computer. The processing unit executes the instructions, is responsible for memory and communication and overall execute the program, that is given to the computer. Processing units consist of a Control Unit, Memory, Arithmetic Logical Unit and I/O elements. All of these parts together form a processing unit capable of performing many different instructions, mathematical operations, memory operations and much more. A basic overview of a processing unit is displayed in following graphic:

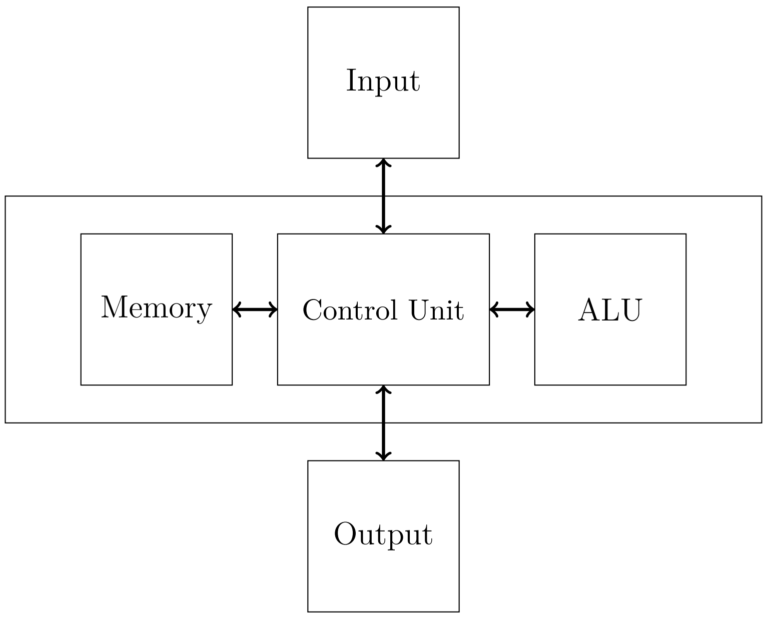


Figure 2.1: Processing unit block diagram

The tasks and functionality of the different parts of a processing unit are described more closely in the dedicated section in chapter 3.
During the long history of computers and processors, different architectures have evolved. This section will give a quick overview of the so called *Von-Neumann-Architecture* and the *Harvard-Architecutre*.
The main difference between those two architectures is the memory access. While a Von-Neumann machine has a commonly used memory for data memory and program memory, a Harvard machine has separately dedicated memory units. This means, a Harvard machine can access to the program memory and the data memory simultaneously. The advantage of a Von-Neumann machine is that there is no limitations on how big the code and the data is. Sometimes, the program takes a lot of memory while the data amount only is very small and sometimes its the other way around. However with a Von-Neumann machine this is not a problem since the memory is used for program and data. In comparison to a Harvard machine, it would be bad to waste a lot of memory space, if the program only takes very little space or the other way around.
However, this advantage of the Von-Neumann architecture is a disadvantage at the same time, since some operations will take much longer than on Harvard machines, because memory cannot be accessed directly after loading an instruction.
In general it can be said, that the Von-Neumann architecture is more simple than the Harvard architecture, but the advantages it originally offered are nowadays solved with several principles and methods using the Harvard architecture. Nowadays, the Harvard architecture is used especially for applications where program size and data size are already known, which often is the case for embedded systems like digital signal processing units etc..
For this project, due to the big complexity of a Harvard architecture, the Von-Neumann architecture is implemented using a shared memory for program and data. [**hellmann2013**]
The block diagrams of the two different architectures are displayed in following figures 2.2, 2.3:

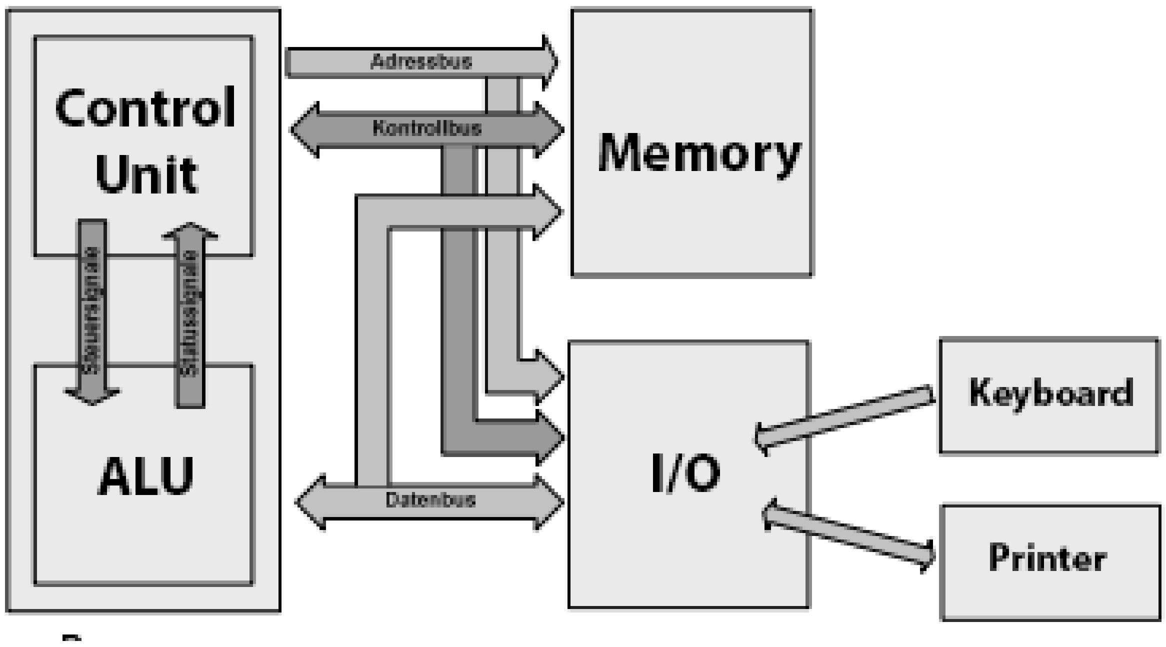


Figure 2.2: Von-Neumann architecture [**hellmann2013**]

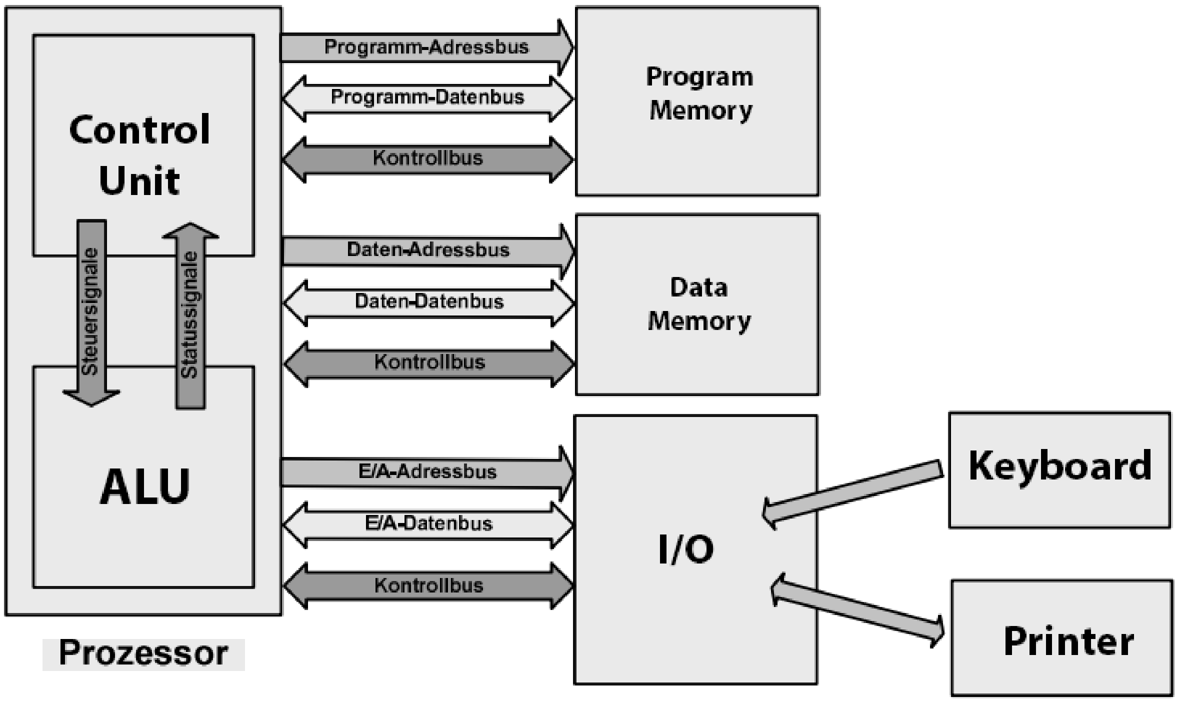


Figure 2.3: Harvard Architecture [**hellmann2013**]

2.2 RISC vs CISC

In the history of processors and computers, speed and efficiency have always been key factors for development and innovation. In the early days of computers, instructions were very simple and straight forward. Coming with time and innovation, instructions and computer architectures became more complex. To prevent instructions from becoming too complex and too big, the Reduced Instruction Set Computer (RISC) architecture was introduced. The main point of RISC is to have a small but highly optimized set of instructions. Another advantage of RISC is a broadly uniform format of instructions and the possibility to establish pipelining which means starting the next instruction while the previous is being executed.
On the other hand, Complex Instruction Set Computer (CISC) machines can have special instructions and more complex instructions to perform many things in one instruction cycle. However, the CPI can get greater than the CPI at for RISC architectures. Since the complexity of instructions increases, the time and computing effort increases as well. In CISC architectures, instructions do not have a standardized format and therefore, can differ in size and complexity. It is also possible to envelope microcode inside of instructions. This means that small pieces of programming can conclude small programs. Since instructions are very individual and not highly optimized, the amount of instructions can get very large. The differences between the two architectures are summarized in table 2.1:

RISC	**CISC**
Single-cycle instructions	Instructions can take several clock cycles
Software-centric design: - High-level compilers take most action	Hardware-centric design: - ISA does as much as possible using hardware circuitry
Simple, standardized instructions	Complex and variable length instructions
One layer instructions	Support microcode
Small number of fixed-length instructions	Large number, variable sized instructions

Table 2.1: RISC vs CISC [**hellmann2013**]

2.3 RISC-V

RISC-V is an open standard Instruction Set Architecture (ISA) developed by the University of California, Berkely. The ISA is based on reduced instruction set computer (RISC) principles. The ISA supports 32, 64 and 128 bit architectures and includes different extensions like Multiplication, Atomic, Floating Point and more [**riscv:unprivileged**]. The ISA is open source and therefore can be used by everyone without licensing issues and high fee requirements. Due to the open source nature of the RISC-V project, many companies like Alibaba and NVIDIA have started to develop hardware based on this ISA. RISC-V opens the opportunity to optimize and configure computer hardware to a level that would not be realizable with licensed ISA like ARM or x86. As a result of this possibility there are many projects and companies working on hardware and software that are beating common CPU in terms of performance and power usage by a lot.
RISC-V is a load store architecture, therefore only load and store operations are able to access the main memory. All other instructions can only operate on register and immediate data. This decreases complexity of the design, if atomic operations are required the A ISA extension can implemented.
Three privilege-levels are supported, making the architecture feasible for every field of operation [**riscv:privileged**]. RISC-V cores can be developed to run small bare-metal implementations as well as full scale operating systems, such as Linux.
One specialty of the ISA is that there is no flag register [**riscv:unprivileged**]. Branches are executed in one instruction by comparing two operands and modifying the Programm Counter (PC) accordingly to the result of the operation.
An additional privileged specification of the ISA introduces so called Control and Status Register (CSR) to the architecture. These are used to control and indicate the status of the core. A status register for example is implemented to reflect the current privilege level in which a hart is running. Other registers provide a framework for exception handling by holding the trap vector and providing registers for trap specific information as well as the jump back address [**riscv:privileged**].
RISC-V is not only comprised off its ISA specification but also multiple open source tools to allow easy workflow. Toolchains such as the gcc compiler are available for RISC-V to allow easy development of code to said platform [**gcc**]. The RISC-V ecosystem is comprised of both, open-source and commercial services. Codasip for example is a company selling RISC-V based Interlectual Property (IP) cores.

2.4 Benchmarks

Benchmarks are measurement methods to evaluate performance of a computer. To measure benchmarks, a testing system is required. This testing environment is often established by using pre defined code or programs. These programs then are compiled and executed. The time this process takes is measured. The goal of a benchmark is to establish a certain comparability between different computers and processing units. [**gessler2014**]
Working with benchmarks, a few principles have to be kept in mind. These vital characteristics of benchmarks are [**kounev2020systems**]:

1. **Relevance**: Only measure relevant features
2. **Representativeness**: The metrics should be broadly accepted by industry and academia
3. **Equity**: fair comparison of all systems
4. **Repeatability**: Verification of results
5. **Cost-effectiveness**: Tests are economical
6. **Scalability**: Tests should work for systems with different range of resources
7. **Transparency**: Metrics should be easy to understand

There are several commonly used benchmarks depending on the type of system to be measured.
A very popular CPU benchmark is the *SPEC CPU*. This benchmark is being released ever since 1998 and gets updated and extended every couple of years. The most recent version is the *SPEC CPU2017*. This benchmark suite concludes a lot of different benchmarks for different use-cases. The suite differs between *rate* and *speed* benchmarks and offers *integer* as well as *floating point* tests. The following table 2.2 gives a brief overview of a few examples and their characteristics (only integer speed benchmark):

Name	**Language**	**Kilo Lines of Code (KLOC)**	**Application**
602.gcc_s	C	1,304	GNU C compiler
625.X264_s	C	96	Video compression
631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search

Table 2.2: SPEC CPU2017 benchmark suite [**kounev2020systems**]

In this thesis, the main focus will be on the so called ⟨*Coremark*⟩ and ⟨*SPECint*⟩ benchmarks.

2.4.1 CoreMark

CoreMark is a openly available benchmark released by the Embedded Microprocessor Benchmark Consortium (EEMBC). The CoreMark benchmark provides a starting point for measuring a processor's core performance. This allows the CoreMark to evaluate a wide range of different devices.
The workload of the CoreMark benchmark contains several algorithms like matrix manipulation, linked list manipulation, state machine operation etc. This mix of operations offers a realistic mixture of load and store, integer and control operations.
The benchmark itself performs following operations for the different benchmarks methods:

1. **Linked List**: Perform multiple find operations (might end up traversing the whole list), sorting using merge sort (based on the value) and then derive a checksum of the data, sort again using merge sort (based on the index)
2. **Matrix Multiply**: 3 matrices A,B,C (NxN size):
    a) Multiply A by a constant into C
    b) Multiply A by column X of B into C
    c) Multiply A by B into C
3. **State Machine**: Perform *switch* and *if* statements using a Moore state machine: parse an input string, extract number → if valid number → return
   Modify input at intervals and invoke state machine on all states

Since CoreMark is an openly available benchmark, some license agreements have to be met and also there is a strict rule for reporting the results of a benchmark. These results have to be published and consist of:

- **N**: Number of iterations per second
- **C**: Compiler version and flags
- **P**: Parameters such as data and code allocation specifics
- **M**: Type of parallel algorithm execution (if used)

2.4.2 SPECint

SPECint is a computer benchmark specification for CPU integer processing power. In this section, the SPECint2006 suite is described in detail. The SPECint2006 suite consists of 12 programs where each is compiled and run three times. The runtimes are measured and the median is used to calculate a runtime ratio. This means that the benchmark compares the measured time to a reference run time. A mathematic aproach for this calculation is given in following formula: $ratio_{program} = \frac{T_{ref}(program)}{T_{SUT}(program)}$ with
$T_{ref}(program)$ = runtime of the specific program on the reference machine
SUT = system under test
$T_{SUT}(program)$ = runtime of the specific program on SUT
Therefore, ratios are higher for faster machines, and lower for slower machines. To determine the whole SPECint2006 score, the geometric mean of all 12 rations is computed. The 12 programs of the SPECint2006 benchmark are displayed in the following table 2.3:

Program	**Language**	**Category**
400.perlbench	C	Perl Programming language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: go playing (complex computer game)
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess playing
462.libquantum	C	Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C	Discrete Event Simulation
473.astar	C	Path-finding Algorithms
483.xalancbmk	C	XML Processing

Table 2.3: SPECint2006 programs [**specint**]

2.5 Memory Management

Memory Management defines how the memory can be accessed and how changes propagate through different levels of the memory hierarchy. The following section will describe what a memory hierarchy is and why it is needed in modern computer designs. Furthermore one particular communication interface or bus system will be introduced which can be used to access memory in an embedded system.

2.5.1 Memory Hierarchy

One of the major performance factors in computing is how fast can information be accessed. Information in this case is both: data and code. Therefore high-speed infinite sized memory would be the optimum to achieve the best possible performance. Unfortunately multiple difficulties are raised by this requirement. How can an infinite sized memory be achieved, how to guarantee that access time does not increase when increasing the memory size and how to keep costs to minimum are only a few of the many questions that can be asked on this topic.
In general it can be said that infinite sized memory is not possible, therefore the new requirement would be: as big as possible. To achieve low access times, the memory must be placed as close as possible to the processor. The amount of extremely fast memory is therefore restricted to a finite size. Increasing the distance to the processor yields more space, hence a bigger possible memory. Access speed is reduced at the same time.
Therefore the only way to have a big high-speed memory is to emulate it. This is done by taking advantage of two principles: temporal- and spatial locality [**patterson:2017**]. Temporal locality suggests that data which is accessed will be used again in the near future. This can be caused by loops inside a routine. Spatial locality is very similar. If data from one particular region is accessed, there is a high probability of an access to the same region in the near future. Many data structures, for example arrays, cause spatial locality.
Figure 2.4 shows an example of a possible memory hierarchy:

The closer a memory element is placed to the processing unit, the faster it gets. This is not only caused by the spatial distance but also by the chosen memory type. For example registers are the closest, followed by SRAM which is typically used for L1,L2 and sometimes even L3 caches [**patterson:2017**].
If a memory access is performed, the memory management system first checks whether or not the desired data is stored in the upper memory levels. A hit occurs when the data block is found in an upper level of the memory hierarchy. In case of a miss, the search for the required data block is continued in the lower levels. If it is found e.g. in main memory,

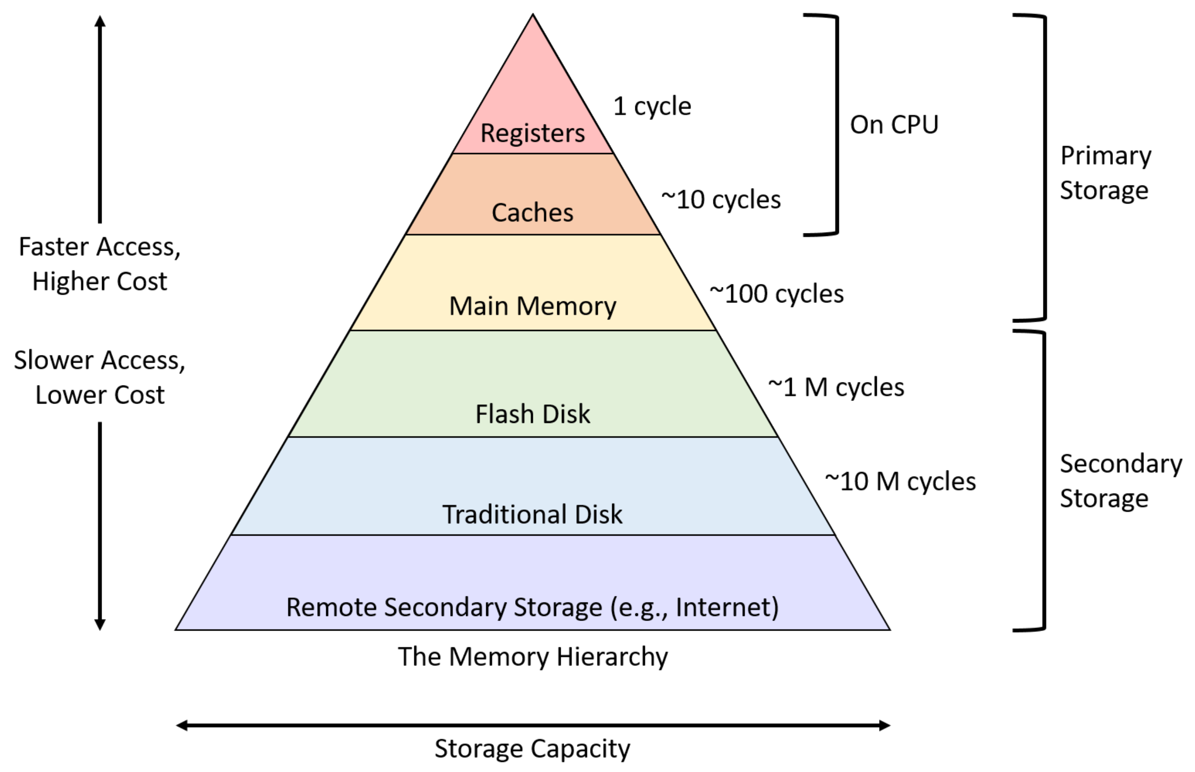


Figure 2.4: Generic Memory Hierarchy [**picture:memoryhierarchy**]

the data will be provided to the processor and copied into the cache. Therefore satisfying the temporal locality principle. Surrounding data blocks are also copied to the cache in order to prevent another miss in case of spatial locality.
Performance of the memory hierarchy can be measured by the hit and miss rate. These describe the portion of miss and hits of the overall memory accesses. Miss rate can be calculated, if the hit rate is known:

$$missrate = (1 - hitrate)$$

To get a significant measurement of the performance, hit time and miss penalty have to be considered as well. The hit time is defined as the time it takes to get the data block from the cache if a hit occurs. After a miss is detected, that block of data has to be retrieved from main memory, stored in the cache and provided to the processor. The overall time required for these three steps is defined as the miss penalty [**patterson:2017**].
Different ways of increasing cache performance are described in [**patterson:2017**].

2.5.2 Communication Interfaces

In order to access memory, some sort of communication interface / bus system is required. There are countless options like the widely used Peripheral Component Interconnect Express (PCIe) and Controller Area Network (CAN) bus or application specific systems, e.g. SpaceWire used in satellite systems or ARMs Advanced Microcontrol Bus Architecture (AMBA).
The following section will provide an overview of the AXI4-Lite protocol which is implemented in the EDRICO.
An AXI interface has four independent channels: read address, write address, read data, write data and write response. Figure 2.5 and 2.6 visualizes these three channels:

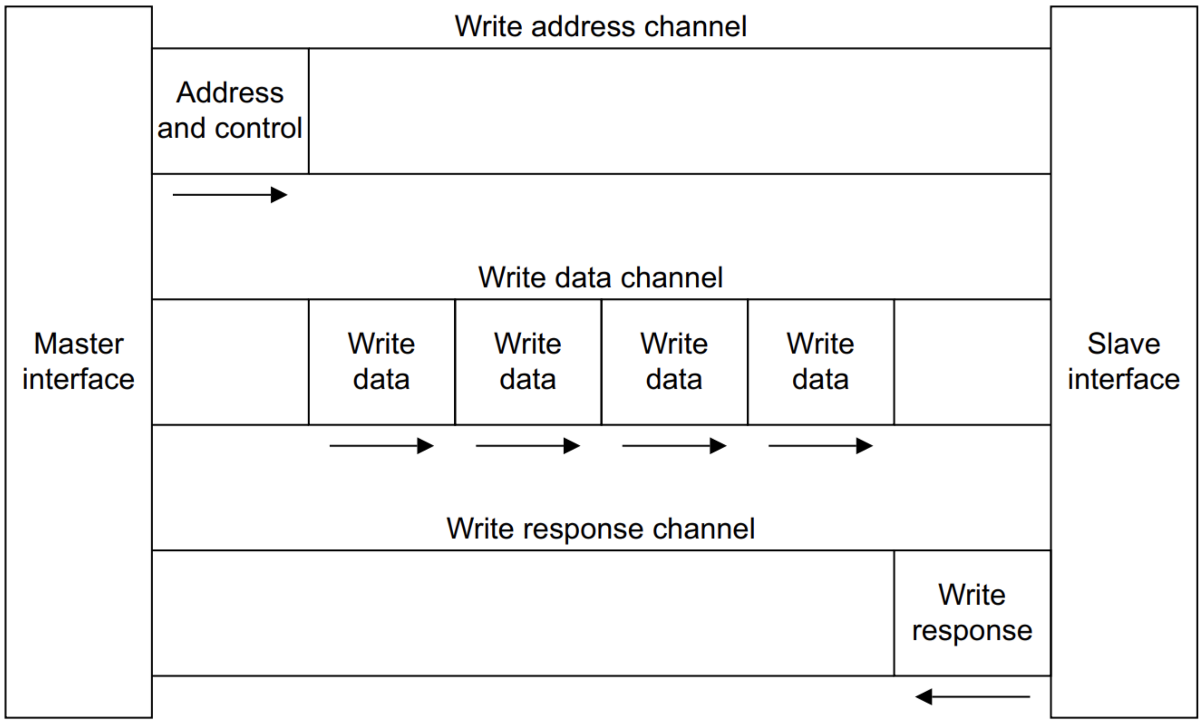


Figure 2.5: AXI4 write channels [**AMBA:AXI**]

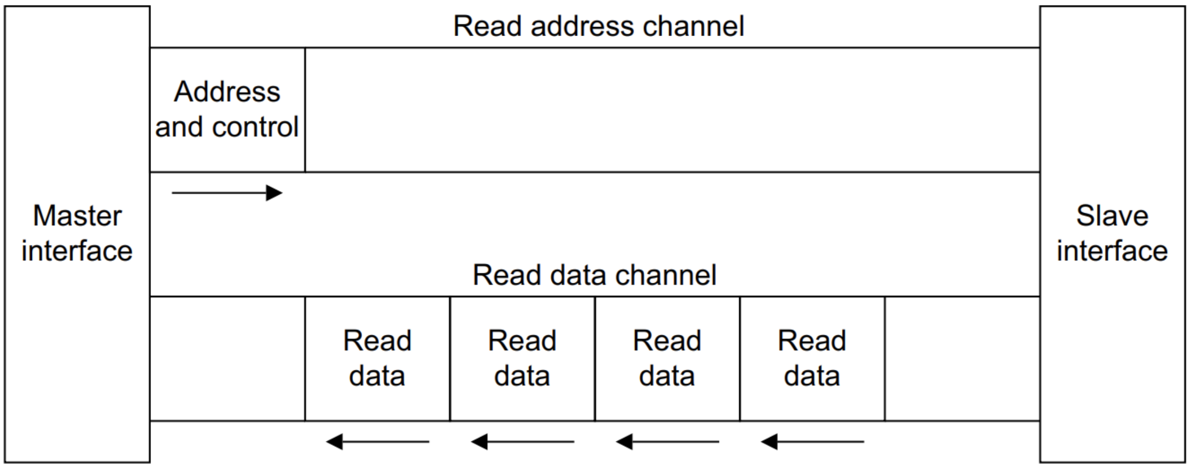


Figure 2.6: AXI4 read channels [**AMBA:AXI**]

The master interface always initiates the transaction by sending either the write or read address and control. In case of a write, the data is send to the slave using the write data channel. A response is given by the slave, containing information on the transfer. The slaves response on a read does not need a separate channel, since it can be transmitted using the read data channel.
AXI4-Lite does not support burst transfers, hence only one data word can be transmitted in one transfer. To transfer any data over any channel, a generic handshake process must be completed. Figure 2.7 shows how the handshake is performed:

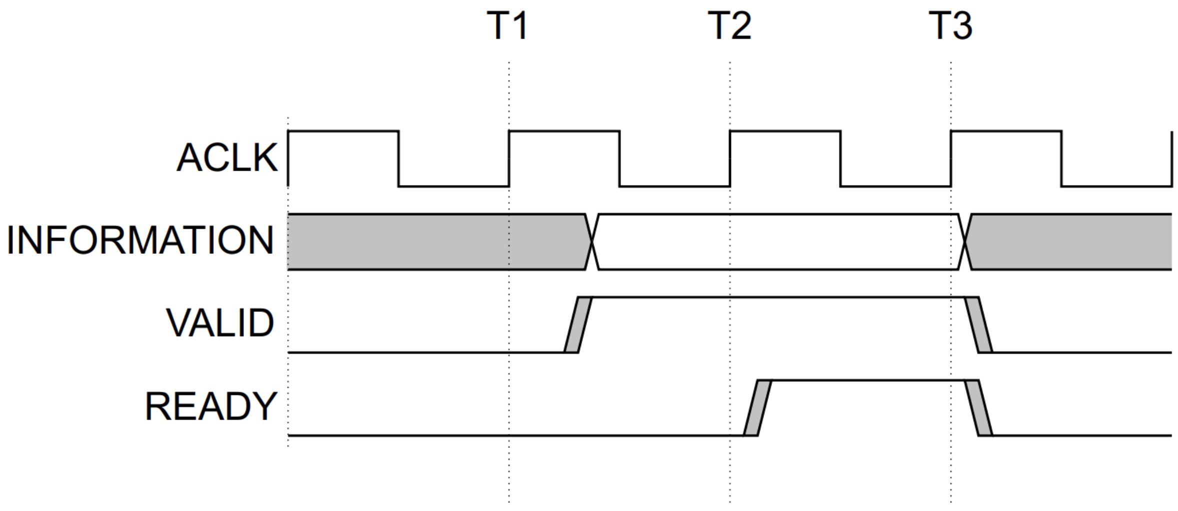


Figure 2.7: AXI handshake [**AMBA:AXI**]

At transfer begin, the sending interface applies the information (e.g. the read address) on the bus. The Valid signal indicates that the information is valid, it must stay valid until the receiving interface applies the ready signal. Ready indicates that the receiving partner

is ready to receive the information. If both signals (valid and ready) are high on a rising clock edge, the information is read by the receiver and the flow control signals are tied to low [**AMBA:AXI**]. This handshake mechanism allows the receiving AXI interface to extend the length of the transfer when needed. Since each of the five AXI4 channels are independent five handshake mechanisms are implemented.
A response of a slave contains the *RRESP* or *BRESP* signals, respectively. They can be set to OKAY, EXOKAY, SLVERR, and DECERR. In case of an okay or exclusive okay, no error has occurred. SLVERR indicates that an error has occurred on the slave side, even though the slave successfully registered the access. If no slave is available on the interrogated address, a DECERR is returned.

Based on the response of the slave, a masters behavior must adapt. If (EX)OKAY is returned it may proceed normal operation. If an error is detected error handling methods such as exceptions must be triggered.
The AMBA protocols are widely used in embedded systems. Many IPs deployed in FPGAs can be interfaced using the AXI4 or AXI4-Lite bus. Therefore it is mandatory to be familiar with this particular bus architecture when working with embedded systems.

2.6 FPGA

To verify a digital circuit software simulations as well as implementing the design on a prototype are common practice. For prototyping and even implementing a finished product, FPGAs are widely used. FPGAs are special fine granularity programmable logic devices. The digital logic can be described using hardware description languages such as Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL). These designs are then synthesized, placed and routed in order to generate a hardware configuration file, also called bitstream. The bitstream can then be loaded onto the FPGA via a programming interface e.g. Joint Test Action Group (JTAG). Many different vendors produce FPGAs, the most famous ones are Xilinx, Altera/Intel and Microchip. Some smaller vendors like NanoXplore produce FPGAs targeting rare use cases like space applications. Despite the many differences in design, the basic architecture always remains the same. An array of logic cells and building blocks of different features like Block RAM (BRAM) and Digital Signal Processor (DSP) slices are connected to each other through configurable routing channels [**kesel:2018**]. Figure 2.8 shows the basic architecture of a Xilinx FPGA:

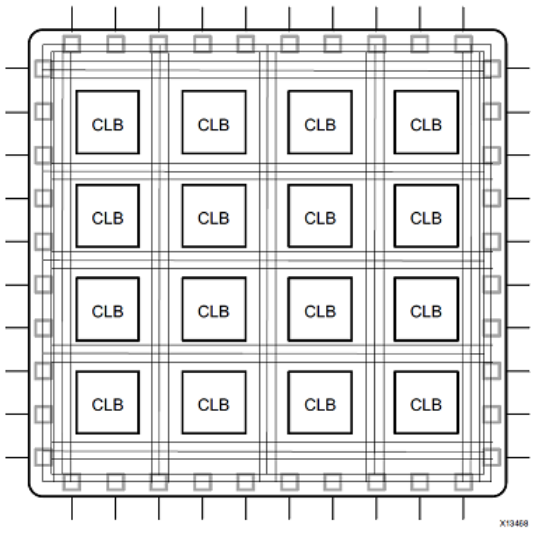


Figure 2.8: Xilinx FPGA [**xilinx:2017**]

2.7 Hardware Description Languages

A Hardware Description Language (HDL) is a computer language specialized to describe structure and behavior of electronic circuits. HDLs were created to implement register-transfer level abstraction which is a model of the data flow and timing of a circuit.
HDL can be used to express designs in structural, register-transfer-lever or behavioral architectures for a specific functionality.
VHDL is a description language used to describe hardware. It is utilized in electronic design to express digital systems such as integrated circuits and FPGA. VHDL is also used as a general-purpose parallel programming language. VHDL is used to write text models that describe or express logic circuits. If the text model is part of the logic design, the model is processed by a synthesis program. Implementation and testing of VHDL is described more closely in following chapters. The critical advantage of VHDL, with regard to system design utilization is that it permits the behavior of the essential system to be verified and modeled in advance of the synthesis tools translation of the design into actual hardware. In addition, it is a dataflow language, which means it can simultaneously consider every statement for execution other than procedural computing languages like C. As a last advantage to be mentioned here, VHDL projects can be multipurpose. It is possible to create units and blocks and re use them for other applications.
Another HDL on the market is Verilog. Verilog is a more compact language and often results in less lines of code and is more similar to programming languages like C. However, Verilog is not as wordy as VHDL, which accounts for its compact nature. Verilog also is more of a hardware modeling language and not so natural in use as VHDL. Nonetheless, Verilog is often considered as easier to learn.

3 EDRICO

The Proposed Processor design named Educational DHBW RISC-V Core (EDRICO) implements a basic RISC-V 32-Bit Integer (RV32I) instruction set architecture. Besides the mandatory Zicsr extension no other instruction set extensions are implemented. To keep the implementation simple and straight-forward only one privilege mode (machine-mode) is implemented. This mode allows full access to the processor and peripherals. Future versions could be extended to implement supervisor- and user-mode.
The core is a simple Single Instruction Single Data (SISD) processor without any pipeline or cache. The basic instruction cycle of fetch, decode, execute is performed for every instruction one at a time.
Figure 3.1 shows the full overview of the processor design:

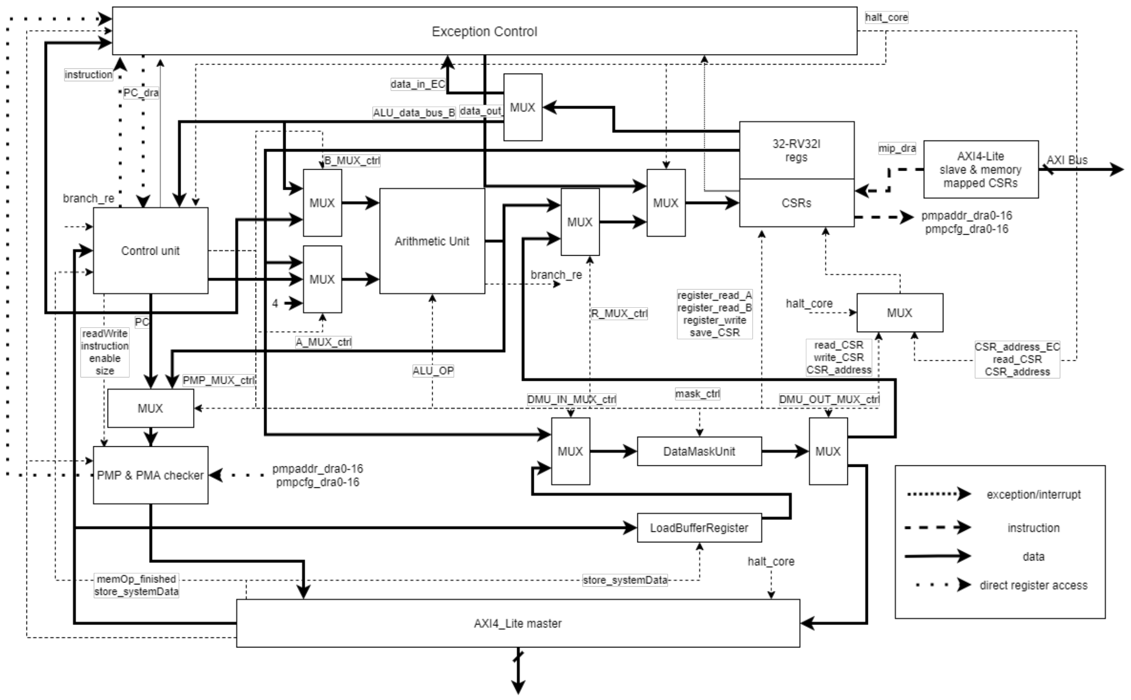


Figure 3.1: EDRICO Overview

Its main components are the *Exception Control*, *Control Unit*, Arithmetic Logical Unit (ALU), *Register Files*, *PMP & PMA checker* and the AXI4 Interfaces. How these components interact with each other, in order to execute Instructions, is specified in the data path. The following sections will describe each one of the components in more detail.

3.1 Data Path

The Data Path specifies how the data is passed through the Processor Core at run time. It therefore determines what registers are read and written at which clock cycle, what control signals need to be applied and how many cycles the instruction execution takes.
To run an instruction, it must be fetched, decoded and executed. Execution varies for different instructions.
To perform for example a load, the target address must be calculated and verified for possible access constraints. After successfully accessing the memory space, the obtained data is modified to satisfy the instruction specific formatting. A load half-word unsigned operation for instance must return a 32-Bit value by zero extending the loaded two bytes of data.
In case of a simple register-register addition, execution is found to be a lot simpler. Addition is performed inside the Arithmetic Logic Unit on two register values. The result is stored to the target register on the next falling clock edge.
With the end of the execute phase, the Programm Counter is updated. This ensures that the 32-Bit register always contains the address of the current instruction to be fetched, decoded and executed.
Figure 3.2 shows the different actions that are performed on each clock cycles during run time:

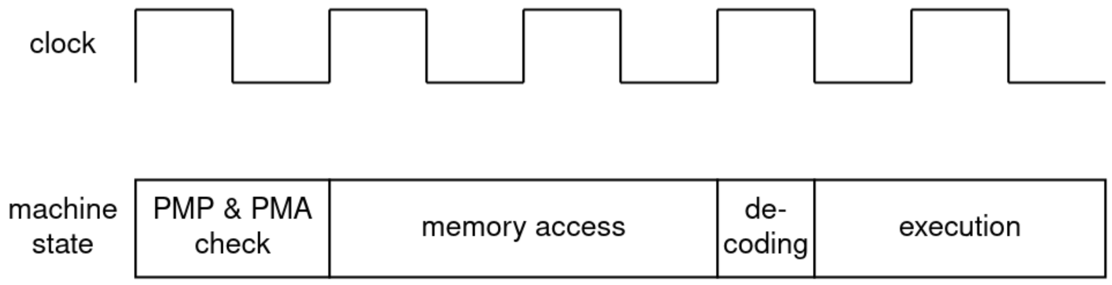


Figure 3.2: Generic Data Path

Prior to memory access, the address must be checked for possible violations of the physical memory protection (PMP) and physical memory attribute (PMA) rules. Therefore the PC is passed to the PMP and PMA checker unit (see: 3.1).
PMP and PMA checks are finished one clock cycle later, if an exception is generated, the trap is taken. Otherwise the data as well as the control signals are passed to the AXI4-Lite master. The AXI4-Lite transfer will take multiple clock cycles. Its length can be increased by external factors, such as the memory to be accessed or the amount of data currently passing through the AXI-interconnect. To simplify the depiction in 3.2 it is assumed to take only two clock cycles.

After successfully accessing the memory, decoding is performed in half a clock cycle. Decoding returns all the control signals of EDRICO set to the correct levels in order to perform the desired operation. Hence, no control signals do change during execution. This is desired since it will allow an easier pipelined implementation of a RISC-V core based on EDRICO.
Execution is displayed to take one and a half clock cycles. The registers are written at the falling clock cycle, one cycle after decoding started. Therefore execution is finished after only one clock cycle and not one and a half. Since the next instruction fetch is issued at a rising clock edge the machine must remain in execution state for an additional half clock cycle.
The fact that some registers are falling-edge and others rising-edge sensitive may seem a little bit disturbing at first glance. It is done to allow easier implementation and achieving of timing-closure during place and rout. If, for example, a critical path is found to be at the decode process, the duty cycle of the clock can be modified in order to achieve a higher possible clock frequency.

3.2 Control Unit

The Control Unit (CU) is the heart of the processor and controls the other parts of the processor depending on the input instruction. The CU is responsible for fetching instructions from the instruction memory, decode the bitstream and set the respective control signals for the other processor components. Due to the complexity of the CU, there are several sub-modules which together form the overall CU.

3.2.1 Architecture and Design

A general overview of the CU architecture is displayed in Figure 3.3.

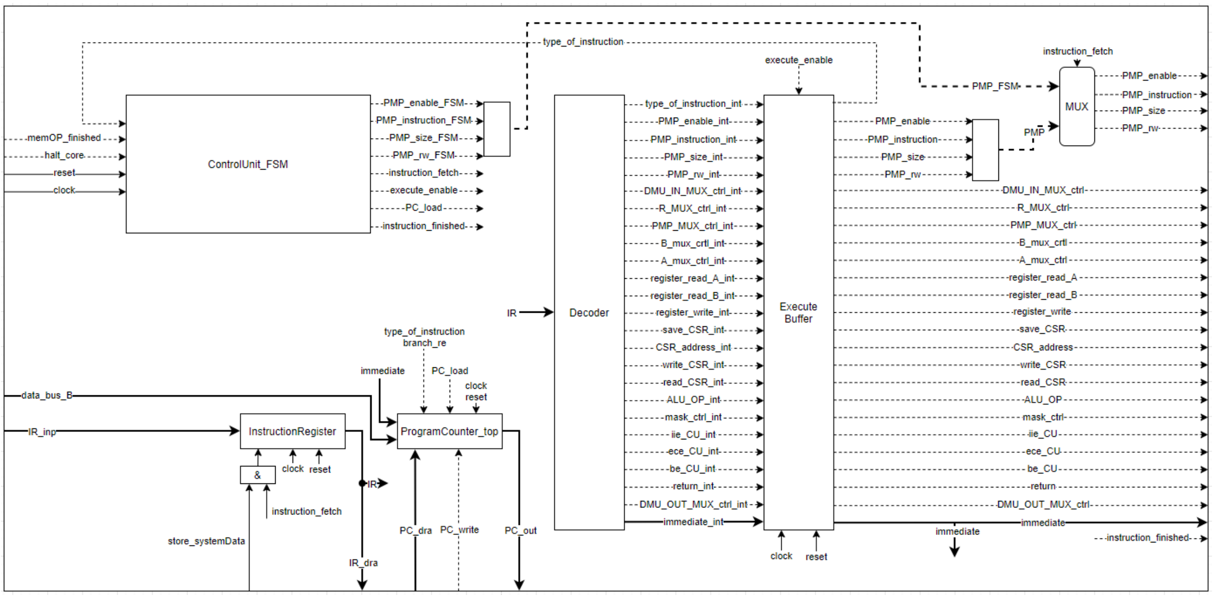


Figure 3.3: Control Unit Architecture

To describe the functionality of the CU in more detail, every sub-module will be described closely.

Since the Control Unit is responsible for the whole processor, it is important to have a persistent and stable procedure for every instruction that shall be executed. The Control Unit FSM is responsible for the correct clock timings which is important due to memory operations and the execution time of the other processor parts. The states and conditions of the FSM are displayed in Figure 3.4.

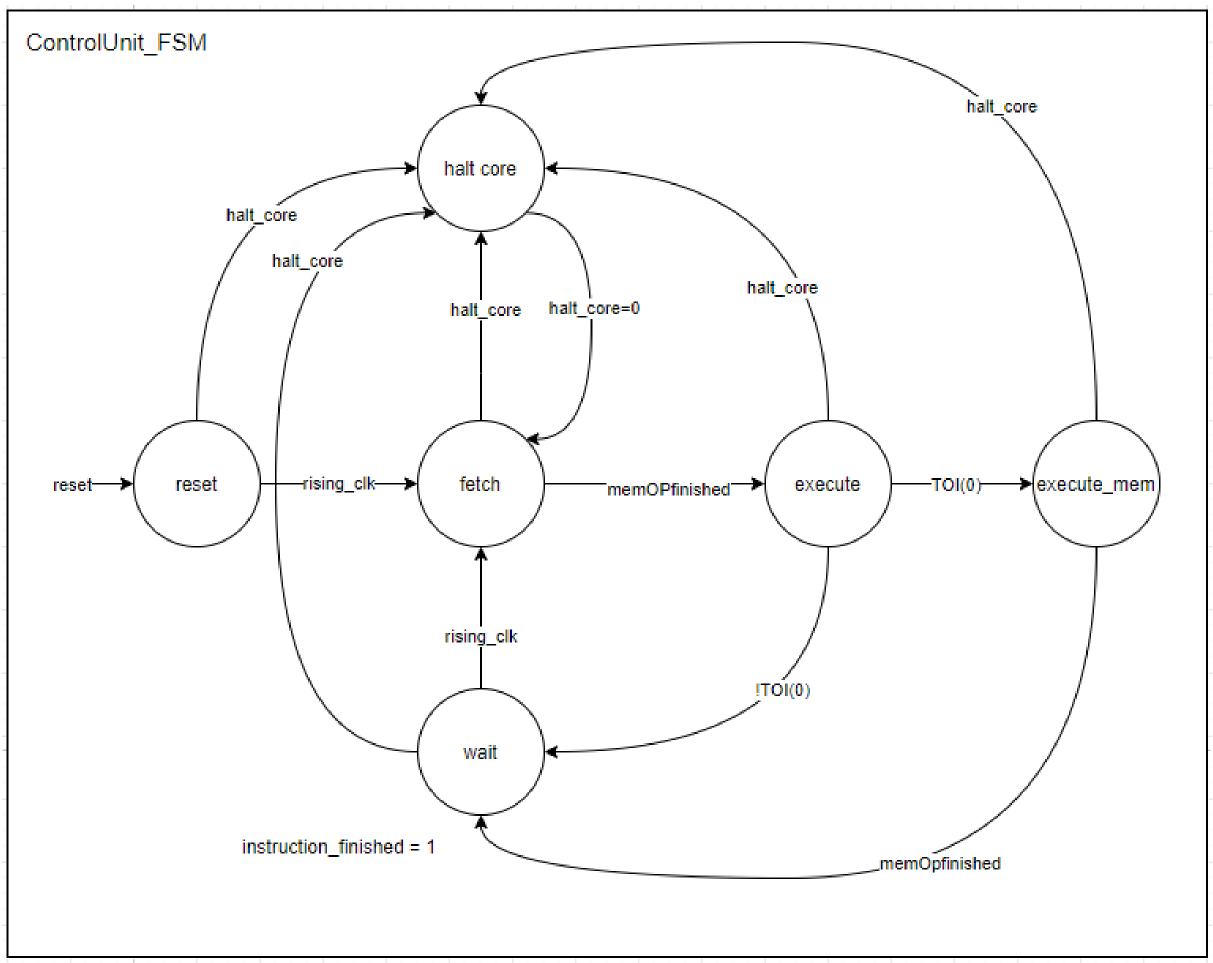


Figure 3.4: Control Unit FSM overview

Table 3.1 shows a more detailed overview of the clock cycles and the corresponding actions and states:

ClockCycle	**Edge**	**Action**	**Signal**
1	rising	pass the PC and enable PMP & PMA checker with respective information	
	falling	N/A	
4	rising	data is ready in instruction register - switch to execute state	*memOPfinished* & *store_systemData* is high
5	rising	execution is started - if memory operation wait for another *memOPfinished* flag, otherwise wait	*execute_enable*
x	rising	during memory operation: data loaded to buffer \store transfer finished → wait state	*memOPfinished* & *store_systemData* is high
	falling	if load: store data form buffer to specified location	
6 / x+1	rising	go to *fetch_state*	

Table 3.1: Timing of FSM

During an execution cycle, the FSM controls the rest of the CU consisting of memory, decoding unit, PC control and the different multiplexers. To understand what the purpose of the different signals are, the other components of the Control Unit are described in the following sections.

After loading an instruction from the memory to the instruction register, the decoding process can begin. The responsible part for this process is the decoding unit which is described below.(Also visible in figure 3.3)
In this project the RISC-V RS32I instruction set is used which consists of 32-bit instruction words. The instruction words have a pre-defined structure and are divided into six instruction formats. The instruction formats are shown in Figure 3.5.

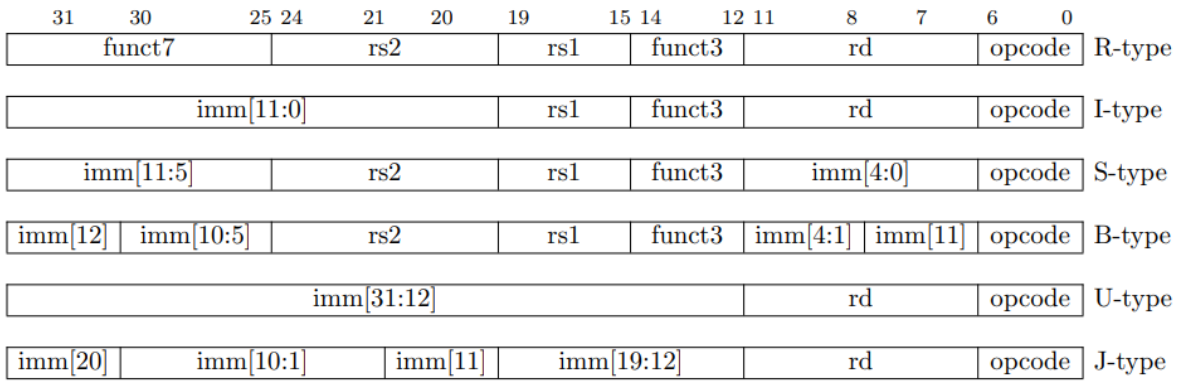


Figure 3.5: RISC-V Instruction formats [**riscv:privileged**]

The different instruction formats are useful for the decoding process since e.g. all LOAD instructions have the same structure and therefore, the effort to decode the 32-bit word can be reduced. Since the control signals are unique for every instruction and depending on the content of the 32-bit word, the decoder has to identify the encoded instruction, extract the information and respectively set the control signals, calculate immediates and control the multiplexers. A more detailed description of the decoding process can be found in section 3.2.2.
After the instruction is decoded, all output control signals are stable and ready to be fed through. Before leaving the CU, the *Execute Buffer* (figure 3.3) buffers the control signals. Once the FSM sets the *execute_enable* flag, the control signals are fed through. This buffer prevents the processor to confuse timing and clock cycles, or use signals which are not yet set correctly.
During an instruction execution, the program counter has to be incremented for the processor to know what instruction will follow. *But* since there are several instructions that modify the program counter, a so called *PC control* is designed. The PC control receives information from the decoder which consists of a 4 bit signal. The different instructions and the respective action as well as the respective control signal are shown in following table 3.2.1:

Instruction	**Action**	**Control Signal**
Default	No action required	**0000**
Branch	Depending on the result of branch operation, PC will be incremented respectively	**0010**
JAL	Target address obtained by adding current PC and immediate, rejump address stored in register	**0100**
JALR	Target address obtained by adding input register to immediate	**1000**

Table 3.2: Program Counter control: Instructions and resulting actions

For instructions which do not influence the program counter, the standard operation performs the **PC + 4** operation.

The instruction register displayed in figure 3.3 manages the instruction string coming from the memory. All of these parts together form the Control Unit and are responsible for the correct execution of the instructions. The implementation of the sub-units in VHDL are described in the following section 3.2.2.

3.2.2 Implementation

The implementation of the Control Unit is split up into multiple sub-implementations. As shown in figure 3.3 those sub-modules are the *FSM, decoder, execute_buffer, PC control and instruction register*. Since the implementation of the FSM is very similar to other FSM implementations in this project, the detailed description of a FSM in VHDL is found in the next chapters.
In this section the implementation of the decoder will be described more closely. As already described in section 3.2.1 the instructions can be separated in different instruction formats. To distinguish the different instructions, so-called *instruction clusters* are created. These clusters sum up instructions which are encoded in the same instruction format or in general are similar. The following table shows the different clusters and the corresponding instructions:

Cluster	**Instructions**
LOAD	Load - Byte \Halfword \Word
STORE	Store - Byte \Halfword \Word
BRANCH	Different Branch Instructions (e.b. Branch if equal)
JALR	only JALR, since it has a unique instruction structure
JAL	only JAL, since it has a unique instruction structure
OP	All arithmetic instructions like ADD, SUB, shift and comparisons
OP-IMM	All arithmetic instructions performed with immediate
AUIPC	only AUIPC, since it has a unique instruction structure
LUI	only LUI, since it has a unique instruction structure

Table 3.3: Decoding instruction clusters

To determine the cluster for each instruction, a decoding procedure is implemented in VHDL based on structure visualized in figure 3.6:

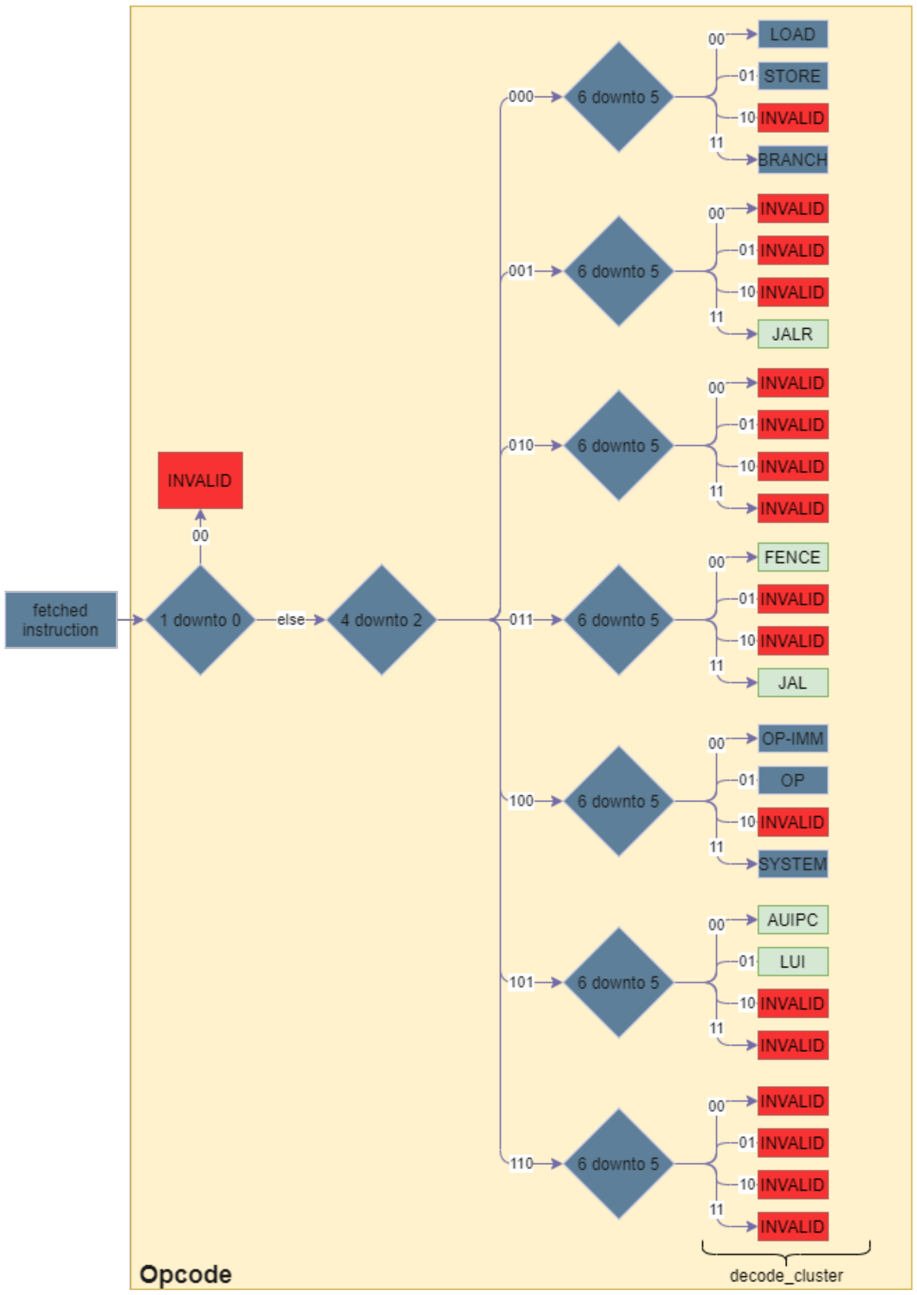
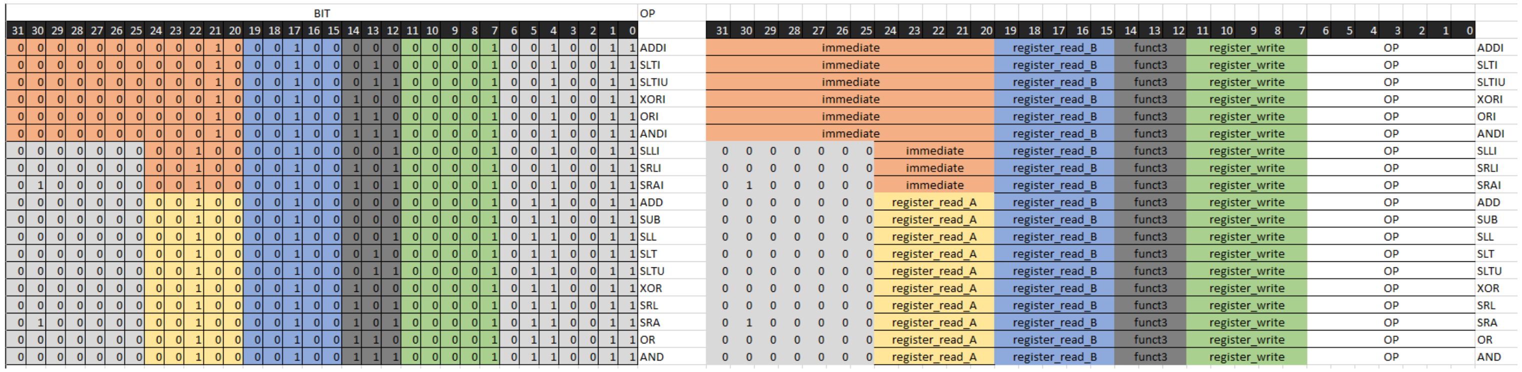


Figure 3.6: Decoding Structure to determine instruction cluster

After determining the cluster, the VHDL code assigns all the outputs visible in figure 3.3 with the respective information. For a better understanding of the information extraction, figure 3.7 shows how the 32-bit instruction word is split up (in this case for the *OP* and *OP-IMM* instructions.)



BIT																																OP
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	ADDI
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	1	SLTI
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	1	SLTIU
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	1	1	XORI
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	ORI
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	ANDI
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1	1	SLLI
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0	1	1	SRLI
0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0	1	1	SRAI
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	ADD
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1	SUB
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	SLL
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	1	1	SLT
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1	0	1	1	0	0	1	1	SLTU
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	0	1	1	XOR
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	SRL
0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	SRA
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1	1	0	0	1	1	OR
0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	1	1	AND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
immediate												register_read_B					funct3			register_write					OP							ADDI
immediate												register_read_B					funct3			register_write					OP							SLTI
immediate												register_read_B					funct3			register_write					OP							SLTIU
immediate												register_read_B					funct3			register_write					OP							XORI
immediate												register_read_B					funct3			register_write					OP							ORI
immediate												register_read_B					funct3			register_write					OP							ANDI
0	0	0	0	0	0	0	immediate					register_read_B					funct3			register_write					OP							SLLI
0	0	0	0	0	0	0	immediate					register_read_B					funct3			register_write					OP							SRLI
0	1	0	0	0	0	0	immediate					register_read_B					funct3			register_write					OP							SRAI
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							ADD
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							SUB
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							SLL
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							SLT
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							SLTU
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							XOR
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							SRL
0	1	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							SRA
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							OR
0	0	0	0	0	0	0	register_read_A					register_read_B					funct3			register_write					OP							AND

Figure 3.7: Information extraction from 32-bit instruction word

3.3 Arithmetic Logical Unit

The Arithmetic Logical Unit (ALU) is the part of the processor that performs the arithmetic and logical operations. Figure 3.8 gives an overview of what type of operations are performed.

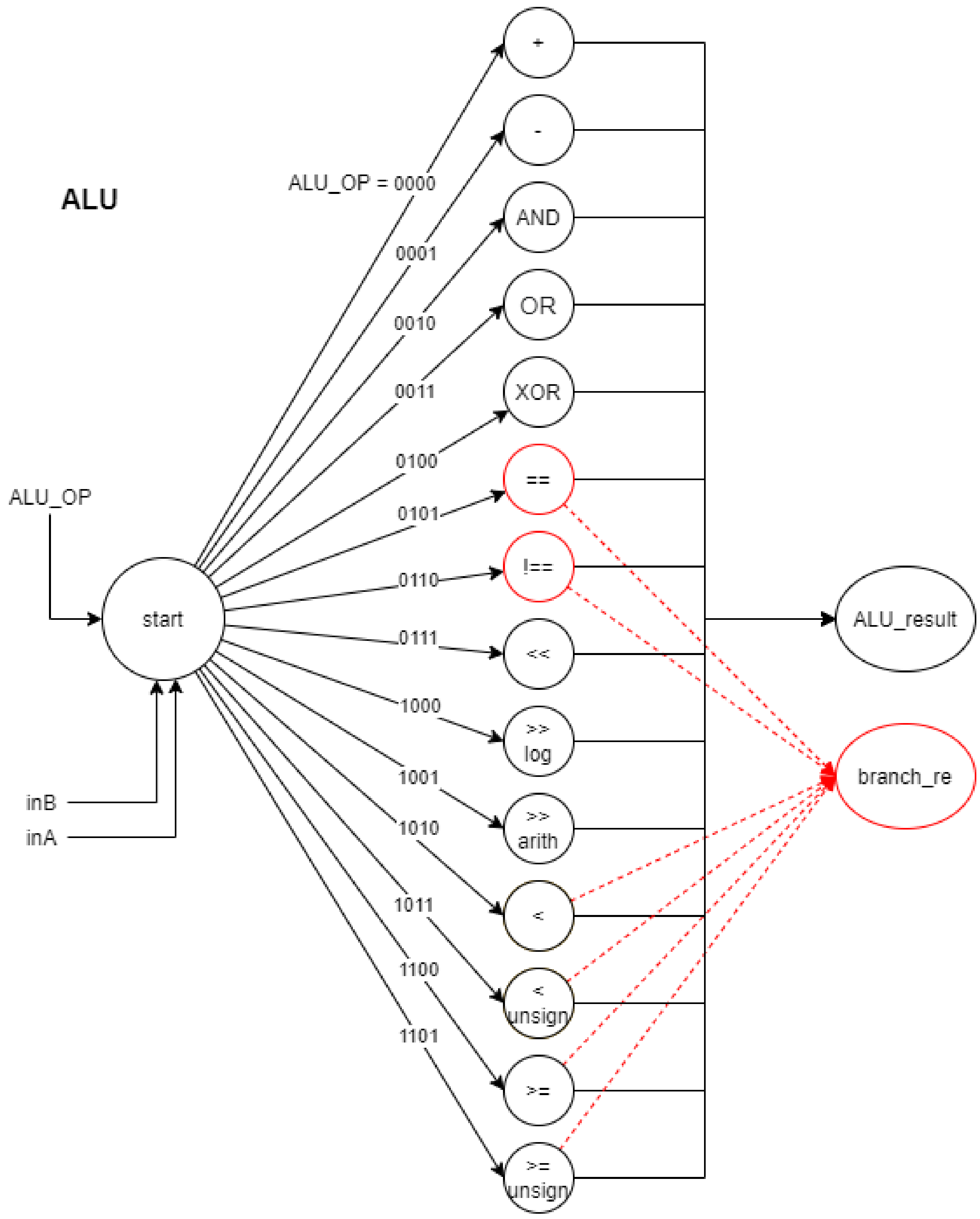


Figure 3.8: ALU operations

3.3.1 Architecture and Design

To implement the ALU, it is required to have the data inputs as well as clock input and a control signal consisting of 4 bits to specify the required operation to be performed. Since there are instructions that require a branch response to know whether the next instructions shall be skipped or not, the ALU needs an additional output called *branch_re* other than the result output of the arithmetic/logical operation. The architecture of the ALU is shown in figure 3.9:

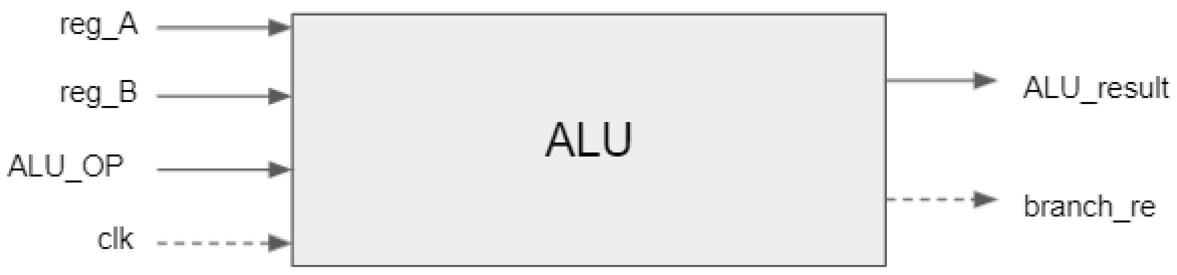


Figure 3.9: ALU operations

Not only the instructions e.g. *ADD, SUB, XOR...* require an arithmetic operation but also *LOAD, STORE..* require an ALU action. While *ADD, SUB, XOR...* require an operation between the two input values (either register-register or register-immediate) to get a mathematical or logical result, the *LOAD, STORE...* instructions require the ALU to build the target addresses for the memory access.

3.3.2 Implementation

The implementation of the ALU is based on figure 3.8 and performs a switch-case on all the different input values of *ALU_OP*. The 4-bit input variable specifies the operation based on following declarations:

ALU_OP	Operation
0000	ADD
0001	SUB
0010	AND
0011	OR
0100	XOR
0101	EQUAL
0110	NEQUAL
0111	shift_left
1000	shift_right
1001	shift_right (arithmetic)
1010	$<$
1011	$<$ (unsigned)
1100	$\geq$
1101	$\geq$ (unsigned)

Table 3.4: Input code and respective operation

To visualize the implementation, a part of the VHDL code is displayed in the following. The case statement is based on the input *alu_op*. The ALU then performs the corresponding operation with the two inputs *in_a and in_b*.

```
1  begin
2    process(in_a, in_b, alu_op)
3    begin
4    --default output is 0
5    branch_re <= '0';
6    alu_result <= "00000000000000000000000000000000";
7      case alu_op is
8        when "0000" =>--"ADD"
9          alu_result <= in_b + in_a;
10       when "0001" =>--"SUB"
11         alu_result <= in_b -in_a;
12       when "0010" =>--"AND"
13         alu_result <= in_b AND in_a;
14       when "0011" =>--"OR"
15         alu_result <= in_b OR in_a;
16       when "0100" =>--"XOR"
17         alu_result <= in_b XOR in_a;
18       when "0101" =>--"EQUAL"
19         if(in_b = in_a) then
20         branch_re <= '1';
21         else
22         branch_re <= '0';
23         end if;
24   ...
```

Listing 3.1: ALU VHDL code

In case the operation determined by *alu_op* might be originating of a branch instruction, the *branch_re* flag has to be set respectively (line 19). Since the branch instructions only include some of the arithmetic and logical operations of the ALU, the default value for the *branch_re* is set as *0*.

3.4 Register File (RF)

A register is a small memory element with high read and write speeds. It is therefore often used inside digital circuits to store data locally. In the case of a processor core, the total data stored in all registers specifies the machine state.
If the contents of each register are saved to some arbitrary memory the current state of the core can later be restored by loading this data to the corresponding registers.
The RISC-V unprivileged specification specifies 32 32-Bit general purpose register for the RV32I base instruction set [**riscv:unprivileged**]. To configure the core as well as storing status and general information about it multiple Control and Status Register (CSR) are introduced by the RISC-V privileged specification [**riscv:privileged**].
EDRICO implements both, all 32 32-Bit integer general-purpose registers as well as a selection of required CSRs.

3.4.1 Architecture and Design

A requirement for the Register Files is to allow access on one input data bus and two output data bus, each with a size of 32-Bit respectively. Multiple control signals can be accessed to control the data flow through the module. Registers are read asynchronously, hence a read operation is not dependent on any clock. Writes to a register are performed on a falling clock edge.
Some of the CSRs need to provide their contents to other modules without using one of the two designated output buses. This decreases unnecessary overhead on e.g. load and store operations.
Figure 3.10 depicts the architecture of the Register File module.

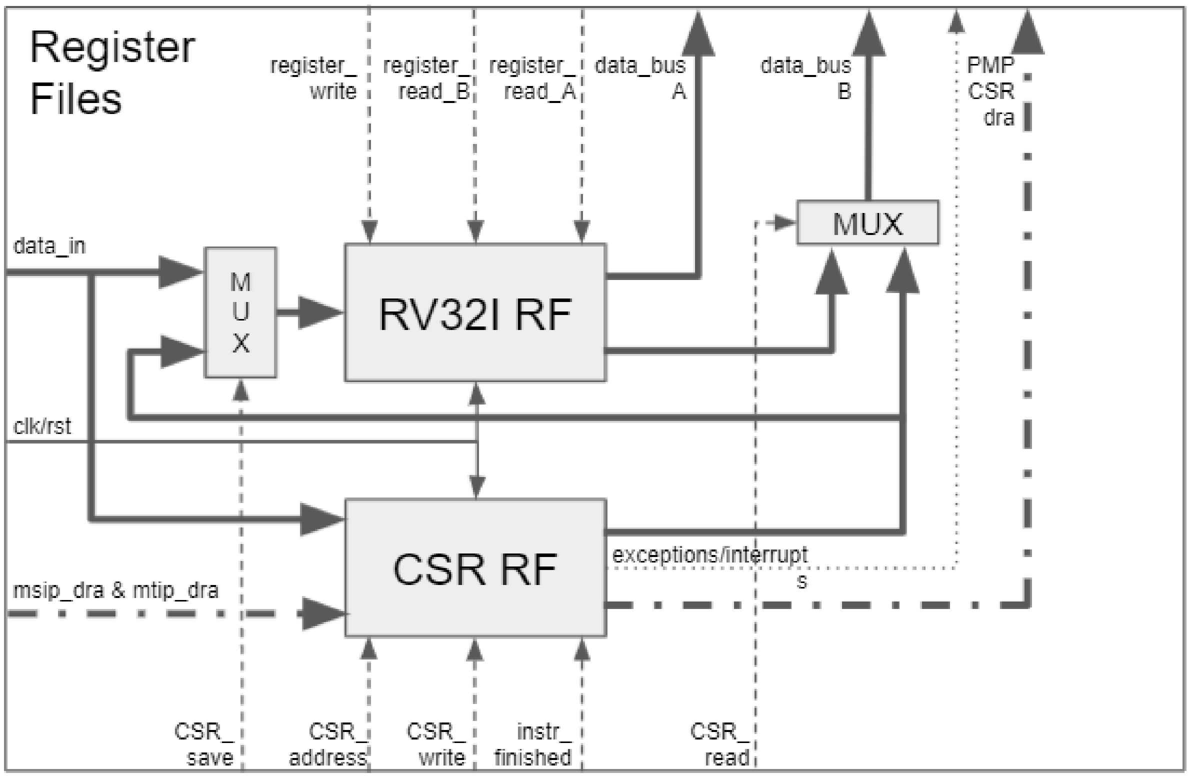


Figure 3.10: Register Files architecture

In order to write data to a register, the value needs to be put on the *data_in* bus and the corresponding control signals need to be configured. Data can be read either via the *data_busA* or *data_busB*, the CSR registers can only be accessed on *data_busB* via a MUX controlled by the *CSR_read* signal. If the bit is set, the CSR specified by *CSR_address* will be visible on *data_busB* at the next rising clock edge.
In order to store data to a CSR register, it has to be put on the *data_in* bus. If the *CSR_write* bit is set, the data is saved to the register specified by *CSR_address* on the next falling clock edge.
Execution of special CSR instructions, like the Atomic Read/Write CSRRW [**riscv:unprivileged**], requires writing to both a general-purpose RV32I register and a CSR. Since the control signals are not allowed to change during execution, it is mandatory to allow the CSR RF to write directly to the RV32I RF without utilizing the *data_in* bus. In order to do so, the *CSR_save* signal is added to design. If it is applied, the CSR Register File output is connected to the data input bus of the general purpose registers.
Some of the CSR allow direct memory accesses. These are implemented in a separate module, including the *mtime*, *mtimecmp* and *msip* registers as well as a dedicated AXI4-Lite slave interface. To set the software and timer interrupt pending bits (which are caused by the memory mapped CSR), the *msip_dra* and *mtip_dra* inputs are implemented.

The CSR Register File is accessed by a 12 Bit addresses, RV32I registers are accessed using a four byte address signal for each data bus, respectively.

General Purpose Registers

The general purpose registers are used to store the data on which operations are performed on. Since RISC-V is a load-store architecture no data from the memory can be modified directly without loading it to a General Purpose Register (GPR) first (except when using atomic operations) [**riscv:unprivileged**].
Figure 3.11 shows the design of RV32I Register File:

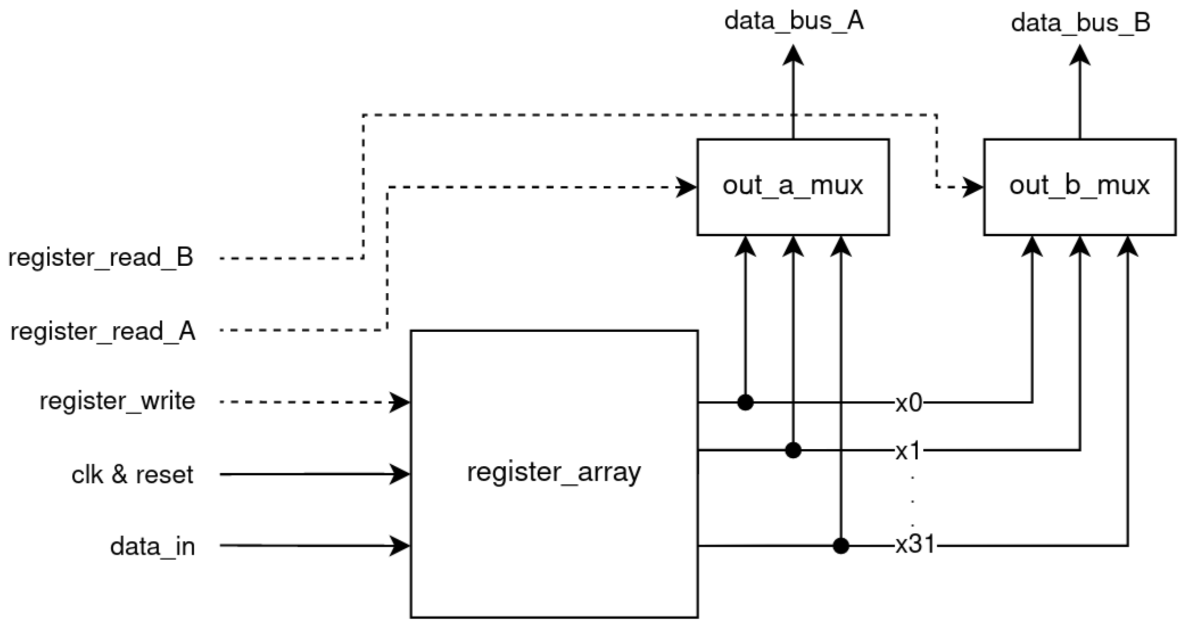


Figure 3.11: General Purpose Registers

Every register may be modified via *data_in* and read via *data_busA* and *data_busB*. The only limitation is *x0*, per definition *x0* must be hardwired to zero. Writing to the register is allowed but has no effect, reading will return *0x00000000*. According to the RV32I programmers model some of the other registers should be used for dedicated purposes, like the stack pointer. There are no hardware checks implemented in order to enforce those rules.
Writes are executed on a falling edge if the corresponding *register_write* signal is high. Reads are performed on a rising edge if the corresponding *register_read_A/B* is high. There are no hardware checks implemented to prevent multiple registers from writing to the same data bus at the same time. This must be prevented by the controlling element e.g. the control unit.
All registers are cleared to *0x00000000* on reset.

Control and Status Register (CSR)

CSRs are Control and Status Registers that are introduced to the design by the RISC-V privileged specification. Some of the values stored inside the CSRs need to be provided to the Exception Unit and the PMP & PMA checker to decrease complexity, some of that accesses can be performed through direct memory access. Some of the CSRs are memory mapped. Memory mapped means they need to be accessible via the system bus by any AXI4-Lite master. In order to implement this, an AXI4-Lite salve module implementing all memory mapped CSRs is added to the design.
Further Information about this module can be found in the chapter 3.8. The CSRs that provide direct access are:

- *msip* and *mtip* bits in the *mpi* register (write)
- *pmpcfg* (read)
- *pmpaddress* (read)

Some registers are specified as WARL registers, meaning anything can be written to them, but the value returned on read must be a legal value. Table 3.4.1 displays every non memory mapped CSR, the corresponding address, type, access possibility, width and a short description:

register	address	type	access	width	description
misa	0x301	WARL	R	32-bit	describes supported ISAs
mvendorid	0xF11	N/A	R	32-bit	describes vendor id
marchid	0xF12	N/A	R	32-bit	describes architecture ID
mimpid	0xF13	N/A	R	32-bit	describes implementation ID
mhartid	0xF14	N/A	R	32-bit	describes hart id
mstatus	0x300	N/A	R/W	32-bit	reflects & controls a hart's current operating state
mtvec	0x305	N/A	R/W	32-bit	holds trap vector configuration
mie	0x304	N/A	R/W	32-bit	reflects interrupt enable state
mip	0x344	N/A	R	32-bit	holds interrupt pending bits
mcycle	0xB00	N/A	R/W	64-bit	holds count of clock cycles
minstret	0xB02	N/A	R/W	64-bit	holds count of executed instructions
mhpcounter(3-31)	0xB03-0xB1F	N/A	R/W	32-bit	holds count of events (lower 32 bit)
mhpcounterh(3-31)	0xB83-0xB9F	N/A		32-bit	holds counter of events (upper 32 bit)
mhpevent(3-31)	0x323-0x33F	N/A	R/W	32-bit	specifies events on which to increment corresponding mhpmcounter
mcountinhibit	0x320	WARL	R/W	32-bit	controls which hardware performance monitoring counters increment (if set, no increment)
mscratch	0x340	N/A	R/W	32-bit	Dedicated for use by machine-mode
mepc	0x341	WARL	R/W	32-bit	holds jump-back address during interrupt
mcause	0x343	N/A	R/W	32-bit	specifies cause of exception/interrupt
mtval		N/A	R/W	32-bit	holds information about trap
pmpcfg0-3	0x3A0-0x3A3	N/A	R/W	32-bit	Physical memory protection configuration
pmpaddr0-15	0x3B0-0x3BF	N/A	R/W	32-bit	Physical memory protection address register

Table 3.5: List of implemented CSRs

The architecture of the CSR Register File is shown in 3.12.

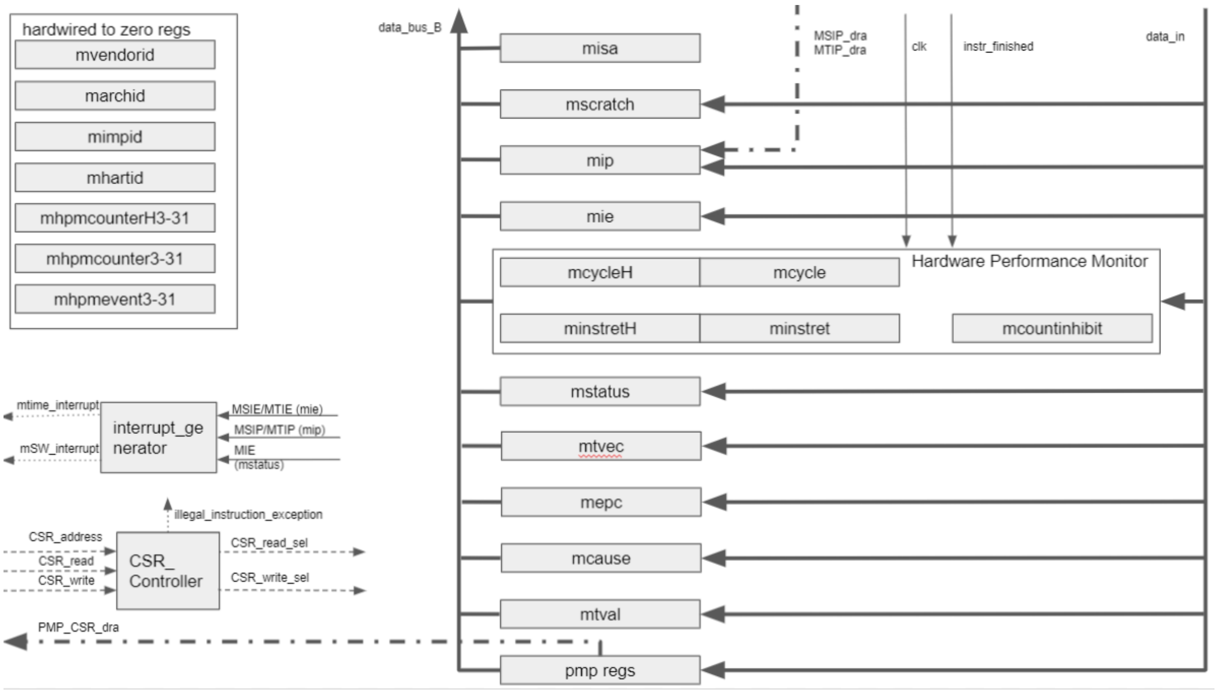


Figure 3.12: CSR Register File Architecture

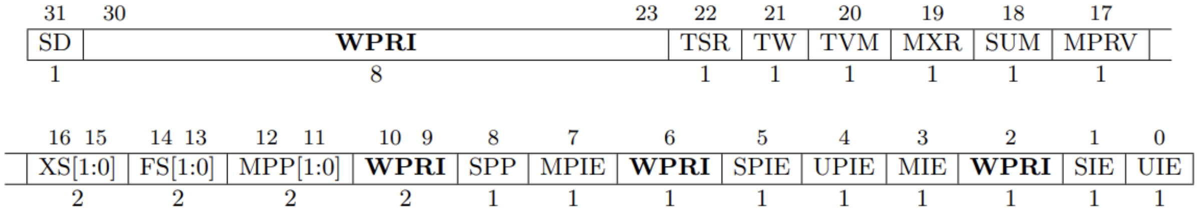
The CSR Register File is comprised of the different CSR Registers, *CSR_Controller* and *interrupt_generator*.
Some of the implemented CSR registers are hardwired to zero, reads to those addresses return *0x00000000* and writes have no effect if the register is defined as read only the write will result in an *illegal_instruction_exception*. Other CSR registers may be defined as read-only.
The *CSR_Controller* checks if a CSR access is allowed and produces access signals for each register from the given address. If an address is not writable but yet a write is requested, the *CSR_Controller* raises an *illegal_instruction_exception*.
The *interrupt_generator* checks if an interrupt is pending, enabled and if interrupts are globally enabled. In that case the corresponding interrupt is raised. The interrupts remain pending, as long as the corresponding direct register access signals it. To prevent the machine from being stuck in the interrupt, the programmer of the Interrupt Service Routine (ISR) must clear the pending interrupts, e.g. the timer interrupt by writing to the memory mapped CSRs.
Some registers have special functionalities. The Hardware Performance Monitor for example counts the number of clock cycles as well as the number of executed instructions. Calculation of the Cycles per Instructions (CPI) can be performed as shown in (3.1):

$$CPI = \frac{mcycleH >> 32 + mcylce}{minstretH >> 32 + minstret} \tag{3.1}$$

During Benchmarks these registers can be used to calculate the performance of the core. For physical memory protection, several attributes such as read and write can be specified inside the Physical Memory Protection (PMP) CSR, a more detailed description of these specific registers can be found in: [**riscv:privileged**].
There are 16 *pmpaddr* and four *pmpcfg* registers. In order to verify that a memory access does not violate any PMP rules, the PMP and PMA checker needs access to these registers. Performing this accesses over the *data_out_B* bus would result in 16+4 data transfers, so 20 register accesses in total. This would impose a significant overhead of at least 20 clock cycles on every memory access. To reduce the effect of checking the PMP rules a direct register access is implemented.

3.4.2 Implementation

Implementing the RV32I RF is done by defining an array of 32 *std_logic_vector* of 32-Bit each. The zeroth element in the array corresponds to the x0 register, therefore it can not be written and is hardwired to zero.
On a write, the 5-Bit *register_write* signal is used to index the corresponding array element. This is only possible since writes to the x0 register do not effect the state of the machine. Hence, no writes are performed if the write signal is set to *0b00000*.
Since some CSR have additional features they need to be implemented individually, therefore the matrix approach used for the RV32I register files can not be taken. There are multiple registers that are hardwired to zero, they are implemented as one and the read write multiplexers access the same register if an access to one of these registers occurs.
To safe resources all CSR implement just the bits that are needed, the others are hardwired to zero. The following listing illustrates this concept, using the *mstatus* register as an example, figure 3.13 shows the mstatus register according to [**riscv:privileged**]:



31 30 23 22 21 20 19 18 17
SD | WPRI | TSR | TW | TVM | MXR | SUM | MPRV
1 8 1 1 1 1 1 1
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
XS[1:0] | FS[1:0] | MPP[1:0] | WPRI | SPP | MPIE | WPRI | SPIE | UPIE | MIE | WPRI | SIE | UIE
2 2 2 2 1 1 1 1 1 1 1 1 1

Figure 3.13: 32-Bit machine mode register (*mstatus*) [**riscv:privileged**]

```vhdl
----------------------------------------------------------
--mstatus register
----------------------------------------------------------
mstatus_proc: process(reset, clk)
begin
if(reset = '1') then
mstatus_reg <= (others => '0');
elsif(clk'event and clk = '0' and write(0) = '1') then
mstatus_reg <= data_in(7) & data_in(3);
end if;
end process;

mstatus <= x"000018" & mstatus_reg(1) & "000" &
    mstatus_reg(0) & "000";
```

Listing 3.2: mstatus implementation

For the chosen implementation only two bits in the status register are relevant, Machine Prior Interrupt Enable (MPIE) and Machine Interrupt Enable (MIE). Every Other bit can be hardwired to a specific value. Since neither supervisor nor user mode are implemented are the corresponding (prior) interrupt enable bits tied to zero. The *xPP* fields hold the previous privilege mode in which the machine was prior to a trap. Only machine mode is implemented, hence Machine Previous Privilege (MPP) can be hardwired to "11" and Supervisor Previous Privilege (SPP) to "00". According to [**riscv:privileged**] all other bits may be hardwired to zero if only machine mode is implemented.

3.5 PMP and PMA Checker

Physical memory protection and attribute checking must be done in order to ensure that only allowed and defined memory regions are accessed, providing minimum of security and preventing the core from accessing not defined regions, which may result in the core getting stuck.

3.5.1 Architecture and Design

An overview of the PMP & Physical Memory Attributes (PMA) Checker is illustrated by 3.14:

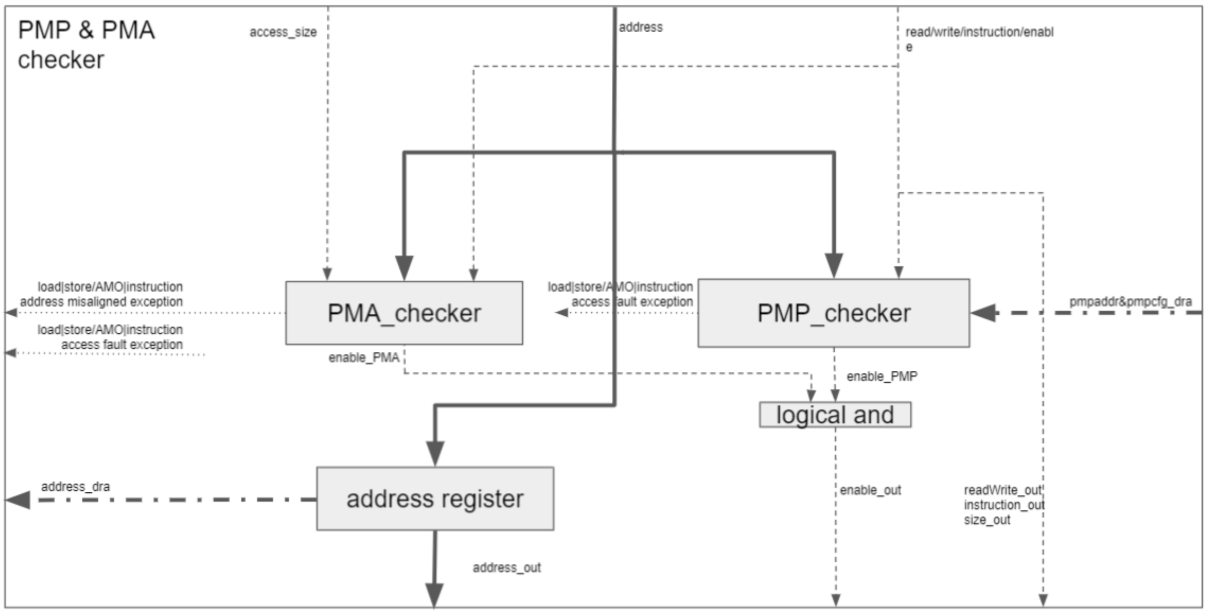


Figure 3.14: PMP & PMA Checker Architecture

The *PMP_checker* is used to define memory regions and enforce several rules onto those, for example if instructions may be fetched from a region or if writes are allowed. In order to apply the rules the corresponding region must be locked by setting the L-bit inside the corresponding *pmpcfg* CSR. Once locked, regions may only be unlocked by a system reset. After a restart every region is unlocked and reset, e.g. a boot loader could enforce several rules for memory accesses in M-Mode before control hand-over to the main software running on the core. If a PMP entry is not locked, every memory access that matches this address space succeeds.
In order to specify, if reads, writes or instruction fetches are possible to a certain address, it needs to be specified inside the *pmpaddr* and corresponding *pmpcfg* CSRs. If an address

is not specified every access is allowed, as long as the hart operates in machine mode. The PMP unit has direct access to those and enforces the rules. Figure 3.15 and 3.16 show these two registers.

7	6 5	4 3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)
1	2	2	1	1	1

Figure 3.15: 8 Bit from 32-Bit *pmpcfg* register [**riscv:privileged**]

31   0
address[33:2] (**WARL**)
32

Figure 3.16: *pmpaddr* register [**riscv:privileged**]

One *pmpcfg* register is comprised of four 8-bit configuration fields, one of which is shown in 3.16. The X, W and R bit specify whether or not the region defined by this entry is fit for instruction accesses, reads or writes. The size and range of the region is specified by the mode chosen by setting the A field in the configuration register. It can be configured to disable the region, specify it as Top of Range (TOR), Natural aligned four-byte region (NA4) or NAPOT.
If the configuration is set to TOR, the memory region to be checked is built from the next lower *pmpaddr* register (or *0x00000000* for *pmpcfg0*) to the corresponding address register. For a NA4 region, the *pmpaddr* bits 31 to 0 have to be equal to the address bits 33:2 of the address to be checked. Since EDRICO implements an address space of only 32 bits the upper two bits are ignored. NAPOT and NA4 regions are very similar, in fact a NA4 region is just a 4-byte NAPOT region [**riscv:privileged**]. Possible sizes are 2^x, with x = 32 being the maximum for this implementation. To check if an address fits in the NAPOT range, the lower x bits of it must match the address specified in the corresponding *pmpaddr* register.
It is important to mark that both the lower and upper address of a memory access have to be defined in the same *pmpregion* in order to match it. If multiple entries match, the lower order entry wins.

Only one PMA check is performed on memory access, to ensure that the address is aligned. A word is 32-bit, half-word 16-bit and byte 8-bit. The smallest addressable data unit is one byte long. Therefore a word access is aligned, if the memory address modulo 4 is 0 (two Least Significant Bits (LSBs) are zero), for a half-word access the memory address modulo 2 must be zero (LSB is zero) and byte accesses are allowed on every address.

Both checker (PMP and PMA) need information about the type of access (size, read/write, instruction) this is provided by the control unit. PMP and PMA checks are applied on the data present as soon as the enable signal is applied. Both checks are simple logical functions and must be performed under a clock cycle. The address register is updated with the corresponding address after every check, independent of its result. This must be done since the Exception Control needs access to the faulty address at the rising clock edge if an exception is risen.
Figure 3.17 shows the design of the *PMP_checker* module:

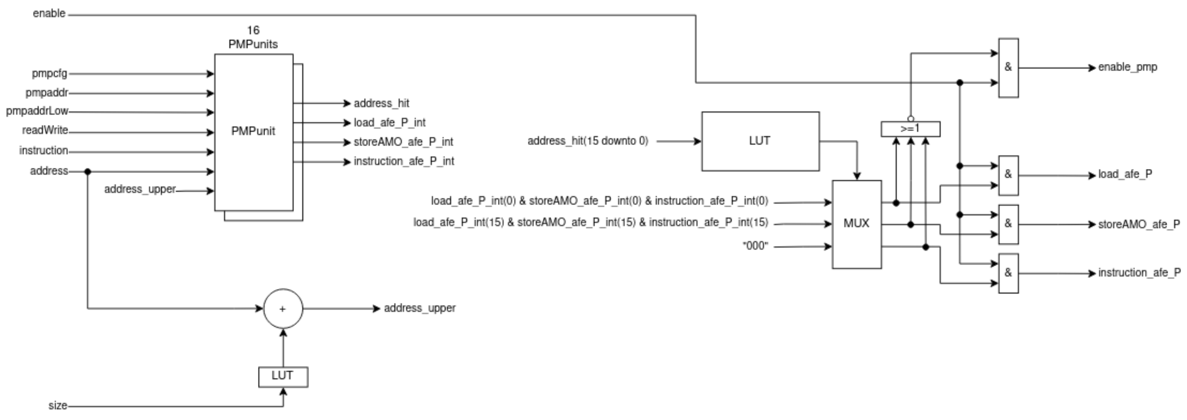


Figure 3.17: *pmp_checker* module design

To increase performance, all 16 PMP checks are performed in parallel using 16 PMP units. The results are four 16-bit vectors, to indicate an address or exception hit. For an exception to be valid, the corresponding PMP entry must also raise an address hit.
The LUT chooses the lowest order PMP region matching the address and switches the multiplexer accordingly.

3.5.2 Implementation

The implementation of the *PMA_checker* is relatively simple. A simple multiplexer chooses whether or not an intern enable signal is set or not, depending on the lower to bits of the address and the size of the transfer. The enable output is connected to the intern enable signal as soon as the enable input is set. A simple and gate is used to connect the intern signal with the external one.
The following listing shows how the access misaligned exception signals are generated inside the *PMA_checker*:

```
load_ame_P <= not enable_int and enable and not
    readWrite;
storeAMO_ame_P <= not enable_int and enable and
    readWrite;
instruction_ame_P <= not enable_int and enable and
    instruction;
```

Listing 3.3: access misaligend exception generation

In order to perform the PMP checks inside a *PMP_unit*, the size of a possible NAPOT region has to be determined. A simple LUT does the trick. Table 3.5.2 shows its configuration:

register	**value**
pmpaddr	NAPOT_size
XXXXXXXXXXXXXXX0	3
XXXXXXXXXXXXXX01	4
XXXXXXXXXXXXX011	5
...	...
XX011111111111111111	32
X0111111111111111111	32
01111111111111111111	32
11111111111111111111	32

Table 3.6: LUT to determine the size of a NAPOT region

Using the generated NAPOT size, the address can be checked according to the flow chart depicted in figure 3.18:

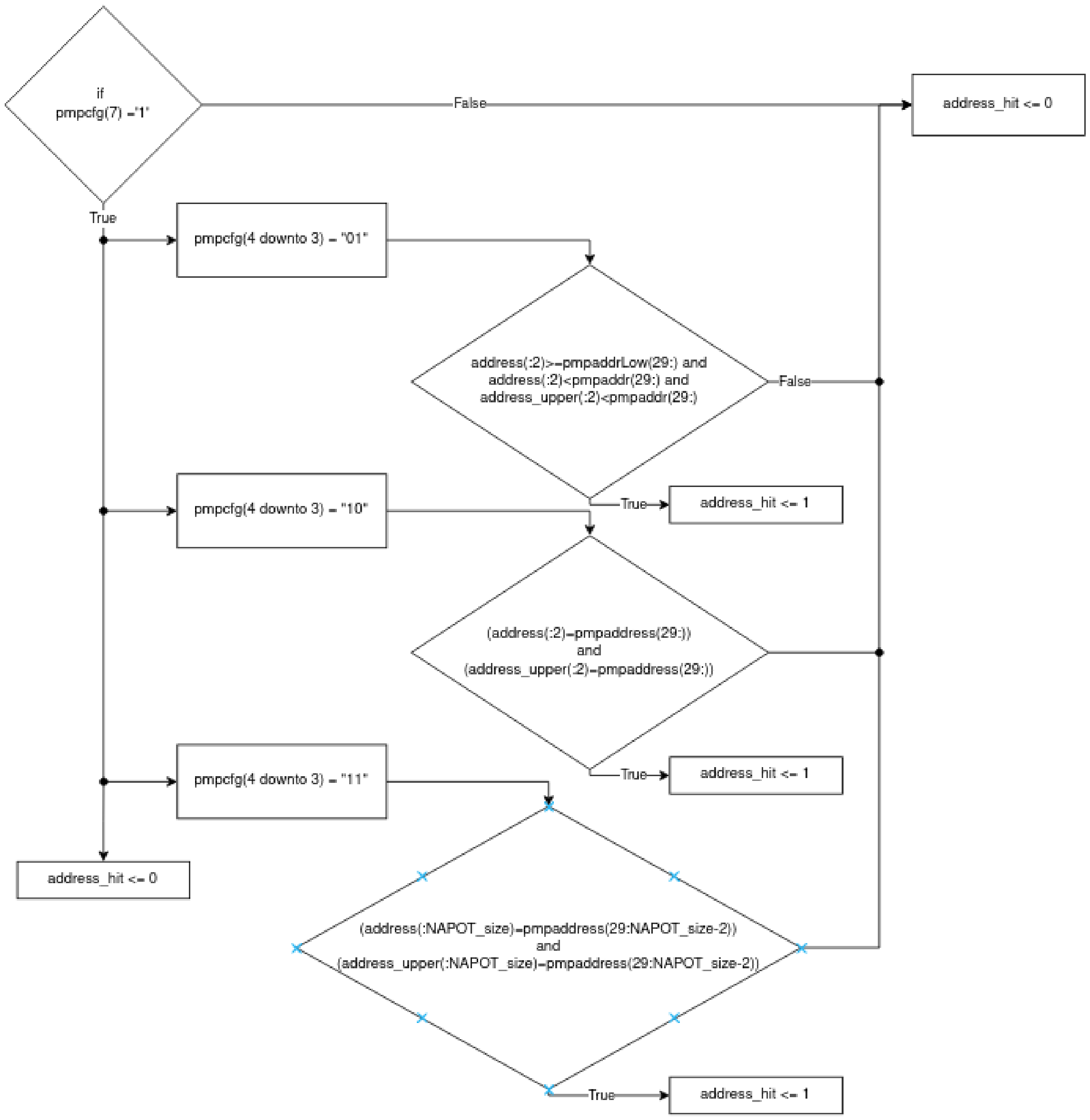

if
pmpcfg(7) ='1'
False
address_hit <= 0
True
pmpcfg(4 downto 3) = "01"
address(:2)>=pmpaddrLow(29:) and
address(:2)<pmpaddr(29:) and
address_upper(:2)<pmpaddr(29:)
False
True
address_hit <= 1
pmpcfg(4 downto 3) = "10"
(address(:2)=pmpaddress(29:))
and
(address_upper(:2)=pmpaddress(29:))
True
address_hit <= 1
pmpcfg(4 downto 3) = "11"
address_hit <= 0
(address(:NAPOT_size)=pmpaddress(29:NAPOT_size-2))
and
(address_upper(:NAPOT_size)=pmpaddress(29:NAPOT_size-2))
True
address_hit <= 1

Figure 3.18: *pmp_unit* flow chart for address hit detection

The address hit is only detected, if the PMP entry is locked. Using the A field inside the corresponding *pmpcfg* register, different scenarios are checked for an address hit. Afterwards, the exception signals for each PMP region is generated, similar to the exception generation inside the *PMA_checker*.

3.6 Exception Control

The Exception Control Unit is used to guard exception entries and exits. Its tasks are to modify the PC accordingly, save the old PC as well as information about the exception. In addition, two interrupt entries are guarded by the unit. A list of all supported exceptions and interrupts is listed below:

Exceptions can be caused by the following modules:

- Control Unit
- CSR register file
- PMP & PMA checker
- AXI4-Lite Master

The Control Unit may cause the following exceptions:

- illegal instruction exception
- breakpoint exception
- environment-call-from-M-mode exception

The PMP & PMA checker may cause the following exceptions:

- load access fault exception
- store/Atomic Memory Operation (AMO) access fault exception
- instruction access fault exception
- load address misaligned exception
- store/AMO address misaligned exception
- instruction address misaligned exception

The CSR register file may cause the following exceptions/interrupts:

- timer interrupt
- software interrupt
- illegal instruction exception

The AXI4-Lite Master may cause the following exceptions

- load access fault exception
- store/AMO access fault exception
- instruction access fault exception

3.6.1 Architecture and Design

Figure 3.19 shows the architecture of the Exception Control:

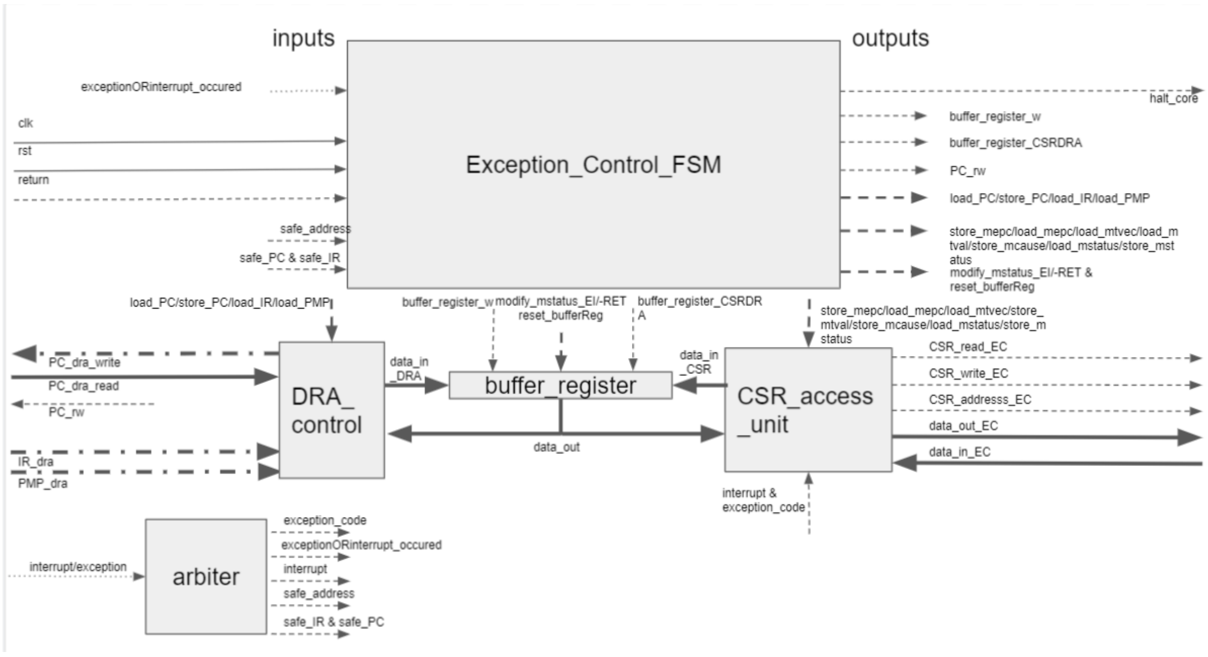


Figure 3.19: Exception Control Architecture

The Exception Control module is comprised of a FSM and an arbiter to decide if an interrupt or exception is taking place and which one shall be handled if multiple are raised at a time. To modify registers a *DRA_control* as well as a *CSR_access_unit* and a buffer register are added to the design. The *DRA_control* module is used to load the Instruction Register, PMP address register and Program Counter. A load to the PC may also be performed, in order to do so the *PC_rw* signals must be asserted one clock cycle earlier by the FSM.
The *buffer_register* is implemented to allow data shares between the *DRA_control* and *CSR_access_unit*. It implements another functionality that allows to set either the MIE register to 1 (trap exit) or the MPIE register to 0 (trap entry) and switch the two bits (MIE and MPIE) on the *data_out* line. This feature is used to modify the *mstatus* register according to [**riscv:privileged**] in no more than two clock cycles.
The *CSR_access_unit* is used to perform register accesses to the CSR register file. During

Exception entry or return the *data_bus_B* must be connected to the *data_in_EC* bus and the *data_in* bus to the *data_out_EC* bus. This is done by implementing a multiplexer at the corresponding buses controlled by the *halt_core* signal.
If an exception or interrupt is raised, the following actions are performed:

- write the current *PC+4* to *mpec*
- modify *mcause* to reflect the cause of the exception/interrupt
- update *mtval* to provide additional information
- modify *mstatus* (safe MIE to the MPIE and set MIE to zero)
- update the PC using the value stored in *mtvec*

If multiple exceptions occur at once, only the highest priority exception is taken, as specified in [**riscv:privileged**].
If an exception is raised, while the processor is in a trap, *mpec*, *mcause* and *mtval* are overwritten. In order to avoid a large hardware register stack, saving the return address and other information stored in those registers is left to the trap handler,.
If an exception entry shall be performed the *exceptionORinterrupt_occured* signal on the FSM is high, if the return signal is high it indicates that an exit shall be performed, however if both signals are high the entry will be performed.

3.6.2 Implementation

The *Exception_Control_FSM* is a meely FSM. It is sensitive to the rising clock edge. The FSM is implemented in two VHDL processes, one synchronous to update the state and one asynchronous to generate the output signals and the next state. Figure 3.20 shows the state diagram of said FSM:

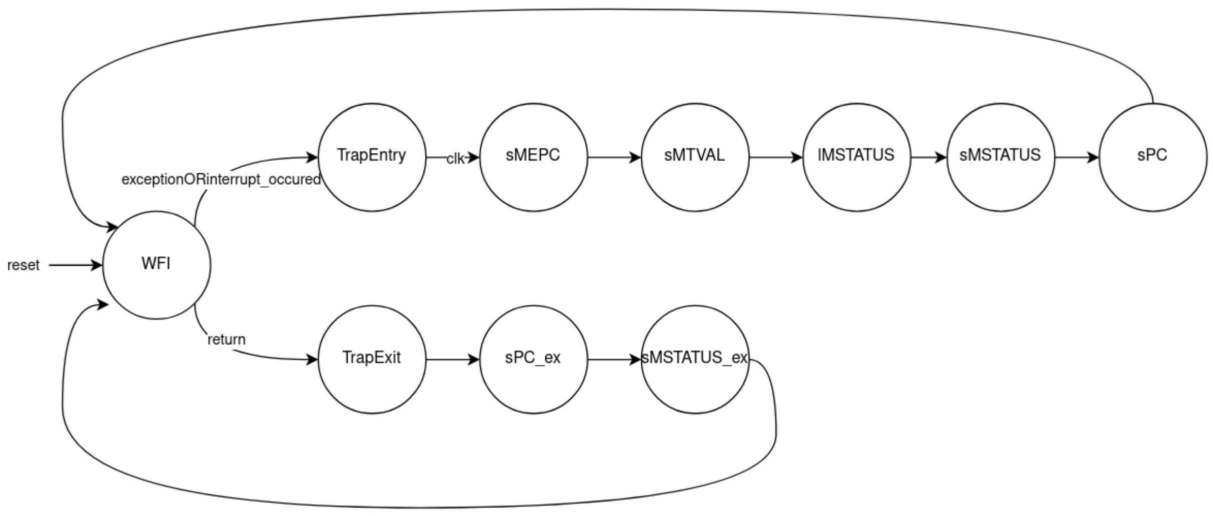


Figure 3.20: state diagram of the *Exception_Control_FSM*

Natively the FSM is in the *WFI* state, waiting for an interrupt, exception or return to be raised. On trap entry, the upper branch of the state diagram is executed, the lower one is performed on trap exit. The *halt_core* signal is set to high, when entering either one of the branches. This stops the core and rerouts all significant control signals of the RF to be connected to the Exception Control unit.
The outputs of each state can be found as a table in the appendix.

3.7 AXI4-Lite Master

To connect the processor to peripherals the AXI4-Lite protocol is used. Due to its popularity many IPs such as BRAM can be connected to each other using an Interconnect. EDRICO contains a master and a slave interface. Both of which are explained in more detail in the following sections:

3.7.1 Architecture and Design

Three substantial building blocks are implemented to form the master interface. Figure 3.21 depicts the architecture of it:

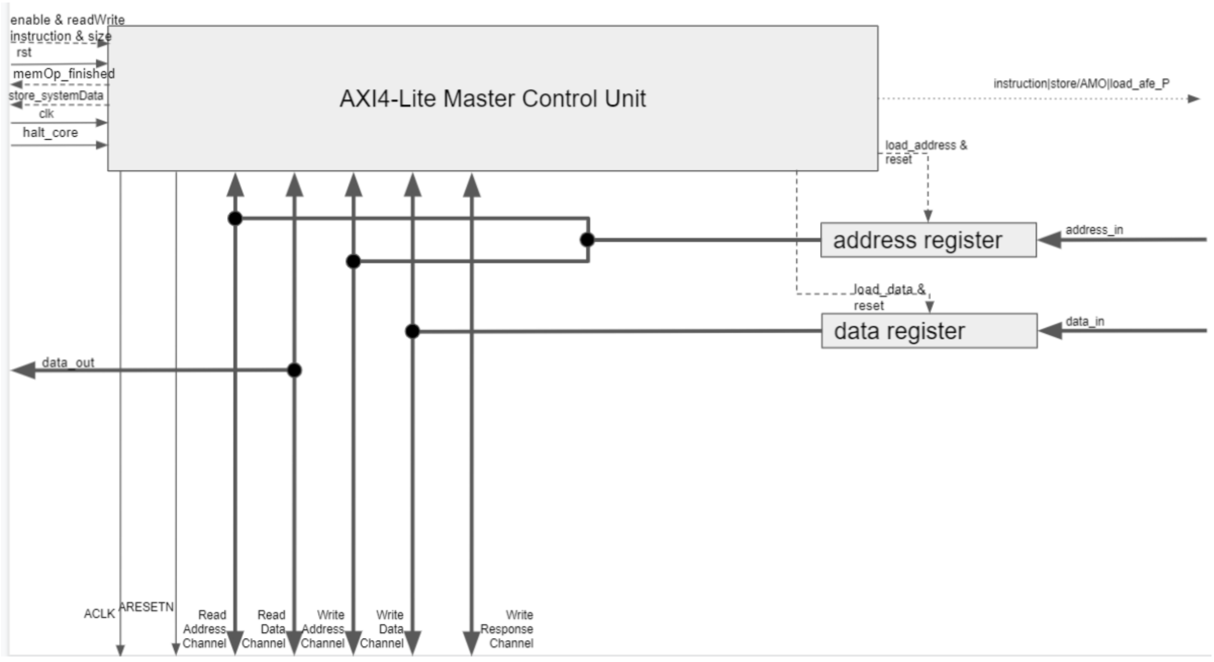


Figure 3.21: AXI4-Lite Master architecture

It includes two registers, for data and address. These are loaded on the rising edge of the system clock, if the corresponding load signal is applied. Both register outputs are constantly connected to the corresponding AXI signals *(AWADDR/ARADDR and WDATA)*.
The *AXI4-Lite Master Control Unit* is in charge of controlling the AXI transfer. It controls the ready and valid signals as well as the register reads and writes. It is clocked with *M_AXI_ACLK*. To reset the module either the reset (active high) or *M_AXI_ARSTN* (active low) may be applied. To start a transfer, the enable signal must be high on a rising clock edge. This will cause the corresponding registers and valid signals to be updated (see section 2.5.2 for more information on the AXI protocol). It is mandatory, that the

correct address and data values are present as soon as the enable signal is high. Otherwise a faulty transfer will be initiated, since the valid signals are set to high even though data and address are not valid yet.
If data is ready to be read from the system bus, it is routed to an output of the master and the *store_systemSystemData* is set to high, in order to ensure a correct read, this process shall not be clocked. The surrounding system is responsible to store the received data in the correct register.
At the end of a data transfer, the *memOp_finished* signal is set to high, informing the Control Unit that it can proceed operations.
If an error occurred during an access, the *instruction_afe_P*, *storeAMO_afe_P* or *load_afe_P* exceptions are raised. And remain high until the *halt_core* signal is applied to the *AXI4-Lite Master Control Unit*.

3.7.2 Implementation

As described in chapter 2 each AXI channel implements a flow-control handshake, using a ready and valid signal. The transmitter interface applies the valid signal as soon as the signal on the corresponding channel is valid. If the receiver is ready to receive data, it ties the ready signal to high. A transfer on one channel is finished as soon as the valid and ready signal are detected to be high on a rising clock edge.
Figure 3.22 shows how the address read(right) and read data(left) handshakes are implemented on EDRICO:

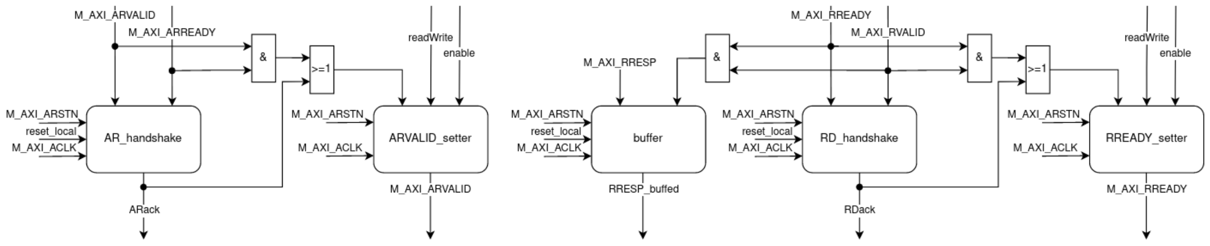


Figure 3.22: read handshake implementation in *AXI4-Lite Control Unit*

Both handshake mechanisms work concurrent and do not interfere with one another. If a handshake is detected, the corresponding acknowledge signal (*ARack; RDack*) is set to high. These acknowledge signals can only be reset by a system wide reset or a local reset initiated by the control FSM described in the following text segment.
M_AXI_ARVALID and *M_AXI_RREADY* are the two handshake signals that are controlled by the implemented master. A setter controls each of these signals. If *readWrite* is low and enable is high, indicating that a read transfer is started, the corresponding output signal is set. It is automatically reset as soon as the handshake is detected (on a

rising clock edge). Since the output signal of the setter is part of the handshake it will de-assert its own reset signal one clock cycle later, causing an unstable valid or ready signal. To prevent this the setter reset is connected to the corresponding acknowledge signal. Guaranteeing that the signal can not be reasserted as long as the AXI transfer is still running.
The read data and write response channel handshakes implement an additional block, buffering the *RRESP* and *BRESP* signals, respectively. Therefore allowing evaluation of the slaves response on a later stage of the transfer.

To control the operation of the AXI master a finite state machine is implemented. Figure 3.23 depicts the corresponding state diagram:

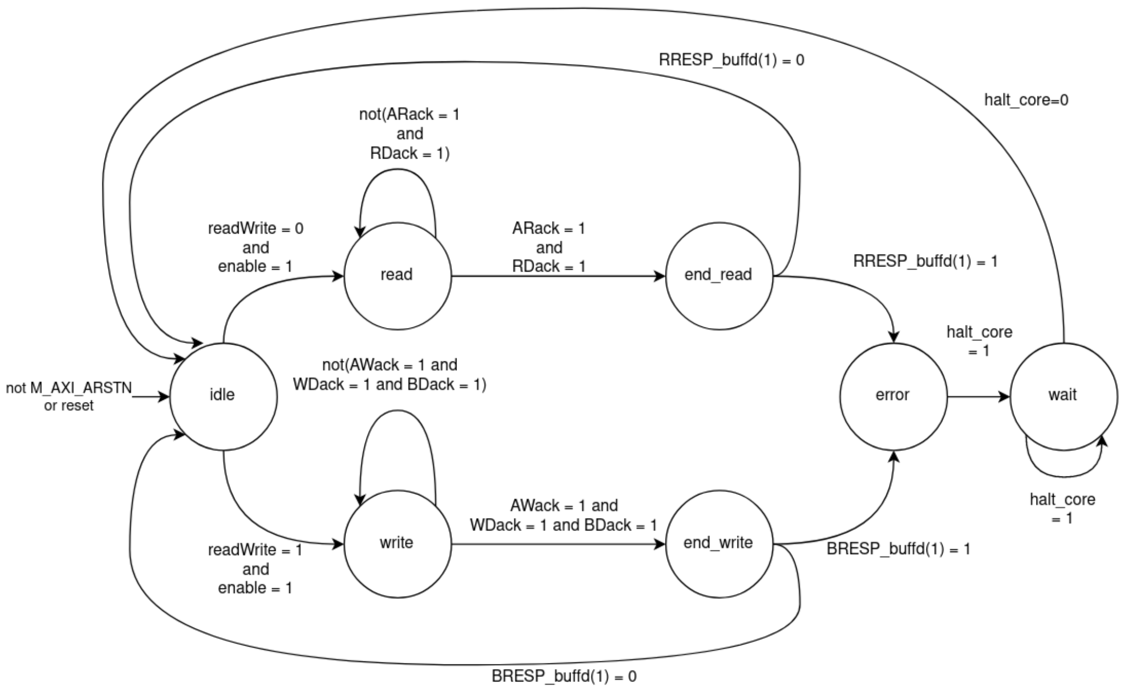


Figure 3.23: control FSM of the *AXI4-Lite Control Unit*

After reset, the FSM remains in *idle* state until the *enable* signal is set to high. It then changes to either *read* or *write*, depending on the value of the *readWrite* signal. The machine remains in the *read/write* state until all acknowledge signals are set to high, hence all necessary handshakes are completed. Performing and detecting the handshake signals independent from the FSM simplifies the design significantly and guarantees that the handshakes do not need to be performed in any particular order.
After performing all expected handshakes, the *end_read/end_write* states are entered, these are used to check if any errors occurred during the transfer.
If that is the case, the *error* state will be entered. The machine remains in *error* state until the trap is handled by the *Exception Control* unit (*halt_core* = 1). The corresponding

access fault exception signals are generated, similar to the exception generation in the *PMP_checker*.
It then changes to *wait* state until *halt_core* is deasserted and the core restarts execution.

Some of the system control signals are generated independently from the AXI control unit, in order to guarantee fast execution and decrease overhead introduced by the state changes inside the FSM. The following figure shows how these are generated:

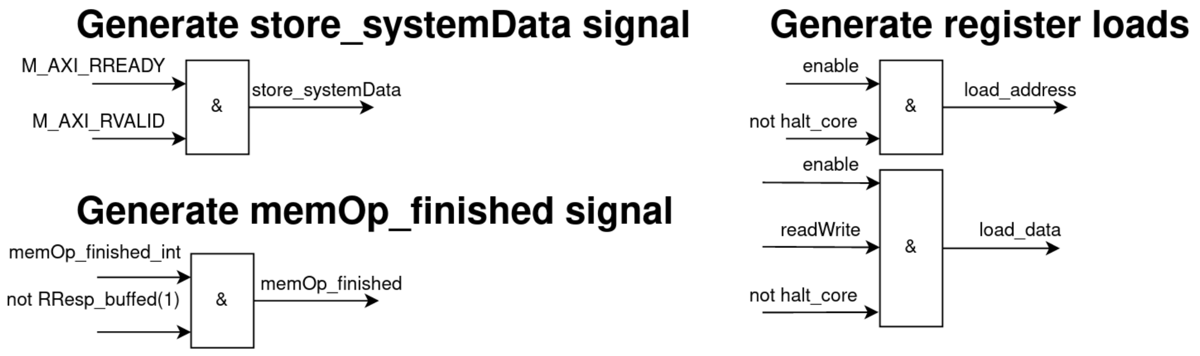


Figure 3.24: Generation of system control signals inside the *AXI4-Lite Control Unit*

Five AXI4-Lite signals are generated without using the FSM, in order to decrease complexity. Figure 3.25 depicts the implementation of said signals:

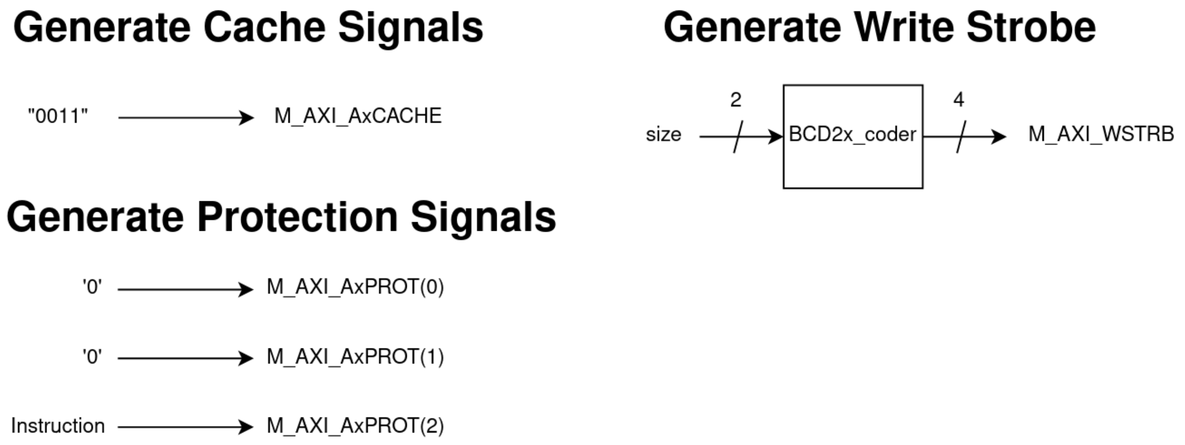


Figure 3.25: Generation of generic AXI4-Lite signals inside the *AXI4-Lite Control Unit*

The *M_AXI_AxCACHE* is hardwired to *"0011"* to determine that the access is normal non-cacheable bufferable. This is specified since EDRICO does not employ any memory consistency model. The first two bits of the protection signals are set to 0, in order to ensure that every access is secure and unprivileged since the only implemented privilege mode is machine-mode. *M_AXI_AxPROT*(2) is adjusted, accordingly to the executed

access (instruction or data).
M_AXI_WSTRB signals what bytes of the transfer are valid to be used. For a word it is set to *0x1*. for a half-word to *0x3* and for a word transfer to *0xF*.

3.8 AXI4-Lite Slave

Some of the RISC-V CSRs must be memory mapped, meaning the must be accessible by other devices via the memory space. The for each CSR is predefined, corresponding to the Core Local Interrupts (CLINTs) module by SiFive since this is the closest thing to an industry standard [**sifive:clint**].

register	address	access	width	description
msip	0x0200_0000	R/W	32-bit	hold software interrupt pending bit
mtimecmp	0x0200_4000	R/W	32-bit	hold time compare value (lower 32 bit)
mtimecmph	0x0200_0004	R/W	32-bit	hold time compare value (upper 32 bit)
mtime	0x0200_BFF8	R/W	32-bit	hold time (lower 32 bit)
mtimeh	0x0200_BFFB	R/W	32-bit	hold time (upper 32 bit)

Table 3.7: Memory Mapped CSRs

3.8.1 Architecture and Design

The AXI4-Lite slave accepts data transfers and performs the reads/writes to the CSR. Figure 3.26 shows the architecture, including the memory mapped CSR:

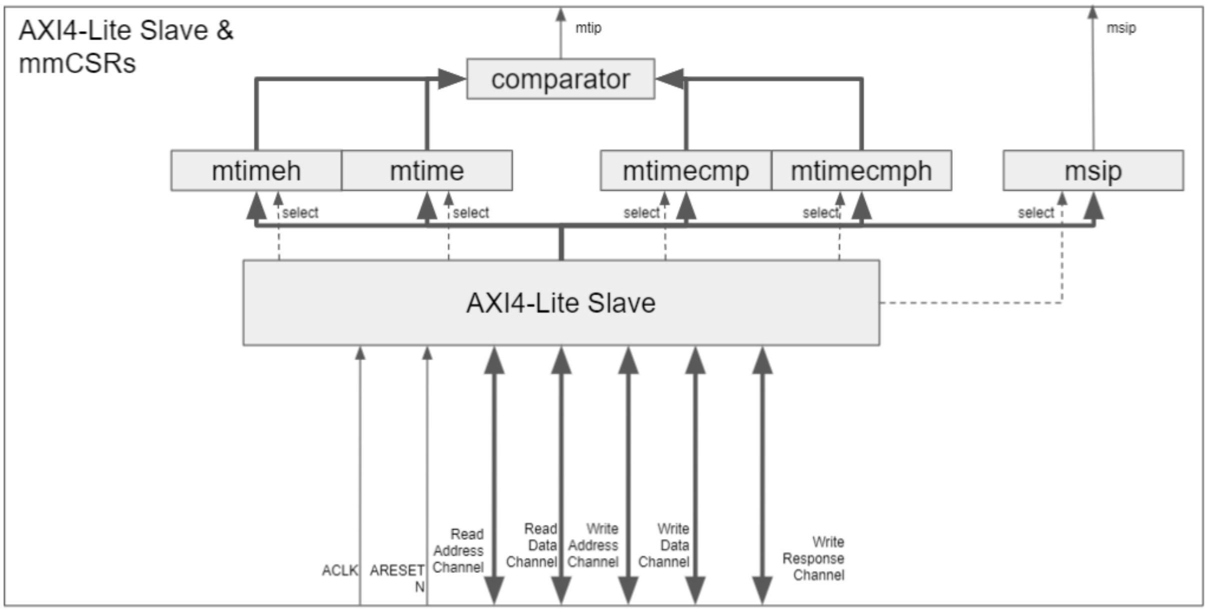


Figure 3.26: AXI4-Lite Slave Architecture

The AXI slave allows reads and writes to the 32 bit registers *mtimeh*, *mtime*, *mtimecmp* and *mtimecmph* as well as the *msip* register.
A comparator is used to trigger a timer interrupt. It compares *mtime* and *mtimecmp*. If the value of *mtime* is equal or greater than the one in *mtimecmp*, the *mtip* signal is asserted.
The interrupt remains posted, until it is cleared by writing to the *mtimecmp* register. Writing a *0xFFFFFFFF* to the *msip* register causes the *msip* signal to be raised. Effectively raising a software interrupt. Table 3.8.1 shows the reset values for each CSR:

register	**value**
msip	0x00000000
mtimeh	0x00000000
mtime	0x00000000
mtimecmph	0xFFFFFFFF
mtimecmp	0xFFFFFFFF

Table 3.8: Memory Mapped CSRs reset values

3.8.2 Implementation

To simplify the development of EDRICO the AXI4-Lite slave is implemented using the automatic AXI4 peripheral generation workflow of Vivado. The generated 5 register salve IP is then modified to include the comperator and to comply to the predefined address space.

4 Test and Verification

As already mentioned in previous chapters, the whole EDRICO CPU is designed and implemented in many units. Since the approach using the V-model consists of working in bottom-up order for the implementation, test and verification processes, the first thing to implement and also test are the units of EDRICO. In the following chapters, the bottom-up way of the right side of the V-model (figure 1.1) is described in detail.

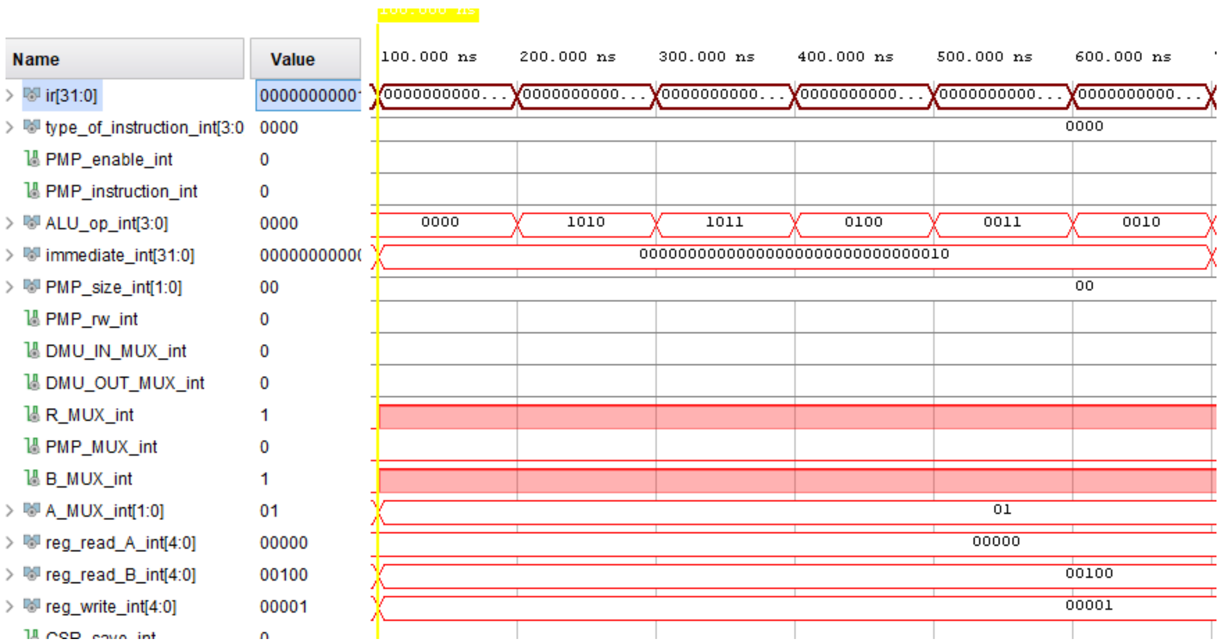
4.1 Unit Verification

The first step in Test and Verification is the Unit Verification. In this step, the lowest units of the project are tested to prevent possible bugs from occurring in later stages of the verification. It is crucial to test every unit of the project because in a later stage of verification it is very complicated and time intensive to localize the bug and fix it in a top-down way. The main goal is to identify and fix bugs as early as possible. In this section, the unit verification process for the CU decoding unit is described in detail.
The task of the decoding unit is to parse the 32-bit instruction string, extract information and set the control signals respectively.
Unit verification starts by inserting source files of the implementation into a Xilinx Vivado©project. To test a unit, it is required to implement a so called testbench. This testbench will serve as a test environment for the Unit Under Test (UUT). Testbenches are used for simulation purpose only (not for synthesis). Therefore, several VHDL constructs like *assert*, *report*, or *loop* can be used. During the testbench simulation, the UUT is stimulated with different input data. In this case, the decoder receives a different 32-bit instruction string. The following code shows how the stimulation process is implemented:

```
stim: process
begin
  -- parse through different instruction strings
  --OPIMM
  ir <= "00000000001000100000000010010011"; --ADDI
  wait for 100ns;
  ir <= "00000000001000100010000010010011"; --SLTI
  wait for 100ns;
  ir <= "00000000001000100011000010010011"; --SLTIU
  wait for 100ns;
  ir <= "00000000001000100100000010010011"; --XORI
  ...
```

Listing 4.1: CU testbench stimulation process

To verify the functionality of the decoder unit, Vivado generates a timing diagram where it is possible to inspect all input and output signals of the UUT. The following figure 4.1 shows the timing diagram for the CU decoder testbench.

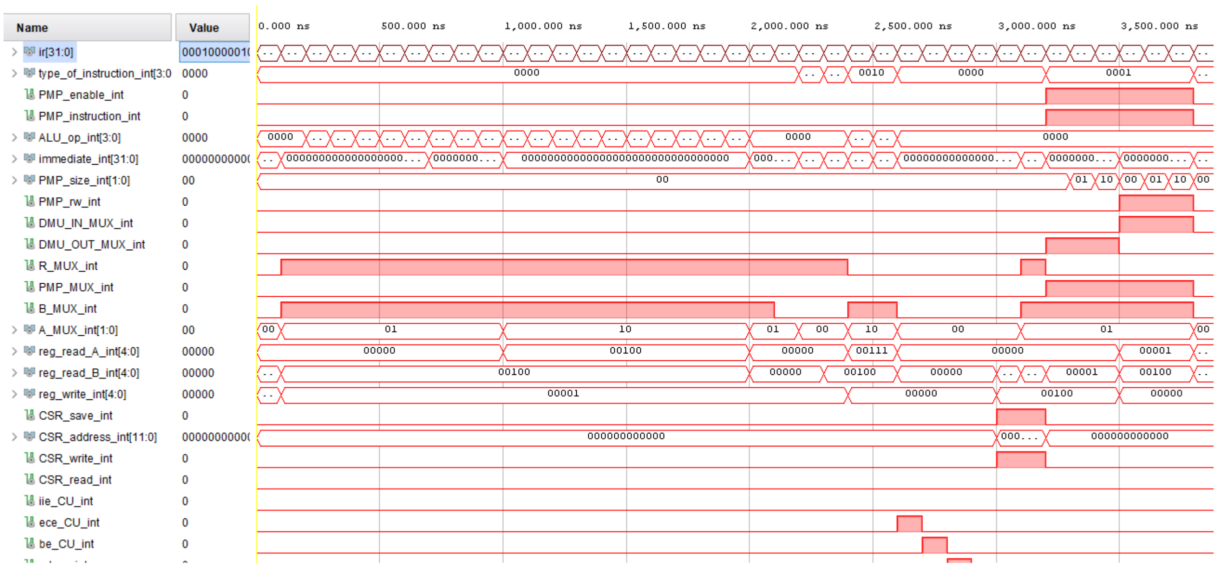


Name	Value
ir[31:0]	0000000000
type_of_instruction_int[3:0	0000
PMP_enable_int	0
PMP_instruction_int	0
ALU_op_int[3:0]	0000
immediate_int[31:0]	0000000000
PMP_size_int[1:0]	00
PMP_rw_int	0
DMU_IN_MUX_int	0
DMU_OUT_MUX_int	0
R_MUX_int	1
PMP_MUX_int	0
B_MUX_int	1
A_MUX_int[1:0]	01
reg_read_A_int[4:0]	00000
reg_read_B_int[4:0]	00100
reg_write_int[4:0]	00001
CSR_save_int	0

Figure 4.1: Vivado timing diagram for OPIMM instructions

For the OPIMM instruction cluster, the relevant output signals are highlighted in red. The top row shows how the instruction string is changing every 100 ns. As a result of that, the *ALU_op* signal changes, as the instruction changes. The first instruction is a *ADDI* instruction which should lead to a 4-bit output signal of **0000**. After that a *SLTI*

instruction is inserted which should lead to a **1010** output (as shown in table 3.3.2). These outputs are correctly set as the timing diagram shows. The same validation technique is executed for all the other relevant signals. After establishing a correct signal for every output and every instruction, the unit can be declared as verified.
Figure 4.2 shows the full timing diagram for the full testbench duration. It is visible that for example the memory signals are only active for memory operations.



Name	Value
ir[31:0]	00010000010
type_of_instruction_int[3:0	0000
PMP_enable_int	0
PMP_instruction_int	0
ALU_op_int[3:0]	0000
immediate_int[31:0]	00000000000
PMP_size_int[1:0]	00
PMP_rw_int	0
DMU_IN_MUX_int	0
DMU_OUT_MUX_int	0
R_MUX_int	0
PMP_MUX_int	0
B_MUX_int	0
A_MUX_int[1:0]	00
reg_read_A_int[4:0]	00000
reg_read_B_int[4:0]	00000
reg_write_int[4:0]	00000
CSR_save_int	0
CSR_address_int[11:0]	00000000000
CSR_write_int	0
CSR_read_int	0
iie_CU_int	0
ece_CU_int	0
be_CU_int	0

Figure 4.2: Vivado timing diagram for full CU decoder testbench

Another unit that had to be verified outside of the control unit is the ALU. Since the complexity of the ALU is not as high as of the control unit, the testbench and also the timing diagram are very simple. The stimulation process of the ALU testbench is shown in the code below:

```
stim: process
begin
  -- set input signal
  in_a <= "00000000000000000000000000000010";
  in_b <= "00000000000000000000000001000000";

  -- set op signal
  alu_op <= "0000"; --ADD
  wait for 100ns;
  alu_op <= "0001"; --SUB
  wait for 100ns;
  alu_op <= "0010"; --AND
  wait for 100ns;
  alu_op <= "0011"; --OR
  wait for 100ns;
  alu_op <= "0100"; --XOR
  wait for 100ns;
  alu_op <= "0101"; --EQUAL
  wait for 100ns;
  alu_op <= "0110"; --NEQUAL
  wait for 100ns;
  alu_op <= "0111"; --shift_left
  wait for 100ns;
  alu_op <= "1000"; --shift_right
  wait for 100ns;
  alu_op <= "1001"; --shift_right (arithmetic)
  ...
```

Listing 4.2: ALU testbench stimulation process

This code shows the first few operations that the ALU performs. First of all a pre defined input is fed to the ALU consisting of *in_a = 0x00000002* and *in_b = 0x00000040*. With these inputs, the different ALU operations are called using the *alu_op* signal. The following table 4.1 will give an overview over what the expected results are:

ALU_OP	ALU_result	branch_re
0000 (ADD)	0x00000042	0
0001 (SUB)	0x0000003e	0
0010 (AND)	0x00000000	0
0011 (OR)	0x00000042	0
0100 (XOR)	0x00000042	0
0101 (EQUAL)	0x00000000	0
0110 (NEQUAL)	0x00000000	1
0111 (shift_left)	0x00000100	0
1000 (shift_right)	0x00000010	0
1001 (shift_right (arith.))	0x00000010	0

Table 4.1: ALU testbench correct outputs

The branch_re output is only relevant for operations that can be called by branch instructions. In this case, the *EQUAL* and *NEQUAL* operations have to deliver a branch response. Since the two inputs are not the same, the *EQUAL* operation shall return 0 while the *NEQUAL* operation shall return 1. The other operations are basic mathematical and arithmetical operations and the output is calculated by just performing the respective operations on the two inputs. After simulating the testbench in Vivado the resulting timing diagram (figure 4.3) shows that everything works as desired.

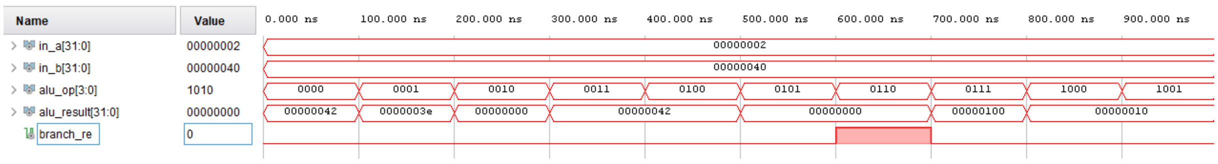


Figure 4.3: Vivado timing diagram for ALU testbench

With the same approach, every other unit is verified and occuring bugs are fixed until simulation and desired output are equal.

4.2 Integration Verification

During this stage of Test and Verification, all of the units have already been tested and verified. In this step, it is verified, that the units created and tested independently can coexist and communicate among themselves. In this project, the Integration Verification consists of creating so called *top-files* which conclude all the corresponding units to a sub system. This section describes how the Control Unit of the EDRICO CPU went through this stage.
Since the CU consists of the units *CU_decoder, CU_execute_enable, CU_FSM and CU_PC*, the top file gets very big since it combines and connects all the mentioned units. As shown in figure 3.3 there are a lot of control signals going out of the control unit and some signals going in. These inputs and outputs are the only signals visible from the 'outside' and therefore are the only signals to be monitored during testing. Since all units are already verified, the internal communication and processing should be working correctly. The creation of the testbench for this integration is much more complex than the testing for the single units. Since the timing of the state machine is working autonomously, the input signal manipulation has to be adapted respectively. Due to this high complexity, a package file was created to outsource input signal manipulation and also output signal monitoring.
In this package file, there are multiple vectors consisting of the different input signals and the desired output signals. An example for these vectors is shown in the code below:

```
type data_stim is array(natural range <>) of std_logic_vector(31 downto 0);
type result_vec is array(natural range <>) of std_logic_vector(149 downto 0);

constant instruction : data_stim(38 downto 0) :=
  (
  1 => "00000000001000100000000010010011",     --ADDI
  2 => "00000000001000100010000010010011",     --SLTI
  ...
  22 => "00000100000100001001000011101111",    --JAL
  23 => "00000010101000100000000011100111",    --JALR
  24 => "10000010011100100000010001110011",    --BEQ
  29 => "00010000010100000000000001110011",    --WFI
  30 => "00000000010000110001001001110011",    --CSRRW
  ...
constant results : result_vec(38 downto 0) :=
  (
  1 => x"00220093" & x"00000008" & ’0’ & ’0’ & "00" & ’0’ & ’0’ & ’0’ & ’1’ & ’0’ & ’1’ & "01" & "00000" & "00100" & "00001" & "000000000000" & ’0’ & ’0’ & ’0’ & "0000" & "100" & ’0’ & ’0’ & ’0’ & ’0’ & x"00000002" & ’1’,     --ADDI
  ...
  10 => x"004200B3" & x"0000002C" & ’0’ & ’0’ & "00" & ’0’ & ’0’ & ’0’ & ’1’ & ’0’ & ’1’ & "10" & "00100" & "00100" & "00001" & "000000000000" & ’0’ & ’0’ & ’0’ & "0000" & "100" & ’0’ & ’0’ & ’0’ & ’0’ & x"00000000" & ’1’,    --ADD
  ...
   35 => x"00120423" & x"00000000" & ’1’ & ’1’ & "00" & ’1’ & ’1’ & ’0’ & ’0’ & ’1’ & ’1’ & "01" & "00001" & "00100" & "00000" & "000000000000" & ’0’ & ’0’ & ’0’ & "0000" & "001" & ’0’ & ’0’ & ’0’ & ’0’ & x"00000008" & ’1’,    --SB
   ...
```

Listing 4.3: CU top testbench package

The *instruction* vector of type *data_stim* contains the 32-bit instruction word. The *results* vector of type *result_vec* contains all the desired output signals for the respective instruction string.
As displayed in the code, every vector contains of in total 38 elements which means that the testbench for the CU integration goes through a *for loops* 38 times to test every single instruction and validate the output signals. To validate the output signals and compare them to the desired output from *results* vector, an if-statement is performed for every output signal and the corresponding position in the *results* vector. This may look for example like stated in following code:

```
if(results(i)(73 downto 69) /= register_read_A) then
  report "register_read_A" & integer'image(i) severity
      error;
  error_flag := true;
  error_cnt := error_cnt + 1;
end if;
if(results(i)(68 downto 64) /= register_read_B) then
  report "register_read_B" & integer'image(i) severity
      error;
  error_flag := true;
  error_cnt := error_cnt + 1;
end if;
  ...
```

Listing 4.4: CU top testbench output validation

With this error checking, the validation process gets easier since not every output has to be checked in the timing diagram. However, the timing diagram is crucial for the validation to proof the timing and correct signal changes. The whole timing diagram for the CU integration is shown in figure 4.4:

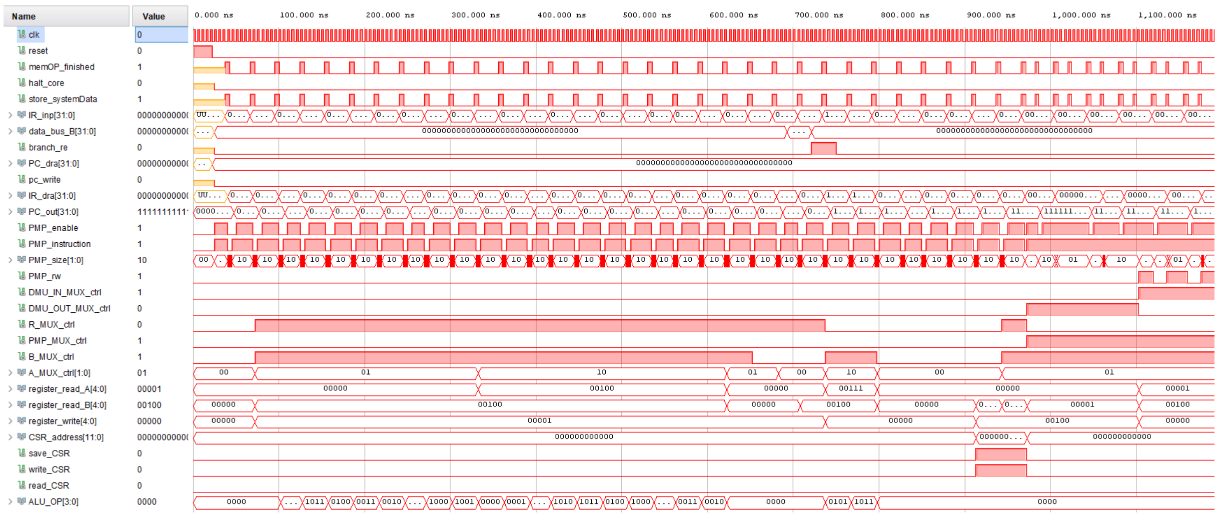


Figure 4.4: Vivado timing diagram for full CU integration testbench

After carefully reviewing all of the output signals and the timing of the CU, the CU integration can be validated. After identifying a few bugs, the CU finally worked as it should and was ready for the next step of Test and Verification.

4.3 System Verification

After successful unit and integration verification, the entire system must be verified. This is done in Simulation to ensure that the system works before performing synthesis and Place and Route (PaR) to map the core in a FPGA.
There are basically two ways to do this. Designated tools exist to test a custom RISC-V core for its functionality. These often require a specific debug interface such as the RISC-V External Debug Support: [**riscv:debug**]. An example for such a verification IP is the Imperas RISC-V Reference Model. Due to the not implemented debug interface as well pressure of time to verify the functionality, another approach for system verification was chosen.
The basic idea is to write a simple test code and define the expected machine state after execution of said code. The code can be written in assembly and assembled using the RISC-V gcc compiler, or any other feasible compiler such as clang. A list of available compilers and software tools can be found at [**RV:software**]. This test code is then executed in simulation.
In order to execute the code a test bench is implemented, containing a the EDRICO IP, an instruction Read Only Memory (ROM), data Random Access Memory (RAM) and a tester Register Transfer Level (RTL) module. The tester is added to the design in order to provide the proper reset and clock signals. As an additional feature it can be configured

to check the machine status after execution and display status messages.
Debug outputs are added to the EDRICO IP for every register inside the RF block as well as the Instruction Register (IR) and PC. The test bench also contains an AXI inteconnect, this allows to connect multiple s to a single AXI-master. The interconnect is, as well as the RAM and ROM, an IP block provided by Xilinx. The test bench can be found in the appendix.
Figure 4.5 shows the memory map of the test bench.

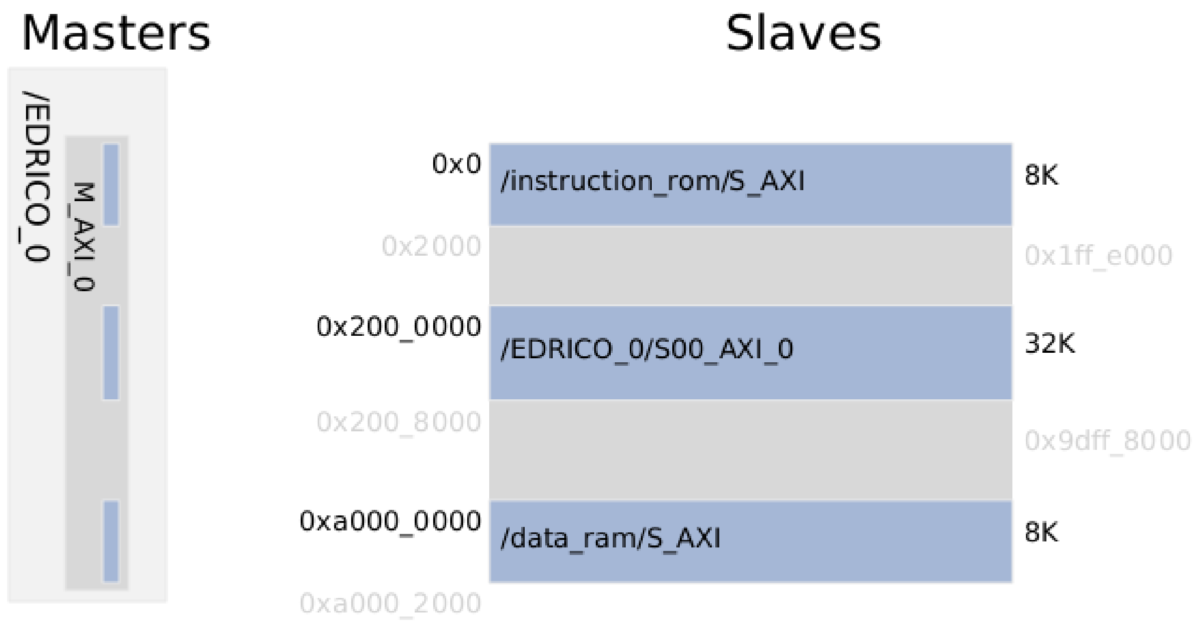


Figure 4.5: memory map of the system verification test bench

The instruction ROM is defined to start at address 0x00000000 it has a size of 8KB. This is sufficient, since the test cases contain a fairly small amount of code to be executed. The same applies to the 8KB data memory at the base address of 0xA0000000.
In between data and instruction memory, a 32KB address space is reserved for the memory mapped CSR.

The test code is written in assembly, it is designed to test every RV32I instruction that is implemented. Therefore in this first test, no Zicsr instructions are tested. Hence only the GPR and PC contents need to be verified after execution. The IR does not need verification, since any error in it will cause the machine to work in an undefined state, which would modify contents of the other registers to be unequal to the expected outcome. Table 4.3 compares expected and actual register values:

Register	Expected	Actual	Register	Expected	Actual
PC	0x000000C0	0x00000000	x16	0x0A000000	0x0A000000
x1	0xA0000000	0xA0000000	x17	0x00010000	0x00010000
x2	0xC0BAD000	0xC0BAD000	x18	0x00000000	0x00000000
x3	0x12345000	0x12345000	x19	0x00000001	0x00000001
x4	0x00001000	0x00001000	x20	0xC0BAC000	0xC0BAC000
x5	0x000000AB	0x000000AB	x21	0xC0BAD000	0xC0BAD000
x6	0x12345014	0x12345014	x22	0x00000000	0x00000000
x7	0x0000C0BA	0x0000D000	x23	0x12344000	0x12344000
x8	0x000000C0	0x00000000	x24	0xD2EF2000	0xD2EF2000
x9	0x12345000	0x13450000	x25	0xF4000000	0xF4000000
x10	0xFFFFC0BA	0xFFFFD000	x26	0x28000000	0x28000000
x11	0xFFFFFFC0	0x00000000	x27	0x00010000	0x00010000
x12	0x00000000	0x00000000	x28	0x00000000	0x00000000
x13	0x123451EA	0x123451EA	x29	0x00000001	0x00000001
x14	0x00000004	0x00000004	x30	0xA0000100	0xA0000100
x15	0xFA000000	0xFA000000	x31	0x000000B8	0x000000B8

Table 4.2: System Verification results and expected values

The test code is made up of 48 instructions, therefore the PC is expected to be:

$$PC = 48 * 4 = 192 = 0xC0$$

after execution. When taking a look at table 4.3 one sees a difference between the expected and actual result for the PC. This problem is caused by the last instruction that is executed. It is a jump back to the start of the address, hence 0x00000000. After investigating the assembly code, the cause for this behavior is found. The last instruction is a Jump and Link Register (JALR) instruction.

```
JAL x31, 8 #jump 8 byte
NOP
JALR x0, x0, 0 #jump to start
```

Listing 4.5: Snippet 1 from the executed test code

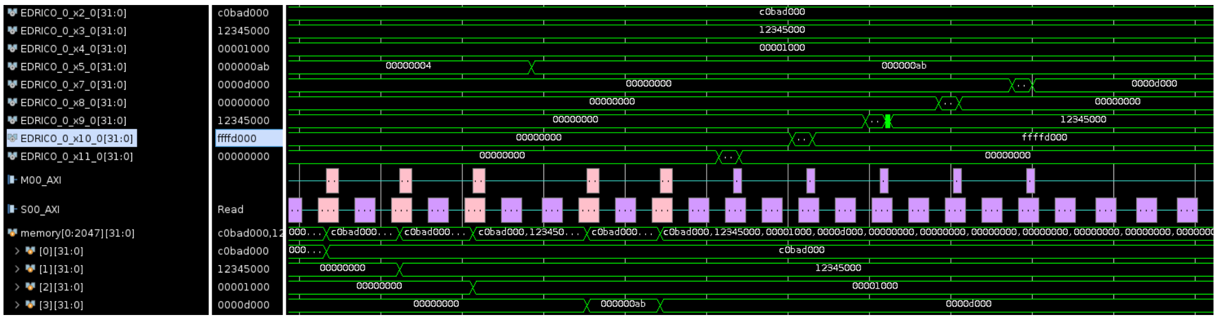
Therefor the PC is set to 0x00000000. Using the simulators waveform viewer, it can be verified that the PC prior to this instruction was set to 0x000000BC. This confirms the correct behavior of the program counter register during execution.
Comparing the remaining results in the table shows another error at registers x7,x8, x10 and x11. The instructions responsible for these registers are shown below:

```
SW x2, 0(x1)
SW x3, 4(x1)
SW x4, 8(x1)
ADDI x5, x0, 0xAB
SB x5, 13(x1)
SH x2, 14(x1)

LB x11, 3(x1) #expect 0xFFFFFFC0
LH x10, 2(x1) #expect 0xFFFFC0BA
LW x9, 4(x1) #expect 0x12345000
LBU x8, 3(x1) #expect 0x000000C0
LHU x7, 2(x1) #expect 0x0000C0BA
```

Listing 4.6: Snippet 2 from the executed test code

This code snippet shows how the memory at base address x1 is modified by multiple store word, half-word and byte instructions. In the second part of the code x11 to x7 are modified by load instructions. It is interesting to note, that only x9 contains the expected results, and that the instruction modifying x9 is a load word instruction. To see if the error is caused by the load instructions or if the store instructions are wrong, the memory is checked. Figure 4.6 shows the memory in the simulation wave form view:



```
EDRICO_0_x2_0[31:0]       c0bad000
EDRICO_0_x3_0[31:0]       12345000
EDRICO_0_x4_0[31:0]       00001000
EDRICO_0_x5_0[31:0]       000000ab
EDRICO_0_x7_0[31:0]       0000d000
EDRICO_0_x8_0[31:0]       00000000
EDRICO_0_x9_0[31:0]       12345000
EDRICO_0_x10_0[31:0]      ffffd000
EDRICO_0_x11_0[31:0]      00000000
M00_AXI
S00_AXI                   Read
memory[0:2047][31:0]      c0bad000,12
  [0][31:0]               c0bad000
  [1][31:0]               12345000
  [2][31:0]               00001000
  [3][31:0]               0000d000
```

Figure 4.6: wave from view of the memory, AXI4-Lite transfer, x2-x5 and x7-x11 during System Verification

The write transfers are displayed in the light pink-orang-ish colour whereas reads are marked in purple. As verified by comparing the memory contents with the registers x2-x4, all three store word operations are executed correctly. The store byte operation to the 0xA000000D should load 0x0000AB00 into the third memory array, followed by a store half-word to 0xA000000E which should modify the memory contents to be 0xD000AB000. However, the fourth memory element is set to 0x000000AB instead of 0x0000AB00. This is then again overwritten to be 0x0000D000. When taking a closer look at the AXI transfer everything seems fine, the right write addresses are specified and the write strobe signal (which indicates what bytes of the transfer are valid) is correct.
This leads to the conclusion that the mismatch in expected and actual data is not caused by the core itself. Much rather the used memory is not properly byte but word addressable. Future tests must verify this.

One important measured value to compare processor performances is the CPI, hence the amount of clock cycles are needed to execute an instruction. As described earlier, the CPI is calculated using the values of the *mcycle* and *minstret* registers:

$$CPI = \frac{mcycleH >> 32 + mcycle}{minstretH >> 32 + minstret} = \frac{0x207}{0x2b} = 12.07$$

Hence in average every instruction takes 12.07 clock cycles to execute. Of course this value is highly dependent on the ratio of load/store instructions to other operations. An instruction accessing memory takes 18 cycles, whereas normal operations take 10 clock cycles. Therefore improving memory access time will decrease the CPI. Of course this assumption is very vague since, according to Amdahl's law, the improvement will not be linear [**patterson:2017**].

5 Lessons Learned

During the different phase of the project multiple issues arose from which important lessons for future projects or continuations of this project can be learned. The first problem that was encountered during this thesis surfaced due to underestimating the complexity of this project. Even though RISC is short for Reduced Instruction Set Computer, this does not mean that the complexity of such a machine is trivial, especially for ongoing bachelor students with limited experience in digital design, processing architecture and VHDL.
The next lesson is strongly related to the first one, but is not solely applicable to the development of a RISC CPU. Despite thoroughly planning every module and nearly every clock cycle of the design, multiple issues were encountered during test and verification. The lesson learned here is to always include extra time for verification, whilst planning the project time line. Two or three weeks are probably not enough to test a design that was developed over the course of approximately five months.
As a last item to lean from, the AXI4-Lite master serves as an example. The design of it is quite complex and took a lot of time even though it is not very RISC-V related. Due to the development from scratch it includes only limited functionality, e.g. burst transactions are not supported. Sometimes it is better to use IP blocks for such generic components. This will decrease time required for development and allow the project to focus on other more important tasks, such as verification.

6 Future Work

Even thought EDRICO is verified to execute all RV32I instructions correctly, there is still a lot of work to be done in order to bring this project to live. The following list contains all steps that are necessary to verify and finish version one of EDRICO:

- verify the correct execution of the Zicsr instruction
- execute acceptance verification (test EDRICO on actual hardware, e.g. a Spartan 6 FPGA or the Arty Z7 development board)
- port benchmarks like SPEC or Coremark to the platform
- run and evaluate said benchmarks

Extended future work could contain developing an improved version two of EDRICO. The following list contains ideas that did not make it to implementation in version one but could be added to future implementations:

- increase performance of the instruction fetches by optimizing the time to access main memory using the AXI4-Lite master
- increasing overall memory performance by adding L1 instruction and data caches
- implementing a small scale pipeline into the design
- implementing a proper debug specification, such as [**riscv:debug**]
- allow atomic operations by implementing the corresponding ISA extension
- add further ISA extensions such as single precision floating point support

7 Conclusion

During the course of this project, a preliminary version of the proposed processor design was developed, implemented and partly verified. Using the V-Modell project management approach, the design was developed leveraging a top-down line of action. Step by step decreasing granularity of the design from Requirements, System Design, Architecture Design to finally Module Design and Implementation.
Accompanying documents are provided to each of the corresponding V-Modell design phases. These mark milestones in the project and provide a good overview of the different levels of abstraction.
EDRICO implements the full RV32I ISA including the Zicsr instruction, running in machine-mode only. It is composed of a *Control Unit*, ALU, *Register File*, *Exception Control Unit*, *PMP & PMA checker* as well as a AXI4-Lite master and slave interface. It executes one instruction at a time by running the fetch, decode and execute phases. All required exceptions as well as the timer and software interrupt are implemented and can be handled by the machine.
EDRICO is shown to work in simulation with an average CPI of 12,07. Especially the time required for memory access imposes a significant overhead to the execution.
Future improvements are best deployed on increasing memory performance, for example by adding a cache structure. The source code, documentation and accompanying documents can be found under the official github repository of EDRICO: Github Repository.

Appendix

.1 Exception Control FSM States

STATE	OUTPUT
WFI	–halt core signal halt_core <= ’0’; –buffer register signals buffer_register_w <= ’0’; buffer_register_CSR_DRA <= ’0’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’0’; load_mstatus <= ’0’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’1’;

TrapEntry	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’1’; buffer_register_CSR_DRA <= ’0’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’1’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’0’; load_mstatus <= ’0’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

sMEPC	–halt core signal
	halt_core <= '1';
	–buffer register signals if (save_address or save_IR or save_PC) = '1' buffer_
	buffer_register_CSR_DRA <= '0';
	modify_mstatus_EI <= '0';
	modify_mstatus_RET <= '0';
	if not (save_address or save_IR or save_PC) = '1' reset_buffer <
	–DRA signal
	PC_rw <= '0';
	–DRA_controll signals
	if save_PC = '1' then load_PC <= '1';
	if save_IR = '1' then load_IR <= '1';
	if save_address = '1' then load_PMP <= '1';
	–CSR_access_unit signals
	load_mepc <= '0';
	load_mtvec <= '0';
	load_mstatus <= '0';
	store_mepc <= '1';
	store_mcause <= '0';
	store_mtval <= '0';
	store_mstatus <= '0';
	–arbiter buffer control
	local_reset <= '0';
	buffer_arbiter <= '0';

sMTVAL	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’0’; buffer_register_CSR_DRA <= ’0’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’0’; load_mstatus <= ’0’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’1’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

lMSTATUS	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’1’; buffer_register_CSR_DRA <= ’1’; modify_mstatus_EI <= ’1’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’0’; load_mstatus <= ’1’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

sMSTATUS	–halt core signal
	halt_core <= '1';
	–buffer register signals
	buffer_register_w <= '0';
	buffer_register_CSR_DRA <= '0';
	modify_mstatus_EI <= '1';
	modify_mstatus_RET <= '0';
	reset_buffer <= '0';
	–DRA signal
	PC_rw <= '0';
	–DRA_controll signals
	load_PC <= '0';
	load_IR <= '0';
	load_PMP <= '0';
	–CSR_access_unit signals
	load_mepc <= '0';
	load_mtvec <= '0';
	load_mstatus <= '0';
	store_mepc <= '0';
	store_mcause <= '0';
	store_mtval <= '0';
	store_mstatus <= '1';
	–arbiter buffer control
	local_reset <= '0';
	buffer_arbiter <= '0';

lMTVEC	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’1’; buffer_register_CSR_DRA <= ’1’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’1’; load_mstatus <= ’0’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

sPC	–halt core signal
	halt_core <= '1';
	–buffer register signals
	buffer_register_w <= '0';
	buffer_register_CSR_DRA <= '0';
	modify_mstatus_EI <= '0';
	modify_mstatus_RET <= '0';
	reset_buffer <= '0';
	–DRA signal
	PC_rw <= '1';
	–DRA_controll signals
	load_PC <= '0';
	load_IR <= '0';
	load_PMP <= '0';
	–CSR_access_unit signals
	load_mepc <= '0';
	load_mtvec <= '0';
	load_mstatus <= '0';
	store_mepc <= '0';
	store_mcause <= '1';
	store_mtval <= '0';
	store_mstatus <= '0';
	–arbiter buffer control
	local_reset <= '1';
	buffer_arbiter <= '0';

TrapExit	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’1’; buffer_register_CSR_DRA <= ’1’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’1’; load_mtvec <= ’0’; load_mstatus <= ’0’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

sPC_ex	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’1’; buffer_register_CSR_DRA <= ’1’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’0’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’1’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’0’; load_mstatus <= ’1’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’0’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

sMSTATUS_ex	–halt core signal halt_core <= ’1’; –buffer register signals buffer_register_w <= ’0’; buffer_register_CSR_DRA <= ’0’; modify_mstatus_EI <= ’0’; modify_mstatus_RET <= ’1’; reset_buffer <= ’0’; –DRA signal PC_rw <= ’0’; –DRA_controll signals load_PC <= ’0’; load_IR <= ’0’; load_PMP <= ’0’; –CSR_access_unit signals load_mepc <= ’0’; load_mtvec <= ’0’; load_mstatus <= ’0’; store_mepc <= ’0’; store_mcause <= ’0’; store_mtval <= ’0’; store_mstatus <= ’1’; –arbiter buffer control local_reset <= ’0’; buffer_arbiter <= ’0’;

Table .1: EC FSM states and corresponding output signals

.2 AXI4-Lite master FSM States

STATE	**OUTPUT**
idle	size_buf <= "00"; reset_local <= ’1’; error_detected <= ’0’; memOp_finished_int <= ’0’;

read	size_buf <= size;
	reset_local <= '0';
	error_detected <= '0';
	memOp_finished_int <= '0';

end_read	reset_local <= '0';
	error_detected <= '0';
	memOp_finished_int <= '1';

write	size_buf <= size;
	reset_local <= '0';
	error_detected <= '0';
	memOp_finished_int <= '0';

end_write	reset_local <= '0';
	error_detected <= '0';
	memOp_finished_int <= '1';

error	reset_local <= '0';
	error_detected <= '1';
	memOp_finished_int <= '0';

wait	reset_local <= '0';
	error_detected <= '0';
	memOp_finished_int <= '0';

Table .2: EC FSM states and corresponding output signals