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

Hereby we solemnly declare:

1. that this Semester Paper , titled *EDRICO - Educational DHBW RISC-V Core* is entirely the product of our own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. we have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Semester Paper has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. we have not published this Semester Paper in the past;
5. the printed version is equivalent to the submitted electronic one.

We are aware that a dishonest declaration will entail legal consequences.

Friedrichshafen, June 2021

Levi-Pascal Bohnacker, Noah Wölki

Abstract

Hlalo dasdoaishdlohiasd

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
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

List of Figures

List of Tables

Listings

1 Introduction

These days one of the key benchmarks for technology is processing speed and calculation power. To realize mathematical operations and execute programs, different platforms can be utilized. The most commonly used unit is the standard processor consisting of transistors realized on silicium and other materials. Another crucial technology that is gaining more attention is the so-called Field Programmable Gate Array (FPGA). The FPGA consists of logical units that can be wired and configured individually for the required use-case. The advantage of FPGA is that the speed of applications can be drastically increased since the hardware will be very optimized for the specific application. This project aims to develop a Interlectual Property (IP)-core based on the Open Source Instruction Set RISC-V. The goal is to build a reusable unit of logic that can interpret compiled C-Code. The IP core is realized in the Very High Speed Integrated Circuit Hardware Description Language (VHDL) language and will be deployed on a FPGA. IP Cores are used in every computer, phone and electronic device that requires to execute some computational function. The developers of these IP Cores are big companies like Intel, ARM or AMD. These IP Cores and Instruction Sets are strictly licensed and not available for everyone. For the development of an own IP Core the Instruction Set is the main source of information and therefore the RISC-V open-source Instruction Set is used for this project.

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. MicroMagic for example announced in 2020 a chip with a total CoreMark score of 13000 and an incredible 110000 Coremark/Watt. This poses a significant development and is approximately 10 times better than any CISC, RISC or MIPS implementation in terms of Performance per Watt. 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 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.

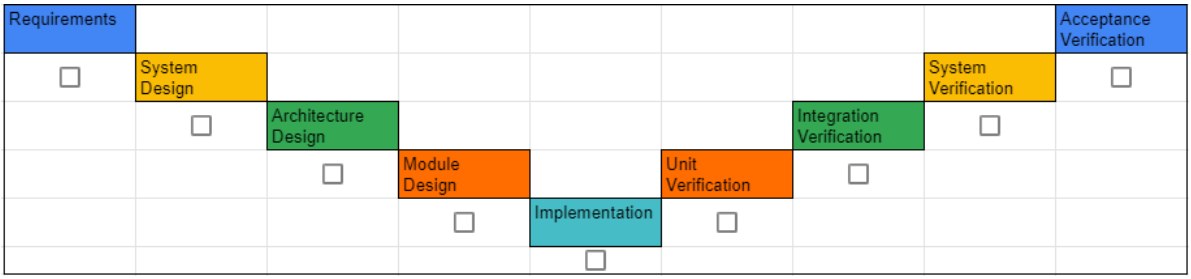

Figure 1.1: V-Model

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)[ORe21]. 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[ORe21].
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 [ORe21]. 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 [ORe21]. 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[ORe21]. 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[ORe21]. 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) Instruction Set Architecture (ISA), more information on this architecture can be found in [HP20].
One of the latest developments of Reduced Instruction Set Computer (RISC) based computers is the developement of the RISC-V ISA in 2010.

2 Basics

This chapter will give a brief overview over some topics that are essential for this project.

2.1 Processing Units

Harvard, Neumann, ALU, Control Unit

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 [Hel16]

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. 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.

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. [Ges14]
Working with benchmarks, a few principles have to be kept in mind. These vital characteristics of benchmarks are [KLK20]:

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 [KLK20]

In this thesis, the main focus will be on the so called $\langle Coremark \rangle$ and $\langle SPECint \rangle$ 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 [SPE06]

2.5 Memory Management

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 [HP17]. 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 2.1 shows an example of a possible memory hierarchy:

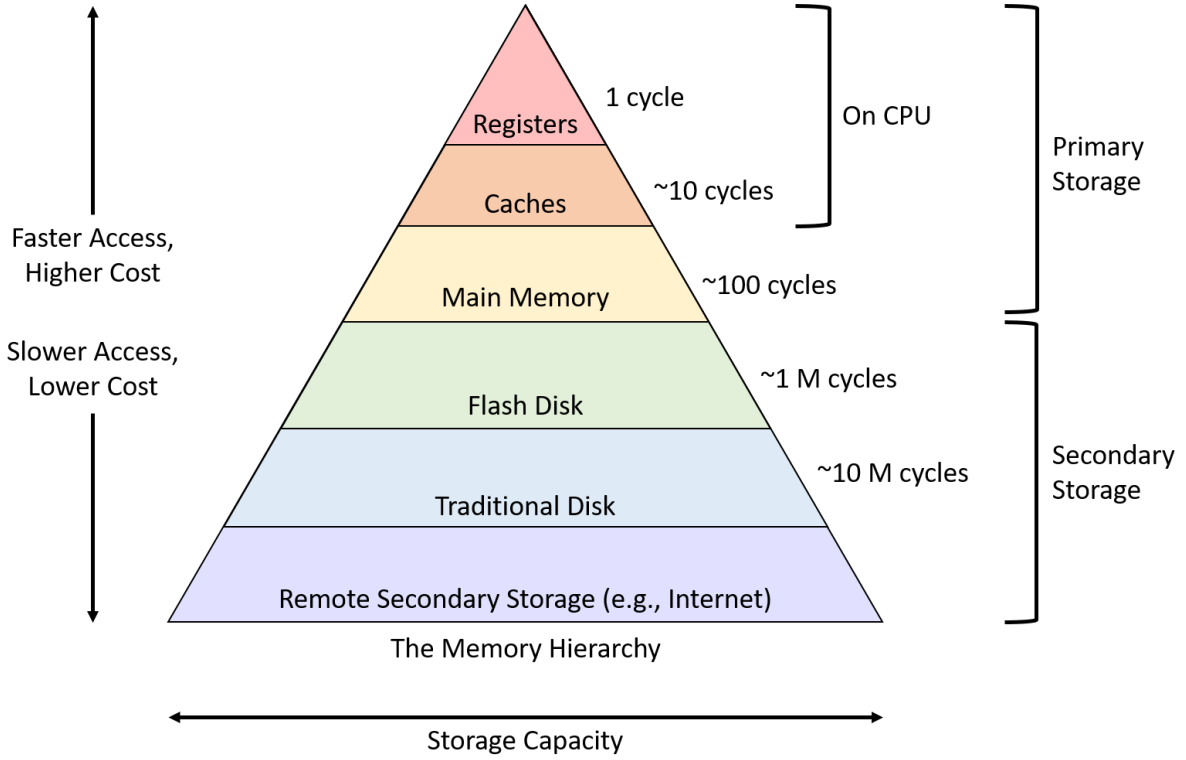

Figure 2.1: Generic Memory Hierarchy [Sys19]

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 [HP17].
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, 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 [HP17].
Different ways of increasing cache performance are described in [HP17].

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 Advanced RISC Machiness (ARMs) Advanced Microcontrol Bus Architecture (AMBA).
The following section will provide an overview of the Advanced eXtensible Interface 4 Lite (AXI4-Lite) protocol which is implemented in the Educational DHBW RISC-V Core (EDRICO).
An AXI interface has four independent channels: read address, write address, read data, write data and write response. Figure 2.2 and 2.3 visualizes these three channels:

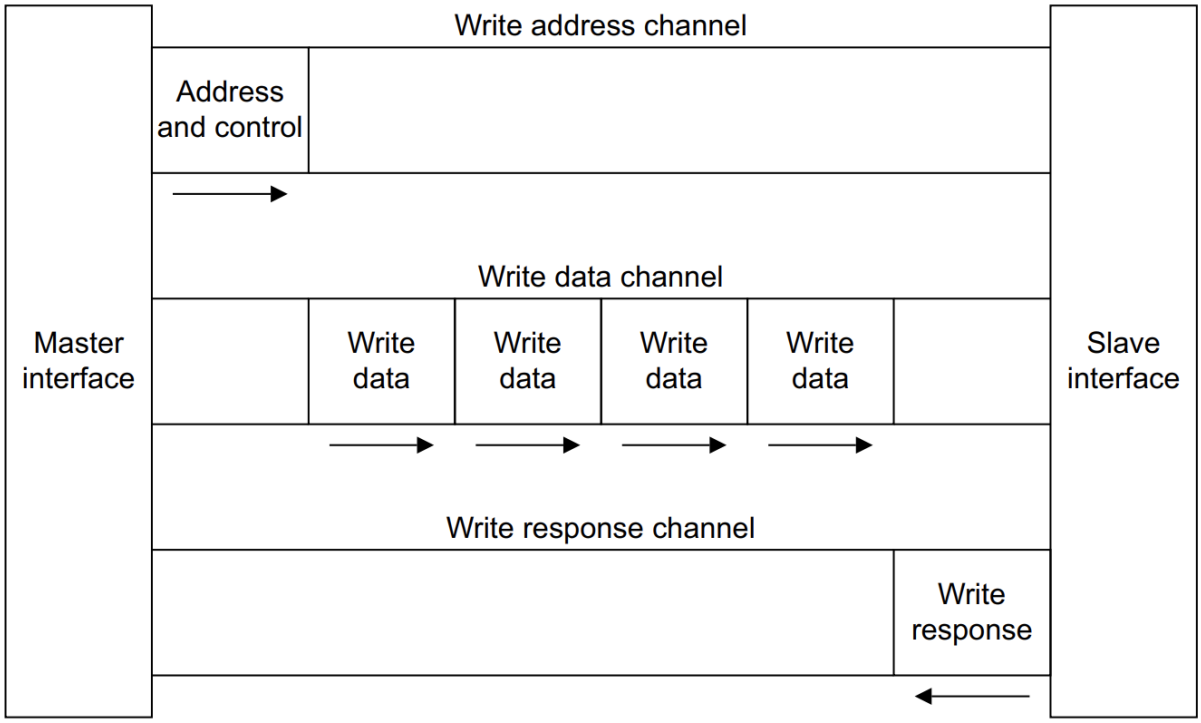

Figure 2.2: AXI4 write channels [ARM13]

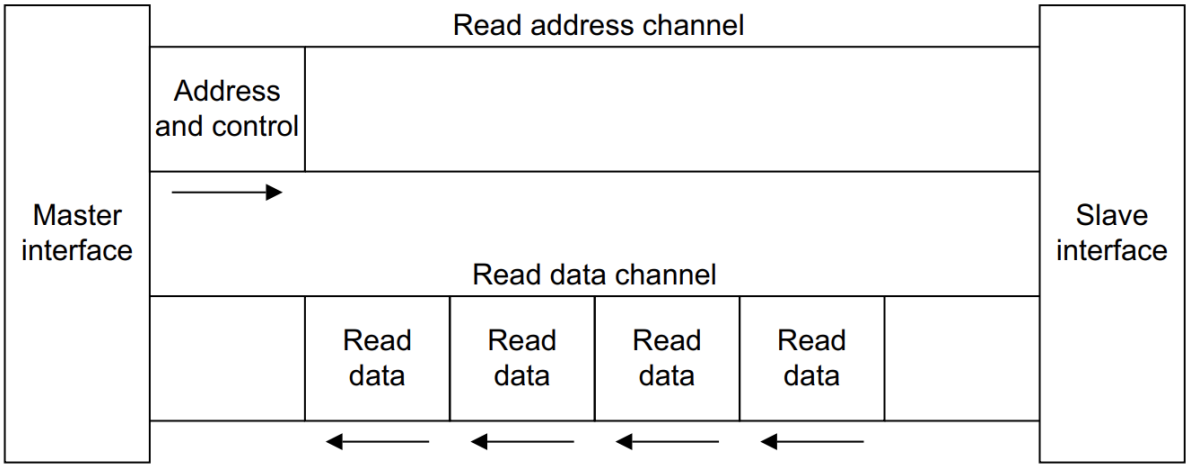

Figure 2.3: AXI4 read channels [ARM13]

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 2.4 shows how the handshake is performed:

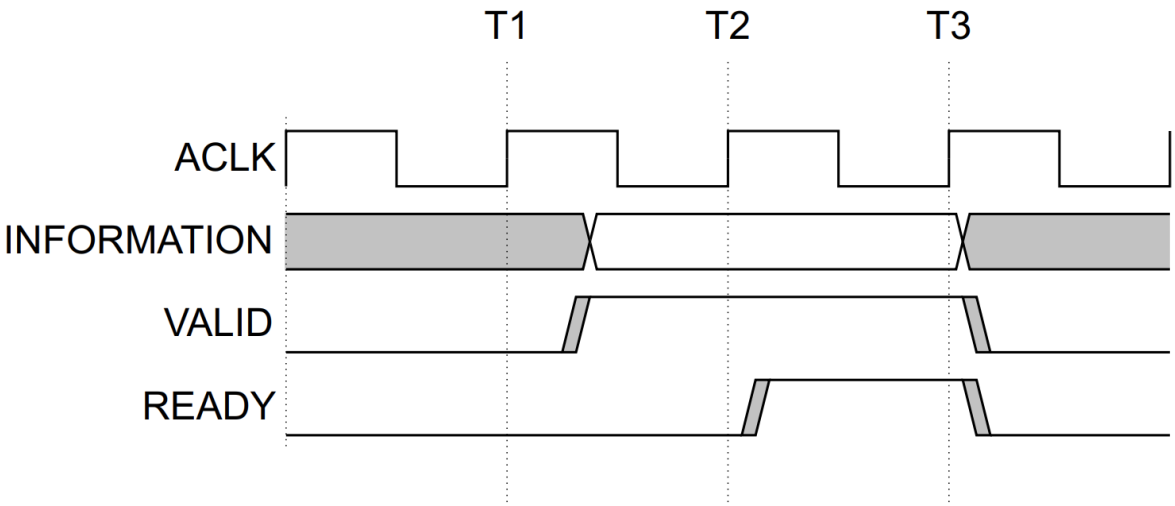

Figure 2.4: AXI handshake [ARM13]

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 [ARM13]. 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, FPGA 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 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. 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 of an FPGA, the basic architecture always remains the same. An array of logic cells and building blocks of different features 9 like BRAM and DSP slices are connected to each other through configurable routing channels. Figure 2.5 shows the basic architecture of a Xilinx FPGA:

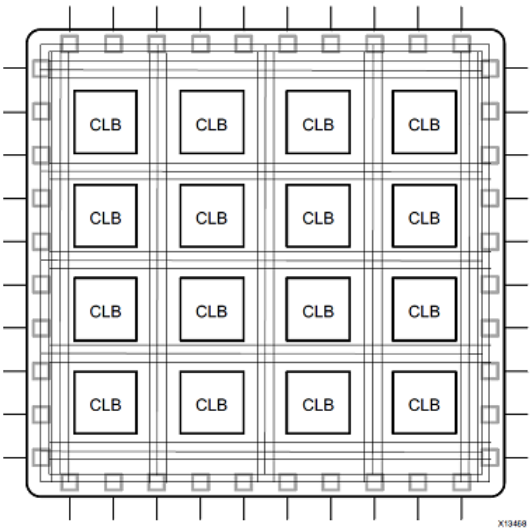

Figure 2.5: Xilinx FPGA [Xil17]

2.7 Hardware Description Languages

3 EDRICO

The Proposed Processor design named Educational DHBW RISC-V Core (EDRICO) implements a basic 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-Mode 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, store is performed for every instruction one at a time.
Figure 3.1 shows the full overview of the processor design:

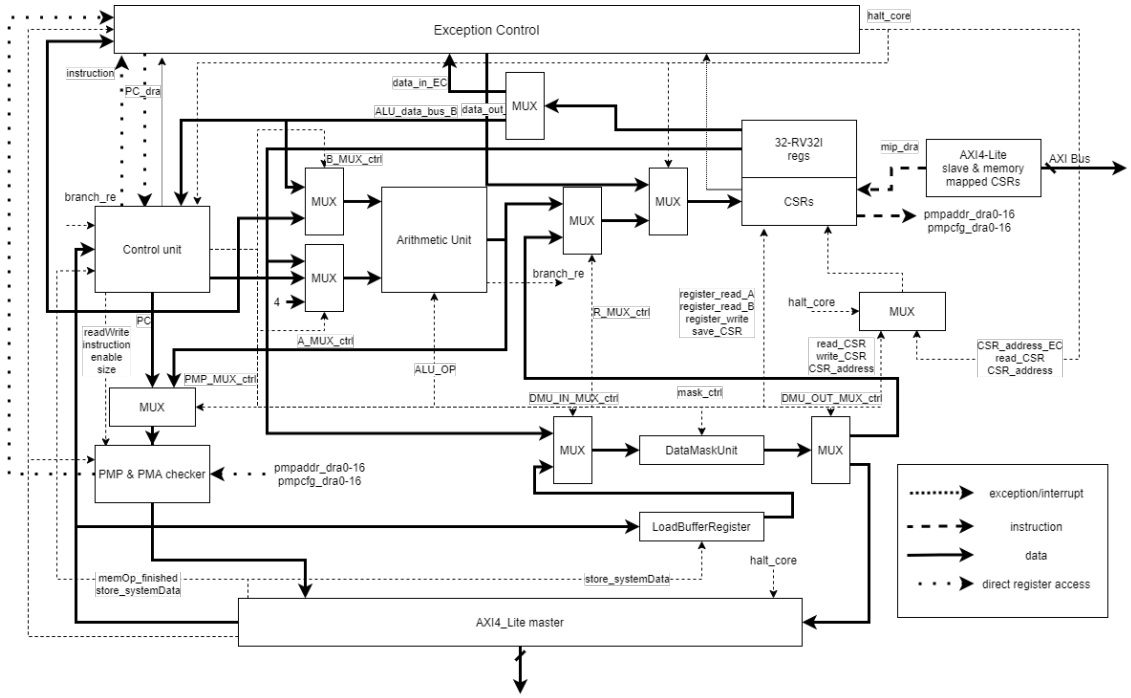

Figure 3.1: EDRICO Overview

Its main components are the Exception Control, Control Unit, Arithmetic Unit, 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 sections 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:

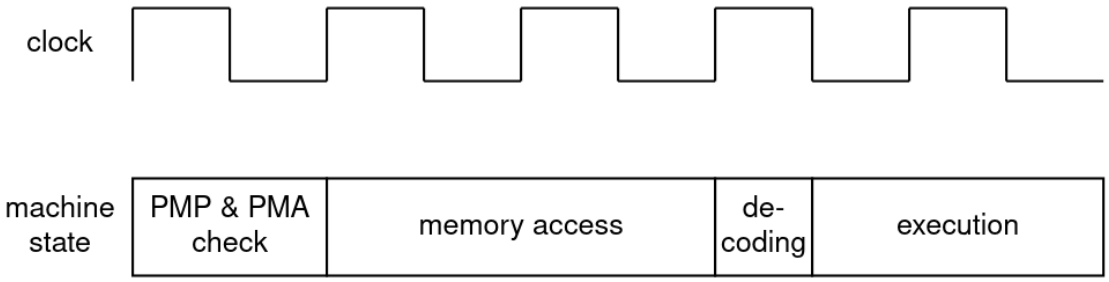

Figure 3.2: Generic Data Path

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.

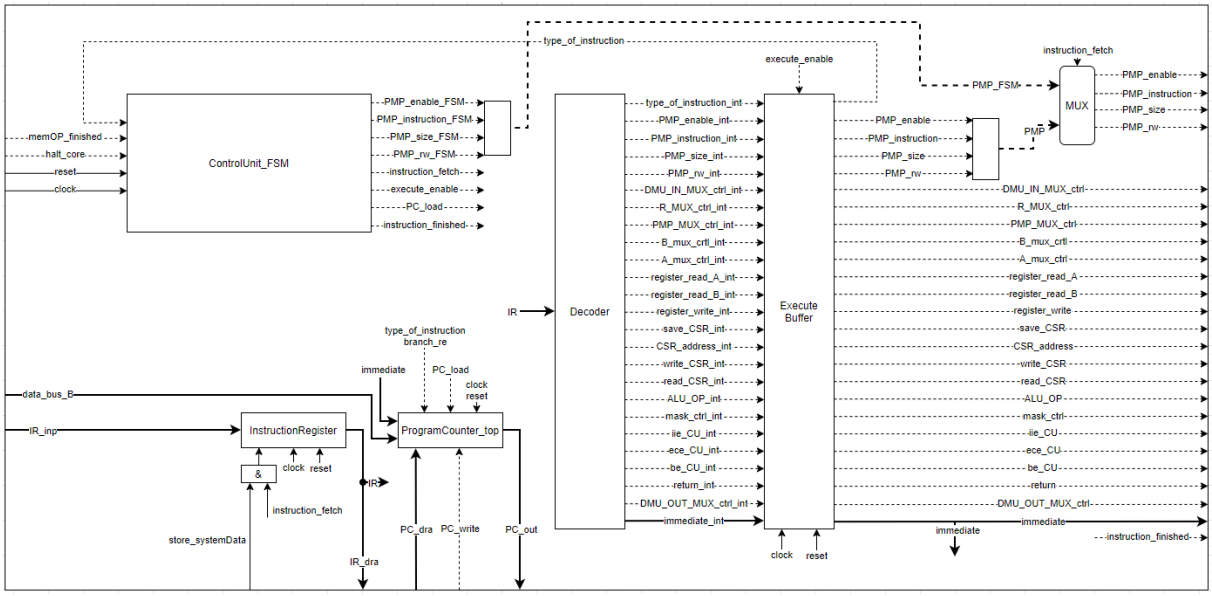

Figure 3.3: Control Unit Architecture

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 Finite State Machine (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.

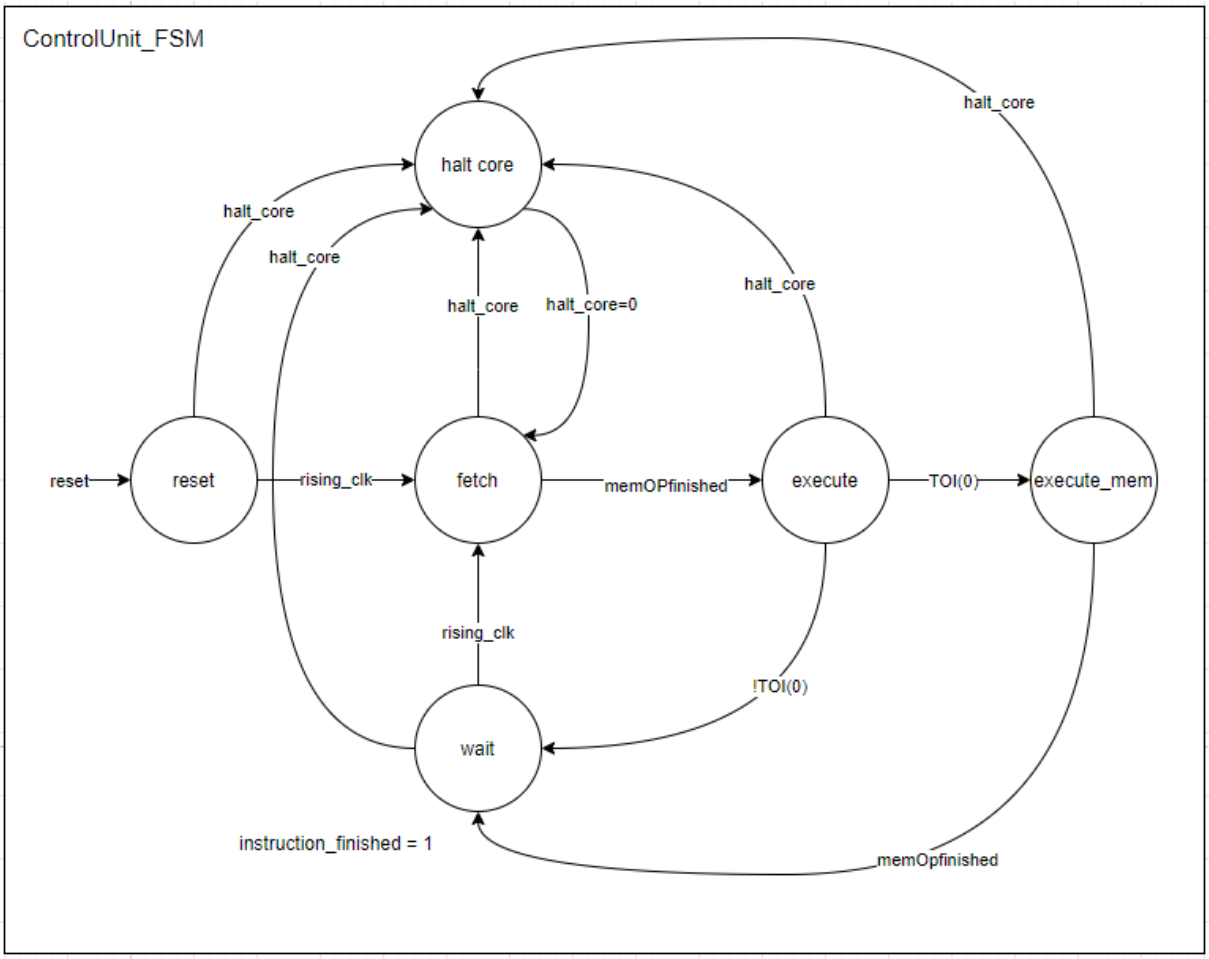

Figure 3.4: Control Unit FSM overview

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.

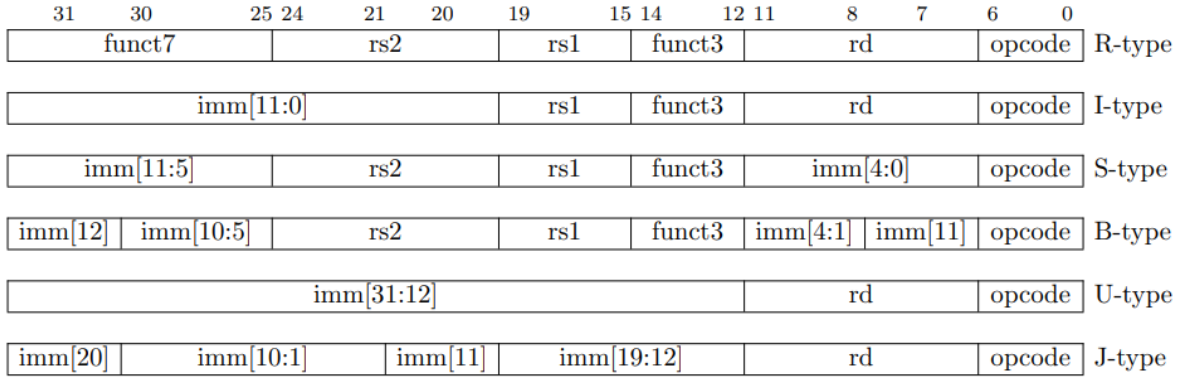

Figure 3.5: RISC-V Instruction formats **riscv**

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:

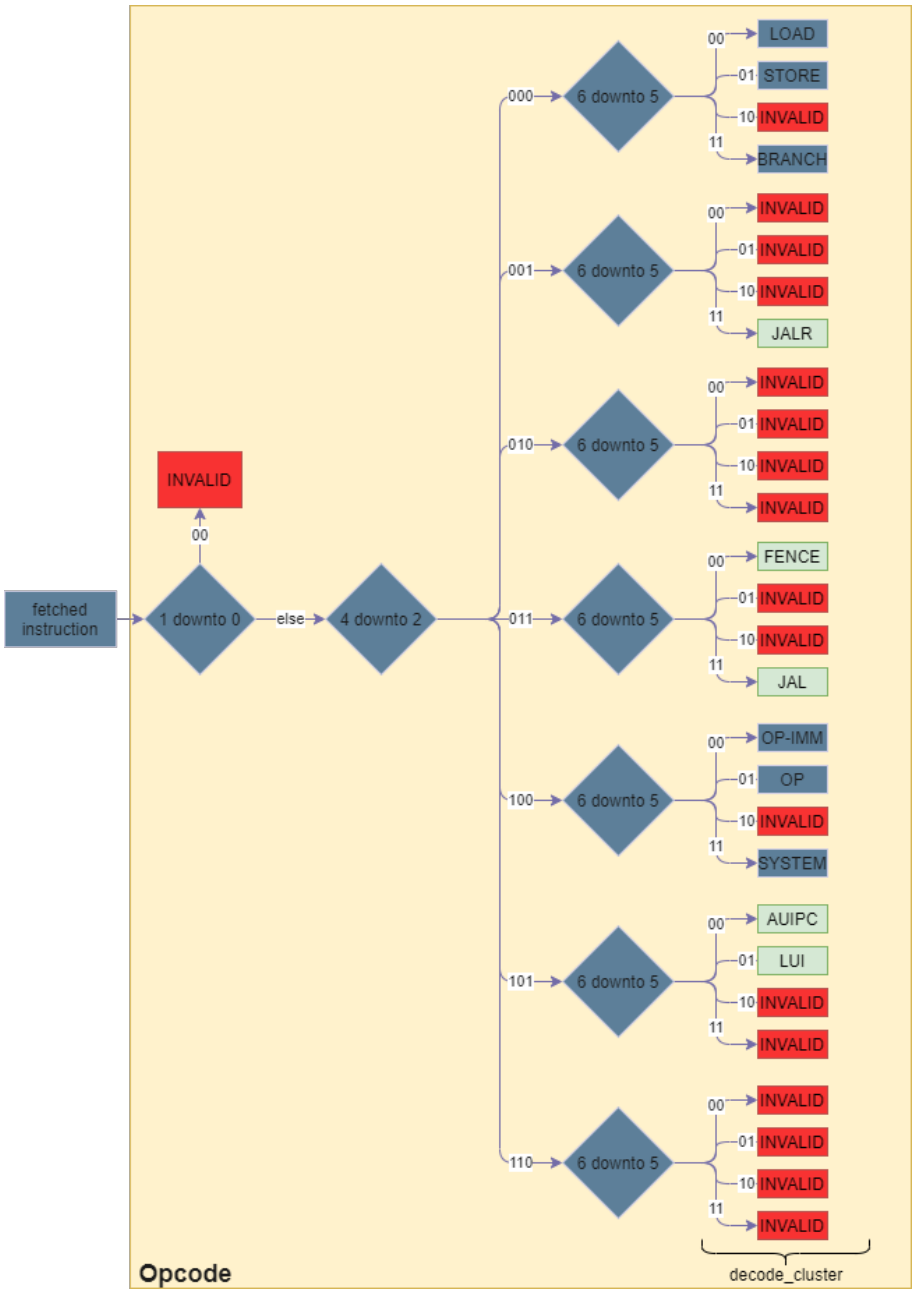

Figure 3.6: Decoding Structure to determine instruction cluster

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.)

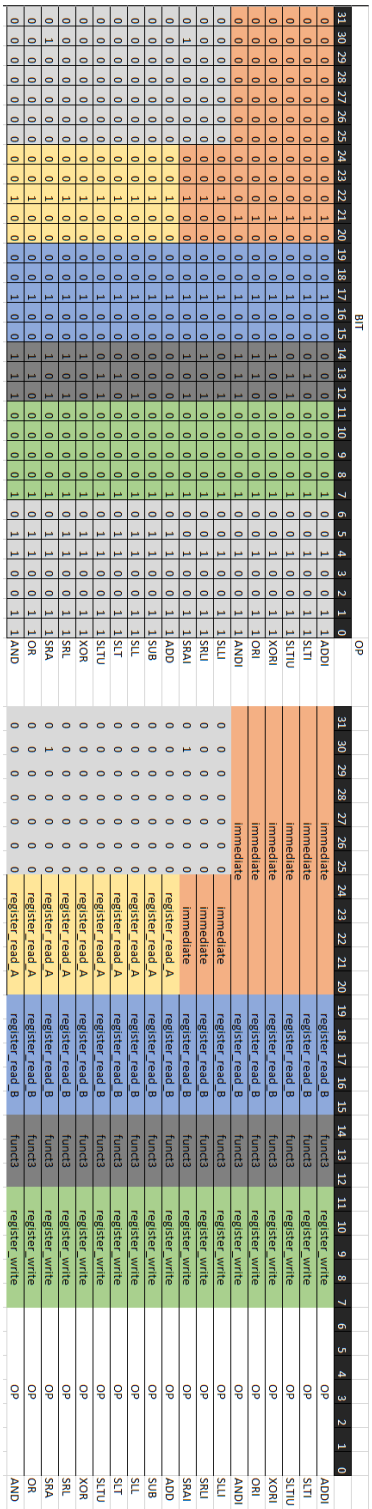

Figure 3.7: Information extraction from 32-bit instruction word

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.

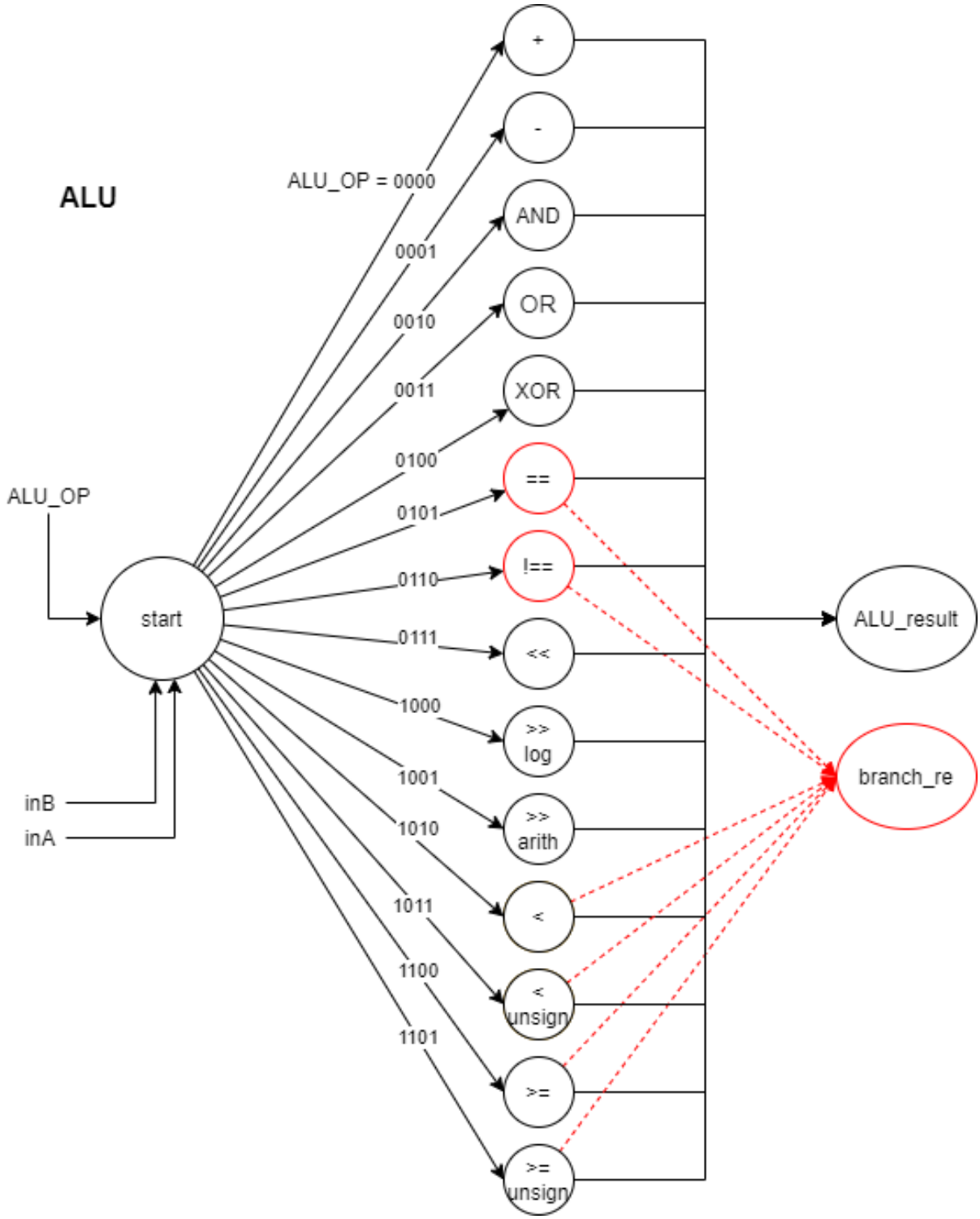

Figure 3.8: ALU operations

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 [And19a]. 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 [And19b].
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.

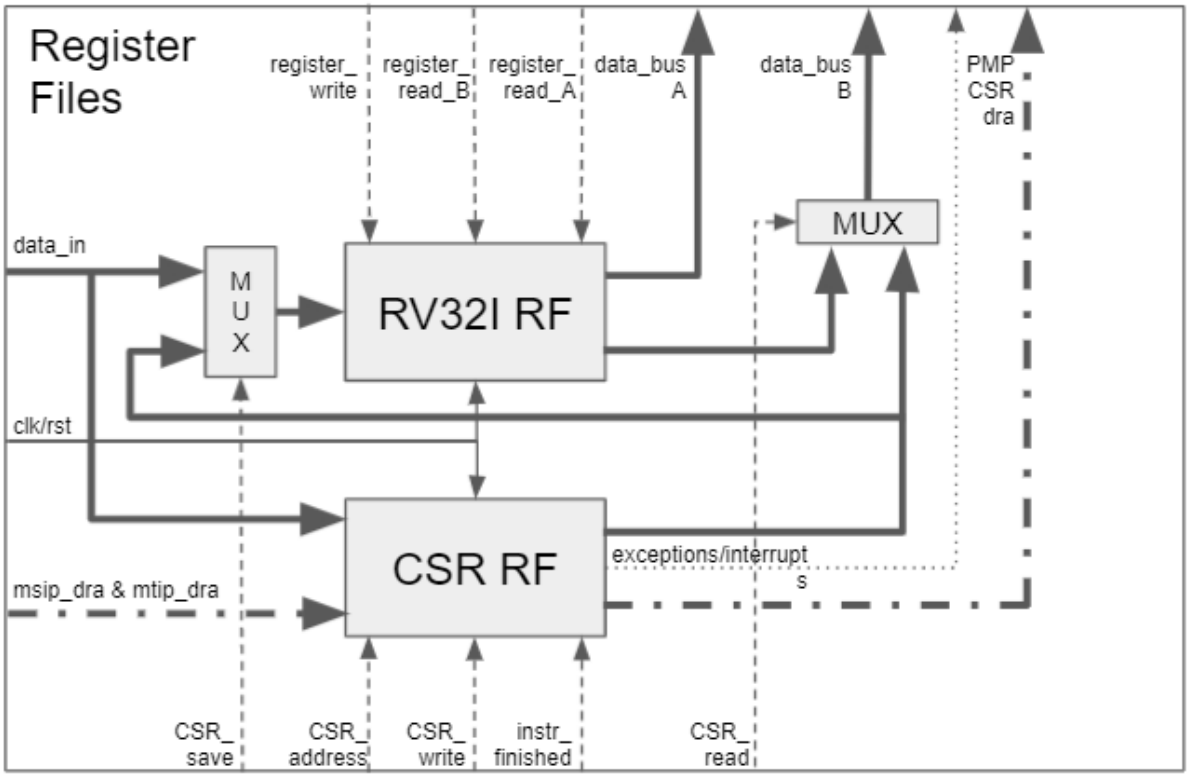

Figure 3.10: Register Files architecture

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 [And19a], 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 RISC-V 32-Bit Integer (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) [And19a].
Figure 3.11 shows the design of RV32I Register File:

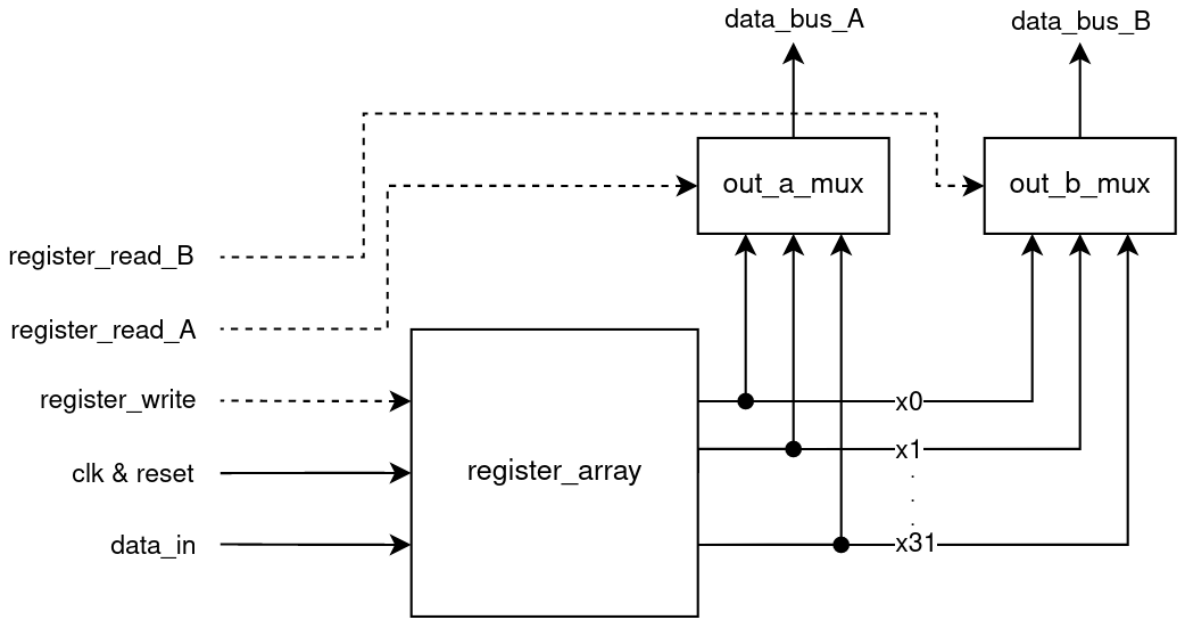

Figure 3.11: General Purpose Registers

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.

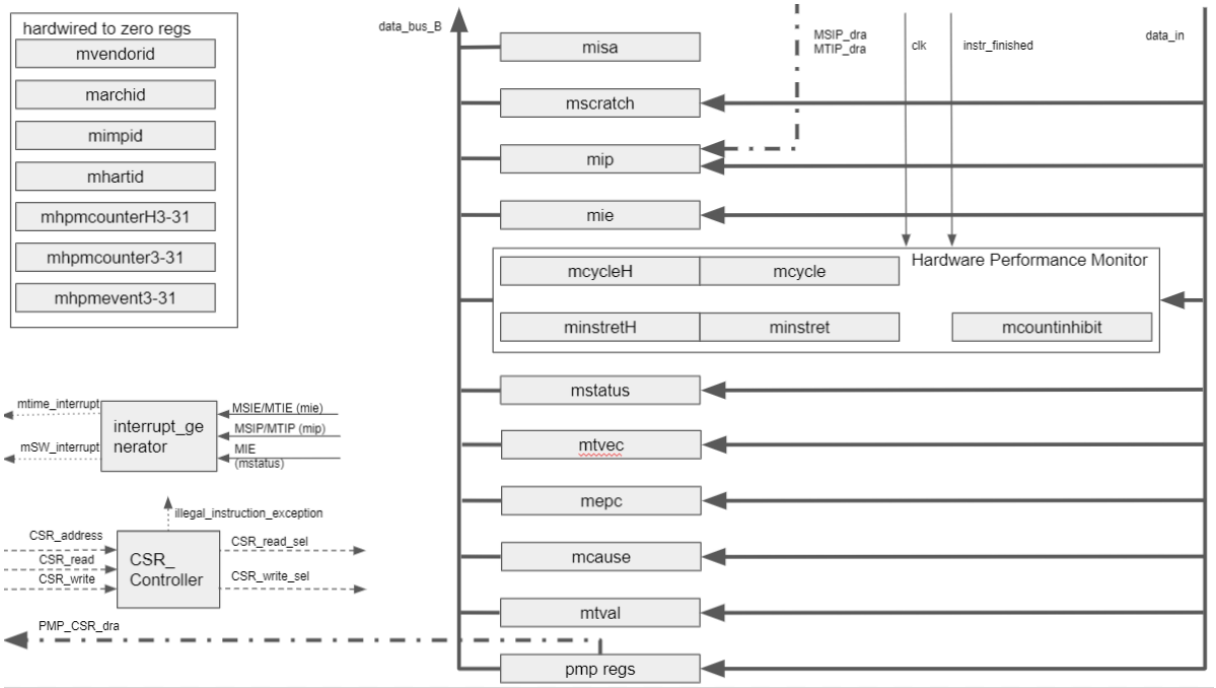

Figure 3.12: CSR Register File Architecture

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: [And19b].
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 [And19b]:

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*) [And19b]

```
1  ------------------------------------------------------------
2  --mstatus register
3  ------------------------------------------------------------
4  mstatus_proc: process(reset, clk)
5  begin
6  if(reset = '1') then
7  mstatus_reg <= (others => '0');
8  elsif(clk'event and clk = '0' and write(0) = '1') then
9  mstatus_reg <= data_in(7) & data_in(3);
10 end if;
11 end process;
12
13 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 [And19b] 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:

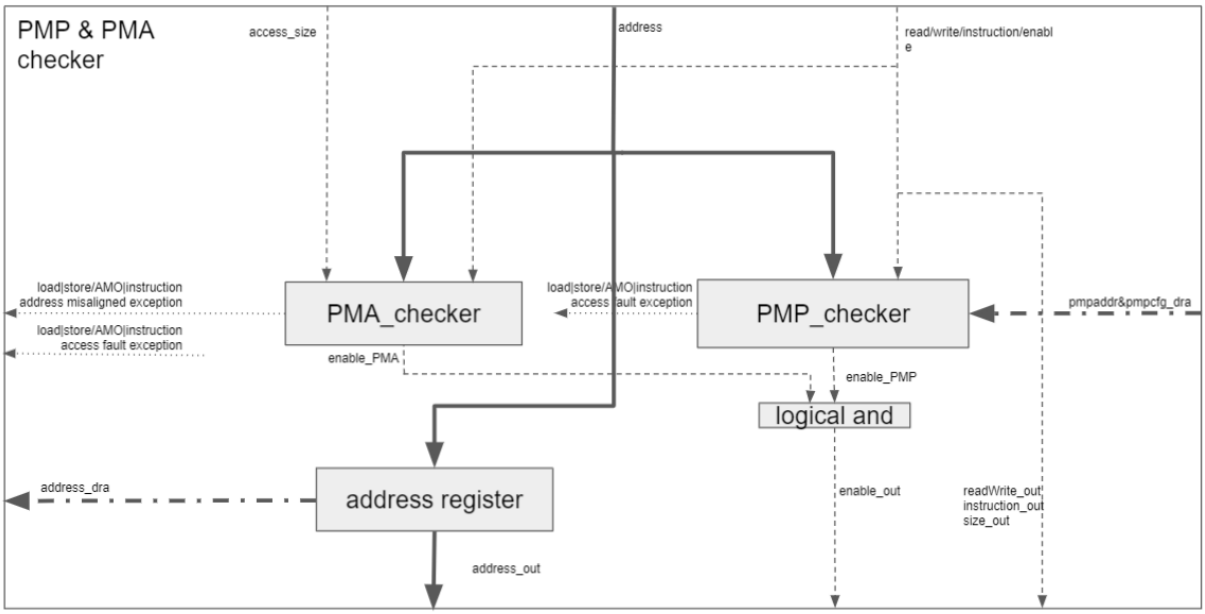

Figure 3.14: PMP & PMA Checker Architecture

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 [And19b]

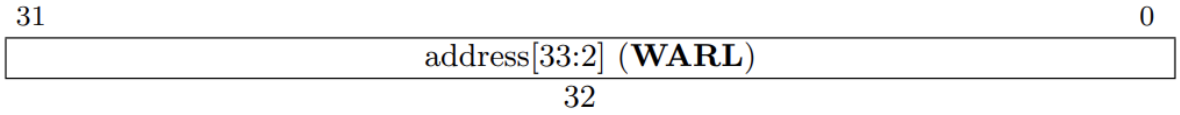

Figure 3.16: *pmpaddr* register [And19b]

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 Naturally aligned power-of-two region (NAPOT).

Right now there is a single PMA check implemented, its job is to check if the memory access is aligned. A word is 32-bit, halfword 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 LSBs are zero), for a halfword access the memory address modulo 2 must be zero (LSB is zero) and byte accesses are allowed on every address.
For both units (PMP and PMA) information about the access like the access size and type must be provided, this is done 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 PMP, 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.
The logical and is used, in order to ensure that both the *PMA_checker* and *PMP_checker* rules are enforced.

3.5.2 Implementation

3.6 Exception Control

The Exception Control Unit is used to guard exception entries and exits one of its tasks is e.g. to modify the PC accordingly and save 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

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/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

3.6.1 Architecture and Design

Figure 3.17 shows the architecture of the Exception Control:

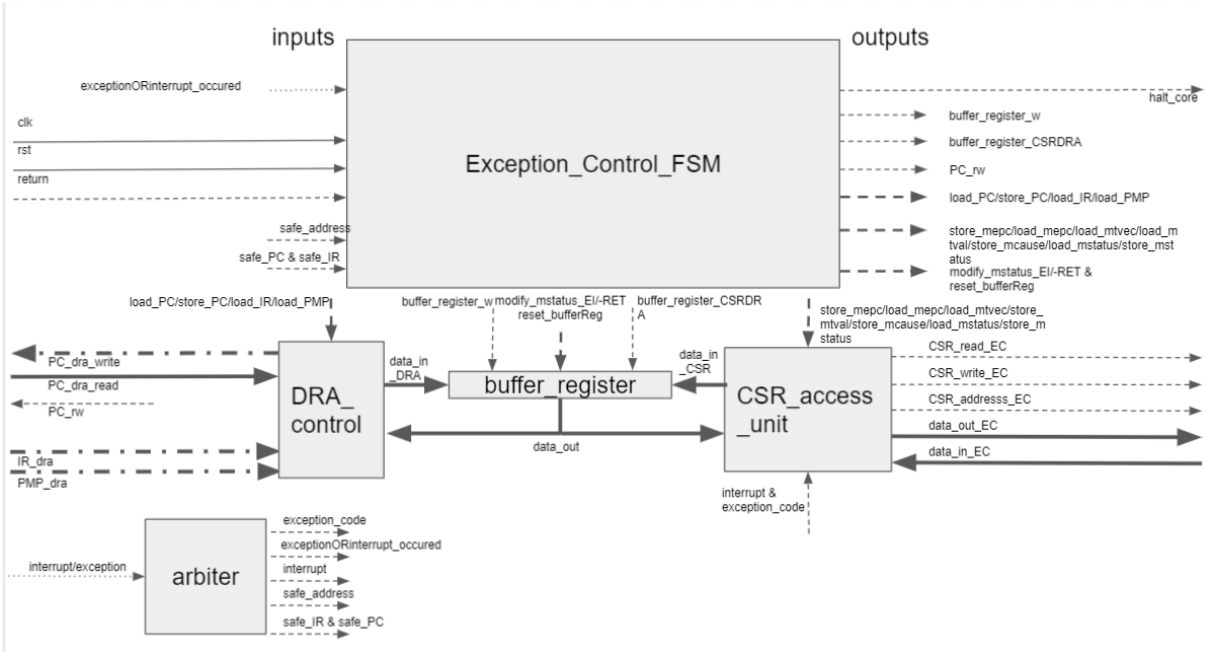

Figure 3.17: Exception Control Architecture

The Exception Control module is comprised of a FSM, 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 arbiter receives the different exception and interrupt signals, it's purpose is to decide which interrupt will be executed and generate the according exception code as well as the signal for the FSM to start the routine.

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 (exit) or the MPIE register to 0 (entry) and switch the two bits (MIE and MPIE) on the *data_out* line. This feature is used during the phase of modifying the *mstatus* register and allows the modify to happen in at least 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 Exception Control unit must write the current *PC+4* to the *mpec* register, modify the *mcause* register to reflect the cause of the exception/interrupt and update the *mtval* CSR to provide additional information on the taken

trap. If multiple exceptions occur at once, only the highest priority exception is taken. If an Exception is raised, while the processor is handling another exception, *mpec*, *mcause* & *mtval* are overwritten. The saving of the return address and other information stored in those registers is left to the trap handler, in order to avoid a large hardware register stack. If an exception entry is 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 (should not happen in normal operation) the entry will be performed.

3.6.2 Implementation

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

The AXI4-Lite Master is implemented in order to allow memory accesses e.g. to a BRAM or UART IP. The following figure depicts the architecture of the master:

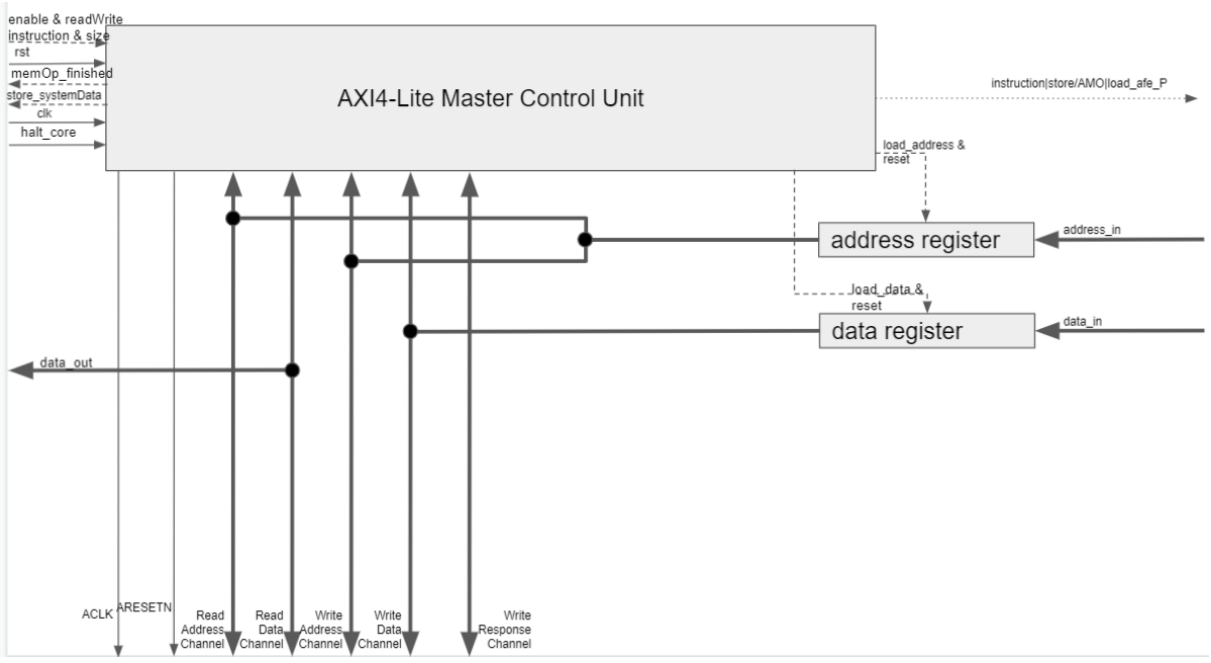

Figure 3.18: AXI4-Lite Master Architecture

It includes two registers, the data register and the address register, these are loaded on the rising edge of the system clock if the corresponding load signal is applied. Both register outputs are constantly applied 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 *clk*. The *rst* signal will reset it. The Master also provides the clock and reset for every slave connected to the AXI interconnect. To start a transfer, the enable signal must be high on a rising edge of the clock, in that case the address and data register are loaded on the next falling edge of system clock.
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 must then process these signals to store the data in the correct register.
At the end of a data transfer, the *memOp_finished* signal is set to high and remains high until a new transfer is initiated.
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

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 addresses can be specified by setting a generic in the VHDL code. A basic address is predefined for each CSR, corresponding to the CLINT module by sifive since this is the closest thing to industry standard.

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 3.6: Memory Mapped CSRs

3.8.1 Architecture and Design

The AXI4-Lite slave is in charge of accepting data transfers and reading/writing the CSRs. Figure 3.19 shows the architecture, including the memory mapped CSRs:

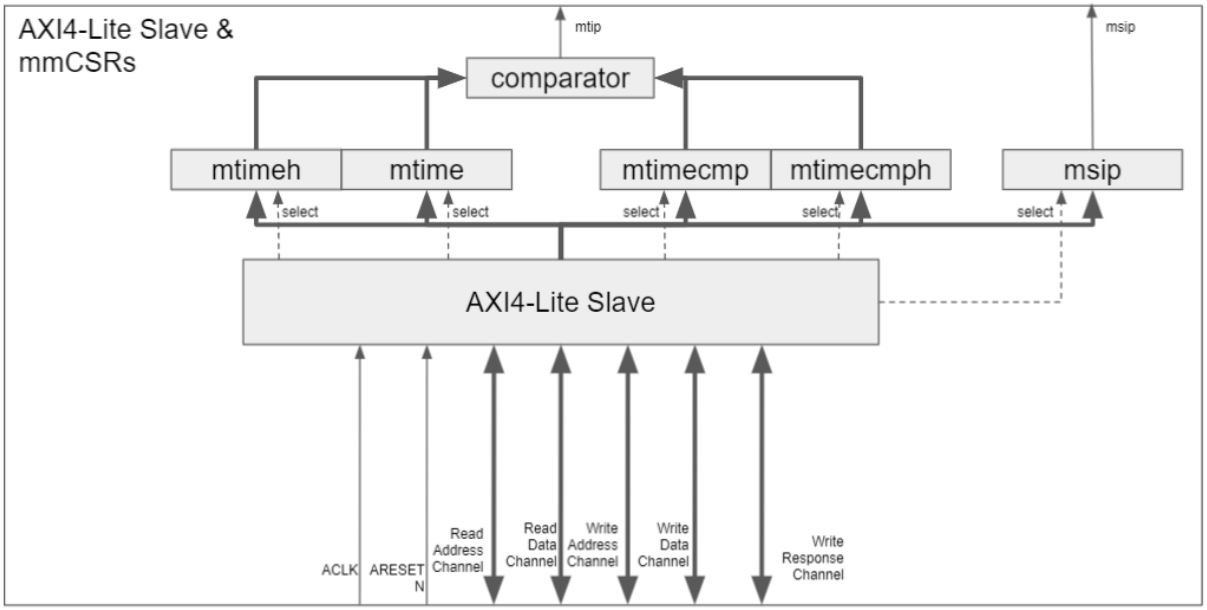

Figure 3.19: AXI4-Lite Slave Architecture

As an AXI4-Lite slave an IP core may be used. In fact the Vivado AXI Interface generator may be used to generate the AXI4-Lite Slave as well as the registers, this IP could then be modified with the comparator and the *msip* and *mtip* output The AXI Slave allows reads and writes to the 32 bit registers *mtimeh*, *mtime*, *mtimecmp* and *mtimecmph* and the one byte *msip* register.
The 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. The *msip* register is a one byte register, the remaining 31 bits are hardwired to zero. Its reset value is zero. If it is set to one, the *msip* bit in *mip* is set and therefore, a software interrupts may be risen.
Table 3.8.1 shows the reset values for each CSR:

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

Table 3.7: Memory Mapped CSRs reset values

3.8.2 Implementation

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.
To get a better understanding of how the decoding process is executed, the source code of the decoder can be found in the appendix (**A. CU decoder**).
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. The corresponding testbench for the decoder can be found in the appendix (**B. CU decoder testbench**). 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:

```
1 stim: process
2 begin
3   -- parse through different instruction strings
4   --OPIMM
```

```
5   ir <= "00000000001000100000000010010011"; --ADDI
6   wait for 100ns;
7   ir <= "00000000001000100010000010010011"; --SLTI
8   wait for 100ns;
9   ir <= "00000000001000100011000010010011"; --SLTIU
10  wait for 100ns;
11  ir <= "00000000001000100100000010010011"; --XORI
12  ...
```

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.

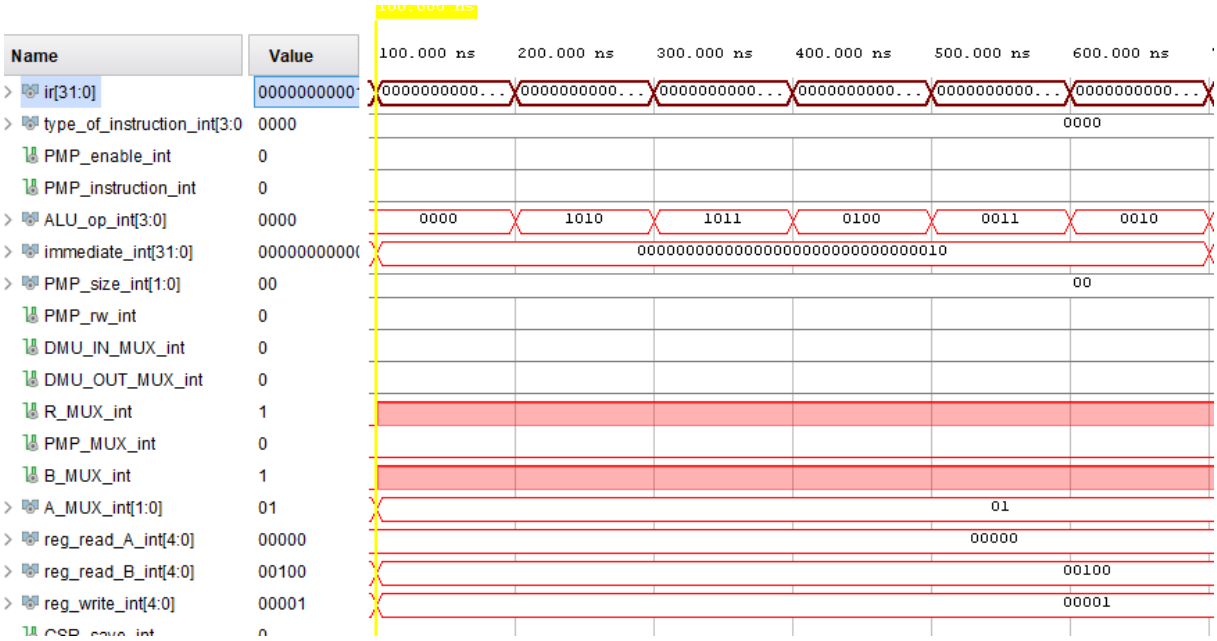

Figure 4.1: Vivado timing diagram for OPIMM instructions

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.

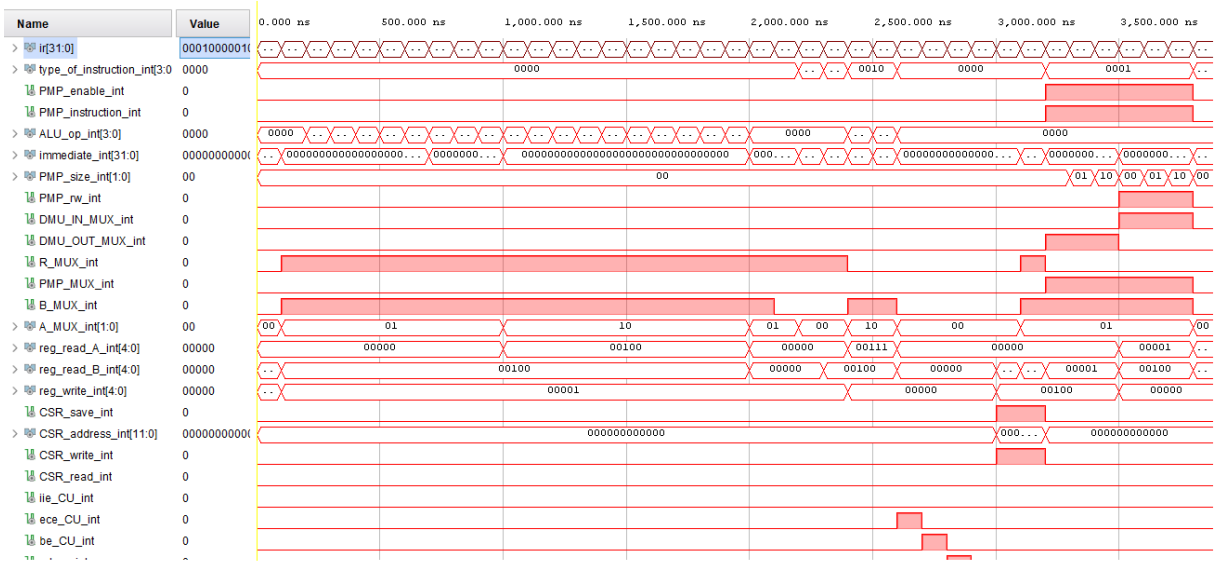

Figure 4.2: Vivado timing diagram for full CU decoder testbench

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
```

```
17    wait for 100ns;
18    alu_op <= "0101"; --EQUAL
19    wait for 100ns;
20    alu_op <= "0110"; --NEQUAL
21    wait for 100ns;
22    alu_op <= "0111"; --shift_left
23    wait for 100ns;
24    alu_op <= "1000"; --shift_right
25    wait for 100ns;
26    alu_op <= "1001"; --shift_right (arithmetic)
27    ...
```

Listing 4.2: CU 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. Apparently, the two inputs are not the same which is the reason for why 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.

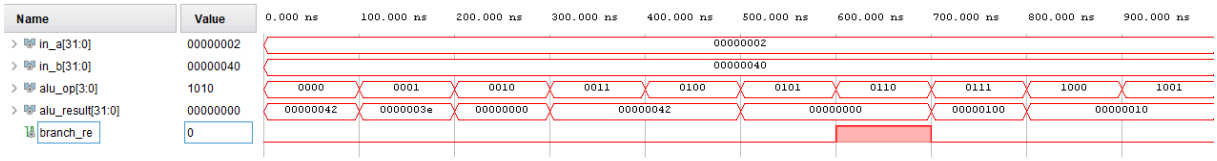

Figure 4.3: Vivado timing diagram for ALU testbench

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.

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: [Tim19]. 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 [RIS]. 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 Programm Counter (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 4.4 shows the memory map of the test bench.

Figure 4.4: 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.

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

Listing 4.3: 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:

```
1  SW x2, 0(x1)
2  SW x3, 4(x1)
3  SW x4, 8(x1)
4  ADDI x5, x0, 0xAB
5  SB x5, 13(x1)
6  SH x2, 14(x1)
7
8  LB x11, 3(x1) #expect 0xFFFFFFC0
9  LH x10, 2(x1) #expect 0xFFFFC0BA
10 LW x9, 4(x1) #expect 0x12345000
11 LBU x8, 3(x1) #expect 0x000000C0
12 LHU x7, 2(x1) #expect 0x0000C0BA
```

Listing 4.4: 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 4.5 shows the memory in the simulation wave form view:

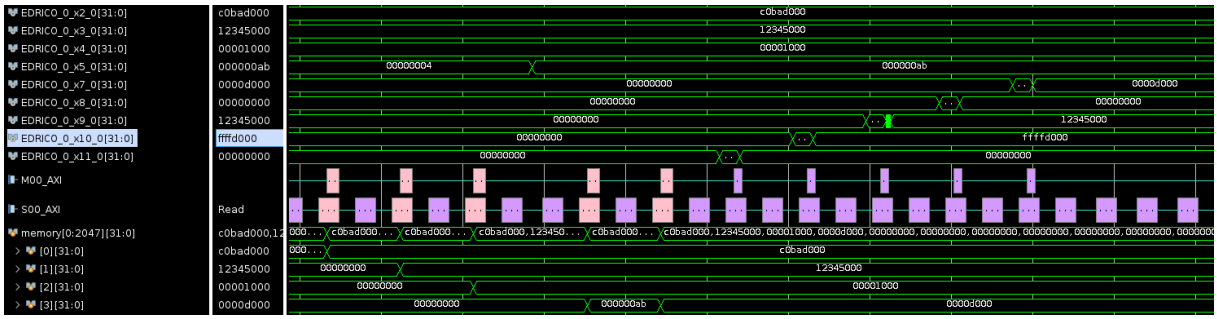

Figure 4.5: wave from view of the memory, AXI4-Lite transfer, x2-x5 and x7-x11 during System Verification

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 [HP17].

5 Future Work

6 Conclusion

Bibliography

[And19a] SiFive Inc. Andrew Waterman Krste Asanovic. *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*. Accessed: 17.07.2021. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[And19b] SiFive Inc. Andrew Waterman Krste Asanovic. *The RISC-V Instruction Set Manual. Volume II: Privileged Achitecture*. Accessed: 17.07.2021. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf.

[ARM13] ARM. *AMBA AXI and ACE Protocoll Specification. AXI3, AXi4, and AXI4-Lite, ACE and ACE-Lite*. Accessed: 18.07.2021. 2013. URL: https://www.arm.com.

[Ges14] Ralf Gessler. *Entwicklung Eingebetteter Systeme*. Wiesbaden: Springer, 2014.

[Hel16] Roland Hellmann. *Rechnerarchitektur*. Aalen: De Gruyter Oldenbourg, 2016.

[HP17] JL Hennesy and DA Patterson. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. o. O.: Elsevier Science Technology Books, 2017.

[HP20] JL Hennesy and DA Patterson. *Computer Organization and Design MIPS Edition: The Hardware Software Interface*. 6th ed. o. O.: Elsevier Science Technology Books, 2020.

[KLK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking*. Würzburg, Germany: Springer, 2020.

[ORe21] Gerard O'Regan. *A Brief History of Computing*. 3rd ed. Cham: Springer International Publishing, 2021.

[RIS] RISC-V. *RISC-V Exchange: Available Software*. URL: https://riscv.org/exchange/software/.

[SPE06] SPEC. *CINT2006 (Integer Component of SPEC CPU2006)*. 2006. URL: https://www.spec.org/cpu2006/CINT2006/.

[Sys19] Dive into Systems. Accessed: 18.07.2021. 2019. URL: https://diveintosystems.org/antora/diveintosystems/1.0/MemHierarchy/mem_hierarchy.html.

[Tim19] Megan Wachs Tim Newsome. *RISC-V External Debug Support. Version 0.13.2.* Accessed: 17.07.2021. 2019. URL: https://github.com/riscv/riscv-debug-spec/blob/release/riscv-debug-release.pdf.

[Xil17] Xilinx. *Understanding FPGA Architecture.* 2017. URL: https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/devices/con-fpga-architecture.html.

Appendix

A. CU decoder code

B. CU decoder testbench code

A. CU decoder


```vhdl
----------------------------------------------------------------------------------
-- Company: DHBW
-- Engineer: Noah Woelki
--
-- Create Date: 05/10/2021 06:04:15 AM
-- Design Name: CU_decoder
-- Module Name: Control Unit
-- Project Name: EDRICO
-- Target Devices: Arty Z7
-- Tool Versions:
-- Description:
--  Decoding unit decodes incoming instruction once its loaded from memory
--
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
----------------------------------------------------------------------------------
--ENTITY
----------------------------------------------------------------------------------
entity CU_decoder is
port (
    ----------------------------------------------------------------------------------
    --input signals
    ----------------------------------------------------------------------------------
    -- instruction register
    ir: in std_logic_vector(31 downto 0);
    ----------------------------------------------------------------------------------
    --output signals
    ----------------------------------------------------------------------------------
    type_of_instruction_int: out std_logic_vector(3 downto 0);
    -- PMP ctrl
    PMP_enable_int: out std_logic;
    PMP_instruction_int: out std_logic;
    PMP_size_int: out std_logic_vector(1 downto 0);
    PMP_rw_int: out std_logic;
    -- MUX ctrl
    DMU_IN_MUX_int: out std_logic;
    DMU_OUT_MUX_int: out std_logic;
    R_MUX_int: out std_logic;
    PMP_MUX_int: out std_logic;
    B_MUX_int: out std_logic;
    A_MUX_int: out std_logic_vector(1 downto 0);
    -- reg ctrl
    reg_read_A_int: out std_logic_vector(4 downto 0);
    reg_read_B_int: out std_logic_vector(4 downto 0);
    reg_write_int: out std_logic_vector(4 downto 0);
    -- CSR ctrl
    CSR_save_int: out std_logic;
    CSR_address_int: out std_logic_vector(11 downto 0);
    CSR_write_int: out std_logic;
    CSR_read_int: out std_logic;
```

A. CU decoder


```
61      -- exception ctrl
62      iie_CU_int: out std_logic;
63      ece_CU_int: out std_logic;
64      be_CU_int: out std_logic;
65      return_int: out std_logic;
66      -- other signals
67      ALU_op_int: out std_logic_vector(3 downto 0);
68      immediate_int: out std_logic_vector(31 downto 0);
69      mask_ctrl_int: out std_logic_vector(2 downto 0)
70  );
71  end entity;
72
73
74  ----------------------------------------------------------------------------------
75  --ARCHITECTURE
76  ----------------------------------------------------------------------------------
77  architecture rtl of CU_decoder is
78  ----------------------------------------------------------------------------------
79  --signals
80  ----------------------------------------------------------------------------------
81  type instruction_cluster is (INVALID, LOAD, STORE, BRANCH, JALR, JAL, FENCE, OPIMM,
    OP, SYSTEM, AUIPC, LUI);
82  signal decoded_cluster : instruction_cluster;
83  ----------------------------------------------------------------------------------
84  --constants
85  ----------------------------------------------------------------------------------
86
87  begin
88      decode: process(ir)
89      begin
90          case ir(1 downto 0) is
91              when "00" =>
92                  decoded_cluster <= INVALID;
93              when others =>
94                  case ir(4 downto 2) is
95                      when "000" =>
96                          case ir(6 downto 5) is
97                              when "00" => decoded_cluster <= LOAD;
98                              when "01" => decoded_cluster <= STORE;
99                              when "10" => decoded_cluster <= INVALID;
100                             when "11" => decoded_cluster <= BRANCH;
101                             when others => decoded_cluster <= INVALID;
102                         end case;
103                     when "001" =>
104                         case ir(6 downto 5) is
105                             when "00" => decoded_cluster <= INVALID;
106                             when "01" => decoded_cluster <= INVALID;
107                             when "10" => decoded_cluster <= INVALID;
108                             when "11" => decoded_cluster <= JALR;
109                             when others => decoded_cluster <= INVALID;
110                         end case;
111                     when "010" =>
112                         decoded_cluster <= INVALID;
113                     when "011" =>
114                         case ir(6 downto 5) is
115                             when "00" => decoded_cluster <= FENCE;
116                             when "01" => decoded_cluster <= INVALID;
117                             when "10" => decoded_cluster <= INVALID;
118                             when "11" => decoded_cluster <= JAL;
119                             when others => decoded_cluster <= INVALID;
```

A. CU decoder


```
120                         end case;
121                     when "100" =>
122                         case ir(6 downto 5) is
123                             when "00" => decoded_cluster <= OPIMM;
124                             when "01" => decoded_cluster <= OP;
125                             when "10" => decoded_cluster <= INVALID;
126                             when "11" => decoded_cluster <= SYSTEM;
127                             when others => decoded_cluster <= INVALID;
128                         end case;
129                     when "101" =>
130                         case ir(6 downto 5) is
131                             when "00" => decoded_cluster <= AUIPC;
132                             when "01" => decoded_cluster <= LUI;
133                             when "10" => decoded_cluster <= INVALID;
134                             when "11" => decoded_cluster <= INVALID;
135                             when others => decoded_cluster <= INVALID;
136                         end case;
137                     when others =>
138                         decoded_cluster <= INVALID;
139                 end case;
140         end case;
141     end process decode;
142 
143     assign: process(ir, decoded_cluster)
144     begin
145         -- default values
146         type_of_instruction_int <= "0000";
147         PMP_enable_int <= '0';
148         PMP_instruction_int <= '0';
149         PMP_size_int <= "00";
150         PMP_rw_int <= '0';
151         DMU_IN_MUX_int <= '0';
152         DMU_OUT_MUX_int <= '0';
153         R_MUX_int <= '0';
154         PMP_MUX_int <= '0';
155         B_MUX_int <= '0';
156         A_MUX_int <= "00";
157         reg_read_A_int <= "00000";
158         reg_read_B_int <= "00000";
159         reg_write_int <= "00000";
160         CSR_save_int <= '0';
161         CSR_address_int <= "000000000000";
162         CSR_write_int <= '0';
163         CSR_read_int <= '0';
164         iie_CU_int <= '0';
165         ece_CU_int <= '0';
166         be_CU_int <= '0';
167         return_int <= '0';
168         ALU_op_int <= "0000";
169         immediate_int <= "00000000000000000000000000000000";
170         mask_ctrl_int <= "100";
171 
172         case decoded_cluster is
173             when LOAD =>
174             -- for LOAD instructions adapt the PMP size and masking control
175             -- ALU performs addition to obtain target address and therefore receives
    an immediate
176             -- and data from register. loaded data stores in respective destination
    register
177                 type_of_instruction_int <= "0001";
```

A. CU decoder


```
178                 PMP_enable_int <= '1';
179                 PMP_instruction_int <= '1';
180                 DMU_OUT_MUX_int <= '1';
181                 PMP_MUX_int <= '1';
182                 B_MUX_int <= '1';
183                 A_MUX_int <= "01";
184                 reg_read_B_int <= ir(19 downto 15);
185                 reg_write_int <= ir(11 downto 7);
186                 immediate_int <= std_logic_vector((31 downto 12 => ir(31)) & ir(31
    downto 20));
187 
188                 case ir(14 downto 12) is
189                     when "000" => --LB
190                         PMP_size_int <= "00";
191                         mask_ctrl_int <= "000";
192                     when "001" => --LH
193                         PMP_size_int <= "01";
194                         mask_ctrl_int <= "010";
195                     when "010" => --LW
196                         PMP_size_int <= "10";
197                         mask_ctrl_int <= "100";
198                     when "100" => --LBU
199                         PMP_size_int <= "00";
200                         mask_ctrl_int <= "001";
201                     when "101" => --LHU
202                         PMP_size_int <= "01";
203                         mask_ctrl_int <= "011";
204                     when others =>
205                         iie_CU_int <= '1';
206                 end case;
207             when STORE =>
208             -- for STORE instructions adapt the PMP size and masking control
209             -- ALU performs addition to obtain target address and therefore receives
    an immediate
210             -- and data from register
211                 type_of_instruction_int <= "0001";
212                 PMP_enable_int <= '1';
213                 PMP_instruction_int <= '1';
214                 PMP_rw_int <= '1';
215                 DMU_IN_MUX_int <= '1';
216                 DMU_OUT_MUX_int <= '0';
217                 R_MUX_int <= '0';
218                 PMP_MUX_int <= '1';
219                 B_MUX_int <= '1';
220                 A_MUX_int <= "01";
221                 reg_read_A_int <= ir(24 downto 20);
222                 reg_read_B_int <= ir(19 downto 15);
223                 immediate_int <= std_logic_vector((31 downto 12 => ir(31)) & ir(31
    downto 25) & ir(11 downto 7));
224 
225                     case ir(14 downto 12) is
226                         when "000" => --SB
227                             PMP_size_int <= "00";
228                             mask_ctrl_int <= "001";
229                         when "001" => --SH
230                             PMP_size_int <= "01";
231                             mask_ctrl_int <= "011";
232                         when "010" => --SW
```

A. CU decoder


```
233                             PMP_size_int <= "10";
234                             mask_ctrl_int <= "100";
235                         when others =>
236                             iie_CU_int <= '1';
237                     end case;
238             when BRANCH =>
239             -- Branch instructions adapt the ALU operation to compare regA and regB
240                 type_of_instruction_int <= "0010";
241                 B_MUX_int <= '1';
242                 A_MUX_int <= "10";
243                 reg_read_A_int <= ir(24 downto 20);
244                 reg_read_B_int <= ir(19 downto 15);
245                 ALU_op_int <= "0000";
246                 immediate_int <= std_logic_vector((31 downto 12 => ir(31)) & ir(31) &
ir(7) & ir(30 downto 25) & ir(11 downto 8));
247 
248                     case ir(14 downto 12) is
249                         when "000" => --BEQ
250                             ALU_op_int <= "0101";

251                         when "001" => --BNE
252                             ALU_op_int <= "0110";
253                         when "100" => --BLT
254                             ALU_op_int <= "1010";
255                         when "101" => --BGE
256                             ALU_op_int <= "1100";
257                         when "110" => --BLTU
258                             ALU_op_int <= "1011";
259                         when "111" => --BLEU
260                             ALU_op_int <= "1101";
261                         when others =>
262                             iie_CU_int <= '1';
263                     end case;
264             when JALR => --JALR
265             -- JALR adds + 4 to the program counter and stores it to rd
266             -- target address is obtained by PC_top control which adds regB to the
immediate
267                 R_MUX_int <= '1';
268                 type_of_instruction_int <= "1000";
269                 reg_read_B_int <= ir(19 downto 15);
270                 reg_write_int <= ir(11 downto 7);
271                 immediate_int <= std_logic_vector((31 downto 12 => ir(31)) & ir(31
downto 20));
272             when FENCE => --NOP
273             when JAL => --JAL
274             -- JAL stores PC +4 into rd, jump target address obtained by PC_top (adds
immediate to PC)
275                 R_MUX_int <= '1';
276                 type_of_instruction_int <= "0100";
277                 reg_write_int <= ir(11 downto 7);
278                 immediate_int <= std_logic_vector((31 downto 20 => ir(31)) & ir(31) &
ir(19 downto 12) & ir(20) & ir(30 downto 21));
279             when OPIMM =>
280             -- Immediate operations perform ALU operations on data from regB and
immediate
281                 R_MUX_int <= '1';
282                 PMP_MUX_int <= '0';
283                 B_MUX_int <= '1';
284                 A_MUX_int <= "01";
285                 reg_read_B_int <= ir(19 downto 15);
```

A. CU decoder


```
286                 reg_write_int <= ir(11 downto 7);
287 
288                     case ir(14 downto 12) is
289                         when "000" => --ADDI
290                             ALU_op_int <= "0000";
291                             immediate_int <= std_logic_vector((31 downto 12 =>
    ir(31)) & ir(31 downto 20));
292                         when "001" => --SLLI
293                             ALU_op_int <= "0111";
294                             immediate_int <= std_logic_vector((31 downto 5 => ir(24))
    & ir(24 downto 20));
295                         when "010" => --SLTI
296                             ALU_op_int <= "1010";
297                             immediate_int <= std_logic_vector((31 downto 12 =>
    ir(31)) & ir(31 downto 20));
298                         when "011" => --SLTIU
299                             ALU_op_int <= "1011";
300                             immediate_int <= std_logic_vector((31 downto 12 =>
    ir(31)) & ir(31 downto 20));
301                         when "100" => --XORI
302                             ALU_op_int <= "0100";
303                             immediate_int <= std_logic_vector((31 downto 12 =>
    ir(31)) & ir(31 downto 20));
304                         when "101" => --SRLI/SRAI
305                             if(ir(30) = '0') then --SRLI
306                                 ALU_op_int <= "1000";
307                                 immediate_int <= std_logic_vector((31 downto 5 =>
    ir(24)) & ir(24 downto 20));
308                             else --SRAI
309                                 ALU_op_int <= "1001";
310                                 immediate_int <= std_logic_vector((31 downto 5 =>
    ir(24)) & ir(24 downto 20));
311                             end if;
312 
313                         when "110" => --ORI
314                             ALU_op_int <= "0011";
315                             immediate_int <= std_logic_vector((31 downto 12 =>
    ir(31)) & ir(31 downto 20));
316                         when "111" => --ANDI
317                             ALU_op_int <= "0010";
318                             immediate_int <= std_logic_vector((31 downto 12 =>
    ir(31)) & ir(31 downto 20));
319                         when others =>
320                             iie_CU_int <= '1';
321                     end case;
322             when OP =>
323             -- Normal operation performs ALU operation on regA and regB, stores in rd
    (regwrite)
324                 R_MUX_int <= '1';
325                 PMP_MUX_int <= '0';
326                 B_MUX_int <= '1';
327                 A_MUX_int <= "10";
328                 reg_read_A_int <= ir(24 downto 20);
329                 reg_read_B_int <= ir(19 downto 15);
330                 reg_write_int <= ir(11 downto 7);
331 
332                     case ir(14 downto 12) is
333                         when "000" => --ADD/SUB
334                             if(ir(30) = '0') then --ADD
                                ALU_op_int <= "0000";
```

A. CU decoder


```
335                             else --SUB
336                                 ALU_op_int <= "0001";
337                             end if;
338                         when "001" => --SLL
339                             ALU_op_int <= "0111";
340                         when "010" => --SLT
341                             ALU_op_int <= "1010";
342                         when "011" => --SLTU
343                             ALU_op_int <= "1011";
344                         when "100" => --XOR
345                             ALU_op_int <= "0100";
346                         when "101" => --SRL/SRA
347                             if(ir(30) = '0') then --SRL
348                                 ALU_op_int <= "1000";
349                             else --SRA
350                                 ALU_op_int <= "1001";
351                             end if;

352                         when "110" => --OR
353                             ALU_op_int <= "0011";
354                         when "111" => --AND
355                             ALU_op_int <= "0010";
356                         when others =>
357                             iie_CU_int <= '1';
358                     end case;
359                 when SYSTEM =>
360                     CSR_save_int <= '1';
361                     CSR_address_int <= ir(31 downto 20);
362                     CSR_write_int <= '1';
363 
364                         case ir(14 downto 12) is
365 
366                             when "000" => --ECALL/EBREAK
367                                 type_of_instruction_int <= "0000";
368                                 PMP_enable_int <= '0';
369                                 PMP_instruction_int <= '0';
370                                 PMP_size_int <= "00";
371                                 PMP_rw_int <= '0';
372                                 DMU_IN_MUX_int <= '0';
373                                 DMU_OUT_MUX_int <= '0';
374                                 R_MUX_int <= '0';
375                                 PMP_MUX_int <= '0';
376                                 B_MUX_int <= '0';
377                                 A_MUX_int <= "00";
378                                 reg_read_A_int <= "00000";
379                                 reg_read_B_int <= "00000";
380                                 reg_write_int <= "00000";
381                                 CSR_save_int <= '0';
382                                 CSR_address_int <= "000000000000";
383                                 CSR_write_int <= '0';
384                                 CSR_read_int <= '0';
385                                 iie_CU_int <= '0';
386                                 return_int <= '0';
387                                 ALU_op_int <= "0000";
388                                 mask_ctrl_int <= "100";
389 
390                                 case ir(22 downto 20) is
391                                     when "000" => --ECALL
392                                         ece_CU_int <= '1';
393                                         be_CU_int <= '0';
```

A. CU decoder


```
394                             when "001" => --EBREAK
395                                 ece_CU_int <= '0';
396                                 be_CU_int <= '1';
397                             when "010" => --MRET
398                                 if(ir(29 downto 28) = "11") then
399                                     ece_CU_int <= '0';
400                                     be_CU_int <= '0';

401                                     return_int <= '1';
402                                 else --INVALID
403                                     ece_CU_int <= '0';
404                                     be_CU_int <= '0';
405                                     iie_CU_int <= '1';
406                                 end if;
407                             when "101" => --WFI -> NOP
408                                 ece_CU_int <= '0';
409                                 be_CU_int <= '0';
410                             when others =>
411                                 ece_CU_int <= '0';
412                                 be_CU_int <= '0';
413                                 iie_CU_int <= '1';
414                         end case;
415                     when "001" => --CSRRW
416                         R_MUX_int <= '0';
417                         B_MUX_int <= '0';
418                         A_MUX_int <= "00";
419                         reg_read_B_int <= ir(19 downto 15);
420                         reg_write_int <= ir(11 downto 7);

421                         CSR_read_int <= '0';
422                         ALU_op_int <= "0000";

423                     when "010" => --CSRRS
424                         R_MUX_int <= '1';
425                         B_MUX_int <= '1';
426                         A_MUX_int <= "10";
427                         reg_read_B_int <= ir(19 downto 15);
428                         reg_write_int <= ir(11 downto 7);

429                         CSR_read_int <= '1';
430                         ALU_op_int <= "0011";
431                     when "011" => --CSRRC
432                         R_MUX_int <= '1';
433                         B_MUX_int <= '1';
434                         A_MUX_int <= "10";
435                         reg_read_B_int <= ir(19 downto 15);
436                         reg_write_int <= ir(11 downto 7);

437                         CSR_read_int <= '1';
438                         ALU_op_int <= "0010";
439                     when "101" => --CSRRWI
440                         R_MUX_int <= '1';
441                         B_MUX_int <= '1';
442                         A_MUX_int <= "01";
443                         reg_read_B_int <= ir(19 downto 15);
444                         reg_write_int <= ir(11 downto 7);

445                         CSR_read_int <= '0';
446                         ALU_op_int <= "0000";
```

A. CU decoder


```
447                             immediate_int <= std_logic_vector((31 downto 5 => '0') &
    ir(19 downto 15));
448                         when "110" => --CSRRSI
449                             R_MUX_int <= '1';
450                             B_MUX_int <= '1';
451                             A_MUX_int <= "01";
452                             reg_read_B_int <= ir(19 downto 15);
453                             reg_write_int <= ir(11 downto 7);
454                             CSR_read_int <= '1';
455                             ALU_op_int <= "0011";
456                             immediate_int <= std_logic_vector((31 downto 5 => '0') &
    ir(19 downto 15));
457                         when "111" => --CSRRCI
458                             R_MUX_int <= '1';
459                             B_MUX_int <= '1';
460                             A_MUX_int <= "01";
461                             reg_read_B_int <= ir(19 downto 15);
462                             reg_write_int <= ir(11 downto 7);
463                             CSR_read_int <= '1';
464                             ALU_op_int <= "0010";
465                             immediate_int <= std_logic_vector((31 downto 5 => '0') &
    ir(19 downto 15));
466                         when others =>
467                             iie_CU_int <= '1';
468                     end case;
469             when AUIPC =>
470             -- AUIPC adds upper immediate to the pgroam counter and stores to rd
    (regwrite)
471                 R_MUX_int <= '1';
472                 PMP_MUX_int <= '0';
473                 B_MUX_int <= '0';
474                 A_MUX_int <= "01";
475                 reg_write_int <= ir(11 downto 7);
476                 immediate_int <= std_logic_vector(ir(31 downto 12) & (11 downto 0 =>
    '0'));
477
478             when LUI =>
479             -- loads upper immediate
480                 R_MUX_int <= '1';
481                 PMP_MUX_int <= '0';
482                 B_MUX_int <= '1';
483                 A_MUX_int <= "01";
484                 reg_write_int <= ir(11 downto 7);
485                 immediate_int <= std_logic_vector(ir(31 downto 12) & (11 downto 0 =>
    '0'));
486
487             when INVALID => --illegal instruction exception
488                 iie_CU_int <= '1';
489             when others => iie_CU_int <= '1';
490         end case;
491     end process assign;
492 end architecture;
```

B. CU decoder testbench


```
1  ----------------------------------------------------------------------------------
2  -- Company: DHBW
3  -- Engineer: Noah Woelki
4  --
5  -- Create Date: 14.05.2021 14:36:33
6  -- Design Name: CU_decoder_tb
7  -- Module Name: CU_decoder_tb
8  -- Project Name: EDRICO
9  -- Target Devices: Arty Z7
10 -- Tool Versions:
11 -- Description:
12 --  testbench for decoding process, insert different instruction strings to
13 --  test the decoding procedure
14 --
15 -- Dependencies:
16 --
17 -- Revision:
18 -- Revision 0.01 - File Created
19 -- Additional Comments:
20 --
21 ----------------------------------------------------------------------------------
22
23 library ieee;
24 use ieee.std_logic_1164.all;
25 use ieee.numeric_std.all;
26 library CU_lib;
27 use CU_lib.CU_pkg.all;
28 ----------------------------------------------------------------------------------
29 --ENTITY
30 ----------------------------------------------------------------------------------
31 entity CU_decoder_tb is
32 end CU_decoder_tb;
33
34
35 ----------------------------------------------------------------------------------
36 --ARCHITECTURE
37 ----------------------------------------------------------------------------------
38 architecture rtl of CU_decoder_tb is
39 component CU_decoder
40 port(
41     ----------------------------------------------------------------------------------
42     --input signals
43     ----------------------------------------------------------------------------------
44     -- instruction register
45     ir: in std_logic_vector(31 downto 0);
46     ----------------------------------------------------------------------------------
47     --output signals
48     ----------------------------------------------------------------------------------
49     type_of_instruction_int: out std_logic_vector(3 downto 0);
50     -- PMP ctrl
51     PMP_enable_int: out std_logic;
52     PMP_instruction_int: out std_logic;
53     PMP_size_int: out std_logic_vector(1 downto 0);
54     PMP_rw_int: out std_logic;
55     -- MUX ctrl
56     DMU_IN_MUX_int: out std_logic;
57     DMU_OUT_MUX_int: out std_logic;
58     R_MUX_int: out std_logic;
59     PMP_MUX_int: out std_logic;
60     B_MUX_int: out std_logic;
```

B. CU decoder testbench


```
61      A_MUX_int: out std_logic_vector(1 downto 0);
62      -- reg ctrl
63      reg_read_A_int: out std_logic_vector(4 downto 0);
64      reg_read_B_int: out std_logic_vector(4 downto 0);
65      reg_write_int: out std_logic_vector(4 downto 0);
66      -- CSR ctrl
67      CSR_save_int: out std_logic;
68      CSR_address_int: out std_logic_vector(11 downto 0);
69      CSR_write_int: out std_logic;
70      CSR_read_int: out std_logic;
71      -- exception ctrl
72      iie_CU_int: out std_logic;
73      ece_CU_int: out std_logic;
74      be_CU_int: out std_logic;
75      return_int: out std_logic;
76      -- other signals
77      ALU_op_int: out std_logic_vector(3 downto 0);
78      immediate_int: out std_logic_vector(31 downto 0);
79      mask_ctrl_int: out std_logic_vector(2 downto 0)
80  );
81  end component;
82  ------------------------------------------------------------------------------
83  --signals
84  ------------------------------------------------------------------------------
85      ------------------------------------------------------------------------------
86      --input signals
87      ------------------------------------------------------------------------------
88      -- instruction register
89      signal ir: std_logic_vector(31 downto 0);
90      ------------------------------------------------------------------------------
91      --output signals
92      ------------------------------------------------------------------------------
93      signal type_of_instruction_int: std_logic_vector(3 downto 0);
94      -- PMP ctrl
95      signal PMP_enable_int: std_logic;
96      signal PMP_instruction_int: std_logic;
97      signal PMP_size_int: std_logic_vector(1 downto 0);
98      signal PMP_rw_int: std_logic;
99      -- MUX ctrl
100     signal DMU_IN_MUX_int: std_logic;
101     signal DMU_OUT_MUX_int: std_logic;
102     signal R_MUX_int: std_logic;
103     signal PMP_MUX_int: std_logic;
104     signal B_MUX_int: std_logic;
105     signal A_MUX_int: std_logic_vector(1 downto 0);
106     -- reg ctrl
107     signal reg_read_A_int: std_logic_vector(4 downto 0);
108     signal reg_read_B_int: std_logic_vector(4 downto 0);
109     signal reg_write_int: std_logic_vector(4 downto 0);
110     -- CSR ctrl
111     signal CSR_save_int: std_logic;
112     signal CSR_address_int: std_logic_vector(11 downto 0);
113     signal CSR_write_int: std_logic;
114     signal CSR_read_int: std_logic;
115     -- exception ctrl
116     signal iie_CU_int: std_logic;
117     signal ece_CU_int: std_logic;
118     signal be_CU_int: std_logic;
119     signal return_int: std_logic;
120     -- other signals
```

B. CU decoder testbench


```
121     signal ALU_op_int: std_logic_vector(3 downto 0);
122     signal immediate_int: std_logic_vector(31 downto 0);
123     signal mask_ctrl_int: std_logic_vector(2 downto 0);
124 ----------------------------------------------------------------------------------
125 --constants
126 ----------------------------------------------------------------------------------
127 
128 -- instantiate uut
129 begin
130 dut: CU_decoder port map(
131         ir => ir,
132         type_of_instruction_int => type_of_instruction_int,
133         PMP_enable_int => PMP_enable_int,
134         PMP_instruction_int => PMP_instruction_int,
135         PMP_size_int => PMP_size_int,
136         PMP_rw_int => PMP_rw_int,
137         DMU_IN_MUX_int => DMU_IN_MUX_int,
138         DMU_OUT_MUX_int => DMU_OUT_MUX_int,
139         R_MUX_int => R_MUX_int,
140         PMP_MUX_int => PMP_MUX_int,
141         B_MUX_int => B_MUX_int,
142         A_MUX_int => A_MUX_int,
143         reg_read_A_int => reg_read_A_int,
144         reg_read_B_int => reg_read_B_int,
145         reg_write_int => reg_write_int,
146         CSR_save_int => CSR_save_int,
147         CSR_address_int => CSR_address_int,
148         CSR_write_int => CSR_write_int,
149         CSR_read_int => CSR_read_int,
150         iie_CU_int => iie_CU_int,
151         ece_CU_int => ece_CU_int,
152         be_CU_int => be_CU_int,
153         return_int => return_int,
154         ALU_op_int => ALU_op_int,
155         immediate_int => immediate_int,
156         mask_ctrl_int => mask_ctrl_int
157 );
158 
159 stim: process
160     begin
161         -- parse through different instruction strings
162         --WFI NOP in the beginning to avoid simulation errors
163         ir <= "00010000010100000000000000001110011"; --WFI
164         wait for 100ns;
165         --OPIMM
166         ir <= "00000000000100010000000010010011"; --ADDI
167         wait for 100ns;
168         ir <= "00000000000100010001000010010011"; --SLTI
169         wait for 100ns;
170         ir <= "00000000000100010001100010010011"; --SLTIU
171         wait for 100ns;
172         ir <= "00000000000100010010000010010011"; --XORI
173         wait for 100ns;
174         ir <= "00000000000100010011000010010011"; --ORI
175         wait for 100ns;
176         ir <= "00000000000100010011100010010011"; --ANDI
177         wait for 100ns;
178         ir <= "00000000010000100001000010010011"; --SLLI
179         wait for 100ns;
180         ir <= "00000000010000100101000010010011"; --SRLI
```

B. CU decoder testbench


```
181         wait for 100ns;
182         ir <= "01000000010000100101000010010011"; --SLAI
183         wait for 100ns;
184         --OP
185         ir <= "00000000010000100000000010110011"; --ADD
186         wait for 100ns;
187         ir <= "01000000010000100000000010110011"; --SUB
188         wait for 100ns;
189         ir <= "00000000010000100001000010110011"; --SLL
190         wait for 100ns;
191         ir <= "00000000010000100010000010110011"; --SLT
192         wait for 100ns;
193         ir <= "00000000010000100011000010110011"; --SLTU
194         wait for 100ns;
195         ir <= "00000000010000100100000010110011"; --XOR
196         wait for 100ns;
197         ir <= "00000000010000100101000010110011"; --SRL
198         wait for 100ns;
199         ir <= "01000000010000100101000010110011"; --SRA
200         wait for 100ns;
201         ir <= "00000000010000100110000010110011"; --OR
202         wait for 100ns;
203         ir <= "00000000010000100111000010110011"; --AND
204         wait for 100ns;
205         ir <= "00000000000000000111000010110111"; --LUI
206         wait for 100ns;
207         ir <= "00000000000000000111000010010111"; --AUIPC
208         wait for 100ns;
209         ir <= "00000100000100001001000011101111"; --JAL
210         assert type_of_instruction_int = "0100" report "JAL -> false
    type_of_instruction" severity note;
211         wait for 100ns;
212         ir <= "00000010101000100000000011100111"; --JALR
213         assert type_of_instruction_int = "1000" report "JALR -> false
    type_of_instruction" severity note;
214         -- branch operation
215         wait for 100ns;
216         ir <= "10000010011100100000100011100011"; --BEQ
217         assert type_of_instruction_int = "0010" report "BEQ -> false
    type_of_instruction" severity note;
218         wait for 100ns;
219         ir <= "10000100011100100110100101100011"; --BLTU
220         assert type_of_instruction_int = "0010" report "BLTU -> false
    type_of_instruction" severity note;
221         --exception operations
222         wait for 100ns;
223         ir <= "00000000000000000000000001110011"; --ECALL
224         wait for 100ns;
225         ir <= "00000000000100000000000001110011"; --EBREAK
226         wait for 100ns;
227         ir <= "00110000001000000000000001110011"; --MRET
228         wait for 100ns;
229         ir <= "00010000010100000000000001110011"; --WFI
230         wait for 100ns;
231         --CSRR
232         ir <= "00000000010000110001001001110011"; --CSRRW
233         wait for 100ns;
234         ir <= "00000000010000100101001001110011"; --CSRRWI
235         wait for 100ns;
236         --LOAD
```

B. CU decoder testbench


```
237         ir <= "00000000001000001000001000000011"; --LB
238         wait for 100ns;
239         ir <= "00000000001000001001001000000011"; --LH
240         wait for 100ns;
241         ir <= "00000000001000001010001000000011"; --LW
242         wait for 100ns;
243         --STORE
244         ir <= "00000000000100100000010000100011"; --SB
245         wait for 100ns;
246         ir <= "00000000000100100001010000100011"; --SH
247         wait for 100ns;
248         ir <= "00000000000100100010010000100011"; --SW
249         wait for 100ns;
250     end process;
251 
252 end architecture;
```