

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
Student ID, Course
Reviewer

31 Weeks
6818486, 5040009, TEN18
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	V
List of Figures	VI
List of Tables	VII
Listings	VIII
1 Introduction	1
1.1 Motivation	1
1.2 Project Planning	2
1.3 History	2
2 Basics	3
2.1 Processing Units	3
2.2 RISC vs CISC	3
2.3 RISC-V	4
2.4 Benchmarks	4
2.4.1 CoreMark	5
2.4.2 SPECint	6
2.5 Memory Management	7
2.5.1 Memory Hierarchy	7
2.5.2 Communication Interfaces	7
2.6 FPGA	7
2.7 Hardware Description Languages	8
3 EDRICO	9
3.1 Control Unit	10
3.1.1 Architecture and Design	10
3.1.2 Implementation	14
3.2 Arithmetic Logical Unit	17
3.2.1 Architecture and Design	18
3.2.2 Implementation	18
3.3 Register File (RF)	20
3.3.1 Architecture and Design	21
3.3.2 Implementation	26
3.4 PMP and PMA Checker	26
3.4.1 Architecture and Design	26
3.4.2 Implementation	27

3.5	Exception Control	27
3.5.1	Architecture and Design	28
3.5.2	Implementation	29
3.6	AXI4-Lite Master	29
3.6.1	Architecture and Design	30
3.6.2	Implementation	31
3.7	AXI4-Lite Slave	31
3.7.1	Architecture and Design	31
3.7.2	Implementation	33
4	Test and Verification	34
4.1	Unit Verification	34
4.2	Integration Verification	38
4.3	System Verification	38
5	Future Work	39
6	Conclusion	40
	Bibliography	41
	Appendix	42

Acronyms

ALU	Arithmetic Logical Unit
CISC	Complex Instruction Set Computer
CSR	Control and Status Registers
CU	Control Unit
EEMBC	Embedded Microprocessor Benchmark Consortium
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IP	Intellectual Property
ISA	Instruction Set Architecture
KLOC	Kilo Lines of Code
PMP	Physical Memory Protection
PMA	Physical Memory Attributes
RF	Register File
RISC	Reduced Instruction Set Computer
SISD	Single Instruction Single Data
UUT	Unit Under Test
VHDL	Very High Speed Integrated Circuit Hardware Description Language

List of Figures

1.1	V-Model	2
2.1	Xilinx FPGA [Xil17]	8
3.1	EDRICO Overview	9
3.2	Control Unit Architecture	10
3.3	Control Unit FSM overview	11
3.4	RISC-V Instruction formats [RIS17]	12
3.5	Decoding Structure to determine instruction cluster	15
3.6	Information extraction from 32-bit instruction word	16
3.7	ALU operations	17
3.8	ALU operations	18
3.9	Register Files architecture	21
3.10	General Purpose Registers	22
3.11	CSR Register File Architecture	25
3.12	PMP & PMA Checker Architecture	26
3.13	Exception Control Architecture	28
3.14	AXI4-Lite Master Architecture	30
3.15	AXI4-Lite Slave Architecture	32
4.1	Vivado timing diagram for OPIMM instructions	35
4.2	Vivado timing diagram for full CU decoder testbench	36
4.3	Vivado timing diagram for ALU testbench	38

List of Tables

2.1	RISC vs CISC [Hel16]	4
2.2	SPEC CPU2017 benchmark suite [KLK20]	5
2.3	SPECint2006 programs [SPE06]	7
3.1	Timing of FSM	11
3.2	Program Counter control: Instructions and resulting actions	13
3.3	Decoding instruction clusters	14
3.4	Input code and respective operation	19
3.5	List of implemented CSRs	24
3.6	Memory Mapped CSRs	31
3.7	Memory Mapped CSRs reset values	33
4.1	ALU testbench correct outputs	37

Listings

3.1	ALU VHDL code	20
4.1	CU testbench stimulation process	34
4.2	CU testbench stimulation process	36

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 Intellectual 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.

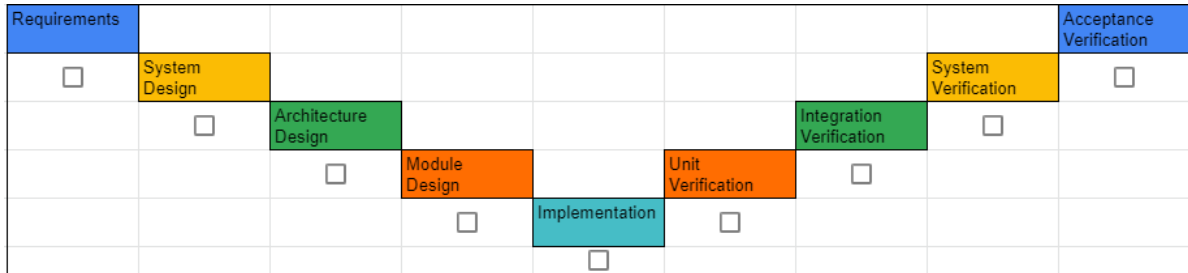


Figure 1.1: V-Model

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

2 Basics

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

2.1 Processing Units

????????????????

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
<i>602.gcc_s</i>	C	1,304	GNU C compiler
<i>625.X264_s</i>	C	96	Video compression
<i>631.deepsjeng_s</i>	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 $\langle \textit{Coremark} \rangle$ and $\langle \textit{SPECint} \rangle$ 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 ratios is computed. The 12 programs of the SPECint2006 benchmark are displayed in the following table 2.3:

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

Table 2.3: SPECint2006 programs [SPE06]

2.5 Memory Management

2.5.1 Memory Hierarchy

2.5.2 Communication Interfaces

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 like BRAM and DSP slices are connected to each other through configurable routing channels. Figure 2.1 shows the basic architecture of a Xilinx FPGA:

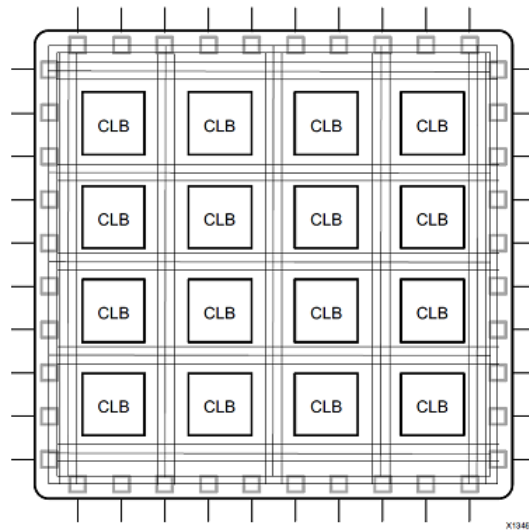


Figure 2.1: Xilinx FPGA [Xil17]

2.7 Hardware Description Languages

The CLBs in this architecture are comprised of LUTs and Flip-Flops, in order to implement boolean functions and allow the design of synchronous circuits. FPGAs produced by Xilinx are mostly SRAM based, other approaches are flash or anti-fuse based architectures.

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 S-Mode and U-Mode.

The core is a simple Single Instruction Single Data (SISD) processor without any pipeline or even 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:

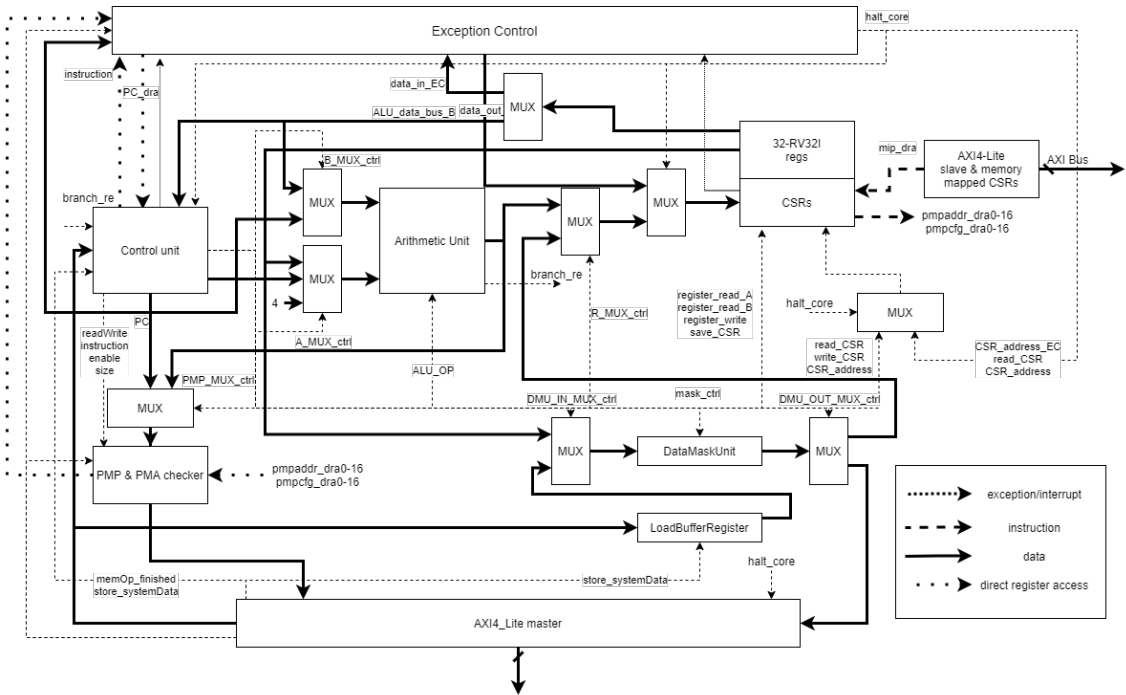


Figure 3.1: EDRICO Overview

Its main components are the Exception Control, Control Unit, Arithmetic Unit, Register Files, PMP & PMA checker and the AXI4 Interfaces. Each one of the components will be described in more detail in the following section.

3.1 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.

3.1.1 Architecture and Design

A general overview of the CU architecture is displayed in Figure 3.2.

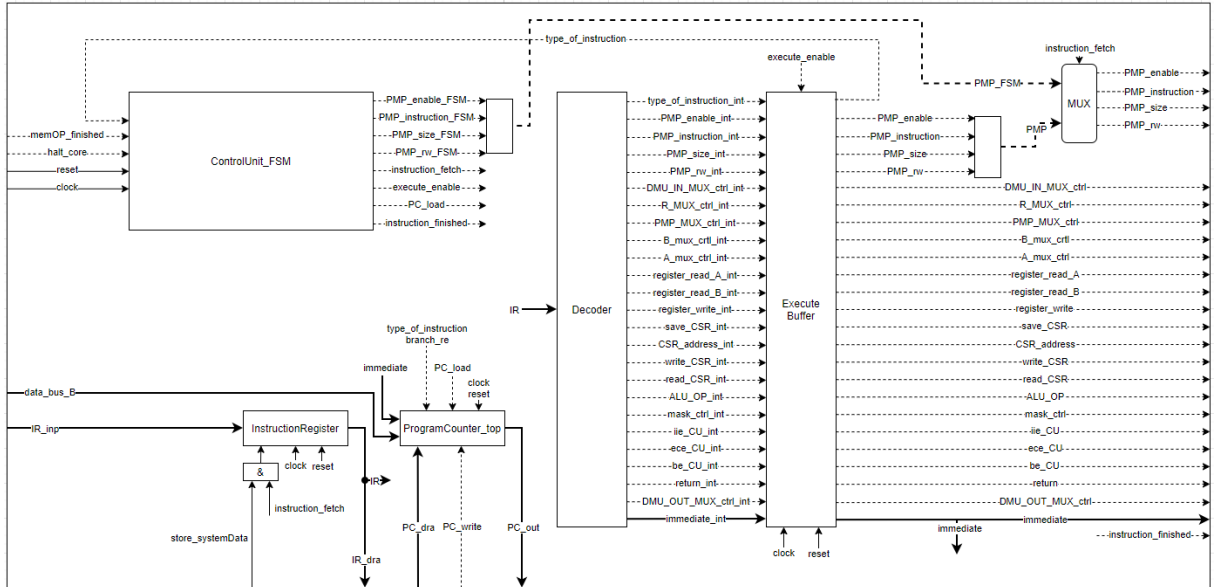


Figure 3.2: Control Unit Architecture

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 3.3.

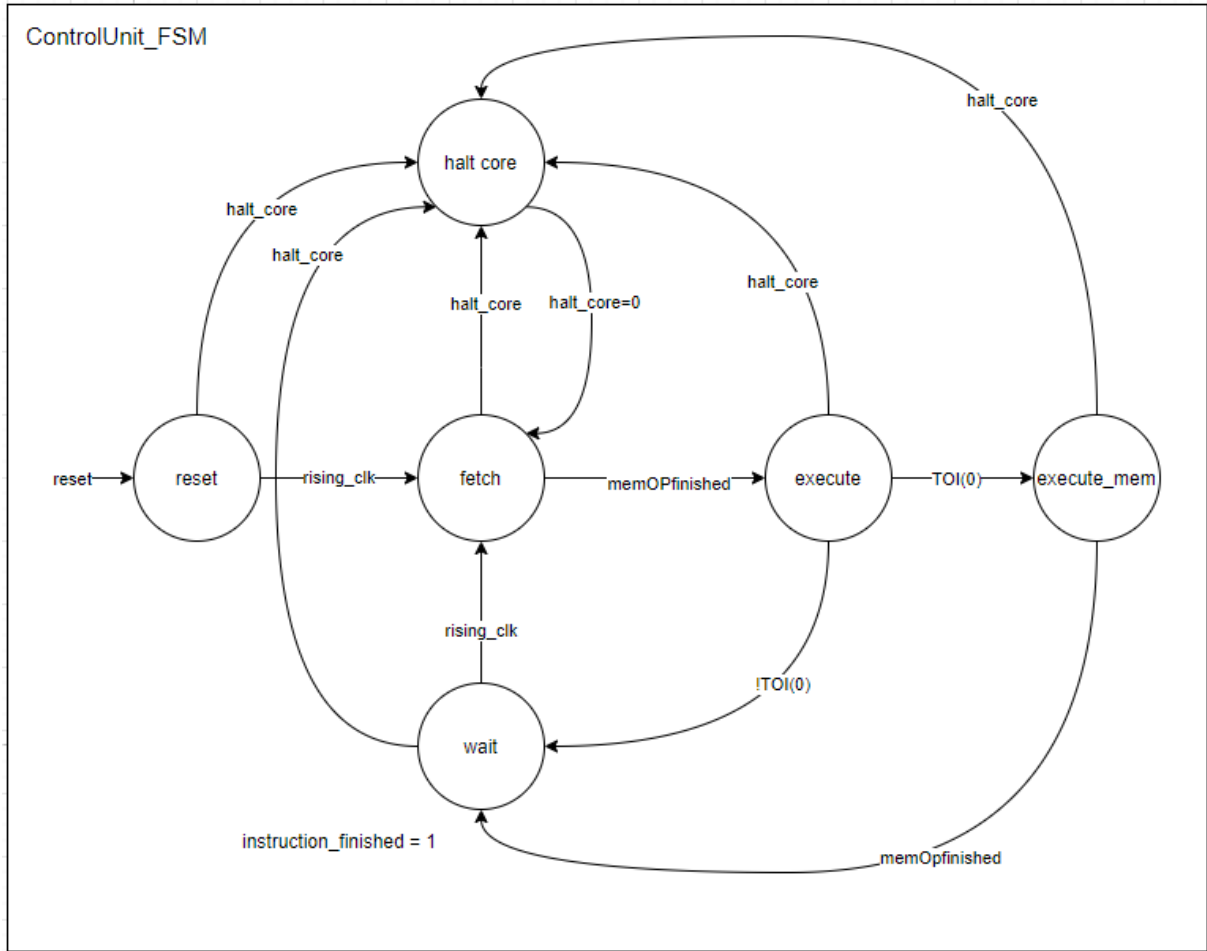


Figure 3.3: Control Unit FSM overview

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	<i>memOPfinished</i> & <i>store_systemData</i> is high
5	rising	execution is started - if memory operation wait for another <i>memOPfinished</i> flag, otherwise wait	<i>execute_enable</i>
x	rising	during memory operation: data loaded to buffer \store transfer finished → wait state	<i>memOPfinished</i> & <i>store_systemData</i> is high
	falling	if load: store data form buffer to specified location	
6 / x+1	rising	go to <i>fetch_state</i>	

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 3.2)

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 3.4.

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

Figure 3.4: RISC-V Instruction formats [RIS17]

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 3.1.2.

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 3.2) 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 3.1.1:

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 3.2 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 3.1.2.

3.1.2 Implementation

The implementation of the Control Unit is split up into multiple sub-implementations. As shown in figure 3.2 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 3.1.1 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 3.5:

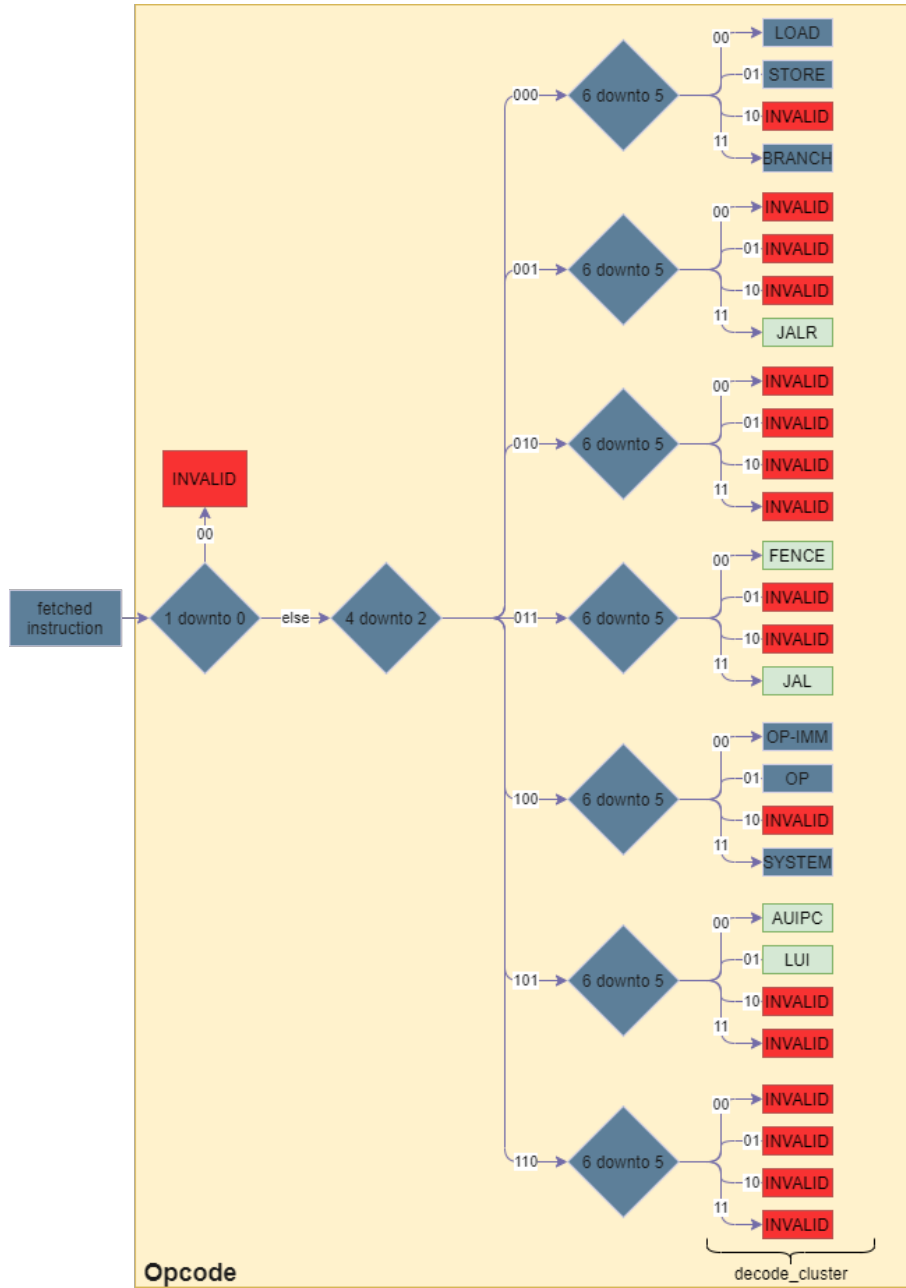


Figure 3.5: Decoding Structure to determine instruction cluster

After determining the cluster, the VHDL code assigns all the outputs visible in figure 3.2 with the respective information. For a better understanding of the information extraction, figure 3.6 shows how the 32-bit instruction word is split up (in this case for the *OP* and *OP-IMM* instructions.)

16

3.2 Arithmetic Logical Unit

The Arithmetic Logical Unit (ALU) is the part of the processor that performs the arithmetic and logical operations. Figure 3.7 gives an overview of what type of operations are performed.

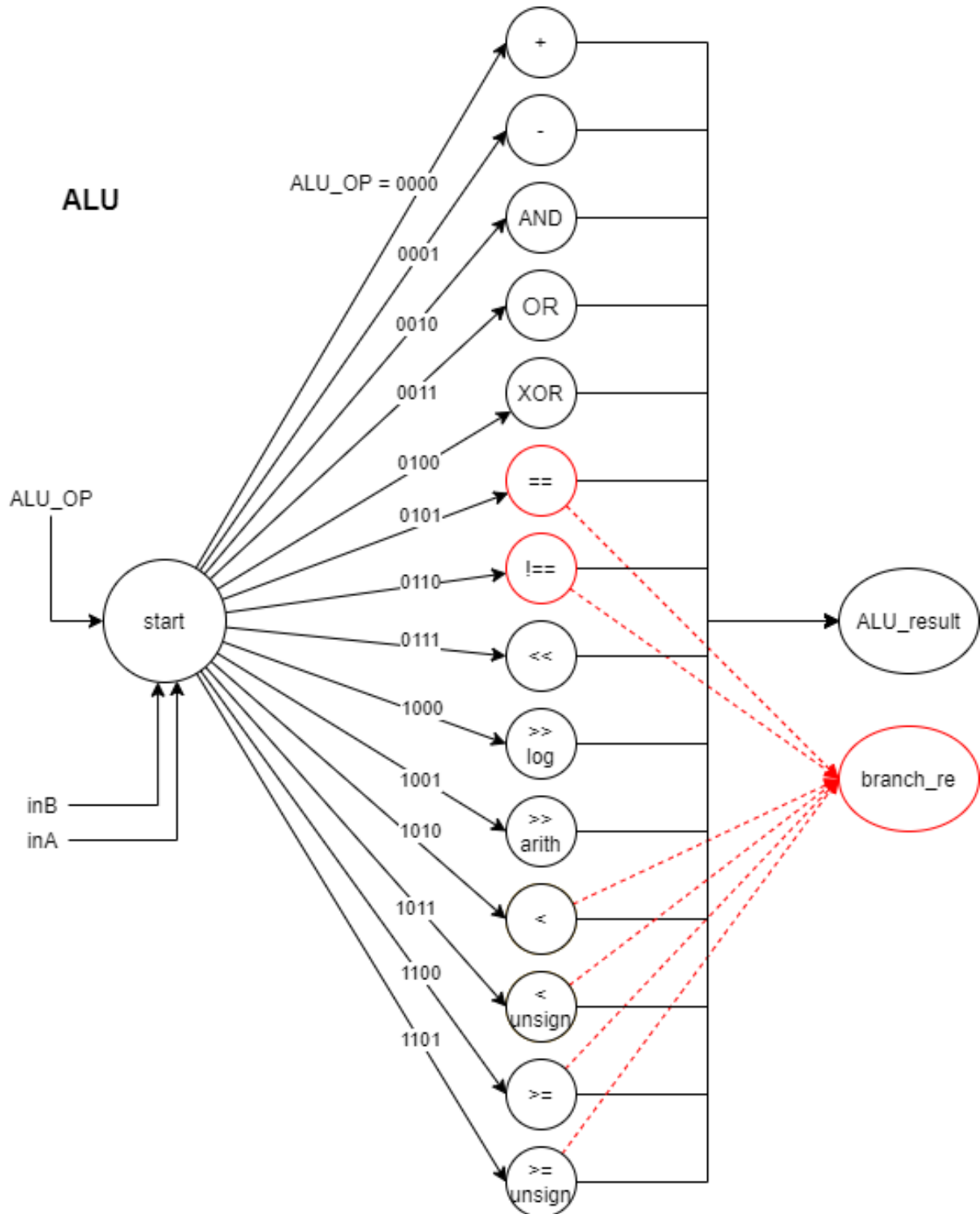


Figure 3.7: ALU operations

3.2.1 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 3.8:



Figure 3.8: 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.

3.2.2 Implementation

The implementation of the ALU is based on figure 3.7 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.

3.3 Register File (RF)

The Register can be used to store data and configure the core. General Purpose registers are mostly used to temporarily store data like operands and results of calculations. The CSR Register File is used to configure the core and retrieve information about it. More detailed information about these two RFs can be found in the following chapters.

3.3.1 Architecture and Design

The architecture of the register files is shown in following figure 3.9

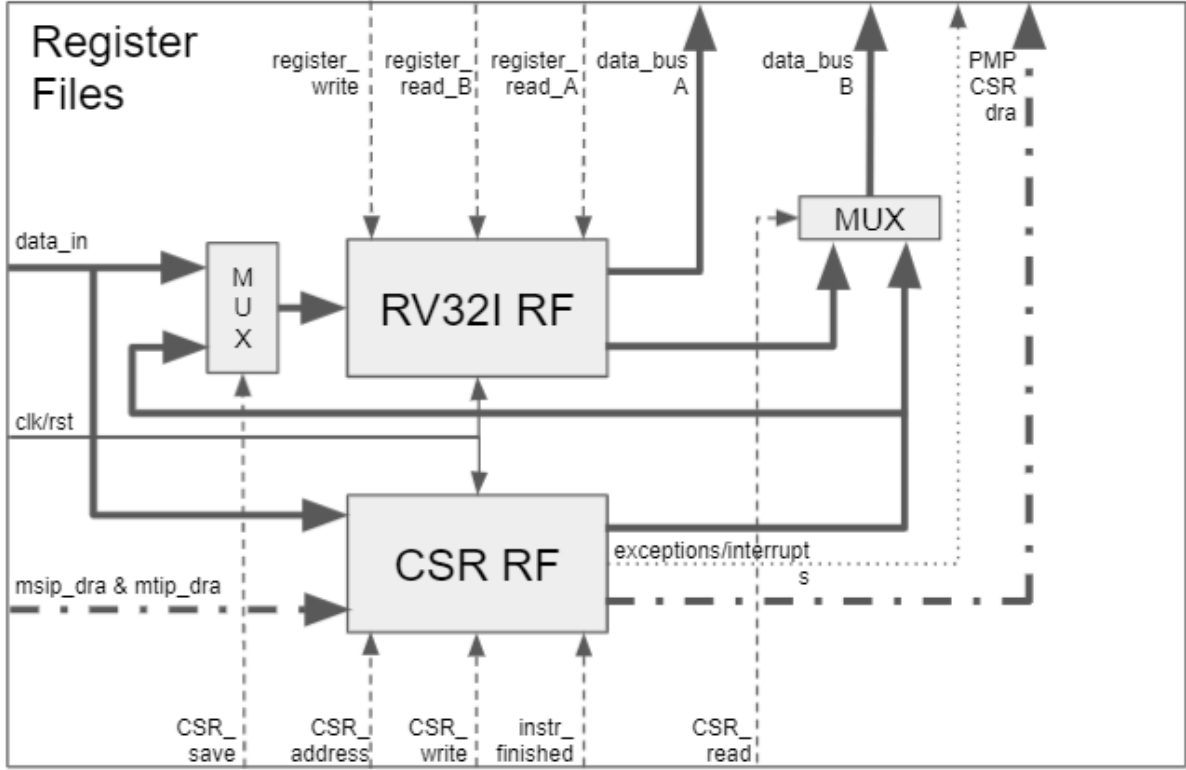


Figure 3.9: Register Files architecture

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.

If the *CSR_save* signal is applied, the CSR Register File output is connected to the data input bus of the general purpose registers. This allows to save CSR data directly to the RV32I registers without using the system data bus. Some of the CSR registers allow direct memory accesses, e.g. the *msip_dra* & *mtip_dra* signals are used to set the software and timer interrupt pending bits respectively. To increase the performance of memory accesses, the PMP CSRs can be read via a dedicated bus.

While the CSR RF is accessed by 12 Bit addresses, RV32I registers are accessed without addressing in order to decrease decoding time. Therefore, each register requires three

Signals resulting in a total of 96 signals for read and write. The RFs are clocked with the system clock and reset on system reset. As per usual, data is stored on the falling clock and displayed on the rising clock.

General Purpose Registers

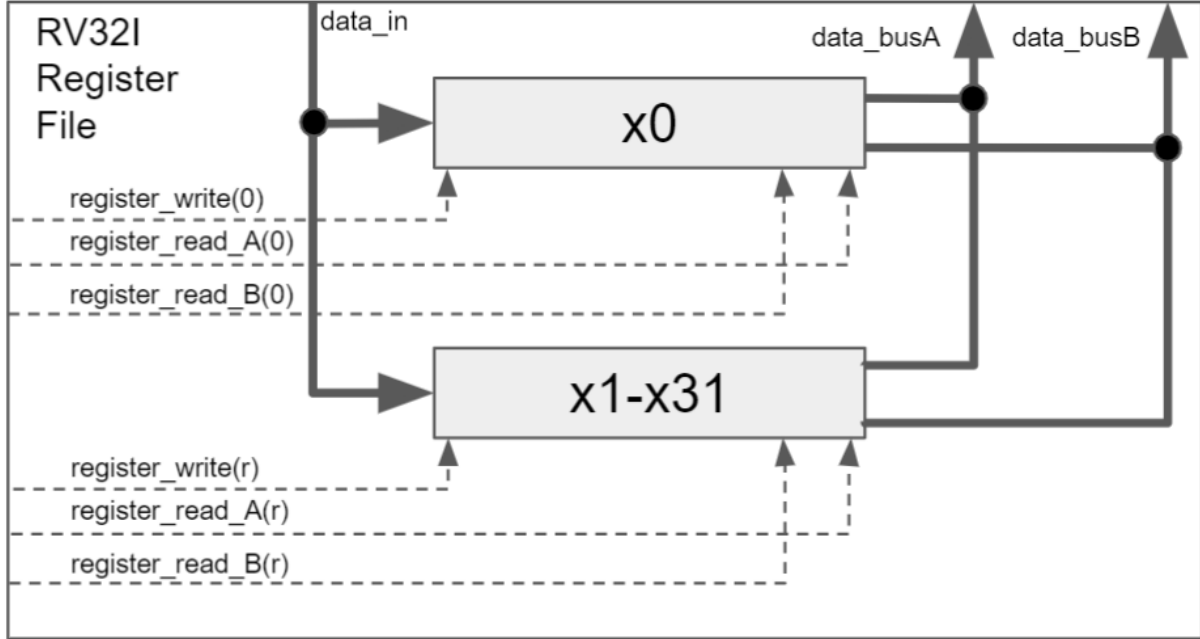


Figure 3.10: General Purpose Registers

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 Registers (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 slave module implementing all memory mapped CSRs is added to the design.

Further Information about this module can be found in the chapter 3.7. 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 3.3.1 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
mvvendorid	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 mhpcounter
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 3.11.

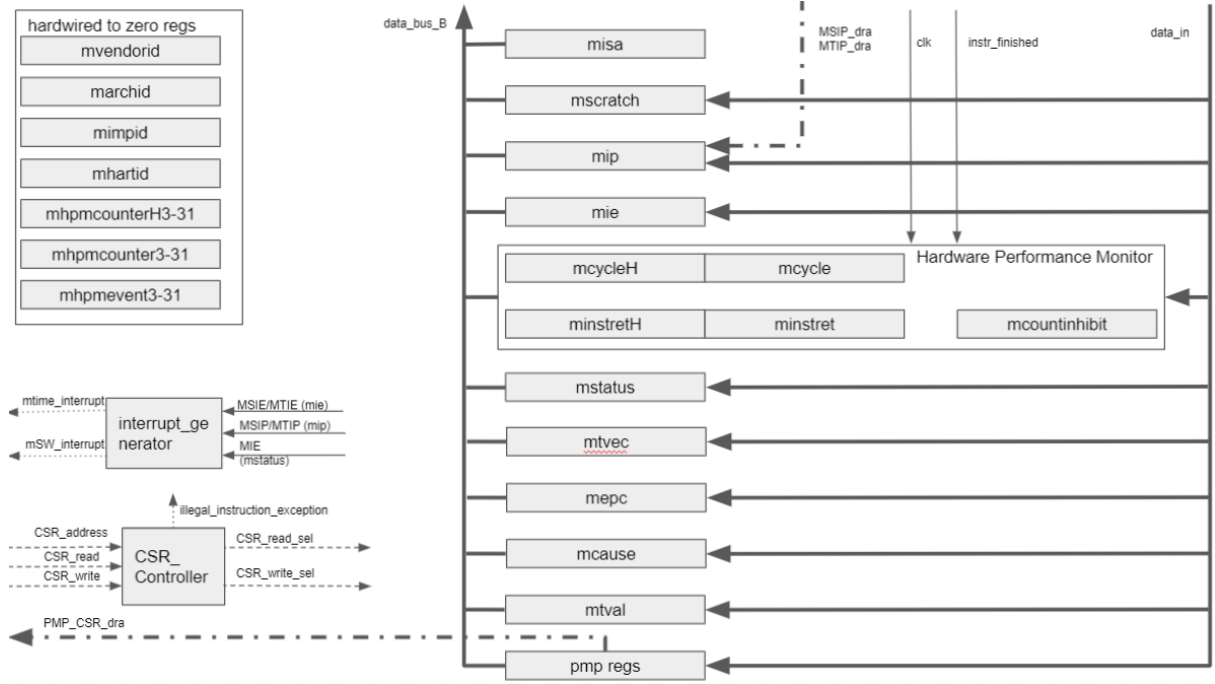


Figure 3.11: CSR Register File Architecture

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 ISR must clear the pending interrupts, e.g. the timer interrupt by writing to the memory mapped CSRs.

The Hardware Performance Monitor registers are incremented at special events (clock cycle and instruction finish) therefore two 64-Bit increment units have to be implemented. The increments are only performed, if the corresponding bit inside the *mcounterinhibit* register is set.

The PMP CSRs are composed of the 16 *pmpaddr* registers and four *pmpcfg* registers. They can be read and written, additional direct memory access is implemented in order to decrease memory access latency

3.3.2 Implementation

3.4 PMP and PMA Checker

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

3.4.1 Architecture and Design

An overview of the PMP & PMA Checker is presented by 3.12:

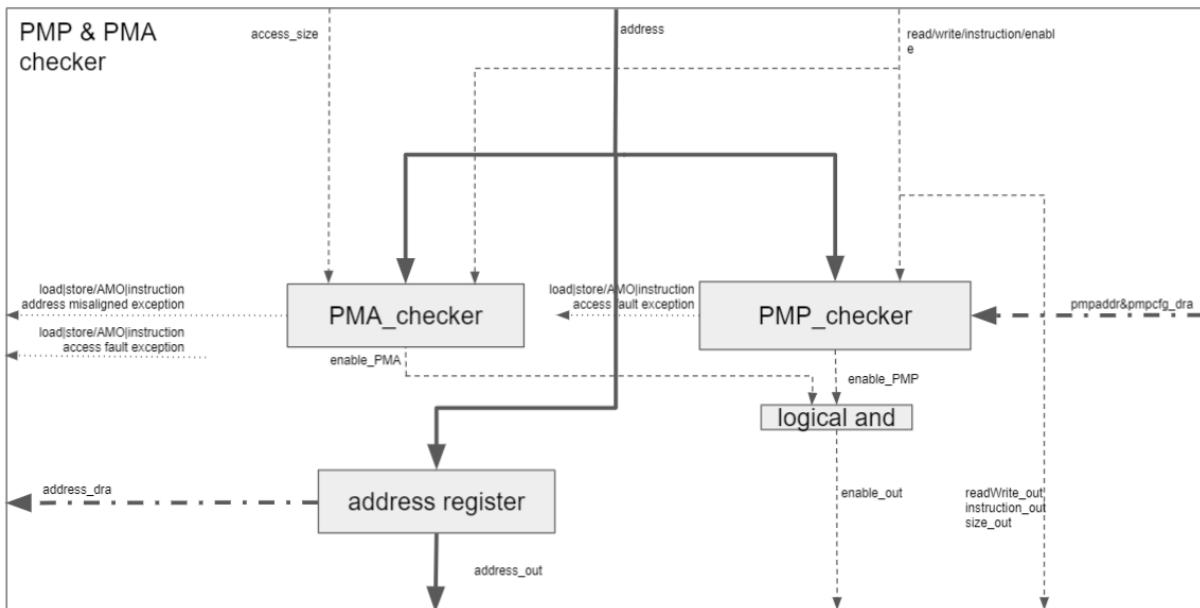


Figure 3.12: PMP & PMA Checker Architecture

The PMP Unit 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 may succeed.

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. The PMP unit has direct access to those and enforces the rules. If an address is not specified every access is allowed.

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.

3.4.2 Implementation

3.5 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

3.5.1 Architecture and Design

Figure 3.13 shows the architecture of the Exception Control:

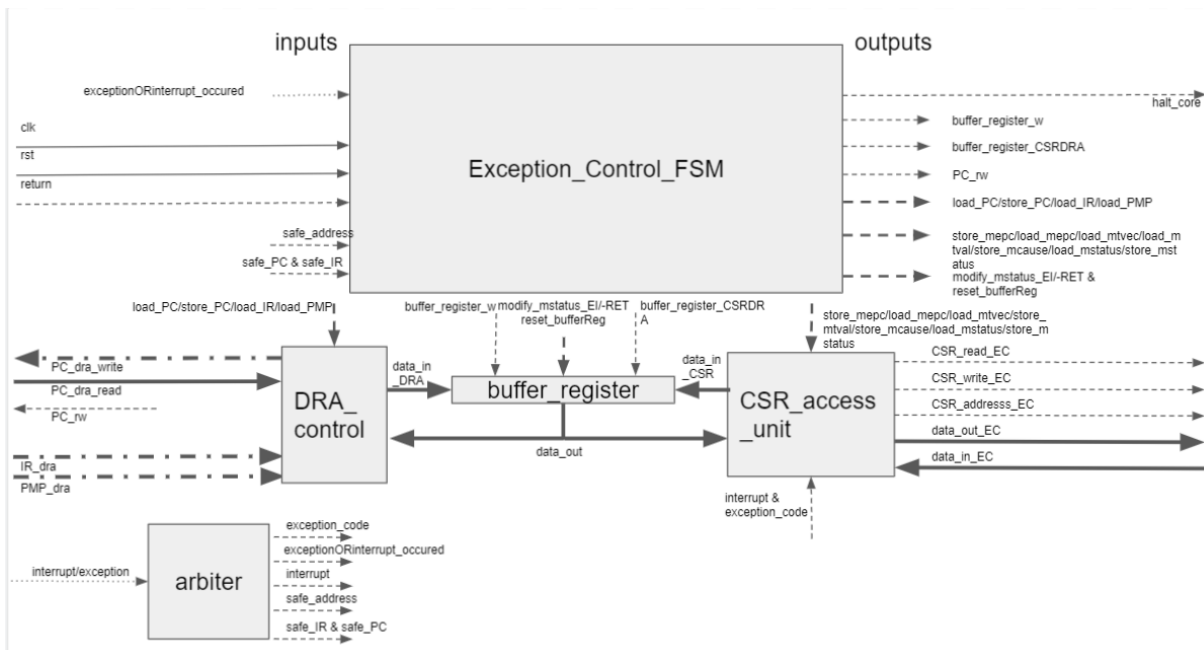


Figure 3.13: Exception Control Architecture

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.

3.5.2 Implementation

3.6 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:

3.6.1 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:

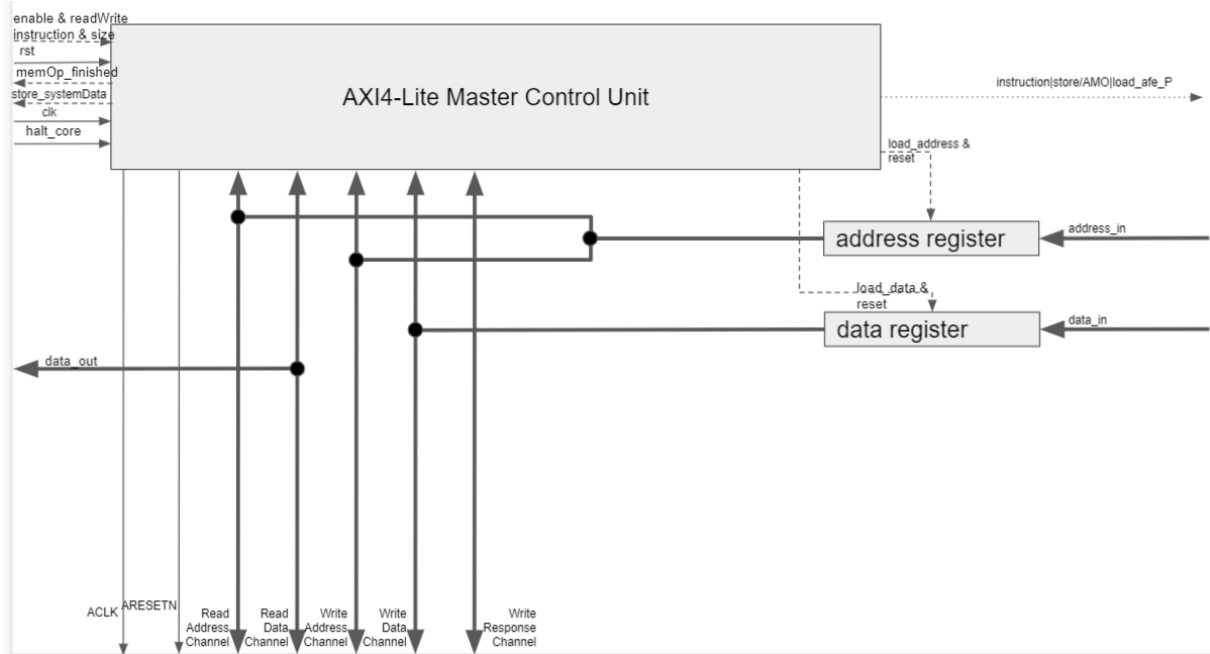


Figure 3.14: AXI4-Lite Master Architecture

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.

3.6.2 Implementation

3.7 AXI4-Lite Slave

Some of the RISC-V CSRs must be memory mapped, meaning they 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

3.7.1 Architecture and Design

The AXI4-Lite slave is in charge of accepting data transfers and reading/writing the CSRs. Figure 3.15 shows the architecture, including the memory mapped CSRs:

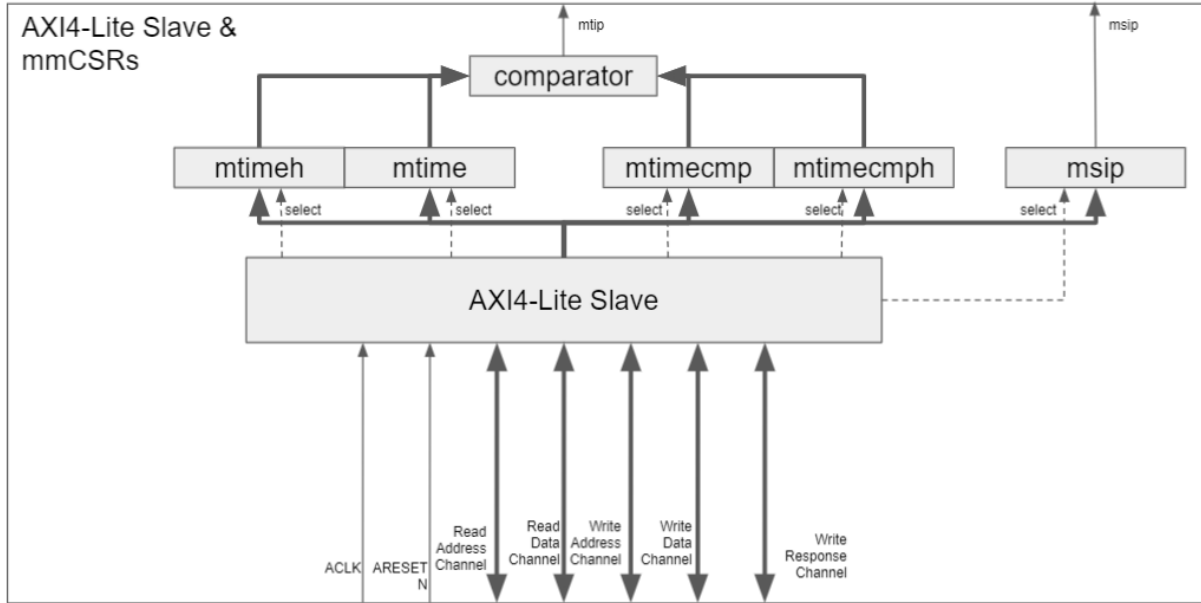


Figure 3.15: AXI4-Lite Slave Architecture

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 3.7.1 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

3.7.2 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.

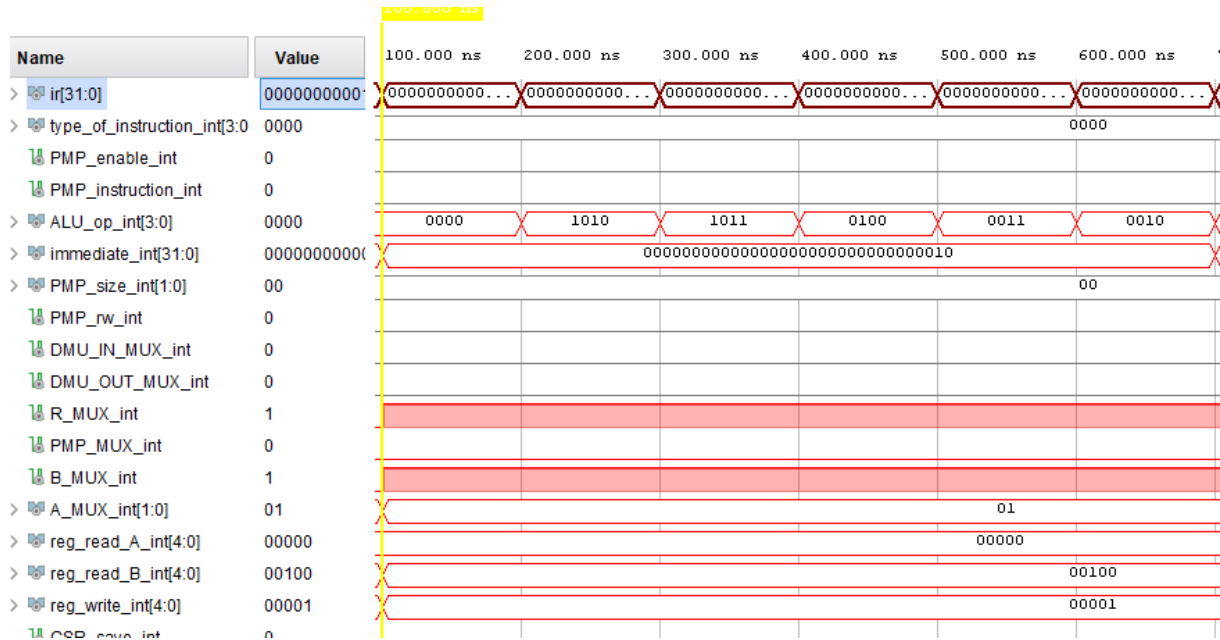


Figure 4.1: Vivado timing diagram for OPIMM instructions

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 3.2.2). 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.

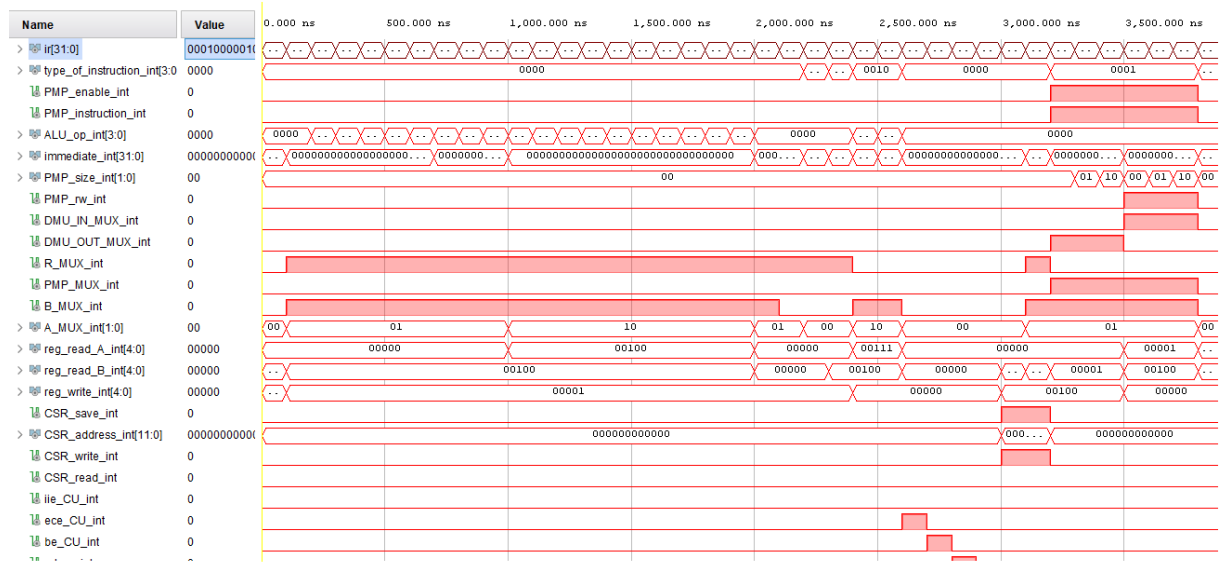


Figure 4.2: Vivado timing diagram for full CU decoder testbench

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:

```

1 stim: process
2 begin
3     -- set input signal
4     in_a <= "0000000000000000000000000000010";
5     in_b <= "00000000000000000000000001000000";
6
7     -- set op signal
8     alu_op <= "0000"; --ADD
9     wait for 100ns;
10    alu_op <= "0001"; --SUB
11    wait for 100ns;
12    alu_op <= "0010"; --AND
13    wait for 100ns;
14    alu_op <= "0011"; --OR
15    wait for 100ns;
16    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.

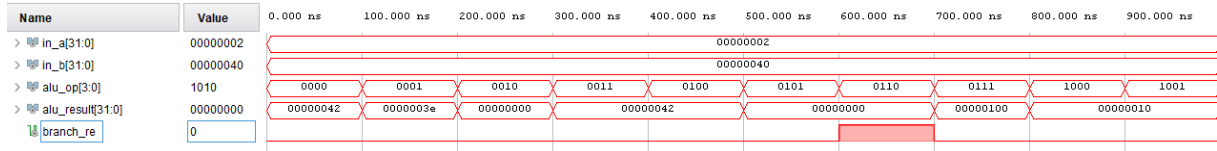


Figure 4.3: Vivado timing diagram for ALU testbench

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

5 Future Work

6 Conclusion

Bibliography

- [Ges14] Ralf Gessler. *Entwicklung Eingebetteter Systeme*. Wiesbaden: Springer, 2014.
- [Hel16] Roland Hellmann. *Rechnerarchitektur*. Aalen: De Gruyter Oldenbourg, 2016.
- [KLK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking*. Würzburg, Germany: Springer, 2020.
- [RIS17] RISC-V. *The RISC-V Instruction Set Manual*. 2017. URL: <https://riscv.org/technical/specifications>.
- [SPE06] SPEC. *CINT2006 (Integer Component of SPEC CPU2006)*. 2006. URL: <https://www.spec.org/cpu2006/CINT2006/>.
- [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

18.7.2021

CU_decoder.vhd

```
1  -----
2  -- Company: DHBW
3  -- Engineer: Noah Woelki
4  --
5  -- Create Date: 05/10/2021 06:04:15 AM
6  -- Design Name: CU_decoder
7  -- Module Name: Control Unit
8  -- Project Name: EDRICO
9  -- Target Devices: Arty Z7
10 -- Tool Versions:
11 -- Description:
12 --   Decoding unit decodes incoming instruction once its loaded from memory
13 --
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 -----
27 --ENTITY
28 -----
29 entity CU_decoder is
30 port (
31     -----
32     --input signals
33     -----
34     -- instruction register
35     ir: in std_logic_vector(31 downto 0);
36     -----
37     --output signals
38     -----
39     type_of_instruction_int: out std_logic_vector(3 downto 0);
40     -- PMP ctrl
41     PMP_enable_int: out std_logic;
42     PMP_instruction_int: out std_logic;
43     PMP_size_int: out std_logic_vector(1 downto 0);
44     PMP_rw_int: out std_logic;
45     -- MUX ctrl
46     DMU_IN_MUX_int: out std_logic;
47     DMU_OUT_MUX_int: out std_logic;
48     R_MUX_int: out std_logic;
49     PMP_MUX_int: out std_logic;
50     B_MUX_int: out std_logic;
51     A_MUX_int: out std_logic_vector(1 downto 0);
52     -- reg ctrl
53     reg_read_A_int: out std_logic_vector(4 downto 0);
54     reg_read_B_int: out std_logic_vector(4 downto 0);
55     reg_write_int: out std_logic_vector(4 downto 0);
56     -- CSR ctrl
57     CSR_save_int: out std_logic;
58     CSR_address_int: out std_logic_vector(11 downto 0);
59     CSR_write_int: out std_logic;
60     CSR_read_int: out std_logic;
```

A. CU decoder

18.7.2021

CU_decoder.vhd

```
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,
82 OP, SYSTEM, AUIPC, Lui);
83 signal decoded_cluster : instruction_cluster;
84 -----
85 --constants
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

18.7.2021

CU_decoder.vhd

```
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
176         an immediate
177         -- and data from register. loaded data stores in respective destination
178         register
179         type_of_instruction_int <= "0001";
```

A. CU decoder

18.7.2021

CU_decoder.vhd

```
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
230         when "001" => --SH
231             PMP_size_int <= "01";
232             mask_ctrl_int <= "011";
233
234         when "010" => --SW
```

A. CU decoder

```
18.7.2021 CU_decoder.vhd
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
252             when "001" => --BNE
253                 ALU_op_int <= "0110";
254             when "100" => --BLT
255                 ALU_op_int <= "1010";
256             when "101" => --BGE
257                 ALU_op_int <= "1100";
258             when "110" => --BLTU
259                 ALU_op_int <= "1011";
260             when "111" => --BLEU
261                 ALU_op_int <= "1101";
262             when others =>
263                 iie_CU_int <= '1';
264         end case;
265     when JALR => --JALR
266         -- JALR adds + 4 to the program counter and stores it to rd
267         -- target address is obtained by PC_top control which adds regB to the
immediate
268         R_MUX_int <= '1';
269         type_of_instruction_int <= "1000";
270         reg_read_B_int <= ir(19 downto 15);
271         reg_write_int <= ir(11 downto 7);
272         immediate_int <= std_logic_vector((31 downto 12 => ir(31)) & ir(31
downto 20));
273     when FENCE => --NOP
274     when JAL => --JAL
275         -- JAL stores PC +4 into rd, jump target address obtained by PC_top (adds
immediate to PC)
276         R_MUX_int <= '1';
277         type_of_instruction_int <= "0100";
278         reg_write_int <= ir(11 downto 7);
279         immediate_int <= std_logic_vector((31 downto 20 => ir(31)) & ir(31) &
ir(19 downto 12) & ir(20) & ir(30 downto 21));
280     when OPIMM =>
281         -- Immediate operations perform ALU operations on data from regB and
immediate
282         R_MUX_int <= '1';
283         PMP_MUX_int <= '0';
284         B_MUX_int <= '1';
285         A_MUX_int <= "01";
286         reg_read_B_int <= ir(19 downto 15);
```


A. CU decoder

```
18.7.2021 CU_decoder.vhd
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             when "110" => --ORI
313                 ALU_op_int <= "0011";
314                 immediate_int <= std_logic_vector((31 downto 12 =>
ir(31)) & ir(31 downto 20));
315             when "111" => --ANDI
316                 ALU_op_int <= "0010";
317                 immediate_int <= std_logic_vector((31 downto 12 =>
ir(31)) & ir(31 downto 20));
318             when others =>
319                 iie_CU_int <= '1';
320         end case;
321     when OP =>
322         -- Normal operation performs ALU operation on regA and regB, stores in rd
(regwrite)
323         R_MUX_int <= '1';
324         PMP_MUX_int <= '0';
325         B_MUX_int <= '1';
326         A_MUX_int <= "10";
327         reg_read_A_int <= ir(24 downto 20);
328         reg_read_B_int <= ir(19 downto 15);
329         reg_write_int <= ir(11 downto 7);
330
331         case ir(14 downto 12) is
332             when "000" => --ADD/SUB
333                 if(ir(30) = '0') then --ADD
334                     ALU_op_int <= "0000";
```

A. CU decoder

18.7.2021

CU_decoder.vhd

```
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
353     when "110" => --OR
354         ALU_op_int <= "0011";
355     when "111" => --AND
356         ALU_op_int <= "0010";
357     when others =>
358         iie_CU_int <= '1';
359     end case;
360 when SYSTEM =>
361     CSR_save_int <= '1';
362     CSR_address_int <= ir(31 downto 20);
363     CSR_write_int <= '1';
364
365     case ir(14 downto 12) is
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

18.7.2021

```
CU_decoder.vhd
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
402         return_int <= '1';
403     else --INVALID
404         ece_CU_int <= '0';
405         be_CU_int <= '0';
406         iie_CU_int <= '1';
407     end if;
408 when "101" => --WFI -> NOP
409     ece_CU_int <= '0';
410     be_CU_int <= '0';
411 when others =>
412     ece_CU_int <= '0';
413     be_CU_int <= '0';
414     iie_CU_int <= '1';
415 end case;
416 when "001" => --CSRRW
417     R_MUX_int <= '0';
418     B_MUX_int <= '0';
419     A_MUX_int <= "00";
420     reg_read_B_int <= ir(19 downto 15);
421     reg_write_int <= ir(11 downto 7);
422
423     CSR_read_int <= '0';
424     ALU_op_int <= "0000";
425
426 when "010" => --CSRRS
427     R_MUX_int <= '1';
428     B_MUX_int <= '1';
429     A_MUX_int <= "10";
430     reg_read_B_int <= ir(19 downto 15);
431     reg_write_int <= ir(11 downto 7);
432
433     CSR_read_int <= '1';
434     ALU_op_int <= "0011";
435 when "011" => --CSRRC
436     R_MUX_int <= '1';
437     B_MUX_int <= '1';
438     A_MUX_int <= "10";
439     reg_read_B_int <= ir(19 downto 15);
440     reg_write_int <= ir(11 downto 7);
441
442     CSR_read_int <= '1';
443     ALU_op_int <= "0010";
444 when "101" => --CSRRWI
445     R_MUX_int <= '1';
446     B_MUX_int <= '1';
447     A_MUX_int <= "01";
448     reg_read_B_int <= ir(19 downto 15);
449     reg_write_int <= ir(11 downto 7);
450
451     CSR_read_int <= '0';
452     ALU_op_int <= "0000";
```

A. CU decoder

```
18.7.2021 CU_decoder.vhd
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 pgroom 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

18.7.2021

CU_decoder_tb.vhd

```
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

18.7.2021

CU_decoder_tb.vhd

```
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

18.7.2021

CU_decoder_tb.vhd

```
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 <= "00010000010100000000000001110011"; --WFI
164         wait for 100ns;
165         --OPIMM
166         ir <= "00000000001000100000000010010011"; --ADDI
167         wait for 100ns;
168         ir <= "00000000001000100010000010010011"; --SLTI
169         wait for 100ns;
170         ir <= "00000000001000100011000010010011"; --SLTIU
171         wait for 100ns;
172         ir <= "00000000001000100100000010010011"; --XORI
173         wait for 100ns;
174         ir <= "00000000001000100110000010010011"; --ORI
175         wait for 100ns;
176         ir <= "00000000001000100111000010010011"; --ANDI
177         wait for 100ns;
178         ir <= "00000000010000100001000010010011"; --SLLI
179         wait for 100ns;
180         ir <= "00000000010000100101000010010011"; --SRLI
```

B. CU decoder testbench

18.7.2021

CU_decoder_tb.vhd

```
181     wait for 100ns;
182     ir <= "01000000010000100101000010010011"; --SLAI
183     wait for 100ns;
184     --OP
185     ir <= "000000000100001000000000010110011"; --ADD
186     wait for 100ns;
187     ir <= "010000000100001000000000010110011"; --SUB
188     wait for 100ns;
189     ir <= "00000000010000100001000010110011"; --SLL
190     wait for 100ns;
191     ir <= "000000000100001000100000010110011"; --SLT
192     wait for 100ns;
193     ir <= "00000000010000100011000010110011"; --SLTU
194     wait for 100ns;
195     ir <= "000000000100001001000000010110011"; --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 <= "00000100000100001001000011011111"; --JAL
210     assert type_of_instruction_int = "0100" report "JAL -> false
type_of_instruction" severity note;
211     wait for 100ns;
212     ir <= "00000010101000100000000001110011"; --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

18.7.2021

CU_decoder_tb.vhd

```
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 <= "0000000000100100000010000100011"; --SB
245     wait for 100ns;
246     ir <= "0000000000100100001010000100011"; --SH
247     wait for 100ns;
248     ir <= "0000000000100100010010000100011"; --SW
249     wait for 100ns;
250     end process;
251
252 end architecture;
```