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	12 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

Contents

Acronyms

ALU	Arithmetic Logical Unit
CU	Control Unit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IP	Intellectual Property
ISA	Instruction Set Architecture
PMP	Physical Memory Protection
PMA	Physical Memory Attributes
RF	Register File
RISC	Reduced Instruction Set Computer
SISD	Single Instruction Single Data
VHDL	Very High Speed Integrated Circuit Hardware Description Language

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 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.

2 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.

3 Project Planning

In order to control the flow of the project, the V-Model 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.

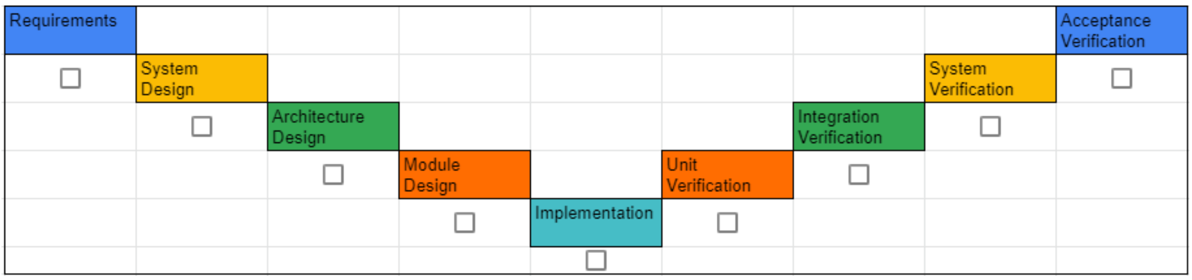


Figure 3.1: V-Model

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.

4 State of the Art

4.1 Basics of Processing Units

4.2 RISC vs CISC

4.3 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.

4.4 Benchmarks

4.4.1 Coremark

4.4.2 SPECint

4.5 Memory Management

4.5.1 Memory Hierarchy

4.5.2 Communication Interfaces

4.6 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 4.1 shows the basic architecture of a Xilinx FPGA:

4.7 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.

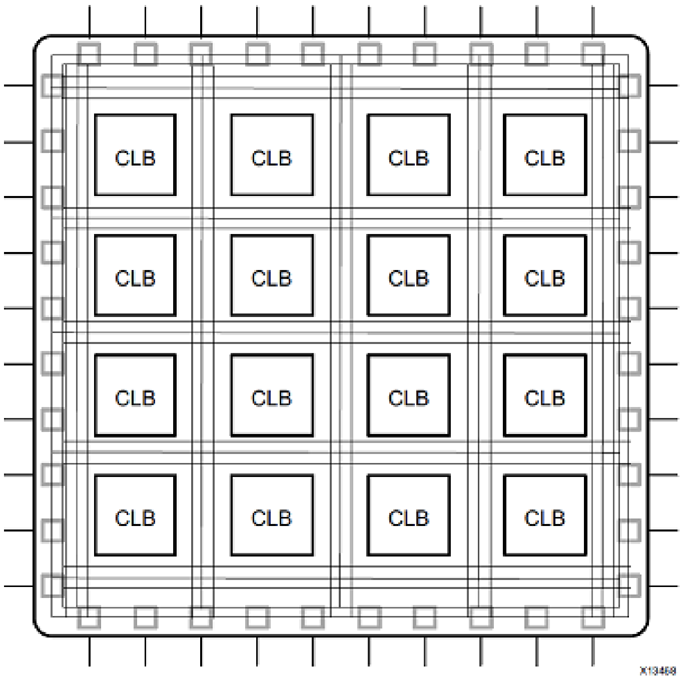


Figure 4.1: Xilinx FPGA [Xil17]

5 EDRICO (Educational DHBW RISC-V Core)

The Proposed Processor design named 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 5.1 shows the full overview of the processor design:

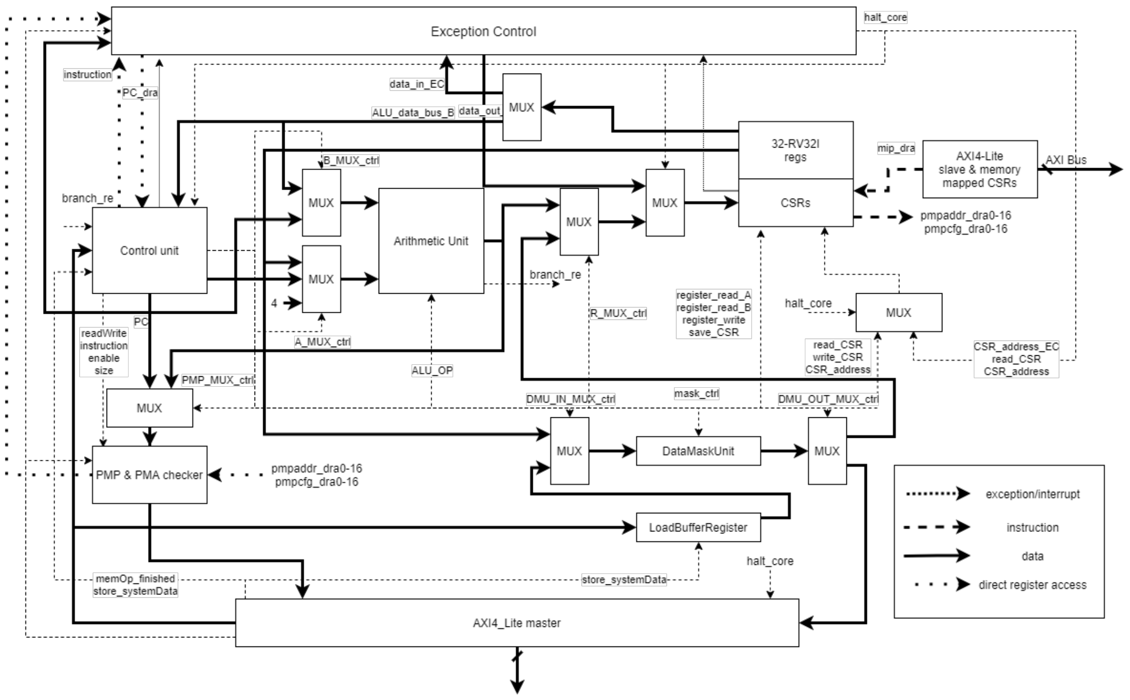


Figure 5.1: EDRICO Overview

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.

5.1 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.

5.1.1 Architecture & Design

A general overview of the CU architecture is displayed in Figure 5.2.

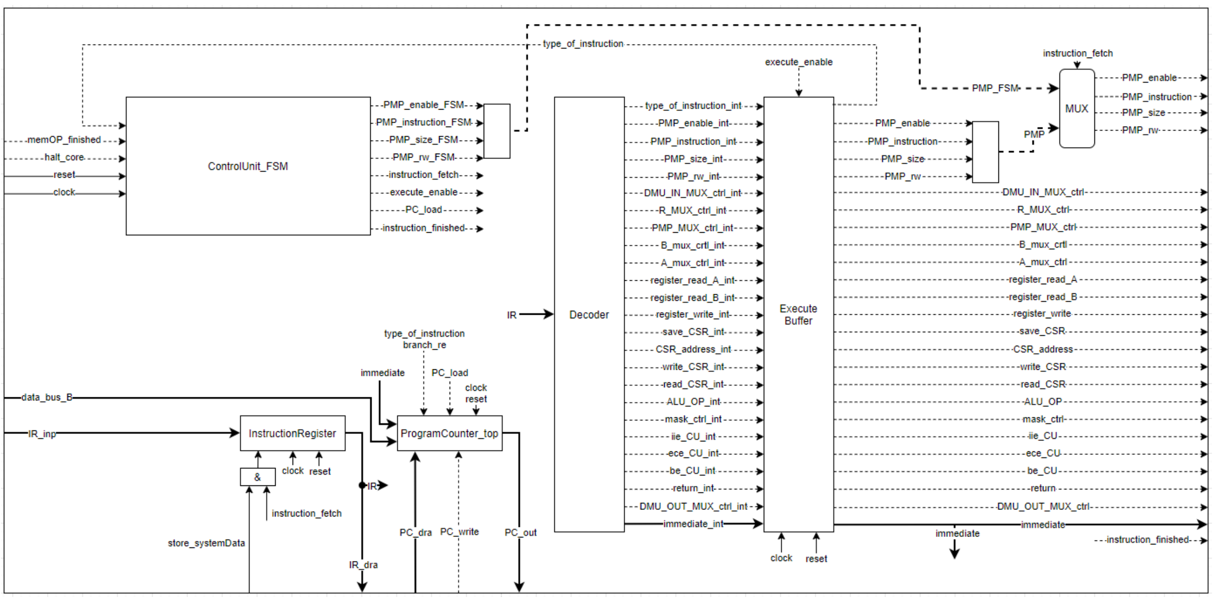


Figure 5.2: Control Unit Architecture

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 5.3.

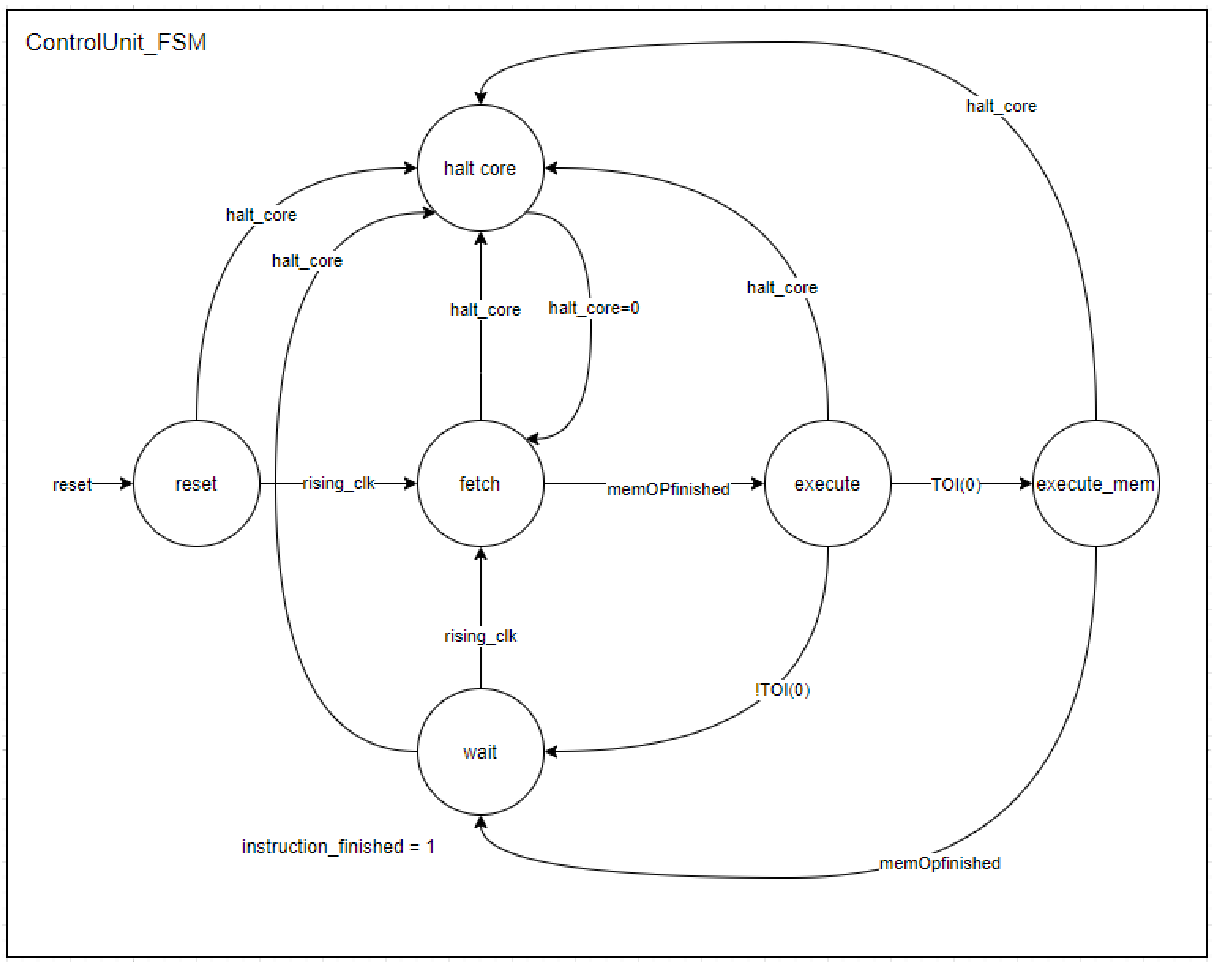


Figure 5.3: Control Unit FSM overview

Table 5.1 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 5.1: 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 5.2)
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 5.4.

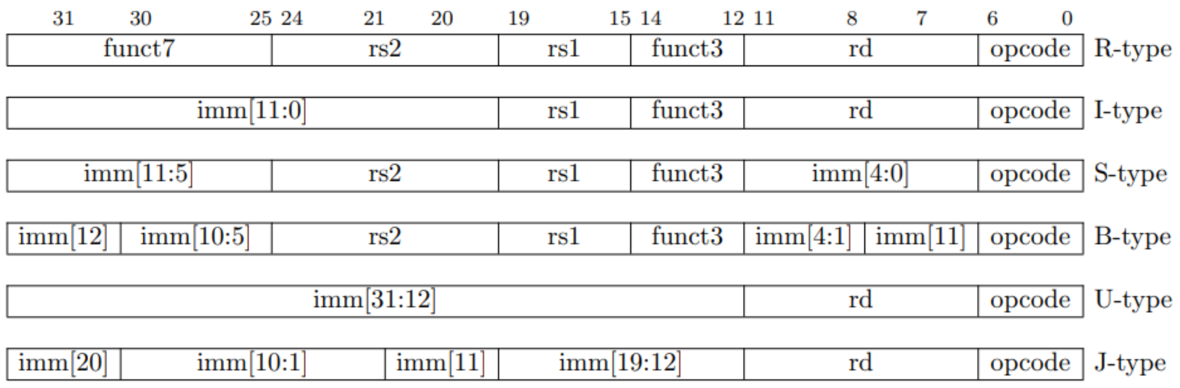

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode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

Figure 5.4: 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 5.1.2.
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 5.2) 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 5.1.1:

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 5.2: 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 5.2 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 5.1.2.

5.1.2 Implementation

The implementation of the Control Unit is split up into multiple sub-implementations. As shown in figure 5.2 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 5.1.1 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 5.3: Decoding instruction clusters

To determine the cluster for each instruction, a decoding procedure is implemented in VHDL based on structure visualized in figure 5.5:

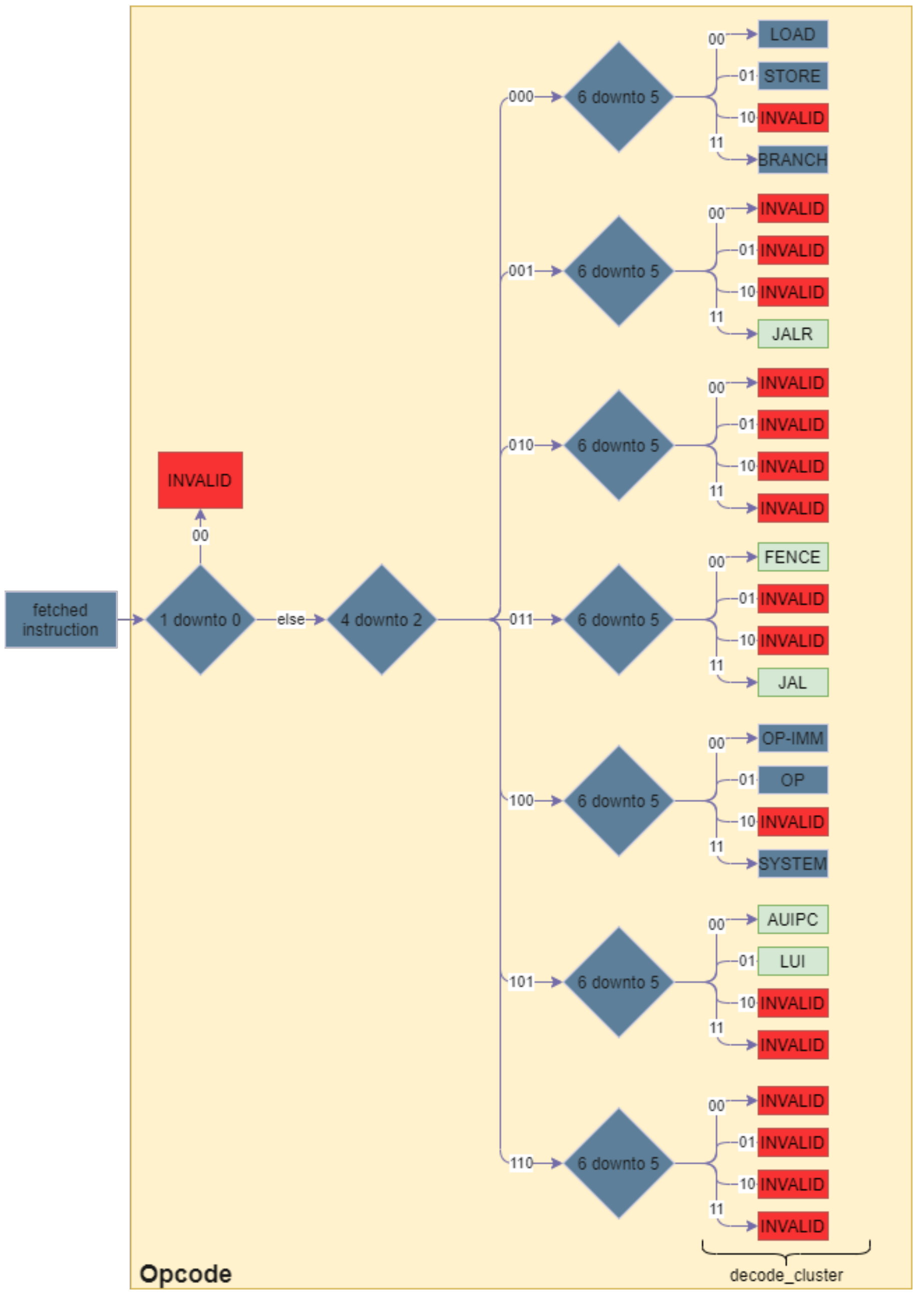
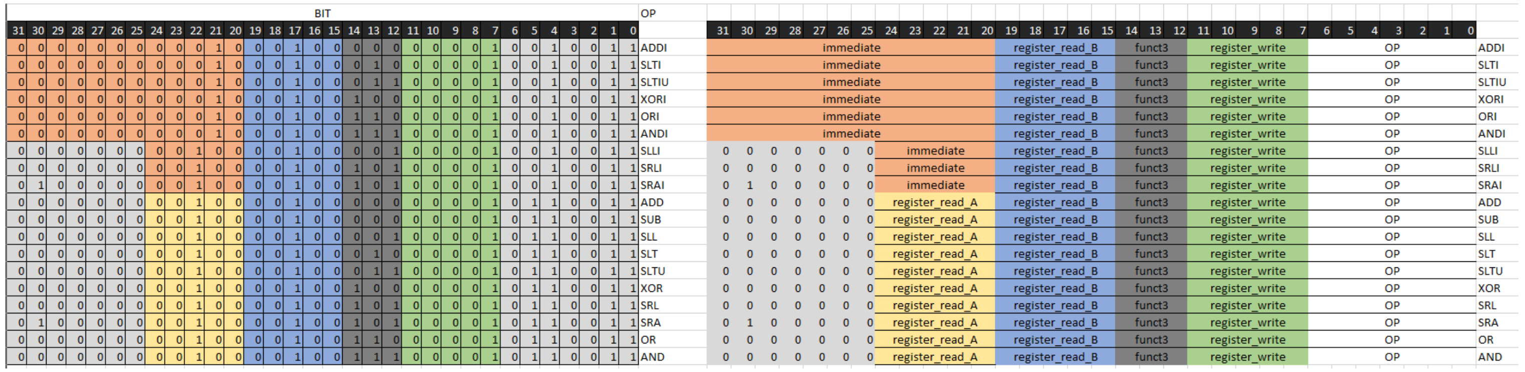


Figure 5.5: Decoding Structure to determine instruction cluster

After determining the cluster, the VHDL code assigns all the outputs visible in figure 5.2 with the respective information. For a better understanding of the information extraction, figure 5.6 shows how the 32-bit instruction word is split up (in this case for the *OP* and *OP-IMM* instructions.)



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

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

Figure 5.6: Information extraction from 32-bit instruction word

5.2 Arithmetic Logical Unit

The Arithmetic Logical Unit (ALU) is the part of the processor that performs the arithmetic and logical operations. Figure 5.7 gives an overview of what type of operations are performed.

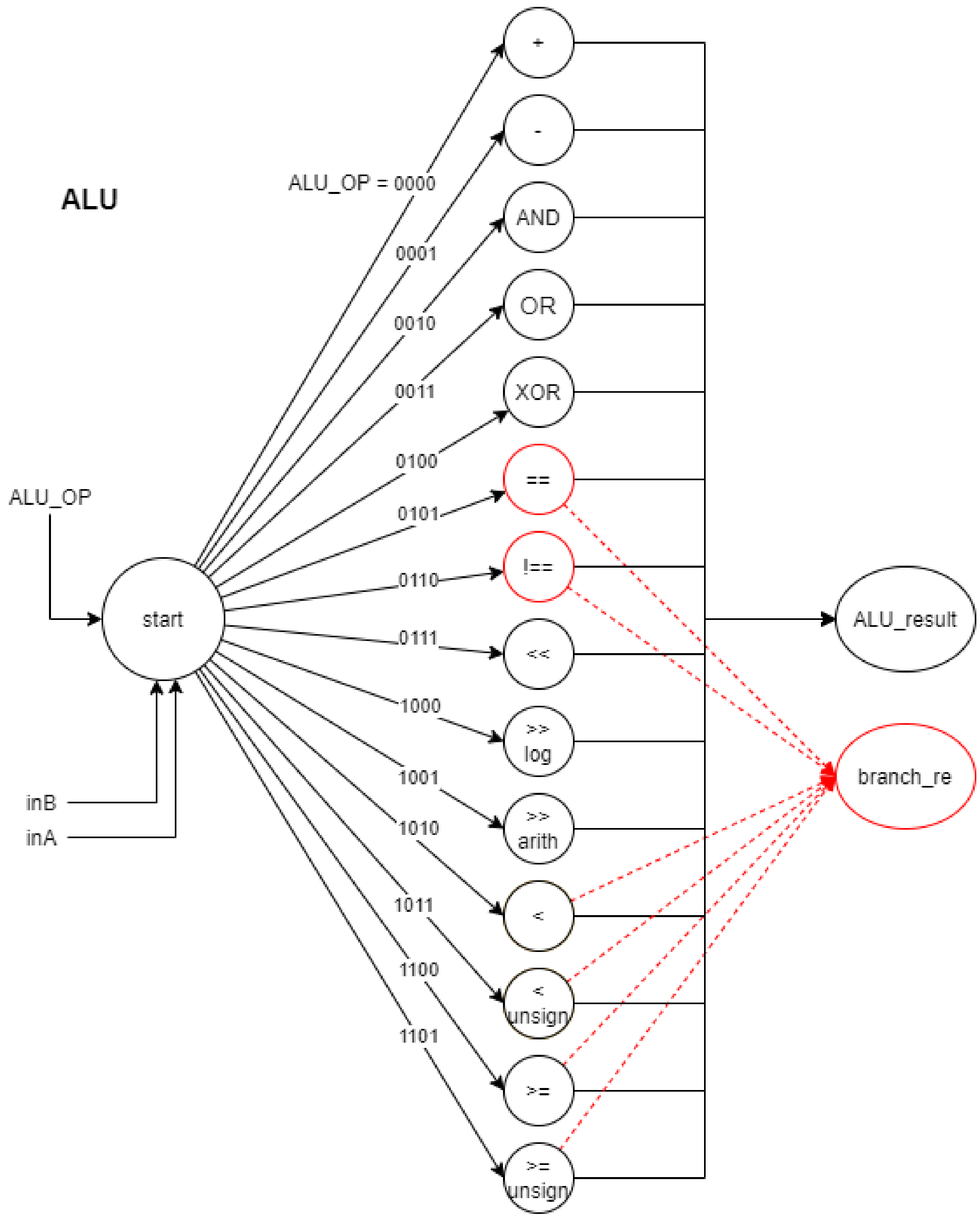


Figure 5.7: ALU operations

5.2.1 Architecture & 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 5.8:

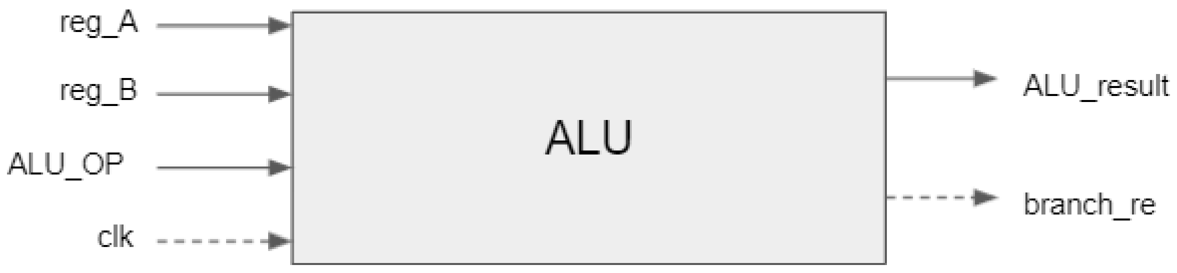


Figure 5.8: ALU operations

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.

5.2.2 Implementation

The implementation of the ALU is based on figure 5.7 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 5.4: 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*.

```vhdl
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 5.1: 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*.

5.3 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.

5.3.1 Architecture & Design

The architecture of the register files is shown in following figure 5.9

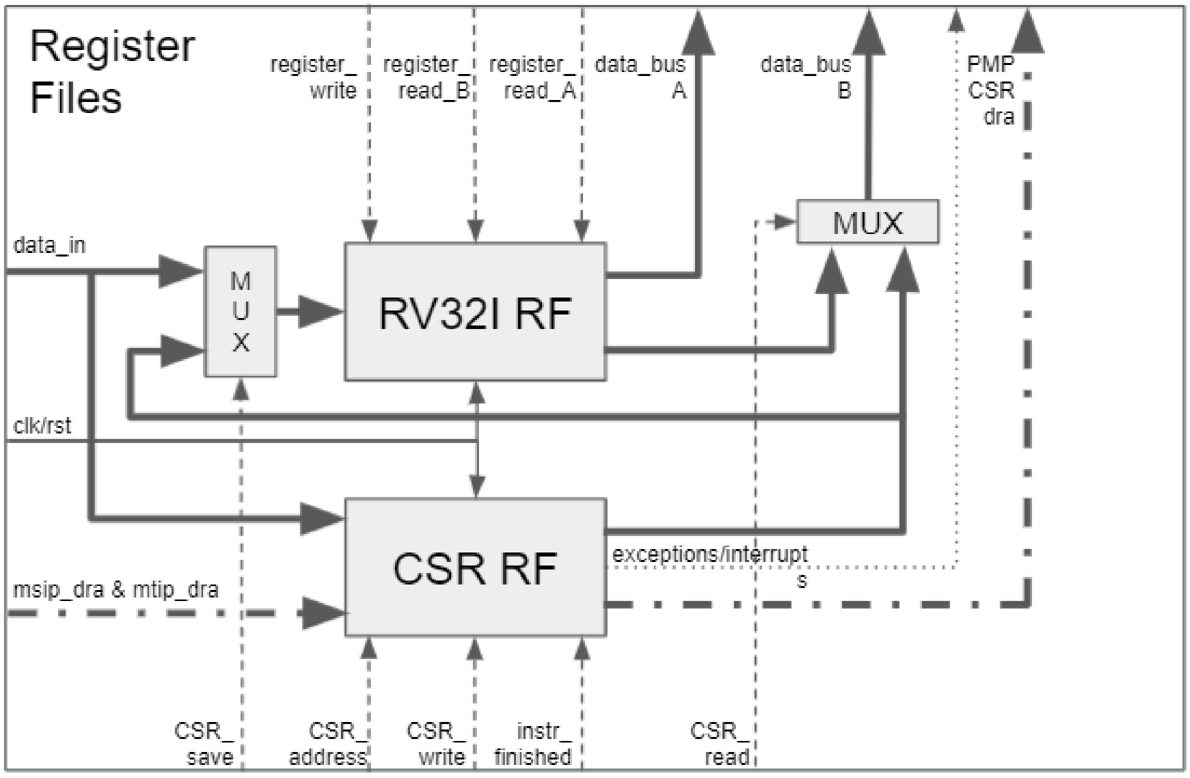


Figure 5.9: Register Files architecture

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.

5.3.2 Implementation

5.4 PMP & PMA Checker

5.4.1 Architecture & Design

5.4.2 Implementation

5.5 Exception Control

5.5.1 Architecture & Design

5.5.2 Implementation

5.6 AXI4-Lite Master

5.6.1 Architecture & Design

5.6.2 Implementation

5.7 AXI4-Lite Slave

5.7.1 Architecture & Design

5.7.2 Implementation

6 Test & Verification

6.1 Unit & Integration Verification

6.2 System Verification

6.3 Acceptance Verification

6.4 Benchmarks

7 Future Work

8 Conclusion

Bibliography

[RIS17] RISC-V. *The RISC-V Instruction Set Manual*. 2017. URL: https://riscv.org/technical/specifications.

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

Appendix