RISC-V 16-bit Processor on FPGA

Esra Mehmedova, Ilia Panayotov, Yiğit İlk, Esin Mehmedova, Paul Hagner
Department of Informatics
Technical University of Munich
Heilbronn, Germany
{esra.mehmedova, ilia.panayotov, yigit.ilk, esin.mehmedova, paul.hagner}@tum.de

1 Introduction

This project focuses on the implementation of a 16-bit RISC-V processor designed to execute basic arithmetic operations like addition and subtraction. The processor architecture integrates key components such as an Arithmetic Logic Unit (ALU), instruction memory, data memory, register file, and program counter, all working cohesively to enable efficient instruction execution. This report provides a detailed overview of the processor's design, highlighting its structure, implementation details, and testing results. The Verilog source code and the testbenches can be found on GitHub[1] for additional reference.

2 Module Description

In this section we discuss the different modules, outlining how each piece of the system works and how they implement the processor's functionality together. For each module, we provide an explanation of their purpose, design, and implementation, along with relevant code snippets.

2.1 Program Counter

The program counter (PC) is responsible for keeping track of the address of the next instruction to be executed. It ensures that instructions are fetched from the instruction memory in the correct order, maintaining the execution pipeline of the processor.

2.1.1 Design and Implementation

The PC updates its value on the rising edge of the clock or when the reset signal is triggered. When reset is active, the PC is set to 0. Otherwise, it increments by one to point to the next instruction address.

```
always @(posedge clk or posedge reset) begin
    if (reset) begin
        pc <= 10'b0;
    end else begin
        pc <= next_pc;
    end
end
```

[1] https://github.com/Imaster171/16-bit-processor

2.2 Instruction Memory

The instruction memory stores the program's machine code instructions in a sequential format. The processor fetches the instruction at the address specified by the program counter during each clock cycle for execution. The instruction memory is read-only, meaning that once the instructions are loaded into memory, they cannot be modified during execution.

2.2.1 Design and Implementation

The instruction memory is implemented using a 16-bit wide array.

```
reg [15:0] memory [0:1023];
```

The instructions are hard coded for testing and simulation purposes but can be modified or extended as needed. Each instruction is stored as a binary value, representing the machine code of the respective operation.

```
initial begin
    // Sample instructions
    memory[0] = 16'b001_011_000_0000011;
    memory[1] = 16'b100_000_011_0000000;
    memory[2] = 16'b001_001_000_0000100;
    memory[3] = 16'b000_000_001_0000_011;
    memory[4] = 16'b011_000_011_0000010;
    // ...
end
```

2.3 Register File

The register file consists of 8 general-purpose registers, each 16 bits wide, used to store intermediate values, operands, and other data required for instruction execution. These registers serve as temporary storage for the processor, providing fast access to data during computation. The register file allows simultaneous read and write operations, ensuring that the processor can retrieve and update data quickly as it executes instructions.

2.3.1 Design and Implementation

The register file includes a set of inputs and outputs that enable it to efficiently manage data storage and facilitate the necessary data access within the processor. They allow for the reading, writing, and updating of data within the registers during processor execution.

Inputs:

- Clock signal (**clk**): Synchronizes the operations of the register file
- Reset signal (**reset**): Resets all registers to zero
- Read address signals (**read_addr1, read_addr2**): Specify which registers to read from
- Write address signal (**write_addr**): Specifies the address of the register to write to
- Write data signal (**write_data**): Carries the 16-bit value to be written to the selected register
- Write enable signal (**write_enable**): Determines whether data is written to the register or not

Outputs:

- Read data signals (**read_data1, read_data2**): Output the 16-bit data stored in the registers specified by the read address signals

The register file is designed to support asynchronous read and synchronous write operations. This means that the data in any register can be read at any time, without waiting for a clock cycle, while data can only be written to the registers in sync with the clock signal. Additionally, the reset functionality ensures that all registers are initialized to zero when the reset signal is active.

```
// Asynchronous read operations
always @(*) begin
    read_data1 = reg_file[read_addr1];
    read_data2 = reg_file[read_addr2];
end

// Synchronous write operation
always @(posedge clk or posedge reset) begin
    if (reset) begin
        // Reset to 0 on reset signal
        reg_file[0] <= 16'b0;
        reg_file[1] <= 16'b0;
        reg_file[2] <= 16'b0;
        reg_file[3] <= 16'b0;
        reg_file[4] <= 16'b0;
        reg_file[5] <= 16'b0;
        reg_file[6] <= 16'b0;
        reg_file[7] <= 16'b0;
    end
    else if (write_enable) begin
        reg_file[write_addr] <= write_data;
    end
end
```

2.4 Arithmetic Logic Unit (ALU)

The ALU is responsible for performing arithmetic and logical operations. It is capable of performing basic signed arithmetic operations, such as addition and subtraction.

2.4.1 Design and Implementation

The ALU operates on two signed 16-bit operands, `a` and `b`. These operands are provided as inputs to the ALU, and based on the opcode, the ALU performs one of the following operations:

```
case (opcode)
    3'b000: result = a + b; // ADD
    3'b001: result = a + b; // ADDI
    3'b010: result = a - b; // SUBI
    default: result = 16'b0;
endcase
```

ADD represents an RRR-type instruction, where both operands (`a` and `b`) are sourced from the register file, and the result of their addition is stored back in the register file.

ADDI and **SUBI** are RRI-type instructions, where one operand (`a`) is from the register file, and the other (`b`) is a signed immediate value.

2.5 Data Memory

The data memory is responsible for storing and retrieving data during program execution. It supports both read and write operations.

2.5.1 Design and Implementation

The data memory is implemented as a synchronous write and asynchronous read memory. It allows data to be written to or read from specific memory addresses.

```
// Synchronous write operation
always @(posedge clk) begin
    if (mem_write) begin
        memory[address] <= write_data;
    end
end

// Asynchronous read operation
always @(*) begin
    read_data = memory[address];
end
```

3 RISC-V Processor

After addressing the implementation of all individual modules, this section discusses how they interact with each other to form a functional processor. It coordinates all the components to fetch, decode, execute, and store results, ensuring seamless data flow and operation.

3.1 Integration Logic

The integration logic of the processor is responsible for coordinating the flow of data between the different modules, ensuring that each component communicates effectively and that operations occur in the correct sequence. These modules are interconnected using a set of wires and control signals.

Instructions are fetched from the instruction memory using the program counter, decoded to determine the operation to be performed, and then the necessary operands are executed with the help of the ALU, register files or memory. Based on the operation type, the ALU performs operations, while the memory and register files handle read and write operations. Additionally, the integration logic handles branching by updating the program counter based on the outcome of the branch instructions. Control signals, derived from the decoded instruction, manage the enable signals for writing to the register file and memory. The structure of the processor is shown in Figure 1.

3.2 Instruction Decoding

The processor's core functionality is determined by its ability to decode instructions and execute operations based on their types. This section discusses the extraction of instruction fields and the decoding logic for determining control signals and data paths.

3.2.1 Instruction Fields Extraction

The decoding stage extracts fields from the 16-bit instruction fetched from memory. These fields include the operation code (`opcode`), register addresses, and immediate values.

```
assign opcode = instruction[15:13];
assign reg_a = instruction[12:10];
assign reg_b = instruction[9:7];
assign reg_c = instruction[2:0];
assign imm = instruction[6:0];
assign imm10 = instruction[9:0];
```

The `opcode` determines the type of operation, while `reg_a`, `reg_b`, and `reg_c` specify the source and destination registers. Immediate values (`imm` and `imm10`) are used in certain instructions like **LUI**, **SW**, **BEQ**, and arithmetic operations.

3.2.2 Register-related Assignments

A control signal is used to determine whether an instruction updates a register. This signal is disabled for **BEQ** (branch) and **SW** (store) instructions, as they do not require writing back to the register file.

```
assign write_enable = (opcode != 3'b110 &&
    opcode != 3'b011);
```

The result to be written back to the register file depends on the instruction type. Key instructions include:

- **JALR (Jump and Link Register)**: Stores the current program counter (`pc`) in the destination register.
- **LUI (Load Upper Immediate)**: Loads a 10-bit immediate value into the upper bits of a register.
- **LW (Load Word)**: Loads a value from memory into a register.
- Arithmetic instructions (**ADD**, **ADDI** and **SUBI**): Write the ALU result back to the register file.

```
assign write_data =
        (opcode == 3'b100) ? pc :
        (opcode == 3'b101) ? {imm10, 6'b0} :
        (opcode == 3'b111) ? mem_read_data :
        alu_result;
```

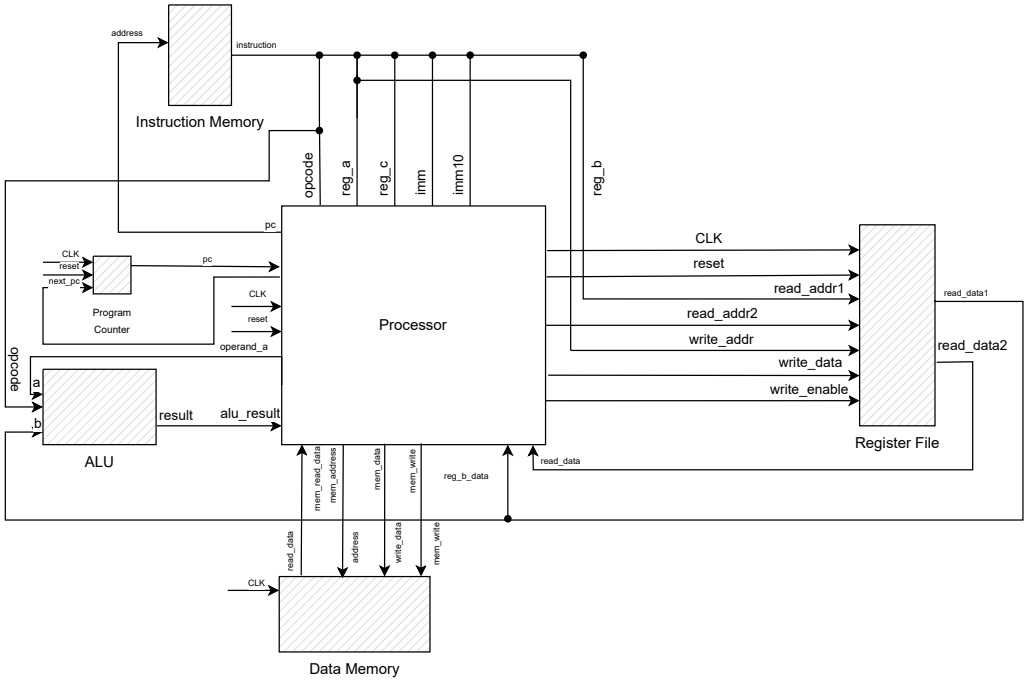

Fig. 1: Processor Structure

The register file reads two addresses, one of which is always `reg_b`, and the other is either either `reg_a` (for **BEQ** and **SW**) or `reg_c` (for other instructions):

```
.read_addr1(reg_b),
.read_addr2((opcode == 3'b011 || opcode ==
    3'b110) ? reg_a : reg_c),
```

The data corresponding to each address is assigned as follows:

```
.read_data1(reg_b_data),
.read_data2(read_data)
```

For `read_data` the instruction determines from which register the data should be read. `reg_a_data` is used for **BEQ** and **SW** instructions, providing the necessary data for comparisons or memory writes. `reg_c_data` is used for all other instructions, ensuring the correct operand is available for operations:

```
assign reg_a_data = (opcode == 3'b110 ||
    opcode == 3'b011) ? read_data : 16'bz;
assign reg_c_data = (opcode != 3'b110 &&
    opcode != 3'b011) ? read_data : 16'bz;
```

3.2.3 Memory-related Assignments

Memory operations involve storing or retrieving data:

- **SW (Store Word)**: Writes a value from a register to a memory address.
- **LW (Load Word)**: Loads a value from memory into a register.

The memory address is computed by adding the base register value (`reg_b_data`) and the sign-extended immediate value (`imm`). The data to be written to memory during a store operation (**SW**) comes from `reg_a_data`.

```
assign mem_write = (opcode == 3'b110); // SW
assign mem_read = (opcode == 3'b111);  // LW
assign mem_address = reg_b_data +
    {{9{imm[6]}}, imm};
assign mem_data = reg_a_data;
```

3.2.4 Operand B Selection

The ALU's second input (`b`) uses the sign-extended immediate value for immediate-based arithmetic instructions (**ADDI** and **SUBI**) or `reg_c_data` for **ADD**:

```
.b((opcode == 3'b001 || opcode == 3'b010) ?
    {{9{imm[6]}}, imm} : reg_c_data)
```

3.2.5 Branching Logic

The processor supports conditional and unconditional branching. The instructions that handle these operations are:

- **BEQ (Branch if Equal)**: Compares two register values and branches to a target address if they are equal.
- **JALR (Jump and Link Register)**: Updates the program counter to the value in a specified register.

```
case (opcode)
    3'b100: next_pc = reg_b_data; // JALR
    3'b011: next_pc = (reg_b_data ==
        reg_a_data) ? pc + {{3{imm[6]}}, imm}
        : pc + 1; // BEQ
    default:
        next_pc = pc + 1;
endcase
```

For **BEQ**, the next program counter value is conditionally updated by adding the sign-extended immediate value (`imm`) to the current program counter. For other instructions, the program counter increments sequentially.

4 Vivado Implementation and Integration with the PYNQ Z2 Board

We used AMD Vivado to develop and integrate the hardware design on the PYNQ Z2 FPGA board. The board has a dual-core ARM processor running Ubuntu 22.04 and provides a Jupyter Python interface.

4.1 Design Overview

The main module in our Vivado design is the `dual_port_axi_instr_mem` module. This module serves both as a memory block and an AXI4 interface. It connects the RISC-V processor and the ARM processor's Python environment. Overall the module is responsible for:

- Managing AXI4 read and write transactions.
- Storing instructions and data in a 32-bit wide memory array with 256 addresses (totalling 4K of storage).
- Allowing communication between the Python interface and the RISC-V processor.

Looking into the module, here, the RISC-V processor is instantiated and directly connected to the AXI interface. This setup makes the transfer of instructions from the Python environment into the processor possible and ensures that the register outputs can be retrieved for verification. The following section shows an example of the processor instantiation:

```
processor riskvProcessor (
    .clk(S_AXI_ACLK),
    .reset(S_AXI_ARESETN),
    .pc(pc),
    .instruction2(instruction),
    .reg_b_data(reg_b),
    .reg_c_data(reg_c),
    .reg_a_data(reg_a),
    .python_instruction(python_instruction));
```

4.2 Memory Mapping and AXI Transactions

Since the main goal of our prototype is to enable communication with the FPGA through Python, the design focuses mostly on mapping memory and ensuring a correct AXI read/write functionality. The internal memory array in `dual_port_axi_instr_mem` is used to send instructions to the processor as well as to capture the processor's register outputs. In the following, you can see how our design allocates the specific memory addresses:

- **Address 0:**
  - `mem[0][15:0]` is used to load an instruction from Python into the processor.
  - `mem[0][31:16]` is used to output the value of register `reg_c` for monitoring.
- **Address 2:**
  - `mem[2][31:16]` outputs the value of register `reg_a`.
  - `mem[2][15:0]` outputs the value of register `reg_b`.

The AXI protocol is implemented by using multiple `always` blocks to define the sequential logic. For example, during a write transaction, the module writes data to the memory array and simultaneously updates the processor's inputs and outputs, as shown in the snippet below:

```
always @(posedge S_AXI_ACLK)

begin
    if (S_AXI_WVALID && axi_wready)
    begin

        // Write data based on write strobe
            signals
        if (S_AXI_WSTRB[0]) mem[S_AXI_AWADDR
            ][7:0]   <= S_AXI_WDATA[7:0];
        if (S_AXI_WSTRB[1]) mem[S_AXI_AWADDR
            ][15:8]  <= S_AXI_WDATA[15:8];
        if (S_AXI_WSTRB[2]) mem[S_AXI_AWADDR
            ][23:16] <= S_AXI_WDATA[23:16];
        if (S_AXI_WSTRB[3]) mem[S_AXI_AWADDR
            ][31:24] <= S_AXI_WDATA[31:24];

        // Transfer instruction from memory to
             the processor
        python_instruction <= mem[0][15:0];

        // Write processor registers back to
            memory for Python verification
        mem[0][31:16] <= reg_c;
        mem[2][31:16] <= reg_a;
        mem[2][15:0] <= reg_b;
    end
end
```

4.3 Interfacing via Python

A key part of our design is the Python-based interface. Through the web interface, users can upload the `.bit` and `.hwh` files created with the AMD Vivado software. This allows to run the design on the FPGA and gives the user access to memory addresses via a Python interface. After that, users can load the design into the PYNQ Overlay and run the `overlay` command to view all available IP packages and memory. Although the AXI IP appears as a memory block in the overlay, it can be accessed via Python using the following commands:

```
axi_addr = ol.ip_dict['dual_port_axi_instr_0
    ']['phys_addr']
axi_range = ol.ip_dict['dual_port_axi_instr_0
    ']['addr_range']
from pynq import MMIO
mmio = MMIO(axi_addr, axi_range)
```

Using this interface, a user can write instructions for the 16 bits of address 0 and read back the register outputs. Unfortunately, due to limitations in the board and the bitfile generation, only 64 bits are really accessible at this moment. However, a theoretically complete implementation would be able to access all 4K of the memory via the Python interface.

In the project repository on GitHub, we included a Jupyter file[2] to show how to interact with the design.

4.4 Utilization of our Verilog code

AMD Vivado allows to view the Utilization of the logic Units inside of it and also the power consumption of them. Hence, we were able to observe that just having the AXI interface and our processor implementation consumes 0.03W which is 2% of the overal power usage of 1.311W by the whole system as can be seen on the images at Github Repository[3]. It also uses very few logic Units showing that RISC-V systems are quite efficient and so is our setup.

4.5 Summary

In short, our Vivado implementation:

- Integrates a custom dual-port AXI memory module that links the RISC-V processor with an ARM-based Python environment.
- Implements standard AXI4 protocol logic to manage memory read/write operations.
- And provides a communication interface between the Python environment and the RISC-V processor on the FPGA.

Generally speaking, the Vivado implementation builds the hardware foundation of our project. Creating a basis for transferring instructions in real-time and enabling processor state monitoring through the PYNQ interface.

[2] https://github.com/Imaster171/16-bit-processor/tree/main/vivadoExplanation#general-explanation

[3] https://github.com/Imaster171/16-bit-processor/blob/main/vivadoExplanation/README.md#utilization-of-logic-units

5 Testing and Debugging

In this section, we will go through the testbenches of each component.

5.1 Testbench Overview

The testbenches were developed to test and verify each module. The primary focus was to validate operations such as read/write operations and arithmetic processing to ensure that it is consistent with the design specifications.

5.2 Component-wise Testing

5.2.1 Program Counter (PC)

In the program counter testbench, we test sequential incrementing, reset functionality, and behaviour for different inputs.

```
initial begin
    // Initialize signals
    clk = 0;
    reset = 1;
    next_pc = 10'b0;

    // Display initial values
    $display("Time = %0t, clk = %b, reset = %b
        , next_pc = %b, pc = %b", $time, clk,
        reset, next_pc, pc);
    #10;

    // Deassert reset
    reset = 0;
    $display("Time = %0t, clk = %b, reset = %b
        , next_pc = %b, pc = %b", $time, clk,
        reset, next_pc, pc);
    #10;

    // Test case 1: Load a new PC value
    next_pc = 10'b1010;
    $display("Time = %0t, clk = %b, reset = %b
        , next_pc = %b, pc = %b", $time, clk,
        reset, next_pc, pc);
    #10;
    /*
    Additional test cases (omitted for
    brevity):
    - Load another PC value (next_pc =
        10'b1111)
    - Reset the PC (reset asserted &
        deasserted)
     */
    $finish;
end
```

TABLE I: Program Counter Output

Time	clk	reset	next_pc	pc
0	0	1	0000000000	xxxxxxxxxx
10000	1	0	0000000000	0000000000
20000	1	0	0000001010	0000000000
30000	1	0	0000001111	0000001010
40000	1	1	0000001111	0000001111

5.2.2 Instruction Memory

Our goal for the instruction memory testbench is to evaluate the retrieval of stored instructions from memory by iterating through specific addresses and verifying expected outputs.

```
initial begin
    // Test each memory location
    $display("Address: %d, Expected
        Instruction: %b, Read Instruction: %b"
        , address, 16'b001_011_000_0000011,
        instruction);
    /*
    Additional tests (omitted for brevity):
    - Address 1: Expected Instruction =
        16'b100_000_011_0000000
    - Address 2: Expected Instruction =
        16'b001_001_000_0000100
    - Address 3: Expected Instruction =
        16'b000_000_001_0000_011
    */
    $finish;
end
```

TABLE II: Instruction Memory Output

Addr	Expected Instr	Read Instr
0	0010110000000011	0010110000000011
1	1000000110000000	1000000110000000
2	001010000001100	001010000001100
3	0000000001000011	0000000001000011

5.2.3 Register File

The register file testbench verifies read and write operations for the different registers and checks whether they follow the correct behaviour during reset and overwriting scenarios.

```
initial begin

    // Write to register 3 and register 5
    write_addr = 3;
    write_data = 16'hA5A5;
    write_enable = 1;
    #10 write_enable = 0;
    write_addr = 5;
    write_data = 16'h5A5A;
    write_enable = 1;
    #10 write_enable = 0;

    // Read from registers 3 and 5
    read_addr1 = 3;
    read_addr2 = 5;
    #10;
    $display("Register: %d, Read Data: %h", 3,
         read_data1);
    $display("Register: %d, Read Data: %h", 5,
         read_data2);
    S
    // Additional tests (reset verification,
        overwrite) omitted for brevity
    $finish;
end
```

TABLE III: Register File Output

Test	Reg	Read Data
Write and read	3	a5a5
Write and read	5	5a5a
After reset	3	0000
After reset	5	0000
Overwrite and read	3	5678

5.2.4 Arithmetic Logic Unit (ALU)

The ALU testbench verifies the correct execution of arithmetic operations. Testing addition, subtraction, and the handling of positive and negative values. The testbench also covers edge cases such as zero detection and default behaviour.

```
initial begin
    // Test 1: ADD operation
    a = 16'sd10;
    b = 16'sd20;
    opcode = 3'b000;
    #10;
    $display("Test 1 - ADD: a = %d, b = %d,
        result = %d, zero = %b", a, b, result,
         zero);
    /*
    Additional tests (omitted for brevity):
    - ADDI with negative values
    - SUBI with positive and negative values
    - Default Case
    - Zero-result cases
    */
    $finish;
end
```

TABLE IV: ALU Output

Test	Op	a	b	Res	Z
1	ADD	10	20	30	0
2	ADDI neg	-15	5	-10	0
3	SUBI	30	50	-20	0
4	SUBI neg	30	-50	80	0
5	Default	10	-10	0	1
6	Zero Res	-25	-25	0	1

5.2.5 Data Memory

The data memory testbench verifies correct read and write operations at different addresses and test boundary conditions. This way, we want to ensure data integrity and memory consistency.

```
initial begin
    // Test 1: Write and read from address 0
    address = 16'd0;
    write_data = 16'hA5A5;
    mem_write = 1;
    #10;
    mem_write = 0;
    #10;
    $display("Address: %d, Expected Data: %h,
        Read Data: %h", address, 16'hA5A5,
        read_data);
    /*
    Additional tests (omitted for brevity):
    - Test write/read from address 1023
    - Test read from uninitialized address (e.
        g., address 200)
    */
    $finish;
end
```

TABLE V: Data Memory Output

Address	Expected Data	Read Data
0	a5a5	a5a5
1023	1234	1234
200	xxxx	xxxx

5.3 Summary

The testing process confirmed that all system components functioned as expected. The main challenges encountered were related to working with Vivado. Furthermore, there were difficulties understanding certain commands, which required extensive research to fully grasp their functionality. To make sure everything worked, we tested each component thoroughly. This includes testing the functionality with negative numbers to realize proper handling of signed values. In the end, these efforts paid off, creating a system that performed reliably.

6 Conclusion

In this project, we designed and implemented a functional 16-bit RISC-V processor on an FPGA. Generally, we focused on basic arithmetic operations. The key components were successfully integrated and functioned as intended. We confirmed that our design works as expected by evaluating and testing each module. To this end, we assessed the component's behavior individually and considered multiple test cases, including its functionality with negative numbers.

Overall, this project was a valuable experience. It challenged us to apply the concepts we learned in the HDL course while also encouraging us to think creatively to simplify the complexities we encountered. First of all, designing and testing modules individually made it easier to identify errors and maintain the code, demonstrating the advantages of a structured, modular approach. Secondly, to meet the project objectives, we needed to revisit the concepts introduced in the HDL course and conduct additional research to fully understand the functionality of various commands. Lastly, learning how to use Vivado was very critical, as this tool turned out to be quite challenging aside from working with the given limitations.

In conclusion, we successfully implemented a functional 16-bit RISC-V processor, verified each module's performance and revisited essential lessons from the HDL course.

7 Contributions

Ilia Panayotov
Scope of Work: Verilog implementation (SW and LW instructions), diagram design, and presentation preparation.

Esra Mehmedova
Scope of Work: Verilog implementation (ADD, ADDI, SUBI, BEQ, JALR, and LUI instructions) and reviewing the paper.

Esin Mehmedova
Scope of Work: Writing the paper (Sections 1, 2, and 3).

Paul Hagner
Scope of Work: Writing the paper (Sections 4, 5, and 6).

Yiğit İlk
Scope of Work: Vivado implementation.

References

[1] https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html

[2] https://www.alpharithms.com/mips-store-word-sw-vs-load-word-lw-475521/

[3] https://ic.unicamp.br/~edson/disciplinas/mc404/material-riscv/extra/RISC-V-refcard.pdf

[4] GitHub Copilot was used to assist in the Verilog implementation. https://github.com/features/copilot