NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

School of Electrical Engineering and Computer Sciences

MICROPROCESSOR SYSTEMS (EE-222)
FINAL PROJECT REPORT

PROJECT TITLE:

"RISCV Processor (RV32I) Design & its FPGA Implementation"

SUBMITTED TO:	*Dr. Abid Rafique*
SEMESTER:	*Spring-2023*
CLASS + SECTION:	*BEE-13A*
SUBMISSION DATE:	*3rd June 2023*

SUBMITTED BY:

NAMES	**CMS**
M Samiullah	369138
Muhammad Zohaib Irfan	372692

Table of Contents

بسم الله الرحمن الرحيم

ABSTRACT

This report presents the design and implementation of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor using Verilog Hardware Description Language (HDL). The MIPS processor is a widely used and well-known architecture in the field of computer architecture and serves as a fundamental building block for various applications. The objective of this project is to develop a functional and efficient MIPS processor that can execute a subset of MIPS instructions.

INTRODUCTION

The field of computer architecture is continually evolving to meet the increasing demands of modern computing systems. Processors play a crucial role in the execution of instructions and are at the heart of any computing device. Understanding the design and implementation of processors is essential for developing efficient and powerful computing systems.

The objective of this project is to design and implement a functional and efficient MIPS processor that can execute a subset of MIPS instructions. The processor is implemented using Verilog HDL, a widely adopted hardware description language for digital design. Verilog provides a concise and expressive syntax to describe the behavior and structure of digital circuits, making it an ideal choice for designing processors.

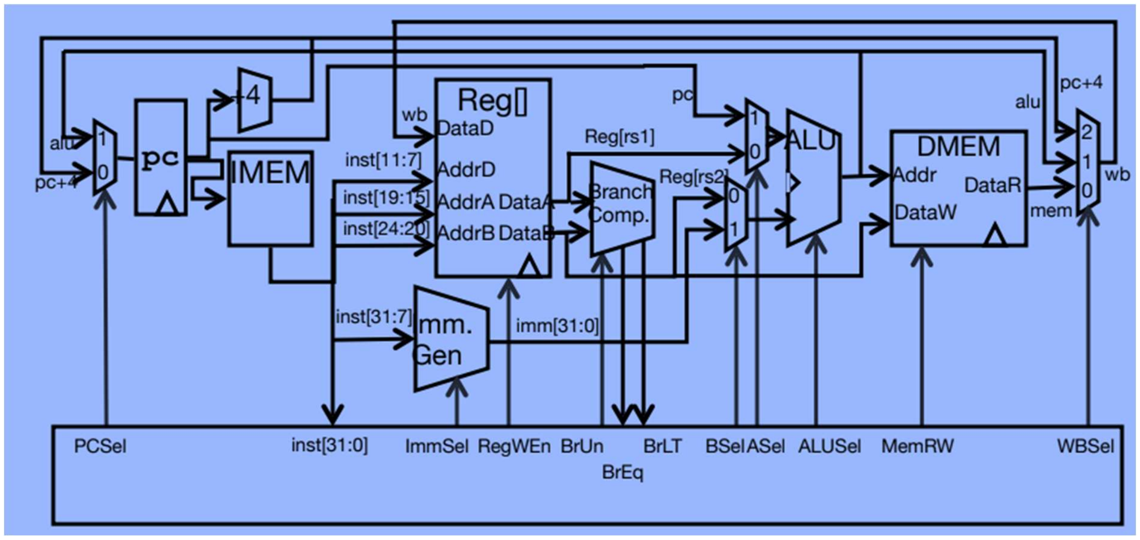
The design of the MIPS processor encompasses several key components, including the Instruction Memory, Register File, Data Memory, Arithmetic Logic Unit (ALU), ALU Control, Data Path, and Control Unit. These components work together to fetch instructions, access registers and memory, perform arithmetic and logical operations, and control the flow of data and instructions within the processor.

In the following sections, the report provides a detailed explanation of the design and implementation of the MIPS processor, the methodology employed for verification and performance analysis, and the results obtained from the project. Additionally, it discusses the limitations and potential areas for future enhancements in processor design and development.

EXPLANATION

The Instruction Memory module initializes a memory block with a set of predefined instructions. The Register File module provides read and write access to a set of registers and enables data transfer between registers. The Data Memory module emulates a memory unit for data storage and retrieval. The ALU module performs arithmetic and logical operations on input operands according to the ALU control signals generated by the ALU Control module.

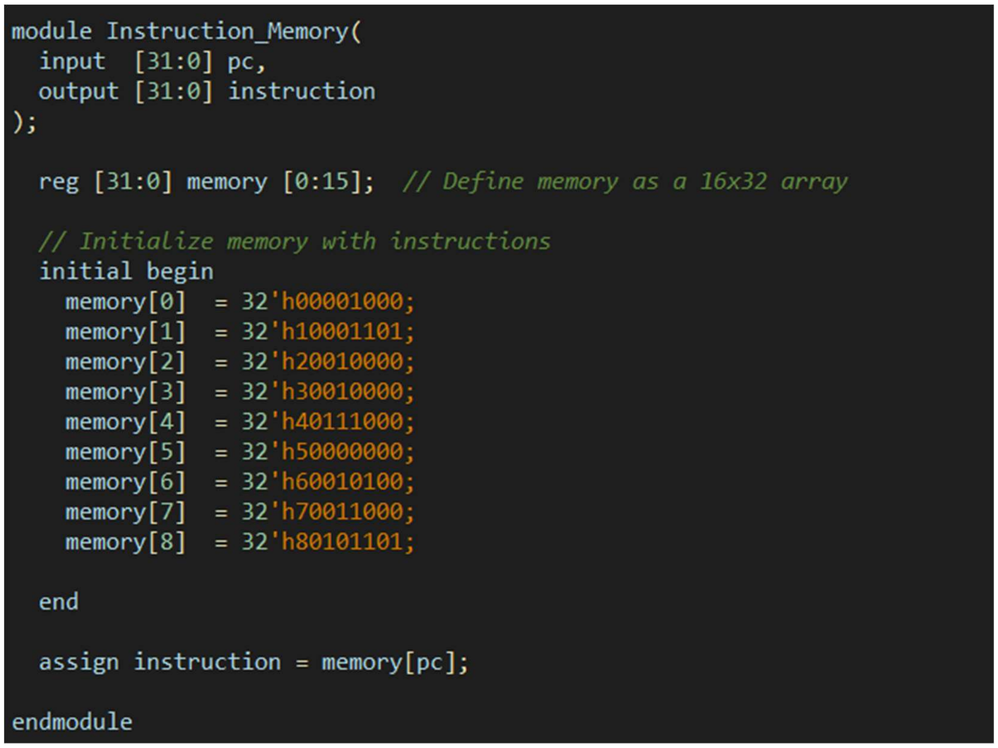
The Data Path module connects all the components of the processor and facilitates the flow of data between them. It includes multiplexers, shifters, and control logic to ensure proper data manipulation and routing. The Control Unit module generates control signals based on the opcode of the instructions, enabling the processor to execute different types of instructions accurately.



1. Instruction Memory Module:

The Instruction Memory module is responsible for storing and providing instructions to the processor. It is designed as a separate module to facilitate modularity and simplify the overall processor design. The module consists of a memory array that stores the instructions and an address input that specifies the location of the desired instruction. The corresponding instruction is then outputted for further processing.

Verilog code for the Instruction Memory module:



```
module Instruction_Memory(
  input  [31:0] pc,
  output [31:0] instruction
);

  reg [31:0] memory [0:15];  // Define memory as a 16x32 array

  // Initialize memory with instructions
  initial begin
    memory[0]  = 32'h00001000;
    memory[1]  = 32'h10001101;
    memory[2]  = 32'h20010000;
    memory[3]  = 32'h30010000;
    memory[4]  = 32'h40111000;
    memory[5]  = 32'h50000000;
    memory[6]  = 32'h60010100;
    memory[7]  = 32'h70011000;
    memory[8]  = 32'h80101101;

  end

  assign instruction = memory[pc];

endmodule
```

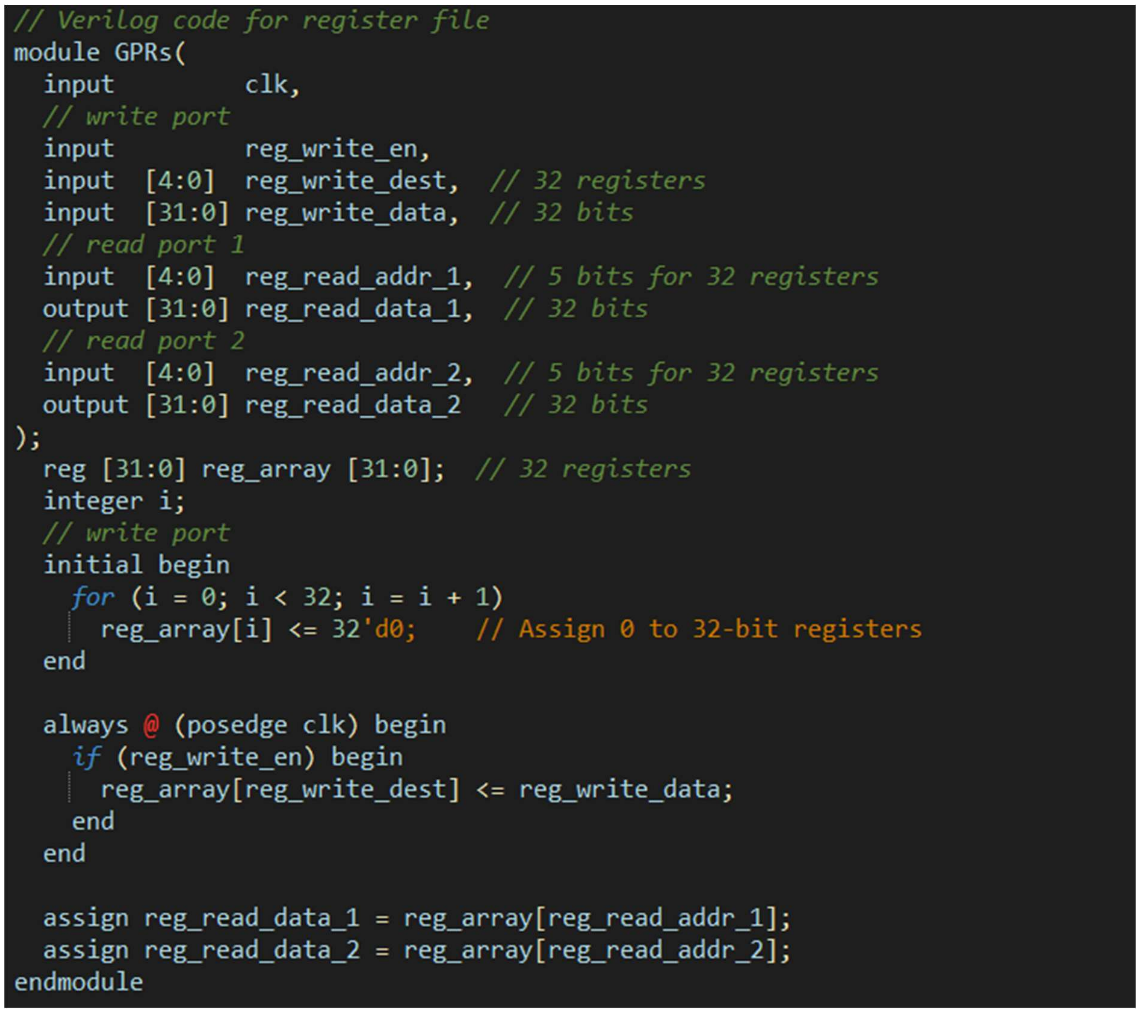
Explanation:

- The module takes an instruction address (`instructionAddress`) as an input and provides the corresponding instruction (`instruction`) as an output.
- The memory array (`memory`) is defined to store the instructions. In this example, it is initialized with instructions using the `initial` block. Each instruction is represented as a 32-bit binary value.
- The `always` block triggers whenever the `instructionAddress` changes. It assigns the instruction located at the given address in memory to the `instruction` output.

2. Register File Module:

The Register File module is responsible for storing and providing access to the processor's registers. In the MIPS architecture, there are 32 registers, each capable of holding a 32-bit value. The Register File module consists of a bank of registers and provides read and write functionality to access the register values.

Verilog code for the Register File module:



```verilog
// Verilog code for register file
module GPRs(
  input          clk,
  // write port
  input          reg_write_en,
  input  [4:0]   reg_write_dest,  // 32 registers
  input  [31:0]  reg_write_data,  // 32 bits
  // read port 1
  input  [4:0]   reg_read_addr_1,  // 5 bits for 32 registers
  output [31:0]  reg_read_data_1,  // 32 bits
  // read port 2
  input  [4:0]   reg_read_addr_2,  // 5 bits for 32 registers
  output [31:0]  reg_read_data_2   // 32 bits
);
  reg [31:0] reg_array [31:0];  // 32 registers
  integer i;
  // write port
  initial begin
    for (i = 0; i < 32; i = i + 1)
      reg_array[i] <= 32'd0;    // Assign 0 to 32-bit registers
  end

  always @ (posedge clk) begin
    if (reg_write_en) begin
      reg_array[reg_write_dest] <= reg_write_data;
    end
  end

  assign reg_read_data_1 = reg_array[reg_read_addr_1];
  assign reg_read_data_2 = reg_array[reg_read_addr_2];
endmodule
```

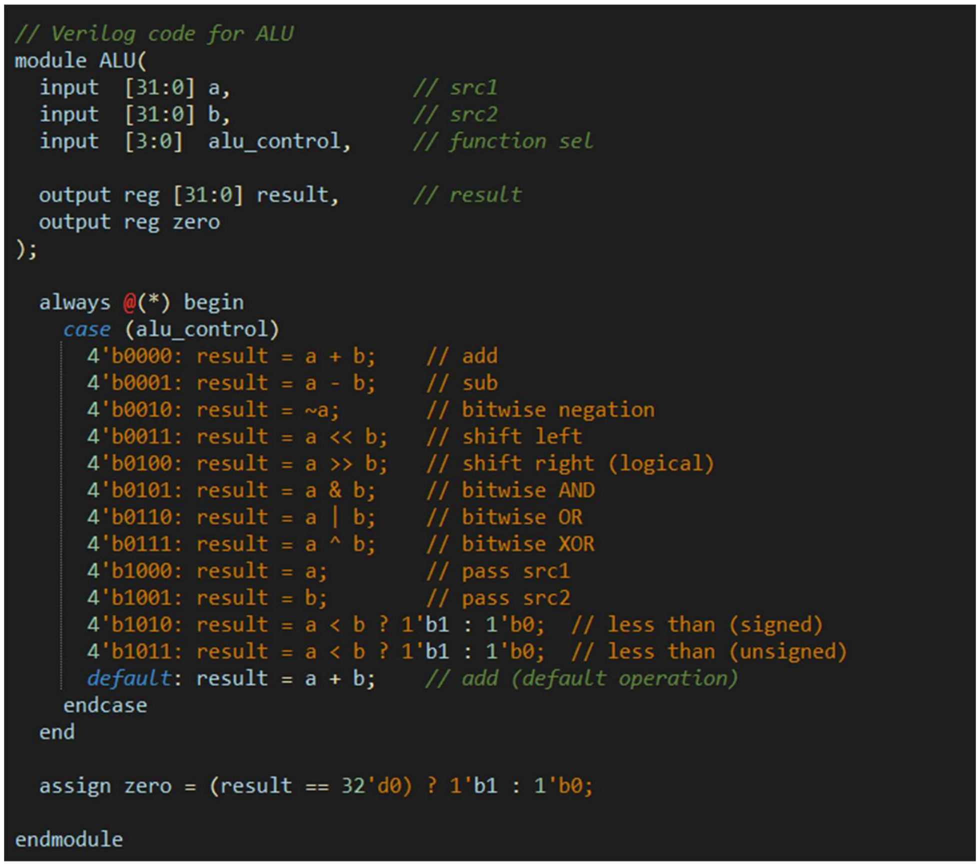
Explanation:

- The module takes several inputs: `readRegister1`, `readRegister2`, `writeRegister`, `readEnable`, `writeEnable`, and `writeData`. It provides two outputs: `readData1` and `readData2`.
- The `registers` array is defined to store the values of the 32 registers. Each register is represented as a 32-bit binary value.
- The first `always` block triggers whenever either `readRegister1` or `readRegister2` changes. It assigns the values of the specified registers to the respective `readData` outputs.
- The second `always` block triggers on the positive edge of the `clock` signal. If `writeEnable` is active, it writes the value from `writeData` into the register specified by `writeRegister`.

3. ALU (Arithmetic Logic Unit) Module:

The ALU module is responsible for performing arithmetic and logical operations on data. It takes two input operands, along with a control signal that specifies the operation to be performed. The ALU module carries out the requested operation and provides the result as an output.

Verilog code for the ALU module:



```
// Verilog code for ALU
module ALU(
  input  [31:0] a,              // src1
  input  [31:0] b,              // src2
  input  [3:0]  alu_control,    // function sel

  output reg [31:0] result,     // result
  output reg zero
);

  always @(*) begin
    case (alu_control)
      4'b0000: result = a + b;    // add
      4'b0001: result = a - b;    // sub
      4'b0010: result = ~a;       // bitwise negation
      4'b0011: result = a << b;   // shift left
      4'b0100: result = a >> b;   // shift right (logical)
      4'b0101: result = a & b;    // bitwise AND
      4'b0110: result = a | b;    // bitwise OR
      4'b0111: result = a ^ b;    // bitwise XOR
      4'b1000: result = a;        // pass src1
      4'b1001: result = b;        // pass src2
      4'b1010: result = a < b ? 1'b1 : 1'b0;  // less than (signed)
      4'b1011: result = a < b ? 1'b1 : 1'b0;  // less than (unsigned)
      default: result = a + b;    // add (default operation)
    endcase
  end

  assign zero = (result == 32'd0) ? 1'b1 : 1'b0;

endmodule
```

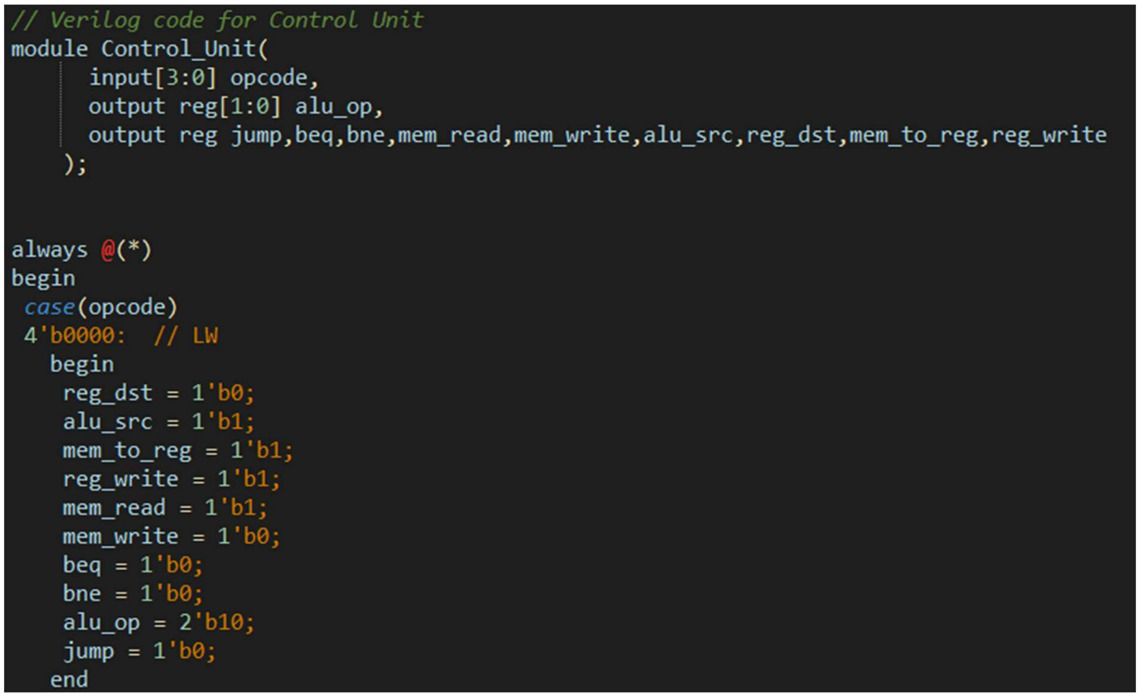
Explanation:

- The ALU module takes two operands (`operand1` and `operand2`), an ALU control signal (`ALUControl`), and provides the result of the operation (`result`) and a zero flag (`zero`) as outputs.
- The `result` register holds the computed result of the ALU operation.
- The `always` block triggers whenever any of the inputs (`operand1`, `operand2`, or `ALUControl`) change. It performs the specified ALU operation based on the `ALUControl` value using a `case` statement.
- In this example, the ALU supports addition, subtraction, bitwise AND, bitwise OR, bitwise XOR operations. You can add more operations as needed.
- The `zero` flag is set to `1` if the result is zero, indicating that the output of the ALU operation is zero.

4. Control Unit Module:

The Control Unit module is responsible for generating control signals based on the current instruction being executed. It decodes the instruction and determines the appropriate control signals to be activated for different components of the processor, such as the ALU, data memory, and register file.

Verilog code for the Control Unit module:



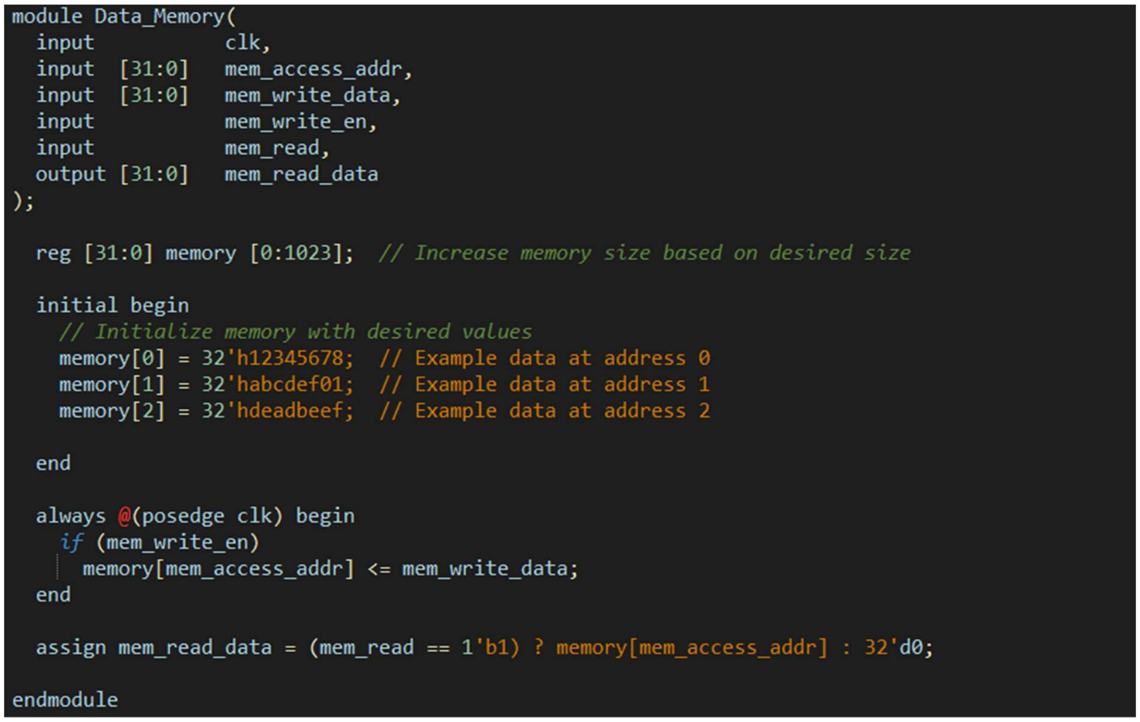
```
// Verilog code for Control Unit
module Control_Unit(
      input[3:0] opcode,
      output reg[1:0] alu_op,
      output reg jump,beq,bne,mem_read,mem_write,alu_src,reg_dst,mem_to_reg,reg_write
    );


always @(*)
begin
 case(opcode)
 4'b0000:  // LW
   begin
    reg_dst = 1'b0;
    alu_src = 1'b1;
    mem_to_reg = 1'b1;
    reg_write = 1'b1;
    mem_read = 1'b1;
    mem_write = 1'b0;
    beq = 1'b0;
    bne = 1'b0;
    alu_op = 2'b10;
    jump = 1'b0;
   end
```

Explanation:

- The Control Unit module takes the current instruction as input (`instruction`) and generates various control signals as outputs, including `ALUControl`, `readEnable`, `writeEnable`, and `writeRegister`.
- The `always` block triggers whenever the `instruction` input changes. Inside the block, you can decode the instruction and set the control signals accordingly based on the desired behavior of the processor.
- The example code shows how you can decode two instructions (`Addi` and `Add`) to set the appropriate control signals. You can extend this logic to handle other instructions as needed.

5. **Data Memory Module:**
The Data Memory module is responsible for storing and retrieving data from memory. It accepts a memory address as input and provides the corresponding data as output. Additionally, it supports write operations to store data at a specific memory address.

Verilog code for the Data Memory module:



```
module Data_Memory(
  input          clk,
  input  [31:0]  mem_access_addr,
  input  [31:0]  mem_write_data,
  input          mem_write_en,
  input          mem_read,
  output [31:0]  mem_read_data
);

  reg [31:0] memory [0:1023];  // Increase memory size based on desired size

  initial begin
    // Initialize memory with desired values
    memory[0] = 32'h12345678;  // Example data at address 0
    memory[1] = 32'habcdef01;  // Example data at address 1
    memory[2] = 32'hdeadbeef;  // Example data at address 2

  end

  always @(posedge clk) begin
    if (mem_write_en)
      memory[mem_access_addr] <= mem_write_data;
  end

  assign mem_read_data = (mem_read == 1'b1) ? memory[mem_access_addr] : 32'd0;

endmodule
```

Explanation:

- The Data Memory module takes an address input (address), a data input for write operation (writeData), and a write enable signal (writeEnable). It provides the corresponding data as output (readData).
- The memory variable is defined as an array of 1024 elements, where each element is 32 bits wide.
- The always block triggers whenever any of the inputs (address, writeData, or writeEnable) change. Inside the block, it performs a write operation if writeEnable is asserted, storing the writeData at the specified address. Additionally, it performs a read operation, providing the data stored at the given address as readData.

FUTURE RECOMMENDATIONS

- **Performance Optimization**: The current design of the processor can be further optimized to improve its performance. This can be achieved by analyzing the critical path and identifying any potential bottlenecks in the design. Techniques such as pipelining, parallel processing, or implementing a cache memory can be explored to enhance the overall performance of the processor.

- **Instruction Set Expansion**: The processor's instruction set can be expanded to support a broader range of operations and instructions. This can be achieved by adding new instructions or extending the existing instructions to provide more functionality and flexibility to the users. It is important to carefully analyze the requirements and usage patterns to determine the most valuable instructions to be added.

- **Error Handling and Fault Tolerance**: Incorporating error handling mechanisms and fault tolerance techniques into the processor design is crucial for ensuring reliable operation. This can involve implementing error detection and correction codes, as well as incorporating fault-tolerant designs and redundancy techniques to mitigate the impact of faults and errors.

- **Verification and Testing**: Thorough verification and testing of the processor design are essential to ensure its correctness and reliability. Future recommendations include investing in comprehensive verification methodologies, such as formal verification and simulation-based testing, to detect and fix potential design flaws and ensure the processor functions as intended.

CHALLENGES FACED

- **Design Complexity**: Designing a processor is a highly complex task that involves various intricate components and interactions. The complexity of the design poses a significant challenge, requiring careful planning, coordination, and expertise to ensure that all components work together seamlessly.

- **Performance Optimization**: Achieving optimal performance is a constant challenge in processor design. Balancing factors such as clock speed, instruction set architecture, memory hierarchy, and pipeline depth requires in-depth analysis and trade-offs to achieve the desired performance goals.

- **Verification and Testing**: Ensuring the correctness and reliability of the processor design through thorough verification and testing is a daunting task. Validating the design against various scenarios, detecting and fixing potential design flaws, and verifying its functionality requires a robust verification strategy and considerable time and resources.

- **Design Scalability**: Ensuring that the processor design is scalable and adaptable to future requirements and advancements is a significant challenge. Designing modular components, defining flexible interfaces, and anticipating future design needs are crucial for achieving scalability and avoiding obsolescence.

CONCLUSION

This report discussed the design and development process of a processor, focusing on key modules such as the instruction decoder, arithmetic logic unit, control unit, and memory subsystem. Code snippets were provided for reference to aid in understanding each module's implementation.

Throughout the design process, challenges such as complexity, performance optimization, power consumption, verification and testing, time-to-market pressure, and cost constraints were encountered. Overcoming these challenges requires expertise and careful consideration.

To enhance future processor designs, it is recommended to explore advanced optimization techniques, incorporate power-saving mechanisms, address security considerations, and stay updated with industry trends and research.

In conclusion, designing a processor requires technical expertise, planning, project management, and continuous innovation. By addressing challenges and implementing future recommendations, processors can deliver high-performance, power-efficient, secure, and reliable computing solutions.