32-BIT RISC PROCESSOR

A project report submitted to the

Department of Electrical and Information Engineering
Faculty of Engineering
University of Ruhuna
Sri Lanka

On 18th February 2024

In completing an assignment for the module

EE5260 - Hardware Description Language

By

Group No: 05

LAKPAHANA A.G.S.	EG/2020/4040
LELWALA L.G.S.R	EG/2020/4047
MALISHA A.P.D	EG/2020/4065

1. Abstract

The design of a RISC processor with memory controller is done in this project. For the best use of memory, this processor contains a memory module and control unit which are included in the processor design. This Processor embodies 15 basic instructions involving Arithmetic, Logical, and Data Transfer and control instructions. To implement these instructions the design incorporates various design blocks like Control Unit (CU), Arithmetic and Logic Unit (ALU), Accumulator, Program Counter (PC), Instruction Register (IR), Memory and additional logic. A new architecture is implemented for the proposed RISC processor with 32 bit input. The processor has small instruction set and control logic design is very much simplified. It is basically designed in order to achieve faster executions and the processor can execute each instruction within one clock cycle. All individual logic blocks are simulated using ModelSim Simulator and top module is obtained by connecting all the blocks in an order.

2. Contents

3. Introduction

RISC processors are increasingly prevalent across various sectors, from industrial applications and consumer electronics to tablet computers and some of the world's most advanced supercomputers, such as the K computer. The core principle behind Reduced Instruction Set Computing (RISC) is to utilize a simplified set of instructions, as opposed to a complex instruction set, thereby enhancing the processor's speed. This paper emphasizes the design of a RISC processor that is equipped with 15 essential instructions for executing arithmetic, logical, data transfer, and control operations, highlighting the efficiency gains from this streamlined approach.

Additionally, the project implements of a simplified memory controller module, which employs various control signals to manage the flow of data to and from the memory unit. Integrating the memory controller directly within the processor architecture potentially boosts the system's overall performance. The memory controller's design uniquely incorporates both RAM and ROM into a single memory unit, offering a cost-effective solution per memory unit. Furthermore, a simplified control unit, utilizing distinct control signals, regulates the data flow and orchestrates the operation of the processor's various modules. This design philosophy makes such processors highly suitable for a broad range of general-purpose applications.

4. Methodology

A newly designed RISC processor integrates a memory controller. It operates as a 32-bit processor with a straightforward instruction set consisting of 15 fundamental instructions covering Arithmetic, Logical, Data Transfer, and control operations. Key design components include the control unit (CU), Arithmetic and Logical Unit (ALU), Accumulator, Program Counter (PC), Instruction Register (IR), Memory, and additional logic blocks. The processor is engineered to optimize performance through parallel processing and pipelining, completing each instruction within a single clock cycle.

Comprising various logic blocks, the processor accepts a 32-bit Data-in input. Its functionality relies on an executable clock, a positive edge-triggered input, alongside one-bit inputs for reset and 'y', and a clock input for fetch. Data input is directed to both the instruction register and memory unit. The instruction register, storing the currently executed instruction, receives inputs from Data in, reset, and load instruction register (ldir).

Output from the instruction register includes a 4-bit opcode (the MSB bits of the 32-bit input), with the remaining 28 bits forming irout, which serves as input to the multiplexer. The Program Counter holds the address of the currently executed instruction. Incrementing its value by 1 occurs when both fetch and incpc signals are HIGH. The instruction register and Program Counter outputs are fed into the multiplexer, generating an output address for memory based on the fetch input.

Memory unit operations involve reading/writing data to/from the provided input address. Data read from memory becomes one input to the ALU, while the other input is the accumulator's output. The ALU executes various operations on these inputs, producing a 32-bit output stored in the accumulator, determined by the opcode. The control unit, streamlined in design, issues command signals to other modules based on different opcode values. When the buffer's 'y' input is HIGH, data out is sourced from the buffer.

The RISC processor design was developed using Verilog, a hardware description language (HDL), to specify and describe the behavior of digital circuits. ModelSim, a widely-used HDL simulation tool, was employed for simulating and verifying the functionality of the design, providing features for debugging, waveform viewing, and timing analysis

5. Architecture and Design

The architecture of the 32-bit RISC processor was meticulously crafted to achieve a balance between performance, simplicity, and scalability. This section provides an in-depth description of the key architectural features and design decisions.

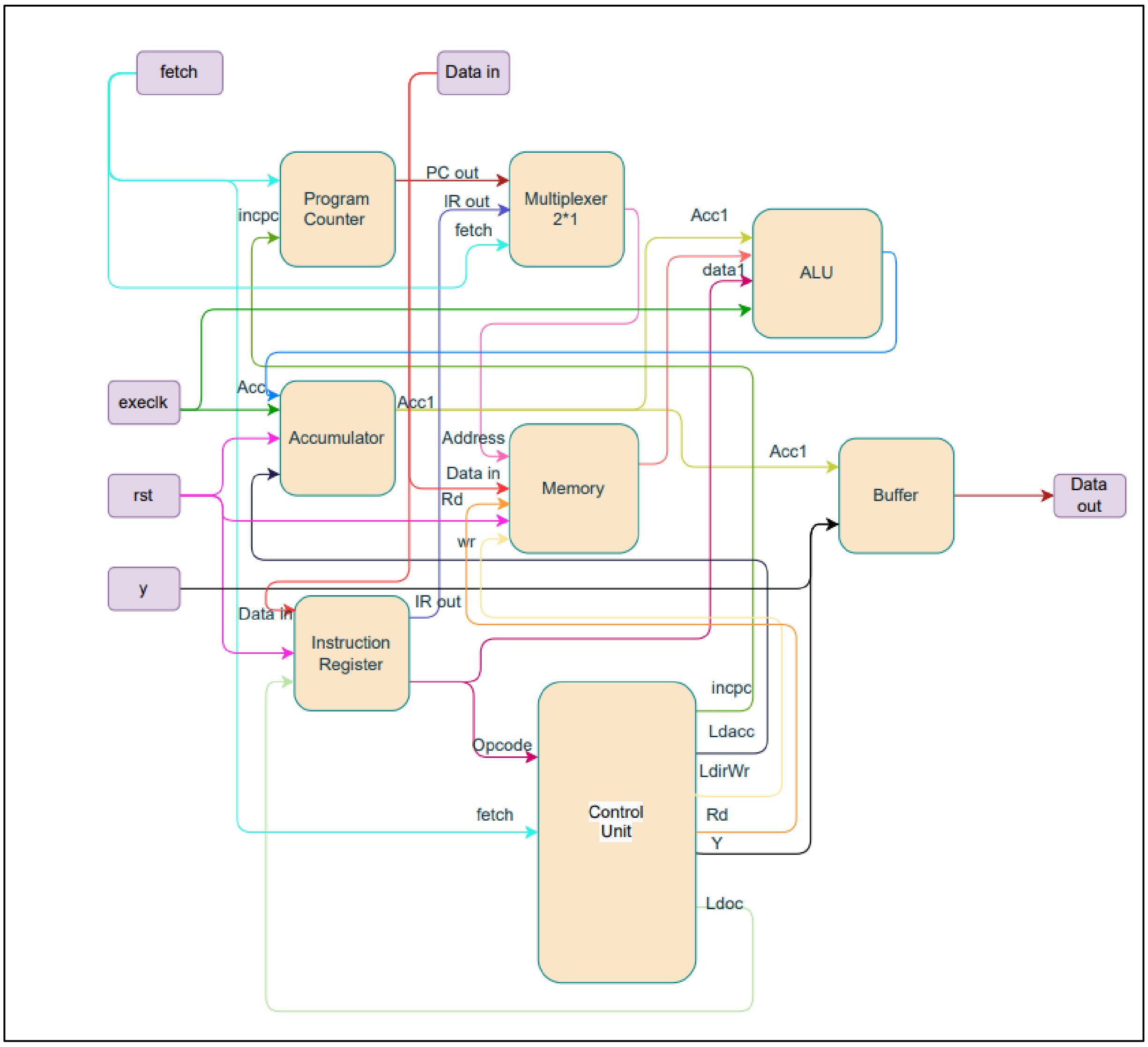


Figure 1: Architecture of the RISC Processor

- **Accumulator**

The Accumulator is a crucial component designed to store and update values as the processor executes instructions. Its operational behavior is captured in the simulation waveform, which illustrates how data is managed within the accumulator. When the reset (`rst`) signal is set to "0", the accumulator's output is reset to zero, ensuring that the processor starts from a known state. Conversely, when the load accumulator (`ldacc`) signal is activated ('1'), the accumulator outputs the stored value, showcasing its capability to dynamically respond to control signals.

- **Arithmetic and Logical Unit (ALU)**

The ALU is at the heart of the processor, capable of performing a wide array of operations, including arithmetic, logical, and data transfer tasks. It processes inputs from two 32-bit data lines (`acc` and `data`) and an opcode that dictates the operation to be performed. The presence of a clock input (`execlk`) ensures that operations are synchronized with the system clock, facilitating timely execution of instructions. Different operations are visualized in the simulation waveform for various opcode values, highlighting the ALU's versatility in handling computational tasks.

- **Instruction Register**

The Instruction Register stores the instruction currently being executed, which is pivotal for the sequential operation of the processor. It accepts instruction data and control signals (`rst` and `ldir`) to manage its operation. The simulation results show how the register outputs a 4-bit opcode and the instruction data (`irout`) based on the state of the reset and load signals. This functionality ensures that the processor decodes and executes the correct instruction at each step of its operation.

- **Program Counter**

The Program Counter (PC) is responsible for tracking the address of the next instruction to be executed, incrementing its value with each instruction cycle. It resets to zero once it reaches its maximum value, ensuring a cyclical flow of instruction execution. The simulation demonstrates the PC's incrimination logic and its critical role in guiding the processor through the instruction sequence, ensuring a seamless flow of execution.

- **Multiplexer**

The Multiplexer plays a key role in directing the flow of data within the processor by selecting between multiple input signals based on a control signal (`fetch`). It switches between `irout` and `pcout` as outputs, depending on the state of the `fetch` signal. This functionality is crucial for dynamic data routing within the processor, allowing for flexible control over data paths based on operational requirements.

- **Memory Module**

The Memory Module, encompassing both RAM and ROM functionalities, with 256 memory locations, each capable of storing 32 bits provides essential storage capabilities within the processor. It supports read and write operations, enabling the processor to interact with memory according to the needs of the instruction being executed. The simulation shows how data is managed within the memory, including writing to and reading from specific locations, as well as resetting data based on control signals, highlighting the memory's role in data storage and retrieval.

- **Control Unit**

The Control Unit orchestrates the operation of the processor by generating control signals based on the current opcode. These signals direct the behavior of other components, such as the ALU, memory, and registers, ensuring coordinated execution of instructions. The simulation results demonstrate how the Control Unit adapts its output to match the requirements of each instruction, facilitating the processor's adaptive response to different operational scenarios.

- **Buffer**

The Buffer temporarily holds data, acting as an intermediary between components to ensure smooth data transfer and synchronization. Its output is contingent on the control signal (`y`), with the simulation indicating that the buffer actively transfers data when `y` is '1'. This selective activation underscores the buffer's role in managing data flow within the processor, contributing to efficient data handling and processing.

6. Implementation

The implementation phase of the 32-bit RISC processor project involved translating the architectural design into Verilog HDL code and realizing the various hardware components necessary for processor functionality. This section provides a detailed overview of the steps taken during the implementation process.

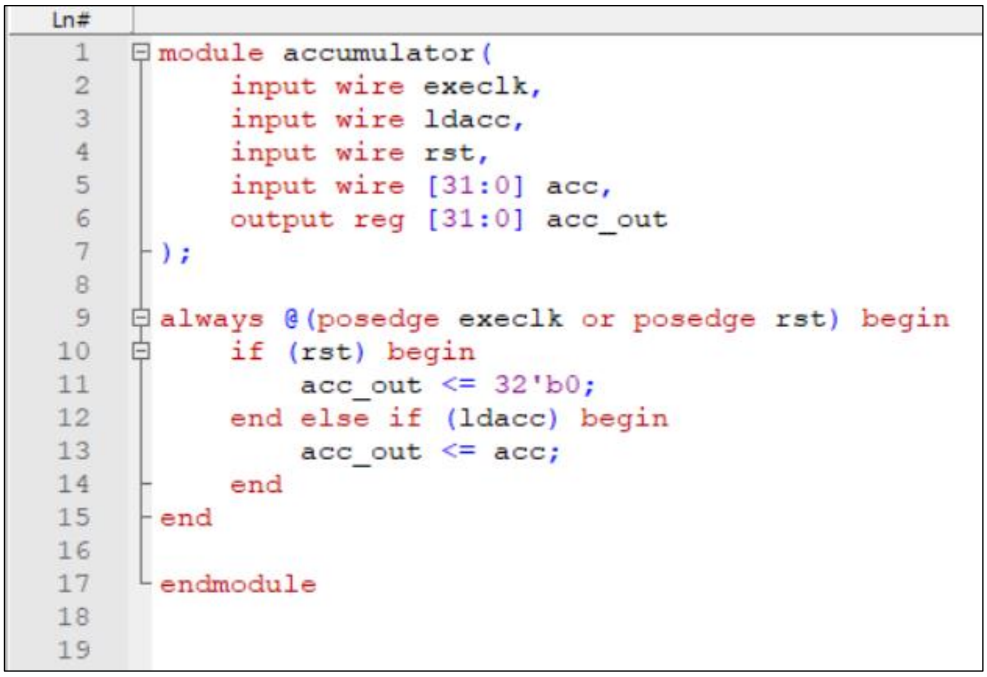
- **Conceptualization and Design**

Black Box Diagrams: For each module (e.g., Accumulator, ALU, Instruction Register, Program Counter, Multiplexer, Memory Module, Control Unit, Buffer), the first step involves drawing black box diagrams. These diagrams are essential for understanding the inputs, outputs, and primary functions of each module without getting bogged down in implementation details.

- **Coding in Verilog**

Verilog Implementation: After finalizing the design, the next step involves coding each module using Verilog, a hardware description language (HDL) favored for its ability to model electronic systems at various levels of abstraction.

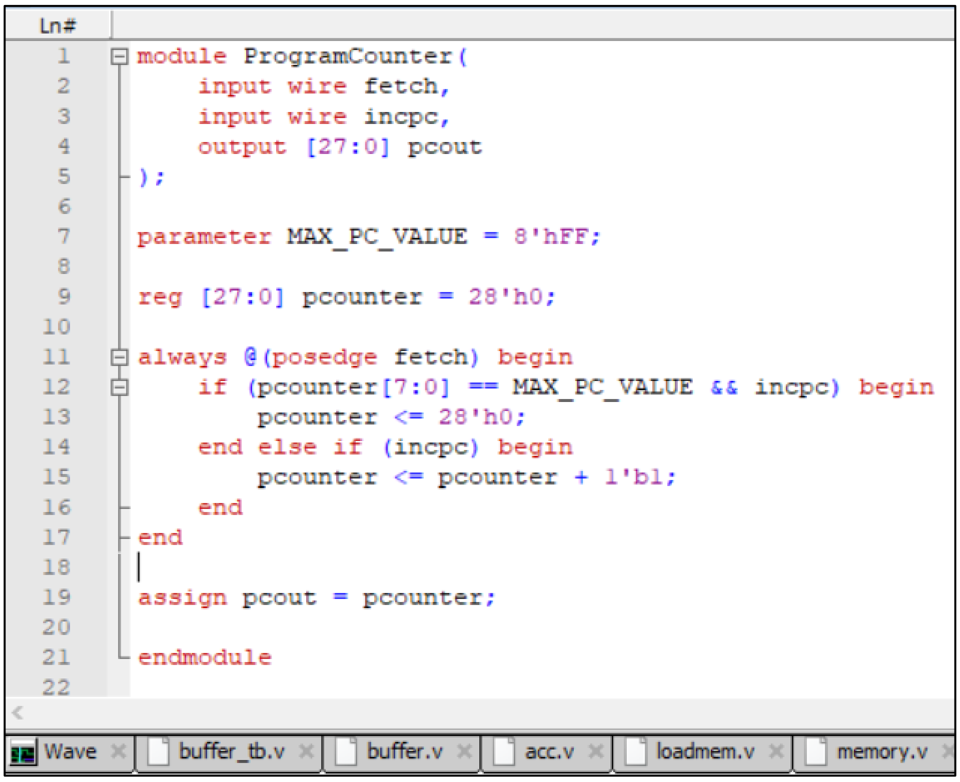
- **Accumulator**



```
module accumulator(
     input wire execlk,
     input wire ldacc,
     input wire rst,
     input wire [31:0] acc,
     output reg [31:0] acc_out
);

always @(posedge execlk or posedge rst) begin
     if (rst) begin
          acc_out <= 32'b0;
     end else if (ldacc) begin
          acc_out <= acc;
     end
end

endmodule
```

Figure 2: Verilog code for Accumulator

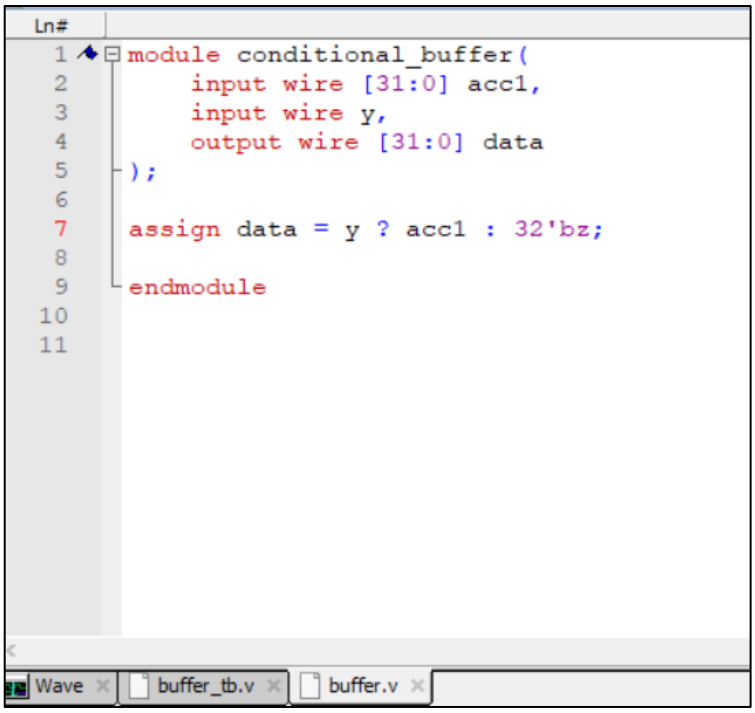
- **Program Counter**



```
module ProgramCounter(
     input wire fetch,
     input wire incpc,
     output [27:0] pcout
);

parameter MAX_PC_VALUE = 8'hFF;

reg [27:0] pcounter = 28'h0;

always @(posedge fetch) begin
     if (pcounter[7:0] == MAX_PC_VALUE && incpc) begin
          pcounter <= 28'h0;
     end else if (incpc) begin
          pcounter <= pcounter + 1'b1;
     end
end

assign pcout = pcounter;

endmodule
```

Figure 3: Verilog code for Program Counter

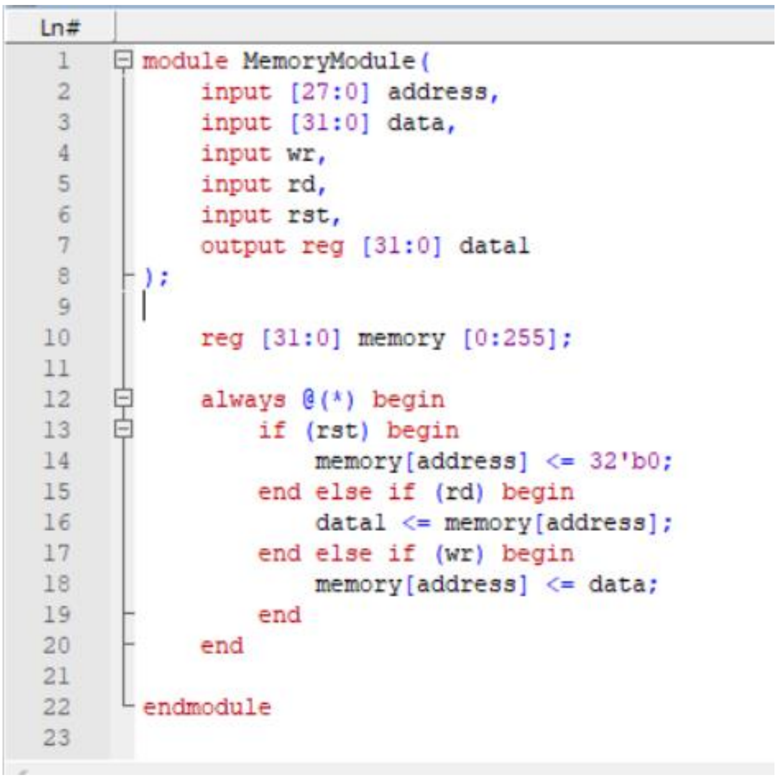
- **Buffer**



```
module conditional_buffer(
     input wire [31:0] acc1,
     input wire y,
     output wire [31:0] data
);

assign data = y ? acc1 : 32'bz;

endmodule
```

Figure 4: Verilog code for Buffer

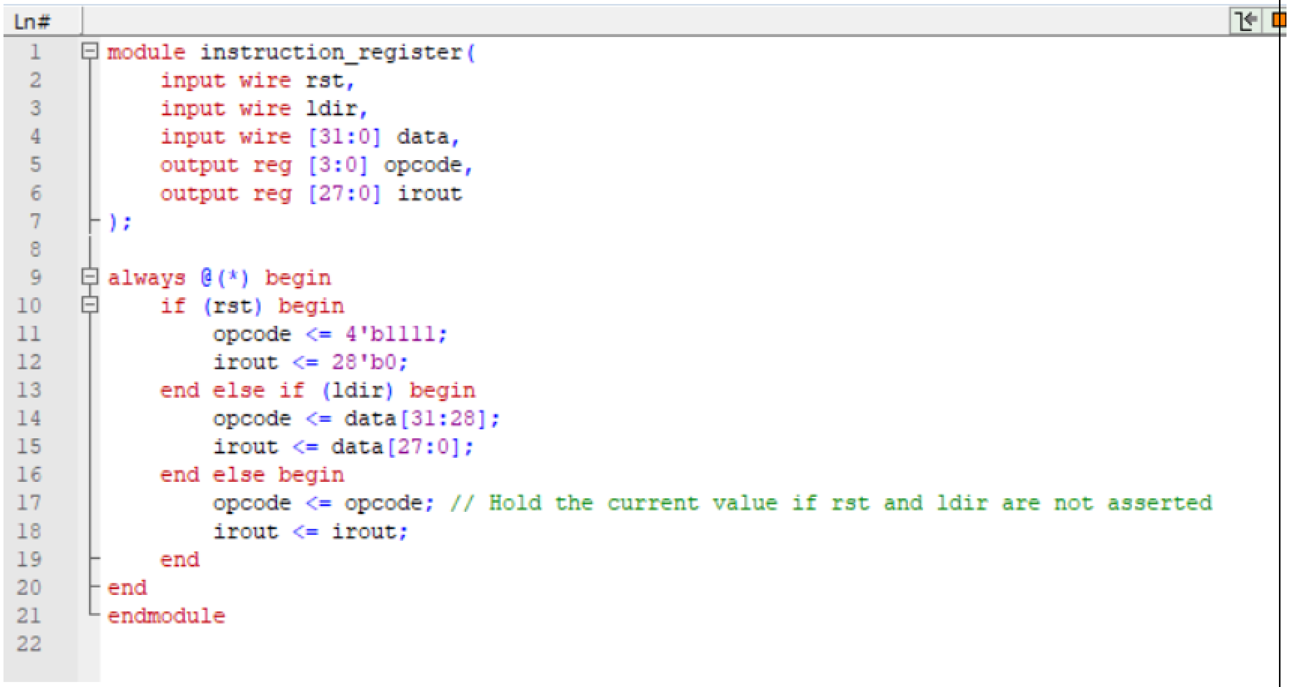
- **Memory Module**



```verilog
module MemoryModule(
    input [27:0] address,
    input [31:0] data,
    input wr,
    input rd,
    input rst,
    output reg [31:0] datal
);

    reg [31:0] memory [0:255];

    always @(*) begin
        if (rst) begin
            memory[address] <= 32'b0;
        end else if (rd) begin
            datal <= memory[address];
        end else if (wr) begin
            memory[address] <= data;
        end
    end

endmodule
```

Figure 5: Verilog code for Memory Module

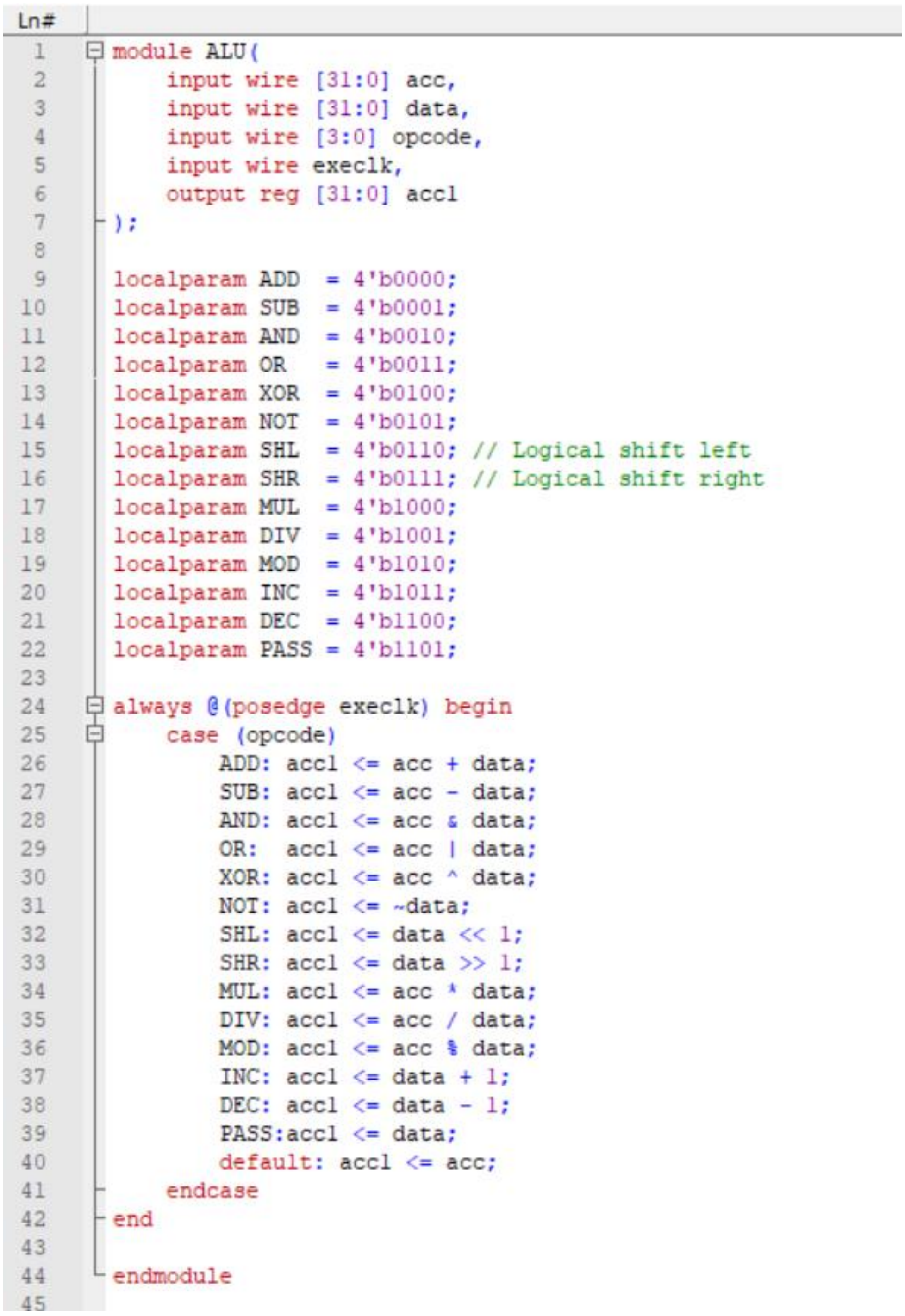
- **Instruction Register**



```verilog
module instruction_register(
    input wire rst,
    input wire ldir,
    input wire [31:0] data,
    output reg [3:0] opcode,
    output reg [27:0] irout
);

always @(*) begin
    if (rst) begin
        opcode <= 4'b1111;
        irout <= 28'b0;
    end else if (ldir) begin
        opcode <= data[31:28];
        irout <= data[27:0];
    end else begin
        opcode <= opcode; // Hold the current value if rst and ldir are not asserted
        irout <= irout;
    end
end
endmodule
```

Figure 6: Verilog code for Instruction Register

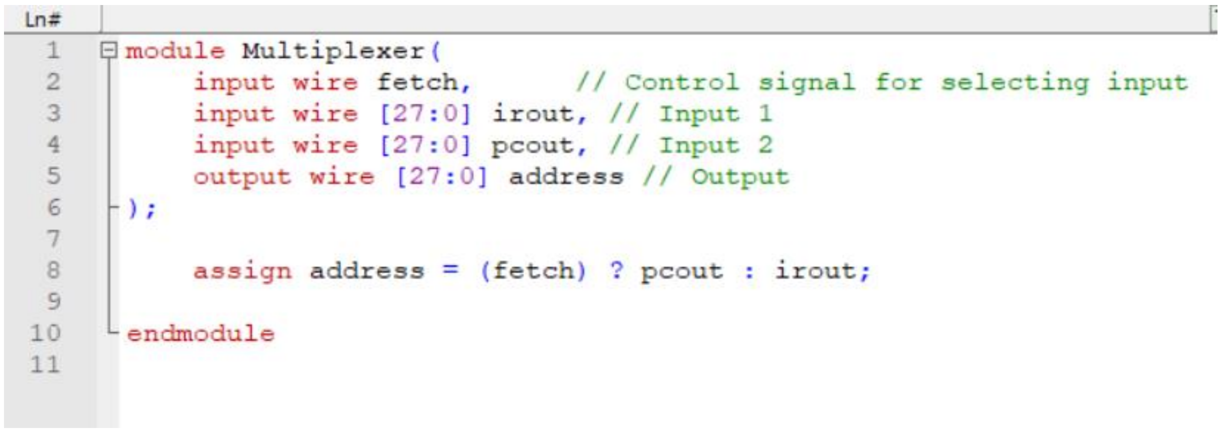
- **ALU**



```
module ALU(
    input wire [31:0] acc,
    input wire [31:0] data,
    input wire [3:0] opcode,
    input wire execlk,
    output reg [31:0] accl
);

localparam ADD  = 4'b0000;
localparam SUB  = 4'b0001;
localparam AND  = 4'b0010;
localparam OR   = 4'b0011;
localparam XOR  = 4'b0100;
localparam NOT  = 4'b0101;
localparam SHL  = 4'b0110; // Logical shift left
localparam SHR  = 4'b0111; // Logical shift right
localparam MUL  = 4'b1000;
localparam DIV  = 4'b1001;
localparam MOD  = 4'b1010;
localparam INC  = 4'b1011;
localparam DEC  = 4'b1100;
localparam PASS = 4'b1101;

always @(posedge execlk) begin
    case (opcode)
        ADD: accl <= acc + data;
        SUB: accl <= acc - data;
        AND: accl <= acc & data;
        OR:  accl <= acc | data;
        XOR: accl <= acc ^ data;
        NOT: accl <= ~data;
        SHL: accl <= data << 1;
        SHR: accl <= data >> 1;
        MUL: accl <= acc * data;
        DIV: accl <= acc / data;
        MOD: accl <= acc % data;
        INC: accl <= data + 1;
        DEC: accl <= data - 1;
        PASS:accl <= data;
        default: accl <= acc;
    endcase
end

endmodule
```

Figure 7: Verilog code for ALU

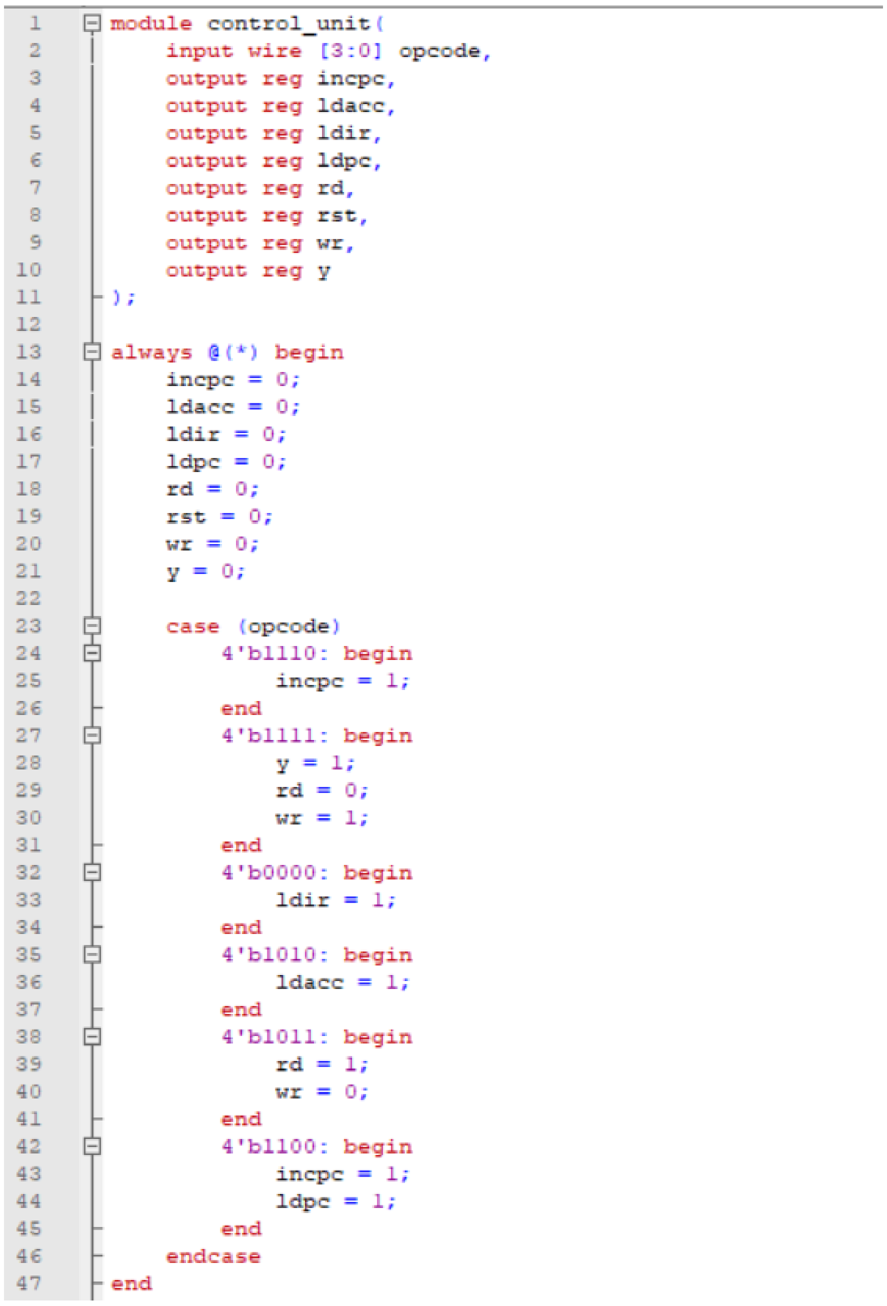
- **MUX**



```
module Multiplexer(
    input wire fetch,        // Control signal for selecting input
    input wire [27:0] irout, // Input 1
    input wire [27:0] pcout, // Input 2
    output wire [27:0] address // Output
);

    assign address = (fetch) ? pcout : irout;

endmodule

```

Figure 8: Verilog code for MUX

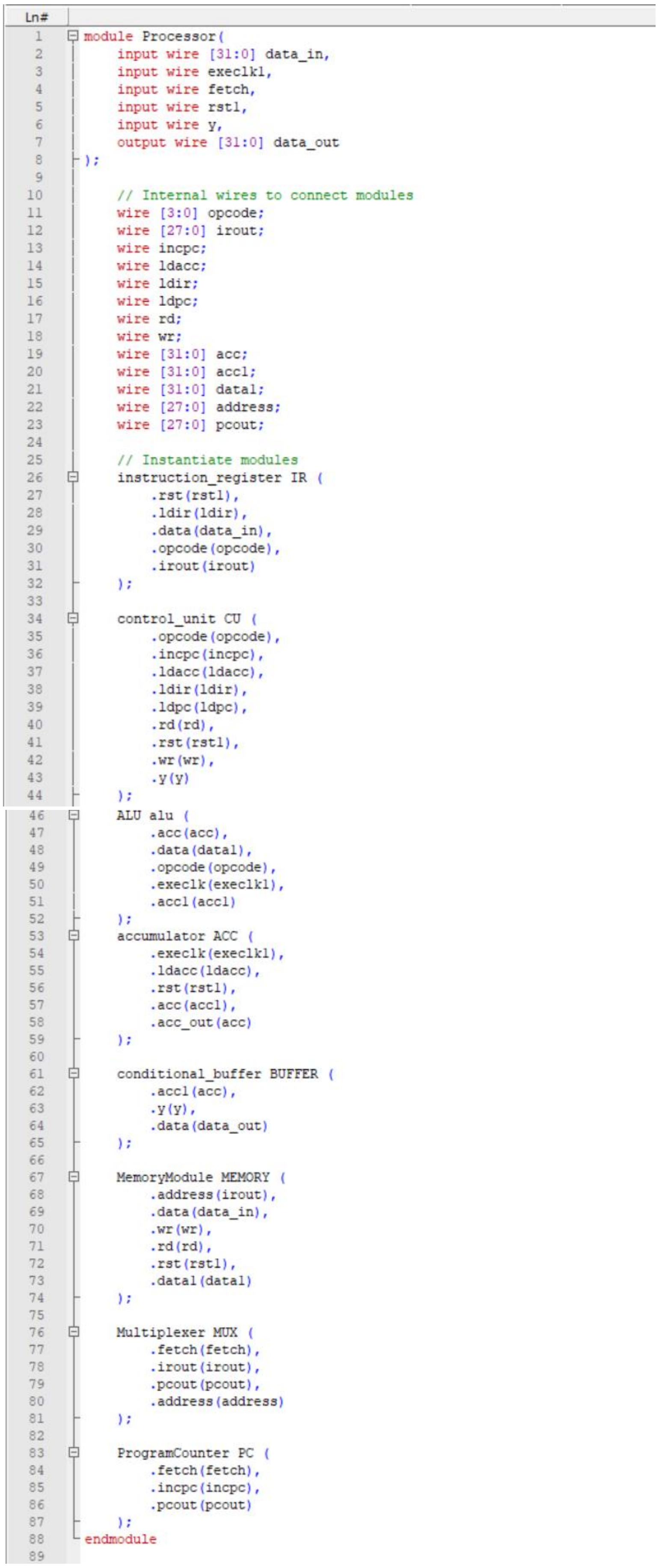
- **Control Unit**



```
module control_unit(
    input wire [3:0] opcode,
    output reg incpc,
    output reg ldacc,
    output reg ldir,
    output reg ldpc,
    output reg rd,
    output reg rst,
    output reg wr,
    output reg y
);

always @(*) begin
    incpc = 0;
    ldacc = 0;
    ldir = 0;
    ldpc = 0;
    rd = 0;
    rst = 0;
    wr = 0;
    y = 0;

    case (opcode)
        4'b1110: begin
            incpc = 1;
        end
        4'b1111: begin
            y = 1;
            rd = 0;
            wr = 1;
        end
        4'b0000: begin
            ldir = 1;
        end
        4'b1010: begin
            ldacc = 1;
        end
        4'b1011: begin
            rd = 1;
            wr = 0;
        end
        4'b1100: begin
            incpc = 1;
            ldpc = 1;
        end
    endcase
end
```

Figure 9: Verilog code for Control Unit

- **Processor**



```
module Processor(
    input wire [31:0] data_in,
    input wire execlkl,
    input wire fetch,
    input wire rstl,
    input wire y,
    output wire [31:0] data_out
);

    // Internal wires to connect modules
    wire [3:0] opcode;
    wire [27:0] irout;
    wire incpc;
    wire ldacc;
    wire ldir;
    wire ldpc;
    wire rd;
    wire wr;
    wire [31:0] acc;
    wire [31:0] accl;
    wire [31:0] datal;
    wire [27:0] address;
    wire [27:0] pcout;

    // Instantiate modules
    instruction_register IR (
        .rst(rstl),
        .ldir(ldir),
        .data(data_in),
        .opcode(opcode),
        .irout(irout)
    );

    control_unit CU (
        .opcode(opcode),
        .incpc(incpc),
        .ldacc(ldacc),
        .ldir(ldir),
        .ldpc(ldpc),
        .rd(rd),
        .rst(rstl),
        .wr(wr),
        .y(y)
    );
    ALU alu (
        .acc(acc),
        .data(datal),
        .opcode(opcode),
        .execlk(execlkl),
        .accl(accl)
    );
    accumulator ACC (
        .execlk(execlkl),
        .ldacc(ldacc),
        .rst(rstl),
        .acc(accl),
        .acc_out(acc)
    );

    conditional_buffer BUFFER (
        .accl(acc),
        .y(y),
        .data(data_out)
    );

    MemoryModule MEMORY (
        .address(irout),
        .data(data_in),
        .wr(wr),
        .rd(rd),
        .rst(rstl),
        .datal(datal)
    );

    Multiplexer MUX (
        .fetch(fetch),
        .irout(irout),
        .pcout(pcout),
        .address(address)
    );

    ProgramCounter PC (
        .fetch(fetch),
        .incpc(incpc),
        .pcout(pcout)
    );
endmodule
```

Figure 10: Verilog code for Processor

- **Challenges and Solutions**

Error Handling: One common challenge involves debugging syntax and logical errors in Verilog code. Solutions often require thorough review of the code, consultation of Verilog documentation, and leveraging online forums and communities for specific error messages.

7. Testing and Verification

- **Accumulator**

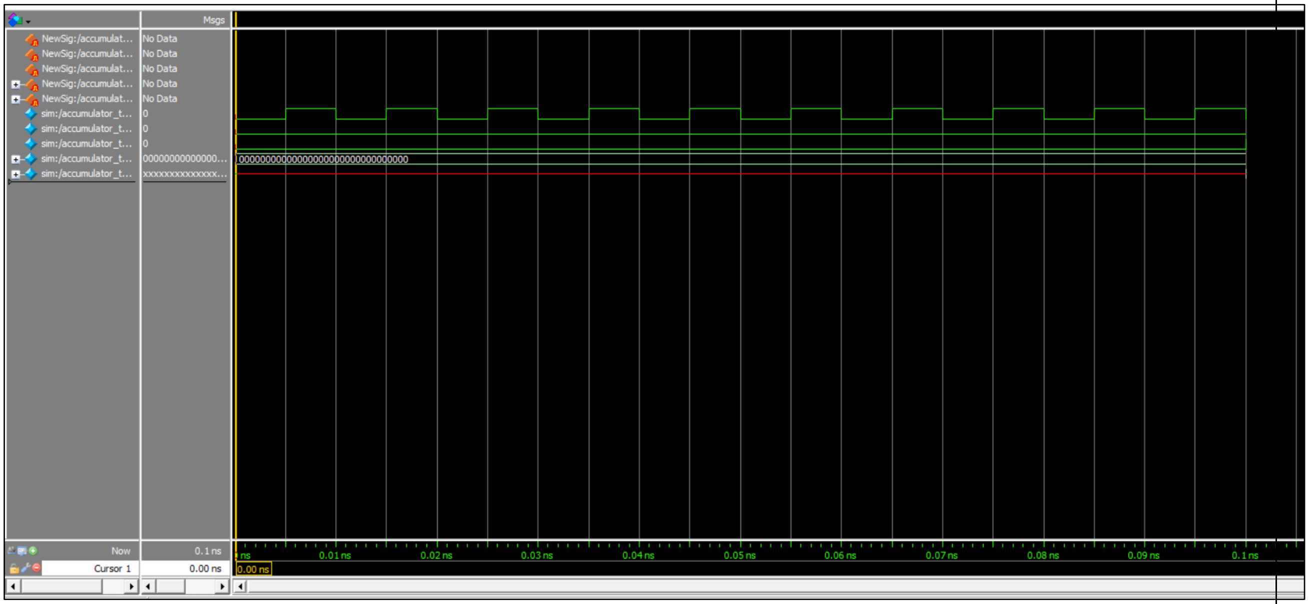

Msgs
NewSig:/accumulat… No Data
NewSig:/accumulat… No Data
NewSig:/accumulat… No Data
NewSig:/accumulat… No Data
NewSig:/accumulat… No Data
sim:/accumulator_t… 0
sim:/accumulator_t… 0
sim:/accumulator_t… 0
sim:/accumulator_t… 00000000000000…
sim:/accumulator_t… xxxxxxxxxxxxxx…
00000000000000000000000000000000
Now 0.1 ns
Cursor 1 0.00 ns
ns 0.01 ns 0.02 ns 0.03 ns 0.04 ns 0.05 ns 0.06 ns 0.07 ns 0.08 ns 0.09 ns 0.1 ns
0.00 ns

Figure 11: Test bench result for Accumulator

- **Buffer**

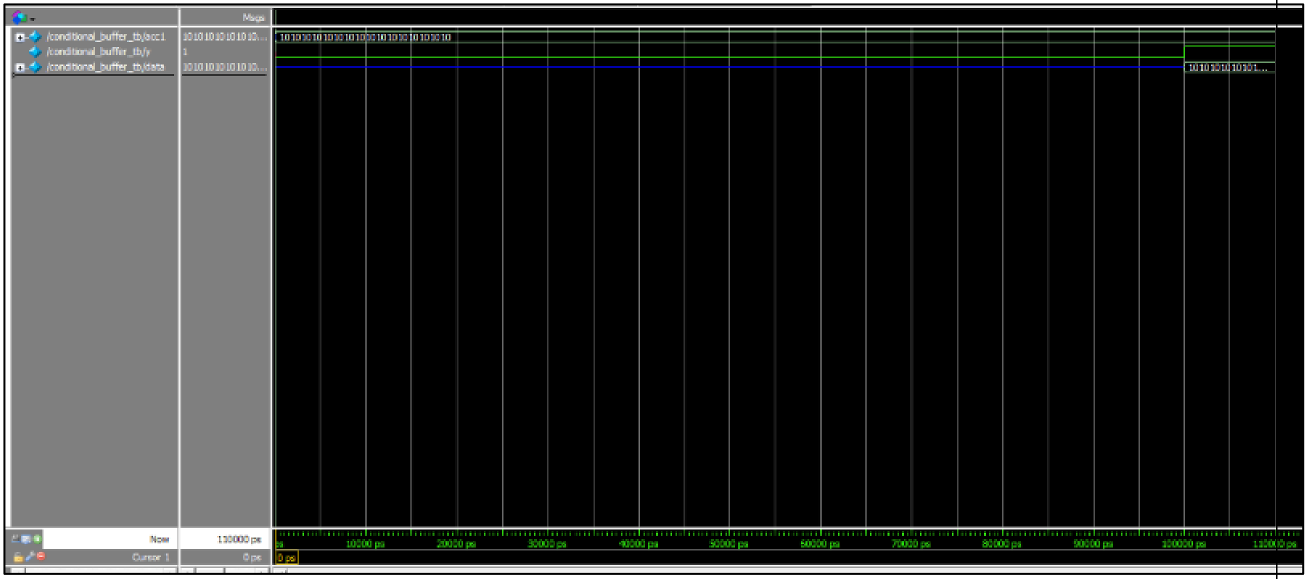

Msgs
/conditional_buffer_tb/acc1 10101010101010…
/conditional_buffer_tb/y 1
/conditional_buffer_tb/data 10101010101010…
10101010101010101010101010101010
1010101010101…
Now 110000 ps
Cursor 1 0 ps
ps 10000 ps 20000 ps 30000 ps 40000 ps 50000 ps 60000 ps 70000 ps 80000 ps 90000 ps 100000 ps 110000 ps
0 ps

Figure 12: Test bench result for Buffer

- **Memory Module**

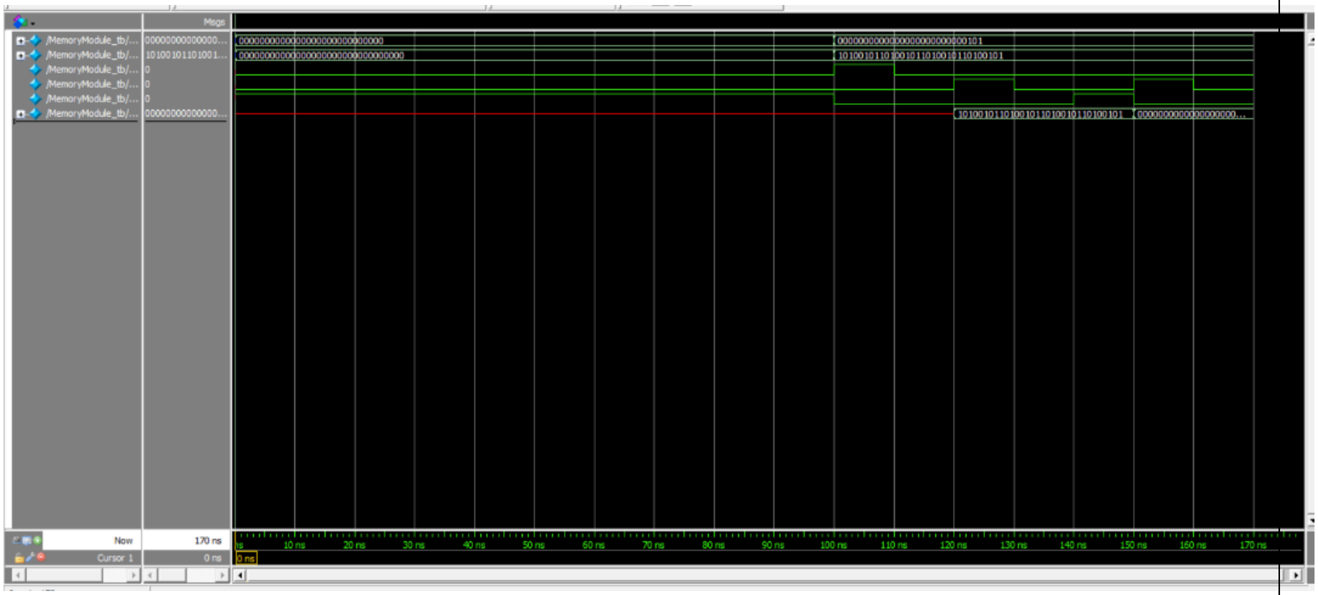


Figure 13: Test bench result for Memory Module

- **Instruction Register**

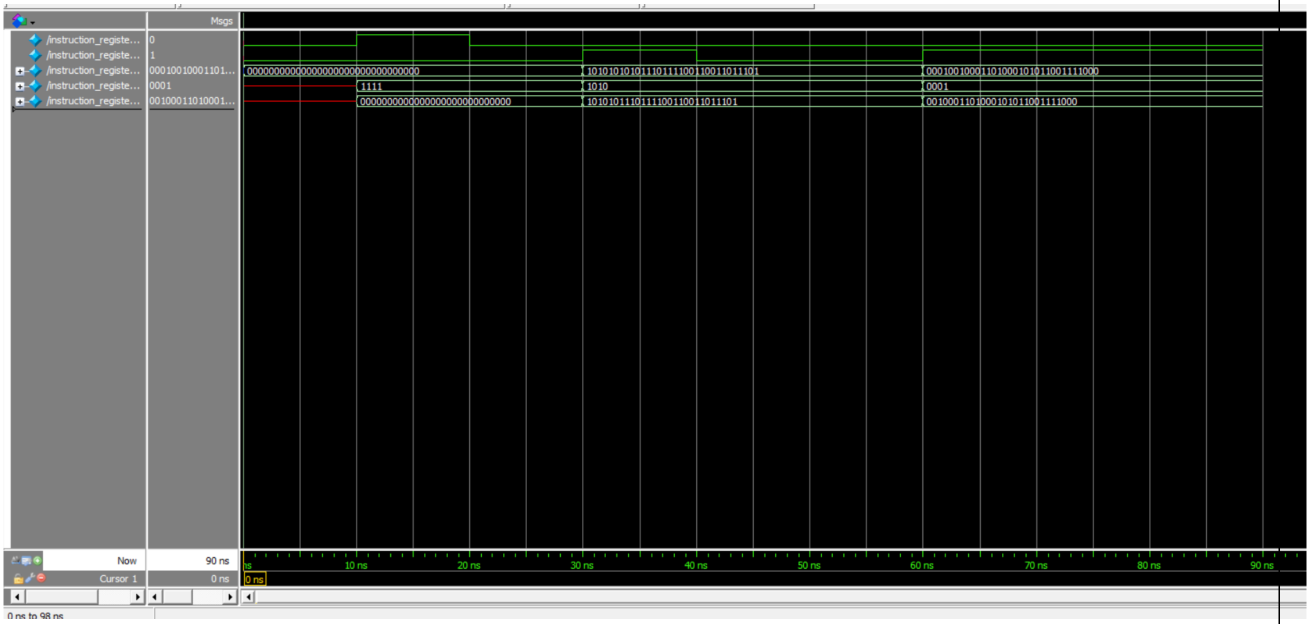


Figure 14: Test bench result for Instruction Register

- **ALU**

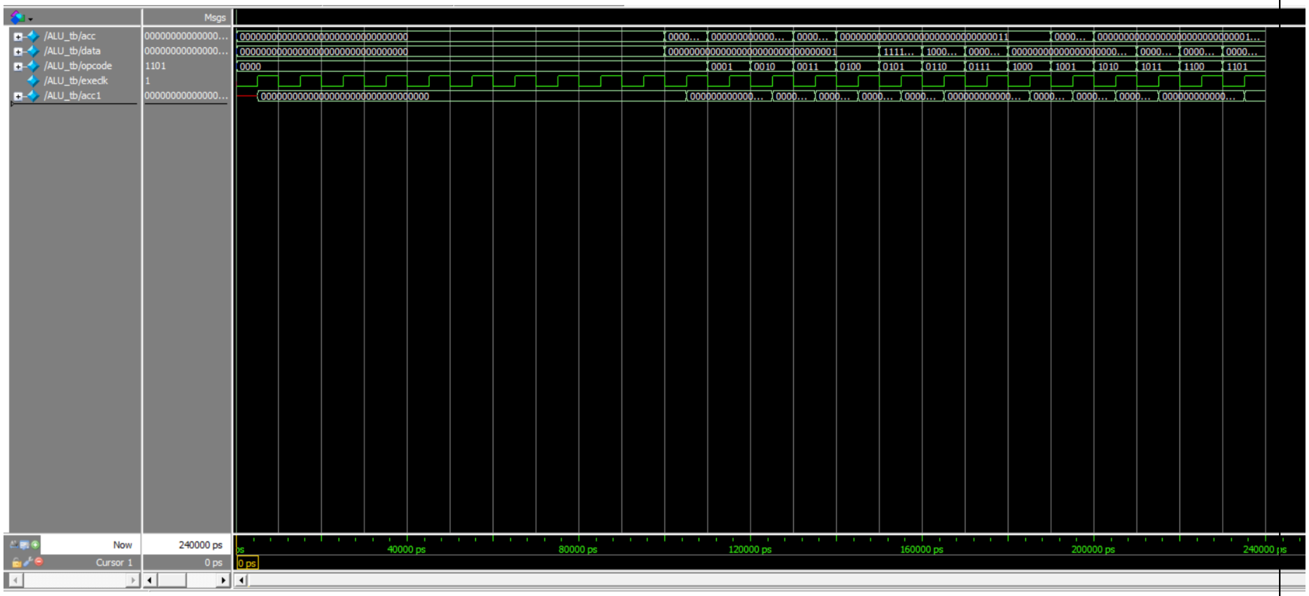


Figure 15: Test bench result for ALU

- **MUX**

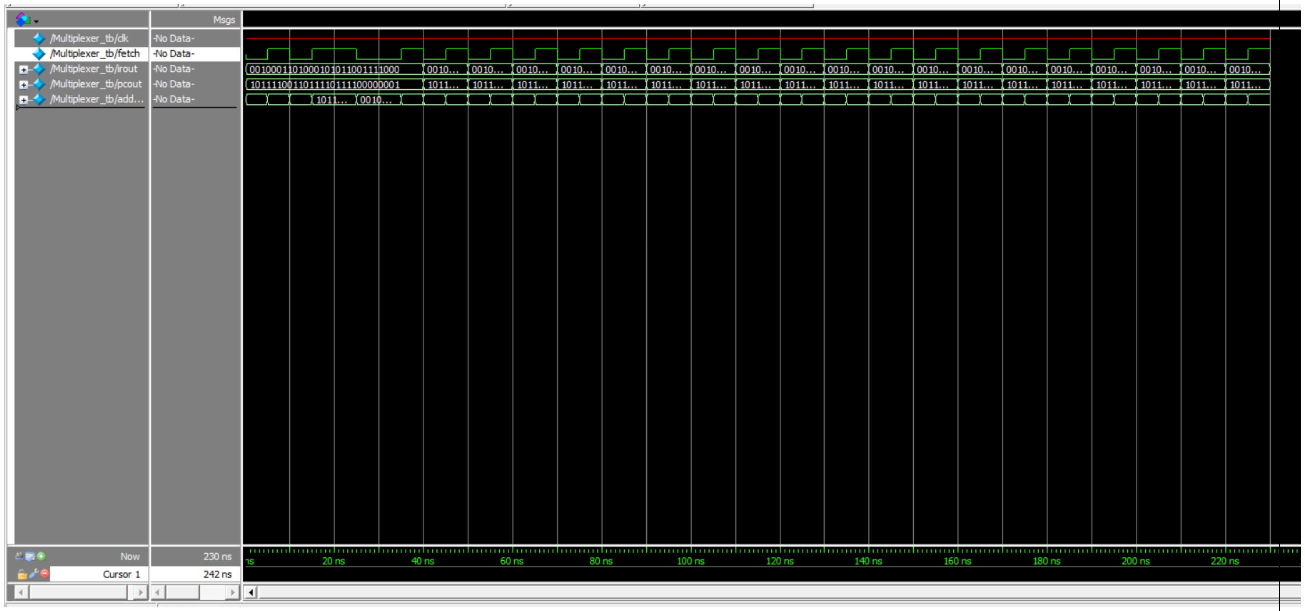


Figure 16: Test bench result for MUX

- **Control Unit**

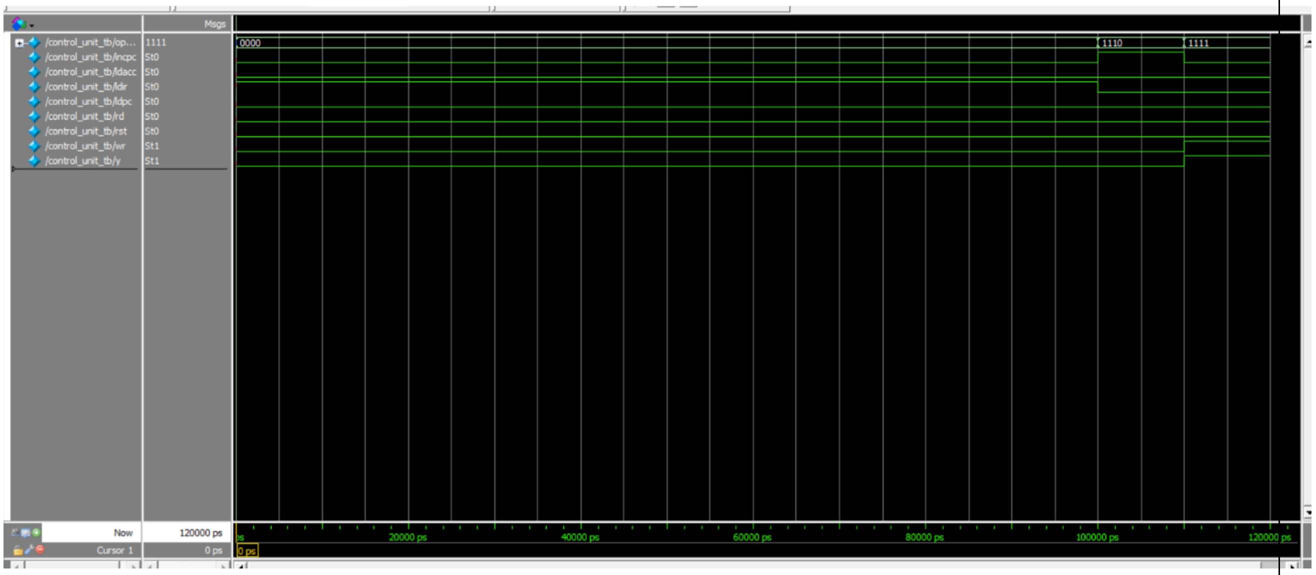


Figure 17: Test bench result for Control Unit

8. Results and Discussion

The overall performance and functionality of the 32-bit RISC processor were evaluated based on the successful verification of individual modules and their integration into a whole system. While we were able to verify each module independently, challenges arose when attempting to synchronize their operations within the complete processor. Despite this limitation, significant progress was made towards achieving our initial objectives.

Comparing the results with initial objectives and expectations, we have largely met our goals in terms of module verification and functionality. However, the inability to seamlessly integrate all modules highlights areas that require further attention and refinement in future iterations of the design process.

9. Conclusion

In conclusion, this project outlines the implementation and simulation of a RISC processor with a memory controller using the ModelSim simulator. RISC processors are widely applied across various domains, ranging from industrial applications to electronic gadgets and even some of the world's fastest supercomputers like the K computer. To optimize memory utilization, a straightforward RISC processor design is chosen. This design integrates several essential blocks including the ALU, Accumulator, Control Unit (CU), Memory, Program Counter, Instruction Register, and Buffer. Employing simplified logic in the memory and control unit components enhances the processor's speed. Furthermore, the processor is engineered to execute each instruction within a single clock cycle, contributing to its overall efficiency and effectiveness.

10. Future Work

While the development of the 32-bit RISC processor marks a significant achievement, there are several avenues for future work and enhancement to explore. This section outlines potential areas for further development and improvement:

- **Performance Optimization**

Branch Prediction: Implementing branch prediction algorithms to reduce the latency associated with control flow changes in the processor. This can significantly improve the processor's performance by minimizing the number of stalls that occur during branching instructions.

Pipeline Optimization: Enhancing the processor's pipeline stages for more efficient instruction processing. This could involve fine-tuning the pipeline to reduce hazards and implementing techniques such as instruction reordering or speculative execution.

- **Instruction Set Expansion**

Adding New Instructions: Extending the instruction set to support a broader range of operations, including complex arithmetic operations, floating-point calculations, and specialized instructions for certain applications. This would increase the versatility and applicability of the processor for various computing tasks.

11. References

[1] Begum, Sd.A. and M, Dr.S. (2015). Implementation of a 32 bit RISC processor with memory controller by using VHDL. *IJIREEICE*, 3(8), pp.110–114. doi:https://doi.org/10.17148/ijireeice.2015.3824.

[2] Trivedi, P. and Tripathi, R.P. (2015). Design & analysis of 16 bit RISC processor using low power pipelining. International Conference on Computing, Communication & Automation. doi:https://doi.org/10.1109/ccaa.2015.7148575.

Project code: https://github.com/lakpahana/32bit_RISC_Processor