Design and Implementation of an 8-bit CPU

Indian Institute of Space Science and Technology
Thiruvananthapuram

Department of Avionics

Submitted by:

Team Logic Architects

Ajit Kumar Singh	SC22B123
Anurag	SC22B125
Saurabh Kumar	SC22B146
Uttam Kumar	SC22B156

December 17, 2024

Contents

Abstract

The design and implementation of an 8-bit CPU have been carried out as part of this project to understand the fundamental principles of processor architecture and microprocessor systems. The 8-bit CPU was designed using Verilog HDL, with a focus on achieving a balance between simplicity and functionality. This processor includes essential components such as the Arithmetic Logic Unit (ALU), control unit, registers, and memory interface, enabling basic computational and control operations.

The project began with defining the architecture, followed by the development of individual modules, their integration, and simulation. Verification strategies were employed to ensure the correctness of each module and the overall system. The CPU is capable of performing basic arithmetic, logical, and control operations, making it suitable for educational and low-complexity applications such as embedded systems and IoT devices.

The results demonstrate the successful implementation of a functional 8-bit CPU that meets the design specifications. This project highlights the importance of modular design, testing, and optimization in digital system development. Future enhancements could include the addition of pipelining, advanced instruction sets, and interfacing capabilities to improve performance and scalability.

Acknowledgment

We would like to express our sincere gratitude to everyone who contributed to the successful completion of this project, Design and Implementation of an 8-bit CPU. We are immensely thankful to Dr. BS Manoj, our project guide, for their invaluable guidance, support, and encouragement throughout this work. Their deep insights and expertise helped us overcome challenges and refine our design approach. We would also like to extend our gratitude to the authority of Digital Signal Processing Lab of Department of Avionics at the Indian Institute of Space Science and Technology for providing the necessary resources, infrastructure, and a conducive learning environment to carry out this project. Our heartfelt thanks to our colleagues and peers for their constructive discussions and cooperation, which helped us improve various aspects of the design.

1 Introduction

1.1 Overview

The central processing unit (CPU) is the core of every computational system, responsible for executing instructions, managing data flow, and ensuring proper coordination of hardware components. This document details the design and implementation of an 8-bit CPU, highlighting its architecture, topologies, and performance.

The CPU is built using a modular approach, ensuring scalability and maintainability. It adheres to the Von Neumann architecture, where data and instructions share the same memory space.

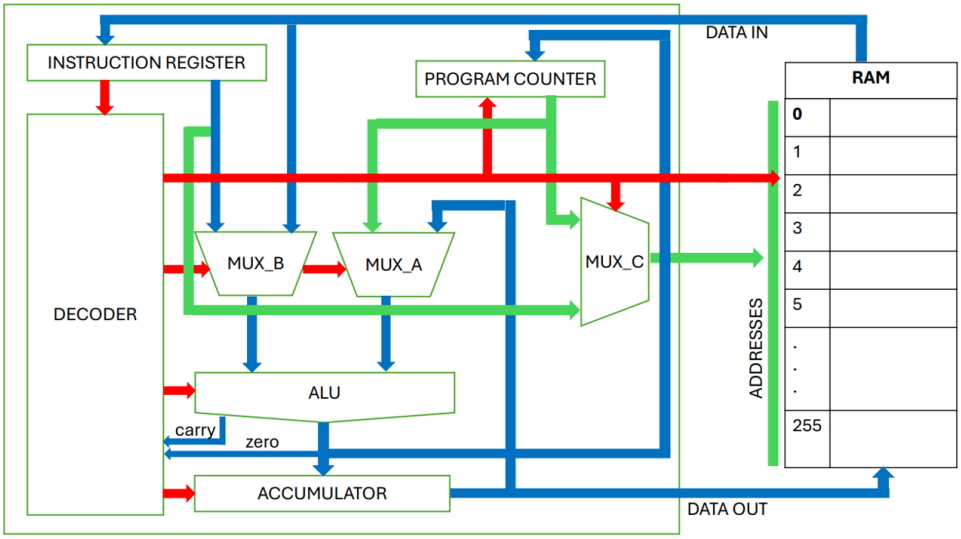

Figure 1: Overview of the CPU Design

1.2 Objective

The primary goals of this project include:

- Understanding and implementing the fundamental components of a CPU.
- Developing a modular design for ease of testing and scaling.
- Exploring different CPU topologies and their applications.

1.3 Applications

The designed CPU can be used in:

- Embedded systems requiring low computational power.
- Educational platforms for teaching CPU architecture.
- Custom logic for specific industrial applications.

2 CPU Topologies

2.1 Von Neumann Architecture

The Von Neumann architecture features a single memory for storing both instructions and data. This design simplifies the hardware but introduces bottlenecks due to shared memory access.

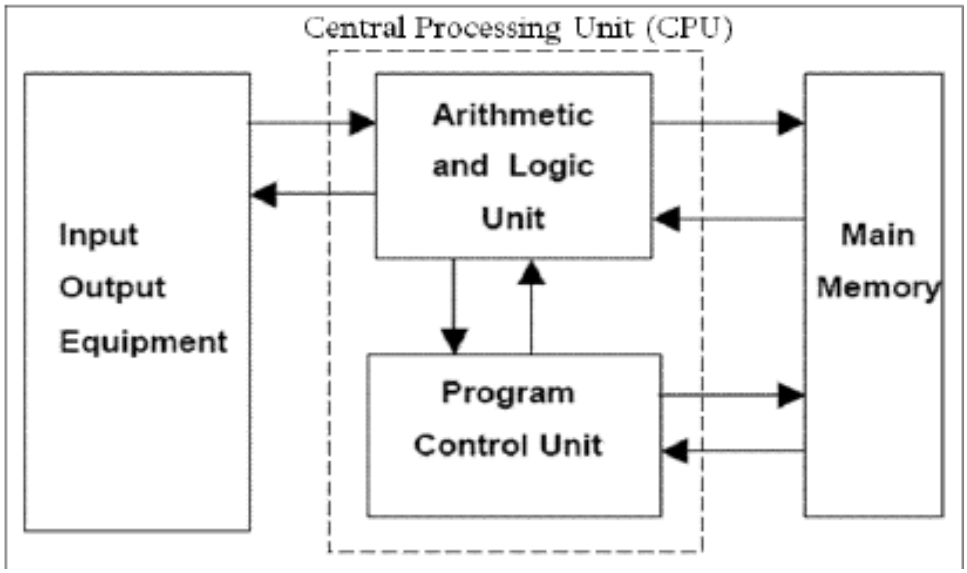

Figure 2: Von Neumann Architecture

2.2 Harvard Architecture

In the Harvard architecture, separate memory spaces are used for instructions and data. This allows simultaneous fetching of instructions and accessing data, significantly improving performance.

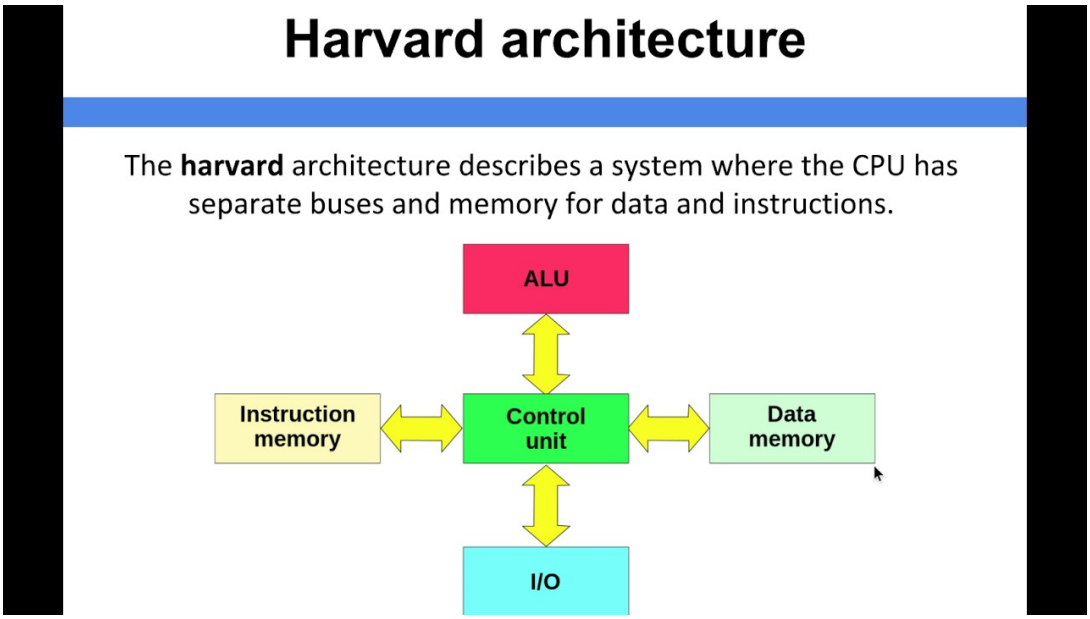

Figure 3: Harvard Architecture

2.3 Pipeline Architecture

Pipeline architecture divides instruction execution into stages, enabling multiple instructions to be processed simultaneously. This increases throughput but adds complexity.

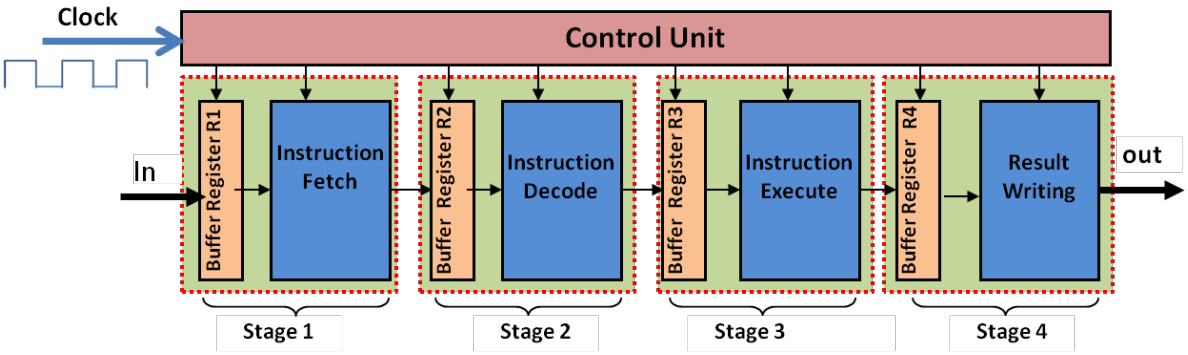

Figure 4: Pipeline Architecture

2.4 Comparison of Architectures

Table 1: Comparison of CPU Architectures

Feature	Von Neumann	Harvard	Pipeline
Memory	Shared	Separate	Depends on design
Performance	Moderate	High	Very High
Complexity	Low	Medium	High
Applications	General-purpose	DSP, Embedded	High-speed CPUs

3 CPU Components

3.1 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a critical component of a processor responsible for performing arithmetic and logical operations. It handles basic computations such as addition, subtraction, multiplication, and division, as well as logical operations like AND, OR, XOR, and comparisons. The ALU processes data retrieved from registers or memory and produces results that are either stored back or used to influence program control. By executing these fundamental operations efficiently, the ALU plays a central role in enabling the processor to carry out complex instructions and tasks required by a program. The desigend ALU performs:

- Addition, Subtraction, Increment, Add and Increment, Subtract and Decrement, Multiplication, and Division.
- Logical operations like Bitwise AND, and Bitwise OR.

```
module alu (input [7:0] A, B,
            input [4:0] SEL,
            output reg [7:0] RESULT,
            output reg Cout);
```

Table 2: ALU Operations

Operation	**Code (SEL)**	**Result**
Addition	00000	A + B
Subtraction	01100	A - B
Multiplication	01110	A * B
Divide	01010	A / B
AND	00001	A & B
OR	00101	A | B
INCREMENT	10100	A+1
ADD and INCREMENT	00100	A+B+1
SUBTRACT and DECREMENT	01000	A-B-1

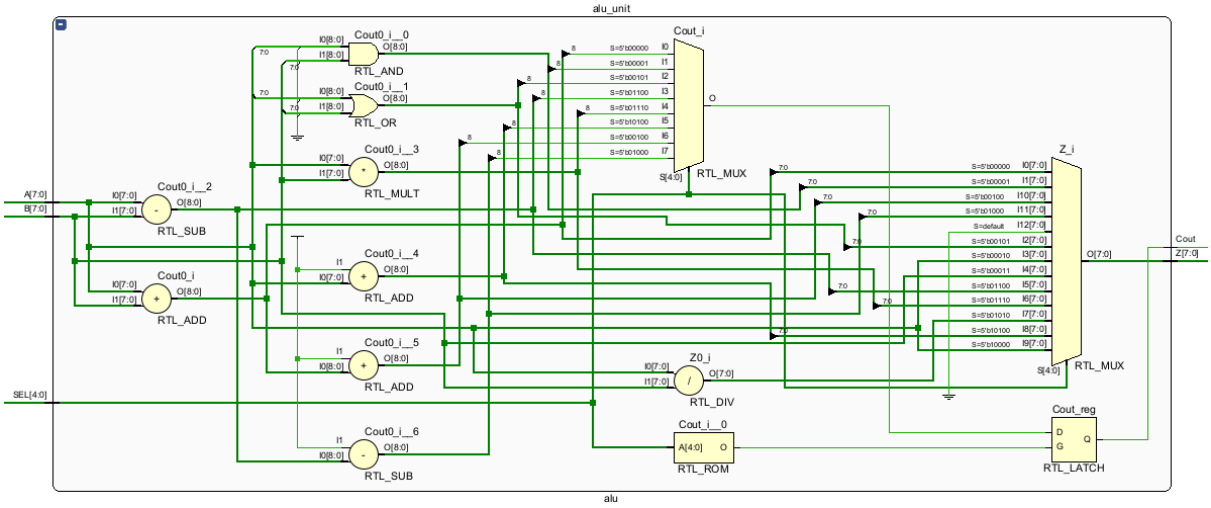

Figure 5: ALU Schematic

3.2 Registers

Registers are high-speed storage locations within a processor, designed to temporarily hold data, instructions, or addresses during execution. They are essential for efficient data manipulation and serve as a crucial interface between the processor and memory. Acting as the processor's working memory, registers enable quick access to frequently used values, reducing the need for slower memory operations. These small, fast storage units support operations like arithmetic, logic, and control flow, and are vital for maintaining intermediate results, managing program execution, and handling memory addresses during processing tasks. Common registers incorporated include:

- Accumulator : Stores results from the ALU (8-bit).
- Program Counter (PC): Points to the next instruction (8-bit).

```
module register(input clk, reset, input [7:0] din, output reg
    [7:0] dout);
always @(posedge clk or posedge reset) begin
    if (reset) dout <= 8'b0;
    else dout <= din;
end
```

- Instruction Register (IR): Receive instruction from memory (16-bit).

```
module register(input clk, reset, input [15:0] din, output
   reg [15:0] dout);
always @(posedge clk or posedge reset) begin
    if (reset) dout <= 16'b0;
    else dout <= din;
end
```

- Status Register : Hold information about Zero or Carry status (2-bit).

```
module register(input clk, reset, input [1:0] din, output reg
    [1:0] dout);
always @(posedge clk or posedge reset) begin
    if (reset) dout <= 2'b0;
    else dout <= din;
end
```

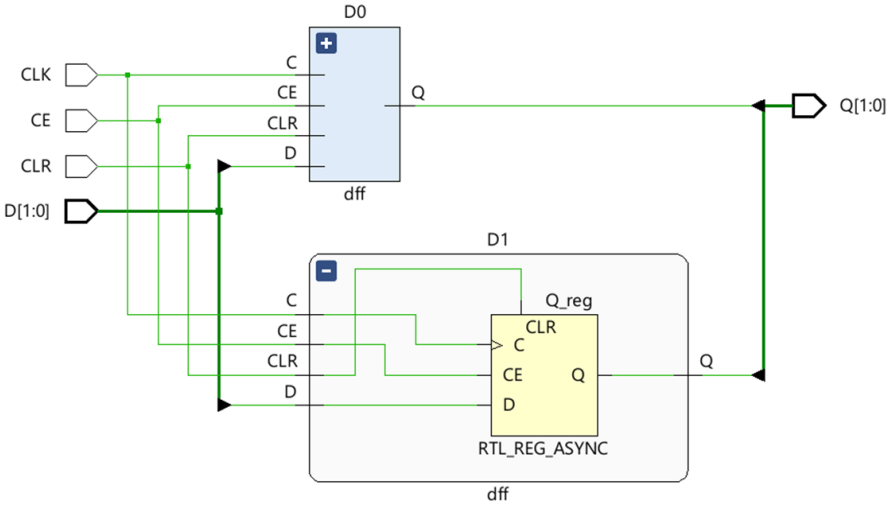

Figure 6: 2 bit Register (Schematic)

3.3 Multiplexer

The system uses three 8-bit multiplexers to efficiently control data flow and address selection, ensuring smooth interaction between the CPU's internal components and external memory. These multiplexers serve distinct roles in the system's operation.

1. Data Mux A: This multiplexer is responsible for selecting between two critical sources of data: the program counter and the accumulator. The program counter holds the address of the next instruction to be executed, while the accumulator stores the results of arithmetic or logical operations. By switching between these outputs, Data Mux A allows the system to dynamically decide whether to prioritize program flow or computation, providing versatility in handling both control and operational tasks.

2. Data Mux B: This multiplexer enables the CPU to select between two input data sources. One source is the CPU data input bus, which brings data from external devices or memory into the processor. The other source is the data byte output of the instruction register, which contains the current instruction being executed. Data Mux B thus facilitates seamless switching between external data and internal instructions, ensuring that the processor can efficiently decode instructions and handle data input simultaneously.

3. Address Mux: This multiplexer is crucial for directing memory access operations. It selects between the data byte output of the instruction register and other address

sources to determine which location in RAM should be accessed for reading or writing. By dynamically controlling address selection, the Address Mux ensures that the processor can access the appropriate memory locations, whether for fetching instructions, storing results, or retrieving data for processing.

These multiplexers work together to optimize the CPU's performance, balancing the flow of data and control signals while maintaining efficient memory access. Their design enhances the processor's flexibility, allowing it to adapt to varying operational needs while minimizing bottlenecks in data and address handling.

```
module mux_2x1_8 (output reg [7:0] Z,
                  input [7:0] A, B,
                  input SEL);
  always @ (A, B, SEL)
    case (SEL)
      0: Z <= A; // if sel is 0 select A input
      1: Z <= B; // if sel is 1 select B input
      default: Z <= 8'b00000000; //default case to prevent latch
         being formed
    endcase
endmodule
```

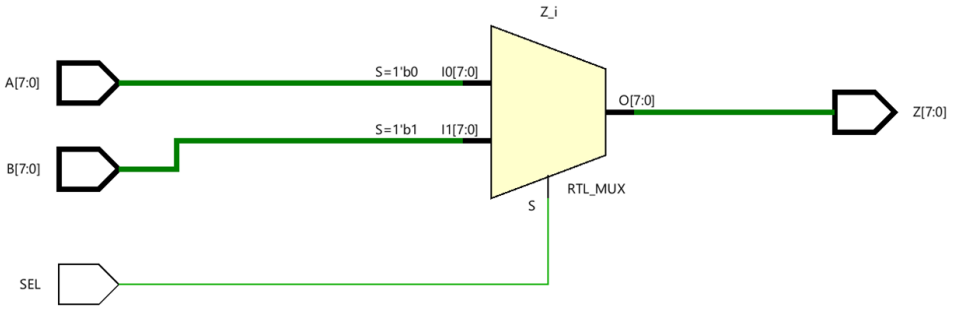

Figure 7: 2X1 8 bit Mux

3.4 RAM

Random Access Memory (RAM) in a 16-bit Von Neumann-structured CPU serves as temporary storage for both data and instructions during program execution. As a volatile memory, it provides fast access to the processor, enabling efficient handling of tasks. In this architecture, the memory space is unified, storing both program instructions and data in the same 16-bit addressable memory locations. This allows the CPU to fetch and process information sequentially but can lead to limitations such as the Von Neumann bottleneck, where the shared bus slows down simultaneous data and instruction access. Despite this, RAM is essential for maintaining the processor's efficiency and ensuring smooth execution of instructions.

```
module ram(input we, clk,
           input [15:0] din,
           input [7:0] addr,
           output reg [15:0] dout);
  reg [15:0] mem [255:0]; //256 16 bit regs

  always @ (posedge clk) begin
    if (we)
        mem[addr] <= din;
  end

  always @ (posedge clk) begin
    if (~we)
      dout <= mem[addr];
  end

endmodule
```

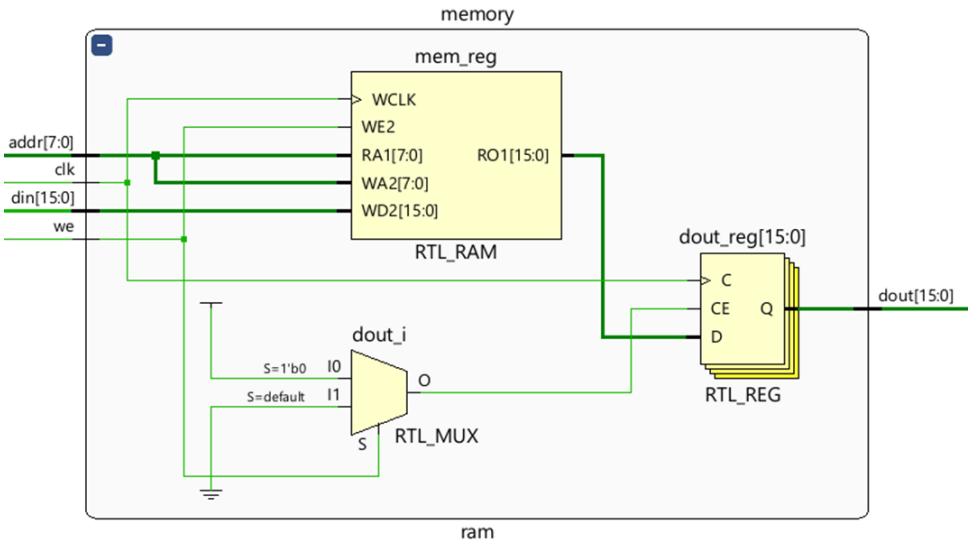

Figure 8: RAM Implementation

3.5 Sequence Generator

The sequence generator in the CPU is a state machine that cycles through four operational states: Fetch (F), Decode (D), Execute (E), and Increment (I). These states are represented using one-hot encoding, and transitions are controlled by a clock signal (CLK), an enable signal (CE), and a clear signal (CLR). On each clock pulse, the sequence generator progresses through the states in a predefined order, ensuring smooth control of the instruction cycle. The clear signal resets the generator to the Fetch state, while the enable signal determines if state transitions should occur.

This module is crucial for orchestrating the operation of the CPU, as it ensures that each phase of instruction execution (fetching from memory, decoding the instruction, executing the operation, and preparing for the next instruction) happens in a synchronized and orderly manner. By using a case statement for state transitions, it provides robust and efficient control, making it easier to reset and manage states during operation.

```
module sequence_generator (input CLK, CE, CLR,
                           output reg F, D, E, I);
  always @ (posedge CLK) begin
    // using case statements here proved more robust than
    // one-hot incrementing as it allows easier reset to fetch (
       default state)
    case ({CLR, CE, F, D, E, I})
      6'b1xxxxx: {F,D,E,I} <= 4'b1000;
      6'b00xxxx: {F,D,E,I} <= {F,D,E,I};
      6'b011000: {F,D,E,I} <= 4'b0100;
      6'b010100: {F,D,E,I} <= 4'b0010;
      6'b010010: {F,D,E,I} <= 4'b0001;
      6'b010001: {F,D,E,I} <= 4'b1000;
      default: {F,D,E,I} <= 4'b1000; // default prevents latching
          circuit
    endcase
  end
endmodule
```

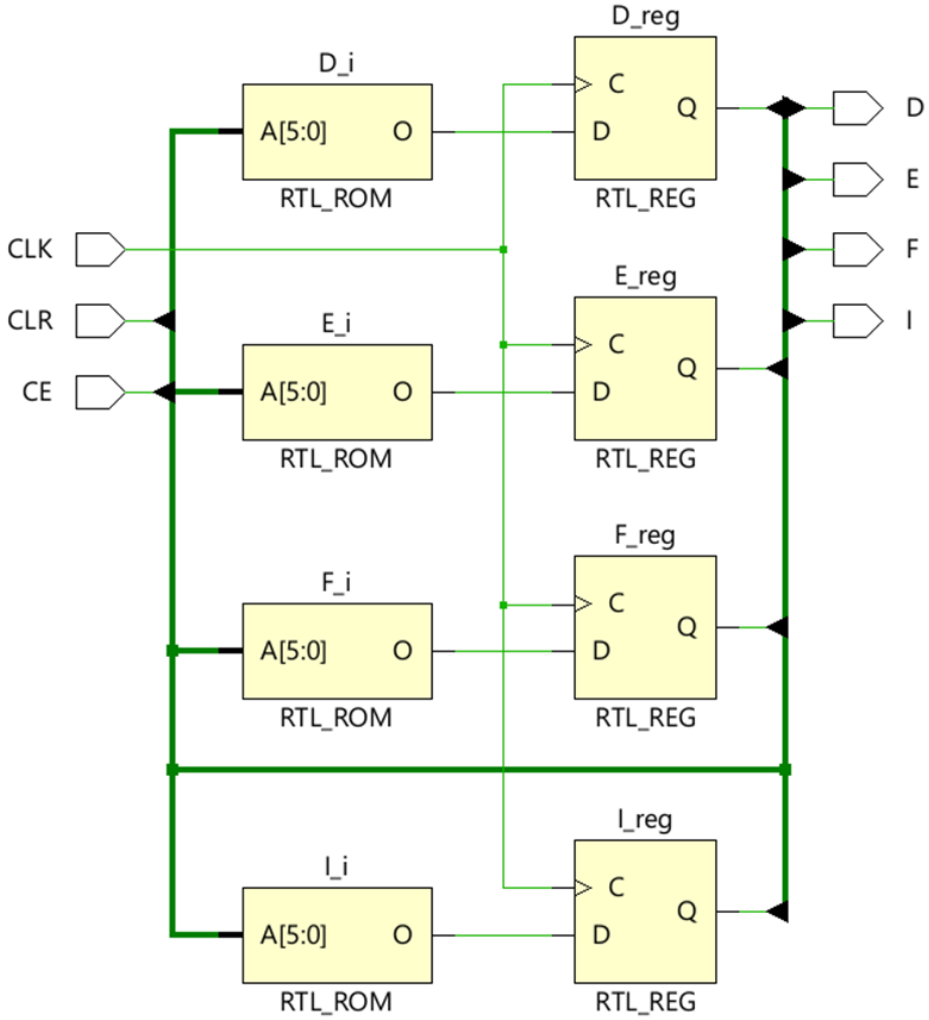

Figure 9: Sequence Generator

3.6 Instruction Decoder

The instruction decoder is a key component in the CPU that interprets the instruction fetched from memory. It takes a binary instruction as input and decodes it into control signals for the various functional units of the CPU, such as the ALU (Arithmetic Logic Unit) and registers. The decoder operates based on the opcode present in the instruction, generating the necessary control signals to execute the operation. It uses a case statement to identify the specific instruction and trigger the corresponding actions, ensuring that the correct operation is performed.

This module is essential for the CPU's ability to understand and execute different types of instructions. By decoding the instruction, the decoder ensures that the CPU responds correctly to the opcode, directing the execution of arithmetic, logic, or data transfer operations. It enables the processor to handle a wide variety of instructions efficiently, playing a critical role in the overall functioning of the CPU.

```
module instruction_decoder (input [7:0] A,
                            input DECODE, EXECUTE,
                            output reg ADD, LOAD, OUTPUT, INPUT,
                               JUMPZ,
                                    JUMP, JUMPNZ, JUMPC, JUMPNC,
                                       SUB, BITAND, BITOR, MUL,
                                       DIV);
  reg or_out;
  always @(*) begin
    or_out = (EXECUTE|DECODE);
    if(or_out)
      casex (A)
        8'b1010xxxx: INPUT = 1;
        8'b1110xxxx: OUTPUT = 1;
        8'b0000xxxx: LOAD = 1;
        8'b0100xxxx: ADD = 1;
        8'b1000xxxx: JUMP = 1;
        8'b0110xxxx: SUB = 1;
        8'b0111xxxx: MUL = 1;
        8'b0101xxxx: DIV = 1;
        8'b0001xxxx: BITAND = 1;
        8'b0011xxxx: BITOR = 1;
        8'b100100xx: JUMPZ = 1;
        8'b100101xx: JUMPNZ = 1;
        8'b100110xx: JUMPC = 1;
        8'b100111xx: JUMPNC = 1;
        default: {ADD,LOAD,OUTPUT,INPUT,JUMPZ,JUMP,JUMPNZ,JUMPC,
           JUMPNC,SUB,BITAND,BITOR,MUL, DIV} = 14'b0000000000000;
      endcase
    else
      {ADD,LOAD,OUTPUT,INPUT,JUMPZ,JUMP,JUMPNZ,JUMPC,JUMPNC,SUB,
         BITAND,BITOR,MUL, DIV} = 14'b0000000000000;
  end
endmodule
```

3.7 Decoder (Control Unit)

The control unit, also known as the instruction decoder, plays a critical role in a processor by interpreting the instruction fetched from memory and generating the necessary control signals to direct the operation of various components. It takes a binary instruction (typically the opcode) as input and decodes it to determine which operation needs to be performed by the processor, such as arithmetic operations, memory access, or register manipulation.

The primary function of the control unit is to generate a set of control signals based on the decoded instruction. These control signals ensure that components like the ALU, registers, and memory interact correctly to execute the instruction. The unit essentially translates the binary instruction into actionable commands, enabling the processor to carry out its tasks.

Internally, the control unit operates using combinational logic that matches opcodes with corresponding control signals. This logic may involve decision structures that select the correct signal based on the instruction type. The decoder's outputs then enable or disable specific components in the processor, ensuring that the correct operation is executed for each instruction

```
1  module decoder (input [7:0] IR,
2                  input Carry, Zero, CLK, CE, CLR,
3                  output reg RAM, ALU_S4, ALU_S3, ALU_S2, ALU_S1,
                      ALU_S0, MUXA, MUXB, MUXC, EN_IN, EN_DA, EN_PC
                      );
4
5    wire fetch, decode, execute, increment, carry_reg, zero_reg,
       add, load, instr_output, instr_input, jumpz, jump, jumpnz,
       jumpc, jumpnc, sub, bitand, bitor, jump_not_taken, mul;
6
7    reg en_st, OR5, FDC_D_B4_INV;
8
9    // One hot encoded ring oscillator for processor state
10   sequence_generator sequence_generator(.CLK(CLK), .CE(CE), .CLR(
       CLR), .F(fetch), .D(decode), .E(execute), .I(increment));
11
12   // 2 bit register for zero/carry
13   register_2 zero_carry(.D({Zero,Carry}), .CLK(CLK), .CE(en_st),
       .CLR(CLR), .Q({zero_reg,carry_reg}));
14
15   // place instruction decoder module with explicit port
       connnections
16   instruction_decoder instruction_decoder(.A(IR), .DECODE(decode)
       , .EXECUTE(execute), .ADD(add), .LOAD(load), .OUTPUT(
       instr_output), .INPUT(instr_input), .JUMPZ(jumpz), .JUMP(
       jump), .JUMPNZ(jumpnz), .JUMPC(jumpc), .JUMPNC(jumpnc), .SUB
       (sub), .BITAND(bitand), .BITOR(bitor), .MUL(mul), .DIV(div))
       ;
17
18
```

```
19    // jump not taken register
20    dff jump_n_taken(.D(~FDC_D_B4_INV), .CE(1'b1), .CLR(CLR), .C(
          CLK), .Q(jump_not_taken));
21
22    always @(*) begin    // combinational logic
23      en_st = (add|sub|bitand|bitor|mul|div);
24      RAM = (execute&instr_output);
25      OR5 = (jump|jumpz|jumpnz|jumpc|jumpnc);
26      ALU_S0 = (OR5|load|instr_input|bitand|bitor);
27      ALU_S1 = (OR5|instr_output|instr_input|load|mul|div);
28      ALU_S2 = (increment|sub|mul|bitor);
29      ALU_S3 = (sub|mul|div);
30      ALU_S4 = increment;
31      MUXA = increment;
32      MUXB = (load|add|bitand|bitor|sub|mul|div);
33      MUXC = (instr_input|instr_output);
34      EN_IN = fetch;
35      EN_DA = (execute&(load|add|sub|bitand|bitor|mul|div|
            instr_input));
36      FDC_D_B4_INV = ((jumpz&zero_reg)|(jumpnz&(~zero_reg))|(jumpc&
            carry_reg)|(jumpnc&(~carry_reg))|jump);
37      EN_PC = ((increment&jump_not_taken)|(execute&FDC_D_B4_INV));
38    end
39  endmodule
```

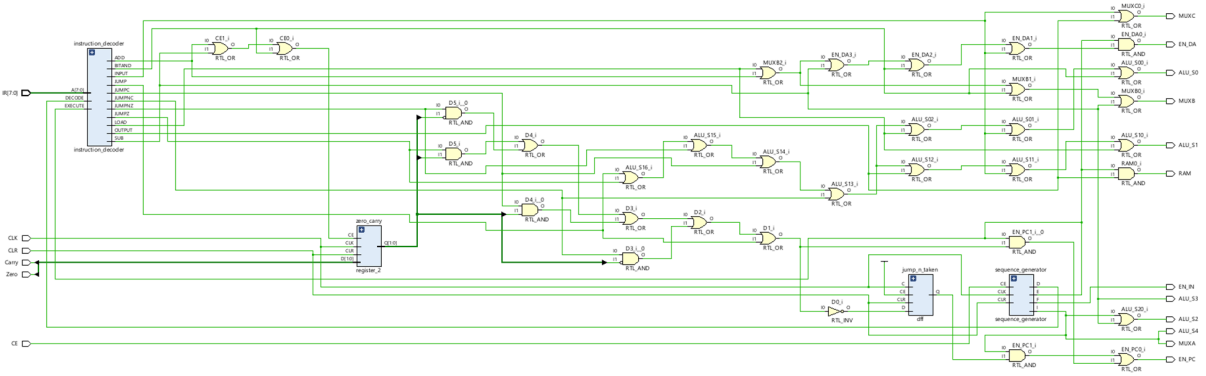

Figure 10: Decoder (Schematic)

3.8 Complete CPU design

The functioning of the CPU revolves around the execution of machine instructions, which are fetched from memory and processed in a sequential manner. Upon fetching an instruction, the control unit decodes it and generates control signals to orchestrate the interaction between the ALU, registers, and memory. The ALU performs the operation specified by the instruction, and the result is stored in a register or memory. The CPU also utilizes buses to transfer data between components, ensuring smooth communication. By continuously fetching, decoding, and executing instructions, the CPU enables the completion of complex tasks in a computer system.

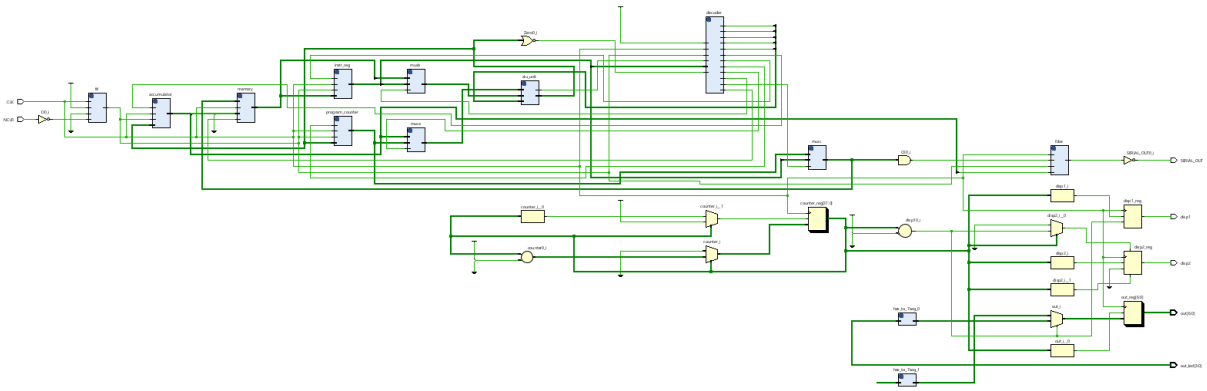

Figure 11: CPU (Schematic)

4 Testing and Results

4.1 Test Program

The CPU was tested using a simple program to calculate the product of two numbers and store output at [255] memory location:

```
mem[8'h00] = {LOAD, 8'h03}; // Load 3 into accumulator
mem[8'h01] = {MUL, 8'h05};  // Multiply by 5
mem[8'h02] = {OUTPUT, 8'hFF};       // Output result
```

4.2 Simulation Results

The waveform below demonstrates the correct execution of the instructions.

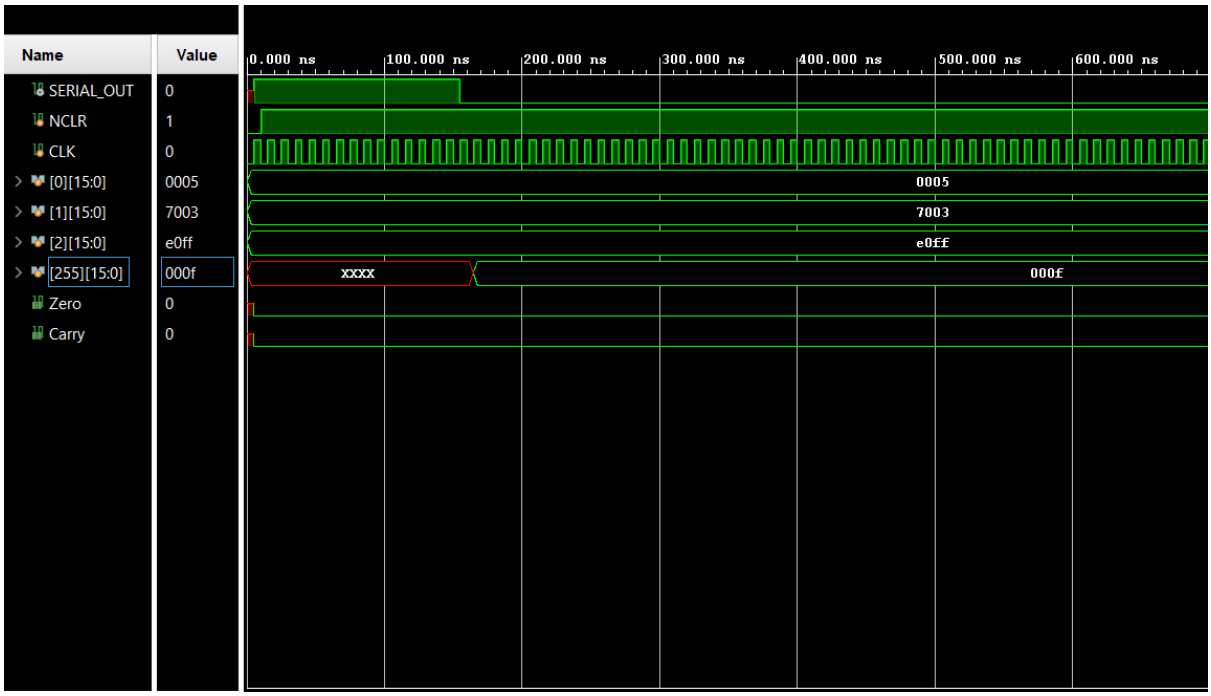

Figure 12: Simulation Waveform

The CPU was successfully implemented on FPGA (PYNQ Z2). Output taken at four in-built LEDs & on two external 7-segment Display. Below is the result of the same test program:

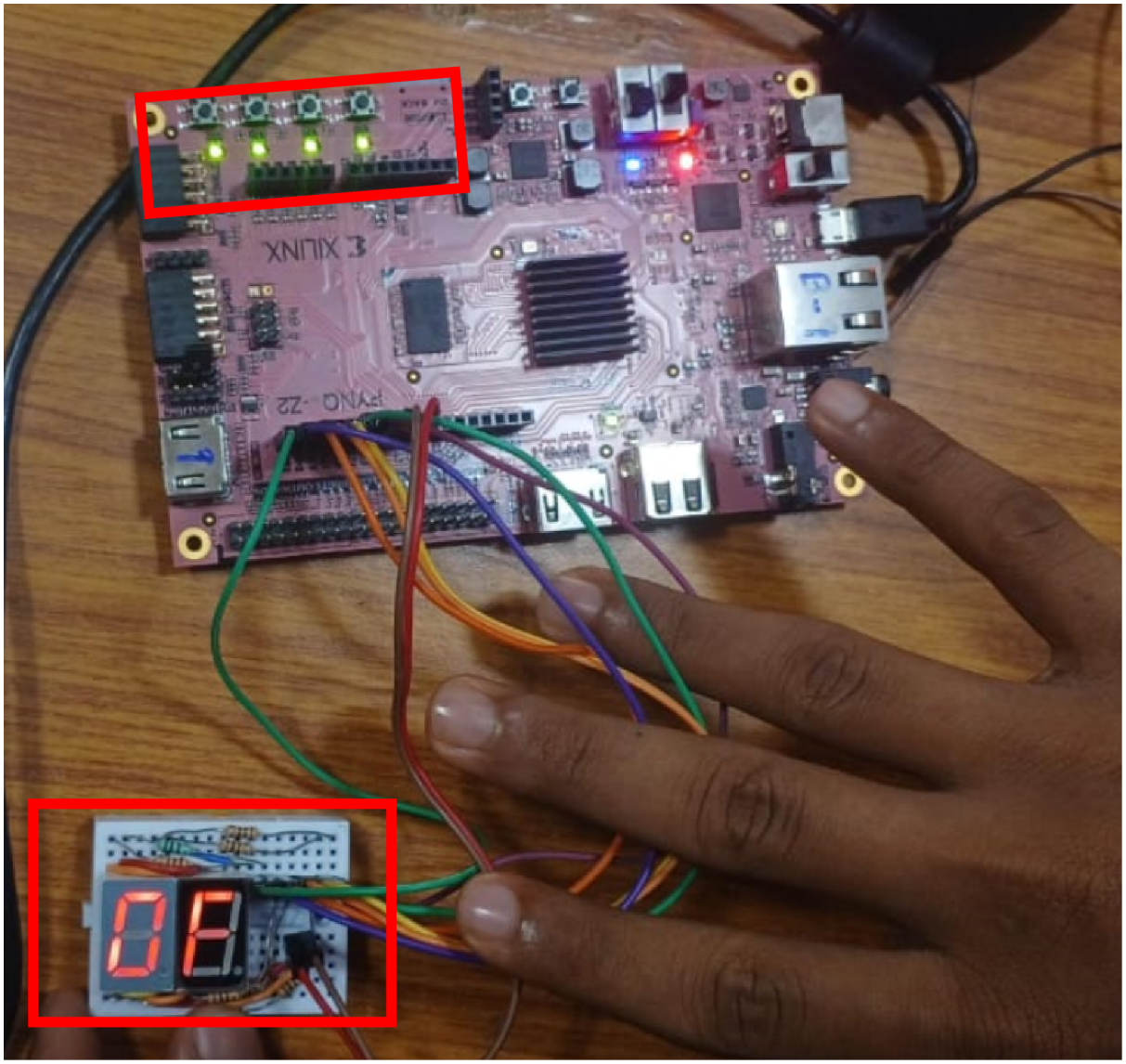

Figure 13: FPGA Implementation

5 Souce Code

Complete design code link: https://github.com/saurabhkr132/8-bit-processor

6 Conclusion

The design and development of this 8-bit CPU offer a comprehensive demonstration of basic computer architecture, focusing on essential components such as the arithmetic logic unit (ALU), registers, control unit, and memory. The CPU's ability to fetch, decode, and execute instructions efficiently showcases the fundamental processes involved in computation. The project highlights the interaction between these components, providing insights into the structure and functioning of a simple processor.

Looking ahead, this design can be expanded and optimized for more complex applications. Future improvements may involve increasing the bit-width of the processor to enhance data handling capabilities, incorporating more sophisticated instruction sets, and refining the control unit for better performance. Additionally, integrating features like pipelining, parallel processing, and low-power design techniques can further enhance the efficiency and scalability of the processor, making it suitable for a broader range of applications in embedded systems and IoT devices.

7 References

1. William Stallings *Computer Organization and Design: Designing for Performance.*
2. Hennessy, J., & Patterson, D. *Computer Organization and Design: The Hardware/-Software Interface.*
3. Xilinx Vivado User Guide.
4. Verilog HDL by Samir Palnitkar.