FACULTY OF ELECTRONIC ENGINEERING & TECHNOLOGY

NMJ31804

PRINCIPLES OF COMPUTER ARCHITECTURE

16-BIT CENTRAL PROCESSING UNIT

PROJECT REPORT

by

YEANAT HOSSAIN NELOY (211020050-5)

TABLE OF CONTENT

1.0 INTRODUCTION

We designed a simple 16-bit CPU that consists of Datapath unit and Control unit in our project. The data path is the collection of functional units such as multiplexers and arithmetic logic units. To carry out data processing tasks, the data path is necessary. Control Unit which we have known as Computer's central processing unit (CPU) which directs to the operation of the processor.

The control unit is an essential component inside the CPU that guides inputs and signals to the correct paths and components for their intended functions. It generates signals and coordinates the movement of data between different units within the processor. The control unit interprets instructions and controls the flow of data, converting external commands into control signals. It manages various execution units like the ALU, data buffers, and registers. Ultimately, the control unit ensures that the CPU responds correctly to instructions by directing signals to the appropriate components.

For our CPU's instruction set, it allows for data transfer between registers, performing arithmetic operations such as addition, subtraction, and multiplication, and executing logical operations for comparisons and bit manipulation. The addressing modes implemented are register addressing, enabling direct data transfer between registers, and immediate addressing, allowing for immediate loading of data into registers. These features provide flexibility and efficiency in data processing and manipulation within our CPU design.

By combining the Datapath unit, Control unit, ROM, and RAM, our CPU design enabled data processing, instruction execution, and storage functionalities. ROMs are the devices that used to store the information permanently while the RAM is the memory cells that are used to store or retrieve data. Thus, in our design, we included ROM and RAM to store the instruction set or the opcode and to store the data respectively.

2.0 FUNCTIONAL STRUCTURE OF 16-BIT CPU

2.1 Register Transfer Level Diagram

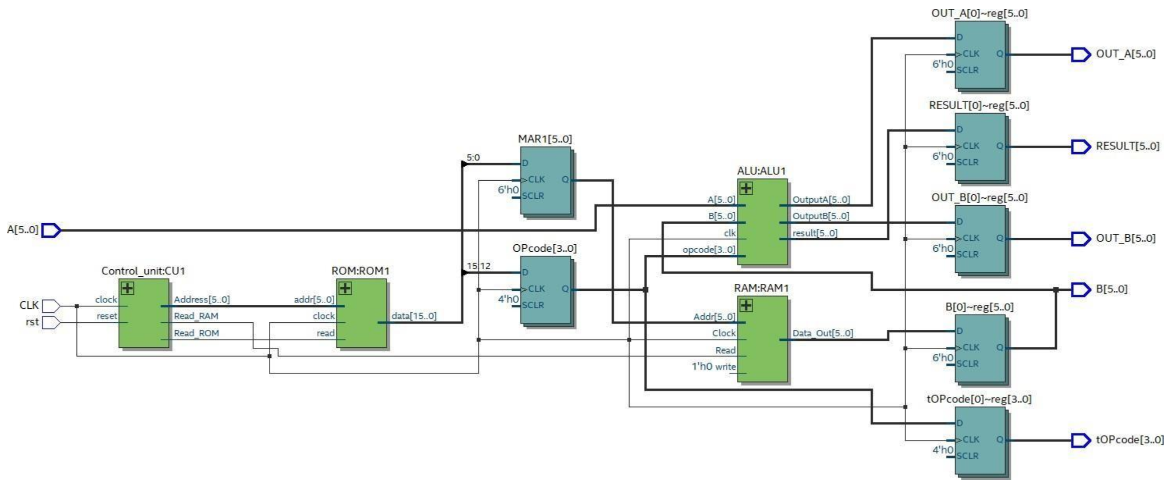


Figure 2.1: RTL Diagram

2.2 Block Diagram of CPU

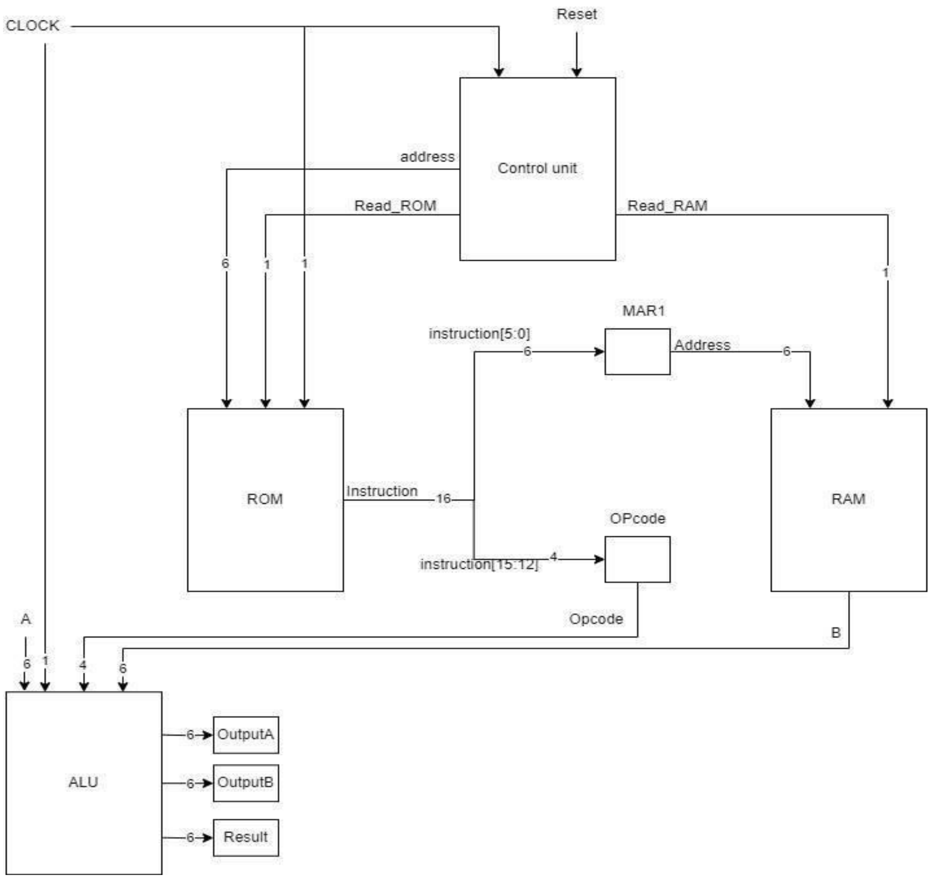


Figure 1.2: Block Diagram of CPU

2.3 Specifications of CPU Components/Modules

6) Central Processing Unit (CPU)

Module Name	CPU
Inputs	CLK, rst, A
Outputs	B, tOPcode, RESULT, OUT_A, OUT_B
Functionality	Implements a CPU with control, memory and arithmetic components

The CPU module acts as the central processing unit of a computer system, carrying out important tasks such fetching instructions, accessing memory, and completing computations. The CPU interacts with many crucial components, such as a control unit, ROM, RAM, and ALU. Together, these units work harmoniously to enable instruction execution, memory operations, and arithmetic/logical operations.

Signal	**Direction**	**Bit Width**	**Description**
CLK	Input	1	Clock input
rst	Input	1	Reset input
A	Input	6	Input bus
Opcode	Internal	4	Register for opcode
MAR1	Internal	6	Register for memory address
PC	Internal	6	Program Counter (PC)
instruction	Internal	16	Instruction from ROM
MBR1	Internal	6	Memory Buffer Register (MBR)
Res	Internal	6	ALU result
OUTPUTA	Internal	6	Output from ALU (A)
OUTPUTB	Internal	6	Output from ALU (B)
C_ROM	Internal	1	Control signal for reading from ROM
C_RAM	Internal	1	Control signal for reading from RAM
B	Output	6	Output register (B)
tOPcode	Output	4	Output register (Opcode)
RESULT	Output	6	Output register (Result)
OUT_A	Output	6	Output register (OUT_A)
OUT_B	Output	6	Output register (OUT_B)

b) Arithmetic Logic Unit (ALU)

Module Name	ALU
Inputs	A, B, clk, opcode
Outputs	Result, A_out, B_out
Functionality	Performs arithmetic and logical operations based on the opcode

The ALU is a crucial component of a CPU responsible for performing arithmetic and logical operations on data. This ALU able to create a basic ALU capable of executing operations such as addition, subtraction, multiplication, bitwise AND, bitwise OR,, and data movement.

The module requires the inputs A and B as well as an opcode designating the intended operation. With an output of a result, it runs on a clock signal (clk). During each clock cycle, the ALU executes the operation specified by the opcode and updates the result accordingly.

The ALU utilizes a case statement to determine the operation to be performed based on the opcode value. For example, if the opcode is 4'h1, the ALU performs addition by adding values A and B. Similarly, different cases handle subtraction, multiplication, AND, OR, and NOT operations. The ALU also includes cases for moving the result to B_out or moving the value of B to A_out.

Signal	Direction	Bit Width	Description
A	Input	6	Input bus
B	Input	6	Input bus
Clk	Input	1	Clock input
opcode	Input	4	Operation code input
result	Output	6	Result output register
OutputA	Output	6	Output register (A)
OutputB	Output	6	Output register (B)

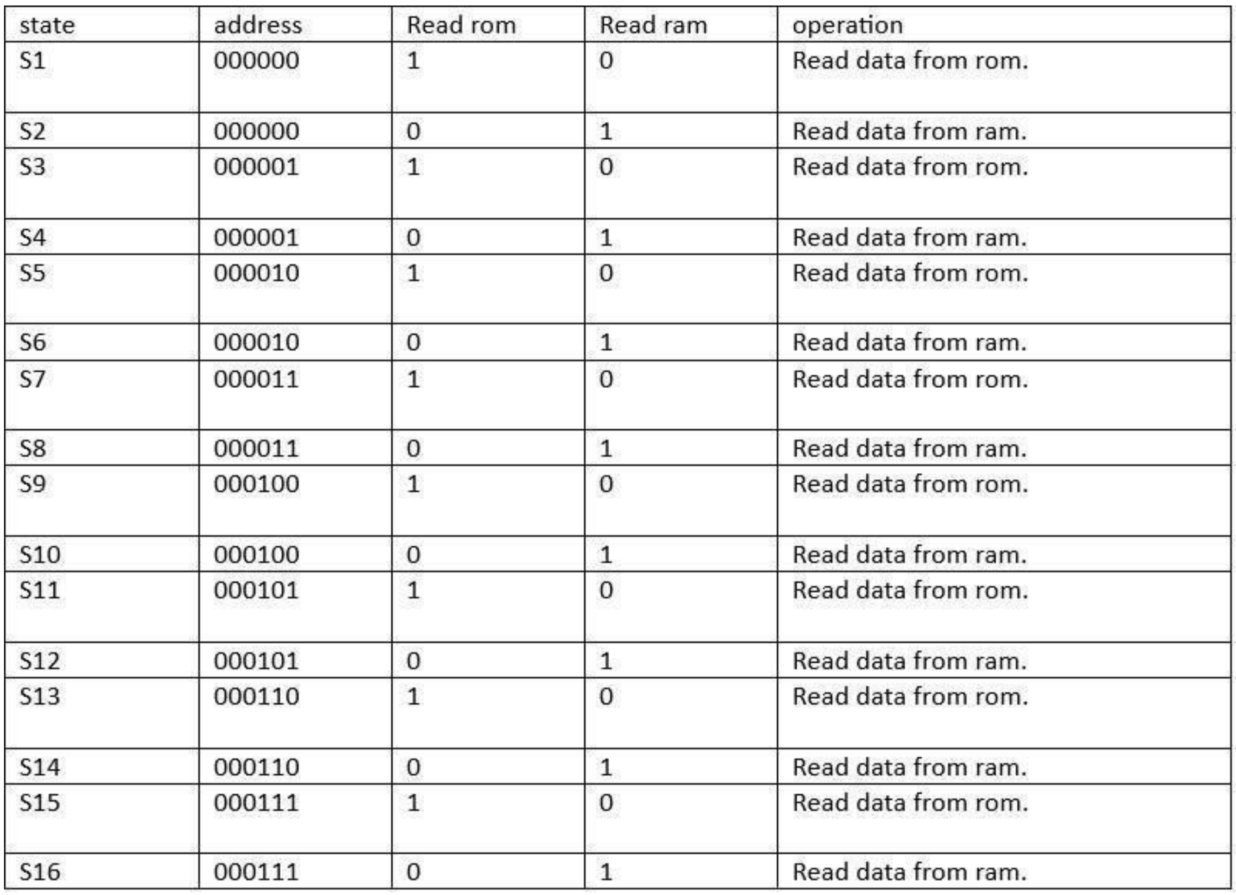
c) Control Unit

Module Name	Control Unit
Inputs	clock, reset
Outputs	Address, Read_ROM, Read_RAM
Internal Signals	y_present, y_next
Constants	s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s16
Registers	Address, Read_ROM, Read_RAM
Functionality	Controls the sequencing and control signals for ROM and RAM operations

To coordinate the actions of other CPU modules, the control unit generates control signals. This control unit may be used to build a basic control system that can manage the retrieval of data from RAM and instructions from ROM.

Signal	Direction	Bit Width	Description
clock	Input	1	Clock input
reset	Input	1	Reset input
Address	Output	6	Address output bus
Read_ROM	Output	1	ROM read control signal
Read_RAM	Output	1	RAM read control signal
y_present	Internal	4	Present state register
y_next	Internal	4	Next state register
s1,s2,…s16	Constants	4	State values
Address	Register	6	Address register
Read_ROM	Register	1	ROM read control register
Read_RAM	Register	1	RAM read control register

The control unit operates based on a clock signal and a reset input. To control the actions of the CPU, it has a state machine that switches between several states. Each state corresponds to a certain operation, such as accessing information in RAM or reading instructions from ROM. For the ROM and RAM modules to communicate with each other properly, the control unit establishes the necessary control signals, such as the address and read signals. The present state and control signal generated are shown in table below:



state	address	Read rom	Read ram	operation
S1	000000	1	0	Read data from rom.
S2	000000	0	1	Read data from ram.
S3	000001	1	0	Read data from rom.
S4	000001	0	1	Read data from ram.
S5	000010	1	0	Read data from rom.
S6	000010	0	1	Read data from ram.
S7	000011	1	0	Read data from rom.
S8	000011	0	1	Read data from ram.
S9	000100	1	0	Read data from rom.
S10	000100	0	1	Read data from ram.
S11	000101	1	0	Read data from rom.
S12	000101	0	1	Read data from ram.
S13	000110	1	0	Read data from rom.
S14	000110	0	1	Read data from ram.
S15	000111	1	0	Read data from rom.
S16	000111	0	1	Read data from ram.

d) Random Access Memory (RAM)

Module Name	RAM
Inputs	Clock, write, Read, Addr
Outputs	Data_Out
Functionality	Implements a RAM module for data storage

RAM (Random Access Memory) operates based on clock signals and control inputs for read and write operations. The module accepts a memory address as input and, if the read control signal is active, during a positive clock edge, extracts the associated data from the RAM. The module verifies the memory address when a read activity is detected and assigns the recovered data to the output port. The module keeps the output data in an undefinable state during write operations. This RAM module may be customised and extended to suit more advanced read-write RAM designs.

Signal	Direction	Bit Width	Description
Clock	Input	1	Clock input
write	Input	1	Write control signal input
Read	Input	1	Read control signal input
Addr	Input	6	Address input bus
Data_Out	Output	6	Data output register

e) Read-Only Memory (ROM)

Module Name	ROM
Inputs	Clock, read, addr
Outputs	Data
Functionality	Reads data from ROM based on the address

The ROM stores pre-defined data and retrieves it based on the given memory address when the read control signal is active. The module runs on a clock signal and accepts inputs for read control and memory address before generating the data that was retrieved. If the read control signal is on at each positive clock edge, the module uses a case statement to examine the memory address. The correct information will be assigned to the output port based on the address value. When a read control signal is not active, the output data is set to an undefinable state. This ROM module offers fundamental read-only memory capability and may be extended as necessary with extra information and addresses.

Signal	Direction	Bit Width	Description
Clock	Input	1	Clock input
read	Input	1	Read control signal input
addr	Input	6	Address input bus
Data	Output	16	Data output bus

3.0 EXPLAINATION OF THE FETCH-EXECUTION CYCLE

The instruction cycle, also known as the fetch-execute cycle, is the process that the central processing unit (CPU) follows from booting up until the computer shuts down in order to process instructions.

Fetch Cycle:

This cycle begins when the reset signal is high. The reset input in the control unit is active low, meaning that if it is low, the entire process will be reset. Therefore, when the reset signal is high, the system starts working by sending the address and activating the Read_ROM signal. This allows the system to fetch the instruction from the specified address of the ROM.

Decoding the instruction:

The fetched 16-bit instruction is decoded into three parts as follows:

1. Opcode: These are the first four most significant bits, which are stored in a four-bit register called "opcode." The opcode is then used to select the operation of the Arithmetic Logic Unit.
2. Ignored bits: The six bits following the opcode are ignored.
3. Address of value B: The last six least significant bits are stored in a six-bit register called "MAR1". The address in MAR1 is then used to fetch data B from the RAM.

Execution Cycle:

In the second state, which occurs after fetching the instruction, the control unit activates the Read_RAM signal. The address stored in MAR1 is sent to the RAM, and the data in that address is sent to Register B. The data in Register B is used as one operand in the Arithmetic Logic Unit (ALU), while the second operand (A) is received as an immediate input. Additionally, the opcode is sent to the ALU to determine the operation to be performed on the given operands. Finally, the result is stored in a register with the same name. Two additional registers, Out_A and Out_B, are used to store the results of data transfer operations.
This fetch-execution cycle is repeated with different addresses using two states: one to fetch the instruction and another to fetch the data of value B, until the reset signal is low.

3.1 Instruction Set Architecture

ALU Selection	ALU Operation
0001	Addition **(Arithmetic), stored in reg A.**
0010	Subtraction **(Arithmetic), stored in result**
0011	Multiplication **(Arithmetic), stored in result**
0100	AND gate **(Logic), stored in result**
0101	OR gate **(Logic), stored in result**
0110	Move result to B_out
0111	Move value in register B to Register A **(Register Addressing)**
1000	Move value in register B directly from RAM to Register A **(Register Addressing)**

TABLE 3.1.1: List of instruction set based on opcode

The system employs a table that defines various operations, such as addition, subtraction, multiplication, AND gate, OR gate, and NOT gate, based on their corresponding opcode values. Some of these operations involve the interaction between the user-provided value A and the value B obtained from the RAM, which is stored in the "Result" register. Other operations are performed on the contents of the register A_out, which represents the value of A displayed in waveform format. Additionally, the system includes instructions for data movement between registers. This can be achieved by transferring the value from the "Result" register to register B using register addressing, or by copying the value from register B to register A. Moreover, one of the operations directly moves the value of register B to the A_out register.

4.0 RESULTS AND DISCUSSION

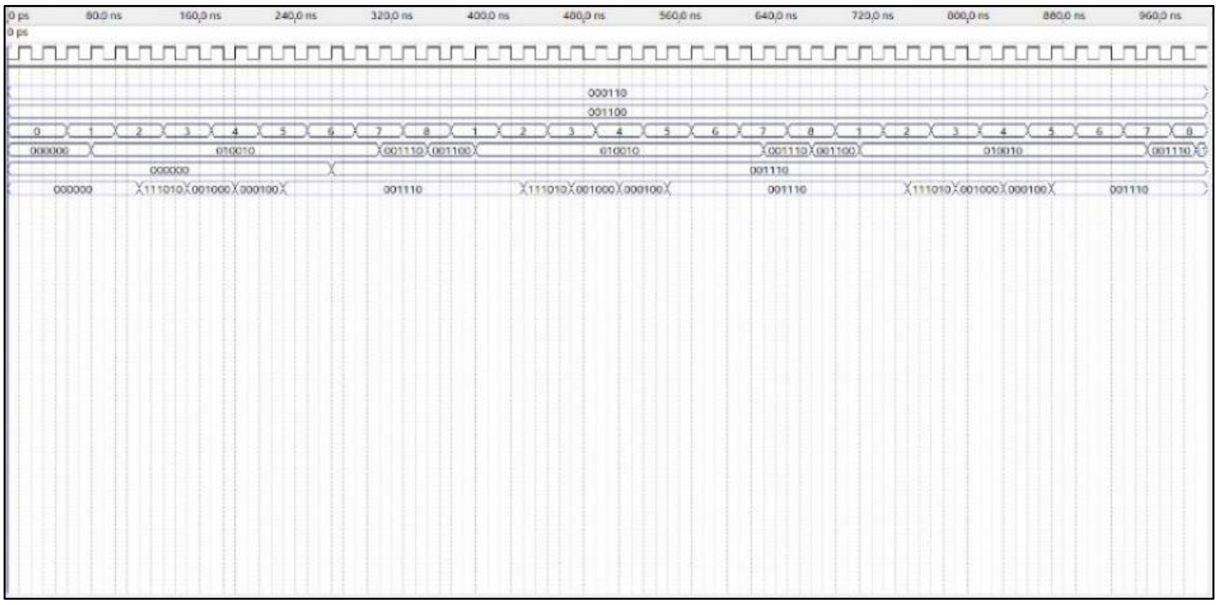


Figure 4.1: Waveform Result

The presented system comprises three input signals, namely clock (clk), reset (rst), and A, along with five output signals, including trace opcode (tOPcode), B, RESULT, OUT_A, and OUT_B. To activate the system, it is imperative to ensure that the active low reset signal (rst) remains inactive by maintaining a high state. Initialization of the system commences with opcode 0, which is deliberately devoid of any operations to prevent undesirable outcomes. For illustrative purposes, both operands, A and B, are fixed at specific values: A = 0001002 (410) and B = 0011002 (1210).

To execute the first opcode, the system necessitates three clock cycles for fetching, with each fetchexecution cycle spanning two clock cycles. Each clock cycle corresponds to the completion of one state within the control unit.

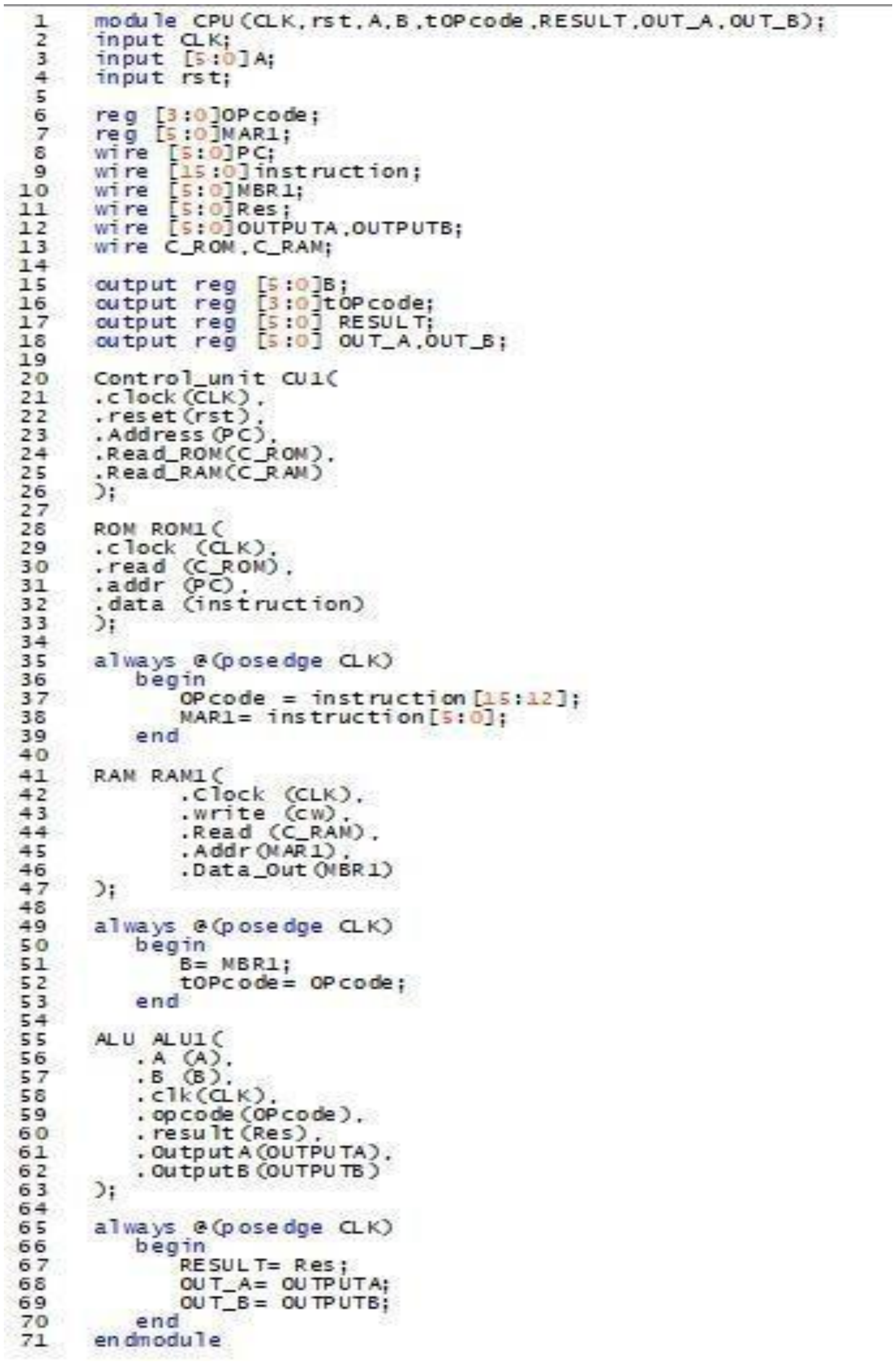
The initial three opcodes are designed to perform arithmetic operations. The first opcode, 00012, facilitates an addition process, resulting in 4 + 12 = 16 = 0100002, which is subsequently stored in register A. Subsequently, the second opcode, 00102, initiates a subtraction process, yielding 4 - 12 = -8 = 1110002 in two's complement form. Lastly, the third opcode, 00112, triggers a multiplication process, leading to 4 x 12 = 48 = 1100002.

The subsequent two opcodes are dedicated to logical operations. The fourth opcode, 01002, enables the system to perform an 'AND' operation, resulting in the output 0001002 through the bitwise operation 0001002 & 0011002. On the other hand, the fifth opcode, 01012, facilitates an 'OR' operation, generating the output 0011002 through the bitwise operation 0001002 | 0011002.

The final three opcodes are associated with data transfer operations. The sixth opcode, 01102, allows the system to transfer data from the result register to the B_out register, denoted as B_out <= RESULT. Subsequently, the seventh opcode, 01112, enables the transfer of data from the B_out register to the A_out register. Lastly, the last opcode, 10002, permits the system to receive immediate data from the RAM and transfer it to the A_out register, represented as A_out<=B.

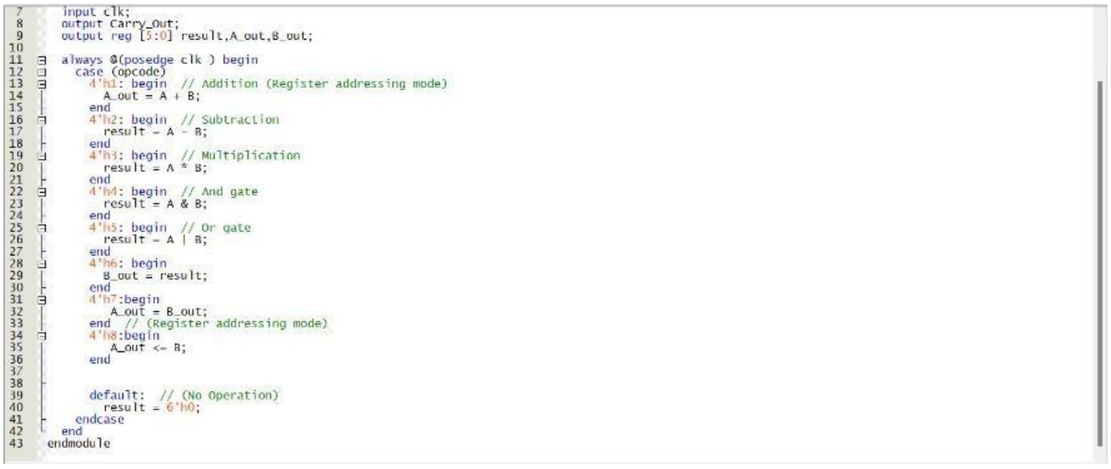
5.0 VERILOG CODE

5.1 Verilog Code for CPU



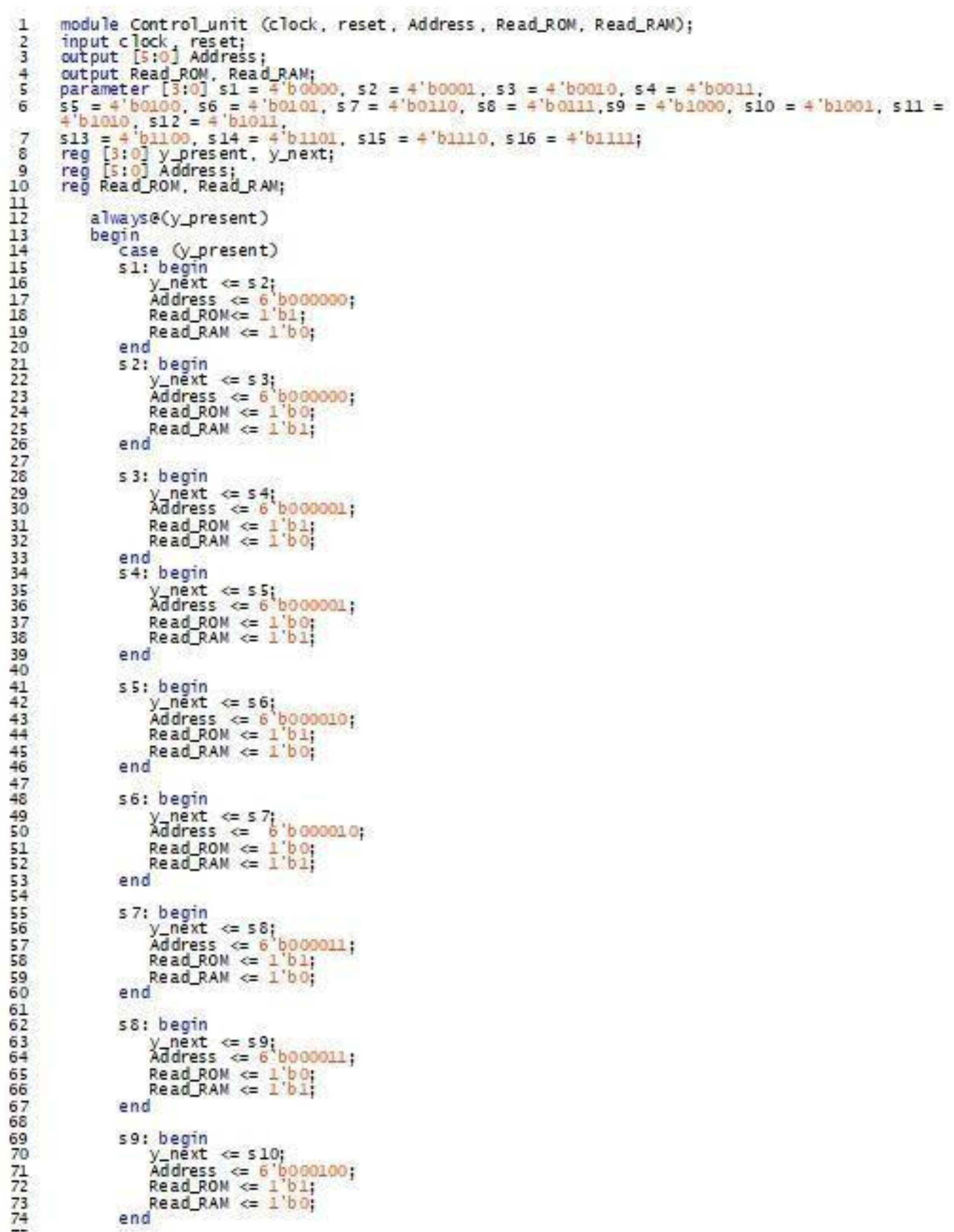
```
module CPU(CLK,rst,A,B,tOPcode,RESULT,OUT_A,OUT_B);
input CLK;
input [5:0]A;
input rst;

reg [3:0]OPcode;
reg [5:0]MAR1;
wire [5:0]PC;
wire [15:0]instruction;
wire [5:0]MBR1;
wire [5:0]Res;
wire [5:0]OUTPUTA,OUTPUTB;
wire C_ROM,C_RAM;

output reg [5:0]B;
output reg [3:0]tOPcode;
output reg [5:0] RESULT;
output reg [5:0] OUT_A,OUT_B;

Control_unit CU1(
.clock(CLK),
.reset(rst),
.Address(PC),
.Read_ROM(C_ROM),
.Read_RAM(C_RAM)
);

ROM ROM1(
.clock (CLK),
.read (C_ROM),
.addr (PC),
.data (instruction)
);

always @(posedge CLK)
   begin
       OPcode = instruction[15:12];
       MAR1= instruction[5:0];
   end

RAM RAM1(
       .Clock (CLK),
       .write (cw),
       .Read (C_RAM),
       .Addr(MAR1),
       .Data_Out(MBR1)
);

always @(posedge CLK)
   begin
       B= MBR1;
       tOPcode= OPcode;
   end

ALU ALU1(
   .A (A),
   .B (B),
   .clk(CLK),
   .opcode(OPcode),
   .result(Res),
   .OutputA(OUTPUTA),
   .OutputB(OUTPUTB)
);

always @(posedge CLK)
   begin
       RESULT= Res;
       OUT_A= OUTPUTA;
       OUT_B= OUTPUTB;
   end
endmodule
```

5.2 Verilog Code for ALU



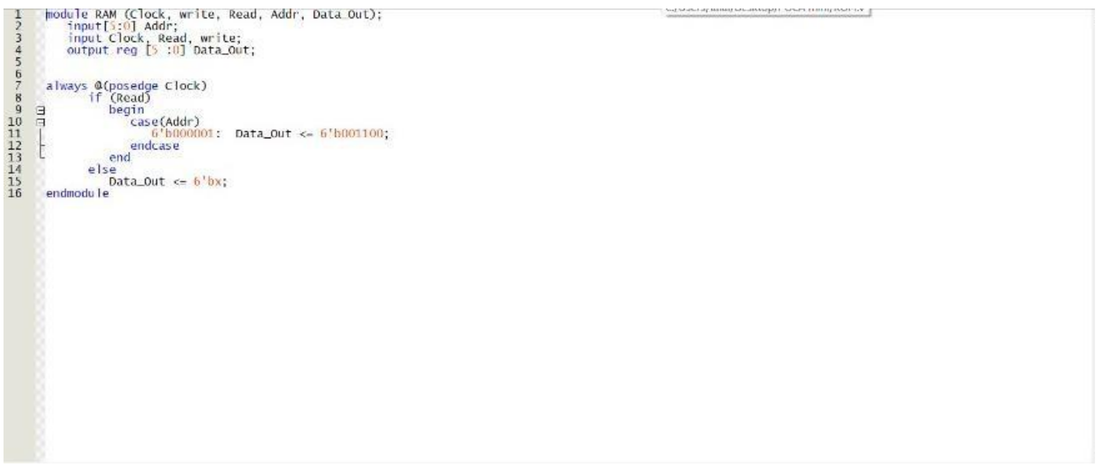
```verilog
    input clk;
    output Carry_Out;
    output reg [5:0] result,A_out,B_out;

    always @(posedge clk ) begin
      case (opcode)
        4'h1: begin  // Addition (Register addressing mode)
          A_out = A + B;
        end
        4'h2: begin  // Subtraction
          result = A - B;
        end
        4'h3: begin  // Multiplication
          result = A * B;
        end
        4'h4: begin  // And gate
          result = A & B;
        end
        4'h5: begin  // Or gate
          result = A | B;
        end
        4'h6: begin
          B_out = result;
        end
        4'h7:begin
           A_out = B_out;
        end  // (Register addressing mode)
        4'h8:begin
           A_out <= B;
        end


        default:  // (No Operation)
          result = 6'h0;
      endcase
    end
  endmodule
```

Verilog Code for Control Unit



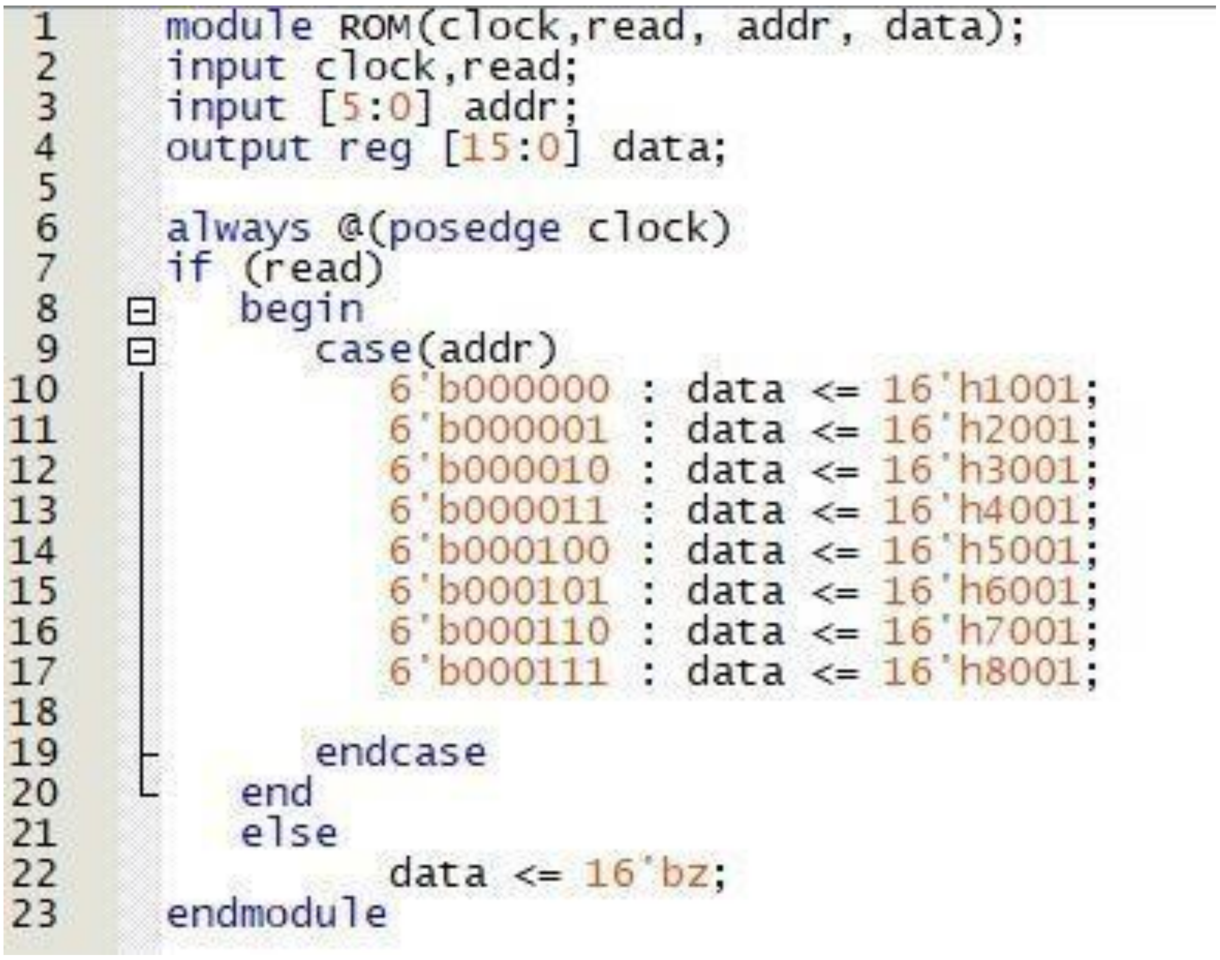
```
module Control_unit (clock, reset, Address, Read_ROM, Read_RAM);
input clock, reset;
output [5:0] Address;
output Read_ROM, Read_RAM;
parameter [3:0] s1 = 4'b0000, s2 = 4'b0001, s3 = 4'b0010, s4 = 4'b0011,
s5 = 4'b0100, s6 = 4'b0101, s7 = 4'b0110, s8 = 4'b0111,s9 = 4'b1000, s10 = 4'b1001, s11 =
4'b1010, s12 = 4'b1011,
s13 = 4'b1100, s14 = 4'b1101, s15 = 4'b1110, s16 = 4'b1111;
reg [3:0] y_present, y_next;
reg [5:0] Address;
reg Read_ROM, Read_RAM;

    always@(y_present)
    begin
       case (y_present)
       s1: begin
          y_next <= s2;
          Address <= 6'b000000;
          Read_ROM<= 1'b1;
          Read_RAM <= 1'b0;
       end
       s2: begin
          y_next <= s3;
          Address <= 6'b000000;
          Read_ROM <= 1'b0;
          Read_RAM <= 1'b1;
       end

       s3: begin
          y_next <= s4;
          Address <= 6'b000001;
          Read_ROM <= 1'b1;
          Read_RAM <= 1'b0;
       end
       s4: begin
          y_next <= s5;
          Address <= 6'b000001;
          Read_ROM <= 1'b0;
          Read_RAM <= 1'b1;
       end

       s5: begin
          y_next <= s6;
          Address <= 6'b000010;
          Read_ROM <= 1'b1;
          Read_RAM <= 1'b0;
       end

       s6: begin
          y_next <= s7;
          Address <=  6'b000010;
          Read_ROM <= 1'b0;
          Read_RAM <= 1'b1;
       end

       s7: begin
          y_next <= s8;
          Address <= 6'b000011;
          Read_ROM <= 1'b1;
          Read_RAM <= 1'b0;
       end

       s8: begin
          y_next <= s9;
          Address <= 6'b000011;
          Read_ROM <= 1'b0;
          Read_RAM <= 1'b1;
       end

       s9: begin
          y_next <= s10;
          Address <= 6'b000100;
          Read_ROM <= 1'b1;
          Read_RAM <= 1'b0;
       end
```

5.3 Verilog Code for RAM



```
module RAM (Clock, write, Read, Addr, Data_Out);
    input[5:0] Addr;
    input Clock, Read, write;
    output reg [5 :0] Data_Out;


always @(posedge Clock)
        if (Read)
            begin
                case(Addr)
                    6'b000001:  Data_Out <= 6'b001100;
                endcase
            end
        else
            Data_Out <= 6'bx;
endmodule
```

5.4 Verilog Code for ROM



```verilog
module ROM(clock,read, addr, data);
input clock,read;
input [5:0] addr;
output reg [15:0] data;

always @(posedge clock)
if (read)
   begin
      case(addr)
         6'b000000 : data <= 16'h1001;
         6'b000001 : data <= 16'h2001;
         6'b000010 : data <= 16'h3001;
         6'b000011 : data <= 16'h4001;
         6'b000100 : data <= 16'h5001;
         6'b000101 : data <= 16'h6001;
         6'b000110 : data <= 16'h7001;
         6'b000111 : data <= 16'h8001;

      endcase
   end
   else
         data <= 16'bz;
endmodule
```