College of Engineering and Applied Sciences

ECE 3570 Introduction to Computer Architecture

Project Assignment #4

Team Members

Raj Basnet

4 April 2020

Introduction

The primary objective of the lab is to design a 10-bit pipelined CPU for the deeper understanding of a multi-cycle architecture implementation and design tradeoffs. The project will provide an insight about the multi cycle architecture implementation for the 10-bit CPU created in the previous project. This lab helps to know effects of different design choices on the execution time, clock period time and hardware complexity. Furthermore, it will enhance the teamwork skills, critical skills and learning of Verilog hardware description language.

Summary of Instructions

1. **load instruction (000)**: This instruction loads the contents of the memory address into the register. ORI Type and Immediate Addressing mode. For example,

   load $R0, M0 means **$R0 <- M[M0]** M0 = immediate memory address 0

load (000)	$R0 (010)	M0 (0000)

2. **store instruction (001)**: This instruction stores the contents of register into the memory address. ORI Type and Immediate Addressing mode. For example,

   store $R7, M1 means **$R0 <- M[M1]** M1 = immediate memory address 1

store (001)	$R7 (111)	M1 (0001)

3. **addi instruction (010):** This instruction adds a constant or immediate address to the register and saves the result into the register. ORI Type and Immediate Addressing mode. For example,

   addi $R5, 1 means **$R5 <- $R5 + 1, addi $R5, M2** means **$R5 <- $R5 + M2**

addi (010)	$R5 (110)	1 or M2 (0001) or (0010)

4. **add instruction (011):** This instruction adds the values of two registers and saves the result into the first register mentioned. ORR Type and Register Addressing. For example,

   add $R3, $R7 means **$R3 <- $R3 + $R7**

add (011)	$R3 (0101)	R7 (0111)

5. **grt instruction (100):** It checks if first register value is greater than second register value and changes the reg_c0 flag accordingly. Register Addressing (ORR type). For example,

   grt $R0, $R1 if $R0 > $R1, then $C0 = 1 else $C0 = 0

grt (100)	$R0 (010)	$R1 (0011)

6. **beq instruction (101):** It branches to mentioned address if the reg_c0 flag is 0. Branch Addressing and OCA type For example,

   beq loop means **Branch to address denoted by loop if $C0 = 0**

beq (101)	loop (_ _ _ _ _ _ _ _)

7. **bne instruction (110):** It branches to mentioned address if the reg_c0 flag is 1. Branch Addressing and OCA type. For example,

   bne loop1 means **Branch to address denoted by loop1 if $C0 = 1**

beq (101)	loop (_ _ _ _ _ _ _ _)

8. **halt instruction (111):** It denotes the end of the program and it uses H type of addressing mode. For example,

   halt takes to the end of the program

opcode (111)	Remaining bits (XXXXXXX)

Detail Description of Component Design and Modules

1. **ALU Module**

    ALU module consists of two arithmetic operators in the gate level logic. One of the them is the 10-bit adder which adds two inputs and provides the sum output as well as carry out. Another operator is greater than operator/instruction which checks if the first input is greater than the second input and sets the C0 output to be 1 or 0 accordingly. The multiplexer is used to provide an output either from the result of adder or grt instruction which is controlled by the ALU control unit.

    ALU Code

```
`timescale 1ns / 1ps

module ALU_module(input ALU_Control, input [9:0] A, input [9:0] B, output wire [9:0]
ALU_output);

        reg Cin;
        initial begin
            Cin = 0;
        end

        wire Cout, C0;
        wire [9:0] Sum;

        adder_10 ALU1(A, B, Cin, Sum, Cout);
        grt_10 ALU2(A, B, C0);

        mux ALU_mux(.A1(Sum), .A2({000000000,C0}), .Sel(ALU_Control),
.Y(ALU_output));
endmodule

module mux(input [9:0] A1, input [9:0] A2, input Sel, output reg [9:0] Y);

    always @(Sel, A1, A2)
        begin
            if (Sel == 0)
            begin
                Y <= A1;
            end
            else if (Sel == 1)
            begin
                Y <= A2;
            end
        end
endmodule

module adder(input A, input B, input Cin, output Sum, output Cout);

        wire w1, w2, w3;
        and(w1, A, B);
        and(w2, A, Cin);
        and (w3, B, Cin);
        or(Cout, w1, w2, w3);
        xor(Sum, A, B, Cin);

endmodule


module adder_10(input [9:0] A, input [9:0] B, input Cin, output [9:0] Sum, output Cout);

        wire [9:1] C;
        adder adder0 (A[0], B[0], Cin, Sum[0], C[1]),
              adder1 (A[1], B[1], C[1], Sum[1], C[2]),
              adder2 (A[2], B[2], C[2], Sum[2], C[3]),
              adder3 (A[3], B[3], C[3], Sum[3], C[4]),
              adder4 (A[4], B[4], C[4], Sum[4], C[5]),
              adder5 (A[5], B[5], C[5], Sum[5], C[6]),
              adder6 (A[6], B[6], C[6], Sum[6], C[7]),
              adder7 (A[7], B[7], C[7], Sum[7], C[8]),
              adder8 (A[8], B[8], C[8], Sum[8], C[9]),
              adder9 (A[9], B[9], C[9], Sum[9], Cout);

endmodule

module grt_10(input [9:0] A, input [9:0] B, output C0);

    wire G1, G2;
    wire x1, x2, x3, x4;
    wire S1, S2, S3, S4, S2_;
    wire A9_, B9_;

    assign G1 = A > B;
    assign G2 = A < B;
    not(A9_, A[9]);
    not(B9_, B[9]);

    and(S1, A[9], B[9]);
    or(S2, A[9], B[9]);
    and(S3, A[9], B9_);
    and(S4, A9_, B[9]);

    not(S2_, S2);

    and(x1, S2_, G1);
    and(x2, S1, G2);
    and(x3, S3, 0);
    and(x4, S4, 1);

    or(C0, x1, x2, x3, x4);

endmodule
```

ALU Testbench

```
`timescale 1ns / 1ps

module ALU_test();

reg ALU_Control; reg [9:0] A; reg [9:0] B;
wire [9:0] ALU_output;

    ALU_module ALU_test(.ALU_Control(ALU_Control), .A(A), .B(B),
.ALU_output(ALU_output));

    initial
        begin

            A = 10'b0011010111;
            B = 10'b0000101110;
            ALU_Control = 0;

            #100
            A = 10'b1110101110;
            B = 10'b0100111110;
            ALU_Control = 0;

            #100
            A = 10'b1110101110;
            B = 10'b1100111110;
            ALU_Control = 0;

            #100
            A = 10'b1010101010;
            B = 10'b1100111110;
            ALU_Control = 1;

            #100
            A = 10'b0011001010;
            B = 10'b0000110101;
            ALU_Control = 1;

            #100
            A = 10'b1110101110;
            B = 10'b0100111110;
            ALU_Control = 1;

        end
endmodule
```

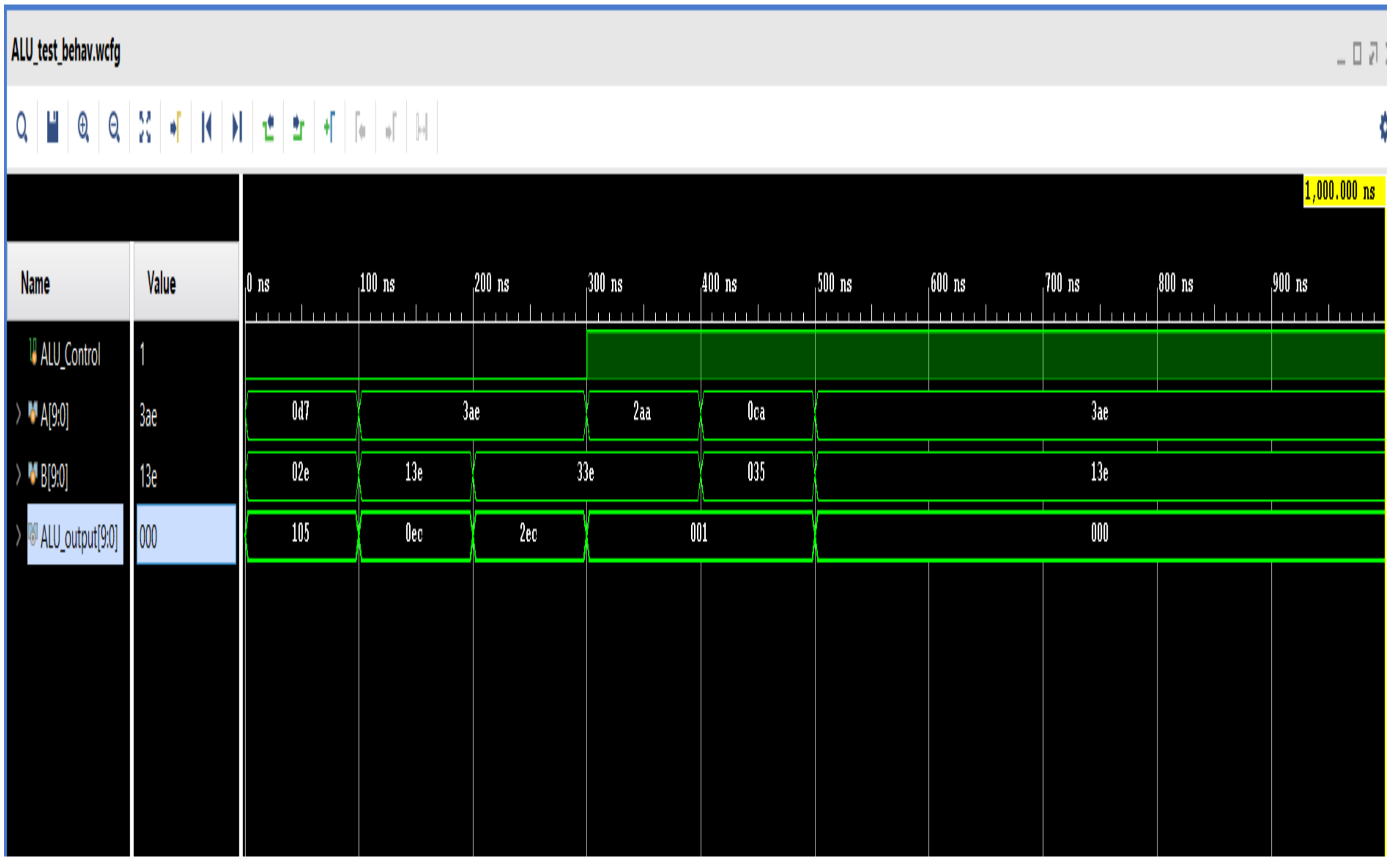


Figure 1: Timing Diagram for ALU_module (ALU_Control = 0 for adder and ALU_Control = 1 for grt instruction)

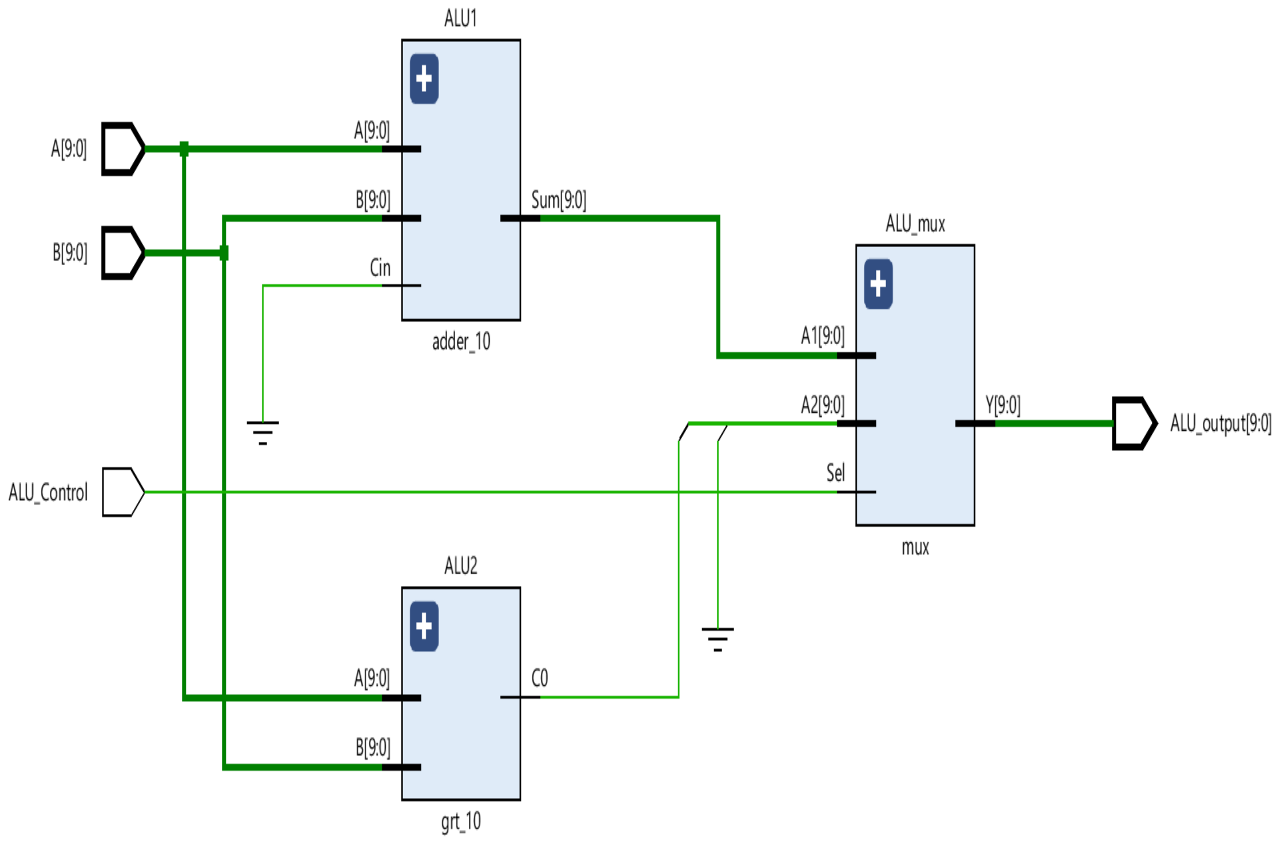
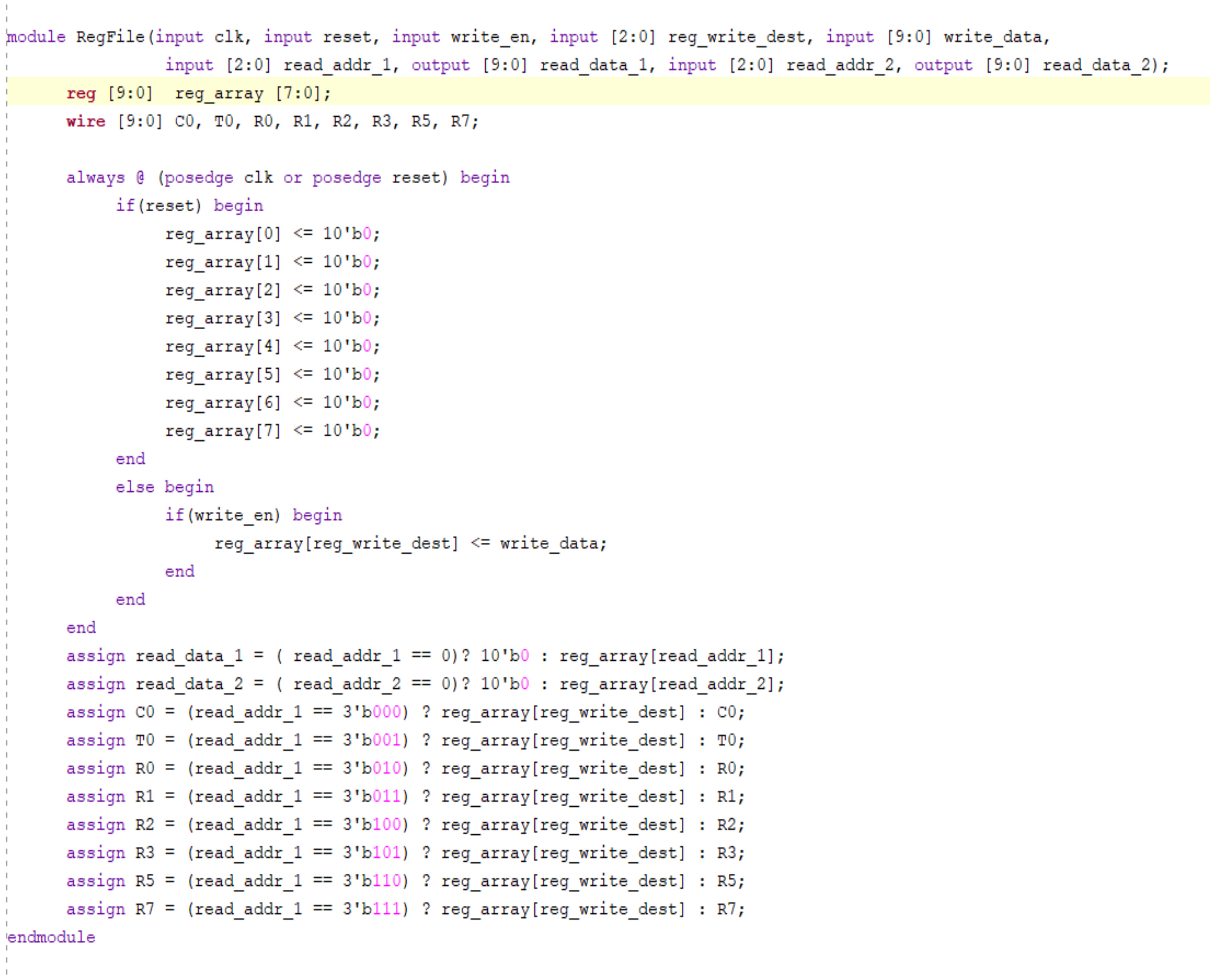


Figure 2: Schematic for ALU_module

2. Register File

The register file has been changed from the previous project to meet the requirement for creating the 10-bit CPU. With one write data input with its enable and two read data outputs with their input read data address, clock and reset signal, the register file is created using array of registers which contains 8 registers each of 10 bits. These registers are C0, T0, R0, R1, R2, R3, R5 and R5 with their respective read address 000, 001, 010, 011, 100, 101, 110 and 111.



```
module RegFile(input clk, input reset, input write_en, input [2:0] reg_write_dest, input [9:0] write_data,
                input [2:0] read_addr_1, output [9:0] read_data_1, input [2:0] read_addr_2, output [9:0] read_data_2);
      reg [9:0]  reg_array [7:0];
      wire [9:0] C0, T0, R0, R1, R2, R3, R5, R7;

      always @ (posedge clk or posedge reset) begin
           if(reset) begin
                reg_array[0] <= 10'b0;
                reg_array[1] <= 10'b0;
                reg_array[2] <= 10'b0;
                reg_array[3] <= 10'b0;
                reg_array[4] <= 10'b0;
                reg_array[5] <= 10'b0;
                reg_array[6] <= 10'b0;
                reg_array[7] <= 10'b0;
           end
           else begin
                if(write_en) begin
                     reg_array[reg_write_dest] <= write_data;
                end
           end
      end
      assign read_data_1 = ( read_addr_1 == 0)? 10'b0 : reg_array[read_addr_1];
      assign read_data_2 = ( read_addr_2 == 0)? 10'b0 : reg_array[read_addr_2];
      assign C0 = (read_addr_1 == 3'b000) ? reg_array[reg_write_dest] : C0;
      assign T0 = (read_addr_1 == 3'b001) ? reg_array[reg_write_dest] : T0;
      assign R0 = (read_addr_1 == 3'b010) ? reg_array[reg_write_dest] : R0;
      assign R1 = (read_addr_1 == 3'b011) ? reg_array[reg_write_dest] : R1;
      assign R2 = (read_addr_1 == 3'b100) ? reg_array[reg_write_dest] : R2;
      assign R3 = (read_addr_1 == 3'b101) ? reg_array[reg_write_dest] : R3;
      assign R5 = (read_addr_1 == 3'b110) ? reg_array[reg_write_dest] : R5;
      assign R7 = (read_addr_1 == 3'b111) ? reg_array[reg_write_dest] : R7;
endmodule
```

Figure 3: Verilog code for Register File

```
`timescale 1ns / 1ps

module RegFile_test();
    reg clk, reset, write_en;
    reg [2:0] reg_write_dest, read_addr_1, read_addr_2;
    reg [9:0] write_data;
    wire [9:0] read_data_1, read_data_2;

    RegFile RegFile_test(.clk(clk), .reset(reset), .write_en(write_en),
.reg_write_dest(reg_write_dest), .write_data(write_data),
                            .read_addr_1(read_addr_1), .read_addr_2(read_addr_2),
.read_data_1(read_data_1), .read_data_2(read_data_2));

    initial begin
        clk = 1;
        reset = 1;
        write_en = 0;

        #100
        reset = 0;
        write_en = 1;
        reg_write_dest = 3'b000;
        write_data = 10'b0010100101;
        read_addr_1 = 3'b000;
        read_addr_2 = 3'b000;

        #100
        write_en = 1;
        reg_write_dest = 3'b001;
        write_data = 10'b0011000011;
        read_addr_1 = 3'b111;
        read_addr_2 = 3'b001;

        #100
        write_en = 1;
        reg_write_dest = 3'b010;
        write_data = 10'b1111000011;
        read_addr_1 = 3'b110;
        read_addr_2 = 3'b010;

        #100
        write_en = 1;
        reg_write_dest = 3'b011;
        write_data = 10'b0000000011;
        read_addr_1 = 3'b011;
        read_addr_2 = 3'b100;
        #100
        write_en = 1;
        reg_write_dest = 3'b100;
        write_data = 10'b0011000011;
        read_addr_1 = 3'b100;
        read_addr_2 = 3'b011;

        #100
        write_en = 1;
        reg_write_dest = 3'b101;
        write_data = 10'b0011000011;
        read_addr_1 = 3'b011;
        read_addr_2 = 3'b101;

        #100
        write_en = 1;
        reg_write_dest = 3'b110;
        write_data = 10'b0011000011;
        read_addr_1 = 3'b110;
        read_addr_2 = 3'b110;

        #100
        write_en = 1;
        reg_write_dest = 3'b111;
        write_data = 10'b0011000011;
        read_addr_1 = 3'b110;
        read_addr_2 = 3'b111;

        #100
        write_en = 0;
        reset = 0;
    end

    always begin
        #50 clk =~clk;
    end
endmodule
```

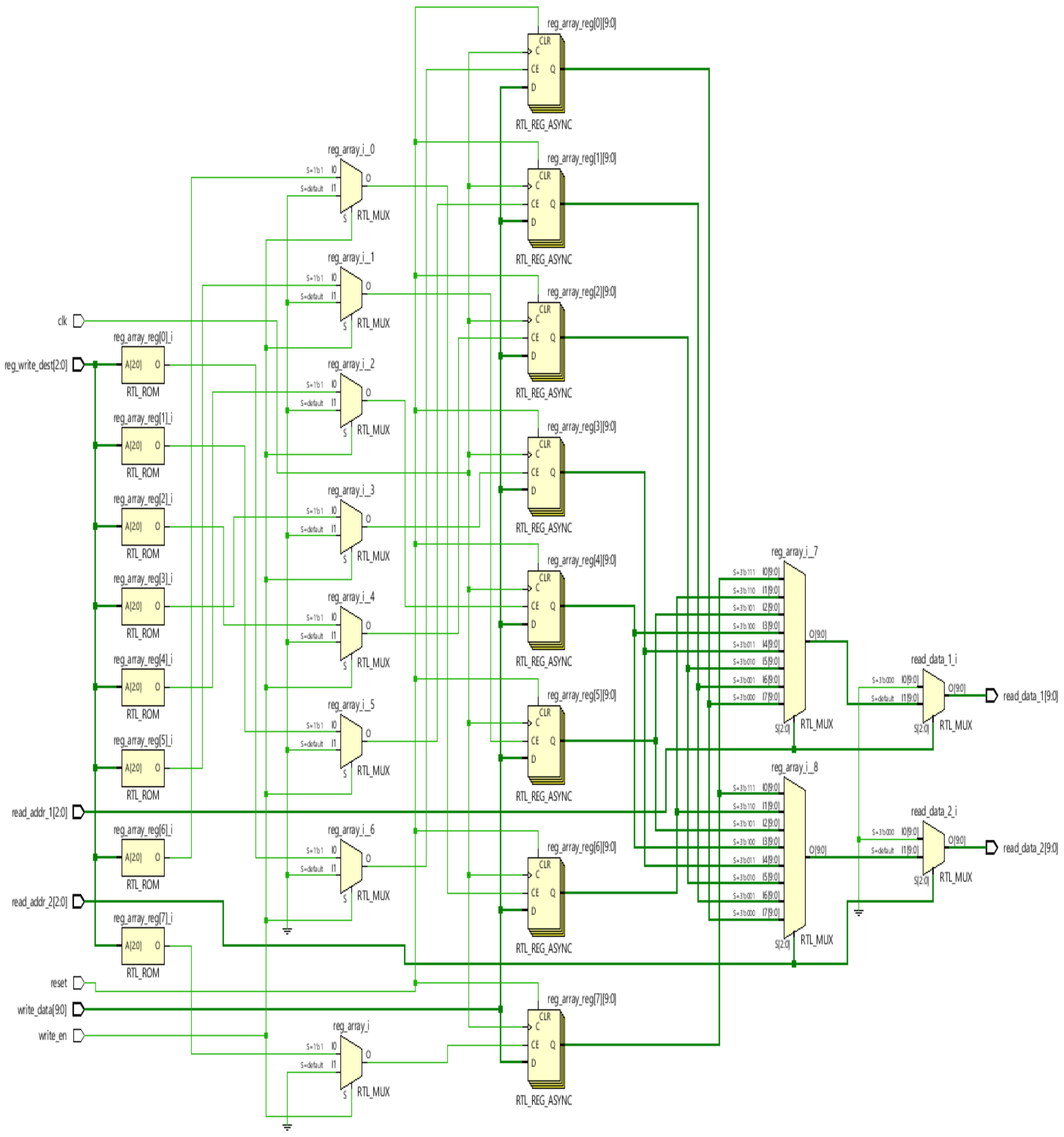


Figure 4: Schematic for Register File

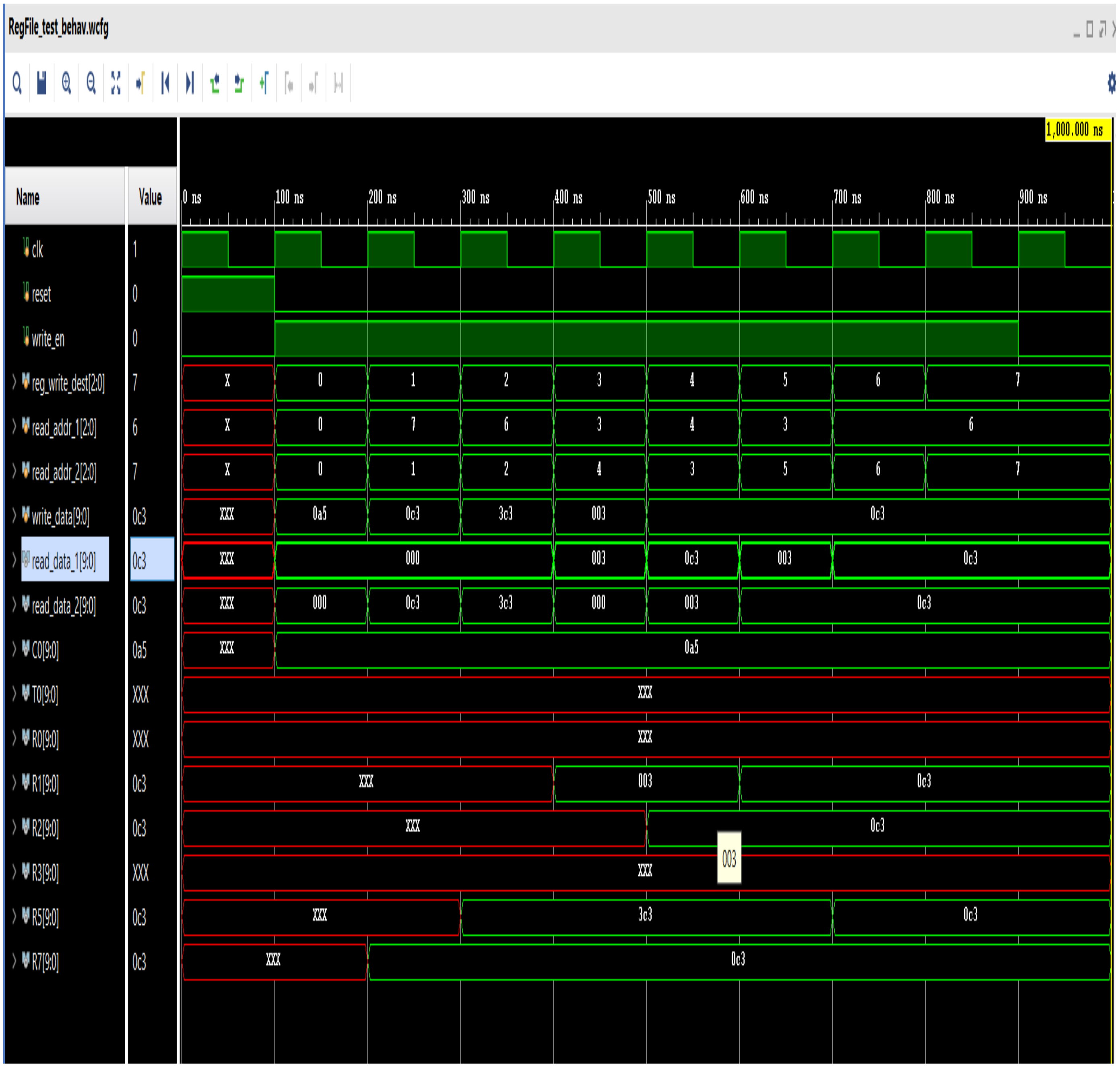
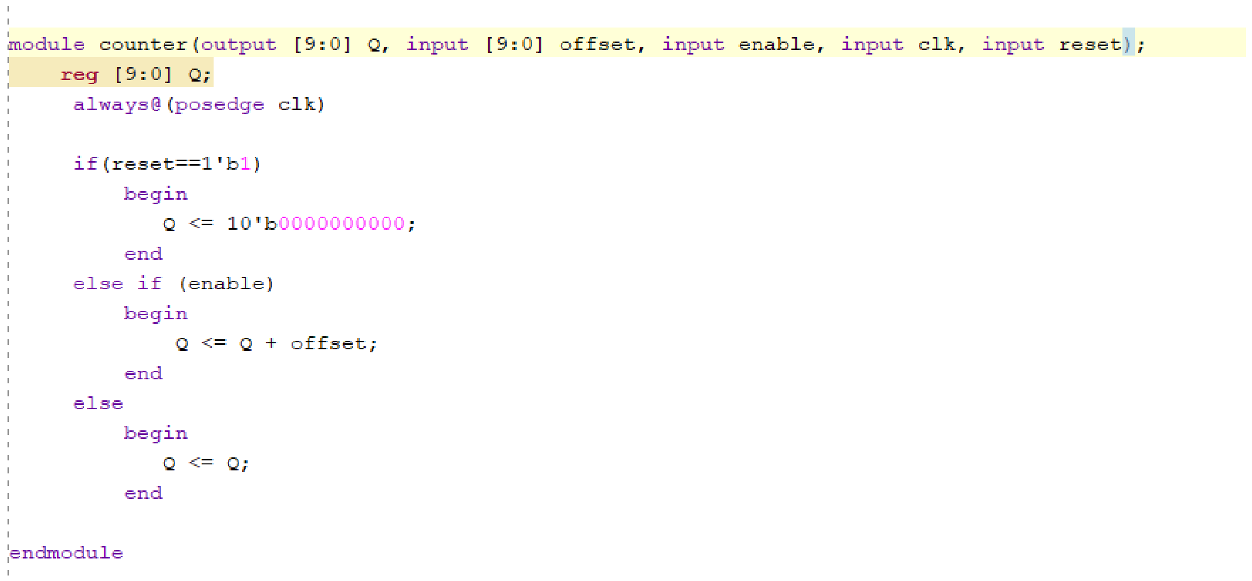


Figure 5: Timing diagram for Register File

3. Program Counter Register

The program counter register (10-bit) is designed in a such a way that any time an instruction is fetched, enable signal is turned on and hence the program counter increases by 1 or specified offset (whenever necessary) for each instruction synchronously. When the reset signal is turned on the program counter resets back to starting address or 0.



```
module counter(output [9:0] Q, input [9:0] offset, input enable, input clk, input reset);
    reg [9:0] Q;
     always@(posedge clk)

     if(reset==1'b1)
         begin
            Q <= 10'b0000000000;
         end
     else if (enable)
         begin
             Q <= Q + offset;
         end
     else
         begin
            Q <= Q;
         end

endmodule
```

Figure 6: Verilog code for 10-bit program counter implementation

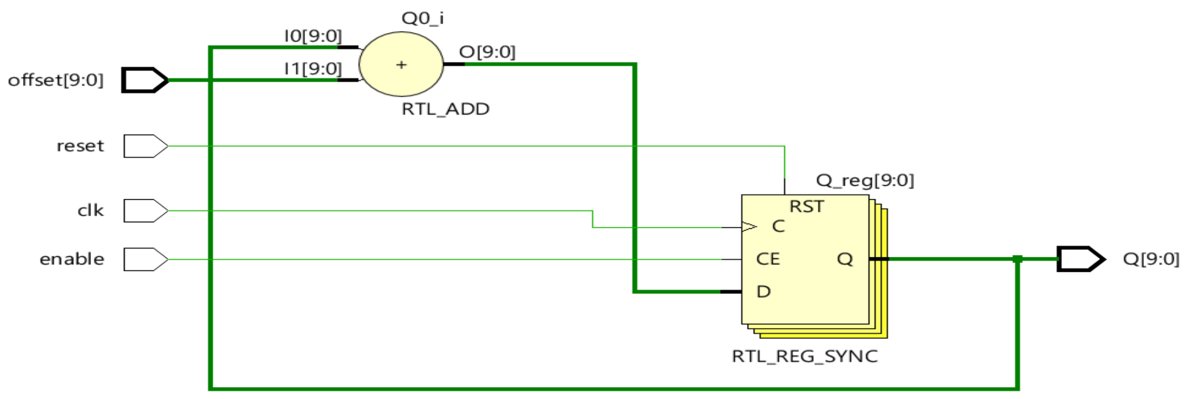


Figure 7: Schematic for 10-bit program counter

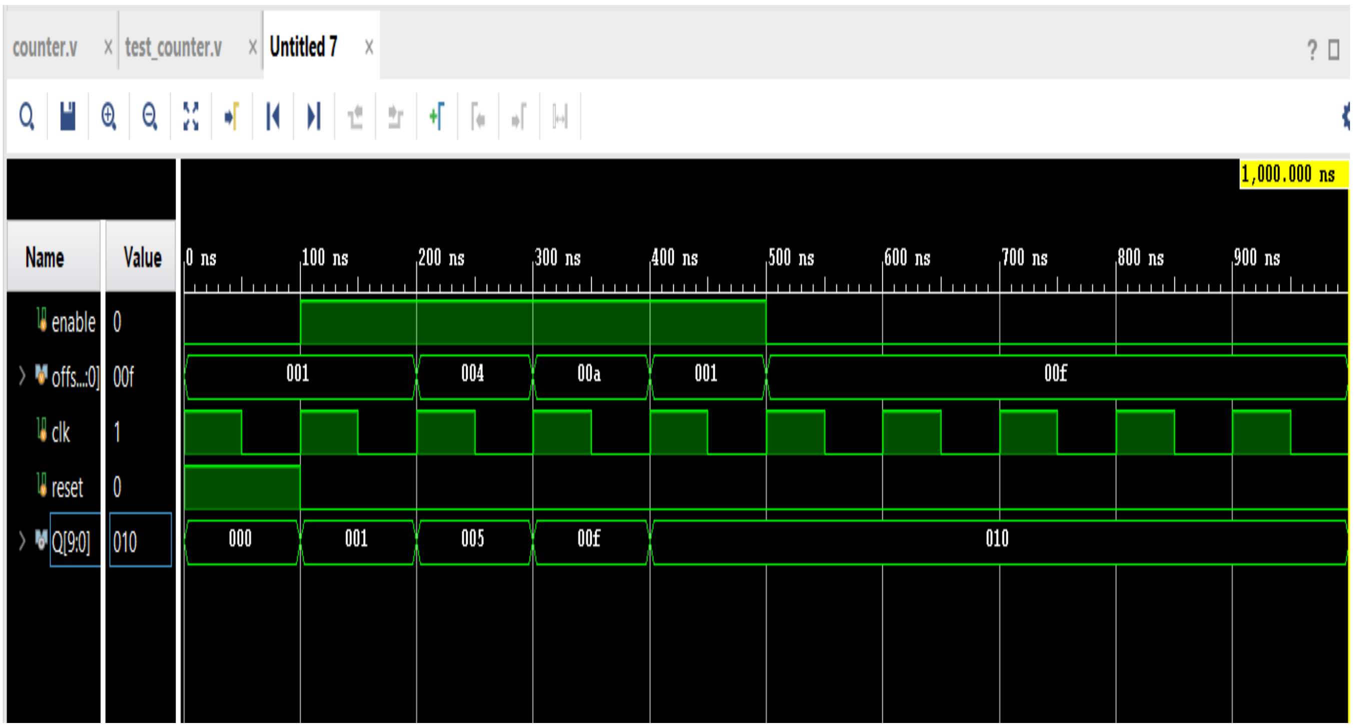


Figure 8: Timing Diagram for 10-bit program counter

Program Counter Testbench

```
`timescale 1ns / 1ps

module test_counter();
   reg enable; reg [9:0] offset; reg clk; reg reset;
   wire [9:0] Q;

   counter test(.enable(enable), .offset(offset), .clk(clk), .reset(reset), .Q(Q));

   initial
      begin
         clk = 1;
         enable = 1'b0;
         reset = 1'b1;
         offset = 10'b0000000001;
         #100

         enable = 1'b1;
         reset = 1'b0;
         offset = 10'b0000000001;
         #100
```

```
            enable = 1'b1;
            reset = 1'b0;
            offset = 10'b0000000100;
            #100

            enable = 1'b1;
            reset = 1'b0;
            offset = 10'b0000001010;
            #100

            enable = 1'b1;
            reset = 1'b0;
            offset = 10'b0000000001;
            #100

            enable = 1'b0;
            reset = 1'b0;
            offset = 10'b0000001111;

            #100
            offset = 10'b0000001111;
        end

        always
        #50 clk = ~clk;
endmodule
```

4. Control Unit

Control unit consists of all the control signals which includes ALU control signals for both adder and greater instruction, load and store instruction, branch signals for branch instructions. Finally, it also contains required control signal for halt instruction as well. Those control signals are asserted or deserted according to the opcode input provided to the control unit after fetching the instruction.

Control Unit Verilog Code

```
`timescale 1ns / 1ps
module Control_Unit(input[2:0] opcode,  output reg[1:0] alu_op, reg alu_src, mem_write,
mem_read, mem_to_reg, write_back, branch, reg_write, sign_or_zero, output reg[1:0]
ALU_Control);

  always @(alu_op)
     casex (alu_op)
      2'b00: ALU_Control = 2'b00;  //adder ALU
      2'b01: ALU_Control = 2'b01;   // grt ALU
      2'b10: ALU_Control = 2'b10;  //send to data memory
      2'b11: ALU_Control = 2'b11; //halt
      default: ALU_Control = 2'b11;
     endcase

    always @(*)
     begin
          case(opcode)
          3'b000: begin // load
                    alu_op = 2'b10;
                    alu_src = 1'b1;
                    mem_write = 1'b0;
                    mem_read = 1'b1;
                    mem_to_reg = 1'b1;
                    write_back = 1'b1;
                    branch = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                  end

          3'b001: begin // store
                    alu_op = 2'b10;
                    alu_src = 1'b1;
                    mem_write = 1'b1;
                    mem_read = 1'b0;
                    mem_to_reg = 1'b0;
                    write_back = 1'b0;
                    branch = 1'b0;
                    reg_write = 1'b0;
                    sign_or_zero = 1'b1;
                  end
          3'b010: begin // addi
                    alu_op = 2'b00;
                    alu_src = 1'b0;
                    mem_write = 1'b0;
                    mem_read = 1'b0;
                    mem_to_reg = 1'b0;
                    write_back = 1'b1;
                    branch = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                  end
          3'b011: begin // add
                    alu_op = 2'b00;
                    alu_src = 1'b1;
                    mem_write = 1'b0;
                    mem_read = 1'b0;
                    mem_to_reg = 1'b0;
                    write_back = 1'b1;
                    branch = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                  end

          3'b100: begin // grt
                    alu_op = 2'b01;
                    alu_src = 1'b1;
                    mem_write = 1'b0;
                    mem_read = 1'b0;
                    mem_to_reg = 1'b0;
                    write_back = 1'b1;
                    branch = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                  end

          3'b101: begin // beq
                    alu_op = 2'bxx;
                    alu_src = 1'bx;
                    mem_write = 1'b0;
                    mem_read = 1'b0;
                    mem_to_reg = 1'b0;
                    write_back = 1'bx;
                    branch = 1'b1;
                    reg_write = 1'bx;
                    sign_or_zero = 1'bx;
                  end

          3'b110: begin // bne
                    alu_op = 2'bxx;
                    alu_src = 1'bx;
                    mem_write = 1'b0;
                    mem_read = 1'b0;
                    mem_to_reg = 1'b0;
                    write_back = 1'bx;
                    branch = 1'b1;
                    reg_write = 1'bx;
                    sign_or_zero = 1'bx;
                    end

          3'b111: begin //  halt
                    alu_op = 2'b11;
                    alu_src = 1'bx;
                    mem_write = 1'bx;
                    mem_read = 1'bx;
                    mem_to_reg = 1'bx;
                    write_back = 1'bx;
                    branch = 1'bx;
                    reg_write = 1'bx;
                    sign_or_zero = 1'bx;
                    end
          default: begin
                    alu_op = 2'bxx;
                    alu_src = 1'bx;
                    mem_write = 1'bx;
                    mem_read = 1'bx;
                    mem_to_reg = 1'bx;
                    write_back = 1'bx;
                    branch = 1'bx;
                    reg_write = 1'bx;
                    sign_or_zero = 1'bx;
                    end
          endcase
          end
      endmodule
```

Control Unit Testbench

```
`timescale 1ns / 1ps

module test_control();
   reg [2:0] opcode;
   wire [2:0] ALU_Control; wire [1:0] alu_op;
   wire alu_src, mem_write, mem_read, mem_to_reg, write_back, branch, reg_write,
sign_or_zero;

   Control_Unit test(.opcode(opcode), .ALU_Control(ALU_Control), .alu_op(alu_op),
.alu_src(alu_src), .mem_write(mem_write), .mem_read(mem_read),  mem_to_reg(mem_to_reg),
.write_back(write_back), .branch(branch), .reg_write(reg_write), .sign_or_zero(sign_or_zero));

     initial
     begin
       opcode = 3'b000;

       #100
       opcode = 3'b000;

       #100
       opcode = 3'b001;

       #100
       opcode = 3'b010;

       #100
       opcode = 3'b011;

       #100
       opcode = 3'b100;

       #100
       opcode = 3'b101;

       #100
       opcode = 3'b110;

       #100
       opcode = 3'b111;

       #100
       opcode = 3'b111;
     end
endmodule
```

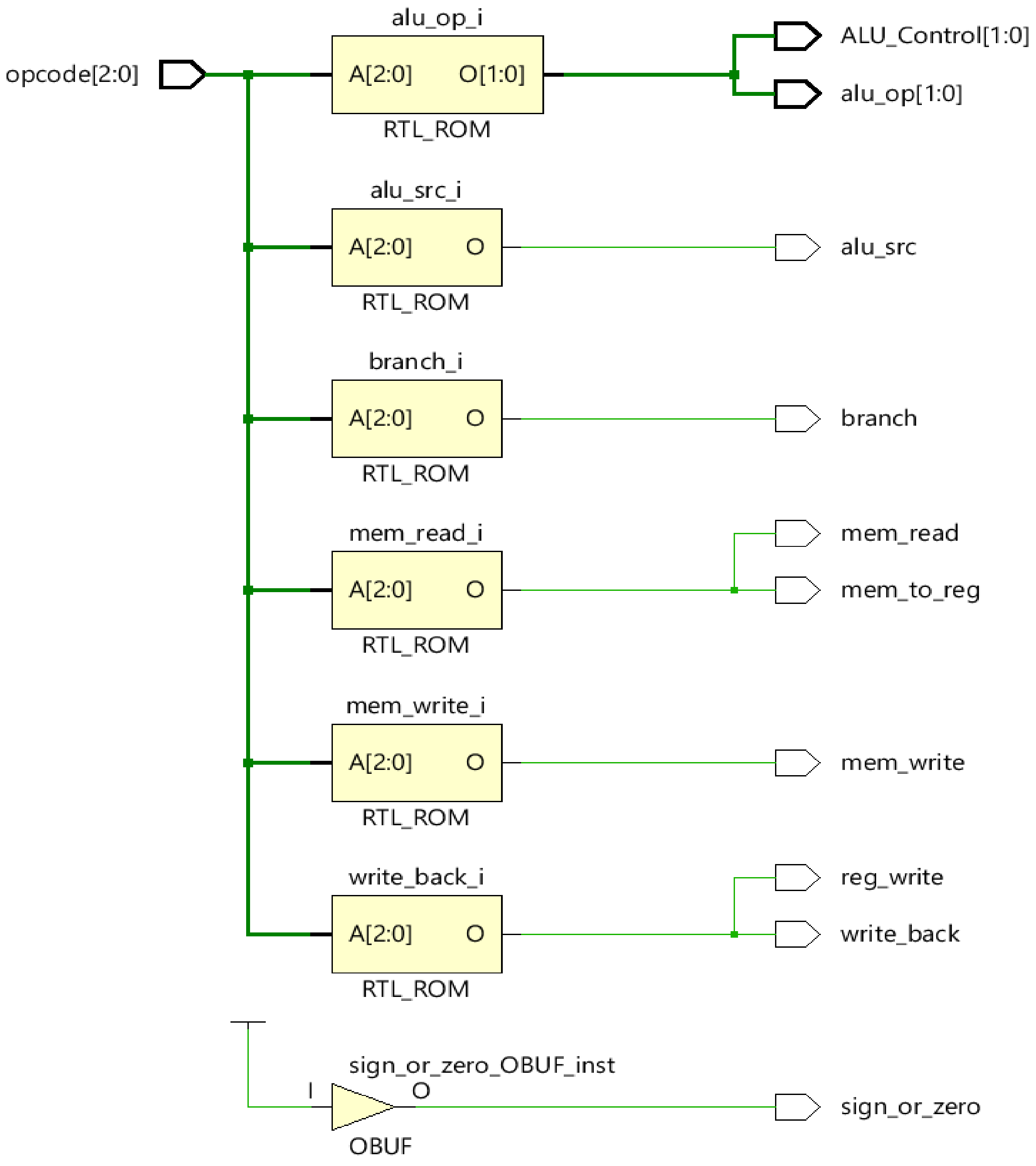


Figure 9: Schematic for Control Unit

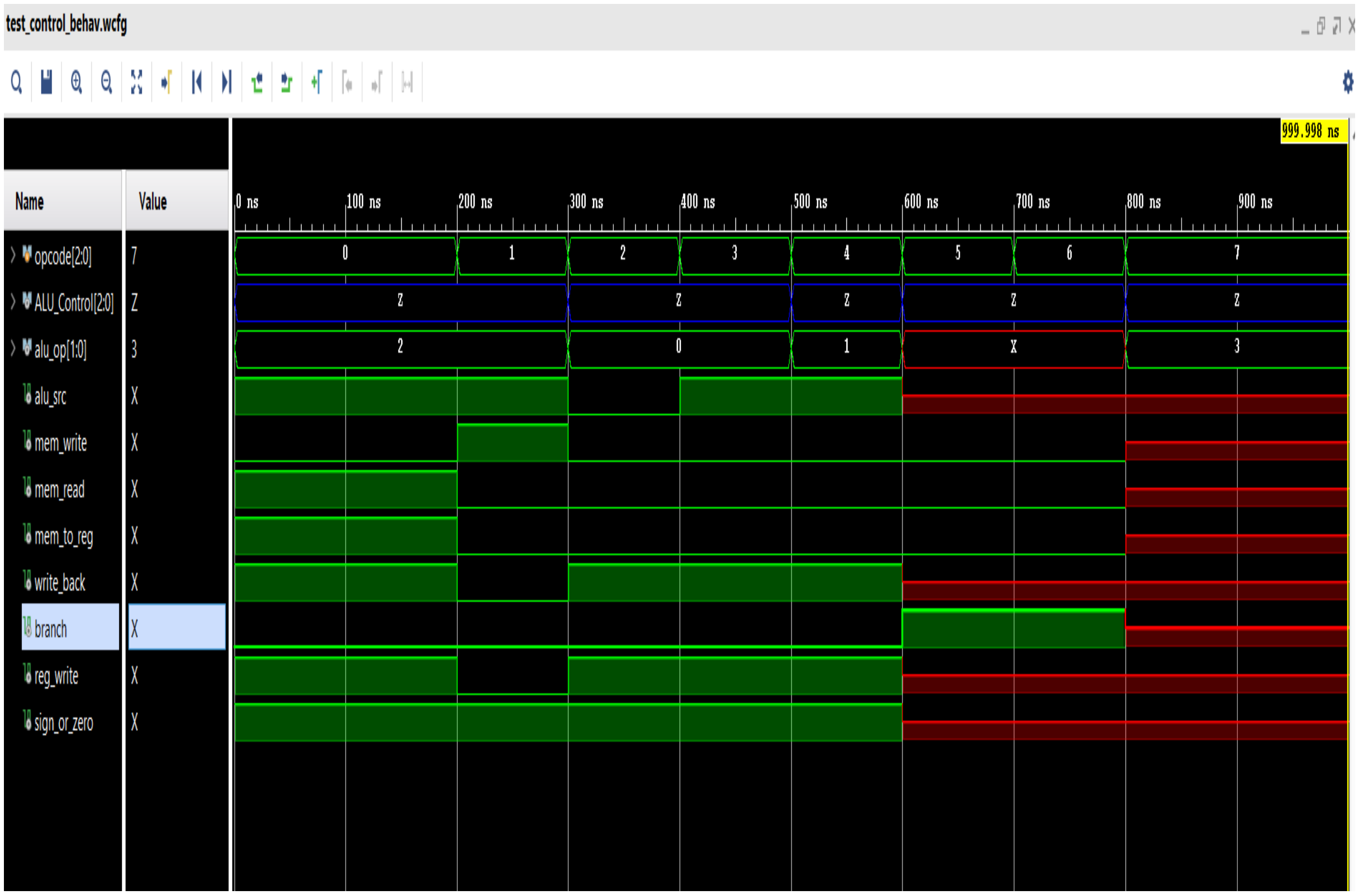


Figure 10: Timing Diagram for Control Unit

The control signals for the control unit are set according to the opcode of the instruction needed to run the program.

5. Instruction and Data Memory

Instruction Memory consists of all the instructions to be fetched to run a specific program. Those instructions are stored in a specific memory locations in the instruction memory and fetched to the main module 10-bit CPU according to the change in program counter or memory address.

Data Memory consists of the read and write data to be used by the 10-bit CPU. It also contains the RAM memory locations to store the values to be saved in the memory address and 16 such memory address or RAM locations are created. It has proper control signals for load and store data from or to memory locations.

IM Verilog code

```
`timescale 1ns / 1ps

module InstructionMemory(input clock, input [9:0] address, output reg [9:0] instruction);
   reg [9:0] memory[32:0];
   reg [1:0] program;
   integer i;
   integer mem0 = 0, mem8 = 8, distance = 29;

   initial begin
      program = 2'b10; //Determines the program to be used
      case(program)
      2'b11:
         begin
         memory[0] <= 10'b0000100000;
         memory[1] <= 10'b0000110001;
         memory[2] <= 10'b0101010000;
         memory[3] <= 10'b0111000010;
         memory[4] <= 10'b0100111111;
         memory[5] <= 10'b1000110101;
         memory[6] <= 10'b1100001100;
         memory[7] <= 10'b0101001100;
         memory[8] <= 10'b0011000010;
         memory[9] <= 10'b111xxxxxxx;
         end
      2'b10:
         begin
```

```
            i = 4;
            memory[0] <= 10'b0000110000;
            memory[1] <= 10'b1000110000;
            memory[2] <= 10'b1010011101; //beq
            memory[3] <= 10'b0010111000;

            while(i < 32)
                begin
                mem0 = mem0 + 1;
                mem8 = mem8 + 1;
                distance = distance - 4;
                memory[i] = {3'b000, 3'b011, mem0[3:0]};
                    i = i + 1;
                memory[i] = 10'b1000110000;
                    i = i + 1;
                memory[i] = {3'b101, distance[6:0]};
                    i = i + 1;
                memory[i] = {3'b001, 3'b011, mem8[3:0]};
                    i = i + 1;
                end

            memory[32] <= 10'b0001001111;
            memory[33] <= 10'b0001001111;
            memory[34] <= 10'b111xxxxxxx;
            end
        endcase
    end
    always @(address)
    begin
        instruction <= memory[address];
    end
endmodule
```

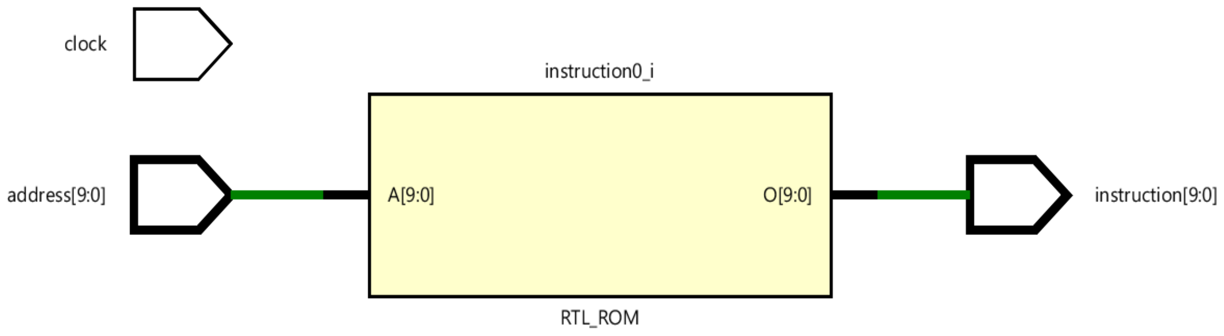
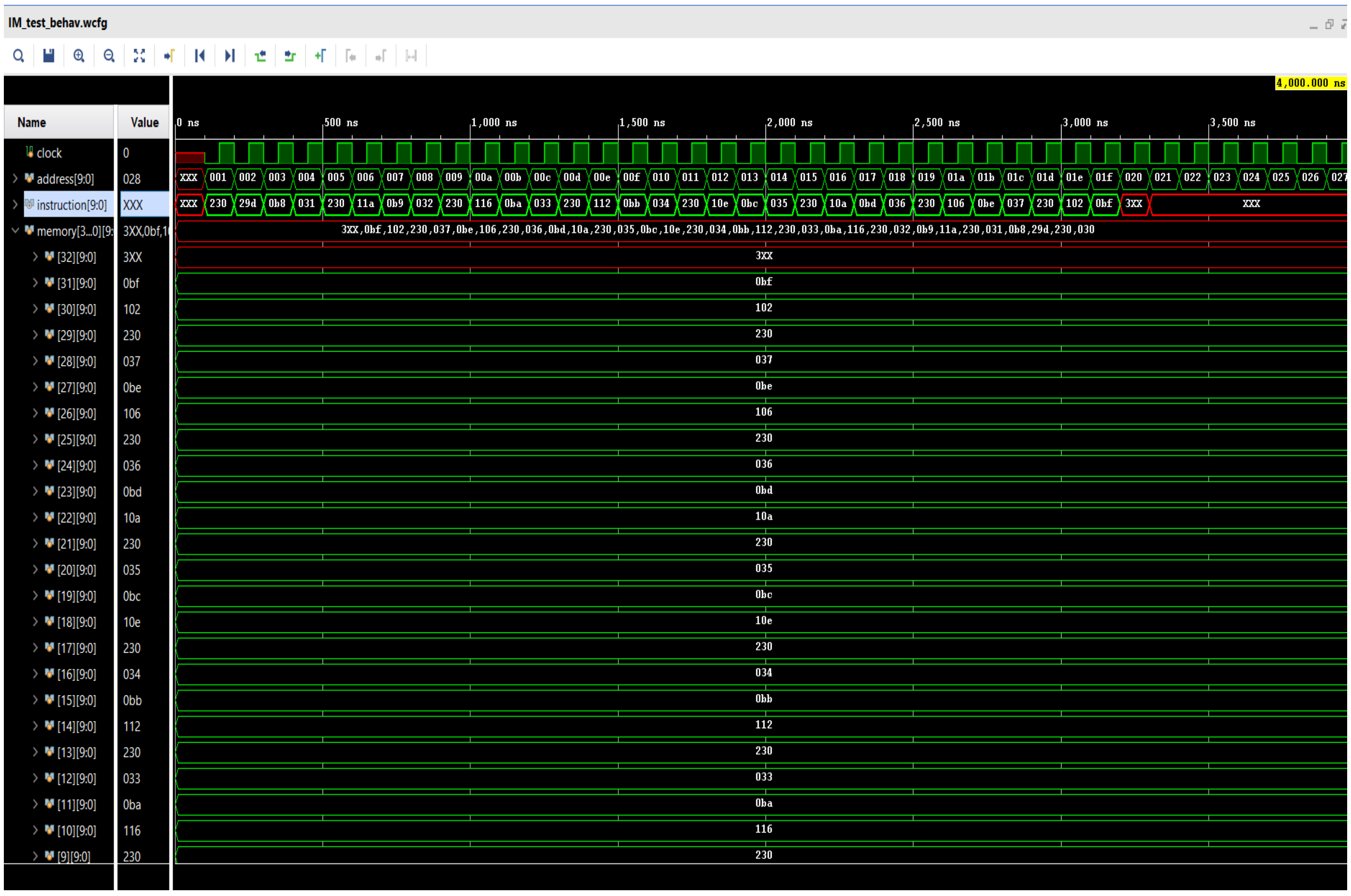


Figure 11: Schematic for Instruction Memory

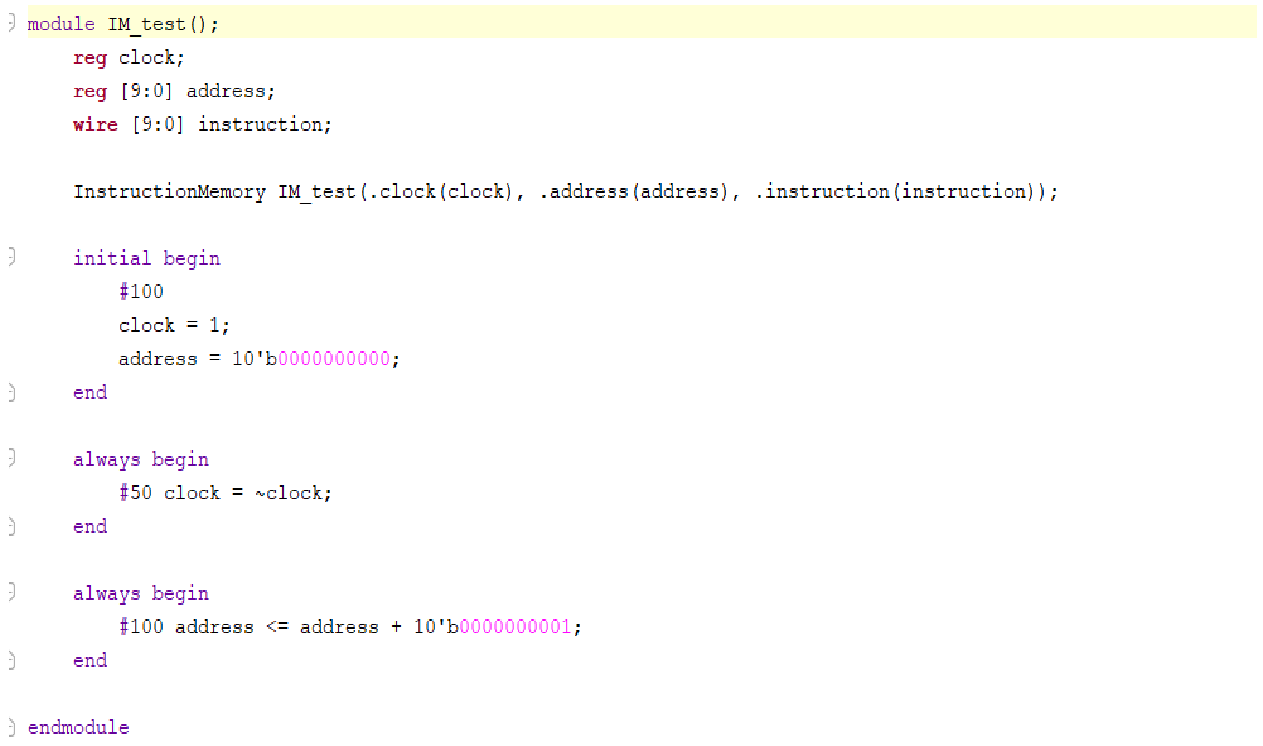


Name	Value
clock	0
address[9:0]	028
instruction[9:0]	XXX
memory[3...0][9:	3XX,0bf,1
[32][9:0]	3XX
[31][9:0]	0bf
[30][9:0]	102
[29][9:0]	230
[28][9:0]	037
[27][9:0]	0be
[26][9:0]	106
[25][9:0]	230
[24][9:0]	036
[23][9:0]	0bd
[22][9:0]	10a
[21][9:0]	230
[20][9:0]	035
[19][9:0]	0bc
[18][9:0]	10e
[17][9:0]	230
[16][9:0]	034
[15][9:0]	0bb
[14][9:0]	112
[13][9:0]	230
[12][9:0]	033
[11][9:0]	0ba
[10][9:0]	116
[9][9:0]	230

```
address[9:0]:     XXX 001 002 003 004 005 006 007 008 009 00a 00b 00c 00d 00e 00f 010 011 012 013 014 015 016 017 018 019 01a 01b 01c 01d 01e 01f 020 021 022 023 024 025 026 027
instruction[9:0]: XXX 230 29d 0b8 031 230 11a 0b9 032 230 116 0ba 033 230 112 0bb 034 230 10e 0bc 035 230 10a 0bd 036 230 106 0be 037 230 102 0bf 3XX XXX
memory:           3XX,0bf,102,230,037,0be,106,230,036,0bd,10a,230,035,0bc,10e,230,034,0bb,112,230,033,0ba,116,230,032,0b9,11a,230,031,0b8,29d,230,030
```

Figure 12: Timing Diagram for Instruction Memory

Instruction stored in the memory address locations are fetched from instruction memory.


```
module IM_test();
    reg clock;
    reg [9:0] address;
    wire [9:0] instruction;

    InstructionMemory IM_test(.clock(clock), .address(address), .instruction(instruction));

    initial begin
        #100
        clock = 1;
        address = 10'b0000000000;
    end

    always begin
        #50 clock = ~clock;
    end

    always begin
        #100 address <= address + 10'b0000000001;
    end

endmodule
```

Figure 13: Testbench code for Instruction Memory

Data Memory Verilog code

```
`timescale 1ns / 1ps

module DataMemory(input clk, input [3:0] address, input [9:0] write_data, input write_en, input
mem_read,
               output [9:0] read_data);
      integer i;
      reg [9:0] ram [15:0];
      wire [3:0] ram_address = address;
      reg [1:0] program;

      initial begin
         program = 2'b10;
         case(program)
         2'b11:
            for(i = 0; i < 16; i=i+1)
            if (i == 0 || i == 1)
               begin
```

```
                    ram[0] <= 10'b0000001000;
                    ram[1] <= 10'b0000000101;
                end
            else
                begin
                    ram[i] <= 10'b0000000000;
                end
        2'b10:
            for(i = 0; i < 16; i=i+1)
            if (i < 8)
                begin
                    ram[0] <= 10'b0000001010;
                    ram[1] <= 10'b0000001111;
                    ram[2] <= 10'b0000001100;
                    ram[3] <= 10'b0000001011;
                    ram[4] <= 10'b0000001101;
                    ram[5] <= 10'b0000001111;
                    ram[6] <= 10'b0000000111;
                    ram[7] <= 10'b0000000000;
                end
            else
                begin
                    ram[i] <= 10'b0000000000;
                end
        endcase
    end

    always @(posedge clk) begin
        if (write_en)
            ram[ram_address] <= write_data;
    end
    assign read_data = (mem_read) ? ram[ram_address] : 10'b0000000000;
endmodule
```

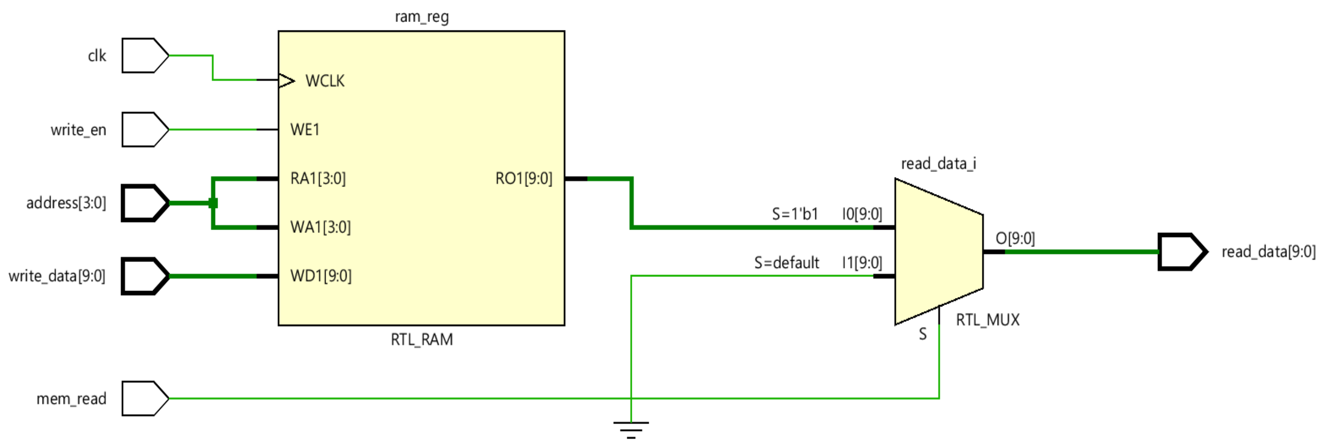
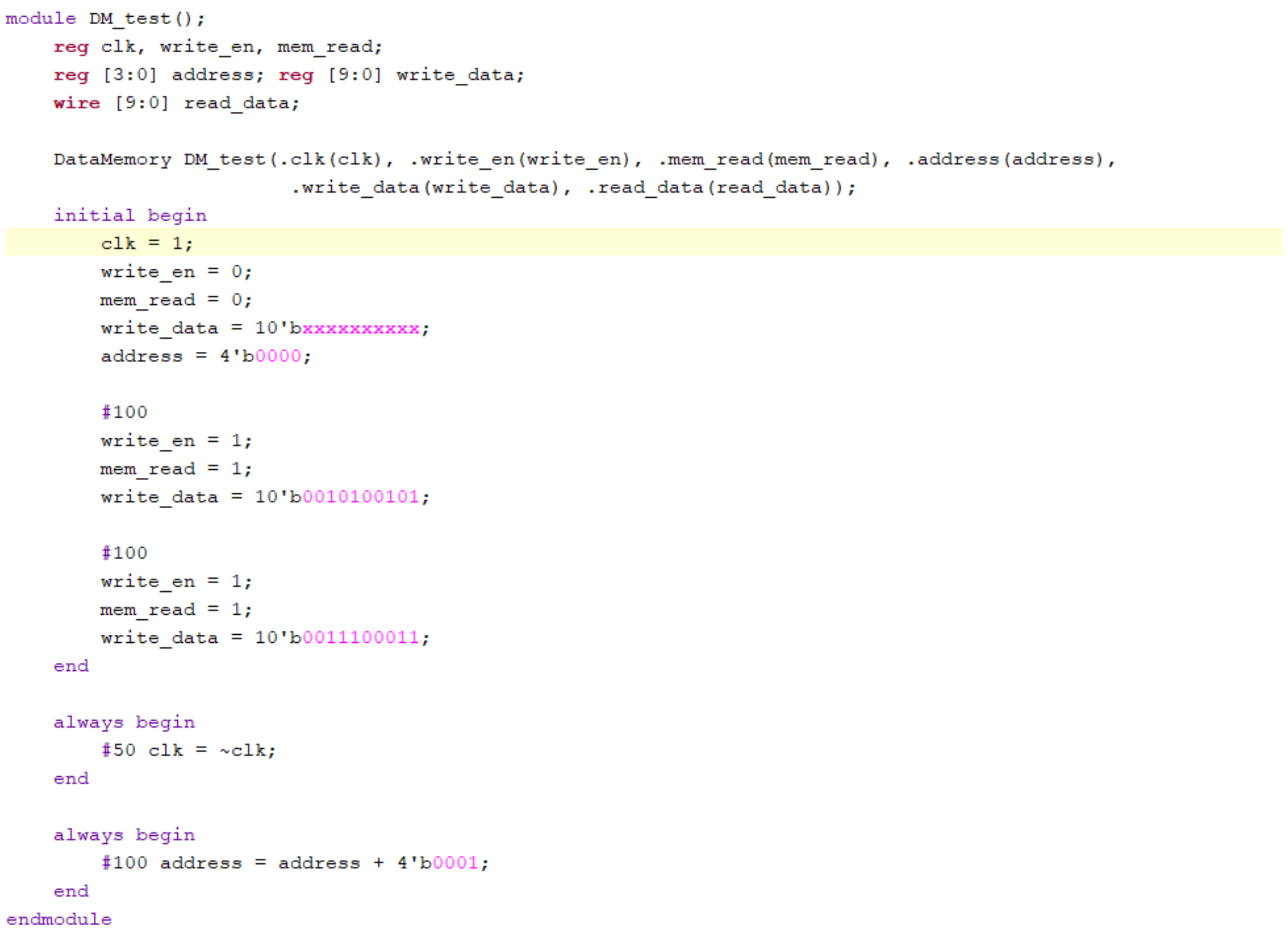


Figure 14: Schematic diagram for Data Memory



```
module DM_test();
    reg clk, write_en, mem_read;
    reg [3:0] address; reg [9:0] write_data;
    wire [9:0] read_data;

    DataMemory DM_test(.clk(clk), .write_en(write_en), .mem_read(mem_read), .address(address),
                    .write_data(write_data), .read_data(read_data));
    initial begin
        clk = 1;
        write_en = 0;
        mem_read = 0;
        write_data = 10'bxxxxxxxxxx;
        address = 4'b0000;

        #100
        write_en = 1;
        mem_read = 1;
        write_data = 10'b0010100101;

        #100
        write_en = 1;
        mem_read = 1;
        write_data = 10'b0011100011;
    end

    always begin
        #50 clk = ~clk;
    end

    always begin
        #100 address = address + 4'b0001;
    end
endmodule
```

Figure 15: Verilog testbench for data memory

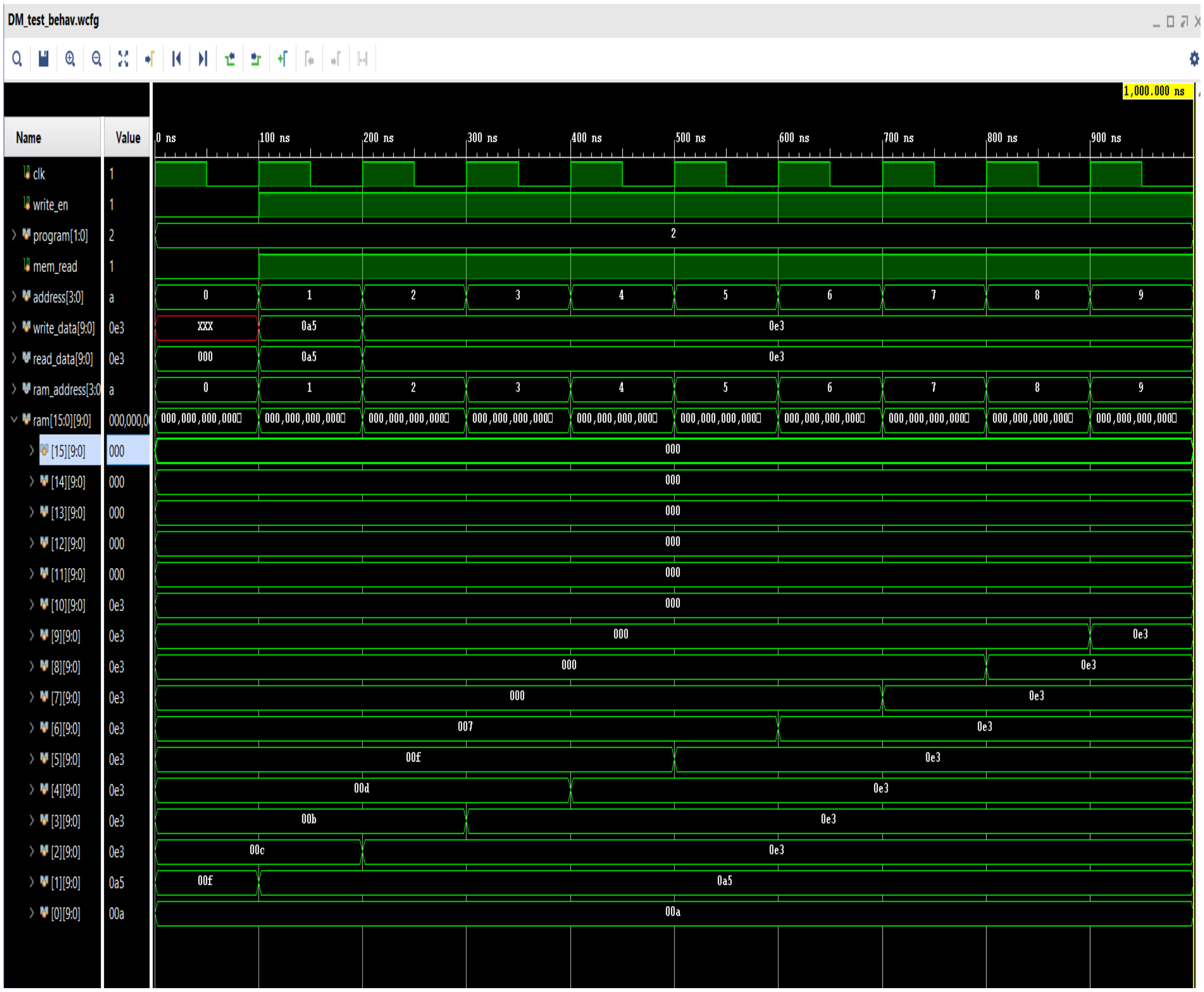


Figure 16: Timing diagram for Data Memory

All the data are accessed through data memory and loaded into the registers.

CPU 10 bits

With the instruction in machine code as an input, the CPU 10 bits decodes the instruction and performs the specified operation according to the control signals for control unit, register module values, ALU control unit and ALU module. It also has access to data memory, instruction memory and program counter register for the running of the program.

Verilog Code for CPU 10 bits

```
module CPU_10bits (input clock, input reset, output done, output [9:0] result);
   wire [9:0] instruction;
   reg [9:0] address;

   wire [1:0] alu_op;
   wire alu_src;
   wire mem_write;
   wire mem_read;
   wire mem_to_reg;
   wire write_back;
   wire branch;
   wire reg_write;
   wire [1:0] ALU_Control;
   wire [9:0] ALU_output;
   wire [9:0] C0;

   always begin
         #100 address = (~reset) ? address + 4'b0001: 'b0000;
         if (branch == 1 && instruction[9:7] == 3'b110 && C0 == 10'b0000000001)
             address = address + {{3{instruction[6]}},instruction[6:0]};
         if (branch == 1 && instruction[9:7] == 3'b101 && C0 == 10'b0000000000)
             address = address + {{3{instruction[6]}},instruction[6:0]};
         if (instruction[9:7] == 3'b111)
             address = 10'bxxxxxxxxxx;
   end

   InstructionMemory IM(.clock(clock), .address(address), .instruction(instruction));
   Control_Unit control(.opcode(instruction[9:7]), .alu_op(alu_op), .alu_src(alu_src),
.mem_write(mem_write), .mem_read(mem_read),
                     .mem_to_reg(mem_to_reg), .write_back(write_back), .branch(branch),
.reg_write(reg_write), .ALU_Control(ALU_Control));

   wire [2:0] reg_dst;
```

```
    wire [2:0] Reg1;
    assign Reg1 = instruction[6:4];
    wire [2:0] Reg2;
    assign Reg2 = instruction[2:0];
    wire [9:0] Reg_or_Num;

    wire [9:0] write_data;
    wire [9:0] read_data;
    wire [9:0] read_data_1;
    wire [9:0] read_data_2;

    assign reg_dst = (instruction[9:7] == 3'b100) ? 3'b000: Reg1;

    RegFile reg_file(.clk(clock), .reset(reset), .write_en(reg_write), .reg_write_dest(reg_dst),
.write_data(write_data),
                    .read_addr_1(Reg1), .read_data_1(read_data_1), .read_addr_2(Reg2),
.read_data_2(read_data_2));

    mux M0(.A1({{6{instruction[3]}},instruction[3:0]}), .A2(read_data_2), .Sel(alu_src),
.Y(Reg_or_Num));

    ALU_module ALU(.ALU_Control(ALU_Control[0]), .A(read_data_1), .B(Reg_or_Num),
.ALU_output(ALU_output));

    DataMemory DM(.clk(clock), .address(instruction[3:0]), .write_data(read_data_1),
.write_en(mem_write), .mem_read(mem_read), .read_data(read_data));

    assign write_data = (mem_to_reg == 1)? read_data: ALU_output;
    assign C0 = (instruction[9:7] == 3'b100)? write_data : C0;

    assign result = (branch == 1) ? 10'bxxxxxxxxxx: (ALU_Control == 2'b10) ? 10'bZZZZZZZZ:
ALU_output;
    assign done = (instruction[9:7] == 3'b111) ? 1 : 0;

endmodule
```

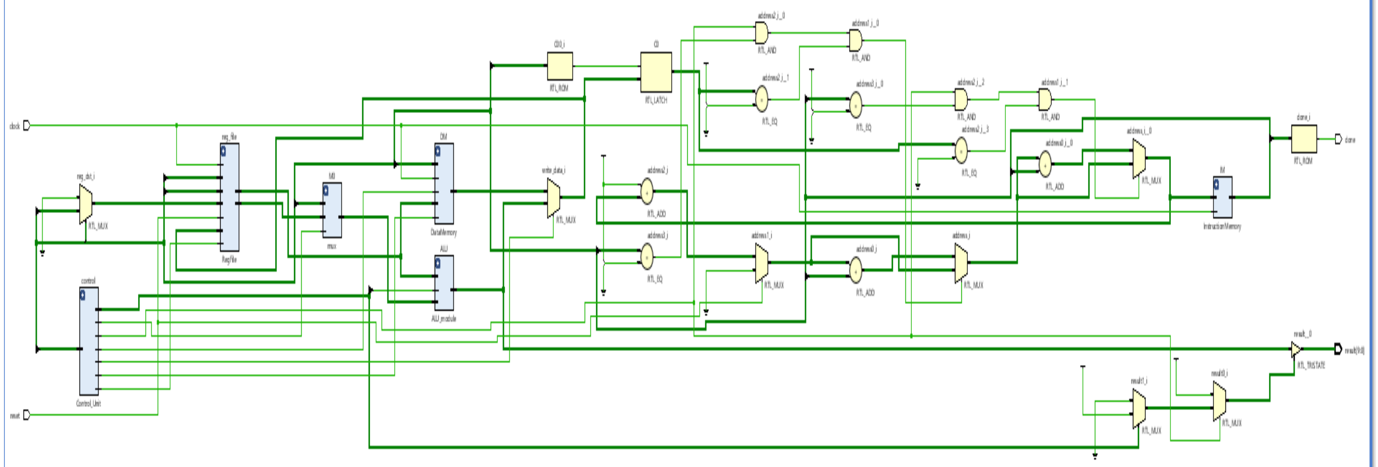
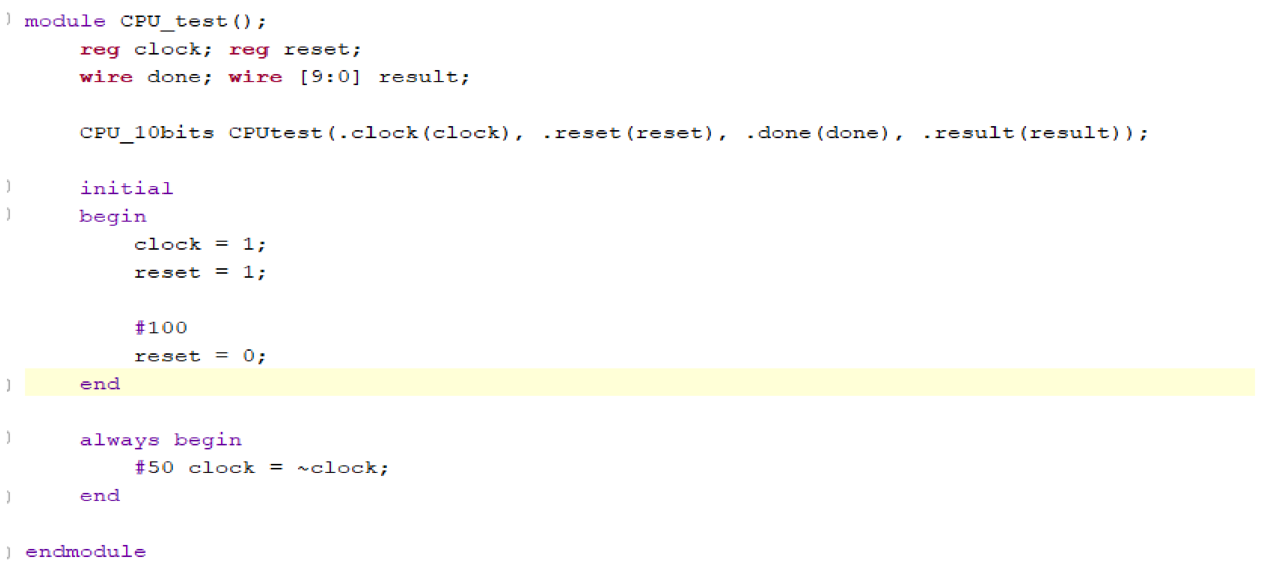


Figure 17: Schematic for CPU 10 bits



```
module CPU_test();
    reg clock; reg reset;
    wire done; wire [9:0] result;

    CPU_10bits CPUtest(.clock(clock), .reset(reset), .done(done), .result(result));

    initial
    begin
        clock = 1;
        reset = 1;

        #100
        reset = 0;
    end

    always begin
        #50 clock = ~clock;
    end

endmodule
```

Figure 18: Verilog testbench for CPU 10 bits

Program 2 (Problem Solving)

Write a program that solves the following equation: f = x*y – 4 (not multiply operation)

	MIPS Code	Machine Code
	load $R0, M0	10'b0000100000
	load $R1, M1	10'b0000110001
	addi $R3, 0	10'b0101010000
loop:	add $R2, $R0	10'b0111000010
	addi $R1, -1	10'b0100111111
	grt $R1, $R3	10'b1000110101
	bne loop	10'b1101111100
	addi $R2, -4	10'b0101001100
	store $R2, M2	10'b0011000010
	halt	10'b111xxxxxxx

For this program, the values of x and y are already stored in the memory location M0 and M1 as x = 8 in M0 and y = 5 in M1 which is also shown in Figure 18 below. And, then finally the result f = 36 or 0x24 after the completion of program is saved in memory location M2. For load and store instruction the 'result' output is set to high Z as the they do not affect ALU output at all and for branch instructions t0 'result' output is set to Xs as shown in Figure 18 below.

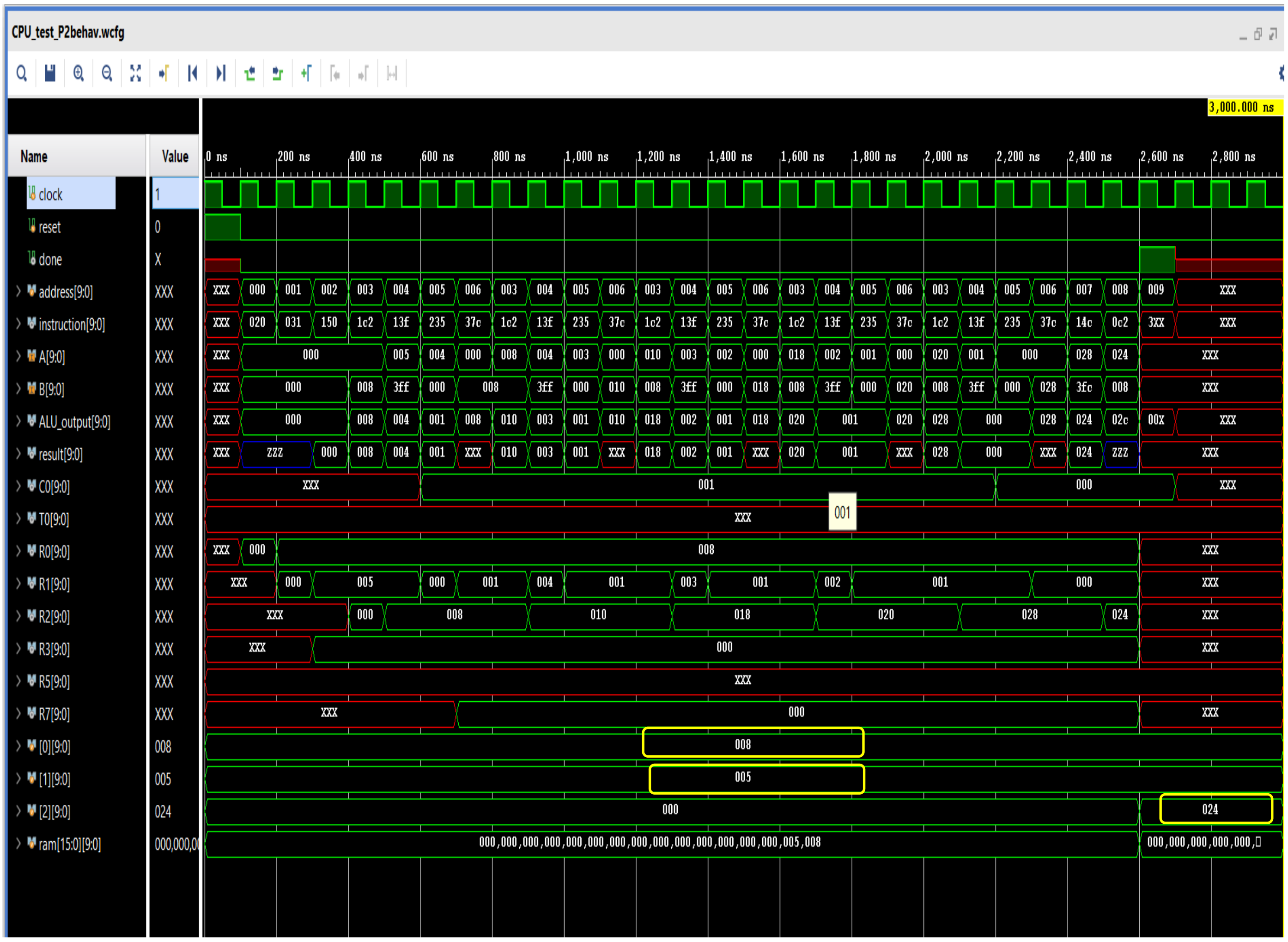
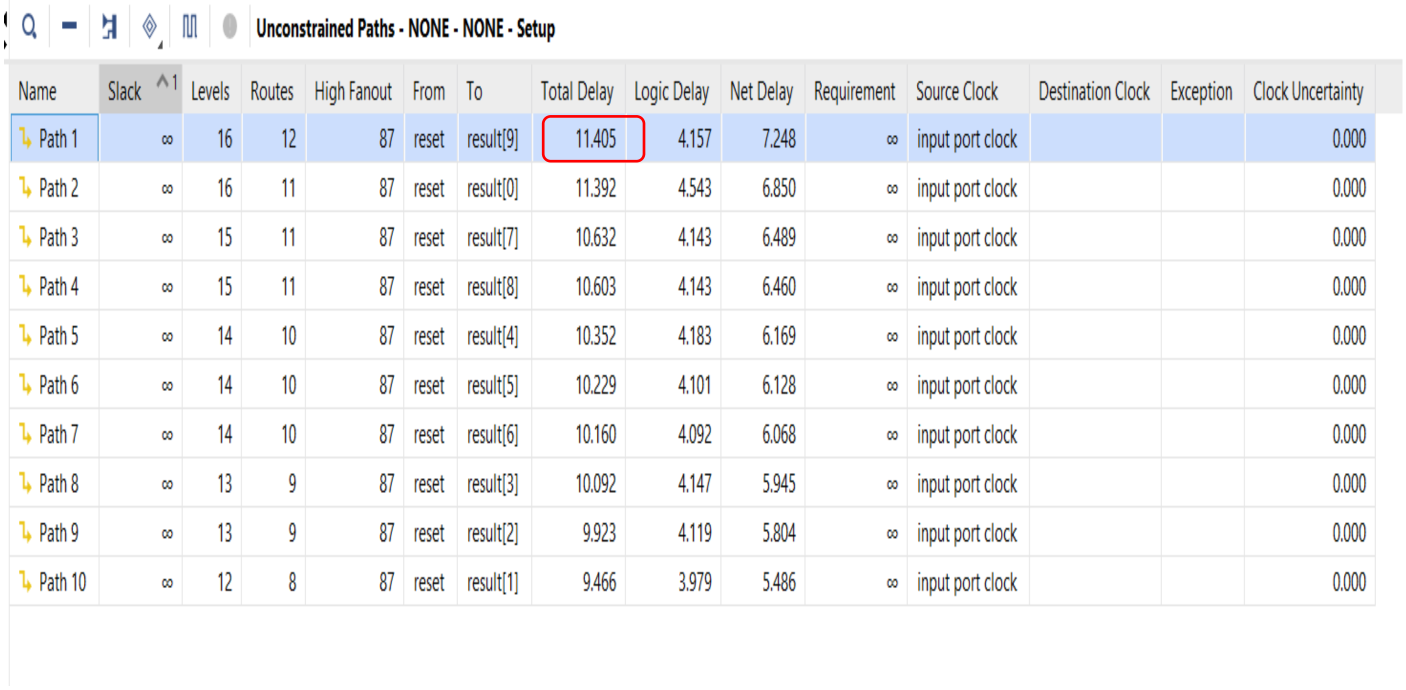


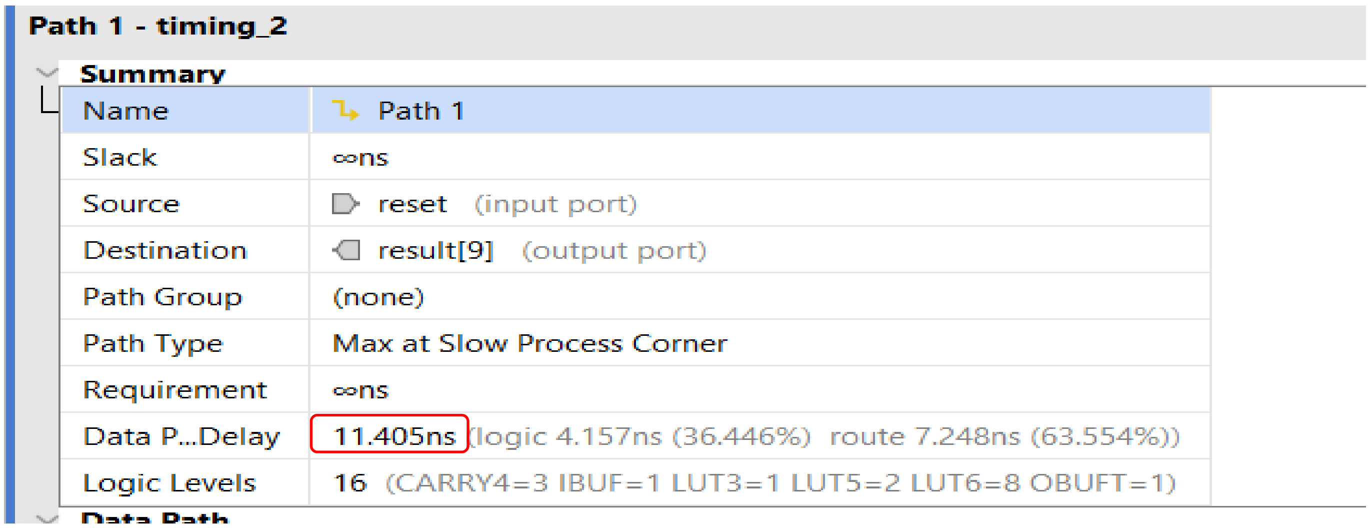
Figure 19: Simulation Timing Diagram for Program 3 (Problem Solving)

The necessary signals needed for the Program 3 (Problem Solving program) is shown in Figure 18 above. The program counter/address is increased synchronously, and the respective instruction stored in the PC/address is fetched accordingly. The two ALU input signals A and B with ALU Output is present. Also, all the register contents are shown in register C0, T0, R0, R1, R2, R3, R5 and R7. Finally, the required memory locations accessed during the running of program or where the values are stored are also shown in the figure above. The output signal Done turns high when the halt instruction (0x3xx or 10'b111xxxxxxx) is executed and the address becomes all Xs.



Unconstrained Paths - NONE - NONE - Setup

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	∞	16	12	87	reset	result[9]	11.405	4.157	7.248	∞	input port clock			0.000
Path 2	∞	16	11	87	reset	result[0]	11.392	4.543	6.850	∞	input port clock			0.000
Path 3	∞	15	11	87	reset	result[7]	10.632	4.143	6.489	∞	input port clock			0.000
Path 4	∞	15	11	87	reset	result[8]	10.603	4.143	6.460	∞	input port clock			0.000
Path 5	∞	14	10	87	reset	result[4]	10.352	4.183	6.169	∞	input port clock			0.000
Path 6	∞	14	10	87	reset	result[5]	10.229	4.101	6.128	∞	input port clock			0.000
Path 7	∞	14	10	87	reset	result[6]	10.160	4.092	6.068	∞	input port clock			0.000
Path 8	∞	13	9	87	reset	result[3]	10.092	4.147	5.945	∞	input port clock			0.000
Path 9	∞	13	9	87	reset	result[2]	9.923	4.119	5.804	∞	input port clock			0.000
Path 10	∞	12	8	87	reset	result[1]	9.466	3.979	5.486	∞	input port clock			0.000



Path 1 - timing_2

Summary

Name	Path 1
Slack	∞ns
Source	reset (input port)
Destination	result[9] (output port)
Path Group	(none)
Path Type	Max at Slow Process Corner
Requirement	∞ns
Data P...Delay	11.405ns (logic 4.157ns (36.446%) route 7.248ns (63.554%))
Logic Levels	16 (CARRY4=3 IBUF=1 LUT3=1 LUT5=2 LUT6=8 OBUFT=1)

Data Path

Figure (i): CCT for Program 2 (Problem Solving): Critical Path Delay is 11.405ns

Program 3 (String Copy)

Write a program that copies a null terminated string from one array in memory to another and returns number 10 in one of the registers after the copy is finished.

MIPS Code	**Machine Code**
load $R1, M0	10’b0000110000
grt $R1, $R0	10’b1000110010
beq END	10’b1010011101
store $R1, M8	10’b0010111000
load $R1, M1	10’b0000110001
grt $R1, $R0	10’b1000110010
beq END	10’b1010011010
store $R1, M9	10’b0010111001
load $R1, M2	10’b0000110010
grt $R1, $R0	10’b1000110010
beq END	10’b1010010110
store $R1, M10	10’b0010111010
load $R1, M3	10’b0000110011
grt $R1, $R0	10’b1000110010
beq END	10’b1010010010
store $R1, M11	10’b0010111011
load $R1, M4	10’b0000110100
grt $R1, $R0	10’b1000110010
beq END	10’b1010001110

```
        store $R1, M12                              10’b0010001100
        load $R1, M5                                10’b0000110101
        grt $R1, $R0                                10’b1000110010
        beq END                                     10’b1010001010
        store $R1, M13                              10’b0010001101
        load $R1, M6                                10’b0000110110
        grt $R1, $R0                                10’b1000110010
        beq END                                     10’b1010000110
        store $R1, M14                              10’b0010001110
        load $R1, M7                                10’b0000110111
        grt $R1, $R0                                10’b1000110010
        beq END                                     10’b1010000010
        store $R1, M14                              10’b0010001110
END
        load $R2, M15                               10’b0001001111
        halt
```

The string to be copied from one memory array to another is stored in memory locations each containing each character as the string is “afcbdf7” stored in M0, M1, M2, M3, M4, M5 and M6 with null terminated in M7. This string is copied to another memory array M8…M14 and a value 10 is returned in register R2 as shown in Figure 20 below. And M15 just contains a value 10 to be loaded to register. Using load and store instructions, the string is copied from one memory array to another and 10 is returned from R2.

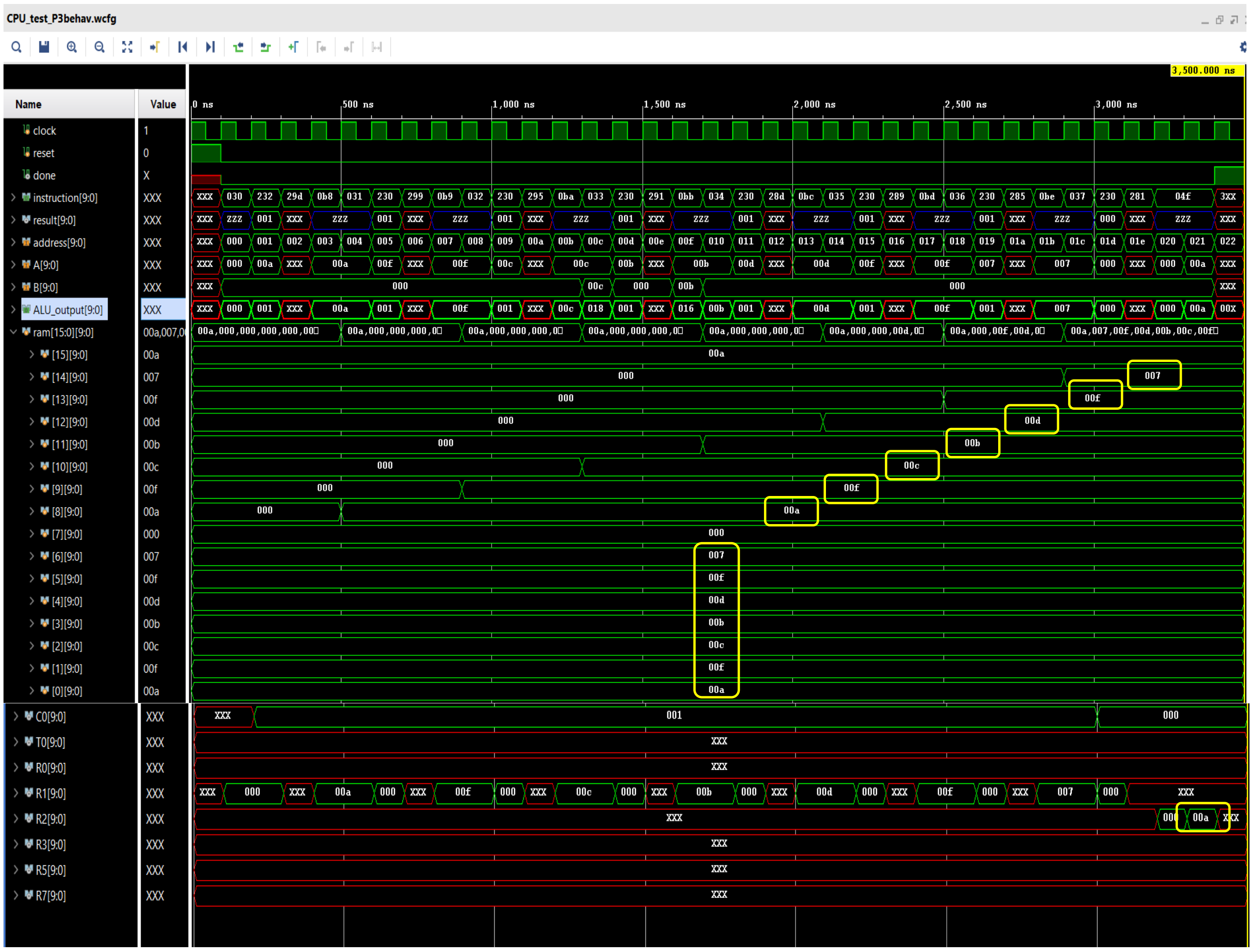
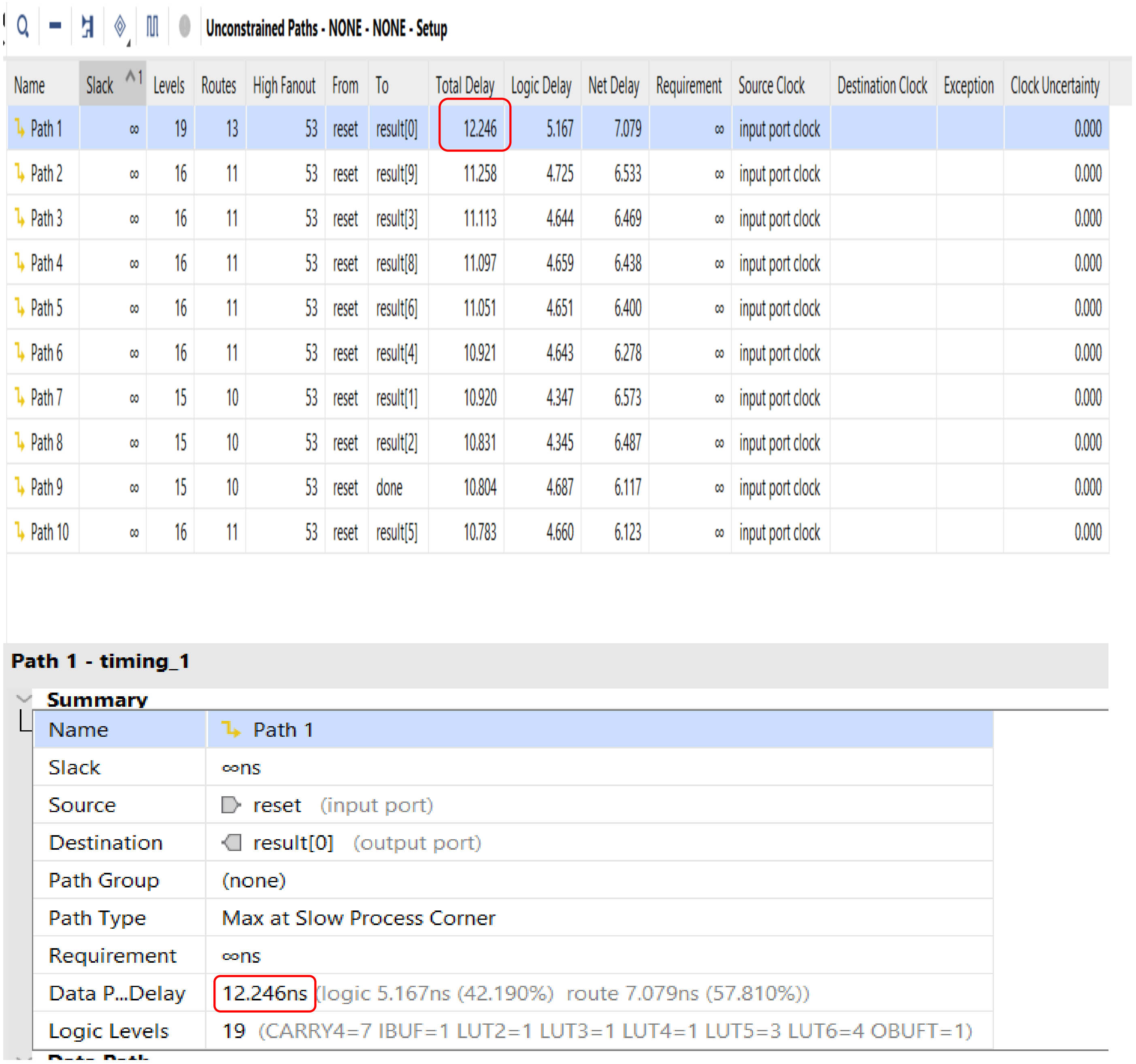


Figure 20: Timing Diagram for Program 3 (String Copy)


Unconstrained Paths - NONE - NONE - Setup

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	∞	19	13	53	reset	result[0]	12.246	5.167	7.079	∞	input port clock			0.000
Path 2	∞	16	11	53	reset	result[9]	11.258	4.725	6.533	∞	input port clock			0.000
Path 3	∞	16	11	53	reset	result[3]	11.113	4.644	6.469	∞	input port clock			0.000
Path 4	∞	16	11	53	reset	result[8]	11.097	4.659	6.438	∞	input port clock			0.000
Path 5	∞	16	11	53	reset	result[6]	11.051	4.651	6.400	∞	input port clock			0.000
Path 6	∞	16	11	53	reset	result[4]	10.921	4.643	6.278	∞	input port clock			0.000
Path 7	∞	15	10	53	reset	result[1]	10.920	4.347	6.573	∞	input port clock			0.000
Path 8	∞	15	10	53	reset	result[2]	10.831	4.345	6.487	∞	input port clock			0.000
Path 9	∞	15	10	53	reset	done	10.804	4.687	6.117	∞	input port clock			0.000
Path 10	∞	16	11	53	reset	result[5]	10.783	4.660	6.123	∞	input port clock			0.000

Path 1 - timing_1

Summary

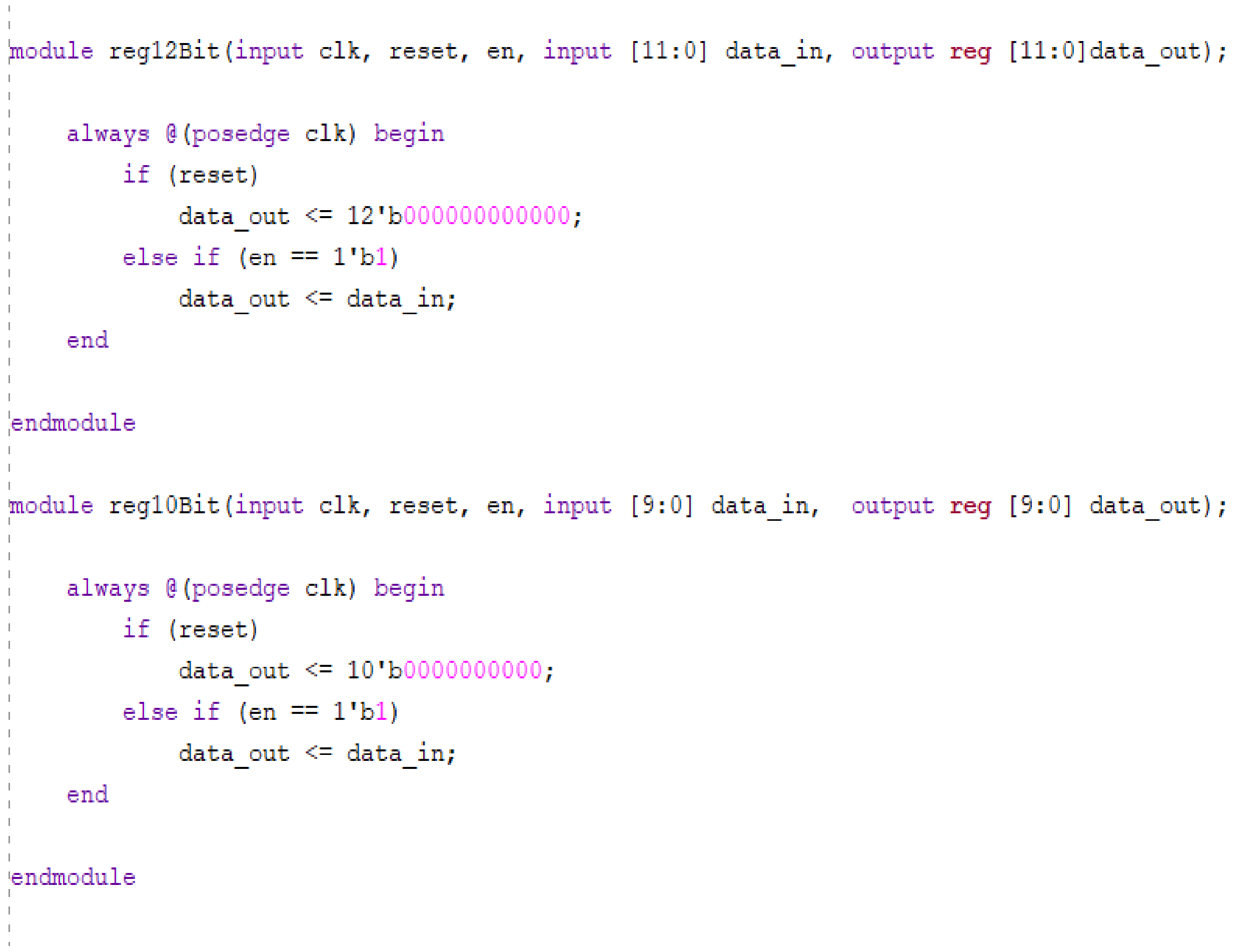
Name	Path 1
Slack	∞ns
Source	reset (input port)
Destination	result[0] (output port)
Path Group	(none)
Path Type	Max at Slow Process Corner
Requirement	∞ns
Data P...Delay	12.246ns (logic 5.167ns (42.190%) route 7.079ns (57.810%))
Logic Levels	19 (CARRY4=7 IBUF=1 LUT2=1 LUT3=1 LUT4=1 LUT5=3 LUT6=4 OBUFT=1)

Figure (ii): CCT for Program 3 (String Copy): Critical Path Delay is 12.246ns

CPU 10 bits Pipeline Components

1. Pipeline Registers

Two pipeline registers are created to fit all the information needed for a certain instruction type to be executed properly. One of the pipeline registers is utilized for the fetch-decode/execute-memory and the second is utilized for the execute-memory/writeback.



```verilog
module reg12Bit(input clk, reset, en, input [11:0] data_in, output reg [11:0]data_out);

    always @(posedge clk) begin
        if (reset)
            data_out <= 12'b000000000000;
        else if (en == 1'b1)
            data_out <= data_in;
    end

endmodule

module reg10Bit(input clk, reset, en, input [9:0] data_in,  output reg [9:0] data_out);

    always @(posedge clk) begin
        if (reset)
            data_out <= 10'b0000000000;
        else if (en == 1'b1)
            data_out <= data_in;
    end

endmodule
```

Figure 21: Verilog code for two pipeline registers (12 bits and 10 bits)

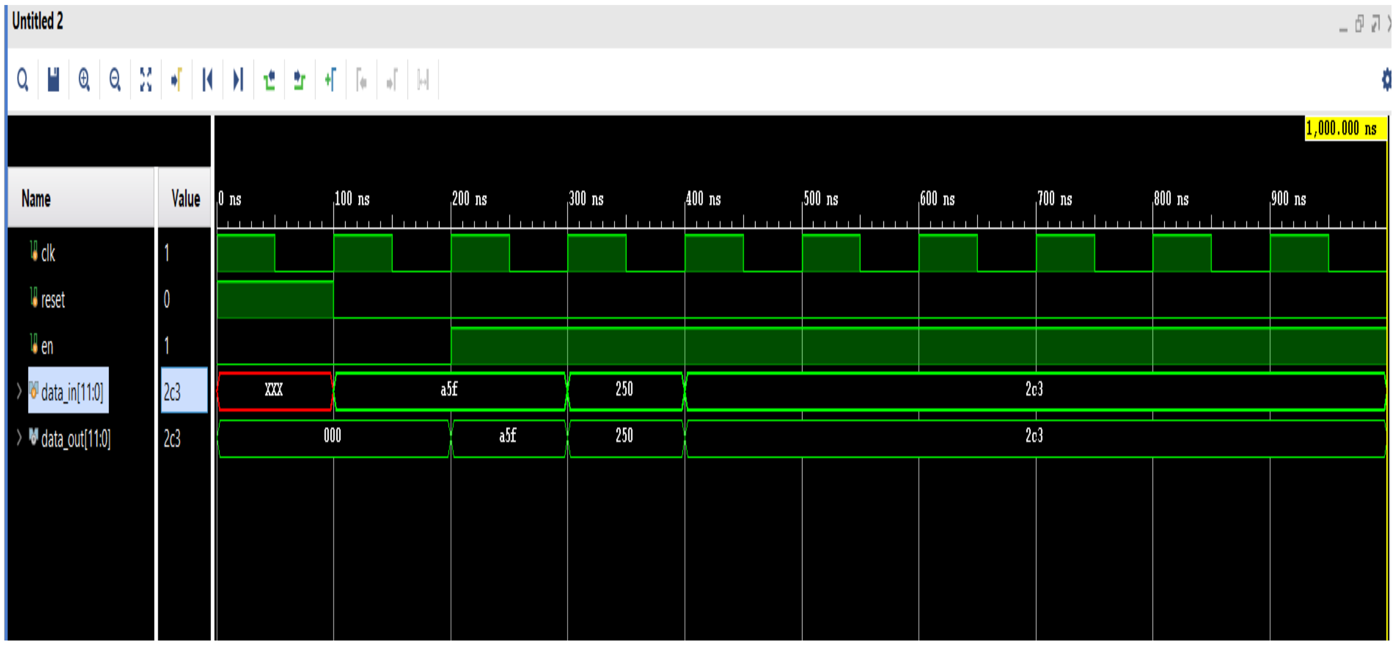


Figure 22: Timing diagram for 12-bit pipeline register

```
module Reg_test();
    reg clk; reg reset; reg en;
    reg [11:0] data_in;
    wire [11:0] data_out;

    reg12Bit Reg_test(.clk(clk), .reset(reset), .en(en), .data_in(data_in), .data_out(data_out));

    initial begin
        clk = 1;
        reset = 1;
        en = 0;

        #100
        reset = 0;
        en = 0;
        data_in = 12'b101001011111;

        #100
        en = 1;
        data_in = 12'b101001011111;

        #100
        data_in = 12'b001001010000;

        #100
        data_in = 12'b001011000011;
    end

    always begin
        #50 clk = ~clk;
    end
endmodule
```

Figure 23: Verilog testbench for 12-bit pipeline register

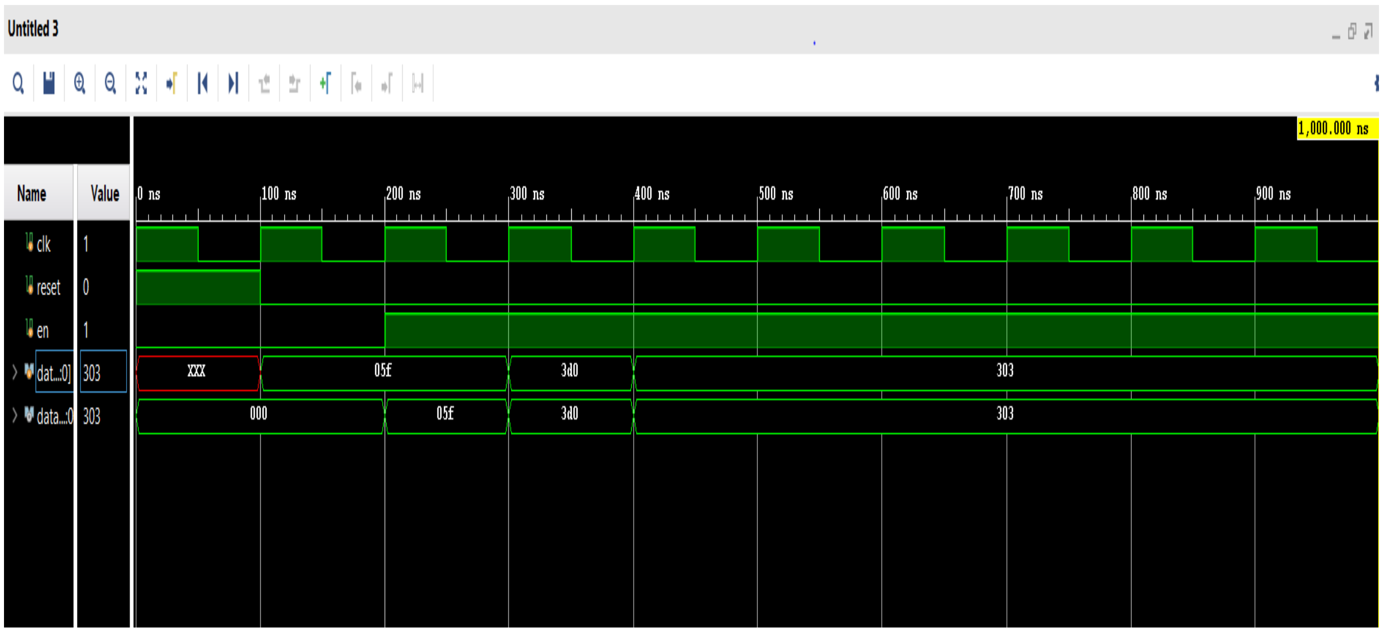


Figure 24: Timing diagram for 10-bit pipeline register

```
module Reg_test();
    reg clk; reg reset; reg en;
    reg [9:0] data_in;
    wire [9:0] data_out;

    reg10Bit Reg_test(.clk(clk), .reset(reset), .en(en), .data_in(data_in), .data_out(data_out));

    initial begin
        clk = 1;
        reset = 1;
        en = 0;

        #100
        reset = 0;
        en = 0;
        data_in = 10'b0001011111;

        #100
        en = 1;
        data_in = 10'b0001011111;

        #100
        data_in = 10'b1111010000;

        #100
        data_in = 10'b1100000011;
    end

    always begin
        #50 clk = ~clk;
    end
endmodule
```

Figure 25: Verilog testbench for 10-bit pipeline register

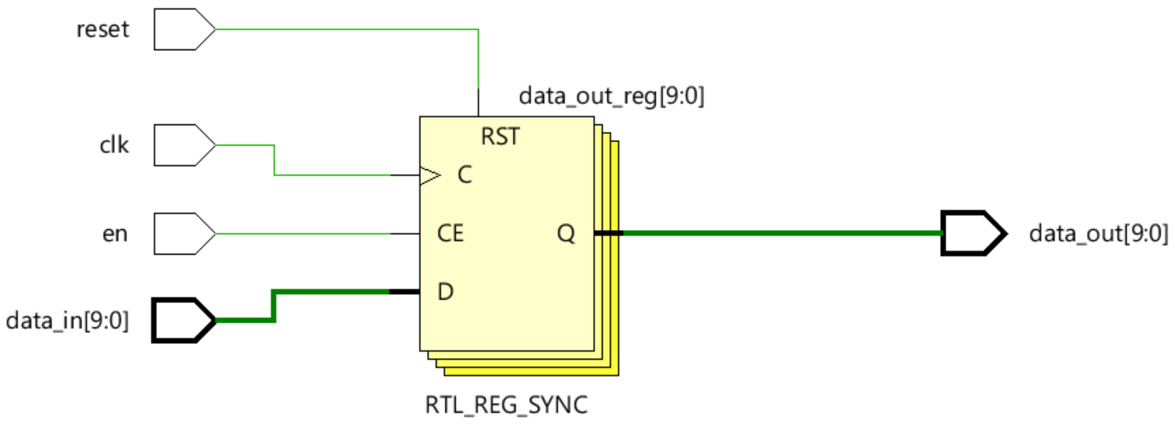


Figure 26: Schematic for 10-bit pipeline register

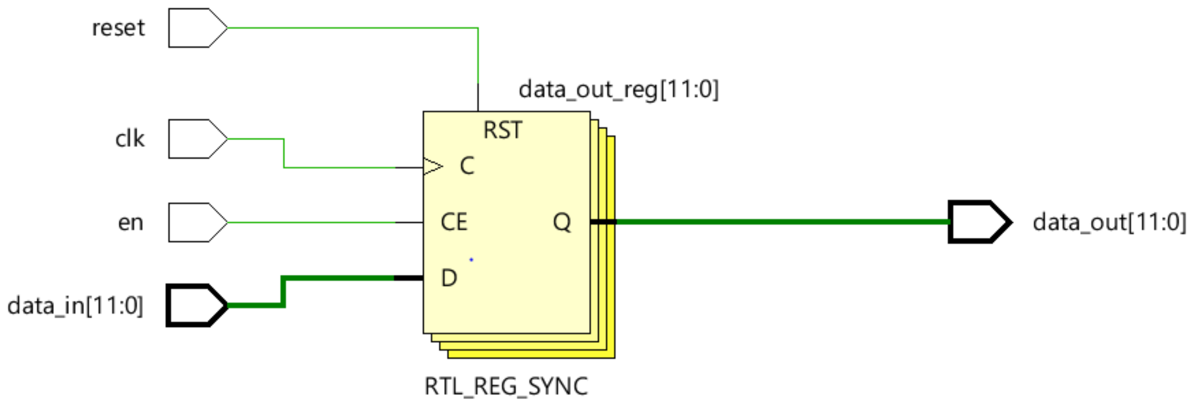


Figure 27: Schematic for 12-bit pipeline register

2. Fetch-Decode Stage

Fetch-Decode stage consists of the instruction memory, fetch unit, control unit and register files module to fetch the instruction and decode it.

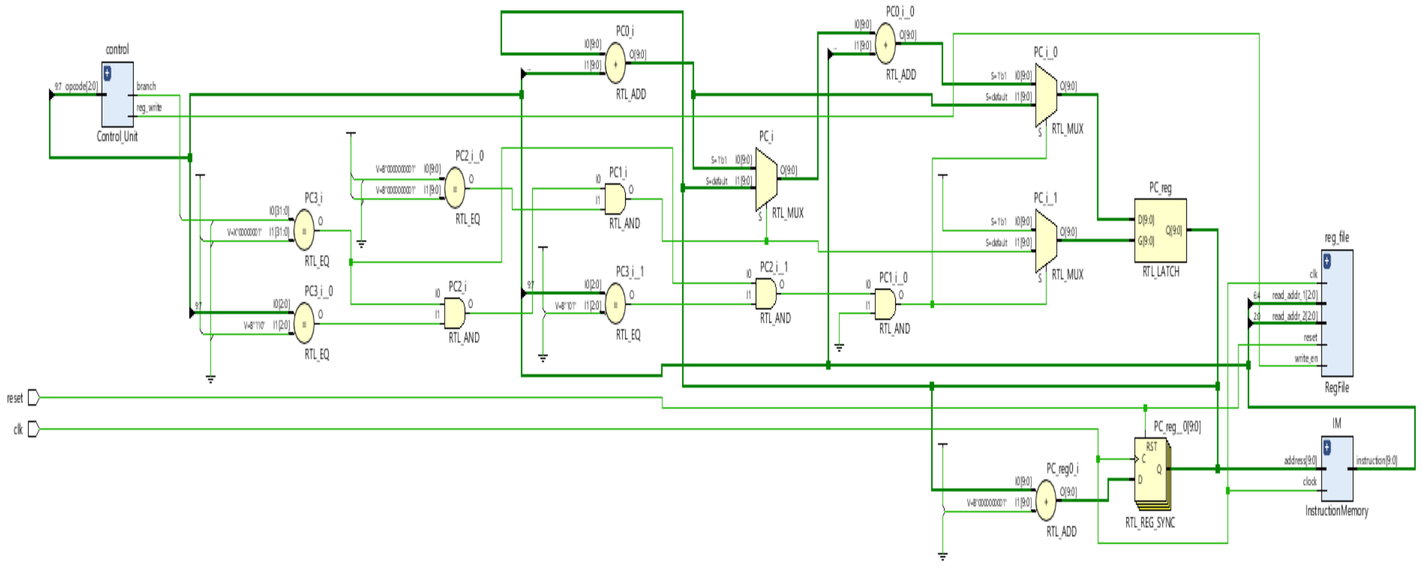


Figure 28: Schematic for Fetch-Decode Stage

3. Execute-Memory Stage

Execute-Memory stage of consists of ALU module and data memory to compute the value and save it into the memory whenever required. It also has special purpose flags or registers.

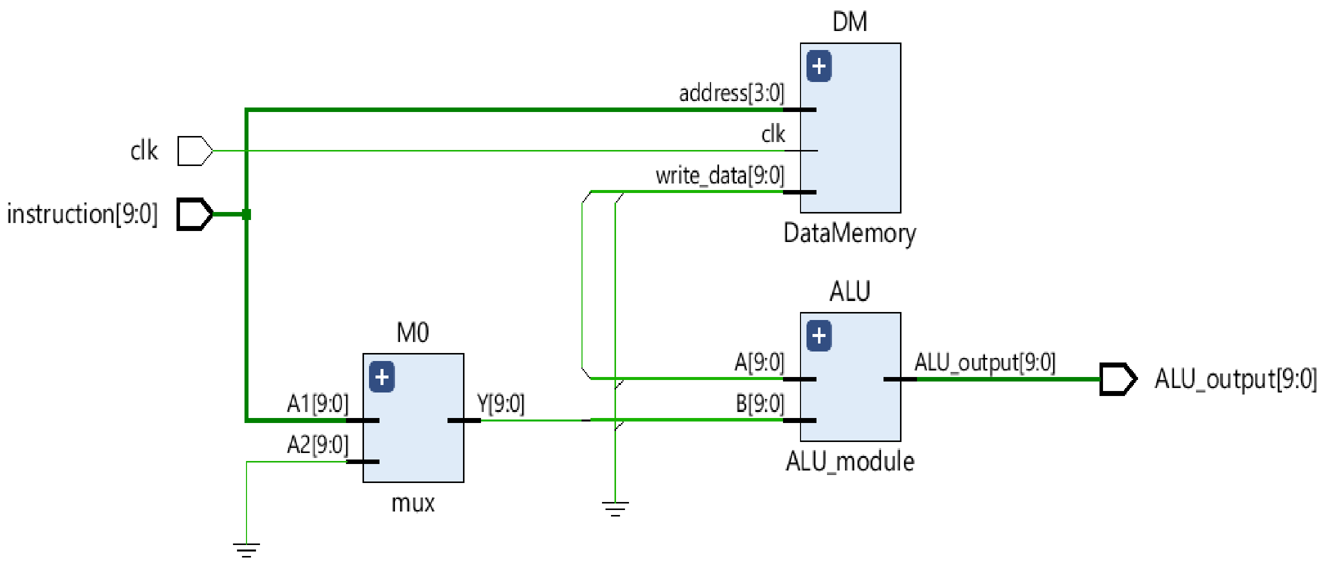


Figure 29: Schematic for Execute-Memory Stage

4. CPU 10-bit pipeline

The Verilog code for 10-bit pipelined CPU is incomplete and I was not able to make it work as needed.

Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/05/2020 08:26:56 AM
// Design Name:
// Module Name: CPU_10bits_pipelined
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module CPU_10bits_pipelined();
endmodule



/*
	This example shows part of a instruction pipeline. Only the first two stages are
given: fetch and decode.
	For this system, there is only two instructions: addi and shift right.
	9-bit instruction format: inst[8] is the opcode, inst[7] selects the register, and
inst[6:0] is the immediate value.
*/

module pipeTB();
	reg clk, rst;

	pipeLine pipe0(clk, rst);
	always #5 clk = ~clk;

	initial begin
		clk = 0;
		rst = 1;

		#15 rst = 0;
	end

endmodule


module pipeLine(clk, rst);
	input clk, rst;
	wire [9:0] decode_out, decode_out_reg;
	wire [9:0] inst, fetch_reg;
	wire [9:0] imm;
	wire ALU_OP, regSel;

	FetchStage fetch0(clk, rst, inst);
	// instead of sending the instruction directly to the next stage, we store it in a
register first
	reg9Bit fetchReg(~clk, rst, 1'b1, inst, fetch_reg);	// notice that the register
stores data on the falling edge of the clock

	DecodeStage d0(fetch_reg, decode_out);
	// create a register the stores all data passed to the next stage
	reg11Bit decodeReg(~clk, rst, 1'b1, decode_out, decode_out_reg);   // notice that
the register stores on the falling edge of the clock

	// decode_out contains 3 signals: ALU_OP, regSel, and the immidiate value/shift
amount
	assign ALU_OP = decode_out_reg[10];
	assign regSel = decode_out_reg[9];
	assign imm = decode_out_reg[8:0];

	// the next stage goes here
endmodule


/*	every clock cycle, this module fetches the next instruction
	from memory and increments the program counter.
*/
module Fetch_Decode_Stage(input clk, input reset); // goes to the instruction memory
```

```
        reg [9:0] PC;
    wire [9:0] address;

        assign address = PC;

        // increment the program counter
        always @(posedge clk) begin
                if (reset) begin
                        PC <= 10'b0000000000;
                end
                else begin
                        PC <= PC + 1;
                end
        end

        wire [9:0] instruction;
        // instruction memory goes here
        InstructionMemory IM(.clock(clk), .address(address), .instruction(instruction));
        /*initial begin
                inst = 9'b000000000;
                #15
                inst = 9'b00000011;     // shift r0, 3

                #10 inst = 9'b100000001;  // add r0, 1
                #10 inst = 9'b111111111;  // add r1, -1
                #10 inst = 9'b010000101;  // shift r1, 5
                #10 inst = 9'b110001111;  // add r1, 15
        end */

        wire [1:0] alu_op;
    wire alu_src;
    wire mem_write;
    wire mem_read;
    wire mem_to_reg;
    wire write_back;
    wire branch;
    wire reg_write;
    wire [1:0] ALU_Control;

        Control_Unit control(.opcode(instruction[9:7]), .alu_op(alu_op),
.alu_src(alu_src), .mem_write(mem_write), .mem_read(mem_read),
                        .mem_to_reg(mem_to_reg), .write_back(write_back), .branch(branch),
.reg_write(reg_write), .ALU_Control(ALU_Control));

    wire [9:0] C0;
    assign C0 = 10'b0000000001;
    always @(branch) begin
        if (branch == 1 && instruction[9:7] == 3'b110 && C0 == 10'b0000000001)
            PC = PC + {{3{instruction[6]}},instruction[6:0]};
        if (branch == 1 && instruction[9:7] == 3'b101 && C0 == 10'b0000000000)
            PC = PC + {{3{instruction[6]}},instruction[6:0]};
        if (instruction[9:7] == 3'b111)
            PC = 10'bxxxxxxxxxx;
    end

    wire [2:0] reg_dst;
    wire [2:0] Reg1;
    wire [2:0] Reg2;
    wire [9:0] Reg_or_Num;
    wire [9:0] write_data;
    wire [9:0] read_data;
    wire [9:0] read_data_1;
    wire [9:0] read_data_2;

    assign Reg1 = (instruction[9:7] == 3'b101 || instruction[9:7] == 3'b110) ? 3'bxxx:
instruction[6:4];
    assign Reg2 = instruction[2:0];

    RegFile reg_file(.clk(clk), .reset(reset), .write_en(reg_write), .reg_write_dest(reg_dst),
.write_data(write_data),
                    .read_addr_1(Reg1), .read_data_1(read_data_1), .read_addr_2(Reg2),
.read_data_2(read_data_2));

endmodule


// In this example, there are only 2 instructions: add register with immediate, and NOP
// There are only 2 registers: r0 and r1.
module Execute_Memory_Stage(input clk, input [9:0] instruction, output [9:0]
ALU_output);

            wire read_data_1;
            wire read_data_2;
            wire alu_src;
            wire Reg_or_Num;
            wire [2:0] ALU_Control;
            wire write_data;
            wire mem_to_reg;
            wire mem_write;
            wire mem_read;
            wire read_data;
            wire C0;
        mux M0(.A1({{6{instruction[3]}},instruction[3:0]}), .A2(read_data_2),
.Sel(alu_src), .Y(Reg_or_Num));

   ALU_module ALU(.ALU_Control(ALU_Control[0]), .A(read_data_1),
.B(Reg_or_Num), .ALU_output(ALU_output));

   DataMemory DM(.clk(clk), .address(instruction[3:0]), .write_data(read_data_1),
.write_en(mem_write), .mem_read(mem_read), .read_data(read_data));

   assign write_data = (mem_to_reg == 1)? read_data: ALU_output;
   assign C0 = (instruction[9:7] == 3'b100)? write_data : C0;
endmodule

module reg12Bit(input clk, reset, en, input [11:0] data_in, output reg [11:0]data_out);

        always @(posedge clk) begin
                if (reset)
                        data_out <= 12'b000000000000;
                else if (en == 1'b1)
                        data_out <= data_in;
        end

endmodule

module reg10Bit(input clk, reset, en, input [9:0] data_in,  output reg [9:0] data_out);

        always @(posedge clk) begin
                if (reset)
                        data_out <= 10'b0000000000;
                else if (en == 1'b1)
                        data_out <= data_in;
        end

endmodule

module InstructionMemory(input clock, input [9:0] address, output reg [9:0] instruction);
   reg [9:0] memory[34:0];
   reg [1:0] program;
   integer i;
   integer mem0 = 0, mem8 = 8, distance = 29;

   initial begin
      program = 2'b10;
      case(program)
      2'b10:
        begin
        memory[0] <= 10'b0000100000;
        memory[1] <= 10'b0000110001;
        memory[2] <= 10'b0101010000;
        memory[3] <= 10'b0111000010;
        memory[4] <= 10'b0100111111;
        memory[5] <= 10'b1000110101;
        memory[6] <= 10'b1101111100;
        memory[7] <= 10'b0101001100;
        memory[8] <= 10'b0011000010;
        memory[9] <= 10'b111xxxxxxx;
        end
      2'b11:
        begin
        i = 4;
        memory[0] <= 10'b0000110000;
        memory[1] <= 10'b1000110010;
        memory[2] <= 10'b1010011101; //beq
        memory[3] <= 10'b0010111000;

        while(i < 32)
            begin
            mem0 = mem0 + 1;
            mem8 = mem8 + 1;
            distance = distance - 4;
            memory[i] = {3'b000, 3'b011, mem0[3:0]};
                i = i + 1;
            memory[i] = 10'b1000110000;
                i = i + 1;
            memory[i] = {3'b101, distance[6:0]};
                i = i + 1;
            memory[i] = {3'b001, 3'b011, mem8[3:0]};
                i = i + 1;
            end
        memory[32] <= 10'b0001001111;
        memory[33] <= 10'b0001001111;
        memory[34] <= 10'b111xxxxxxx;
        end
      endcase
   end
   always @(address)
   begin
      instruction <= memory[address];
   end
endmodule
```

```
module DataMemory(input clk, input [3:0] address, input [9:0] write_data, input
write_en, input mem_read,
          output [9:0] read_data);
    integer i;
    reg [9:0] ram [15:0];
    wire [3:0] ram_address = address;
    reg [1:0] program;

    initial begin
      program = 2'b10;
      case(program)
      2'b10:
        for(i = 0; i < 16; i=i+1)
        if (i == 0 || i == 1)
          begin
            ram[0] <= 10'b0000001000;
            ram[1] <= 10'b0000000101;
          end
        else
          begin
            ram[i] <= 10'b0000000000;
          end
      2'b11:
        for(i = 0; i < 16; i=i+1)
        if (i < 8 || i == 15)
          begin
            ram[0] <= 10'b0000001010;
            ram[1] <= 10'b0000001111;
            ram[2] <= 10'b0000001100;
            ram[3] <= 10'b0000001011;
            ram[4] <= 10'b0000001101;
            ram[5] <= 10'b0000001111;
            ram[6] <= 10'b0000000111;
            ram[7] <= 10'b0000000000;
            ram[15] <= 10'b0000001010;
          end
        else
          begin
            ram[i] <= 10'b0000000000;
          end
      endcase
    end

    always @(posedge clk) begin
      if (write_en)
        ram[ram_address] <= write_data;
        end
        assign read_data = (mem_read) ? ram[ram_address] : 10'b0000000000;
endmodule

module mux(input [9:0] A1, input [9:0] A2, input Sel, output reg [9:0] Y);

    always @(Sel, A1, A2)
        begin
            if (Sel == 0)
            begin
                Y <= A1;
            end
            else if (Sel == 1)
            begin
                Y <= A2;
            end
        end
endmodule

module Control_Unit(input[2:0] opcode,  output reg[1:0] alu_op, reg alu_src,
mem_write, mem_read, mem_to_reg, write_back,
                    branch, reg_write, sign_or_zero, output reg[1:0] ALU_Control);

     always @(alu_op)
     casex (alu_op)
      2'b00: ALU_Control = 2'b00;  //adder ALU
      2'b01: ALU_Control = 2'b01;   // grt ALU
      2'b10: ALU_Control = 2'b10;  //send to data memory
      2'b11: ALU_Control = 2'b11; //halt
      default: ALU_Control = 2'b11;
      endcase

        always @(*)
     begin
          case(opcode)
          3'b000: begin // load
                    alu_op = 2'b10;
                    alu_src = 1'b1;
                    mem_write = 1'b0;
                    mem_read = 1'b1;
                    mem_to_reg = 1'b1;
                    write_back = 1'b1;
                    branch = 1'b0;
                    reg_write = 1'b1;
                    sign_or_zero = 1'b1;
                  end
```

```
3'b001: begin // store
     alu_op = 2'b10;
     alu_src = 1'b1;
     mem_write = 1'b1;
     mem_read = 1'b0;
     mem_to_reg = 1'b0;
     write_back = 1'b0;
     branch = 1'b0;
     reg_write = 1'b0;
     sign_or_zero = 1'b1;
    end
3'b010: begin // addi
     alu_op = 2'b00;
     alu_src = 1'b0;
     mem_write = 1'b0;
     mem_read = 1'b0;
     mem_to_reg = 1'b0;
     write_back = 1'b1;
     branch = 1'b0;
     reg_write = 1'b1;
     sign_or_zero = 1'b1;
    end

3'b011: begin // add
     alu_op = 2'b00;
     alu_src = 1'b1;
     mem_write = 1'b0;
     mem_read = 1'b0;
     mem_to_reg = 1'b0;
     write_back = 1'b1;
     branch = 1'b0;
     reg_write = 1'b1;
     sign_or_zero = 1'b1;
    end

3'b100: begin // grt
     alu_op = 2'b01;
     alu_src = 1'b1;
     mem_write = 1'b0;
     mem_read = 1'b0;
     mem_to_reg = 1'b0;
     write_back = 1'b1;
     branch = 1'b0;
     reg_write = 1'b0;
     sign_or_zero = 1'b1;
    end

3'b101: begin // beq
     alu_op = 2'bxx;
     alu_src = 1'bx;
     mem_write = 1'b0;
     mem_read = 1'b0;
     mem_to_reg = 1'b0;
     write_back = 1'b0;
     branch = 1'b1;
     reg_write = 1'b0;
     sign_or_zero = 1'bx;
    end

3'b110: begin // bne
     alu_op = 2'bxx;
     alu_src = 1'bx;
     mem_write = 1'b0;
     mem_read = 1'b0;
     mem_to_reg = 1'b0;
     write_back = 1'b0;
     branch = 1'b1;
     reg_write = 1'b0;
     sign_or_zero = 1'bx;
     end

3'b111: begin //  halt
     alu_op = 2'b11;
     alu_src = 1'bx;
     mem_write = 1'bx;
     mem_read = 1'bx;
     mem_to_reg = 1'b1;
     write_back = 1'b1;
     branch = 1'bx;
     reg_write = 1'b1;
     sign_or_zero = 1'bx;
     end

default: begin
     alu_op = 2'bxx;
     alu_src = 1'bx;
     mem_write = 1'bx;
     mem_read = 1'bx;
     mem_to_reg = 1'bx;
     write_back = 1'bx;
     branch = 1'bx;
```

```
                    reg_write = 1'bx;
                    sign_or_zero = 1'bx;
                    end
          endcase
          end
endmodule

module RegFile(input clk, input reset, input write_en, input [2:0] reg_write_dest, input
[9:0] write_data,
                input [2:0] read_addr_1, output [9:0] read_data_1, input [2:0] read_addr_2,
output [9:0] read_data_2
 );
      reg [9:0]  reg_array [7:0];
      wire [9:0] C0, T0, R0, R1, R2, R3, R5, R7;
      always @ (posedge clk or posedge reset) begin
           if(reset) begin
                reg_array[0] <= 10'b0;
                reg_array[1] <= 10'b0;
                reg_array[2] <= 10'b0;
                reg_array[3] <= 10'b0;
                reg_array[4] <= 10'b0;
                reg_array[5] <= 10'b0;
                reg_array[6] <= 10'b0;
                reg_array[7] <= 10'b0;
           end
           else begin
                if(write_en) begin
                     reg_array[reg_write_dest] <= write_data;
                end
           end
      end
      assign read_data_1 = ( read_addr_1 == 0)? 10'b0 : reg_array[read_addr_1];
      assign read_data_2 = ( read_addr_2 == 0)? 10'b0 : reg_array[read_addr_2];
      assign C0 = (read_addr_1 == 3'b000) ? reg_array[reg_write_dest] : C0;
      assign T0 = (read_addr_1 == 3'b001) ? reg_array[reg_write_dest] : T0;
      assign R0 = (read_addr_1 == 3'b010) ? reg_array[reg_write_dest] : R0;
      assign R1 = (read_addr_1 == 3'b011) ? reg_array[reg_write_dest] : R1;
      assign R2 = (read_addr_1 == 3'b100) ? reg_array[reg_write_dest] : R2;
      assign R3 = (read_addr_1 == 3'b101) ? reg_array[reg_write_dest] : R3;
      assign R5 = (read_addr_1 == 3'b110) ? reg_array[reg_write_dest] : R5;
      assign R7 = (read_addr_1 == 3'b111) ? reg_array[reg_write_dest] : R7;

 endmodule
```

```
module ALU_module(input ALU_Control, input [9:0] A, input [9:0] B, output wire [9:0]
ALU_output);

		reg Cin;
		initial begin
		   Cin = 0;
		end

		wire Cout, C0;
		wire [9:0] Sum;

		adder_10 ALU1(A, B, Cin, Sum, Cout);
		grt_10 ALU2(A, B, C0);

		mux ALU_mux(.A1(Sum), .A2({000000000,C0}), .Sel(ALU_Control),
.Y(ALU_output));

endmodule

module adder(input A, input B, input Cin, output Sum, output Cout);

      wire w1, w2, w3;
      and(w1, A, B);
      and(w2, A, Cin);
      and (w3, B, Cin);
      or(Cout, w1, w2, w3);
      xor(Sum, A, B, Cin);

endmodule


module adder_10(input [9:0] A, input [9:0] B, input Cin, output [9:0] Sum, output Cout);

      wire [9:1] C;
      adder adder0 (A[0], B[0], Cin, Sum[0], C[1]),
           adder1 (A[1], B[1], C[1], Sum[1], C[2]),
           adder2 (A[2], B[2], C[2], Sum[2], C[3]),
           adder3 (A[3], B[3], C[3], Sum[3], C[4]),
           adder4 (A[4], B[4], C[4], Sum[4], C[5]),
           adder5 (A[5], B[5], C[5], Sum[5], C[6]),
           adder6 (A[6], B[6], C[6], Sum[6], C[7]),
           adder7 (A[7], B[7], C[7], Sum[7], C[8]),
           adder8 (A[8], B[8], C[8], Sum[8], C[9]),
           adder9 (A[9], B[9], C[9], Sum[9], Cout);

endmodule
```

```
module grt_10(input [9:0] A, input [9:0] B, output C0);

    wire G1, G2;
    wire x1, x2, x3, x4;
    wire S1, S2, S3, S4, S2_;
    wire A9_, B9_;

    assign G1 = A > B;
    assign G2 = A < B;

    not(A9_, A[9]);
    not(B9_, B[9]);

    and(S1, A[9], B[9]);
    or(S2, A[9], B[9]);
    and(S3, A[9], B9_);
    and(S4, A9_, B[9]);

    not(S2_, S2);

    and(x1, S2_, G1);
    and(x2, S1, G2);
    and(x3, S3, 0);
    and(x4, S4, 1);

    or(C0, x1, x2, x3, x4);

endmodule
```