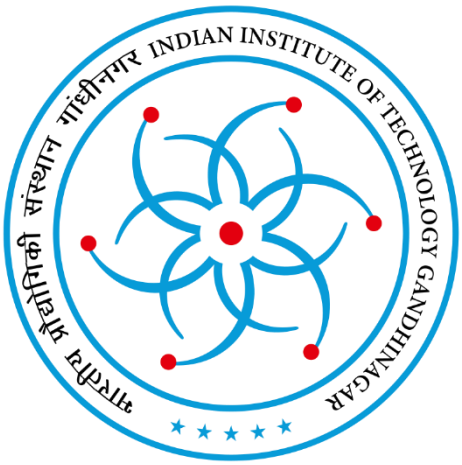
ES 204: DIGITAL SYSTEMS

TAKE HOME LAB ASSIGNMENT 5

DESIGNING AND IMPLEMENTING

OF A TINY PROCESSOR

USING VERILOG



SUBMITTED BY:

ANIRUDH MITTAL (23110029)

ANURAG SINGH (23110035)

TO:

PROF. JOYCEE MEKIE (INSTRUCTOR)

ARNAV (T.A.)

Question

Implementation of the Complete “Tiny” Processor Design, which runs one program. The following processor has a register file consisting of 16 registers, each of 8 bits. The processor can execute the following instructions. The instructions that need 2 operands will take one of the operands from the Register file and another from the accumulator. The result will be transferred to the Accumulator. There is an 8-bit extended (EXT) register used only during multiplication. This register stores the higher-order bits during multiplication and the quotient during division. The C/B register holds the carry and borrow during addition and subtraction, respectively.

1. Register bank is of 16 registers, each of 8 bits.
2. Accumulator, Program counter, etc., are different registers not connected with the bank
3. Branch and Return instructions update the Program counter with the address provided. (These two need not be used to make a code.)
4. The program is stored in memory, which you need to create in the design. Alternatively, you can give the instructions using a testbench.

The Instruction Set Table was given in the PDF.

HELPER CODES FOR ALU UNIT

Full Adder Module:

```
module full_adder(
    input A,
    input B,
    input Cin,
    output SUM,
    output Cout
);
    assign SUM  = A ^ B ^ Cin;
    assign Cout = (A & B) | (B & Cin) | (Cin & A);
endmodule
```

Adder Module Combined:

```
module adder(
    input [7:0] A,
    input [7:0] B,
    output [7:0] SUM,
    output CARRY
);
    wire [7:0] carry;

    full_adder FA0 (A[0], B[0], 1'b0, SUM[0], carry[0]);
    full_adder FA1 (A[1], B[1], carry[0], SUM[1], carry[1]);
    full_adder FA2 (A[2], B[2], carry[1], SUM[2], carry[2]);
    full_adder FA3 (A[3], B[3], carry[2], SUM[3], carry[3]);
    full_adder FA4 (A[4], B[4], carry[3], SUM[4], carry[4]);
    full_adder FA5 (A[5], B[5], carry[4], SUM[5], carry[5]);
    full_adder FA6 (A[6], B[6], carry[5], SUM[6], carry[6]);
```

```
    full_adder FA7 (A[7], B[7], carry[6], SUM[7], carry[7]);

    assign CARRY = carry[7];
endmodule
```

Subtractor Module:

```
module subtractor(
    input [7:0] A,
    input [7:0] B,
    output [7:0] DIFF,
    output BORROW
);
    wire [7:0] B_complement;
    wire [7:0] temp_sum;
    wire carry;

    // Take 2's complement of B => ~B + 1
    assign B_complement = ~B;

    // Use the same adder to compute A + (~B + 1)
    adder ADDER(.A(A), .B(B_complement + 1'b1), .SUM(DIFF), .CARRY(carry));

    assign BORROW = ~carry;  // If carry is not generated, it was a borrow
endmodule
```

Shifter Module:

```
module shifter(D, clk, Q, mode);
    parameter n = 8;
    input [n-1:0] D;
    input clk;
    input [3:0] mode;
    output reg [n-1:0] Q;

    integer k;

    // Mode parameters
    parameter logical_right     = 4'b0010;
    parameter arithematic_right = 4'b0101;
    parameter circular_left     = 4'b0100;
    parameter circular_right    = 4'b0011;
    parameter logical_left      = 4'b0001;

    always @(posedge clk)
    begin
       case (mode)
          logical_right:
          begin
```

```
            for (k = 0; k < n-1; k = k + 1)
               Q[k] <= Q[k+1];
            Q[n-1] <= 1'b0; // fill MSB with 0
         end

         arithematic_right:
         begin
            for (k = 0; k < n-1; k = k + 1)
               Q[k] <= Q[k+1];
            Q[n-1] <= Q[n-1]; // preserve MSB
         end

         circular_left:
         begin
            for (k = n-1; k > 0; k = k - 1)
               Q[k] <= Q[k-1];
            Q[0] <= Q[n-1]; // wrap MSB to LSB
         end

         circular_right:
         begin
            for (k = 0; k < n-1; k = k + 1)
               Q[k] <= Q[k+1];
            Q[n-1] <= Q[0]; // wrap LSB to MSB
         end

         logical_left:
         begin
            for (k = n-1; k > 0; k = k - 1)
               Q[k] <= Q[k-1];
            Q[0] <= 1'b0; // fill LSB with 0
         end

         default:
            Q <= D; // load new data if mode is unknown
      endcase
   end
endmodule
```

Comparator Module:

```
module comparator(
A, B, out
   );
input [7:0] A, B;
output out;

wire [7:0] X;
assign X = A^B;
```

```
assign out = (A >= B) ? 1'b1 : 1'b0;
endmodule
```

Multiplier Module:

```
module multiplier (
   input [7:0] A, // Multiplicand
   input [7:0] B, // Multiplier
   output [15:0] P // Product
);

   // Generate partial products pp[i][j] = A[j] AND B[i]
   wire [7:0] pp [7:0];
   genvar i, j;
   generate
      for (i = 0; i < 8; i = i + 1) begin : gen_pp_rows
         for (j = 0; j < 8; j = j + 1) begin : gen_pp_bits
            assign pp[i][j] = A[j] & B[i];
         end
      end
   endgenerate

   // Shift partial products to align
   wire [15:0] pp_sh [7:0];
   genvar k;
   generate
      for (k = 0; k < 8; k = k + 1) begin : gen_shift
         assign pp_sh[k] = {{8{1'b0}}, pp[k]} << k;
      end
   endgenerate

   // Iteratively add shifted partial products using 8-bit adder module
   wire [15:0] sum_arr [7:0];
   wire carry_lo [7:0];
   wire carry_hi [7:0];

   assign sum_arr[0] = pp_sh[0];

   generate
      for (k = 1; k < 8; k = k + 1) begin : gen_add
         // Lower 8 bits addition
         adder add_lo (
            .A(sum_arr[k-1][7:0]),
            .B(pp_sh[k][7:0]),
            .SUM(sum_arr[k][7:0]),
            .CARRY(carry_lo[k])
         );
         // Upper 8 bits addition with carry
         adder add_hi (
```

```
            .A(sum_arr[k-1][15:8]),
            .B(pp_sh[k][15:8]),
            .SUM(sum_arr[k][15:8]),
            .CARRY(carry_hi[k])
        );
    end
  endgenerate

  // Final product is sum_arr[7]
  assign P = sum_arr[7];

endmodule
```

Increment Module [uses adder]:

```
module adder(
  input [7:0] A,
  input [7:0] B,
  output [7:0] SUM,
  output CARRY
);
  wire [7:0] carry;

  full_adder FA0 (A[0], B[0], 1'b0, SUM[0], carry[0]);
  full_adder FA1 (A[1], B[1], carry[0], SUM[1], carry[1]);
  full_adder FA2 (A[2], B[2], carry[1], SUM[2], carry[2]);
  full_adder FA3 (A[3], B[3], carry[2], SUM[3], carry[3]);
  full_adder FA4 (A[4], B[4], carry[3], SUM[4], carry[4]);
  full_adder FA5 (A[5], B[5], carry[4], SUM[5], carry[5]);
  full_adder FA6 (A[6], B[6], carry[5], SUM[6], carry[6]);
  full_adder FA7 (A[7], B[7], carry[6], SUM[7], carry[7]);

  assign CARRY = carry[7];
endmodule
```

Decrement Module [uses subtractor]:

```
module subtractor(
  input [7:0] A,
  input [7:0] B,
  output [7:0] DIFF,
  output BORROW
);
  wire [7:0] B_complement;
  wire [7:0] temp_sum;
  wire carry;
  assign B_complement = ~B;
  adder ADDER(.A(A), .B(B_complement + 1'b1), .SUM(DIFF), .CARRY(carry));
```

```
    assign BORROW = ~carry;
endmodule
```

SEPARATE MODULE CODES

Memory Module:

```
module memory(data_in, clk, data_out, address, write);

input [7:0]data_in; // writing 8-bit data input to the memory
input write; // enable signal
// enables 1 for writing to memory and 0 for read only

input clk;
input [3:0]address; // 4-bit address
output [7:0]data_out;
reg [7:0] memo [0:15];

assign data_out = memo[address]; // The value written at that address is extracted from the memory
always@(posedge clk)
begin
  if (write) // enabling the write logic
    memo[address] = data_in;
end
endmodule
```

Instruction Memory Module:

```
module instruction_memory(
    input [3:0] address,         // 4-bit address from PC
    output reg [7:0] instruct    // 8-bit instruction output extracting from memory
);

    // writing the 16 bytes (each 8 bits) of the instruction memory, which
   // stores 16 sets of instructions where each instruction is an 8-bit value
    reg [7:0] memory [0:15];

    // Initializing memory with all set of instructions
    initial begin
        memory[0] = 8'b0000_0000; // NOP  No operation
        memory[1] = 8'b0001_0001; // ADD R1  Add value in register R1 to accumulator
        memory[2] = 8'b0010_0010; // SUB R2   Subtract value in R2 from accumulator
        memory[3] = 8'b0011_0011; // MUL R3  Multiply accumulator with value in R3
        memory[4] = 8'b0101_0100; // AND R4   Bitwise AND with R4
        memory[5] = 8'b0110_0101; // XRA R5    Bitwise XOR with R5
```

```
        memory[6] = 8'b0111_0110; // CMP R6    Compare accumulator with R6
        memory[7] = 8'b0000_0001; // LSL ACC  Logical shift left accumulator
        memory[8] = 8'b0000_0010; // LSR ACC  Logical shift right accumulator
        memory[9] = 8'b0000_0011; // CIR ACC  Circular right shift accumulator
        memory[10] = 8'b0000_0100; // CIL ACC Circular left shift accumulator
        memory[11] = 8'b0000_0101; // ASR ACC Arithmetic shift right accumulator
        memory[12] = 8'b0000_0110; // INC ACC  Increment accumulator
        memory[13] = 8'b0000_0111; // DEC ACC  Decrement accumulator
        memory[14] = 8'b1000_0010; // Br 12   Branch to address 2
        memory[15] = 8'b1111_1111; // HLT    Halt the processor
    end

    // Output instruction at the given address
    always @(*) begin
        instruct = memory[address];
    end

endmodule
```

Arithmetic Logic Unit Module:

```
module ALU(opcode, acc, data_reg, clk, acc_out, ext, cb);

input [7:0] opcode; // 8-bit operation code
input [7:0] acc, data_reg;
input clk;
output reg [7:0] acc_out, ext; // defining accumulator output and extended result of multiplier
output reg cb; // defining borrow carry

// Intermediate wires to hold outputs of operations
wire [7:0] add_res, sub_res, and_res, xra_res, shift_res, mul_lower, mul_upper, inc_res, dec_res;
wire add_c, sub_b, comp, inc_c, dec_b;

// Instantiating Operation modules as defined earlier

adder ADD(acc, data_reg, add_res, add_c); // Performs accumulator + data_reg
subtractor SUB(acc, data_reg, sub_res, sub_b); // Performs accumulator - data_reg
assign and_res = acc & data_reg; // Bitwise AND
assign xra_res = acc ^ data_reg; // Bitwise XOR
shifter SHIFT(acc, clk, shift_res, opcode[3:0]);  // Shift operation
comparator COMP(acc, data_reg, comp); // Comparison
multiplier MUL(acc, data_reg, {mul_upper, mul_lower});  // Multiplies accumulator * data_reg
adder inc(acc, 1, inc_res, inc_c); // Increments accumulator by 1
subtractor dec(acc, 1, dec_res, dec_b); // Decrements accumulator by 1


always @(*)
begin
```

```
// Default values
    acc_out = 8'b0;
    ext = 8'b0;
    cb = 1'b0;

// Choosing operation based on the opcode
    casez (opcode)
        8'b0001_????:
        begin
            acc_out = add_res; // ADD
            cb = add_c; // Set carry flag
        end
        8'b0010_????:
        begin
            acc_out = sub_res; // SUB
            cb = sub_b; // Set borrow flag
        end
        8'b0011_????:   // MUL
        begin
            acc_out = mul_lower;  // Lower 8 bits of result
            ext = mul_upper;  // Upper 8 bits of result
        end
       // Shift instructions
        8'b0000_0001:
        begin
            acc_out = shift_res; // Left shift
        end
        8'b0000_0010:
        begin
            acc_out = shift_res; // Right shift
        end
        8'b0000_0011:
        begin
            acc_out = shift_res; // Circular Right
        end
        8'b0000_0100:
        begin
            acc_out = shift_res; // Circular Left
        end
        8'b0000_0101:
        begin
            acc_out = shift_res; // Arithmetic Right Shift
        end
        8'b0101_????:
        begin
            acc_out = and_res; // AND
        end
        8'b0110_????:
        begin
            acc_out = xra_res; // XOR
```

```
        end
        8'b0111_????:
        begin
            acc_out = comp; // COMPARE
        end
        8'b0000_0110:
        begin
            acc_out = inc_res; // Increment
            cb = inc_c; // Carry from increment
        end
        8'b0000_0111: // Decrement
        begin
            acc_out = dec_res;
            cb = dec_b; // Borrow from decrement
        end

        default:
        begin
            acc_out = acc;
        end
    endcase

end
endmodule
```

Main Processor Module

```
module Processor(input clk, input rst);
    // defining wires for different units
    wire [3:0] pc; // program counter output
    wire [7:0] opcode; // operational code instruction taken from memory

// last 4-bit address taken from the opcode to fetch the register from the register bank
// as opcode is YYYY XXXX format where YYYY is instruction and XXXX is register address
   wire [3:0] add = opcode[3:0];

   wire [7:0] data_reg; // data from register file
   wire [3:0] pc_next = pc + 1; // updating the value of program counter

    reg pc_l; // enabling PC load
    reg [3:0] pc_in; // Program counter input value
    program_counter prct(
        .clk(clk),
        .rst(rst),
        .load(pc_l),
        .pc_in(pc_in),
        .pc_out(pc)
    );
```

```
    // Calling Instruction Memory
    // Extracting the instruction from the memory using the PC address
    instruction_memory inst_mem(
        .address(pc),
        .instruct(opcode)
    );

    // Register File
    reg [7:0] data_in; // Data value
    reg write; // Signal to control the writing of data
    memory reg_file(
        .data_in(data_in),
        .data_out(data_reg),
        .address(add),
        .write(write)
    );

    // Calling Internal Registers
    reg [7:0] acc; // Accumulator register
    reg [7:0] ext; // External Register
    reg cb; // Borrow Carry Register
    reg acc_ld, ext_ld, cb_ld; // Control signal for internal signals

    // ALU outputs
    wire [7:0] alu_acc_out; // Output which will be stored in accumulator
    wire [7:0] alu_ext_out; // Output which will be stored in accumulator
    wire alu_cb_out; // Output which will be stored in the accumulator

    ALU alu(.opcode(opcode), .acc(acc), .data_reg(data_reg), .clk(clk), .acc_out(alu_acc_out),
.ext(alu_ext_out), .cb(alu_cb_out));

    reg [7:0] acc_next_value; // Accumulator value to be taken in next clock cycle

    // Combinational control logic
    always @(*) begin
        // Default values
        acc_ld = 0;
        ext_ld = 0;
        cb_ld  = 0;
        write  = 0;
        data_in = 8'h00;
        pc_l = 1;
        pc_in = pc_next;
        acc_next_value = alu_acc_out;

        // Decoding Instructions
        casez (opcode)
            // NOP
            8'b0000_0000: ;
            // ADD
```

```
        8'b0001_????: begin acc_ld=1; cb_ld=1; end
        // SUB
        8'b0010_????: begin acc_ld=1; cb_ld=1; end
        // MUL
        8'b0011_????: begin acc_ld=1; ext_ld=1; end
        // ASR
        8'b0000_0001,
        8'b0000_0010,
        8'b0000_0011,
        8'b0000_0100,
        8'b0000_0101: begin acc_ld=1; end
        // XOR
        8'b0110_????: begin acc_ld=1; end
        // Comparator
        8'b0111_????: begin cb_ld=1; end
        // Increment
        8'b0000_0110: begin acc_ld=1; cb_ld=1; end
        // Decrement
        8'b0000_0111: begin acc_ld=1; cb_ld=1; end
         // AND
        8'b0101_????: begin acc_ld=1; end

        // Performing Conditional Branching, i.e. branch if carry-borrow is set.
        8'b1000_????:
         if (cb)
begin
pc_l=1;
pc_in=add;
end
        // Loading data from Register to Accumulator
        8'b1001_????:
        begin
                acc_ld=1;
                acc_next_value = data_reg; // Source for ACC is data_reg, not ALU output
                end
     // Storing the accumulator value to register
        8'b1010_????:
            begin
write=1;
data_in=acc;
    end

        8'b1011_????:
                begin
pc_l=1;
pc_in=add;
end                     // RET (Assuming RET is like JMP)
        8'b1111_1111: begin
           pc_l = 0; // Disabling PC update
        end
```

```
        default: ; // Treat unrecognized opcodes as NOPs
      endcase
   end

   always @(posedge clk or posedge rst) begin
      if (rst) begin
         acc <= 8'b0;
         ext <= 8'b0;
         cb  <= 1'b0;
      end else begin
         // Loading ACC from the selected source
         if (acc_ld) acc <= acc_next_value;
         if (ext_ld) ext <= alu_ext_out;
         if (cb_ld ) cb  <= alu_cb_out;
         // Register file write happens inside the memory module, triggered by 'write' signal only.
      end
   end

endmodule
```

SEPARATE TESTBENCH CODES TO CHECK MODULES

Testbench of ALU:

```
module ALU_tb();

reg [7:0] opcode;
reg [7:0] acc;
reg [7:0] data_reg;
reg clk;
wire [7:0] acc_out;
wire [7:0] ext;
wire cb;

// Instantiating the ALU
ALU uut (
   .opcode(opcode),
   .acc(acc),
   .data_reg(data_reg),
   .clk(clk),
   .acc_out(acc_out),
   .ext(ext),
   .cb(cb)
);

initial begin
clk = 0;
forever #10 clk=~clk;
end
```

```
initial begin

    // ADD
    acc = 8'd10; data_reg = 8'd15; opcode = 8'b0001_0000; #10;

    // SUB
    acc = 8'd30; data_reg = 8'd12; opcode = 8'b0010_0000; #10;

    // MUL
    acc = 8'd7; data_reg = 8'd6; opcode = 8'b0011_0000; #10;

    // AND
    acc = 8'b11001100; data_reg = 8'b10101010; opcode = 8'b0101_0000; #10;

    // XOR
    acc = 8'b11001100; data_reg = 8'b10101010; opcode = 8'b0110_0000; #10;

    // Comparator
    acc = 8'd50; data_reg = 8'd100; opcode = 8'b0111_0000; #10;

    // INC
    acc = 8'd200; data_reg = 8'd0; opcode = 8'b0000_0110; #10;

    // DEC
    acc = 8'd100; data_reg = 8'd0; opcode = 8'b0000_0111; #10;

    // LSL
    acc = 8'b00001111; data_reg = 8'b0; opcode = 8'b0000_0001; #10;

    // LSR
    acc = 8'b11110000; data_reg = 8'b0; opcode = 8'b0000_0010;

end
endmodule
```

Testbench of Instruction Memory:

```
module instruction_memory_tb();
reg [3:0] address;
wire [7:0] instruction;

// Instantiate the Instruction Memory
instruction_memory uut (.address(address), .instruct(instruction));

initial begin

    address = 4'd0; #10;
    address = 4'd1; #10;
```

```
    address = 4'd2; #10;
    address = 4'd3; #10;
    address = 4'd4; #10;
    address = 4'd5; #10;
    address = 4'd6; #10;
    address = 4'd15;
end
endmodule
```

Testbench of Memory(Registers):

```
module memory_tb();

reg [7:0]data_in;
reg write , clk;
reg [3:0]address;

wire [7:0]data_out;

memory uut( .data_in(data_in) , .data_out(data_out) , .write(write), .clk(clk) , .address(address));

initial begin
clk = 0;
forever #5
clk = ~ clk;
end

initial begin
data_in  = 8'd11 ; write = 0 ; address = 4'd9; #10;
data_in  = 8'd19 ; write = 1 ; address = 4'd9; #10;
data_in  = 8'd29 ; write = 0 ; address = 4'd9; #10;
data_in  = 8'd01 ; write = 1 ; address = 4'd9; #10;
data_in  = 8'd29 ; write = 1 ; address = 4'd9; #10;
data_in  = 8'd9 ; write = 0 ; address = 4'd9;

end
endmodule
```

PROCESSOR TESTBENCH CODES

Since in one test bench only 16 instructions can run, the other instructions are shown in the follow-up test bench [2].

TestBench to Show all the Instructions Running [1]

```
module Processor_tb2();
reg clk;
```

```
reg rst;

Processor uut(
    .clk(clk),
    .rst(rst)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin

    uut.reg_file.memo[0] = 8'd1;         // R0 = 1 (Decimal)
    uut.reg_file.memo[1] = 8'd5;         // R1 = 5 (Decimal)
    uut.reg_file.memo[2] = 8'd10;        // R2 = 10 (Decimal)
    uut.reg_file.memo[3] = 8'd3;         // R3 = 3 (Decimal)
    uut.reg_file.memo[4] = 8'd200;         // R4 = 2 (Decimal)
    uut.reg_file.memo[5] = 8'd19;        // R5 = 19 (Decimal)
    uut.reg_file.memo[6] = 8'd10;        // R6 = 10 (Decimal)
    uut.reg_file.memo[7] = 8'b10101010;  // R7 = 170 (Decimal) or AA (Hex)
    uut.reg_file.memo[8] = 8'b01010101;  // R8 = 85 (Decimal) or 55 (Hex)
    uut.reg_file.memo[9] = 8'd10;         // R9 = 0
    uut.reg_file.memo[10] = 8'd04;        // R10 = 0
    uut.reg_file.memo[11] = 8'd20;        // R11 = 0
    uut.reg_file.memo[12] = 8'd40;        // R12 = 0
    uut.reg_file.memo[13] = 8'd26;        // R13 = 0
    uut.reg_file.memo[14] = 8'd19;        // R14 = 0
    uut.reg_file.memo[15] = 8'd16;        // R15 = 0

end



// Instruction Memory Initialization
initial begin
    uut.inst_mem.memory[0] = 8'b0000_0000;
    uut.inst_mem.memory[1] = 8'b0001_0001;
    uut.inst_mem.memory[2] = 8'b0010_0010;
    uut.inst_mem.memory[3] = 8'b0011_0011;
    uut.inst_mem.memory[4] = 8'b0101_0100;
    uut.inst_mem.memory[5] = 8'b0110_0101;
    uut.inst_mem.memory[6] = 8'b0111_0110;
    uut.inst_mem.memory[7] = 8'b0000_0001;
    uut.inst_mem.memory[8] = 8'b0000_0010;
    uut.inst_mem.memory[9] = 8'b0000_0011;
    uut.inst_mem.memory[10] = 8'b0000_0100;
    uut.inst_mem.memory[11] = 8'b0000_0101;
    uut.inst_mem.memory[12] = 8'b0000_0110;
```

```
    uut.inst_mem.memory[13] = 8'b0000_0111;
    uut.inst_mem.memory[14] = 8'b1000_0010;
    uut.inst_mem.memory[15] = 8'b1111_1111;
end

initial begin

    rst = 1;
    #10;
    rst = 0;
    #1000;
end

endmodule
```

TestBench to Show all the Instructions Running [2]

```
module Processor_tb3();
reg clk;
reg rst;

Processor uut(
    .clk(clk),
    .rst(rst)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin

    uut.reg_file.memo[0] = 8'd1;         // R0 = 1 (Decimal)
    uut.reg_file.memo[1] = 8'd5;         // R1 = 5 (Decimal)
    uut.reg_file.memo[2] = 8'd10;        // R2 = 10 (Decimal)
    uut.reg_file.memo[3] = 8'd3;         // R3 = 3 (Decimal)
    uut.reg_file.memo[4] = 8'd200;       // R4 = 2 (Decimal)
    uut.reg_file.memo[5] = 8'd19;        // R5 = 19 (Decimal)
    uut.reg_file.memo[6] = 8'd10;        // R6 = 10 (Decimal)
    uut.reg_file.memo[7] = 8'b10101010;  // R7 = 170 (Decimal) or AA (Hex)
    uut.reg_file.memo[8] = 8'b01010101;  // R8 = 85 (Decimal) or 55 (Hex)
    uut.reg_file.memo[9] = 8'd10;        // R9 = 0
    uut.reg_file.memo[10] = 8'd04;       // R10 = 0
    uut.reg_file.memo[11] = 8'd20;       // R11 = 0
    uut.reg_file.memo[12] = 8'd40;       // R12 = 0
    uut.reg_file.memo[13] = 8'd26;       // R13 = 0
    uut.reg_file.memo[14] = 8'd19;       // R14 = 0
    uut.reg_file.memo[15] = 8'd16;       // R15 = 0
```

```
end


// Instruction Memory Initialization
initial begin
    uut.inst_mem.memory[0] = 8'b0000_0000;
    uut.inst_mem.memory[1] = 8'b0001_0001;
    uut.inst_mem.memory[2] = 8'b0010_0010;
    uut.inst_mem.memory[3] = 8'b0011_0011;
    uut.inst_mem.memory[4] = 8'b1010_0101;
    uut.inst_mem.memory[5] = 8'b1011_0110;
    uut.inst_mem.memory[6] = 8'b1000_0010;
    uut.inst_mem.memory[7] = 8'b0000_0001;
    uut.inst_mem.memory[8] = 8'b0000_0010;
    uut.inst_mem.memory[9] = 8'b0000_0011;
    uut.inst_mem.memory[10] = 8'b0000_0100;
    uut.inst_mem.memory[11] = 8'b0000_0101;
    uut.inst_mem.memory[12] = 8'b0000_0110;
    uut.inst_mem.memory[13] = 8'b0000_0111;
    uut.inst_mem.memory[14] = 8'b1000_0010;
    uut.inst_mem.memory[15] = 8'b1111_1111;
end

initial begin
    rst = 1;
    #10;
    rst = 0;
    #1000;

end
endmodule
```

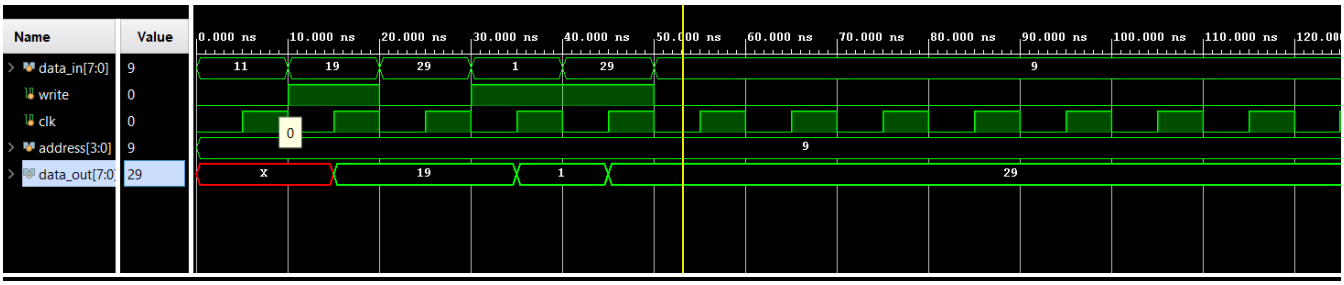
TestBench to Show Specific Program:

```
module Processor_tb();
reg clk;
reg rst;

Processor uut(
    .clk(clk),
    .rst(rst)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
```

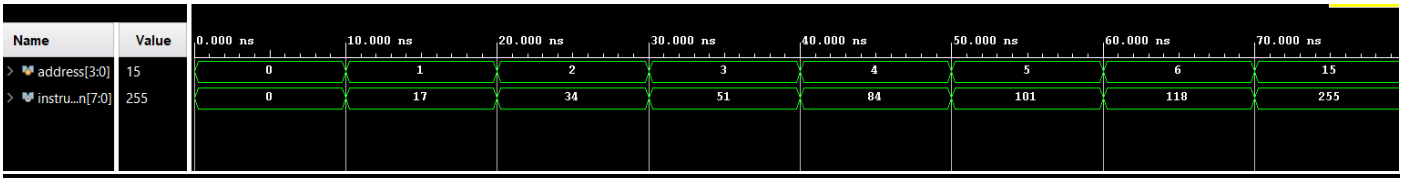
```
initial begin
uut.reg_file.memo[0] = 8'd1;         // R0 = 1 (Decimal)
   uut.reg_file.memo[1] = 8'd5;        // R1 = 5 (Decimal)
   uut.reg_file.memo[2] = 8'd10;       // R2 = 10 (Decimal)
   uut.reg_file.memo[3] = 8'd3;        // R3 = 3 (Decimal)
   uut.reg_file.memo[4] = 8'd200;        // R4 = 2 (Decimal)
   uut.reg_file.memo[5] = 8'd19;       // R5 = 19 (Decimal)
   uut.reg_file.memo[6] = 8'd10;       // R6 = 10 (Decimal)
   uut.reg_file.memo[7] = 8'b10101010;  // R7 = 170 (Decimal) or AA (Hex)
   uut.reg_file.memo[8] = 8'b01010101;  // R8 = 85 (Decimal) or 55 (Hex)
   uut.reg_file.memo[9] = 8'd10;        // R9 = 0
   uut.reg_file.memo[10] = 8'd04;       // R10 = 0
   uut.reg_file.memo[11] = 8'd20;       // R11 = 0
   uut.reg_file.memo[12] = 8'd40;       // R12 = 0
   uut.reg_file.memo[13] = 8'd26;       // R13 = 0
   uut.reg_file.memo[14] = 8'd19;       // R14 = 0
   uut.reg_file.memo[15] = 8'd16;       // R15 = 0

end

// Instruction Memory Initialization
initial begin
   uut.inst_mem.memory[0] = 8'b1001_0001;  // ADD R1       -> ACC = ACC + R1
   uut.inst_mem.memory[1] = 8'b0001_0010;  // SUB R2       -> ACC = ACC - R2
   uut.inst_mem.memory[2] = 8'b0010_0011;  // MOV ACC, R3  -> ACC = R3
   uut.inst_mem.memory[3] = 8'b0011_0100;  // MOV R4, ACC  -> R4 = ACC
   uut.inst_mem.memory[4] = 8'b1000_0011;  // XRA R5       -> ACC ^= R5
   uut.inst_mem.memory[5] = 8'b1111_1111;  // BR R9        -> if CB = 1, jump to R9
   uut.inst_mem.memory[6] = 8'b0000_0000;  // HLT          -> Stop execution
   uut.inst_mem.memory[7] = 8'b0000_0000;
   uut.inst_mem.memory[8] = 8'b0000_0000;
   uut.inst_mem.memory[9] = 8'b0000_0000;
   uut.inst_mem.memory[10] = 8'b0000_0000;
   uut.inst_mem.memory[11] = 8'b0000_0000;
   uut.inst_mem.memory[12] = 8'b0000_0000;
   uut.inst_mem.memory[13] = 8'b0000_0000;
   uut.inst_mem.memory[14] = 8'b0000_0000;
   uut.inst_mem.memory[15] = 8'b0000_0000;
end

initial begin
   rst = 1;
   #10;
   rst = 0;
   #1000;
end
endmodule
```

SIMULATION RESULTS

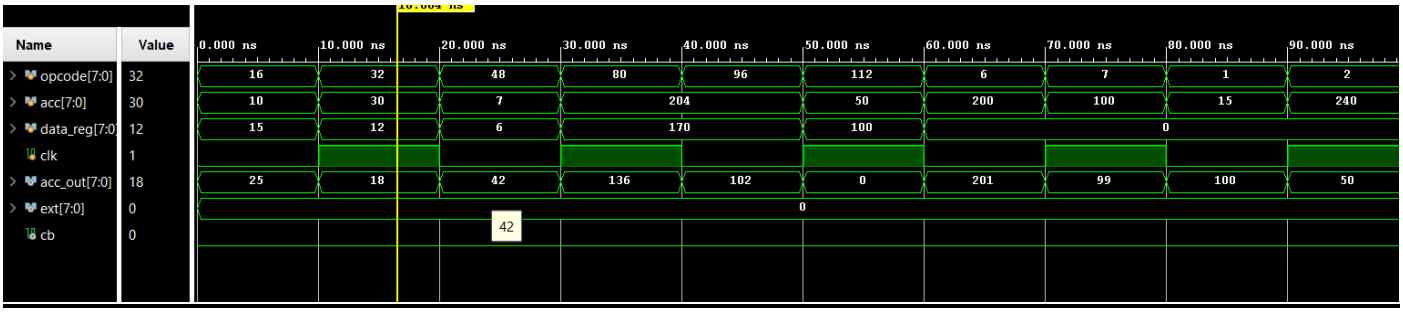
Memory (Register) Testbench



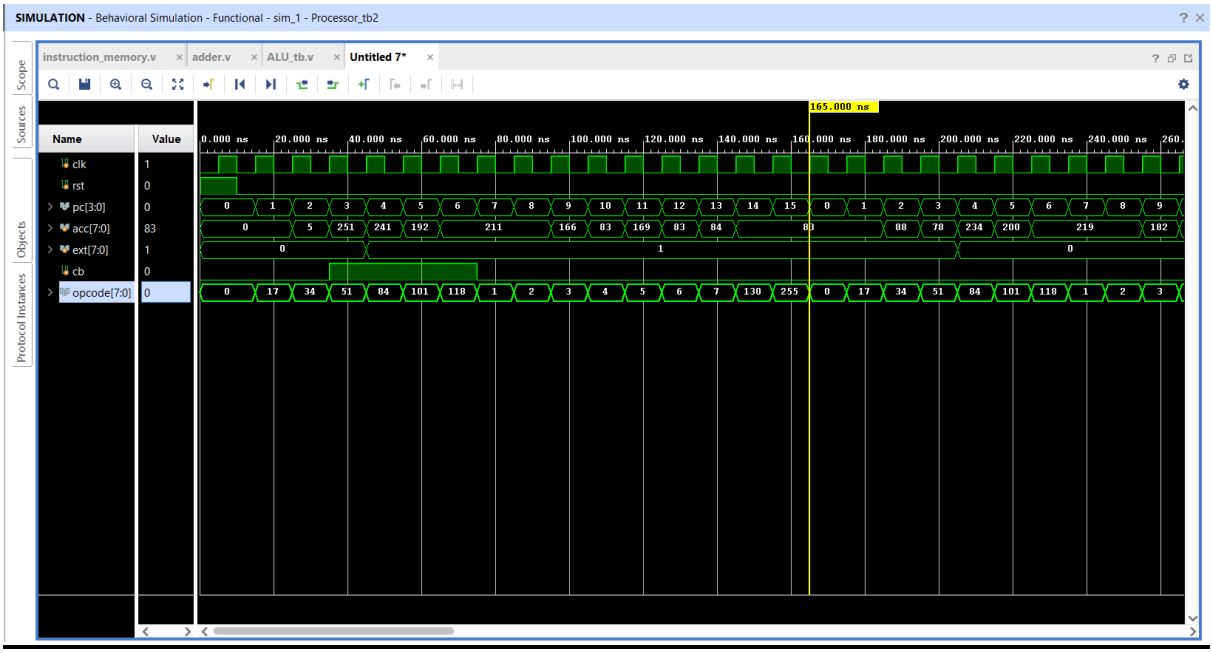
Instruction Memory Testbench

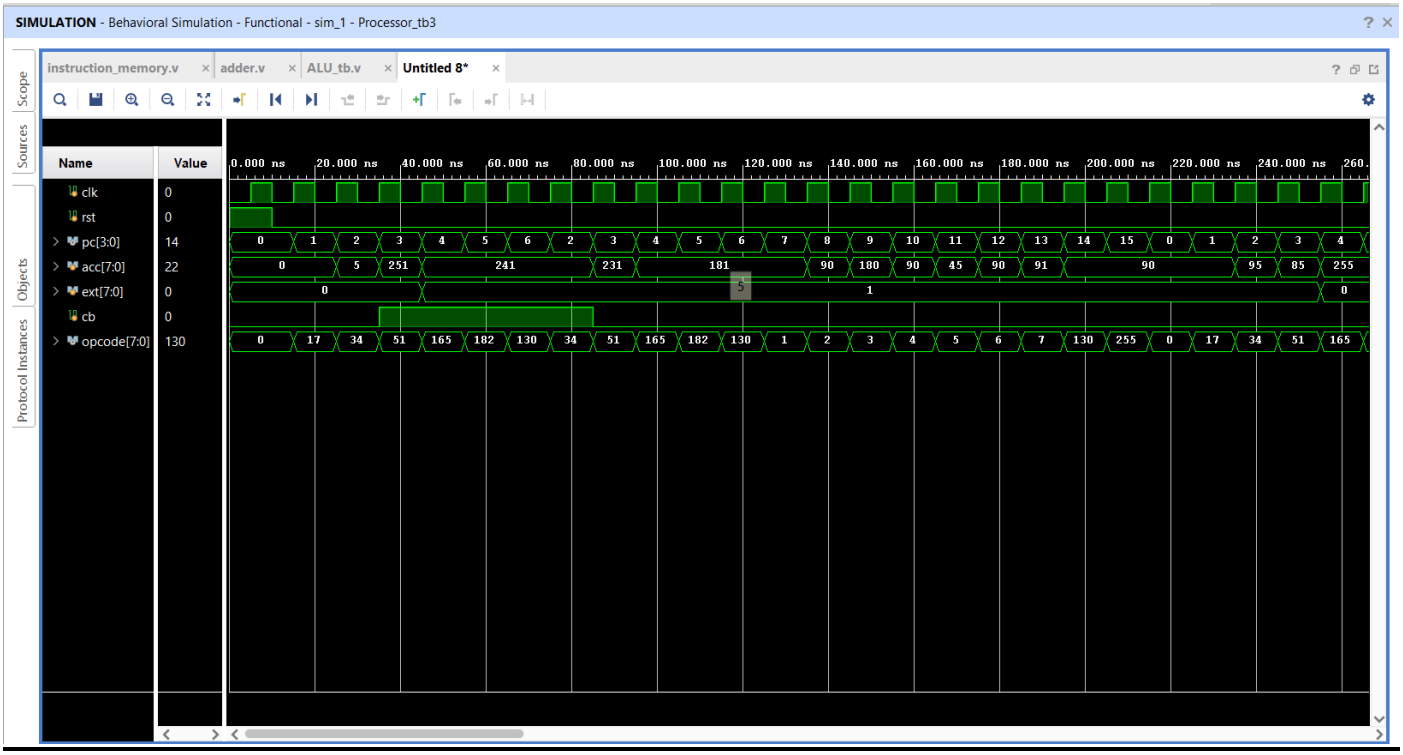


ALU Testbench

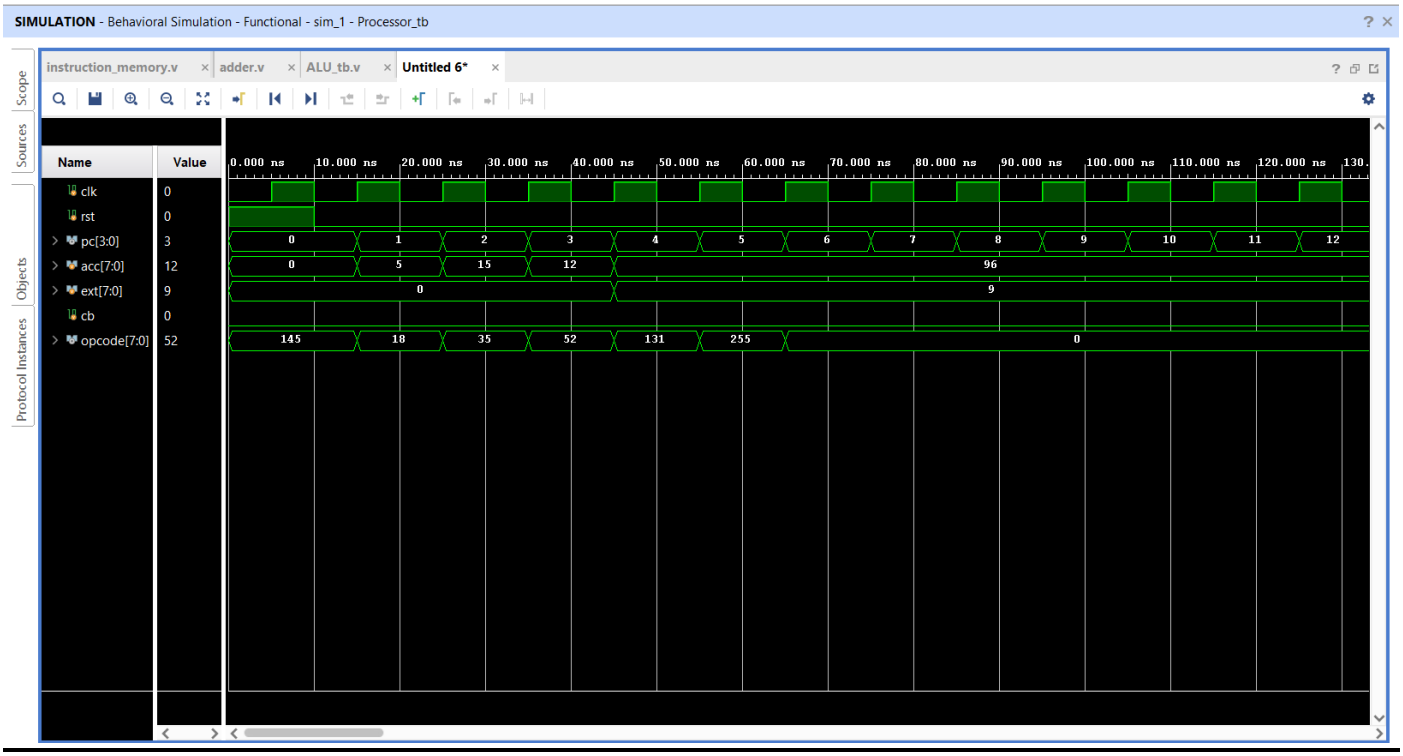


Main Testbench Showing All Instructions

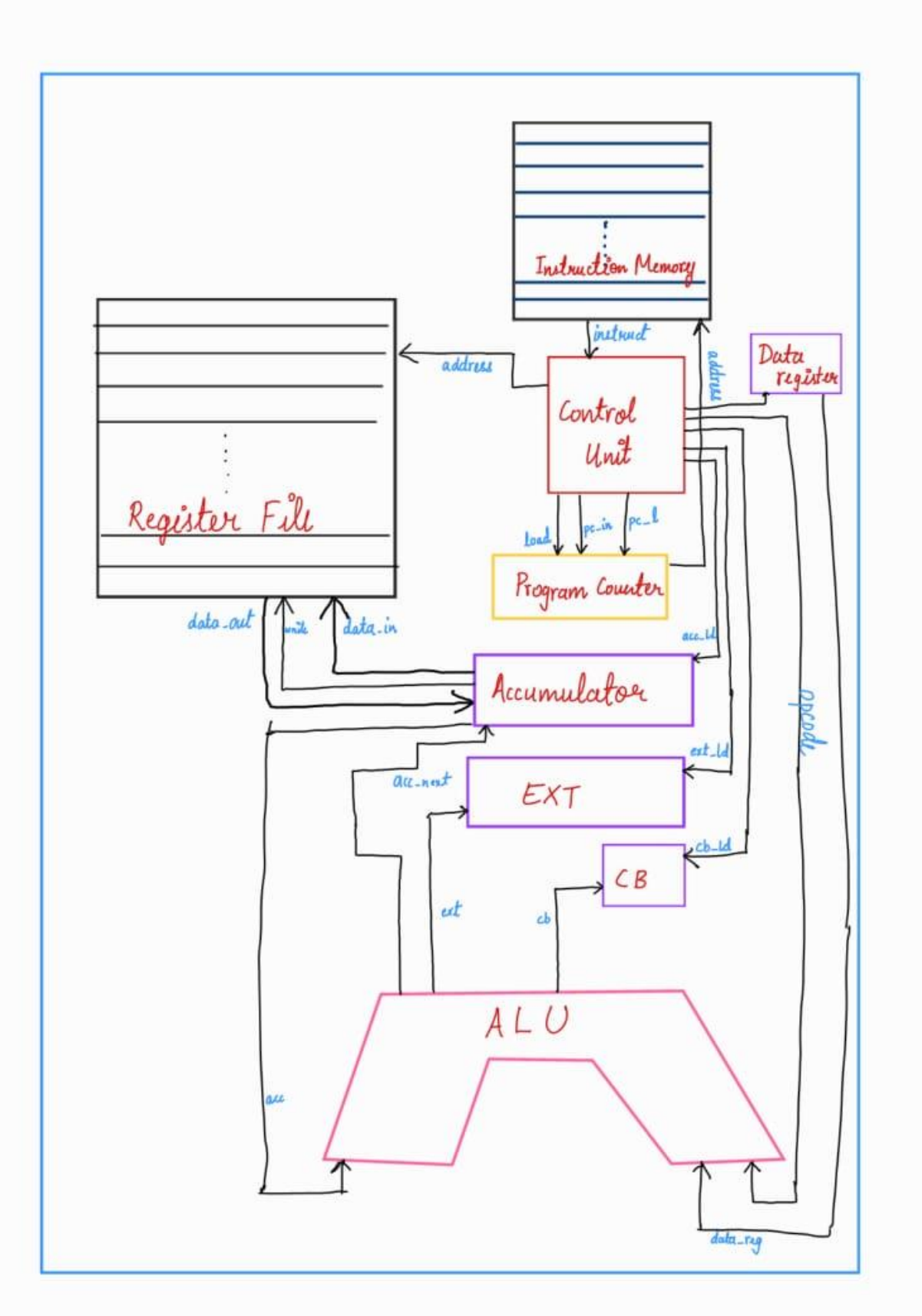




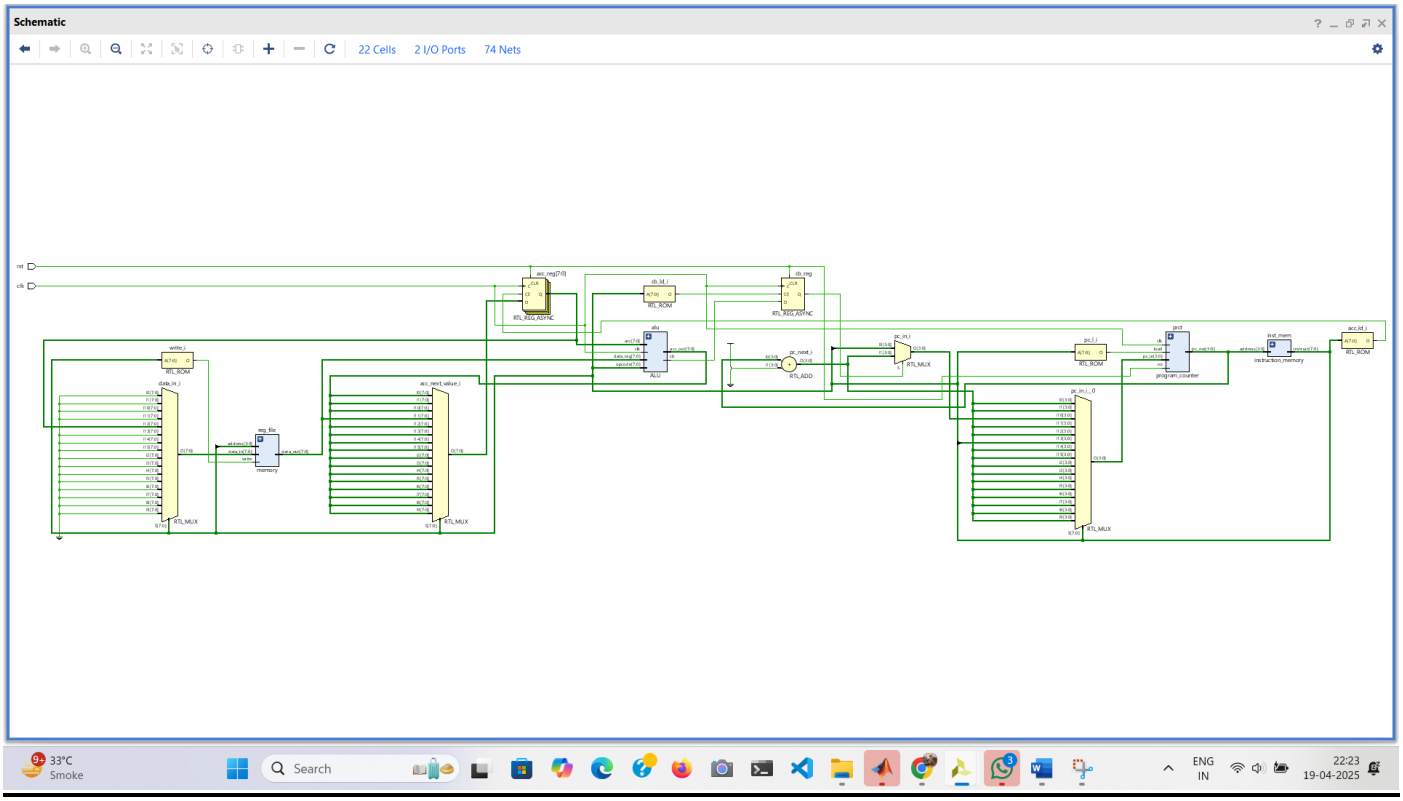
Simulation for Specific Program



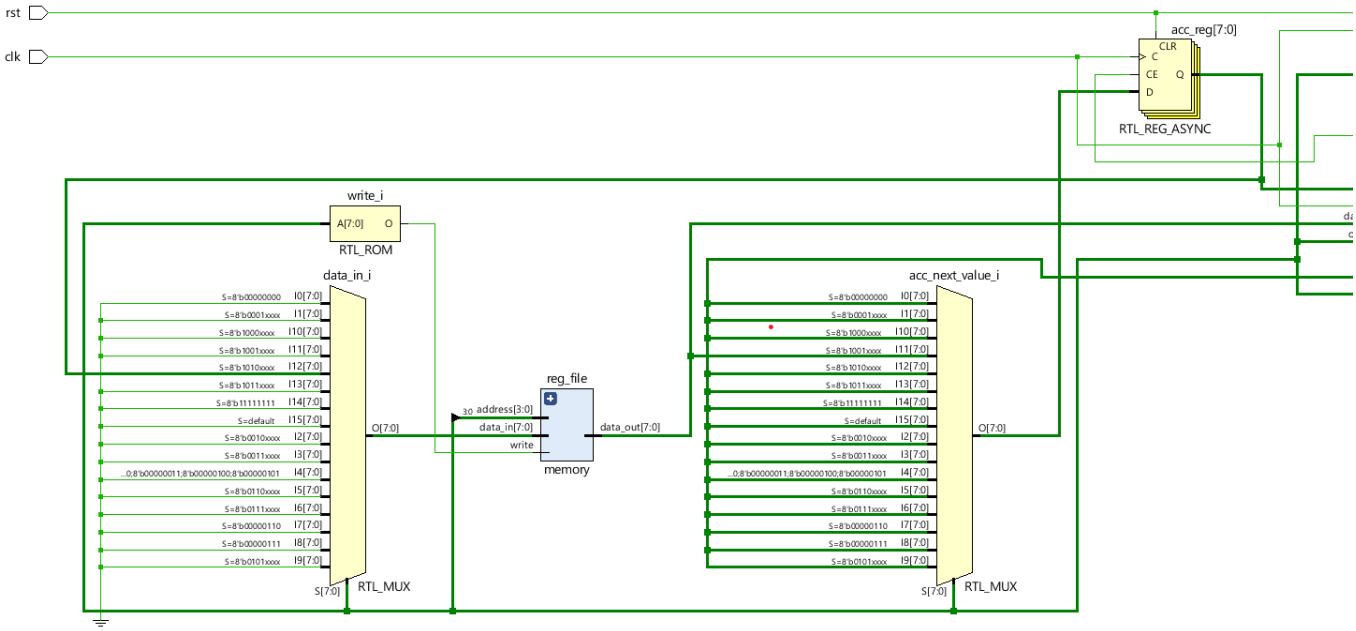
BLOCK DIAGRAM

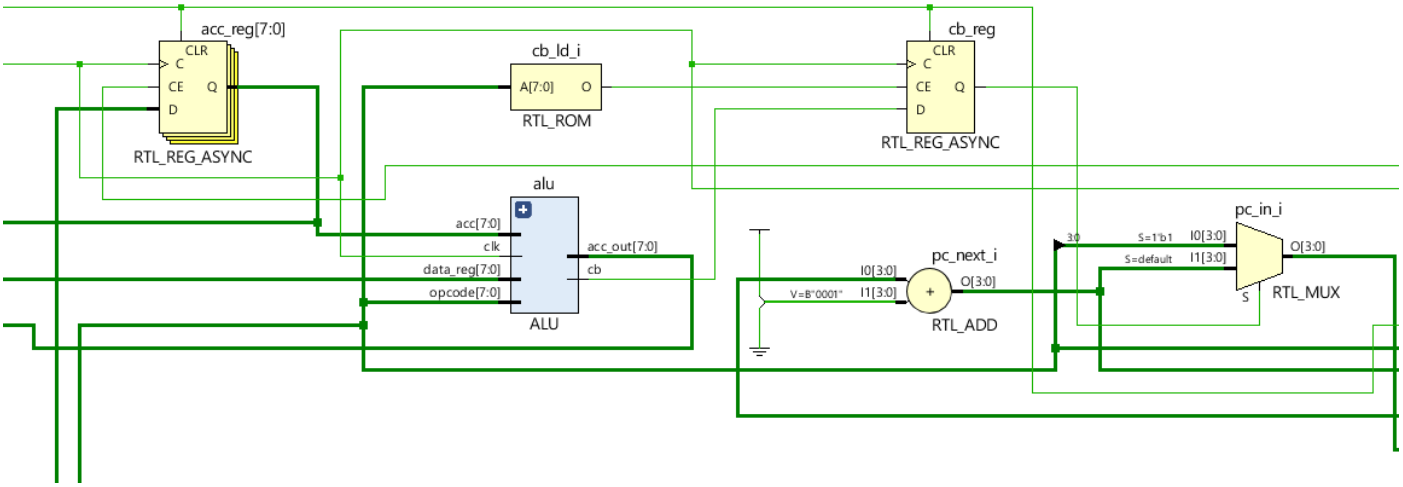


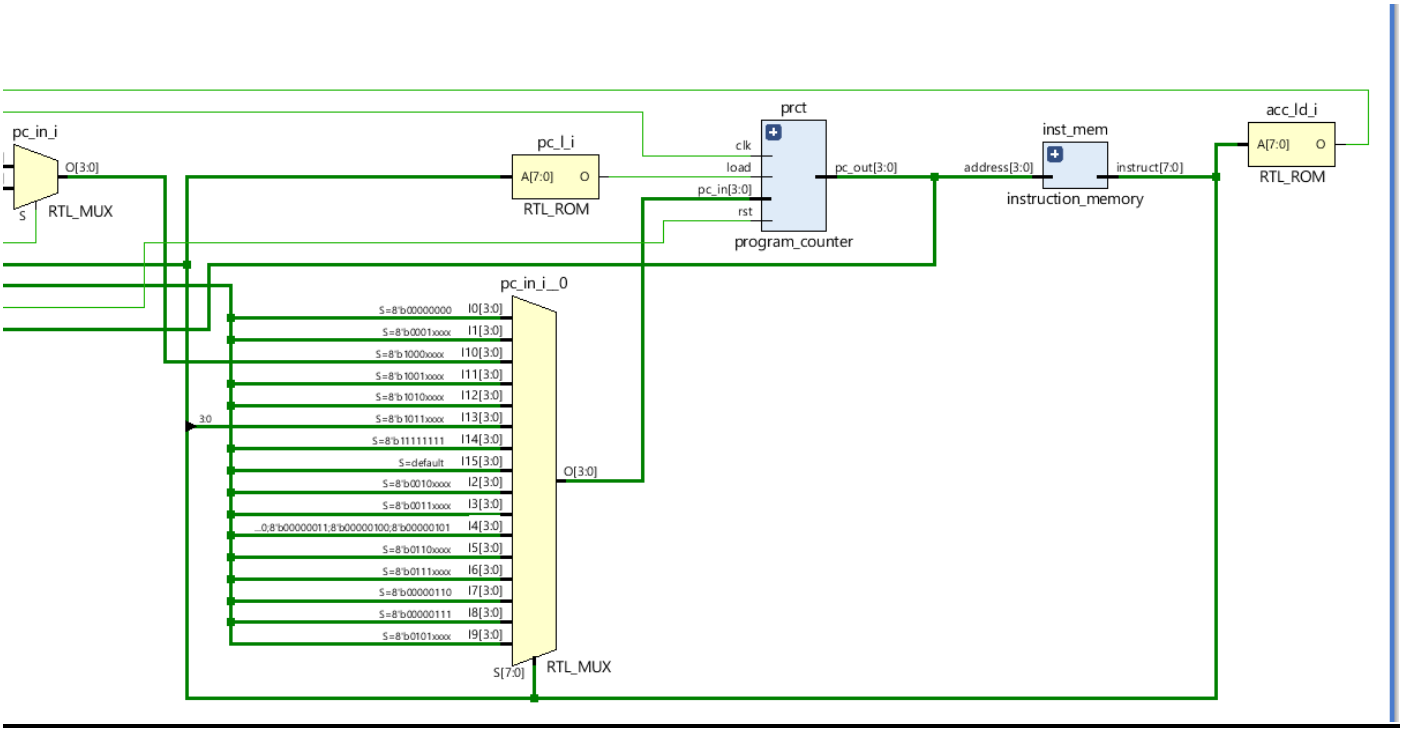
SCHEMATIC OF THE PROCESSOR



Zoomed in views




acc_reg[7:0]
CLR
C
CE
Q
D
RTL_REG_ASYNC
cb_ld_i
A[7:0]
O
RTL_ROM
cb_reg
RTL_REG_ASYNC
alu
acc[7:0]
clk
data_reg[7:0]
opcode[7:0]
acc_out[7:0]
cb
ALU
pc_next_i
I0[3:0]
V="0001"
I1[3:0]
O[3:0]
RTL_ADD
pc_in_i
S=1'b1
S=default
RTL_MUX


pc_in_i
RTL_MUX
pc_l_i
RTL_ROM
prct
clk
load
pc_in[3:0]
rst
pc_out[3:0]
program_counter
inst_mem
address[3:0]
instruct[7:0]
instruction_memory
acc_ld_i
RTL_ROM
pc_in_i_0
RTL_MUX

***