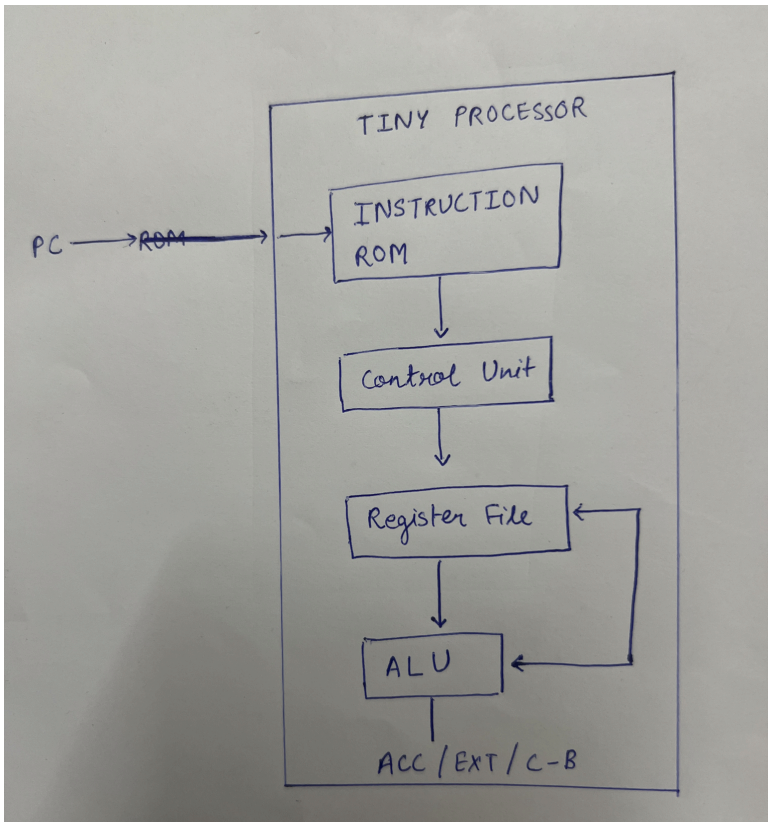
Tiny Processor

Block Diagram:



Simulation code:

```
module Instruction_ROM (
    input [3:0] PC,
    output reg [7:0] instruction
);
    reg [7:0] ROM [0:15];


    always @(*) begin
        instruction = ROM[PC];
    end
endmodule

module Control_Unit (
    input [7:0] instruction,
    output reg [3:0] opcode,
    output reg [3:0] reg_addr
);
    always @(*) begin
        opcode    = instruction[7:4];
        reg_addr  = instruction[3:0];
    end
```

```
endmodule

module Register_File (
    input clk,
    input we,
    input [3:0] read_addr,
    input [3:0] write_addr,
    input [7:0] data_in,
    output reg [7:0] data_out
);
    reg [7:0] regs [0:15];

    always @(posedge clk) begin
        if (we)
            regs[write_addr] <= data_in;
    end

    always @(*) begin
        data_out = regs[read_addr];
    end
endmodule

module ALU (
    input [3:0] opcode,
    input [7:0] ACC,
    input [7:0] RegIn,
    output reg [7:0] result,
    output reg C_B,
    output reg [7:0] EXT
);
    always @(*) begin
        C_B = 0;
        EXT = 0;
        case (opcode)
            4'b0001: {C_B, result} = ACC + RegIn; // ADD
            4'b0010: {C_B, result} = ACC - RegIn; // SUB
            4'b0011: {EXT, result} = ACC * RegIn; // MUL
            4'b0101: result = ACC & RegIn; // AND
            4'b0110: result = ACC ^ RegIn; // XRA
            4'b0111: begin // CMP
                C_B = (ACC < RegIn) ? 1 : 0;
            end
            default: result = ACC;
        endcase
    end
endmodule

module TinyProcessor (
    input clk,
```

```
    input reset
);
    reg [3:0] PC;
    reg [7:0] ACC, EXT;
    reg C_B;

    wire [7:0] instruction;
    wire [3:0] opcode, reg_addr;
    wire [7:0] reg_data;
    wire [7:0] alu_out, ext_out;
    wire alu_CB;

    reg write_enable;
    reg [3:0] write_reg;

    Instruction_ROM rom(PC, instruction);
    Control_Unit cu(instruction, opcode, reg_addr);
    Register_File rf(clk, write_enable, reg_addr, write_reg, ACC, reg_data);
    ALU alu(opcode, ACC, reg_data, alu_out, alu_CB, ext_out);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            PC <= 0;
            ACC <= 0;
            EXT <= 0;
            C_B <= 0;
        end else begin
            write_enable <= 0;
            case (opcode)
                4'b0000: begin
                    case (instruction[3:0])
                        4'b0000: ; // NOP
                        4'b0001: ACC <= ACC << 1; // LSL
                        4'b0010: ACC <= ACC >> 1; // LSR
                        4'b0011: ACC <= {ACC[0], ACC[7:1]}; // CIR
                        4'b0100: ACC <= {ACC[6:0], ACC[7]}; // CIL
                        4'b0101: ACC <= {1'b1, ACC[7:1]}; // ASR
                        4'b0110: begin
                            ACC <= ACC + 1;
                            C_B <= (ACC == 8'hFF);
                        end
                        4'b0111: begin
                            ACC <= ACC - 1;
                            C_B <= (ACC == 8'h00);
                        end
                    endcase
                end
                4'b0001, 4'b0010, 4'b0101, 4'b0110, 4'b0011, 4'b0111: begin
                    ACC <= alu_out;
```

```
                EXT <= ext_out;
                C_B <= alu_CB;
            end
            4'b1001: ACC <= reg_data; // MOV ACC, Ri
            4'b1010: begin
                write_enable <= 1;
                write_reg <= reg_addr;
            end
            4'b1000: if (C_B == 1) PC <= reg_addr; // BR addr
            4'b1011: PC <= reg_addr; // RET addr
            4'b1111: PC <= PC; // HLT
        endcase
        if (opcode != 4'b1111 && !(opcode == 4'b1000 && C_B == 1) && opcode != 4'b1011)
            PC <= PC + 1;
    end
  end
endmodule
```

Simulation testbench:

```
module TinyProcessor_tb;

    reg clk;
    reg reset;

    // Instantiate the processor
    TinyProcessor uut (
        .clk(clk),
        .reset(reset)
    );

    // Clock generation
    always #5 clk = ~clk; // 10ns clock period

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

// Preload values
        uut.rf.regs[0]  = 8'd10;
        uut.rf.regs[1]  = 8'd5;    // R1
        uut.rf.regs[2]  = 8'd20;
        uut.rf.regs[3]  = 8'd30;
        uut.rf.regs[4]  = 8'd40;
        uut.rf.regs[5]  = 8'd13;    // R5
        uut.rf.regs[6]  = 8'd9;    // R6
```

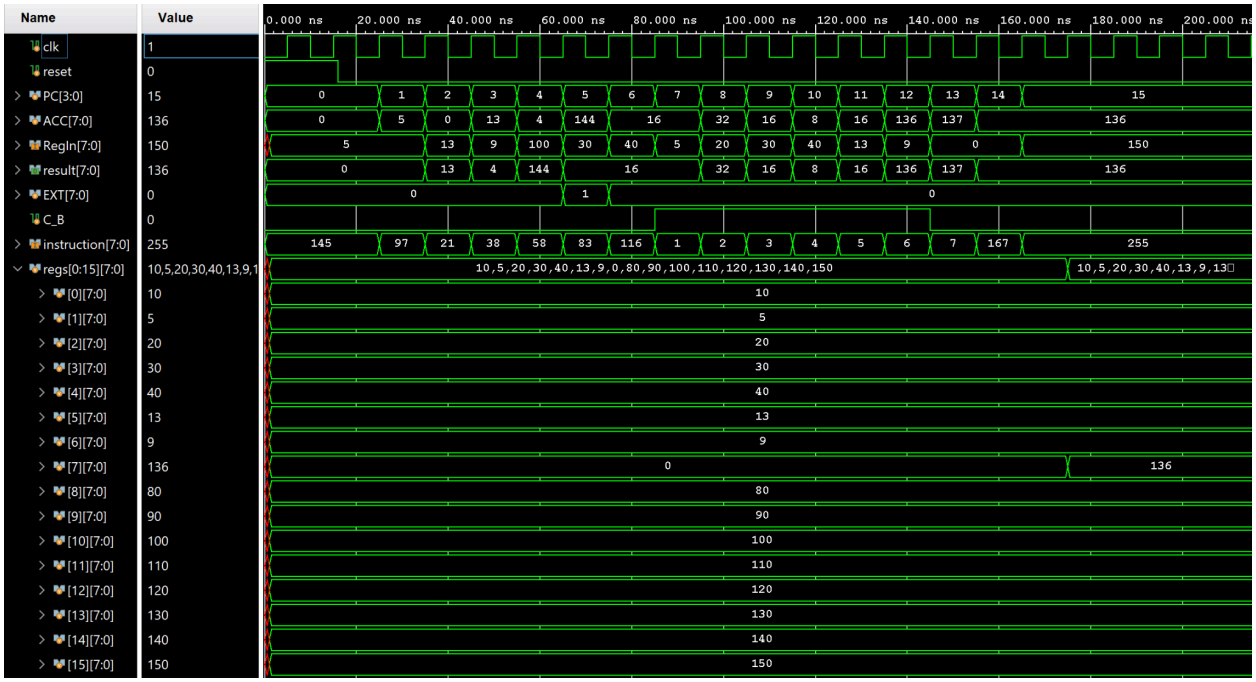
```
        uut.rf.regs[7]  = 8'd0;    // R7
        uut.rf.regs[8]  = 8'd80;
        uut.rf.regs[9]  = 8'd90;
        uut.rf.regs[10] = 8'd100;
        uut.rf.regs[11] = 8'd110;
        uut.rf.regs[12] = 8'd120;
        uut.rf.regs[13] = 8'd130;
        uut.rf.regs[14] = 8'd140;
        uut.rf.regs[15] = 8'd150;

    // Program 1
        uut.rom.ROM[0]  = 8'b1001_0001; // MOV ACC, R1
        uut.rom.ROM[1]  = 8'b0110_0001; // XRA R1 (clears ACC)
        uut.rom.ROM[2]  = 8'b0001_0101; // ADD R5
        uut.rom.ROM[3]  = 8'b0010_0110; // SUB R6
        uut.rom.ROM[4]  = 8'b0011_1010; // MUL R10
        uut.rom.ROM[5]  = 8'b0101_0011; // AND R3
        uut.rom.ROM[6]  = 8'b0111_0100; // CMP R4
        uut.rom.ROM[7]  = 8'b0000_0001; // LSL
        uut.rom.ROM[8]  = 8'b0000_0010; // LSR
        uut.rom.ROM[9]  = 8'b0000_0011; // CIR
        uut.rom.ROM[10] = 8'b0000_0100; // CIL
        uut.rom.ROM[11] = 8'b0000_0101; // ASR
        uut.rom.ROM[12] = 8'b0000_0110; // INC
        uut.rom.ROM[13] = 8'b0000_0111; // DEC
        uut.rom.ROM[14] = 8'b1010_0111; // MOV R7, ACC
        uut.rom.ROM[15] = 8'b1111_1111; // HLT

    // Program 2
    //      uut.rom.ROM[0]  = 8'b1001_0001; // MOV ACC, R1
    //      uut.rom.ROM[1]  = 8'b0110_0001; // XRA R1 (clears ACC)
    //      uut.rom.ROM[2]  = 8'b0001_0101; // ADD R5
    //      uut.rom.ROM[3]  = 8'b1011_0110; // RETURN TO 6
    //      uut.rom.ROM[4]  = 8'b0011_1010; // MUL R10
    //      uut.rom.ROM[5]  = 8'b0101_0011; // AND R3
    //      uut.rom.ROM[6]  = 8'b0111_0100; // CMP R4
    //      uut.rom.ROM[7]  = 8'b1000_1101; // BRANCH TO 13
    //      uut.rom.ROM[8]  = 8'b0000_0010; // LSR
    //      uut.rom.ROM[9]  = 8'b0000_0011; // CIR
    //      uut.rom.ROM[10] = 8'b0000_0100; // CIL
    //      uut.rom.ROM[11] = 8'b0000_0101; // ASR
    //      uut.rom.ROM[12] = 8'b0000_0110; // INC
    //      uut.rom.ROM[13] = 8'b0000_0111; // DEC
    //      uut.rom.ROM[14] = 8'b1010_0111; // MOV R7, ACC
    //      uut.rom.ROM[15] = 8'b1111_1111; // HLT


     // Program 3
     // uut.rom.ROM[0] = 8'b1001_0101; // MOV ACC, R5
```

```
 // uut.rom.ROM[1] = 8'b0001_0110; // ADD R6
 // uut.rom.ROM[2] = 8'b1010_0111; // MOV R7, ACC
 // uut.rom.ROM[3] = 8'b1111_1111; // HLT


      #15; reset = 0;

      #200; $finish();
   end

endmodule
```

Simulation result:

Testing all operations:

Program 1:
Control flow:
MOV ACC, R1 – Load the contents of register R1 into ACC.
XRA R1 – XOR ACC with R1; this clears ACC if R1 equals itself.
ADD R5 – Add the value of R5 to ACC.
SUB R6 – Subtract the value of R6 from ACC.
MUL R10 – Multiply ACC with the value in R10.
AND R3 – Perform a bitwise AND between ACC and R3.
CMP R4 – Compare ACC with R4; sets flags, no change to ACC.
LSL – Logical shift left; multiply ACC by 2.
LSR – Logical shift right; divide ACC by 2.
CIR – Circular shift right; rotates bits of ACC right.
CIL – Circular shift left; rotates bits of ACC left.
ASR – Arithmetic shift right; shifts right while preserving sign.
INC – Increment ACC by 1.
DEC – Decrement ACC by 1.
MOV R7, ACC – Copy the contents of ACC into register R7.
HLT – Halt the program execution.



Program 2:

Control flow:
MOV ACC, R1 – Load the contents of register R1 into ACC.
XRA R1 – This would have cleared ACC if not commented.
ADD R5 – Add the value in R5 to ACC.
RET 6 – Return to instruction at address 6.
MUL R10 – Skipped due to the return instruction.
AND R3 – Skipped due to the return instruction.
CMP R4 – Compare ACC with R4; flags updated, ACC unchanged.
BR 13 – Branch to instruction at address 13.
LSR – Skipped due to branch.
CIR – Skipped due to branch.
CIL – Skipped due to branch.
ASR – Skipped due to branch.
INC – Skipped due to branch.
DEC – Decrement ACC by 1.
MOV R7, ACC – Store the current value of ACC into register R7.
HLT – Halt execution.

Program 3: Sample code

Add the contents of R5 and R6, and store the result in R7.