

DIGITAL SYSTEM DESIGN CEP

DESIGN AND SIMULATION OF A 16-BIT MIPS-BASED CPU WITH SINGLE-CYCLE AND MULTI-CYCLE ARCHITECTURES

INTRODUCTION

This project presents the complete design and implementation of a custom 16-bit MIPS-based processor using Verilog Hardware Description Language (HDL). The processor has been developed from the ground up, including the instruction set architecture, datapath, control unit, register file, arithmetic logic unit, and memory system, fulfilling all requirements of a modern digital processor.

To provide both performance and design flexibility, two processor architectures were implemented:

- A single-cycle processor, where each instruction completes in one clock cycle.
- A multi-cycle processor, where instructions are executed over multiple clock cycles under the control of a finite state machine (FSM).

The purpose of implementing both architectures is to analyze and compare their execution behavior, control complexity, hardware utilization, and timing efficiency. The single-cycle design offers simplicity and fast instruction execution, while the multi-cycle design improves hardware efficiency by sharing functional units across different instruction stages.

Both processors were fully verified using ModelSim simulation, where instruction execution, register operations, ALU functionality, and memory access were observed through waveform analysis. This project demonstrates how real processors are designed at the hardware level and provides a strong foundation for advanced topics such as pipelining, performance optimization, and embedded processor design.

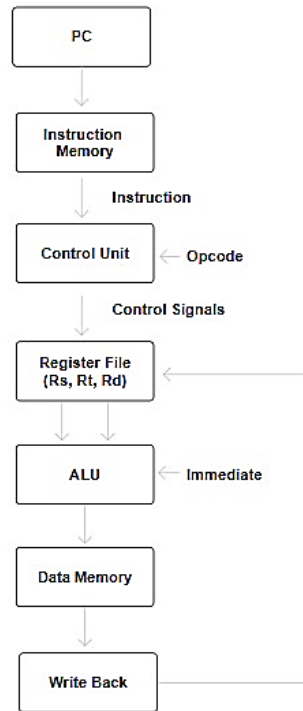
INSTRUCTION SET ARCHITECTURE (ISA)

Instructions	Type	Operation
ADD	R	$Rd \leftarrow Rs + Rt$
SUB	R	$Rd \leftarrow Rs - Rt$
AND	R	$Rd \leftarrow Rs \& Rt$
OR	R	$Rd \leftarrow Rs Rt$
LW	I	$Rt \leftarrow Mem[Rs + Imm]$
SW	I	$Mem[Rs + Imm] \leftarrow Rt$
BEQ	I	If $Rs == Rt$, branch
J	J	Jump to address

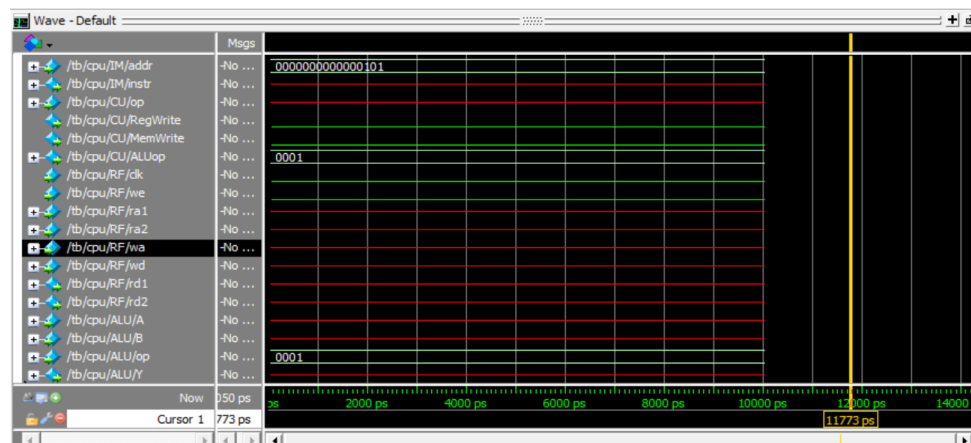
SINGLE-CYCLE CPU DESIGN

In the single-cycle CPU, all stages (fetch, decode, execute, memory, write-back) are completed in one clock cycle.

Block Diagram:



Simulation Results:

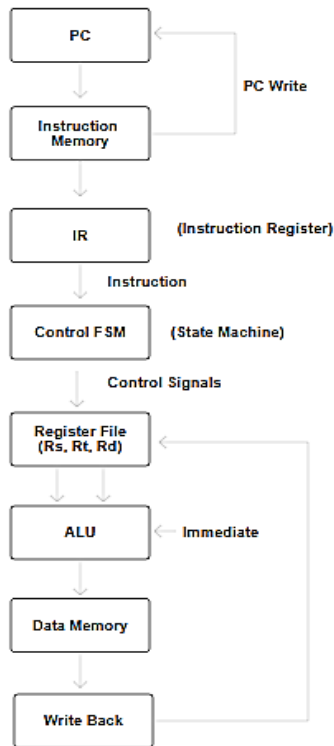


The waveform shows PC update, instruction fetch, ALU operation, and register write occurring in a single clock.

MULTI-CYCLE CPU DESIGN

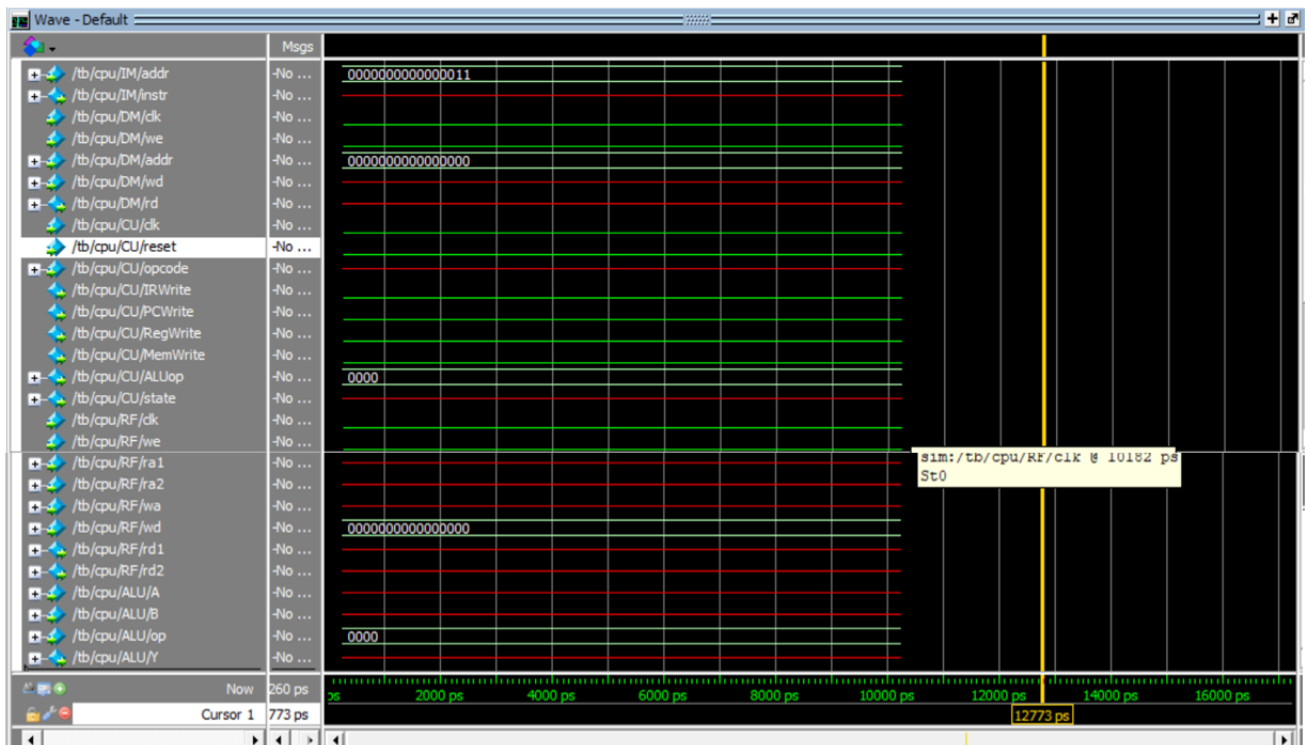
The multi-cycle CPU uses an FSM-based control unit. Each instruction is executed in five stages: Fetch, Decode, Execute, Memory, and Write-Back.

Block Diagram:

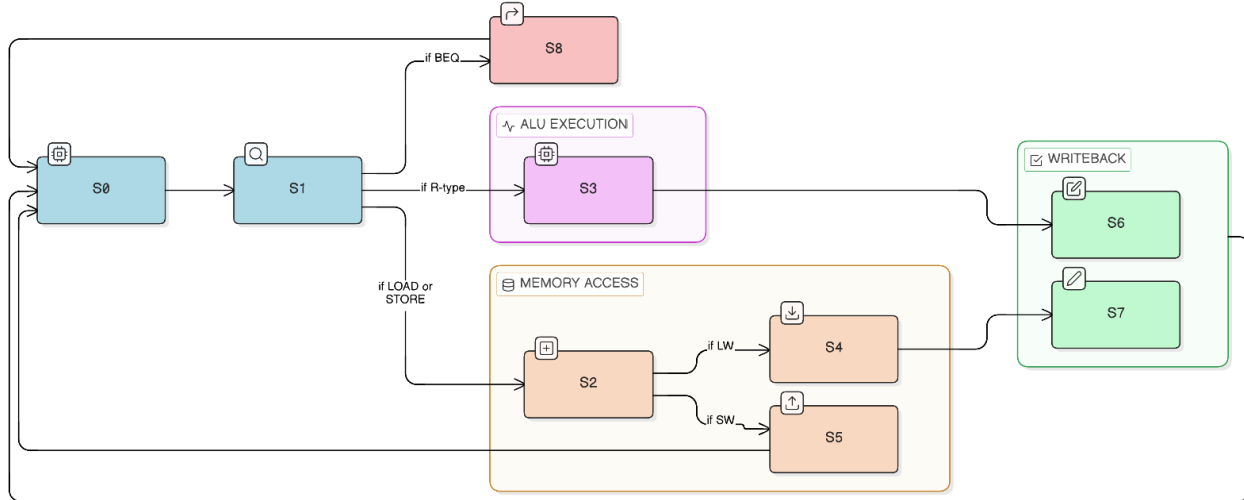


Simulation Results:

The waveform clearly shows state transitions and control signals such as IRWrite, PCWrite, RegWrite, and MemWrite, proving correct multi-cycle execution.



Multi-Cycle MIPS FSM State Diagram:



This state diagram represents the control logic of a Multi-Cycle MIPS Processor. It operates as a Moore Finite State Machine, moving from instruction fetch and decode (S0–S1) into specific paths for memory access, arithmetic execution, or branching. By dividing instructions into these discrete steps, the design allows the CPU to reuse hardware components like the ALU and ensures each operation only uses the clock cycles it truly requires.

COMPARISON TABLE

Feature	Single-Cycle	Multi-Cycle
Clock cycles per instruction	1	3–5
Control	Simple	FSM-based
Hardware usage	High	Lower
Performance	Fast per instruction	Efficient overall
Complexity	Low	High (better design)
Clock cycles per instruction	1	3–5
Control	Simple	FSM-based
Hardware usage	High	Lower

CONCLUSION

The CEP successfully implemented both single-cycle and multi-cycle CPUs. The multi-cycle processor is more hardware-efficient and closer to real MIPS architecture. Waveform verification proves correct instruction execution in both designs.

HDL SOURCE CODES

1. ALU:

```
module alu(input [15:0] A,B, input [3:0] op, output reg [15:0] Y);
always @(*) begin
    case(op)
        4'b0000: Y=A+B;
        4'b0001: Y=A-B;
        4'b0010: Y=A&B;
        4'b0011: Y=A|B;
        4'b0100: Y=A^B;
        4'b0101: Y=(A<B)?16'd1:16'd0;
        default: Y=0;
    endcase
end
endmodule
```

2. Register File:

```
module regfile(input clk,we,
input [2:0] ra1,ra2,wa,
input [15:0] wd,
output [15:0] rd1,rd2);

reg [15:0] R[7:0];
assign rd1 = R[ra1];
assign rd2 = R[ra2];

always @(posedge clk)
    if(we && wa!=0) R[wa] <= wd;
endmodule
```

3. Instruction Memory:

```
module imem(input [15:0] addr, output [15:0] instr);
reg [15:0] mem[255:0];
initial begin
    mem[0]=16'b0000_001_010_011_000; // ADD R1,R2,R3
    mem[1]=16'b0001_100_001_010_000; // SUB R4,R1,R2
end
assign instr = mem[addr];
endmodule
```

4. Data Memory:

```
module dmem(input clk,we, input [15:0] addr,wd, output [15:0] rd);
reg [15:0] mem[255:0];
assign rd = mem[addr];
always @(posedge clk)
    if(we) mem[addr]<=wd;
endmodule
```

5. Single-Cycle Control Unit:

```
module control(input [3:0] op,
output reg RegWrite,MemWrite,
output reg [3:0] ALUOp);

always @(*) begin
    RegWrite=0; MemWrite=0;
    case(op)
        4'b0000: begin RegWrite=1; ALUOp=4'b0000; end
        4'b0001: begin RegWrite=1; ALUOp=4'b0001; end
        4'b0010: begin RegWrite=1; ALUOp=4'b0010; end
        4'b0111: MemWrite=1;
    endcase
end
endmodule
```

6. 16-bit Single-Cycle Processor Top Module:

```
module S16(input clk);
```

```

reg [15:0] PC=0;
wire [15:0] instr;
imem IM(PC,instr);

wire [3:0] op = instr[15:12];
wire [2:0] rd=instr[11:9], rs=instr[8:6], rt=instr[5:3];

wire RW,MW; wire [3:0] aluop;
control CU(op,RW,MW,aluop);

wire [15:0] A,B,Y;
regfile RF(clk,RW,rs,rt,rd,Y,A,B);
alu ALU(A,B,aluop,Y);

always @(posedge clk) PC<=PC+1;
endmodule

```

7. Multi-Cycle Control Unit:

```

module mc_control(
    input clk, reset,
    input [3:0] opcode,
    output reg IRWrite, PCWrite,
    output reg RegWrite, MemWrite,
    output reg [3:0] ALUOp,
    output reg [2:0] state
);

always @(posedge clk or posedge reset) begin
    if(reset) state <= 0;
    else begin
        case(state)
            0: state <= 1;      // Fetch
            1: state <= 2;      // Decode
            2: state <= (opcode==4'b0110 || opcode==4'b0111) ? 3 : 4;
            3: state <= 4;      // Memory
            4: state <= 0;      // Writeback → Fetch
        endcase
    end
end

always @(*) begin
    IRWrite=0; PCWrite=0; RegWrite=0; MemWrite=0; ALUOp=0;
    case(state)
        0: begin IRWrite=1; PCWrite=1; ALUOp=4'b0000; end
        2: begin ALUOp = opcode; end
        3: if(opcode==4'b0111) MemWrite=1;
        4: if(opcode!=4'b0111) RegWrite=1;
    endcase
end
endmodule

```

8. 16-bit Multi-Cycle Processor Top Module:

```

module S16_MC(input clk, reset);

    reg [15:0] PC;
    reg [15:0] IR, Areg, Breg, ALUOut, MDR;

    wire [15:0] instr, memdata;
    wire [3:0] opcode;

    assign opcode = IR[15:12];

    // ----- Memories -----
    imem IM(PC, instr);
    dmem DM(clk, MemWrite, ALUOut, Breg, memdata);

    // ----- Control -----
    wire IRWrite, PCWrite, RegWrite;
    wire [3:0] ALUOp;

```

```

wire [2:0] state;

mc_control CU(clk, reset, opcode, IRWrite, PCWrite, RegWrite, MemWrite, ALUop, state);

// ----- Register File -----
wire [15:0] rd1, rd2;
regfile RF(
    clk,
    RegWrite,
    IR[8:6], // Rs
    IR[5:3], // Rt
    IR[11:9], // Rd
    MDR,
    rd1,
    rd2
);

// ----- ALU -----
wire [15:0] ALUres;
alu ALU(Areg, Breg, ALUop, ALUres);

// ----- Datapath Registers -----
always @(posedge clk or posedge reset) begin
    if(reset) begin
        PC <= 0;
        IR <= 0;
        Areg <= 0;
        Breg <= 0;
        ALUOut <= 0;
        MDR <= 0;
    end
    else begin
        if(IRWrite) IR <= instr;
        if(state == 3'd1) begin
            Areg <= rd1;
            Breg <= rd2;
        end
        if(state == 3'd2)
            ALUOut <= ALUres;
        if(state == 3'd3)
            MDR <= memdata;
        if(PCWrite)
            PC <= PC + 1;
    end
end

endmodule

```

TEST BENCHES

1. tb for single-cycle:

```

module tb;
    reg clk;
    S16 cpu(clk);

    initial begin
        clk=0;
        repeat(10) #5 clk=~clk;
    end
endmodule

```

2. tb for multi-cycle:

```

module tb;
    reg clk, reset;

    S16_MC cpu(clk, reset);

    initial begin
        reset = 1;
    end
endmodule

```

```
    clk = 0;
    #10 reset = 0;    // release reset
    repeat(50) #5 clk = ~clk;
end

initial begin
    $dumpfile("cpu_mc.vcd");
    $dumpvars(0,tb);
end

endmodule
```