**DIGITAL SYSTEM DESIGN**

**FINAL PROJECT REPORT**

INSTRUCTOR: DR. VO MINH THANH

**I. Project Topic 2**: "Design and Implementation of a Multi-Cycle MIPS Microprocessor in an Altera FPGA Kit using Verilog HDL"

**II, Description**:

This project aims to design and implement a multi-cycle MIPS (Microprocessor without Interlocked Pipeline Stages) microprocessor using the Verilog Hardware Description Language (HDL) and an Altera FPGA (Field-Programmable Gate Array) development kit. The objective is to develop a fully functional microprocessor capable of executing MIPS instructions, adhering to the multi-cycle architecture.

**III. Objectives**:

Familiarize yourself with the architecture and instruction set of a multi-cycle MIPS microprocessor.

* Design the various components of the microprocessor, including the instruction fetch, instruction decode, execution, memory, and writeback units.
* Develop a detailed block diagram and data path for the multi-cycle MIPS microprocessor.
* Implement the microprocessor's components in Verilog HDL, ensuring proper functionality and adherence to the design specifications.
* Integrate and test the individual components to form a complete multi-cycle MIPS microprocessor.
* Verify the correctness of the microprocessor's implementation through simulation and functional testing.
* Synthesize the Verilog design for the target Altera FPGA device and generate the programming file.
* Configure the Altera FPGA development kit with the microprocessor design and perform hardware testing.
* Develop a set of test programs to validate the microprocessor's functionality and performance.
* Evaluate the performance of the implemented microprocessor and compare it with theoretical expectations and industry standards.

**IV. Deliverables:**

* Detailed project report documenting the design process, implementation details, and test results.
* Verilog HDL source code for the microprocessor's components.
* Block diagram and data path diagrams illustrating the microprocessor's architecture.
* Simulation results demonstrating the correctness of the design.
* FPGA programming file for configuring the target Altera FPGA device.
* Test programs and their corresponding results to validate the microprocessor's functionality.
* Final evaluation report highlighting the performance and limitations of the implemented microprocessor.

### **V. DESCRIPTION**

### **Multi Cycle Microprocessor datapath to be implemented is in figure 2.1.**

### A diagram of a computer Description automatically generated

### Figure 2.1: Multi-cyclye Cycle Microprocessor DataPath

Instruction Operation codes:

**A close-up of a list

Description automatically generated**

Instruction Formats:

Timeline

Description automatically generated

Table

Description automatically generated

Register names and orders:



Assume the Assembly code code start from address PC=0x00000000, one instruction is store in one memory location.

**Testing Assembly Program 1:**

Instruction Meaning

Begin: addi $s2, $zero, 0x55 // load immediate value 0x55 to register $s2

addi $s3, $zero, 0x22 // load immediate value 0x22 to register $s3

addi $s5, $zero, 0x77 // load immediate value 0x77 to register $s5

add $s4, $s2, $s3 // $s4 = $s2 + $s3 => R20=0x77   
sub $s1, $s2, $s3 // $s1 = $s2 – $s3 => R17=0x22   
sw $s1, 0x02($s2) // Memory[$s2+0x02] = $s1

lw $s6, 0x02($s2) // $s6 = Memory[$s2+0x02]

bne $s5, $s4, End // Next instr. is at End if $s5 != $s4

addi $s8, $zero, 0x10 // load immediate value 10 to register $s8  
beq $s5,$s4, End // Next instr. is at End if $s7 == $s4

addi $s8, $zero, 0x20 // load immediate value 20 to register $s8

End: j End // jump End

**Testing Assembly Program 2:**

Instruction Meaning

Begin: addi $s2, $zero, 0x55 // load immediate value 0x55 to register $s2

addi $s3, $zero, 0x22 // load immediate value 0x22 to register $s3

addi $s5, $zero, 0x77 // load immediate value 0x77 to register $s5

add $s4, $s2, $s3 // $s4 = $s2 + $s3 => R20=0x77   
sub $s1, $s2, $s3 // $s1 = $s2 – $s3 => R17=0x22   
sw $s1, 0x02($s2) // Memory[$s2+0x02] = $s1

lw $s6, 0x02($s2) // $s6 = Memory[$s2+0x02]

beq $s5,$s4, End // Next instr. is at End if $s7 == $s4

addi $s8, $zero, 0x10 // load immediate value 10 to register $s8

bne $s5, $s4, End // Next instr. is at End if $s5 != $s4

addi $s8, $zero, 0x20 // load immediate value 20 to register $s8

End: j End // jump End

**VI. Code Verilog HDL design**

**Program Counter**

module Program\_Counter (clk, reset, PC\_write ,PC\_in, PC\_out);

    input clk, reset,PC\_write;

    input [31:0] PC\_in;

    output reg [31:0] PC\_out;

    // Program counter with reset and write enable

    always @ (posedge clk or posedge reset)

    begin

        if(reset)

            PC\_out <= 32'b0;

        else if (PC\_write)

            PC\_out <= PC\_in;

    end

endmodule

**ALU**

module alu(

    input [2:0] alufn,

    input [31:0] ra,

    input [31:0] rb\_or\_imm,

    output reg [31:0] aluout,

    output reg zero);

    parameter   ALU\_OP\_ADD      = 3'b000,

                ALU\_OP\_SUB      = 3'b001,

                ALU\_OP\_AND      = 3'b010,

                ALU\_OP\_OR       = 3'b011,

                ALU\_OP\_XOR      = 3'b100,

                ALU\_OP\_LW       = 3'b101,

                ALU\_OP\_SW       = 3'b110,

                ALU\_OP\_BEQ      = 3'b111;

     // ALU operation based on alufn input

    always @(\*)

        begin

          case(alufn)

            ALU\_OP\_ADD      : aluout = ra + rb\_or\_imm;

            ALU\_OP\_SUB      : aluout = ra - rb\_or\_imm;

            ALU\_OP\_AND      : aluout = ra & rb\_or\_imm;

            ALU\_OP\_OR       : aluout = ra | rb\_or\_imm;

            ALU\_OP\_XOR      : aluout = ra ^ rb\_or\_imm;

            ALU\_OP\_LW       : aluout = ra + rb\_or\_imm;

            ALU\_OP\_SW       : aluout = ra + rb\_or\_imm;

            ALU\_OP\_BEQ      : begin

                                zero = (ra==rb\_or\_imm)?1'b1:1'b0;

                                aluout = ra - rb\_or\_imm;

                              end

          endcase

        end

endmodule

**Shift left 2**

module shift\_left\_2 (sign\_in, sign\_out);

    input [31:0] sign\_in;

    output [31:0] sign\_out;

    // Shift left by 2

    assign sign\_out[31:2]=sign\_in[29:0];

    assign sign\_out[1:0]=2'b00;

endmodule

**Concate**

module concate(PC\_in,IR\_in,PC\_out);

    input [3:0] PC\_in;

    input [27:0] IR\_in;

    output[31:0] PC\_out;

    assign PC\_out={PC\_in, IR\_in};

endmodule

**MUX 32-bit**

module Mux4\_32\_bit (in0, in1,in2, in3, mux\_out, select);

    input [31:0] in0, in1,in2,in3;

    output [31:0] mux\_out;

    input [1:0]select;

    assign mux\_out = select[1]? (select[0]?in3: in2):(select[0]?in1:in0);

endmodule

**Hex SSD**

module hex\_ssd (BIN, SSD);

  input [3:0] BIN;

  output reg [0:6] SSD;

  always@(\*) begin

    case(BIN)

      0:SSD=7'b0000001;

      1:SSD=7'b1001111;

      2:SSD=7'b0010010;

      3:SSD=7'b0000110;

      4:SSD=7'b1001100;

      5:SSD=7'b0100100;

      6:SSD=7'b0100000;

      7:SSD=7'b0001111;

      8:SSD=7'b0000000;

      9:SSD=7'b0001100;

      10:SSD=7'b0001000;

      11:SSD=7'b1100000;

      12:SSD=7'b0110001;

      13:SSD=7'b1000010;

      14:SSD=7'b0110000;

      15:SSD=7'b0111000;

    endcase

  end

endmodule

**Holding register**module holding\_reg(output\_data, input\_data, write, clk, reset);

  // inputs

  input [31:0] input\_data;

  input write, clk, reset;

  // outputs

  output reg [31:0] output\_data;

  // Register content and output assignment

    // update regisiter contents

  always @(posedge clk or posedge reset)

  begin

    if (reset)

    begin

      output\_data <= 32'b0;

    end

    else if (write)

    begin

      output\_data <= input\_data;

    end

  end

endmodule

**Register file**module Register\_File (clk,read\_addr\_1, read\_addr\_2, write\_addr, read\_data\_1, read\_data\_2, write\_data, RegWrite);

    input [4:0] read\_addr\_1, read\_addr\_2, write\_addr;

    input [31:0] write\_data;

    input  clk,RegWrite;

    reg checkRegWrite;

    output reg [31:0] read\_data\_1, read\_data\_2;

    reg [31:0] Regfile [31:0];

    integer k;

    initial

        begin

            for (k=0; k<32; k=k+1)

                begin

                    Regfile[k] = 32'd0;

                end

            Regfile[8]=32'd1;//$t0

            Regfile[9]=32'd2;//$t1

            Regfile[10]=32'd3; //$t2

            Regfile[11]=32'd4; //$t3

            Regfile[17]=32'h99;//$s1

            Regfile[18]=32'h60;//$s2

            Regfile[19]=32'h30;//$s3

        end

    //assign read\_data\_1 = Regfile[read\_addr\_1];

        always @(read\_data\_1 or Regfile[read\_addr\_1])

            begin

              if (read\_addr\_1 == 0) read\_data\_1 = 0;

              else

              begin

              read\_data\_1 = Regfile[read\_addr\_1];

              //$display("read\_addr\_1=%d,read\_data\_1=%h",read\_addr\_1,read\_data\_1);

              end

            end

    //assign read\_data\_2 = Regfile[read\_addr\_2];

        always @(read\_data\_2 or Regfile[read\_addr\_2])

            begin

              if (read\_addr\_2 == 0) read\_data\_2 = 0;

              else

              begin

              read\_data\_2 = Regfile[read\_addr\_2];

              //$display("read\_addr\_2=%d,read\_data\_2=%h",read\_addr\_2,read\_data\_2);

              end

            end

    always @(posedge clk)

            begin

              if (RegWrite == 1'b1)

                 begin

                     Regfile[write\_addr] = write\_data;

                     $display("Register File write\_addr=%d write\_data=%d",write\_addr,write\_data);

                 end

            end

endmodule

**MUX 5-bit**module Mux\_5\_bit (in0, in1, mux\_out, select);

    parameter N = 5;

    input [N-1:0] in0, in1;

    output [N-1:0] mux\_out;

    input select;

    // 5-bit multiplexer

    assign mux\_out = select? in1: in0 ;

endmodule

**Sign extension**

module Sign\_Extension (sign\_in, sign\_out);

    input [15:0] sign\_in;

    output [31:0] sign\_out;

    // Sign-extend a 16-bit input to a 32-bit output

    assign sign\_out[15:0]=sign\_in[15:0];

    assign sign\_out[31:16]=sign\_in[15]?16'b1111111111111111:16'b0;

endmodule

**Data Memory**module Data\_Memory (clk,addr, write\_data, read\_data, MemRead, MemWrite);

    input [31:0] addr;

    input [31:0] write\_data;

    output [31:0] read\_data;

    input MemRead, MemWrite,clk;

    reg [31:0] DMemory [63:0];

    integer k;

    initial begin

        for (k=0; k<64; k=k+1)

            begin

                DMemory[k] = 32'b0;

            end

        //sw  $s1, 0x02($s2)        //  Memory[$s2+0x02] = $s1

        DMemory[0] = 32'b10101110010100010000000000000010;

        //add $s4,  $s2, $s3        //  $s4 = $s2 + $s3  => R20=0x90

        DMemory[4] = 32'b00000010010100111010000000100000;

        //add $s5 $t0 $t1       //r[21]=t0+t1=1+2=3

        DMemory[8] = 32'b00000001000010011010100000100000;

        //sub $s1, $s2, $s3     //  $s1 = $s2   $s3  => R17=0x22=d30

        DMemory[12] = 32'b00000010010100111000100000100010;

        //sw  $s1, 0x02($s2)        //  Memory[$s2+0x02] = $s1 = d30  //memory[62]=d30

        DMemory[16] = 32'b10101110010100010000000000000010;

        //lw $s1, 0x02($s2)         //  $s1 = Memory[$s2+0x02]

        //R[17]=memory[62]=d30

        DMemory[20] = 32'b10001110010100010000000000000010;

        //beq $t2,$t3, End      //beq $t2,$t3, 0x03

        DMemory[24] = 32'b00010001010010110000000000000011;

        //addi $s7, $zero, 0x16  //R[23]=0x16

        DMemory[28] = 32'b00100000000101110000000000010000;

        //addi $s2, $zero, 0x55 //  load immediate value 0x55 to register $s2

        DMemory[32] = 32'b00100000000100100000000001010101;

        // //  load immediate value 0x22 to register $s3

        DMemory[36] = 32'b00100000000100110000000000100010;

        // //  load immediate value 0x77 to register $s5

        DMemory[40] = 32'b00100000000101010000000001110111;

        //j 0x00

        DMemory[44] = 32'b00001000000000000000000000000000;

        end

    assign read\_data = (MemRead) ? DMemory[addr] : 32'bx;

    always @(posedge clk)

        begin

            if (MemWrite)

            begin

               DMemory[addr] = write\_data;

               $display("Data memory write\_addr=%d write\_data=%d",addr,write\_data);

            end

        end

endmodule

**Data path for Multi\_Cycle processor**

module  Datapath\_Multi\_cycle\_Processor(clk, reset,Iord,MemWrite,MemtoReg,RegWrite,RegDst,ALUSrcA,PCWr,IRwrite,MemRead,ALUSrcB,PCSource,Operation\_ALU,ALU\_in\_B,ALU\_in\_A,ALU\_out,B\_data,mux\_2\_out,Jump\_addr,PC\_in,PC\_out,Mem\_Read\_data,instruction,MDR\_out,ALU\_out\_hold,jump\_28\_bit,mux\_1\_out);

    input clk, reset;

    input Iord,MemWrite,MemtoReg,RegWrite,RegDst,ALUSrcA,PCWr,IRwrite,MemRead;

    input [1:0] ALUSrcB,PCSource;

    input [2:0] Operation\_ALU;

    output [31:0] ALU\_in\_B,ALU\_in\_A,ALU\_out,B\_data,mux\_2\_out,Jump\_addr,mux\_1\_out;

    output [31:0] PC\_in,PC\_out,Mem\_Read\_data,instruction,MDR\_out,ALU\_out\_hold;

    output [27:0] jump\_28\_bit;

    // Internal wires for various connections

    wire [31:0] W\_RD2, W\_RD1,Extend\_out,Branch\_addr,A\_data;

    wire [4:0] mux\_3\_out;

    wire zero,PCWrcond,and\_out;

    // Instantiate various components of the datapath

    Program\_Counter     comp1(clk, reset, PCWr, PC\_in, PC\_out);

    Mux\_32\_bit          comp2(PC\_out, ALU\_out\_hold, mux\_1\_out, Iord);

    Data\_Memory         comp3(clk,mux\_1\_out, B\_data, Mem\_Read\_data, MemRead, MemWrite);

    holding\_reg         comp4(instruction, Mem\_Read\_data, IRwrite, clk, reset);

    holding\_reg         comp5(MDR\_out, Mem\_Read\_data, 1'b1, clk, reset);

    Mux\_32\_bit          comp6(MDR\_out,ALU\_out\_hold, mux\_2\_out, MemtoReg);

    Register\_File       comp7(clk,instruction[25:21], instruction[20:16], mux\_3\_out, W\_RD1, W\_RD2, mux\_2\_out, RegWrite);

    Mux\_5\_bit           comp8(instruction[20:16], instruction[15:11], mux\_3\_out, RegDst);

    Sign\_Extension      comp9(instruction[15:0], Extend\_out);

    shift\_left\_2        comp10(Extend\_out, Branch\_addr);

    holding\_reg         comp11(A\_data, W\_RD1, 1'b1, clk, reset);

    holding\_reg         comp12(B\_data, W\_RD2, 1'b1, clk, reset);

    Mux\_32\_bit          comp13(PC\_out, A\_data, ALU\_in\_A, ALUSrcA);

    Mux4\_32\_bit         comp14(B\_data, 32'd4,Extend\_out,Branch\_addr , ALU\_in\_B, ALUSrcB);

    alu                 comp15(Operation\_ALU, ALU\_in\_A, ALU\_in\_B, ALU\_out,zero);

    holding\_reg         comp16(ALU\_out\_hold, ALU\_out , 1'b1, clk, reset);

    shift\_left\_2\_28bit  comp17(instruction[25:0], jump\_28\_bit);

    concate             comp18(PC\_out[31:28],jump\_28\_bit,Jump\_addr);

    Mux4\_32\_bit         comp19(ALU\_out, ALU\_out\_hold,Jump\_addr, 32'b0, PC\_in, PCSource);

endmodule

**VII. Test bench for each module**

**Program Counter**

module **Tbench**;

    reg clk, reset, PC\_write;

    reg [31:0] PC\_in;

    wire [31:0] PC\_out;

    Program\_Counter dut(clk, reset, PC\_write, PC\_in, PC\_out);

    always #5 clk = ~clk;

    initial begin

        $monitor(**$time**, "clk = %b, reset = %b, PC\_write = %b, PC\_in = %d, PC\_out = %d",

                 clk, reset, PC\_write, PC\_in, PC\_out);

        clk = 0; reset = 1; PC\_write = 0; PC\_in = 32'b0;

        #10 reset = 0;

        #10 PC\_write = 1; PC\_in = 32'b000111;

        #10 PC\_in = 32'b010101;

        #10 PC\_in = 32'b011111;

        #10 PC\_in = 32'b100111;

        #10 **$finish**;

    end

endmodule

**A screenshot of a computer screen

Description automatically generated**

**ALU**

module **alu\_tb**;

    reg [2:0] alufn;

    reg [31:0] ra, rb\_or\_imm;

    wire [31:0] aluout;

    wire zero;

    alu dut(

        .alufn(alufn),

        .ra(ra),

        .rb\_or\_imm(rb\_or\_imm),

        .aluout(aluout),

        .zero(zero)

    );

    initial begin

        $monitor(**$time**, " alufn = %b, ra = %d, rb\_or\_imm = %d, aluout = %d, zero = %b",

                 alufn, ra, rb\_or\_imm, aluout, zero);

*// Test ADD*

        alufn = 3'b000; ra = 32'd10; rb\_or\_imm = 32'd5;

        #10;

*// Test SUB*

        alufn = 3'b001; ra = 32'd10; rb\_or\_imm = 32'd5;

        #10;

*// Test AND*

        alufn = 3'b010; ra = 32'hFF00; rb\_or\_imm = 32'h0FF0;

        #10;

*// Test OR*

        alufn = 3'b011; ra = 32'hFF00; rb\_or\_imm = 32'h0FF0;

        #10;

*// Test XOR*

        alufn = 3'b100; ra = 32'hFF00; rb\_or\_imm = 32'h0FF0;

        #10;

*// Test LW (same as ADD)*

        alufn = 3'b101; ra = 32'd100; rb\_or\_imm = 32'd4;

        #10;

*// Test SW (same as ADD)*

        alufn = 3'b110; ra = 32'd100; rb\_or\_imm = 32'd4;

        #10;

*// Test BEQ (equal case)*

        alufn = 3'b111; ra = 32'd20; rb\_or\_imm = 32'd20;

        #10;

*// Test BEQ (not equal case)*

        alufn = 3'b111; ra = 32'd20; rb\_or\_imm = 32'd21;

        #10;

**$finish**;

    end

endmodule

**A close-up of a number

Description automatically generated**

**Shift Left 2**

module **shift\_left\_2\_tb**;

    reg [31:0] sign\_in;

    wire [31:0] sign\_out;

    shift\_left\_2 dut(

        .sign\_in(sign\_in),

        .sign\_out(sign\_out)

    );

    initial begin

        $monitor(**$time**, " sign\_in = %b, sign\_out = %b", sign\_in, sign\_out);

*// Test case 1: All zeros*

        sign\_in = 32'b0;

        #10;

*// Test case 2: All ones*

        sign\_in = 32'hFFFFFFFF;

        #10;

*// Test case 3: Alternating bits*

        sign\_in = 32'b10101010101010101010101010101010;

        #10;

*// Test case 4: Random value*

        sign\_in = 32'h12345678;

        #10;

*// Test case 5: Value with least significant bits set*

        sign\_in = 32'h00000003;

        #10;

**$finish**;

    end

endmodule

**A computer screen shot of a number

Description automatically generated**

**Concate**

module **concate\_tb**;

    reg [3:0] PC\_in;

    reg [27:0] IR\_in;

    wire [31:0] PC\_out;

    concate dut(

        .PC\_in(PC\_in),

        .IR\_in(IR\_in),

        .PC\_out(PC\_out)

    );

    initial begin

        $monitor(**$time**, " PC\_in = %b, IR\_in = %b, PC\_out = %b", PC\_in, IR\_in, PC\_out);

*// Test case 1: All zeros*

        PC\_in = 4'b0000; IR\_in = 28'b0;

        #10;

*// Test case 2: All ones*

        PC\_in = 4'b1111; IR\_in = 28'hFFFFFFF;

        #10;

*// Test case 3: Alternating bits*

        PC\_in = 4'b1010; IR\_in = 28'b1010101010101010101010101010;

        #10;

*// Test case 4: Random values*

        PC\_in = 4'b1100; IR\_in = 28'h1234567;

        #10;

*// Test case 5: Edge case*

        PC\_in = 4'b0001; IR\_in = 28'hFFFFFFF;

        #10;

**$finish**;

    end

endmodule

**A number of binary code

Description automatically generated with medium confidence**

**Mux4 32-bit**

module **Mux4\_32\_bit\_tb**;

    reg [31:0] in0, in1, in2, in3;

    reg [1:0] select;

    wire [31:0] mux\_out;

    Mux4\_32\_bit dut(

        .in0(in0),

        .in1(in1),

        .in2(in2),

        .in3(in3),

        .mux\_out(mux\_out),

        .select(select)

    );

    initial begin

        $monitor(**$time**, " select = %b, in0 = %h, in1 = %h, in2 = %h, in3 = %h, mux\_out = %h",

                 select, in0, in1, in2, in3, mux\_out);

*// Initialize inputs*

        in0 = 32'hAAAAAAAA;

        in1 = 32'hBBBBBBBB;

        in2 = 32'hCCCCCCCC;

        in3 = 32'hDDDDDDDD;

*// Test all select combinations*

        select = 2'b00; #10; *// Should select in0*

        select = 2'b01; #10; *// Should select in1*

        select = 2'b10; #10; *// Should select in2*

        select = 2'b11; #10; *// Should select in3*

*// Test with different input values*

        in0 = 32'h11111111;

        in1 = 32'h22222222;

        in2 = 32'h33333333;

        in3 = 32'h44444444;

        select = 2'b00; #10;

        select = 2'b01; #10;

        select = 2'b10; #10;

        select = 2'b11; #10;

**$finish**;

    end

endmodule

**A screenshot of a computer code

Description automatically generated**

**Sign extension**

module **Sign\_Extension\_tb**;

    reg [15:0] sign\_in;

    wire [31:0] sign\_out;

    Sign\_Extension dut(

        .sign\_in(sign\_in),

        .sign\_out(sign\_out)

    );

    initial begin

        $monitor(**$time**, " sign\_in = %b (%d), sign\_out = %b (%d)",

                 sign\_in, $signed(sign\_in), sign\_out, $signed(sign\_out));

*// Test positive numbers*

      sign\_in = 16'h0001; #10; *// Small positive number*

        sign\_in = 16'h7FFF; #10; *// Largest positive number*

*// Test negative numbers*

        sign\_in = 16'hFFFF; #10; *// -1*

        sign\_in = 16'h8000; #10; *// Smallest negative number*

*// Test zero*

        sign\_in = 16'h0000; #10;

*// Test some random values*

        sign\_in = 16'h1234; #10;

        sign\_in = 16'hABCD; #10;

**$finish**;

    end

endmodule

**A screen shot of a computer code

Description automatically generated**

**Mux 5-bit**

module **Mux\_5\_bit\_tb**;

    parameter N = 5;

    reg [N-1:0] in0, in1;

    reg select;

    wire [N-1:0] mux\_out;

    Mux\_5\_bit dut(

        .in0(in0),

        .in1(in1),

        .mux\_out(mux\_out),

        .select(select)

    );

    initial begin

        $monitor(**$time**, " select = %b, in0 = %b, in1 = %b, mux\_out = %b",

                 select, in0, in1, mux\_out);

*// Test case 1: select = 0*

        in0 = 5'b10101; in1 = 5'b01010; select = 0;

        #10;

*// Test case 2: select = 1*

        in0 = 5'b10101; in1 = 5'b01010; select = 1;

        #10;

*// Test case 3: Different values, select = 0*

        in0 = 5'b11111; in1 = 5'b00000; select = 0;

        #10;

*// Test case 4: Different values, select = 1*

        in0 = 5'b11111; in1 = 5'b00000; select = 1;

        #10;

*// Test case 5: Same values, select = 0*

        in0 = 5'b11001; in1 = 5'b11001; select = 0;

        #10;

*// Test case 6: Same values, select = 1*

        in0 = 5'b11001; in1 = 5'b11001; select = 1;

        #10;

**$finish**;

    end

endmodule

**A table of numbers and symbols

Description automatically generated with medium confidence**

**Register file:**

module **testbench**();

    reg [4:0] read\_addr\_1, read\_addr\_2, write\_addr;

    reg [31:0] write\_data;

    reg clk, RegWrite;

    wire [31:0] read\_data\_1, read\_data\_2;

    Register\_File DUT (clk, read\_addr\_1, read\_addr\_2, write\_addr, read\_data\_1, read\_data\_2, write\_data, RegWrite);

    always #5 clk = ~clk;

    initial begin

        clk = 0;

        $monitor(**$time**, " clk=%b, read\_addr\_1=%d, read\_addr\_2=%d, read\_data\_1=%d, read\_data\_2=%d, RegWrite=%d, write\_addr=%d, write\_data=%d ",

                clk, read\_addr\_1, read\_addr\_2, read\_data\_1, read\_data\_2, RegWrite, write\_addr, write\_data);

        #10 RegWrite = 1;

        #25 write\_addr = 32'b1011; write\_data = 32'b1011;

        #5 RegWrite = 0;

        #5 read\_addr\_1 = 32'b1011;

        #5 read\_addr\_2 = 32'b1011;

        #10 **$finish**;

    end

endmodule

Table

Description automatically generated

**Holding register**

module **holding\_reg**(output\_data, input\_data, write, clk, reset);

*// data size*

    parameter word\_size = 32;

*// inputs*

    input [word\_size-1:0] input\_data;

    input write, clk, reset;

*// outputs*

    output [word\_size-1:0] output\_data;

*// Register content and output assignment*

    reg [word\_size-1:0] content;

    assign output\_data = content;

*// update register contents*

    always @(posedge clk) begin

        if (reset) begin

            content <= 0;

        end

        else if (write) begin

            content <= input\_data;

        end

    end

endmodule

**A screenshot of a computer program

Description automatically generated**

**Data memory**

module **Data\_Memory** (clk, addr, write\_data, read\_data, MemRead, MemWrite);

    input [7:0] addr;

    input [31:0] write\_data;

    output [31:0] read\_data;

    input MemRead, MemWrite, clk;

    reg [31:0] DMemory [255:0];

    integer k;

    initial begin

        for (k=0; k<256; k=k+1)

        begin

            DMemory[k] = 32'b0;

        end

        DMemory[0] = 32'b10101100110100010000000000000010;  *// sw  $s1, 0x02($s2)      // Memory[$s2+0x02] = $s1*

        DMemory[4] = 32'b00000010010100110100000000100000;  *// add $s4, $s2, $s3      // $s4 = $s2 + $s3  => R20=0x90*

        DMemory[8] = 32'b00000010101101010000000000100000;  *// add $s5 $t0 $t1*

        DMemory[12] = 32'b00000010001100110100100000100010; *// sub $s1, $s2, $s3      // $s1 = $s2 - $s3  => R17=0x22*

        DMemory[16] = 32'b10101100110100010000000000000010; *// sw  $s1, 0x02($s2)      // Memory[$s2+0x02] = $s1*

        DMemory[20] = 32'b10001100110100010000000000000010; *// lw  $s1, 0x02($s2)      // $s1 = Memory[$s2+0x02]*

        DMemory[24] = 32'b00010010001100110000000000000011; *// beq $s2, $s3, End       // beq $t2,$t3, 0x03*

        DMemory[28] = 32'b00100000000101110000000000010000; *// addi $s7, $zero, 0x10   // j 0x00*

        DMemory[32] = 32'b00100000000100100000000001010101; *// addi $s2, $zero, 0x55   // load immediate value 0x55 to register $s2*

        DMemory[36] = 32'b00100000000100110000000000100010; *// addi $s3, $zero, 0x22   // load immediate value 0x22 to register $s3*

        DMemory[40] = 32'b00100000000101010000000001110111; *// addi $s5, $zero, 0x77   // load immediate value 0x77 to register $s5*

    end

    assign read\_data = (MemRead) ? DMemory[addr] : 32'bx;

    always @(posedge clk)

    begin

        if (MemWrite)

        begin

            DMemory[addr] = write\_data;

**$display**("Data memory write addr=%b write data=%h", addr, write\_data);

        end

    end

endmodule

**A close-up of a computer code

Description automatically generated**