به نام خدا

محمدفرحان بهرامی

401105729

پروژه امتیازی DSD

سوال هفتم میانترم

ماژول های مورد نیاز:

ماژول ALU بر اساس مقدار opcode ورودی، دو عملیات جمع و ضرب را انجام می‌دهد. ورودی‌های ماژول شامل opcode سیگنال کنترلی برای تعیین نوع عملیات، A1و A2 داده‌های ورودی 512 بیتی هستند و خروجی آن alu_result نتیجه عملیات 1024 بیتی است. در داخل ماژول، چندین متغیر داخلی از جمله iشمارنده حلقه، A1_partو A2_part بخش‌های 32 بیتی از ورودی‌ها، alu_part نتیجه موقتی عملیات 64 بیتی و overflow نشانگر سرریز تعریف شده‌اند.

در بلوک always حساس به تغییرات هر ورودی، بر اساس مقدار opcode، دو عملیات متفاوت انجام می‌شود. اگر opcode برابر با 0 باشد، عملیات جمع اجرا می‌شود. برای هر 16 بخش 32 بیتی از ورودی‌ها، جمع این بخش‌ها انجام شده و نتیجه در alu_part ذخیره می‌شود. سپس، بررسی سرریز انجام می‌شود. اگر سرریز رخ دهد، بیت‌های بالایی alu_part تنظیم می‌شوند تا سرریز را نشان دهند. در نهایت، نتیجه هر بخش 64 بیتی در alu_result ذخیره می‌شود.

اگر opcodeبرابر با 1 باشد، عملیات ضرب اجرا می‌شود. برای هر 16 بخش 32 بیتی از ورودی‌ها، ضرب این بخش‌ها انجام شده و نتیجه در alu_part ذخیره می‌شود. سپس، نتیجه هر بخش 64 بیتی در alu_result ذخیره می‌شود.

کد مربوطه در صفحه بعد

ALU.v

```v
module ALU (
    input wire opcode,
    input wire [511:0] A1,
    input wire [511:0] A2,
    output reg [1023:0] alu_result
);
integer i;
reg signed [31:0] A1_part, A2_part;
reg signed [63:0] alu_part;
reg overflow;
always @(*) begin
    case (opcode)
        1'b0: begin
            for (i = 0; i < 16; i = i + 1) begin
                A1_part = A1[i * 32 +: 32];
                A2_part = A2[i * 32 +: 32];
                alu_part = A1_part + A2_part;
                overflow = (~A1_part[31] & ~A2_part[31] & alu_part[31]) |
                           (A1_part[31] & A2_part[31] & ~alu_part[31]);
                if (overflow) begin
                    if (alu_part[32]) begin
                        alu_part[63:33] <= 31'b1;
                    end
                    else if (alu_part[31]) begin
                        alu_part[63:32] <= 32'b1;
                    end
                end
                alu_result[i * 64 +: 64] = alu_part;
            end
        end
        1'b1: begin
            for (i = 0; i < 16; i = i + 1) begin
                A1_part = A1[i * 32 +: 32];
                A2_part = A2[i * 32 +: 32];

                alu_part = A1_part * A2_part;
                alu_result[i * 64 +: 64] = alu_part;
            end
        end
    endcase
end
endmodule

```

ماژول ممووری:

ماژول memory یک حافظه 512 کلمه‌ای با عرض داده 32 بیت را پیاده‌سازی می‌کند. ورودی‌های ماژول شامل clk سیگنال ساعت، rst سیگنال ریست، write_en فعال‌سازی نوشتن، input_data داده‌های ورودی 512 بیتی و address آدرس 9 بیتی هستند و خروجی ماژول output_data داده‌های خروجی 512 بیتی است.

در بلوک always حساس به لبه نزولی سیگنال ساعت یا ریست، اگر سیگنال rstغیر فعال باشد، یعنی در حالت ریست، یک حلقه برای مقداردهی صفر به تمام مکان‌های حافظه اجرا می‌شود. در صورت فعال بودن سیگنال write_en، آدرس پایانی end_addr به اندازه 16 کلمه بعد از آدرس ورودی تنظیم می‌شود و حلقه‌ای برای نوشتن داده‌ها از input_data به مکان‌های مشخص شده در حافظه اجرا می‌شود. هر بخش 32 بیتی از input_data به مکان مربوطه در حافظه نوشته می‌شود.

در بلوک always حساس به لبه بالارونده سیگنال ساعت، ابتدا output_data به صفر تنظیم می‌شود. سپس آدرس پایانی end_addr به اندازه 16 کلمه بعد از آدرس ورودی تنظیم می‌شود و حلقه‌ای برای خواندن داده‌ها از حافظه و انتقال آن‌ها به output_data اجرا می‌شود. هر بخش 32 بیتی از حافظه به مکان مربوطه در output_data منتقل می‌شود.

```
module memory (
    input wire clk,
    input wire rst,
    input wire write_en,
    input wire [511:0] input_data,
    input wire [8:0] address,
    output reg [511:0] output_data
);
    reg signed [31:0] memory [0:511];
    reg [9:0] end_addr;
    integer i;
    always @(negedge clk or negedge rst) begin
        if (!rst) begin
            for (i = 0; i < 512; i = i + 1) begin
                memory[i] <= 32'b0;
            end
        end else if (write_en) begin
            end_addr = address + 16;
            for (i = address; i < end_addr; i = i + 1) begin
                if (i < 512) begin
                    memory[i] <= input_data[(i - address) * 32 +: 32];
                end
            end
        end
    end
    always @(posedge clk) begin
        output_data <= 512'b0;
        end_addr = address + 16;
        for (i = address; i < end_addr; i = i + 1) begin
            if (i < 512) begin
                output_data[(i - address) * 32 +: 32] <= memory[i];
            end
        end
    end
endmodule
```

ماژول رجیستر فایل:

ماژول register_file یک فایل رجیستر با چهار رجیستر 512 بیتی را پیاده‌سازی می‌کند که قابلیت نوشتن و بارگذاری داده‌ها را دارد. ورودی‌های ماژول شامل clk ساعت، rst ریست، write_en فعال‌سازی نوشتن، load_en فعال‌سازی بارگذاری داده، dst_reg رجیستر مقصد، load_data داده‌های بارگذاری و alu_result نتیجه واحد ALU هستند. خروجی ماژول یک آرایه A از چهار رجیستر 512 بیتی است.

در بلوک always حساس به لبه نزولی ساعت و ریست، ابتدا مقادیر رجیسترها در صورت فعال بودن سیگنال ریست تنظیم می‌شوند. در این حالت، رجیسترهای A[0] و A[1] با مقادیر از پیش تعیین‌شده مقداردهی می‌شوند و رجیسترهای A[2] و A[3] به صفر تنظیم می‌شوند. اگر سیگنال write_en فعال باشد، عملیات نوشتن انجام می‌شود. در صورتی که load_en نیز فعال باشد، داده‌های ورودی load_data در رجیستر مقصد dst_reg نوشته می‌شوند. در غیر این صورت، داده‌های alu_result در رجیسترهای A[2] و A[3] توزیع می‌شوند. برای این منظور، داده‌های 32 بیتی از alu_result به صورت تکه‌های 64 بیتی به A[2] و A[3] منتقل می‌شوند.

```
module register_file (
    input wire clk,
    input wire rst,
    input wire write_en,
    input wire load_en,
    input wire [1:0] dst_reg,
    input reg [511:0] load_data,
    input reg [1023:0] alu_result,
    output reg [511:0] A [0:3]
);
integer i;
always @(negedge clk or negedge rst) begin
    if (!rst) begin
        // Initialize register values
        A[0] <= 512'hFFFFFFFF_FFFFFFFF_FFFFFFFF_7FFFFFFF_7FFFFFFF_7FFFFFFF_80000000_80000000_80000000_00000000_00000000_00000000_00000001_00000001_00000001_00000001;
        A[1] <= 512'hFFFFFFFF_7FFFFFFF_80000000_FFFFFFFF_7FFFFFFF_80000000_FFFFFFFF_7FFFFFFF_80000000_FFFFFFFF_7FFFFFFF_80000000_FFFFFFFF_7FFFFFFF_80000000_00000001;
        A[2] <= 512'b0;
        A[3] <= 512'b0;
    end else if (write_en) begin
        if (load_en) begin
            A[dst_reg] <= load_data;
        end else begin
            for (i = 0; i < 16; i = i + 1) begin
                A[2][i * 32 +: 32] <= alu_result[i * 64 +: 32];
                A[3][i * 32 +: 32] <= alu_result[i * 64 + 32 +: 32];
            end
        end
    end
end
endmodule

```

ماژول پردازنده:

ورودی‌های ماژول شامل clk سیگنال ساعت، rst سیگنال بازنشانی و instruction دستورالعمل 13 بیتی هستند. خروجی ماژول یک آرایه Aاز چهار رجیستر 512 بیتی است.

دستورالعمل 13 بیتی شامل سه بخش است opcode :بیت‌های 12 و 11، reg_no بیت‌های 10 و 9 و address بیت‌های 8 تا 0
opcodeمشخص‌کننده نوع عملیات است:

- 00 برای جمع
- 01 برای ضرب
- 10 برای بارگذاری
- 11 برای ذخیره

```
3   module processor (
4       input wire clk,
5       input wire rst,
6       input wire [12:0] instruction,
7       output wire [511:0] A [0:3])
8   ;
9   wire [1023:0] alu_result;
10  wire [511:0] memory_result;
11  ALU inst1 (
12      .opcode(instruction[11]),
13      .A1(A[0]),
14      .A2(A[1]),
15      .alu_result(alu_result)
16      );
17  register_file inst2 (
18      .clk(clk),
19      .rst(rst),
20      .write_en(~(instruction[12] & instruction[11])),
21      .load_en(instruction[12] & ~instruction[11]),
22      .dst_reg(instruction[10:9]),
23      .load_data(memory_result),
24      .alu_result(alu_result),
25      .A(A)
26      );
27  memory inst3 (
28      .clk(clk),
29      .rst(rst),
30      .write_en(instruction[12] & instruction[11]),
31      .input_data(A[instruction[10:9]]),
32      .address(instruction[8:0]),
33      .output_data(memory_result)
34      );
35  endmodule
```

ماژول تست:

در تست اول، عملیات جمع دو عدد تست می‌شود. ابتدا مقادیر 1000 و 2000 به ترتیب به رجیسترهای 0 و 1 بارگذاری می‌شوند. این کار با استفاده از دستورالعمل‌های load انجام می‌شود. سپس، با استفاده از دستورالعمل add، مقدارهای موجود در رجیسترهای 0 و 1 جمع شده و نتیجه در رجیسترهای 2 و 3 ذخیره می‌شود.

در تست دوم، عملیات ضرب دو عدد تست می‌شود. ابتدا مقادیر 3 و 4 به ترتیب به رجیسترهای 0 و 1 بارگذاری می‌شوند. سپس، با استفاده از دستورالعمل multiply، مقدارهای موجود در رجیسترهای 0 و 1 ضرب شده و نتیجه در رجیسترهای 2 و 3 ذخیره می‌شود.

در تست سوم، عملیات ذخیره مقدار به حافظه تست می‌شود. مقدار موجود در رجیستر 2 (نتیجه جمع) به آدرس حافظه 5 ذخیره می‌شود.

در تست چهارم، عملیات بارگذاری مقدار از حافظه تست می‌شود. مقدار موجود در آدرس حافظه 5 به رجیستر 0 بارگذاری می‌شود.

```
module processor_tb;
  reg clk;
  reg rst;
  reg [12:0] instruction;
  wire [511:0] A [0:3];
  processor uut (
    .clk(clk),
    .rst(rst),
    .instruction(instruction),
    .A(A)
  );
  initial begin
    // Initialize signals
    clk = 0;
    rst = 1;
    instruction = 13'b0;
    // Monitor signals
    $monitor("Time=%0t, instruction=%b, A[0]=%h, A[1]=%h, A[2]=%h, A[3]=%h",
             $time, instruction, A[0], A[1], A[2], A[3]);
    // Reset the processor
    #5 rst = 0;
    #5 rst = 1;
    // Wait for reset
    #10;
```

```
    // Test 1: Add two numbers
    // Load values into registers
    // Load value 1000 into register 0
    instruction = 13'b100_00_000000000; // load R0
    #10;
    instruction = 13'b000_00_000000001; //   instruction to wait
    #10;
    // Load value 2000 into register 1
    instruction = 13'b100_01_000000010; // load R1
    #10;
    instruction = 13'b000_00_000000001; //   instruction to wait
    #10;
    // Perform addition: R0 + R1 -> R2, R3
    instruction = 13'b000_00_000000000; // add
    #10;
    // Test 2: Multiply two numbers
    // Load value 3 into register 0
    instruction = 13'b100_00_000000011; // load R0
    #10;
    instruction = 13'b000_00_000000001; //   instruction to wait
    #10;
    // Load value 4 into register 1
    instruction = 13'b100_01_000000100; // load R1
    #10;
    instruction = 13'b000_00_000000001; //   instruction to wait
    #10;
    // Perform multiplication: R0 * R1 -> R2, R3
    instruction = 13'b001_00_000000000; // multiply
    #10;
    // Test 3: Store value to memory
    // Store the result of addition (R2) to memory address 5
    instruction = 13'b110_10_000000101; // store R2
    #10;
    // Test 4: Load value from memory
    // Load the value from memory address 5 to register 0
    instruction = 13'b101_00_000000101; // load R0
    #10;
    instruction = 13'b000_00_000000001; //   instruction to wait
    #10;
    // End of simulation
    $finish;
  end
  // Generate clock signal
  always #5 clk = ~clk;

endmodule
```

که در نهایت بعد از شبیه سازی آن و خروجی گرفتن در مدل سیم به نتایج زیر دست پیدا میکنیم: