به نام خدا

جبرانی پایان‌ترم

درس: طراحی سیستم های دیجیتال

استاد: مهندس فصحتی

نویسنده: شمیم رحیمی

شماره دانشجویی: ۴۰۱۱۰۵۹۵۶

سوال ۷)

برای ساخت یک پردازنده‌ی آرایه‌ای ۵۱۲ بیتی ابتدا ۳ بخش اصلی آن را طراحی می‌کنیم:

۱- یک رجیسترفایل با قابلیت ذخیره سازی ۴ آرایه‌ی ۵۱۲ بیتی با نام های A1 تا A4 ماژول طراحی شده به این شکل است:

```
module Register_File(
    input wire clk,
    input wire reset,
    input wire [1:0] write_sel,
    input wire [511:0] write_data,
    input wire [1:0] write_sel2,
    input wire [511:0] write_data2,
    input wire write_en,
    input wire write_en2,
    input wire read_en,
    input wire [1:0] read_sel,
    output wire [511:0] read_data,
    output signed [511:0] A1,
    output signed [511:0] A2,
    output signed [511:0] A3,
    output signed [511:0] A4
);

    reg signed [511:0] registers [0:3];
    integer index;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (index = 0; index < 4; index = index + 1) begin
                registers[index] <= 512'b0;
            end
        end else begin
            if (write_en) begin
                case (write_sel)
                    2'b00: registers[0] <= write_data;
                    2'b01: registers[1] <= write_data;
                    2'b10: registers[2] <= write_data;
                    2'b11: registers[3] <= write_data;
                endcase
            end
            if (write_en2) begin
                case (write_sel2)
                    2'b00: registers[0] <= write_data2;
                    2'b01: registers[1] <= write_data2;
                    2'b10: registers[2] <= write_data2;
                    2'b11: registers[3] <= write_data2;
                endcase
            end
            if (read_en) begin
                case (read_sel)
                2'b00: read_data <= registers[0];
                2'b01: read_data <= registers[1];
                2'b10: read_data <= registers[2];
                2'b11: read_data <= registers[3];
            endcase
            end
        end
    end

    assign A1 = registers[0];
    assign A2 = registers[1];
    assign A3 = registers[2];
    assign A4 = registers[3];

endmodule
```

توضیح ماژول رجیستر فایل:
ابتدا ورودی و خروجی های لازم را تعریف میکنیم. طبعا دو ورودی clock و reset دارد. می‌خواهیم ماژول همزمان بتواند روی دو رجیستر عملیات نوشتن را انجام دهد پس از ورودی های write enable، write address(select) و write data دو تا داریم. برای عملیات خواندن هم read_en، read data و read select(register number) داریم. همچنین چون می خواهیم در ماژول نهایی تست‌بنچ، محتوای رجیسترها را نمایش دهیم، خود رجیسترها را نیز خروجی می‌گیریم.

آدرس هایی که شماره‌ی رجیستر مقصد را مشخص می‌کنند (write sel, write sel2, read sel)، دو بیتی هستند زیرا باید بتوانیم ۴ عدد را با آنها نمایش دهیم چون ۴ رجیستر داریم.

همچنین دیتا های ورودی برای نوشتن و خروجی برای خواندن هم ۵۱۲ بیتی هستند زیرا این پردازنده ۵۱۲ بیتی است.

در ابتدا ۴ رجیستر ۵۱۲ بیتی تعریف می‌کنیم.

سپس در بلاک always که به لبه‌ی بالارونده‌ی clk و reset حساس است، بررسی می‌کنیم که reset یک شده‌است یا خیر. اگر یک بود همه‌ی رجیستر ها را ریست و صفر می‌کنیم و در غیر این صورت در صورت یک بودن هر کدام از write_enable ها، عملیات write مربوطه انجام می‌شود و با توجه به اینکه آدرس رجیستر کدام است، داخل همان رجیستر نوشته می‌شود.
اگر read_en هم فعال باشد، محتوای رجیستری که توسط read_sel مشخص شده‌است را داخل read_data میریزد که خروجی رجیسترفایل است.
همچنین محتوای رجیسترها را نیز در خروجی های مربوط به خودشان میریزیم.

۲- یک واحد ALU که قابلیت ضرب و جمع دارد. دقت کنید که این ALU در اصل ورودی های ۵۱۲ بیتی دارد اما چون طول کلمه‌ها ۳۲ بیت است، در اصل روی ۱۶ تا کلمه‌ی ۳۲ بیتی عملیات انجام می‌دهد.

```
module ALU(
    input signed [511:0] A1,
    input signed [511:0] A2,
    input wire [1:0] operation, // 00: Addition, 01: Multiplication
    output reg signed [1023:0] output_data
);

    integer k;
    reg [8:0] read_addr;
    reg [31:0] data_chunk;

    always @* begin
        // Default assignment
        output_data = 1024'b0;

        // Perform operation based on operation select
        case (operation)
            2'b00: begin // Addition
                for (k = 0; k<16; k=k+1) begin
                    read_addr = i * 32 +: 32;
                    data_chunk = A1[read_addr] + A2[read_addr];
                    output_data[i * 64 +: 64] = data_chunk;
                end
            end
            2'b01: begin // Multiplication
                for (k = 0; k<16; k=k+1) begin
                    read_addr = i * 32 +: 32;
                    data_chunk = A1[read_addr] * A2[read_addr];
                    output_data[i * 64 +: 64] = data_chunk;
                end
            end
            default: begin // Default case (no operation)
                output_data = 1024'b0;
            end
        endcase
    end

endmodule
```

این ماژول دو ورودی رجیستر ۵۱۲ بیتی دارد. (دو وکتور شامل ۱۶ عدد صحیح علامت دار ۳۲ بیتی) یک ورودی ۲ بیتی که عملیات آن را مشخص می‌کند و یک وکتور ۱۰۲۴ بیتی شامل ۱۶ عدد ۶۴ بیتی که همان نتایج عملیات ها روی ۱۶ عدد ۳۲ بیتی هستند.
این یک مدار ترکیبی هستند با تغییر هرکدام از ورودی ها کار می‌کند. داخل بلاک always اگر operation == 00 یعنی جمع باید انجام شود و اگر operation == 01 بود یعنی ضرب و در غیر این صورت مقادیر خروجی را صفر می‌کند.
این واحد، عملیات حسابی را میان مولفه های ۳۲ بیتی متناظر بردارها انجام می‌دهد و حاصل ۶۴ بیتی را در مولفه‌ی متناظر بردار خروجی قرار می‌دهد.

۳- یک حافظه با عمق ۵۱۲ و عرض ۳۲ بیت:

```
module Memory(
    input wire clock,
    input wire rst,
    input wire wr_enable,
    input wire rd_enable,
    input [8:0] addr, // 9-bit address for 512 memory locations
    input signed [511:0] wr_data,
    output reg signed [511:0] rd_data
);

    reg [31:0] memory_array [0:511];
    integer k;
    reg [8:0] write_addr;
    reg [31:0] data_chunk;

    // Asynchronous reset and memory initialization
    always @(posedge clock or posedge rst) begin
        if (rst) begin
            $readmemh("hex.txt", memory_array);
        end
    end

    // Write operation
    always @(posedge clock) begin
        if (!rst && wr_enable) begin
            for (k = 0; k < 16; k = k + 1) begin
                write_addr = (addr + k) % 512;
                data_chunk = wr_data[(k * 32) +: 32];
                memory_array[write_addr] <= data_chunk;
            end
        end
    end

    // Read operation
    always @(posedge clock) begin
        if (!rst && rd_enable) begin
            rd_data = 'b0; // Clear read_data before assignment
            for (k = 0; k < 16; k = k + 1) begin
                read_addr = (addr + k) % 512;
                temp_data = memory_array[read_addr];
                rd_data[(k * 32) +: 32] <= temp_data;
            end
        end
    end
endmodule
```

توضیح ماژول حافظه:

این ماژول ورودی های کلاک و ریست دارد. write enable و read enable دارد که مشخص می‌کنند عملیات خواندن و نوشتن انجام شوند یا خیر. همچنین چون عمق آن ۵۱۲ بیتی است پس ۹ بیت برای آدرس دهی نیاز دارد. ۵۱۲ بیت دیتای ورودی برای نوشتن و ۵۱۲ بیت دیتای خروجی برای خواندن.

ابتدا ۳۲ خانه‌ی ۵۱۲ بیتی که همان حافظه‌ی ما است را تعریف می‌کنیم.

دقت کنید که برای initialize کردن حافظه از دستور

$readmemh("hex.txt", mem);

استفاده کنیم و محتوای فایل txt می‌تواند به صورت رندوم با یک کد پایتون تولید شده باشد اما در اینجا چون می‌خواهیم مقادیر مرزی و بسیار بزرگ و بسیار کوچک را تست کنیم، محتوای فایل hex را به صورت دستی با مقادیر مرزی و حساس پر می‌کنیم.

اگر write enable فعال باشد، می‌نویسم و اگر read enable فعال باشد، می‌خوانیم و همه‌ی این عملیات ها را در بلاک های جدا انجام می‌دهیم.
دقت کنید که عملیات خواندن و نوشتن به صورت چرخشی انجام می‌شود. ما آدرس پایه را به حافظه می‌دهیم و خروجی حافظه یک وکتور ۵۱۲ بیتی شامل ۱۶ عدد صحیح ۳۲ بیتی است که در ۱۶ خانه‌ی متوالی هستند که آدرس ابتدای آن همان آدرس پایه است.

در حلقه های مربوط به خواندن و نوشتن، ابتدا آدرس خانه‌ی مدنظر با ایندکس محاسبه می‌کنیم و سپس محتوای آن ایندکس را می‌خوانیم.

حال که این سه ماژول مورد نیاز را ساختیم، این ها را کنار هم می‌گذاریم تا پردازنده‌ی اصلی ساخته شود. ۴ دستور داریم: load, store, add, multiply
پس دستورات ما ۲ بیت آپکد می‌خواهند. دو بیت برای مشخص کردن شماره‌ی رجیستر مورد نیاز در دستورات load و store و همچنین ۹ بیت برای آدرس دهی. دقت کنید که دستورات add و multiply فقط با همان دو بیت آپکد مشخص می‌شوند و ورودی و خروجی های آنها از قبل در صورت سوال مشخص شده‌اند.

12 11	10 9	8 0
opcode	register number	memory address

پس ماژول پردازنده به این شکل است: (در ۲ عکس)

```
module Vector_Processor(
    input wire clk,
    input wire reset,
    input wire [12:0] instruction, // 00: load, 01: store, 10: addition, 11: multiplication
    output signed [511:0] A1,
    output signed [511:0] A2,
    output signed [511:0] A3,
    output signed [511:0] A4
);

    // Register File
    reg signed [511:0] RF_in;
    reg signed [511:0] RF_in2;
    wire signed [511:0] RF_A1;
    wire signed [511:0] RF_A2;
    wire signed [511:0] RF_A3;
    wire signed [511:0] RF_A4;
    reg [1:0] write_sel;
    reg [1:0] write_sel2;
    reg [1:0] read_sel;
    reg write_en, write_en2, read_en;
    Register_File register_file (
        .clk(clk),
        .reset(reset),
        .write_sel(write_sel),
        .write_data(RF_in),
        .write_sel2(write_sel2),
        .write_data2(RF_in2),
        .write_en(write_en),
        .write_en2(write_en2),
        .read_en(read_en),
        .read_sel(read_sel),
        .read_data(RF_out),
        .A1(RF_A1),
        .A2(RF_A2),
        .A3(RF_A3),
        .A4(RF_A4)
    );

    // ALU
    reg signed [511:0] ALU_in_1;
    reg signed [511:0] ALU_in_2;
    reg [1:0] ALUOp;
    wire signed [1023:0] ALU_out;
    ALU alu (
        .A1(ALU_in_1),
        .A2(ALU_in_2),
        .operation(ALUOp),
        .output_data(ALU_out)
    );

    // Data Memory
    reg signed [511:0] DM_in;
    wire signed [511:0] DM_out;
    reg [8:0] DM_address;
    reg DM_write_enable, DM_read_enable;
    Memory data_memory (
        .clk(clk),
        .reset(reset),
        .write_en(DM_write_enable),
        .read_en(DM_read_enable),
        .address(DM_address),
        .write_data(DM_in),
        .read_data(DM_out)
    );

    // Output assignments
    assign A1 = RF_A1;
    assign A2 = RF_A2;
    assign A3 = RF_A3;
    assign A4 = RF_A4;
```

این ماژول ورودی و خروجی های کلاک و ریست و دستور را دارد. همچنین محتوای رجیستر ها را برای نمایش در تست بنچ، خروجی می‌گیریم.

ورودی ها و خروجی های سه ماژول ساخته شده را نیز تعریف می‌کنیم و آنها (رجیسترفایل، ALU و حافظه) را می‌سازیم.

همچنین خروجی های رجیسترفایل را به خروجی های ماژول اساین می‌کنیم.

```
    // State machine
    always @(posedge clk) begin
        if (reset) begin
            write_en <= 0;
            write_en2 <= 0;
            read_en <= 0;
            DM_write_enable <= 0;
            DM_read_enable <= 0;
            write_sel <= 2'b00;
            write_sel2 <= 2'b00;
            read_sel <= 2'b00;
            ALUOp <= 2'b00;
        end else begin
            case (instruction[12:11])
                2'b00: begin // load
                    DM_write_enable <= 0;
                    DM_read_enable <= 1;
                    DM_address <= instruction[8:0];
                    write_en <= 1;
                    write_sel <= instruction[10:9];
                    RF_in <= DM_out;
                end
                2'b01: begin // store
                    DM_write_enable <= 1;
                    DM_read_enable <= 0;
                    DM_address <= instruction[8:0];
                    write_en <= 0;
                    read_en <= 1;
                    read_sel <= instruction[10:9];
                    DM_in <= RF_out;
                end
                2'b10: begin // addition
                    DM_write_enable <= 0;
                    DM_read_enable <= 0;
                    write_en <= 1;
                    write_en2 <= 1;
                    write_sel <= 2'b10;
                    write_sel2 <= 2'b11;
                    ALUOp <= 2'b00;
                    ALU_in_1 <= RF_A1;
                    ALU_in_2 <= RF_A2;
                    for (integer i = 0; i < 16; i = i + 1) begin
                        RF_in[i * 32 +: 32] <= ALU_out[i * 64 +: 32];
                        RF_in2[i * 32 +: 32] <= ALU_out[i * 64 + 32 +: 32];
                    end
                end
                2'b11: begin // multiplication
                    DM_write_enable <= 0;
                    DM_read_enable <= 0;
                    write_en <= 1;
                    write_en2 <= 1;
                    write_sel <= 2'b10;
                    write_sel2 <= 2'b11;
                    ALUOp <= 2'b01;
                    ALU_in_1 <= RF_A1;
                    ALU_in_2 <= RF_A2;
                    for (integer i = 0; i < 16; i = i + 1) begin
                        RF_in[i * 32 +: 32] <= ALU_out[i * 64 +: 32];
                        RF_in2[i * 32 +: 32] <= ALU_out[i * 64 + 32 +: 32];
                    end
                end
            endcase
        end
    end
endmodule
```

در این بلاک، در هر کلاک به آپکد نگاه می‌کنیم. اگر آپکد 00 باشد یعنی load. پس می‌خواهیم از حافظه داخل رجیستر فایل بنویسیم. پس read enable مربوط به حافظه و write enable مربوط به رجیستر فایل را ۱ می‌کنیم. آدرس حافظه را به memory و آدرس رجیستر را به register file می‌دهیم تا عملیات انجام شود و در نهایت خروجی حافظه را پس از مدتی تاخیر برای اطمینان از خروجی، در ورودی رجیسترفایل میریزیم تا در رجیستر مشخص شده نوشته شود.

اگر آپکد 01 باشد یعنی store. پس می‌خواهیم از رجیستر فایل داخل حافظه بنویسیم. پس write enable مربوط به حافظه را ۱ می‌کنیم و write enable مربوط به رجیسترفایل را ۰ می‌کنیم. آدرس حافظه را به memory و آدرس رجیستر را به register file می‌دهیم تا عملیات انجام شود و در نهایت خروجی رجیسترفایل را پس از مدتی تاخیر برای اطمینان از خروجی، در ورودی حافظه میریزیم تا در خانه‌ی مشخص شده نوشته شود.

اگر آپکد 10 باشد یعنی add. پس با حافظه کاری نداریم. Write enable های رجیستر فایل را یک می‌کنیم و شماره‌ی رجیستر های A3 و A4 را برای ذخیره‌ی خروجی به آن می‌دهیم. ALUop را 00 می‌کنیم تا جمع انجام دهد و A1 و A2 را به ورودی های ALU می‌دهیم. در نهایت خروجی ALU را پس از مدتی تاخیر برای اطمینان از خروجی، در ورودی رجیسترفایل میریزیم تا در رجیسترهای مشخص شده نوشته شود.

اگر آپکد 11 باشد یعنی multiply. پس با حافظه کاری نداریم. Write enable های رجیستر فایل را یک می‌کنیم و شماره‌ی رجیستر های A3 و A4 را برای ذخیره‌ی خروجی به آن می‌دهیم. ALUop را 01 می‌کنیم تا ضرب انجام دهد و A1 و A2 را به ورودی های ALU می‌دهیم. در نهایت خروجی ALU را پس از مدتی تاخیر برای اطمینان از خروجی، در ورودی رجیسترفایل میریزیم تا در رجیسترهای مشخص شده نوشته شود.

حال ماژول تست‌بنچ را طراحی می‌کنیم:

```
module TB;
    reg clk;
    reg reset;
    reg [12:0] instruction;
    wire [511:0] A1;
    wire [511:0] A2;
    wire [511:0] A3;
    wire [511:0] A4;

    Vector_Processor processor (clk, reset, instruction, A1, A2, A3, A4);

    initial
        clk = 0;
    always
        #25 clk = ~clk;
```

کلاک و ریست و محتوای رجیسترها را می‌سازیم و با این‌ها یک instance از پردازنده‌ی ساخته شده می‌گیریم.

```
    initial begin
        reset <= 1;
        #45
        reset <= 0;
        instruction <= 1000000000000;  //  A4:A3 = A1 + A2
        #40
        instruction <= 1100000000000;  // A4:A3 = A1 * A2
        #40
        instruction <= 0000000000001;  //  A1 <= memory[000000001]
        #40
        instruction <= 0001000001000;  //  A2 <= memory[000001000]
        #40
        instruction <= 0100000000000;  //  A4:A3 = A1 + A2
        #40
        instruction <= 0110000000001;  //  memory[000000001] <= A3
        #40
        instruction <= 0111000001000;  //  memory[000001000] <= A4
        #40
        instruction <= 0000000000010;  //  A1 <= memory[000000010]
        #40
        instruction <= 0001000000011;  //  A2 <= memory[000000011]
        #40
        instruction <= 1100000000000;  //  A4:A3 = A1 * A2
        #40
        instruction <= 0110000000010;  //  memory[000000010] <= A3
        #40
        instruction <= 0111000000011;  //  memory[000000011] <= A4

        $stop;
    end

    initial
        $monitor($time, ":\nA1 = %h\nA2 = %h\nA3 = %h\nA4 = %h",
                    A1, A2, A3, A4);

endmodule
```

به این ترتیب این دستورات را اجرا می‌کنیم. کنار هرکدام نوشته شده که چه کاری انجام می‌دهند.
خروجی تست:

```
Transcript
VSIM 15> run -all
#                    0:
# A1 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# A2 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# A3 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# A4 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# --------------------
#                  150:
# A1 = 0000000000000150000ffffffff00007fffffff000ff0000f0000f00f0000f0800000000000000100000001000000ff0000000f0000000ffffffff00000000
# A2 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# A3 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# A4 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# --------------------
#                  200:
# A1 = 0000000000000150000ffffffff00007fffffff000ff0000f0000f00f0000f0800000000000000100000001000000ff0000000f0000000ffffffff00000000
# A2 = 8000000000000010000000f0000000ff0000000ffffffffffffffff0000000000000150000ffffffff00007fffffff000ff0000f0000f00f0000f080000000
# A3 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# A4 = 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# --------------------
#                  250:
# A1 = 0000000000000150000ffffffff00007fffffff000ff0000f0000f00f0000f0800000000000000100000001000000ff0000000f0000000ffffffff00000000
# A2 = 8000000000000010000000f0000000ff0000000ffffffffffffffff0000000000000150000ffffffff00007fffffff000ff0000f0000f00f0000f080000000
# A3 = 8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f00f0000ef80000000
# A4 = ffffffff0000000000000000ffffffff0000000000000000000000000000000ffffffff00000000ffffffff00000000ffffffffffffffff00000000ffffffff
# --------------------
#                  350:
# A1 = ffffffff8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f00f0000ef
# A2 = 8000000000000010000000f0000000ff0000000ffffffffffffffff0000000000000150000ffffffff00007fffffff000ff0000f0000f00f0000f080000000
# A3 = 8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f00f0000ef80000000
# A4 = ffffffff0000000000000000ffffffff0000000000000000000000000000000ffffffff00000000ffffffff00000000ffffffffffffffff00000000ffffffff
# --------------------
#                  400:
# A1 = ffffffff8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f00f0000ef
# A2 = ffffffffffffffff8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f0
# A3 = 8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f00f0000ef80000000
# A4 = ffffffff0000000000000000ffffffff0000000000000000000000000000000ffffffff00000000ffffffff00000000ffffffffffffffff00000000ffffffff
# --------------------
#                  450:
# A1 = ffffffff8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f00f0000ef
# A2 = ffffffffffffffff8000000000000160001000effff000f6fffffff000fefff0f0000ef0f0000f0800000150001000 0ffff00018000000ef00ff000ff0000f0
# A3 = 0000000180000000000000000160134000100d29000fff18ff01001ffe10f111100e0103b0013b00015000000100007ff2000e20df20000ef100002100e010
# A4 = 0000000000000000ffffff500000000ffffffffffff90060006f8ff0000ef0f00e1001cf87fff89ffff8000ffffffff00007fff07f807ff000ff001fff1000d
# --------------------
Wave  Objects  Processes  Library  Project  Memory List  sim  Transcript
```

همانطور که مشخص است، عملیات ها روی مقادیر مرزی انجام شده‌است و خروجی ها صحیح هستند.