بسمه تعالی

درس طراحی سیستم‌های دیجیتال

جبرانی پایان‌ترم – سوال ۷ (پردازنده آرایه‌ای)

استاد

مهندس امین فصحتی

دانشجو

محمد عماد چنگیزی آشتیانی

۴۰۱۱۰۵۸۲۶

دانشگاه صنعتی شریف

بهار ۱۴۰۳

فهرست

مقدمه ۳
پردازنده آرایی ۳
پیاده‌سازی ۳
طراحی رجیستر فایل ۳
طراحی memory ۴
طراحی ALU ۸
طراحی ماژول vector_processor ۹
آزمون مدار ۱۲
نتیجه: ۱۵

مقدمه

قرار است پردازنده آرایه ای را بسازیم. در این پروژه ابتدا به صورت اجمالی درباره مفهوم پردازنده آرایه‌ای صحبت کرده و در ادامه با توصیف این پردازنده در زبان وریلاگ آن را به طریقی مناسب طراحی می‌کنیم. برای این پروژه لازم است که بخش‌های مختلفی که یک پردازنده نیاز دارد مانند رجیستر فایل، واحد ALU را جداگانه طراحی کنیم و هم‌چنین یک حافظه نیز باید برای این پردازنده در نظر بگیریم. هم‌چنین برای خود پردازنده باید یک ISA طراحی کنیم تا بتوانیم دستورات مختلف را روی آن اجرا کنیم.

پردازنده آرایی

پردازنده آرایی (vector processor) نوعی پردازنده است که در آن یک تک پردازنده است که یک دستور واحد را بر روی آرایه از داده‌ها به صورت همزمان اجرا می‌کند.

پیاده‌سازی

طراحی رجیستر فایل

لازم است که در ابتدا register-file را در زبان وریلاگ توصیف کنیم.

```
module register_file (
    input[511:0] A3, A4,load_data,
    input[1:0] load_addr_reg, store_addr_reg, //todo specify the size for this
part
    input clk , reset, load, store, write_enable, read, set,
    output reg [511:0] A1, A2, store_data
);

    reg[511:0] reg_file[0:3]; // a register file with size 4*512
    integer i;
    always @(*) begin
        if (reset) begin
            for (i = 0; i < 4; i = i + 1) begin
                reg_file[i] <= 0;
            end
```

```
        end
        if(set) begin
           reg_file[0] <= {1'b1, 511'b0};
           reg_file[1] <= {1'b1, 511'b0};
           reg_file[2] <= 1000;
           reg_file[3] <= 2000;
        end
        if (write_enable) begin
            reg_file[2] <= A3;
            reg_file[3] <= A4;
        end
        if(read) begin
            A1 <= reg_file[0];
            A2 <= reg_file[1];
        end
        if(load) begin
         reg_file[load_addr_reg] <= load_data;
        end
        if(store) begin
            store_data <= reg_file[store_addr_reg];
        end

    end
endmodule
```

توضیح: همانطور که در سوال خواسته شده است، این رجیستر فایل از ۴ رجیستر ۵۱۲ بیتی تشکیل شده است. برای این ماژول پورت های آدرس دهی برای نوشتن روی رجیستر فایل و هم‌چنین خواندن از روی رجیستر فایل را در نظر گرفتیم. بقیه پورت ها نیز ورودی و خروجی‌های لازم این ماژول هستند.

به منظور نشان دادن عملکرد این مدار در ابتدا با سیگنال set می‌توانیم مقادیر از پیش‌تعیین شده‌ای را در این ماژول قرار دهیم که برای تست مقادیر مرزی مورد استفاده قرار می‌گیرند.

طراحی memory

در این بخش قصد داریم حافظه خواسته شده در پروژه را نیز طراحی کنیم. این حافظه از ۵۱۲ کلمه ۳۲ بیتی ساخته شده است بنابراین حافظه را بدین صورت طراحی می‌کنیم:

```
module memory (
    input clk, write_enable, reset, read_enable,
    input [8:0] read_address, write_address,
    input [511: 0] data,
```

```
    output reg [511:0] out
);
    reg[31:0] single_port_mem[0:511];
    integer i;
    integer j;
    always @(*) begin
        if (reset) begin
            for (i = 0; i < 512; i = i + 1) begin
                single_port_mem[i] <= 0;
            end
        end
        else if(write_enable) begin
            j = 0;
            for (i = write_address; i < write_address + 16; i = i + 1) begin
                single_port_mem[i] <= data[j +: 32];
                j = j + 32;
            end
        end
        else if(read_enable) begin
            j = 0;
            for (i = read_address; i < read_address + 16; i = i + 1) begin
                out[j +: 32] <= single_port_mem[i];
                j = j + 32;
            end
        end
    end

endmodule
```

توضیح: برای آدرس دهی به این حافظه از آنجا که ۵۱۲ خانه است از دو پورت آدرس‌دهی ۹ بیتی برای خواندن و نوشتن بر روی این حافظه استفاده کرده‌ایم. پورت data ورودی این حافظه است و اطلاعاتی است که قرار است بر روی این حافظه بنویسیم. این اطلاعات در فرمت که سیگنال ۵۱۲ بیتی به این ماژول داده می‌شود و در ادامه به روش مناسب از آدرسی که به این ماژول داده میشود ۳۲ بیت ۳۲ بیت مقادیر این رجیستر را بر روی ۱۶ خانه متوالی از این حافظه می‌نویسیم.

برای خواندن اطلاعات نیز از آدرس شروع تا ۱۶ خانه مقادیر را میخوانیم و آن را در reg ۵۱۲ بیتی out قرار می‌دهیم.

برای اینکه صرفا از صحت عملکرد ماژول حافظه مطمئن شویم یک testbench برای آن می‌نویسیم و آن را در model sim اجرا می‌کنیم.

```
module memory_tb;
```

```
reg clk, reset, write_enable, read_enable;
reg [8:0] read_address, write_address;
reg [511:0] data;
wire [511:0] out;

memory uut(
    .clk(clk),
    .reset(reset),
    .write_enable(write_enable),
    .read_enable(read_enable),
    .read_address(read_address),
    .write_address(write_address),
    .data(data),
    .out(out)
);

always #5 clk = ~clk;

initial begin
    clk = 0;
    reset = 0;
    write_enable = 0;
    read_enable = 0;
    read_address = 0;
    write_address = 0;
    data = 0;

    #10;
    $display("Test 1: reset the memory!");
    reset = 1;
    #10;
    reset = 0;
    read_enable = 1;
    read_address = 10;
    #10;
    $display("time: %0t, the output is: %b", $time, out);
    $display("Test 2: write some data on memory and read the data");
    #10;
    read_enable = 0;
    write_enable = 1;
    write_address = 10;
    data = 120;
    #10;
    read_enable = 1;
```

```verilog
        write_enable = 0;
        read_address = 10;
        #10;
        $display("time: %0t, the output is: %b", $time, out);
        $display("Test 3: write another data on that location and read");
        #10;
        read_enable = 0;
        write_enable = 1;
        write_address = 10;
        data = 15;
        #10;
        read_enable = 1;
        write_enable = 0;
        read_address = 10;
        #10;
        $display("time: %0t, the output is: %b", $time, out);
        $display("Test 4: reset the memory on that location");
        reset = 1;
        #10
        reset = 0;
        write_enable = 0;
        read_enable = 1;
        read_address = 10;
        #10
        $display("time :%0t, the output is: %b", $time, out);
        $stop;
    end


endmodule
```

اکنون با شبیه‌سازی این آزمون می‌خواهیم مطمئن شویم که عملیات نوشتن و خواندن بر روی ۱۶ خانه متوالی از حافظه به درستی انجام می‌شود.

```
VSIM 30> run -all
# Test 1: reset the memory!
# time: 30, the output is: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# Test 2: write some data on memory and read the data
# time: 60, the output is: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111000
# Test 3: write another data on that location and read
# time: 90, the output is: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111
# Test 4: reset the memory on that location
# time :110, the output is: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
# Break in Module memory_tb at C:/altera/13.1/dsd/final/memory_tb.v line 73
```

بنابراین از صحت عملکرد این ماژول به تنهایی اطمینان خاطر داریم.

طراحی ALU

در این بخش ALU را طراحی می‌کنیم. واحد محاسبات خواسته شده در سوال فقط لازم است از عملیات ضرب و جمع پشتیبانی کند:

```
module ALU (
    input clk, mul, add, reset,
    input[511:0] A1, A2,
    output [511:0] A3, A4
);
    reg[1023:0] alu_result;
    assign A3 = alu_result[511:0];
    assign A4 = alu_result[1023:512];
    always @(*) begin
        if (reset)
            alu_result = 0;
        else if (mul)
            alu_result = A1 * A2;
        else if (add) begin
            alu_result = A1 + A2;
        end
    end

endmodule
```

توضیح: در این ماژول همانطور که پیداست، ۵۱۲ بیت پرارزش در رجیستر A4 و ۵۱۲ بیت پرارزش و ۵۱۲ بیت کم ارزش حاصل از محاسبات در رجیستر A3 قرار می‌گیرد.

طراحی ماژول vector_processor

اکنون که بخش‌های مختلف این پردازنده را طراحی کرده‌ایم، لازم است با در طراحی یک DataPath و یک ControlUnit و با کنار هم قرار دادن ماژول های بالا، پردازنده آرایه‌ای خود را بسازیم. پیش از آن لازم است که یک ISA برای این پردازنده طراحی کنیم.

در این پردازنده طبق خواسته سوال ۴ نوع دستور وجود دارد که عبارتند از:

۱. بارگذاری از حافظه بر روی یکی از رجیسترها

۲. ذخیره کردن مقادیر یکی از رجیسترها بر روی حافظه

۳. انجام عملیات جمع

۴. انجام عملیات ضرب

اکنون برای ۴ دستور بالا دستورات زیر را ارائه می‌دهیم:

Load Reg , Mem = <opcode = 00> <Reg_number = 2bit> <mem_location = 9bit>

Store Reg, Mem = <opcode = 01> <Reg_number = 2bit> <mem_location = 9bit>

Add Reg1, Reg2 = <opcode = 10> <00000000000>

Mul Reg1 , Reg2 = <opcode = 11> <0000000000>

بنابراین مجموعا ۴ دستور داریم که طول همه آن ها یکسان و ۱۳ بیت است و به شرح بالا می‌باشد.

دقت شود که از آنجا که در سوال همواره برای عملیات جمع و ضرب دو رجیستر ثابت A1 ,A2 را به ترتیب جمع و ضرب می‌کنیم و در رجیسترهای A3, A4 می‌نویسیم، صرفا با داشتن opcode می‌توانیم آن‌ها را اجرا کنیم.

اکنون با این توضیحات ماژول زیر را تعریف می‌کنیم:

```
module vector_processor (
    input clk, reset, set,
    input[12:0] instruction_set,
    output [511:0] mem_written, A3_out, A4_out, Register_out
);

    reg[1:0] opcode;

    // connections of the alu instance
    reg mul, add;
    wire[511:0] A1, A2;
    wire[511:0] A3, A4;
```

```
// connections of the mem instance
reg mem_read_enable, mem_write_enable;
reg[8:0] mem_read_address, mem_write_address;
wire [511:0] mem_data;
wire[511:0] out;

// connection of the reg_f instance
wire [511:0] regf_A3, regf_A4, load_data;
wire [511:0] A1_regf, A2_regf, regf_store_data;
reg[1:0] load_addr_reg, store_addr_reg;
reg load_regf, regf_store, regf_write_enable, regf_read_enable;

register_file reg_f(
    .A3(regf_A3),
    .A4(regf_A4),
    .load_data(load_data),
    .clk(clk),
    .reset(reset),
    .load(load_regf),
    .store(regf_store),
    .set(set),
    .write_enable(regf_write_enable),
    .read(regf_read_enable),
    .load_addr_reg(load_addr_reg),
    .store_addr_reg(store_addr_reg),
    .A1(A1_regf),
    .A2(A2_regf),
    .store_data(regf_store_data)
);

ALU alu(
    .clk(clk),
    .mul(mul),
    .add(add),
    .reset(reset),
    .A1(A1),
    .A2(A2),
    .A3(A3),
    .A4(A4)
);

memory mem(
    .clk(clk),
    .reset(reset),
```

```
        .write_enable(mem_write_enable),
        .read_enable(mem_read_enable),
        .read_address(mem_read_address),
        .write_address(mem_write_address),
        .data(mem_data),
        .out(out)
    );

    // initialize the register file with random numbers


    assign mem_written = regf_store_data;
    assign mem_data = regf_store_data;
    assign A3_out = A3;
    assign A4_out = A4;
    assign regf_A3 = A3;
    assign regf_A4 = A4;
    assign Register_out = out;
    assign load_data = out;
    assign A1 = A1_regf;
    assign A2 = A2_regf;
    always @(posedge clk) begin
        opcode = instruction_set[12:11];
        load_addr_reg = instruction_set[10:9];
        store_addr_reg = instruction_set[10:9];
        mem_read_address = 0;
        load_regf = 0;
        mem_read_enable = 0;
        mem_write_enable = 0;
        regf_store = 0;
        add = 0;
        mul = 0;
        regf_read_enable = 0;
        regf_write_enable = 0;
        case (opcode)
            2'b00: begin // load from memory to register
                mem_read_enable = 1;
                mem_read_address = instruction_set[8:0];
                load_regf = 1;
            end
            2'b01: begin // store register content in memory
                regf_store = 1;
                mem_write_enable = 1;
                mem_write_address = instruction_set[8:0];
            end
```

```
            2'b10: begin // ALU add
                regf_read_enable = 1;
                add = 1;
                regf_read_enable = 1;
                regf_write_enable = 1;
            end
            2'b11: begin // ALU Multiplication
                regf_read_enable = 1;
                mul = 1;
                regf_write_enable = 1;
            end
        endcase
    end
endmodule
```

در این ماژول این امکان را فراهم کرده ایم که صرفا با دادن instruction به این ماژول تمامی دستورات خواسته شده به بهترین شکل اجرا شوند.

اکنون و در ادامه برای بررسی صحت عمل باید یک ماژول testbench برای پردازنده آرایی طراحی شده بنویسیم:

آزمون مدار

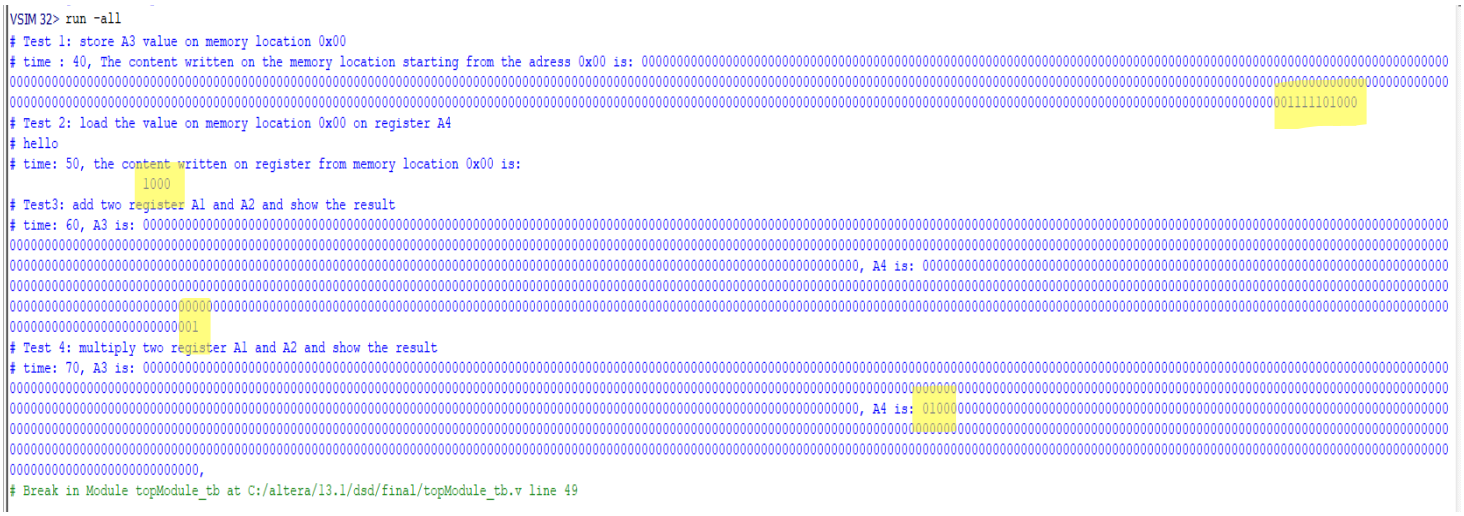
در ادامه ماژول testbench را به شرح زیر در وریلاگ توصیف می‌کنیم:

```
module topModule_tb;
    reg clk, reset, set;
    reg [12:0] command;
    wire[511:0] mem_written, A3_out, A4_out, Register_out;



    vector_processor proc (
        .clk(clk),
        .set(set),
        .reset(reset),
        .instruction_set(command),
        .mem_written(mem_written),
        .A3_out(A3_out),
        .A4_out(A4_out),
        .Register_out(Register_out)
    );
```

```
    always #5 clk = ~clk;

    initial begin
        clk = 0;
        reset = 0;
        #10
        reset = 1;
        #10
        reset = 0;
        set = 1;
        #10
        set = 0;
        $display("Test 1: store A3 value on memory location 0x00");
        command = 13'b0110000000000;
        #10
        $display("time : %0t, The content written on the memory location starting
from the adress 0x00 is: %b", $time, mem_written);
        $display("Test 2: load the value on memory location 0x00 on register
A4");
        command = 13'b0011000000000;
        #10
        $display("time: %0t, the content written on register from memory location
0x00 is: %d", $time, Register_out);
        $display("Test3: add two register A1 and A2 and show the result");
        command = 13'b1000000000000;
        #10
        $display("time: %0t, A3 is: %b, A4 is: %b", $time, A3_out, A4_out);
        $display("Test 4: multiply two register A1 and A2 and show the result");
        command = 13'b1100000000000;
        #10
        $display("time: %0t, A3 is: %b, A4 is: %b,", $time, A3_out, A4_out);
        $stop;
    end



endmodule
```

در این ماژول در ۴ آزمون مختلف دستورات مختلف را مورد ارزیابی قرار می‌دهیم. هر بار صرفا دستور متفاوتی را به پردازنده می‌دهیم و پردازنده با پردازش ماشین کد داده شده، آن دستور را اجرا می‌کند.

اکنون با شبیه سازی کردن در نرم افزار model sim نتیجه زیر حاصل می‌شود:



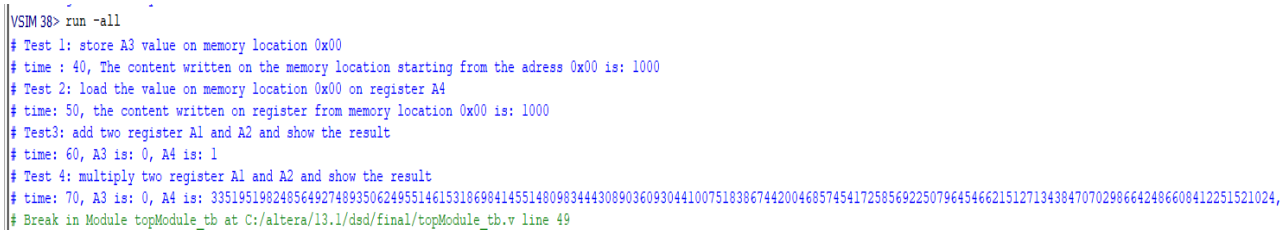
توضیح: در آزمون اول همانطور که پیدا است، مقداری رو بر روی حافظه از رجیستر A3 می‌نویسیم. می‌دانیم مقدار اولیه رجیستر A3 عدد ۱۰۰۰ است و در اینجا عدد ۱۰۰۰ به صورت دودویی در حافظه نوشته شده است و در اینجا آن را نمایش داده‌ایم.

در آزمون دوم نیز محتوای نوشته در آن آدرس را مجددا بر روی رجیستر A4 می‌نویسیم و آن را نمایش میدهیم. که مقدار آن ۱۰۰۰ شده است. در صورتی که مقدار اولیه رجیستر A4، ۲۰۰۰ بود. دقت شود که عدد را در اینجا با فرمت دهدهی نشان داده‌ایم.

در آزمون سوم محتوای دو رجیستر A1 و A2 را با هم جمع کرده‌ایم. در اینجا مستقیما از یک مقدار مرزی استفاده کرده ایم تا مطمئن شویم در صورتی که حاصل جمع بیشتر از ۵۱۲ بیت شود نیز جمع به درستی انجام می‌شود. در اینجا مقدار اولیه رجیستر A1 و A2، ۲ به توان ۵۱۱ بوده است. واضح است که جمع این دو رجیستر در ۵۱۲ بیت جا نمی‌شود. همانطور که در شکل هایلایت نیز شده است. کوچک ترین بیت رجیستر A4 یک شده است و مابقی بیت ها ۰ هستند. این یعنی که عملیات جمع به درستی انجام شده است زیرا بیت اول رجیستر A4 بیت ۵۱۳ ام است که ۱ باید در آن قرار می‌گرفت.

در آزمون چهارم نیز محتوای دو رجیستر A1 و A2َ نیز در هم ضرب شده اند که نتیجه نیز واضحا درست است و آن را هایلات کرده ام.

حال برای اینکه نتایج را در مبنای دهدهی نیز ببینیم تا نمایش مرتب‌تری داشته باشیم. مجددا آزمون فوق را انجام میدهیم ولی در مبنای ذکر شده:



نتیجه:

در این پروژه توانستیم یک پردازنده آرایه‌ای بسیار ساده را در زبان وریلاگ توصیف کنیم. در این پردازنده بسیار از موارد مانند موازی سازی، پایپ‌لاین و ... که در یک پردازنده واقعی نیاز به طراحی شدن دارند، طراحی نشده است اما فریم و قالب کلی هر پردازنده که شامل یک رجیستر فایل و واحد ALU است طراحی شده اند. برای هر کدام از ماژول های جداگانه نیز testbench ای نوشته ام که آن را ضمیمه فایل‌ها کرده و در سایت گیت هاب قرار داده ام.