آرین گرامی راد 401110631

بخش اول: توضیح کد

پردازنده از 4 ماژول ساخته شده. بصورت Bottom - up توضیح می دهم.

پایین ترین ماژول در طراحی من ALU هست که در ماژول Register file استفاده می شود.

ماژول ALU:

```
1  module alu(
2      input clk,
3      input rst,
4      input [511:0] A1,
5      input [511:0] A2,
6      input [511:0] A3,
7      input [511:0] A4,
8      input [1:0] op,
9      output reg [511:0] write_on_A3,
10     output reg [511:0] write_on_A4
11 );
12     always @(clk) begin
13         if(rst) begin
14             case (op)
15                 2'b10: begin
16                     {write_on_A4, write_on_A3} <= A1 + A2;
17                 end
18                 2'b11: begin
19                     {write_on_A4, write_on_A3} <= A1 * A2;
20                 end
21                 default: begin
22                     write_on_A3 <= A3;
23                     write_on_A4 <= A4;
24                 end
25             endcase
26         end
27
28     end
29 endmodule
```

جدا از ورودی clk و rst که کارکردشان مشخص هست، این ماژول 5 ورودی دیگر دارد.

A1 : مقدار رجیستر A1 _ A2 : مقدار رجیستر A2 _ A3 : مقدار رجیستر A3 _ A4 : مقدار رجیستر A4

op : عددی 2 بیتی به منظور دادن دستور (Operation Code)

شاید بگویید این ماژول فقط A1 و A2 را برای انجام عملیات جمع و ضرب نیاز دارد پس چرا A3 و A4 هم خوانده شود؟

بدلیل آنکه در دو دستور بعدی Op=00 و Op=01 این ماژول باید حالت پیشین خود را حفظ کند.

در آخر این ماژول به ازای Op=10 جمع و Op=11 ضرب انجام می‌دهد و هر دوی این عملیات نتایج خود را بر دو خروجی Write_on_A3 و Write_on_A4 می‌نویسند.

اگر جمع ما منجر به overflow شود، این مقدار اضافه بر روی Write_on_A4 می‌رود.

در ضرب دو عدد 512 بیتی هم 512 بیت کوچکتر در Write_on_A3 و 512 بیت بزرگتر هم بر Write_on_A4 می‌رود.

در تصویر بالا به ازای rst=0 کاری انجام نمی‌شود که البته این مشکل دارد و در کد نهایی برطرف خواهد شد.

ماژول Register file:

```v
module register_file(
    input clk,
    input rst,
    input [1:0] op_code,
    input [1:0] read_addr,
    input [1:0] write_addr,
    input [511:0] write_data,
    output reg [511:0] read_data
);
    reg [511:0] registers [3:0];
    wire [511:0] A3_res;
    wire [511:0] A4_res;

    initial begin
        $monitor("time: %t\nA1: %d\nA2: %d\nA3: %d\nA4: %d\n-------",$time, registers[0],registers[1],registers[2],registers[3]);
    end

    alu alu_unit(
        .clk(clk),
        .rst(rst),
        .A1(registers[0]),
        .A2(registers[1]),
        .A3(registers[2]),
        .A4(registers[3]),
        .op(op_code),
        .write_on_A3(A3_res),
        .write_on_A4(A4_res)
    );

    always @(posedge clk) begin
        if (!rst) begin
            registers[0] <= 512'b0;
            registers[1] <= 512'b0;
            registers[2] <= 512'b0;
            registers[3] <= 512'b0;
        end
        else begin
            case (op_code)
                2'b00: begin
                    #1 registers[write_addr] <= write_data;
                end
                2'b01: begin
                    read_data <= registers[read_addr];
                end
                default: begin
                    registers[2] <= A3_res;
                    registers[3] <= A4_res;
                end
            endcase
        end
    end

endmodule
```

پس از ماژول ALU به سراغ ماژول رجیسترها می رویم. در این ماژول هر تغییر بخواهد روی مقادیر رجیسترها صورت بگیرد، انجام می‌شود.

همچنان جدا از CLK و rst که واضح هست. ما چند ورودی دیگر داریم:

- Op_code : همان Operation Code برای دستور دادن به پردازنده هست.

- read_addr : آدرسی 2 بیتی هست که در هنگام Op=01، مقدار خانه رجیستر متناظر با آن آدرس را بر روی خروجی read_data می برد.

- write_addr : آدرسی 2 بیتی ای می باشد که در Op=00 مقدار ورودی دیگر یعنی write_data را بر روی رجیستر با این آدرس می ریزد.

- write_data : در بالا توضیح داده شد.

این ماژول 4 رجیستر 512 بیتی دارد. همچنین 2 net دارد تا مقادیر خروجی ALU را به رجیستر منتقل کند. که به ترتیب هر یک A3_res و A4_res نام دارند.

همچنین این ماژول برای تست یک monitor دارد تا تغییرات مقادیر رجیسترها را نمایش دهد. باید توجه داشت در هنگام ست نهایی این بخش حذف گردد. این مورد صرفا برای تست و خروجی گرفتن تعبیه شده بود.

در ماژول یک instance از ALU تعریف دارد که در Op Code متناظر جمع و ضرب انجام دهد.

در نهایت می بینید که ماژول ما حساس به لبه مثبت CLK هست و rst بصورت Synchron می باشد.

در دستور Op=00 که مشابه دستور Load هست، داده write_data بر آدرس write_addr نوشته می شود.
در دستور Op=01 که مشابه دستور Store هست، داده آدرس read_addr خروجی داده می شود.
در دو دستور Op=11 و Op=10 مقادیر ALU در A3 و A4 ذخیره می گردد.

* پس از پایان طراحی اولیه این دو ماژول با استفاده از یک test bench از صحت عملکرد آن مطمئن میشیم. نتایج این تست را می‌توانید در صفحه اصلی ریپو Github ملاحظه فرمایید.

ماژول Memory :

```
1  module memory(
2      input clk,
3      input rst,
4      input [1:0] op_code,
5      input [8:0] mem_addr,
6      input [511:0] mem_wr_data,
7      output reg [511:0] mem_rd_data
8  );
9      reg [31:0] mem_array [511:0];
10     integer i;
11     integer j;
12     integer k;
13     reg [511:0] dummy;
14
15     initial begin
16         for (i = 0; i < 512; i = i + 1) begin
17             mem_array[i] <= 32'b0;
18         end
19
20         mem_array[0] <= 32'd1234;
21
22         mem_array[16] <= 32'd8765;
23     end
24
25     always @(posedge clk) begin
26         if (!rst) begin
27             for (i = 0; i < 512; i = i + 1) begin
28                 mem_array[i] <= 32'b0;
29             end
30         end
31
32         else if (op_code == 2'b01) begin
33             #1;
34             for (j = 0; j < 16; j = j + 1) begin
35                 mem_array[mem_addr + j] <= mem_wr_data[(j*32) +: 32];
36             end
37         end
38
39
40         else if (op_code == 2'b00) begin
41             for (k = 0; k < 16; k = k + 1) begin
42                 mem_rd_data[(k*32) +: 32] <= mem_array[mem_addr + k];
43             end
44         end
45     end
46
47 endmodule
```

این خط اضافه هست برای dummy data ولیست گذاشته شده بود و باید پاک گردد.

در کل ساختار حافظه ی ما به این شکل هست که 512 خط حافظه 32 بیتی دارد. این یعنی برای خواندن و نوشتن داده های 512 بیتی رجیستر ها ما باید 16 خانه حافظه را بخوانیم یا بنویسیم.

در صورت rst=0 تک تک خانه های حافظه مقدار صفر می‌گیرند.

ادامه در صفحه بعد.

بطور کلی دو دستور پردازنده هست که با آن سروی دیگری میتوند یکی Op=00 و یکی Op=01 .

دستور Op=00 : این دستور ، دستور Load هست پس باید با گرفتن این دستور با استفاده از mem_addr برویم و از خانه متناظر با آن آدرس، تا 16 خانه بعد از آن را بخوانیم.
و داده خوانده شده را بر روی mem_rd_data بذیریم تا در ماژول نهایی به Register file بدهیم.

دستور Op=01 : این دستور ، دستور Store هست و باید با گرفتن آن برویم و داده ورودی mem_wr_data را بر روی آدرس mem_addr تا 16 خانه بعد بذیریم.

* پس از پایان طراحی اولیه این ماژول هم با استفاده از یک test bench از صحت عملکرد آن مطمئن شدیم . نتایج این تست را میتوانید در صفحه اصلی ریپو Github ملاحظه فرمایید.

ماژول Vec Processor :

```
module vector_processor(
    input clk,
    input rst,
    input [1:0] op_code,
    input [1:0] reg_addr_to_write,
    input [1:0] reg_addr_to_read,
    input [8:0] mem_addr
);
    wire [511:0] mem_out_to_reg_file;
    wire [511:0] reg_out_to_mem;

    register_file rf (
        .clk(clk),
        .rst(rst),
        .op_code(op_code),
        .read_addr(reg_addr_to_read),
        .write_addr(reg_addr_to_write),
        .write_data(mem_out_to_reg_file),
        .read_data(reg_out_to_mem)
    );

    memory mem(
        .clk(clk),
        .rst(rst),
        .op_code(op_code),
        .mem_addr(mem_addr),
        .mem_wr_data(reg_out_to_mem),
        .mem_rd_data(mem_out_to_reg_file)
    );

endmodule
```

در نهایت در این ماژول memory را به رجیستر فایل متصل کردم.

کلاً 4 ورودی اساسی دارد:

1. Op_code : که میان تمام instance ها یکسان هست و برای دستور دادن هست.
2. reg_addr_to_write : آدرس برای نوشتن در رجیستر فایل
3. reg_addr_to_read : " " " خواندن " " "
4. mem_addr : آدرس برای خواندن و نوشتن در حافظه.

در نهایت پردازنده این چنین کار می‌کند:

Op = 00 (Load) : با mem_addr می‌رود و داده را از حافظه می‌خواند و با reg_addr_to_write می‌فهمد در کدام رجیستر بنویسد.

Op = 01 (Store) : با reg_addr_to_read می‌فهمد کدام رجیستر را بخواند و با mem_addr می‌فهمد در کجا حافظه ذخیره کند.

Op = 10 (sum) : مقدار A1 را با A2 جمع می‌کند و در A3 و A4 ذخیره می‌کند.

Op = 11 (Multiply) : مقدار A1 را با A2 ضرب می‌کند و در A3 و A4 ذخیره می‌کند.

* پس از پایان طراحی اولیه این ماژول هم با استفاده از یک test bench از صحت عملکرد آن مطمئن شدم. نتایج این تست را می‌توانید در صفحه اصلی ریپو Github ملاحظه فرمایید.