مشخصات پردازنده طراحی شده به شرح زیر می باشد:

- این پردازنده از کلمات ۴ بایتی پشتیبانی میکند
- در این پردازنده از یک حافظه با ظرفیت 2kB استفاده می گردد
- ۴ رجیستر ۵۱۲ بیتی ( یا ۱۶ کلمه ای ) در این پردازنده موجود می باشد که با اندیس های ۰۰ تا ۱۱ شماره گذاری شده اند.

در این پردازنده از ۴ نوع دستور پشتیبانی میگردد:

- دستور load: ۱۶ کلمه از حافظه را به صورت موازی در یک رجیستر ذخیره میکند
- دستور store: ۱۶ کلمه را از یک رجیستر درون حافظه به صورت موازی ذخیره میکند.
- دستور add: حاصل جمع رجیستر ۰۰ و ۰۱ را محاسبه کرده و در دو رجیستر ۱۰ و ۱۱ ذخیره می کند ( رجیستر ۱۱ برای بیت های پرارزش تر)
- دستور multiply: حاصل ضرب رجیستر ۰۰ و ۰۱ را محاسبه کرده و در دو رجیستر ۱۰ و ۱۱ ذخیره می کند ( رجیستر ۱۱ برای بیت های پرارزش تر)

همچنین فرمت نوشتن این دستورات به صورت زیر می باشد:

| Instruction | Opcode | Rs | Memory Address |
|---|---|---|---|
| Load | 00 | 00 to 11 | 00000 to 11111 |
| Store | 01 | 00 to 11 | 00000 to 11111 |
| Add | 10 | X | X |
| Multiply | 11 | X | X |

در این ادامه به بررسی قسمت های مختلف پردازنده طراحی شده می پردازیم.

## ALU:

```
module ALU(
    input opcode,
    input [511:0] in1, in2,
    output [511:0] out1, out2
);

    genvar i;
    generate
        for (i = 0; i < 16; i = i + 1) begin
            assign {out2[32 * i +: 32], out1[32 * i +: 32]} = opcode == 0 ?
            $signed(in1[32 * i +: 32]) + $signed(in2[32 * i +: 32])
            : $signed(in1[32 * i +: 32]) * $signed(in2[32 * i +: 32]);
        end
    endgenerate

endmodule
```

در این ماژول، واحد محاسباتی را طراحی کرده ایم، به این صورت که اگر opcode ای که توسط cpu به این بخش داده می شود ( یا همان بیت دوم instruction از سمت چپ) برابر ۰ باشد، عملیات جمع و اگر ۱ باشد عملیات ضرب انجام می شود. لازم به ذکر است که هر دو عملیات های ضرب و جمع به صورت elementwise و برداری انجام می شوند و به همین دلیل از بلوک generate استفاده کرده ایم که ۳۲ بیت ۳۲ بیت اعداد متناظر را جمع و یا ضرب کند.

## Memory:

```
module Memory (
    input clk,
    input is_store,
    input is_load,
    input [4:0] address,
    input [511:0] in_data,
    output reg [511:0] out_data
);
    reg [31:0] memory [0:511];
    initial begin
        $readmemh("in_memory.txt", memory);
    end

    wire [8:0] int_address = address << 4;
```

در تصویر فوق متغیر های مهم ماژول حافظه مشاهده می شود. ورودی های is_store, is_load مشخص میکنند که آیا دستور از جنس store یا load ( یا هیچ کدام ) است. همچنین address آدرس بلوک ۱۶ تایی ( از کلمات ۴ بایتی ) که میخواهیم در آن خانه ها بنویسیم یا بخوانیم را ذخیره دارد. in_data, out_data نیز مربوط به داده ورودی و محل قرار دادن خروجی می باشند. متغیر memory وظیفه ذخیره سازی مقادیر main memory را دارد. برای اینکه در ابتدای برنامه از حافظه ورودی مقادیر را در این متغیر بریزیم از بلوک initial استفاده کرده ایم. در آخر متغیر int_address را تعریف کردیم که آدرس بلوک را به آدرس کلمه تبدیل کند.

```
    integer i;
    always @(posedge clk) begin
        if(is_load)
        begin
            for (i = 0; i < 16; i = i + 1)
                out_data[32 * i +: 32] = memory[int_address + i];
        end
        if(is_store)
        begin
            for (i = 0; i < 16; i = i + 1)
                memory[int_address + i] = in_data[32 * i +: 32];
            $writememh("out_memory.txt", memory);
        end
    end
endmodule
```

در این قسمت عملکرد حافظه که به صورت synchronous هست را مشاهده میکنید. با دستور load، ۱۶ کلمه ۴ بایتی در رجیستر مد نظر ریخته می شود. همچنین با دستور store عکس این اتفاق افتاده است. همچنین برای اینکه مقادیر داخل memory بعد از اتمام برنامه هم قابل رویت باشد، مقادیر را در فایل دیگری با نام out_memory ذخیره میکنیم. این فایل با هر بار استفاده از دستور store بروز رسانی می شود.

## :RegisterFile

```
module RegisterFile(
    input clk,
    input we1, input we2,
    input [1:0] in_reg1, in_reg2,
    input [511:0] in_data1, in_data2,
    output [511:0] register0, register1, register2, register3
);

    reg [511:0] registers [0:3];
    assign register0 = registers[0];
    assign register1 = registers[1];
    assign register2 = registers[2];
    assign register3 = registers[3];

    always @(posedge clk) begin
        if (we1) begin
            registers[in_reg1] = in_data1;
        end
        if (we2) begin
            registers[in_reg2] = in_data2;
        end
    end

endmodule
```

این ماژول برای پیاده سازی register های ۵۱۲ بیتی نوشته شده است. مقادیر رجیستر ها در متغیر registers ذخیره می شوند. همچنین در هر کلاک میتوانیم تا ۲ رجیستر به صورت همزمان write کنیم. عملیات read در رجیستر ها نیز به صورت asynchronous قابل انجام است.

## CPU:

این ماژول، ماژول اصلی است که سایر قطعات را به یکدیگر متصل میکند.

```
module CPU(
    input clk,
    input [8:0] instruction
);

    wire [1:0] rs = instruction[6:5];
    wire [4:0] address = instruction[4:0];
    wire is_load = instruction[8:7] == 2'b00;
    wire is_store = instruction[8:7] == 2'b01;
    wire is_alu = instruction[8] == 1'b1;
```

در تصویر فوق متغیر های تعریف شده را مشاهده میکنید. متغیر rs, address برای جداسازی field های instruction تعریف شده اند. همچنین متغیر های is_load, is_store, is_alu هر کدام شرط بودن یک دستور را بر اساس دوبیت اول instruction مشخص میکنند.

```
wire [511:0] registers [0:3];
wire [511:0] data_in0 = is_load ? mem_out_data : ALU_out[0];
wire [511:0] data_in1 = ALU_out[1];
wire [1:0] in_reg0 = is_load ? rs : 2'b10;
wire [1:0] in_reg1 = 2'b11;
wire we0 = is_store ? 0 : 1;
wire we1 = is_alu ? 1 : 0;

RegisterFile rf(clk, we0, we1, in_reg0, in_reg1, data_in0, data_in1, registers[0], registers[1], registers[2], registers[3]);
```

در ادامه، متغیر های مربوط به ماژول RegisterFile مشهود است. متغیر registers که خروجی این ماژول می باشد. متغیر های data_in به عنوان write data برای این ماژول استفاده می شوند که با توجه به حالات مختلف دستورات مقدار گرفته اند. متغیر های in_reg, we هم به عنوان write register, write enable برای این ماژول استفاده شده اند که باز متناسب با هر دستور مقدار گرفته اند. علت اینکه از هر متغیر دو تا داریم نیز این است که قابلیت نوشتن همزمان در دو رجیستر از رجیسترفایل را دارا هستیم.

```
wire [511:0] ALU_out [0:1];
ALU alu(instruction[7], registers[0], registers[1], ALU_out[0], ALU_out[1]);

wire [511:0] mem_in_data = registers[rs];
wire [511:0] mem_out_data;
Memory memory(clk, is_store, is_load, address, mem_in_data, mem_out_data);
```

در انتها، بخش های مربوط به ALU, Memory دیده می شوند. ورودی های ALU دو رجیستر اولی و خروجی های آن در ALU_out ریخته می شود. همچنین به عنوان opcode برای alu بیت instruction[7] ( یا همان بیت دوم از سمت چپ ) را داده ایم که بر حسب آن، جمع یا ضرب انجام دهد. برای memory نیز یک متغیر mem_in_data داریم که برای دستور store استفاده شده و مقدار آن، مقدار رجیستری است که rs مشخص میکند. متغیر mem_out_data نیز خروجی memory را برای نوشته شدن در رجیستر ها دریافت میکند.