

# پروژه پایانی درس طراحی سیستم‌های دیجیتال

دانشگاه صنعتی شریف - نیمسال دوم ۱۴۰۲

استاد: دکتر فصحتی  
دانشجو: روزین تقی زادگان  
شماره دانشجویی: ۴۰۱۱۰۵۷۷۵



**فهرست مطالب**

2 .....	ماژول‌های مورد نیاز
2.....	ALU
3.....	Memory
4.....	Register File
5 .....	ماژول نهایی پردازنده
6 .....	Test Bench
8.....	تست حالات مرزی
9.....	نتیجه تست حالات مرزی
11.....	تست حالات رندوم
12.....	نتیجه تست حالات رندوم

در این پژوهه قصد داریم یک پردازنده آرایه‌ای را که روی بخش‌های ۱۲ بیتی رجیسترها و ۵ بیتی کار می‌کند طراحی کنیم.

## ماژول‌های مورد نیاز

با توجه به توصیف داده شده از پردازنده آرایه‌ای، به ماژول‌های زیر برای ساخت این پردازنده احتیاج داریم:

- ALU
- Memory
- Register File

```
module ALU (clk, opcode, R1, R2, R3, R4);

input clk;
input [1:0] opcode;
input [511:0] R1, R2;
output reg [511:0] R3, R4;

reg signed [31:0] A [15:0];
reg signed [31:0] B [15:0];
reg signed [63:0] C [15:0];

always @ (posedge clk) begin
    R3 = 512'bZ;
    R4 = 512'bZ;
    #1;

    case (opcode)
        2'b10: begin
            for (integer i = 0; i < 16; i = i + 1) begin
                A[i] = $signed(R1[i*32 +: 32]);
                B[i] = $signed(R2[i*32 +: 32]);
                C[i] = $signed(A[i] + B[i]);
                R3[i*32 +: 32] = C[i][31:0];
                R4[i*32 +: 32] = C[i][63:32];
            end
        end
        2'b11: begin
            for (integer i = 0; i < 16; i = i + 1) begin
                A[i] = $signed(R1[i*32 +: 32]);
                B[i] = $signed(R2[i*32 +: 32]);
                C[i] = $signed(A[i] * B[i]);
                R3[i*32 +: 32] = C[i][31:0];
                R4[i*32 +: 32] = C[i][63:32];
            end
        end
    endcase
end
```

### ALU

در این ماژول انجام دو عمل جمع و ضرب در پردازنده آرایه‌ای پیاده‌سازی شده است. در ابتدا ورودی‌ها و خروجی‌های ماژول معرفی شده‌اند.

سپس دو آرایه A و B به گونه‌ای تعریف شده‌اند که در هر مرحله یک بخش ۳۲ بیتی از ورودی اول و دوم ماژول را در خود نگه می‌دارند و آرایه C به گونه‌ای تعریف شده که در هر مرحله نتیجه ۶۴ بیتی عمل جمع یا ضرب A و B را در خود نگه می‌دارد. سپس بخش کم‌ارزش C در یک بخش ۳۲ بیتی از خروجی اول (R3) و بخش پرارزش آن در یک بخش ۳۲ بیتی از خروجی دوم (R4) ذخیره می‌شود.

به این ترتیب در ابتدای هر عملیات در این ماژول، ابتدا وارد یک حلقه ۱۶ تایی می‌شویم تا بتوانیم عملیات مورد نظر را ۱۶ بار روی بخش‌های ۳۲ بیتی ورودی‌ها انجام دهیم تا همه ۵۱۲ بیت ورودی‌ها را تحت پوشش قرار دهیم. سپس در هر دور حلقه یک بخش ۳۲ بیتی از ورودی اول و دوم را در A و B متناظر نگه می‌داریم و در C متناظر عملیات مورد نظر را انجام می‌دهیم و سپس در نتیجه را در ۳۲ بیت متناظر از R3 و R4 قرار می‌دهیم.

## Memory

```

module Memory (clk, opcode, in, mem_addr, out);

input clk;
input [1:0] opcode;
input [511:0] in;
input [8:0] mem_addr;
output reg [511:0] out;

reg [31:0] memory [0:511];
wire [8:0] aligned_addr;
assign aligned_addr = mem_addr - (mem_addr % 16);

initial begin
    $readmemh("hex_file.txt", memory);
end

integer i;
always @ (posedge clk) begin
    out = 512'bZ;

    case (opcode)
        2'b00: begin
            for (i = 0; i < 16; i = i + 1)
                out[i*32 +: 32] <= memory[aligned_addr + i];
        end

        2'b01: begin
            #5;
            for (i = 0; i < 16; i = i + 1)
                memory[aligned_addr + i] <= in[i*32 +: 32];
        end
    endcase
end

endmodule

```

```

import random

def generate_512bit_hex():
    return ''.join(random.choices('0123456789ABCDEF', k=8))

def write_to_file(filename, num_lines):
    with open(filename, 'w') as f:
        for _ in range(num_lines):
            hex_number = generate_512bit_hex()
            f.write(f"{hex_number}\n")

# Generate a file with 512 lines of 512-bit hex numbers
write_to_file('hex_file.txt', 512)

```

کد برای generate python کردن اعداد رندوم برای ذخیره در حافظه

حافظه پردازنده در این مازول پیاده‌سازی شده است. در ابتدا ورودی‌ها و خروجی‌های مازول تعریف شده‌اند.

سپس بخش حافظه را تعریف می‌کنیم که یک آرایه ۱۲۵۱۲ بازه‌های reg می‌باشد. همچنین برای این که آدرس‌هایی که در این مازول به آن‌ها دسترسی داده می‌شود باشند، آدرس ورودی مازول را منهای باقی‌مانده آن بر ۱۶ می‌کنیم تا آدرسی که به آن دسترسی انجام می‌دهیم همواره aligned باشد. همچنین این کار باعث می‌شود در صورتی که آدرس حافظه از عدد ۴۹۶ بزرگتر باشد و عمللا ۱۶ بیت برای خواندن یا نوشتن وجود نداشته باشد، پردازنده به مشکل نخورد.

برای مقداردهی اولیه به حافظه، با استفاده از کد پایتون یک فایل تکست شامل خط ۵۱۲ عدد ۳۲ بیتی hexadecimal می‌کنیم و این فایل را به عنوان ورودی به حافظه می‌دهیم.

در نهایت با توجه به opcode ورودی مازول، در صورتی که opcode برابر با ۰۰ باشد عملیات load از حافظه و در صورتی که opcode برابر با ۰۱ باشد عملیات store در حافظه انجام می‌شود.

برای عملیات load در هر مرحله یک خانه ۳۲ بیتی از حافظه خوانده می‌شود و در یک بخش ۳۲ بیتی از خروجی ۵۱۲ بیتی قرار داده می‌شود و این کار ۱۶ بار تکرار می‌شود. برای store هم به طریق مشابه در هر مرحله یک بخش ۳۲ بیتی از ورودی ۵۱۲ بیتی در یک خانه حافظه ذخیره می‌شود.

## Register File

```

module RegisterFile (clk, rst, opcode, reg_sel, data_in_1, data_in_2,
                     data_out_1, data_out_2, R1, R2, R3, R4);

    input clk, rst;
    input [1:0] opcode;
    input [1:0] reg_sel;
    input [511:0] data_in_1, data_in_2;
    output reg [511:0] data_out_1, data_out_2, R1, R2, R3, R4;

    initial begin
        R1[511:0] = 10;
        R2[511:0] = 20;
        R3[511:0] = 30;
        R4[511:0] = 40;
    end

```

در ابتدای این مأژول ابتدا ورودی‌ها و خروجی‌ها را تعریف می‌کنیم. ورودی شامل سیگنال‌های کلک و ریست،  
برای تعیین عملکرد، reg\_sel برای این که مشخص شود عملیات روی کدام رجیستر انجام شود و دو ورودی ۵۱۲ بیتی  
برای انجام عملیات load در رجیستر است. خروجی شامل ۴ رجیستر ۱۲ بیتی است که در واقع همان رجیسترهاي  
رجیسترفایل هستند به همراه دو خروجی data\_out که حاصل عملیات read از رجیستر در این خروجی‌ها ذخیره می‌شود.

```

always @(posedge clk or posedge rst) begin
    if (rst) begin
        R1 <= 0;
        R2 <= 0;
        R3 <= 0;
        R4 <= 0;
        data_out_1 <= 0;
        data_out_2 <= 0;
    end
    else begin
        case (opcode)
            2'b00:
                case (reg_sel)
                    2'b00: #5 R1 = data_in_1;
                    2'b01: #5 R2 = data_in_1;
                    2'b10: #5 R3 = data_in_1;
                    2'b11: #5 R4 = data_in_1;
                endcase

            2'b01:
                case (reg_sel)
                    2'b00: data_out_1 = R1;
                    2'b01: data_out_1 = R2;
                    2'b10: data_out_1 = R3;
                    2'b11: data_out_1 = R4;
                endcase

            2'b10: begin
                data_out_1 = R1;
                data_out_2 = R2;
                #20;
                R3 = data_in_1;
                R4 = data_in_2;
            end

            2'b11: begin
                data_out_1 = R1;
                data_out_2 = R2;
                #20;
                R3 = data_in_1;
                R4 = data_in_2;
            end
        endcase
    end
end

```

آپکد ۰۰ عملیات load در رجیستر را انجام می‌دهد و در نتیجه آن با توجه  
به یکی از رجیسترهاي این رجیسترفایل ذخیره  
می‌شود.

آپکد ۰۱ عملیات read را در رجیستر انجام می‌دهد و در نتیجه آن با توجه  
به reg\_sel، داده ذخیره شده در یکی از رجیسترها در data\_in\_1 قرار  
می‌گیرد.

آپکد ۱۰ عملیات رجیسترها حین عمل جمع را کنترل می‌کند. در ابتدای مقادیر  
دو رجیستر R1 و R2 در خروجی‌های data\_out\_1 و data\_out\_2 قرار  
می‌گیرند تا به عنوان ورودی به مأژول ALU داده شوند. سپس ۲۰ واحد  
زمانی برای اتمام عمل جمع در واحد ALU در نظر گرفته می‌شود و جواب‌های  
ALU در data\_in\_2 و data\_in\_1 قرار می‌گیرند که به عنوان ورودی به  
این مأژول می‌آیند. در نهایت این دو مقدار در رجیسترهاي R3 و R4 ذخیره  
می‌شوند.

آپکد ۱۱ عملیات رجیسترها حین ضرب را کنترل می‌کند که مشابه قسمت  
جمع است.

## ماژول نهایی پردازنده

```

`include "RegisterFile.v"
`include "ALU.v"
`include "Memory.v"

module Processor (clk, rst, instruction, R1, R2, R3, R4);

input clk, rst;
input [12:0] instruction;
output signed [511:0] R1, R2, R3, R4;

wire signed [511:0] data_in_1, data_in_2, data_out_1, data_out_2;

wire [1:0] opcode = instruction[12:11];
wire [1:0] reg_sel = instruction[10:9];
wire [8:0] mem_addr = instruction[8:0];

RegisterFile reg_file1 (clk, rst, opcode, reg_sel, data_in_1, data_in_2, data_out_1, data_out_2, R1, R2, R3, R4);
ALU ALU1 (clk, opcode, data_out_1, data_out_2, data_in_1, data_in_2);
Memory mem1 (clk, opcode, data_out_1, mem_addr, data_in_1);

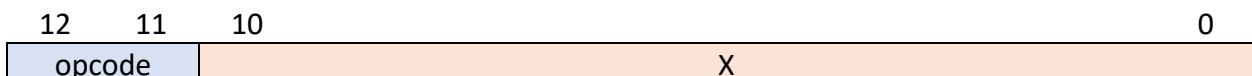
endmodule

```

ماژول پردازنده شامل ورودی‌های کلک و ریست و یک ورودی دستور و چهار خروجی شامل مقادیر رجیسترهاي رجیسترفايل می باشد.

در این پردازنده، دو نوع دستور داریم:

- (۱) نوع اول شامل دستورهای load و store بین حافظه و رجیسترفايل است.
- (۲) نوع دوم شامل دستورات جمع و ضرب آرایه‌ای است.



جدول ها به صورت زیر است:

opcode	instruction
00	load from memory in specified register
01	store specified register in memory
10	add R1 and R2, store low part in R3 and high part in R4
11	multiply R1 and R2, store low part in R3 and high part in R4

## Test Bench

در مازول آزمون task‌هایی تعریف کردہ‌ایم که دستورات مختلف را پیاده‌سازی کرده‌اند. سپس از دستورات مختلف برای تست مازول استفاده می‌کنیم.

```

task reset_processor;
begin
    rst = 1;
    #10; // Wait for reset to complete
    rst = 0;
    $display("After reset: \nR1 = %h, \nR2 = %h, \nR3 = %h, \nR4 = %h\n", R1, R2, R3, R4);
end
endtask

task load_memory_to_register (input [8:0] addr, input [1:0] reg_num);
begin
    instruction[12:11] = 2'b00;
    instruction[10:9] = reg_num;
    instruction[8:0] = addr;
    #100; // Wait for read to complete
    $display("After load: \nR1 = %h, \nR2 = %h, \nR3 = %h, \nR4 = %h\n", R1, R2, R3, R4);
end
endtask

task store_register_in_memory (input [8:0] addr, input [1:0] reg_num);
begin
    instruction[12:11] = 2'b01;
    instruction[10:9] = reg_num;
    instruction[8:0] = addr;
    #100; // Wait for write to complete
    $display("\nStore R%0h in memory\n", reg_num+1);
end
endtask

task alu_add();
begin
    instruction[12:11] = 2'b10;
    instruction[10:0] = 11'bZ;
    #100; // Wait for ALU addition to complete
    $display("ALU add result: \nR1 = %h, \nR2 = %h, \nR3 = %h, \nR4 = %h\n", R1, R2, R3, R4);
end
endtask

task alu_multiply();
begin
    instruction[12:11] = 2'b11;
    instruction[10:0] = 11'bZ;
    #100; // Wait for ALU multiplication to complete
    $display("ALU multiplication result: \nR1 = %h, \nR2 = %h, \nR3 = %h, \nR4 = %h\n", R1, R2, R3, R4);
end
endtask

```

یک نمونه Test Bench می‌تواند به صورت زیر باشد:

```
module Processor_TB();

reg clk = 0, rst = 0;
reg [12:0] instruction;
wire signed [511:0] R1;
wire signed [511:0] R2;
wire signed [511:0] R3;
wire signed [511:0] R4;

Processor Processor1 (clk, rst, instruction, R1, R2, R3, R4);

always
#5 clk <= ~clk;

initial begin
    load_memory_to_register (9'b000010100, 2'b00);
    load_memory_to_register (9'b000010101, 2'b10);
    alu_multiply();
    alu_add();
    reset_processor();
    store_register_in_memory(9'b000010011, 2'b11);
    load_memory_to_register(9'b101010101, 2'b01);
    load_memory_to_register(9'b101000001, 2'b00);
    alu_add();
    alu_multiply();
    store_register_in_memory(9'b110000011, 2'b10);
    store_register_in_memory(9'b001100110, 2'b11);
    reset_processor();
    load_memory_to_register (9'b000010101, 2'b00);
    $finish;
end
```

که عملیات مشخص شده در بخش initial از میان task‌های تعریف شده انتخاب شده‌اند.

برای تست ماژول پردازنده، یک بار حالات مرزی را تست می‌کنیم و یک بار عملکرد پردازنده را به صورت رندوم بررسی می‌کنیم.

### تست حالات مرزی

```

initial begin
    reset_processor();

    load_memory_to_register(9'b0000000000, 2'b00);    // biggest negative number
    load_memory_to_register(9'b0000000000, 2'b01);    // biggest negative number
    alu_add();
    alu_multiply();

    load_memory_to_register(9'b0000000000, 2'b00);    // biggest negative number
    load_memory_to_register(9'b0001000000, 2'b01);    // biggest positive number
    alu_add();
    alu_multiply();

    load_memory_to_register(9'b0000000000, 2'b00);    // biggest negative number
    load_memory_to_register(9'b0000100000, 2'b01);    // -1
    alu_add();
    alu_multiply();

    load_memory_to_register(9'b0001000000, 2'b00);    // biggest positive number
    load_memory_to_register(9'b0001000000, 2'b01);    // biggest positive number
    alu_add();
    alu_multiply();

    load_memory_to_register(9'b0001000000, 2'b00);    // biggest positive number
    load_memory_to_register(9'b0000100000, 2'b01);    // -1
    alu_add();
    alu_multiply();

    load_memory_to_register(9'b0000100000, 2'b00);    // -1
    load_memory_to_register(9'b0000100000, 2'b01);    // -1
    alu_add();
    alu_multiply();

    $finish;
end

```

1	80000000
2	80000000
3	80000000
4	80000000
5	80000000
6	80000000
7	80000000
8	80000000
9	80000000
10	80000000
11	80000000
12	80000000
13	80000000
14	80000000
15	80000000
16	80000000
17	FFFFFFF
18	FFFFFFF
19	FFFFFFF
20	FFFFFFF
21	FFFFFFF
22	FFFFFFF
23	FFFFFFF
24	FFFFFFF
25	FFFFFFF
26	FFFFFFF
27	FFFFFFF
28	FFFFFFF
29	FFFFFFF
30	FFFFFFF
31	FFFFFFF
32	FFFFFFF
33	7FFFFFF
34	7FFFFFF
35	7FFFFFF
36	7FFFFFF
37	7FFFFFF
38	7FFFFFF
39	7FFFFFF
40	7FFFFFF
41	7FFFFFF
42	7FFFFFF
43	7FFFFFF
44	7FFFFFF
45	7FFFFFF
46	7FFFFFF
47	7FFFFFF
48	7FFFFFF

نتیجه تست حالات مرزی

After load:

After load:

ALU add result:

### ALU multiplication result:

After load:

After load:

ALU add result:

All multiplication result:

After load:

After load:

R4 = TTTTTTTTTT

R4 = |||||||

ALU multiplication result:  
R1 = fffffffffffffffffff...  
R2 = fffffffffffffffffff...

## تست حالات رندوم

```

initial begin
    reset_processor();
    load_memory_to_register (9'b000010100, 2'b00);
    load_memory_to_register (9'b000010101, 2'b10);
    alu_multiply();
    alu_add();
    reset_processor();
    store_register_in_memory(9'b000010011, 2'b11);
    load_memory_to_register(9'b101010101, 2'b01);
    load_memory_to_register(9'b101000001, 2'b00);
    alu_add();
    alu_multiply();
    store_register_in_memory(9'b110000011, 2'b10);
    store_register_in_memory(9'b001100110, 2'b11);
    reset_processor();
    load_memory_to_register (9'b000010101, 2'b00);
    $finish;
end

```

1	CE6E2560
2	A38C9CE1
3	6B367A8A
4	D4EBFA59
5	1C9FD9A5
6	7828F0D3
7	9268BFF7
8	BD99009F
9	743B1EA7
10	5C30B29A
11	BC78624A
12	5E26C659
13	BA4AC310
14	211651EE
15	23E8B3D0
16	46D472E2
17	CABE2EE9
18	C65CC603
19	D82762B8
20	4BFC86E1
21	E1DBC933
22	9060368F
23	C8D53B2C
24	8D0BC881
25	DC5F9026
26	100340EB
27	AECF43BC
28	571B16FF
29	AFF6C780
30	CE4769AE
31	62DC97FA
32	995D6974
33	1F1F3649
34	E2806F55
35	F844E227
36	1E1C8489
37	C03EC50C
38	07B4299C
39	A73C010C
40	5A937010

نتیجه تست حالات رندهم