
# گزارش آزمایش دهم طراحی سیستم‌های دیجیتال

## گروه شش

اعضا:

احمد سلیمی

همیلا میلی

درنا دهقانی

# شرح آزمایش

در این آزمایش می‌خواهیم یک پردازنده ساده با معماری پشته‌ای طراحی کنیم. طبق فرضیاتی که در دستور کار مطرح شده، این پردازنده یک پشته با ۸ ثبات ۸ بیتی و حافظه‌ای با ظرفیت ۲۵۶ خانه‌ی ۸ بیتی دارد که ۸ خانه‌ی آخر آن (از F8 تا FF) برای I/O به صورت Memory Mapped I/O مورد استفاده قرار می‌گیرند.

این پردازنده ۸ دستور دارد که به شرح زیرند:

| | | |
|---|---|---|
| مقدار ۸ بیتی ثابت C را در پشته PUSH می‌کند. | **0000 C** | **PUSHC** |
| مقدار خانه حافظه یا درگاه که آدرس M دارد را در پشته PUSH می‌کند. | **0001 M** | **PUSH** |
| از پشته POP کرده و مقدار را در آدرس یا درگاه M قرار می‌دهد. | **0010 M** | **POP** |
| از پشته POP کرده و مقدار را در PC قرار می‌دهد. | **0011** | **JUMP** |
| اگر پرچم Z ۱ باشد، از پشته POP کرده و مقدار را در PC قرار می‌دهد. | **0100** | **JZ** |
| اگر پرچم S ۱ باشد، از پشته POP کرده و مقدار را در PC قرار می‌دهد. | **0101** | **JS** |
| دو مقدار بالای پشته را POP کرده، آنها را جمع می‌کند و حاصل را در پشته PUSH می‌کند. | **0110** | **ADD** |
| دو مقدار بالای پشته را POP کرده، آنها را تفریق می‌کند و حاصل را در پشته PUSH می‌کند. | **0111** | **SUB** |

طبق دستورات فوق مشخص است که دو پرچم S (نشان‌دهنده‌ی منفی بودن حاصل آخرین جمع یا تفریق) و Z (نشان‌دهنده‌ی صفر بودن حاصل آخرین جمع یا تفریق) داریم که تنها با دستورات ADD و SUB می‌توان مقدارشان را تغییر داد. ضمناً تمامی محاسبات علامت‌دار و با مکمل ۲ انجام می‌شوند.

در نهایت می‌خواهیم مقدار X را از ورودی بخوانیم و حاصل زیر را توسط پردازنده فوق محاسبه کنیم:

$$Y = \big((X + 23) * 2\big) - 12$$

## Stack_machine.v

این ماژول، تنها ماژول آزمایش بوده و در آن به پیاده‌سازی پردازنده می‌پردازیم.

ورودی‌ها:

- Clk
- rstN
- in: مقدار ورودی ۸ بیتی.

خروجی:

- Out: مقدار خروجی ۸ بیتی.

ابتدا به مقداردهی‌های ابتدایی و بخش ترکیبی مدار می‌پردازیم.

برای نمایش حافظه (data_mem) از یک آرایه ۲۵۶تایی که هر خانه‌ی آن ۸ بیت است استفاده می‌کنیم. برای نمایش حافظه‌ی دستورات (inst_mem) از یک آرایه ۳۲تایی که هر خانه‌ی آن ۱۲ بیت است (حداکثر طول دستورات برابر ۱۲ = ۸ + ۴ بیت است) استفاده می‌کنیم. برای نمایش پشته (stack) از یک آرایه ۸تایی که هر خانه‌ی آن ۸ بیت است استفاده می‌کنیم. با توجه به اینکه حافظه‌ی دستورات، ۳۲ خانه دارد، از pc که یک رجیستر ۵ بیتی است برای مشخص کردن دستوری که در آن قرار داریم، استفاده می‌کنیم. برای مشخص کردن سر پشته، از رجیستر ۳ بیتی sp استفاده می‌کنیم.

مقادیر مفروض برای opcode هر دستور را توسط parameter مشخص می‌کنیم.

برای هر ورودی و خروجی in، out خانه‌ای از حافظه را اختصاص می‌دهیم. با این کار در واقع I/O ها به خانه‌هایی از حافظه map می‌شوند که همان طراحی memory mapped I/O می‌باشد.

بیت‌های اول تا چهارم دستوری که در آن قرار داریم، به عنوان opcode آن دستور مشخص می‌شوند (inst_op). بیت‌های پنجم تا دوازدهم هر دستور در inst_value قرار می‌گیرد که مهم بودن یا نبودن این مقدار بستگی به دستور دارد.

مقدار موجود در خانه‌ی out_addr از حافظه در خروجی قرار می‌گیرد.

حاصل جمع و تفریق دو خانه‌ی آخر پشته، محاسبه شده و در add_result و sub_result قرار می‌گیرند و در دستور مناسب، push خواهند شد.

کد وریلاگ این بخش از آزمایش به شرح زیر است:

```
module stack_machine (
    input           clk,
    input           rstN,
    input   [7:0]   in,
    output  [7:0]   out
);

reg     [7:0]   data_mem    [255:0];
reg     [7:0]   stack       [7:0];
reg     [1:12]  inst_mem    [31:0];

reg     [4:0]   pc;
reg     [2:0]   sp;

reg             s_flag = 0, z_flag = 0;
wire    [3:0]   inst_op;
wire    [7:0]   inst_value;
wire    [7:0]   add_result, sub_result;


parameter op_pushc      = 0;
parameter op_pushmem    = 1;
parameter op_pop        = 2;
parameter op_j          = 3;
parameter op_jz         = 4;
parameter op_js         = 5;
parameter op_add        = 6;
parameter op_sub        = 7;

parameter in_addr       = 254;
parameter out_addr      = 255;

assign inst_op          = inst_mem[pc][1:4];
assign inst_value       = inst_mem[pc][5:12];
assign out              = data_mem[out_addr];
assign add_result       = stack[sp - 2] + stack[sp - 1];
assign sub_result       = stack[sp - 2] - stack[sp - 1];


```

حال به پیاده‌سازی بخش ترتیبی مدار می‌پردازیم. از یک always block استفاده می‌کنیم که به لبه‌ی بالارونده کلاک و لبه‌ی پایین‌رونده ریست حساس است.

در صورت ۰ شدن rstN، مقادیر pc، sp، s_flag، z_flag و تمام خانه‌های پشته برابر با صفر می‌شوند.

در غیر این صورت و با بالا رفتن لبه‌ی کلاک، مقدار ورودی در خانه‌ی in_addr از حافظه قرار می‌گیرد.

سپس با توجه به مقدار inst_op عملیاتی انجام می‌شود:

- ۰ – pushc: مقدار inst_value در بالاترین خانه‌ی پشته push شده و اشاره‌گر به بالای پشته یکی افزایش می‌یابد.

- ۱ ـ pushmem: مقدار موجود در خانه‌ی inst_value از حافظه در بالاترین خانه‌ی پشته push شده و اشاره‌گر به بالای پشته یکی افزایش می‌یابد.
- ۲ – pop: مقدار موجود در بالاترین خانه‌ی پشته در خانه‌ی inst_value از حافظه قرار گرفته و اشاره‌گر به بالای پشته یکی کاهش می‌یابد.
- ۳ – j: مقدار آخرین خانه‌ی پشته در pc قرار گرفته و اشاره‌گر به بالای پشته یکی کاهش می‌یابد.
- ۴ – jz: در صورت یک بودن مقدار z_flag، مقدار آخرین خانه‌ی پشته در pc قرار گرفته و اشاره‌گر به بالای پشته یکی کاهش می‌یابد.
- ۵ – js: در صورت یک بودن مقدار s_flag، مقدار آخرین خانه‌ی پشته در pc قرار گرفته و اشاره‌گر به بالای پشته یکی کاهش می‌یابد.
- ۶ – add: مقدار جمع دو خانه‌ی آخر پشته در بالاترین خانه‌ی پشته (پس از pop شدن دو خانه‌ی آخر) قرار گرفته و اشاره‌گر به بالای پشته یکی کاهش می‌یابد. مقدار z_flag برابر با NOR بیت‌های حاصل جمع می‌شود و اگر این حاصل صفر باشد، z_flag برابر با ۱ و در غیر این صورت برابر با صفر می‌شود. مقدار s_flag در صورت منفی بودن مقدار حاصل جمع برابر با ۱ و در غیر این صورت ۰ می‌شود.
- ۷ – sub: مقدار تفریق دو خانه‌ی آخر پشته در بالاترین خانه‌ی پشته (پس از pop شدن دو خانه‌ی آخر) قرار گرفته و اشاره‌گر به بالای پشته یکی کاهش می‌یابد. مقدار z_flag برابر با NOR بیت‌های حاصل تفریق می‌شود و اگر این حاصل صفر باشد، z_flag برابر با ۱ و در غیر این صورت برابر با صفر می‌شود. مقدار s_flag در صورت منفی بودن مقدار حاصل تفریق برابر با ۱ و در غیر این صورت ۰ می‌شود.

کد وریلاگ این بخش از مدار به شرح زیر است:

```
integer i;
always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        pc                              <= 0;
        sp                              <= 0;
        s_flag                          <= 0;
        z_flag                          <= 0;
        for (i = 0; i < 8; i = i + 1)
            stack[i]                    <= 0;
    end
    else begin
        data_mem[in_addr]               <= in;

        pc                              <= pc + 1;
        case (inst_op)
            /* PUSH CONSTANT */
            op_pushc: begin
                stack[sp]               <= inst_value;
                sp                      <= sp + 1;
            end

            /* PUSH MEMORY */
            op_pushmem: begin
                stack[sp]               <= data_mem[inst_value];
                sp                      <= sp + 1;
            end

            /* POP */
            op_pop: begin
                data_mem[inst_value]    <= stack[sp - 1];
                sp                      <= sp - 1;
            end

            /* JUMP */
            op_j: begin
                pc                      <= stack[sp - 1];
                sp                      <= sp - 1;
            end
```

```
            /* JUMP Z */
            op_jz: if (z_flag) begin
                pc                      <= stack[sp - 1];
                sp                      <= sp - 1;
            end

            /* JUMP S */
            op_js: if (s_flag) begin
                pc                      <= stack[sp - 1];
                sp                      <= sp - 1;
            end

            /* ADD */
            op_add: begin
                stack[sp - 2]           <= add_result;
                stack[sp - 1]           <= 0;
                sp                      <= sp - 1;
                z_flag                  <= ~|add_result;
                s_flag                  <= $signed(add_result) < 0;
            end

            /* SUB */
            op_sub: begin
                stack[sp - 2]           <= sub_result;
                stack[sp - 1]           <= 0;
                sp                      <= sp - 1;
                z_flag                  <= ~|sub_result;
                s_flag                  <= $signed(sub_result) < 0;
            end
        endcase
    end
end

endmodule
```

# Formula.v

در این ماژول به محاسبه مقدار خواسته‌شده در دستور کار می‌پردازیم.

برای مشخص کردن دستورات، برای سهولت کار از یک ماکرو کمک گرفتیم که مقادیر addr و opcode و value را به عنوان ورودی گرفته، مقدار opcode را ۸ بیت (طول value) به چپ شیفت میدهد و آن را با OR value می‌کند. با این کار دستور مورد نظر ساخته می‌شود و فقط کافیست در خانه‌ی addr از حافظه‌ی دستورات قرار گیرد.

ابتدا از ماژول stack_machine یک شیء با ورودی‌ها و خروجی مورد نظر می‌سازیم. مقدار error در صورت وجود خطا ۱ می‌شود. در صورت بیشتر بودن خروجی out از ۱۲۷، چون طبق دستورکار از حوزه قابل نمایش خارج است و یا در صورت منفی بودن ورودی (۱ بودن پرارزش‌ترین بیت in)، خطا رخ می‌دهد.

در این test bench، یک loop داریم که ۳ بار تکرار می‌شود. در هر بار یک ورودی می‌گیرد و مقدار y را برای آن محاسبه می‌کند و خروجی می‌دهد. در صورت اتمام این loop، از آن خارج شده و برنامه به اتمام می‌رسد.

آدرس ۰ از حافظه را به tmp اختصاص می‌دهیم که متغیر کمکی برای انجام محاسبات است. آدرس ۷ از حافظه را به counter اختصاص می‌دهیم که تعداد دفعاتی‌ست که می‌خواهیم ورودی بگیریم. برای پایان کد از سطر ۲۵ دستورات که عملاً خالیست استفاده می‌شود.

در یک initial block دستورات لازم برای ۳ بار گرفتن ورودی و انجام محاسبات لازم برای هر یک از ورودی‌ها وارد می‌شود. سپس با ۳ ورودی، کارایی ماژول را بررسی کرده و آنهارا مانیتور می‌کنیم. کد این بخش از آزمایش به شرح زیر است:

```
1  `define inst(ADDR, OP, VAL=0)    cpu.inst_mem[ADDR] = (OP << 8) | VAL
2
3  module formula;
4
5  reg            clk = 1, rstN = 0;
6  reg     [7:0]  in;
7  wire    [7:0]  out;
8
9  stack_machine cpu(clk, rstN, in, out);
10
11 assign error = (out > 127) || (in[7] == 1);
12 always #10 clk = ~clk;
13
14 /* Pointers to important memories */
15 localparam counter_p    = 7;
16 localparam tmp_p        = 0;
17
18 /* Address of exit program */
19 localparam exit         = 25;
20
21 initial begin
22     /* THE CODE */
23
24     /* counter = 3 */
25     `inst(0,    cpu.op_pushc,    3);            // s_head  = 3
26     `inst(1,    cpu.op_pop,      counter_p);    // counter = 3
27
28     /* LOOP CONDITION */
29
30     /* counter = counter - 1 */
31     `inst(2,    cpu.op_pushmem, counter_p);     // s_head  = counter
32     `inst(3,    cpu.op_pushc,   1);             // s_head  = 1
33     `inst(4,    cpu.op_sub);                    // s_head  = counter - 1
34     `inst(5,    cpu.op_pop,     counter_p);     // counter = counter - 1
35
36     /* if (counter < 0): goto exit */
37     `inst(6,    cpu.op_pushc,   exit);          // s_head  = exit
38     `inst(7,    cpu.op_js);                     // if (counter == 0): goto exit
39     `inst(8,    cpu.op_pop,     tmp_p);         // else: sp = sp - 1
40
```

```
    /* LOOP BODY */
    `inst(9,    cpu.op_pushmem, cpu.in_addr);   // s_head  = x
    `inst(10,   cpu.op_pushc,   23);            // s_head  = 23
    `inst(11,   cpu.op_add);                    // s_head  = x + 23
    `inst(12,   cpu.op_pop,     tmp_p);         // tmp     = x + 23
    `inst(13,   cpu.op_pushmem, tmp_p);         // s_head  = x + 23
    `inst(14,   cpu.op_pushmem, tmp_p);         // s_head  = x + 23
    `inst(15,   cpu.op_add);                    // s_head  = (x + 23) * 2
    `inst(16,   cpu.op_pushc,   12);            // s_head  = 12
    `inst(17,   cpu.op_sub);                    // s_head  = ((X + 23) * 2) - 12
    `inst(18,   cpu.op_pop,     cpu.out_addr);  // y       = ((X + 23) * 2) - 12

    /* jump to start of loop */
    `inst(19,   cpu.op_pushc,   2);             // s_head  = 2
    `inst(20,   cpu.op_j);                      // goto    2
end

initial begin

    #10 rstN = 1;
    wait(cpu.pc == 7);
    in = 13;
    wait(cpu.pc == 19);
    $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(in), $signed(out), error);

    wait(cpu.pc == 7);
    in = 1;
    wait(cpu.pc == 19);
    $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(in), $signed(out), error);

    wait(cpu.pc == 7);
    in = -121;
    wait(cpu.pc == 19);
    $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(in), $signed(out), error);

    wait(cpu.pc == exit);
    $stop;
end

endmodule
```

با شبیه‌سازی این ماژول توسط modelsim می‌توان نتایج زیر را مشاهده کرد:

```
VSIM 11> run 5000
# ((  13 + 23) * 2) - 12 =   60, error = 0
# ((   1 + 23) * 2) - 12 =   36, error = 0
# ((-121 + 23) * 2) - 12 =   48, error = 1
```