به نام خداوند بخشنده مهربان

آز سیستم های دیجیتال

دکتر اجلالی

مبینا حیدری، عاطفه قندهاری، نیکا قادری
بهار ۱۴۰۳

تاریخ گزارش: ۱۳ خرداد ۱۴۰۳ | **پیاده سازی یک پردازنده** | آزمایش دهم

۱ مقدمه و شرح آزمایش

در این آزمایش می خواهیم یک پردازنده ساده طراحی کنیم.
این پردازنده یک پشته با ۸ ثبات ۸ بیتی دارد و همچنین دارای حافظه ای به ظرفیت ۲۵۶ خانه ۸ بیتی می باشد. البته ۸ خانه آخر آن به ورودی/خروجی اختصاص داده شده اند. از آنجایی که بخشی از حافظه به I/O اختصاص داده شده می توان گفت این یک سیستم $Memory\ Mapped\ I/O$ می‌باشد. همچنین این پردازنده دارای هشت دستور است که به شرح زیر می باشند:

0000 PUSHC C

این دستور مقدار ثابت (Constant) ۸ بیتی C را در پشته PUSH می‌کند.

0001 PUSH M

این دستور مقدار خانه حافظه (یا درگاه) که با آدرس M (آدرس ۸ بیتی) مشخص شده است را خوانده و در پشته PUSH می‌کند.

0010 POP M

مقدار را از پشته POP کرده و آن را در خانه حافظه با آدرس M قرار می‌دهد (یا به درگاه با آدرس M ارسال می‌کند).

0011 JUMP

از پشته POP کرده و در PC قرار می‌دهد.

0100 JZ

اگر پرچم Z برابر 1 باشد از پشته POP کرده و در PC قرار می‌دهد.

0101 JS

اگر پرچم S برابر 1 باشد از پشته POP کرده و در PC قرار می‌دهد.

0110 ADD

دو داده بالای پشته را POP کرده با هم جمع کرده و حاصل را در بالای پشته PUSH می‌کند.

0111 SUB

دو داده بالای پشته را POP کرده عمل تفریق را برروی آن‌ها انجام می‌دهد و حاصل را در بالای پشته PUSH می‌کند.

شکل ۱: فهرست دستورات

همچنین دو فلگ S و Z داریم که تنها با دستورات add و sub می توان مقدارشان را تغییر داد. فلگ S مشخص می کند حاصل آخرین جمع یا تفریق مثبت بوده است یا منفی. فلگ Z هم مشخص می کند حاصل آخرین عملیات حسابی صفر بوده است یا خیر.
در آخر نیز می خواهیم این حاصل را با استفاده از عدد ورودی به دست بیاوریم:
$out = ((in + ۲۳) * ۲) - ۱۲$

۲ ماژول ها و توضیحات

۱.۲ $Stack\_machine$

۱.۱.۲ هدف

این ماژول، ماژول اصلی آزمایش است و عملکرد یک پردازنده را پیاده سازی می کند.

۲.۱.۲ ورودی ها

۱. Clk: سیگنال ساعت.

۲. $rstN$: سیگنال ریست، که با صفر شدن فعال می شود.

۳. in: ورودی هشت بیتی.

۳.۱.۲ خروجی ها

۱. out: خروجی هشت بیتی.

۴.۱.۲ توضیحات

برای نمایش حافظه، از یک ترکیب آرایه و بردار به نام $data\_mem$ استفاده می کنیم که اندازه آن ۲۵۶ خانه است و هر خانه شامل ۸ بیت می باشد.
همچنین برای نمایش حافظه دستورات (این دو جدا در نظر گرفته شده اند) از یک آرایه دیگر به نام $inst\_mem$ استفاده می کنیم که ۳۲ خانه دارد و هر خانه آن شامل دوازده بیت می باشد، چرا که فرض می کنیم دستورات ما دوازده بیتی می باشند.
برای نمایش پشته، از یک آرایه هشت تایی که هر خانه ی آن هشت بیت است استفاده می کنیم. همچنین برای این که مشخص شود در حال حاظر در کدام دستور هستیم از یک pc استفاده می کنیم، از آنجایی که تعداد دستورات سی و دو می باشد، کافی است شمارنده دستورات پنج بیت داشته باشد. برای مشخص کردن سر پشته نیز از یک sp به طول سه بیت استفاده می کنیم. (از آنجائیکه طول پشته هشت می باشد.)
$opcode$ های دستورات مقادیر مختلفی می توانند داشته باشند. جهت خوانایی بیشتر، آن ها را در پارامترها ذخیره می کنیم تا دسترسی به آن ها برای مشخص کردن نوع دستور راحت تر باشد. بیت های اول تا چهارم هر دستور، همین $opcode$ می باشند که تحت عنوان $inst\_op$ از بقیه قسمت های دستور، جدا می شوند. بیت های پنجم تا دوازدهم دستور در $inst\_value$ ذخیره شده که بعدا پردازش می شود.
حاصل جمع و تفریق دو خانه آخر پشته، محاسبه شده و در add_result و sub_result قرار می گیرند. همچنین، خروجی این ماژول در واقع مقداری می باشد که در خانه out_addr ذخیره شده است.
پیاده سازی توضیحات بالا به صورت زیر می باشد:

```
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];
```

شکل ۲: ماژول $stack\_machine$

در قسمت بعدی، وارد بخش دوم این ماژول می شویم که مداری ترتیبی را پیاده سازی می کند. محتویات این بخش درون یک بلاک $always$ قرار دارند که به لبه بالاورنده سیگنال ساعت و لبه پایین رونده سیگنال ریست حساس است. ابتدا در صورت صفر شدن سیگنال ریست، تمام خانه های حافظه به همراه pc، sp، s_flag، z_flag صفر می شوند. در غیر این صورت، سیگنال ورودی در خانه in_addr قرار می گیرد. سپس مقدار $inst\_op$ بررسی می شود و با توجه به آن ادامه برنامه اجرا می شود:

۱. $pushc$: مقدار $inst\_value$ در بالاترین خانه حافظه پوش می شود و sp یک واحد افزایش می یابد.

۲. $pushmem$: ابتدا به آدرس موجود در $inst\_value$ درون حافظه رفته و مقدار آن را می خوانیم. سپس این مقدار را در پشته پوش می کنیم و اشاره گر پشته را مثل حالت قبل، یک واحد افزایش می دهیم.

۳. pop: مقدار موجود در بالاترین خانه پشته در $inst\_value$ قرار می گیرد و اشاره گر پشته یک واحد کاهش می یابد.

۴. j: مقدار آخرین خانه پشته در pc قرار می گیرد و اشاره گر پشته یک واحد کاهش می یابد.

۵. jz: ابتدا فلگ z را چک می کنیم. اگر یک باشد، مانند حالت j عمل می شود.

۶. js: اگر فلگ s (s_flag) یک باشد، مانند حالت j عمل می شود.

۷. add: دو خانه آخر پاپ می شود، جمع آنها در استک پوش می شود، و اشاره گر در کل یک واحد کاهش می یابد. مقدار z_flag برابر با عکس nor بیت های حاصل می باشد. یعنی اگر حاصل صفر شود، این فلگ باید یک شود. مقدار s_flag هم مشخص می کند حاصل منفی است یا مثبت.

۸. sub: عینا مانند حالت جمع می باشد، با این تفاوت که به جای جمع، تفریق انجام می شود.

در نهایت، کد وریلاگ این بخش نیز به صورت زیر نوشته می شود:

```
40  integer i;
41  always @(posedge clk or negedge rstN) begin
42      if (~rstN) begin
43          pc                          <= 0;
44          sp                          <= 0;
45          s_flag                      <= 0;
46          z_flag                      <= 0;
47          for (i = 0; i < 8; i = i + 1)
48              stack[i]                <= 0;
49      end
50      else begin
51          data_mem[in_addr]           <= in;
52
53          pc                          <= pc + 1;
54          case (inst_op)
55              /* PUSH CONSTANT */
56              op_pushc: begin
57                  stack[sp]           <= inst_value;
58                  sp                  <= sp + 1;
59              end
60
61              /* PUSH MEMORY */
62              op_pushmem: begin
63                  stack[sp]           <= data_mem[inst_value];
64                  sp                  <= sp + 1;
65              end
66
67              /* POP */
68              op_pop: begin
69                  data_mem[inst_value]    <= stack[sp - 1];
70                  sp                  <= sp - 1;
71              end
72
73              /* JUMP */
74              op_j: begin
75                  pc                  <= stack[sp - 1];
76                  sp                  <= sp - 1;
```

```
75                  pc                  <= stack[sp - 1];
76                  sp                  <= sp - 1;
77              end
78
79
80              /* JUMP Z */
81              op_jz: if (z_flag) begin
82                  pc                  <= stack[sp - 1];
83                  sp                  <= sp - 1;
84              end
85
86              /* JUMP S */
87              op_js: if (s_flag) begin
88                  pc                  <= stack[sp - 1];
89                  sp                  <= sp - 1;
90              end
91
92              /* ADD */
93              op_add: begin
94                  stack[sp - 2]       <= add_result;
95                  sp                  <= sp - 1;
96                  z_flag              <= ~|add_result;
97                  s_flag              <= $signed(add_result) < 0;
98              end
99
100             /* SUB */
101             op_sub: begin
102                 stack[sp - 2]       <= sub_result;
103                 sp                  <= sp - 1;
104                 z_flag              <= ~|sub_result;
105                 s_flag              <= $signed(sub_result) < 0;
106             end
107         endcase
108     end
109 end
110
111 endmodule
```

۲.۲ $formula$

۱.۲.۲ هدف

در این ماژول، با استفاده از پردازنده ای که در قسمت قبل ساختیم، عبارت ریاضی خواسته شده در صورت مساله را حساب می کنیم و خروجی می دهیم.

۲.۲.۲ توضیحات

ابتدا توجه کنید که با گرفتن مقادیر جدا برای $opcode$ و $value$، دستور را به راحتی با شیفت دادن و ترکیب کردن دو ورودی ساخت. (جهت افزایش خوانایی کد) حال یک از ماژول قبلی یک نمونه می گیریم. بیت اروری هم قرار می دهیم که در صورت بروز خطا، یک شود. برای مقال وقتی که ورودی in منفی باشد؛ یعنی پرارزش ترین بیت آن برابر با یک باشد. یا وقتی که خروجی از ۱۲۷ بیشتر باشد.
از آنجایی که در برخی از دستورات احتیاج به مقدار صفر برای بخش $value$ دستور داریم، از یک متغیر کمکی به نام $temp\_p$ کمک می گیریم که

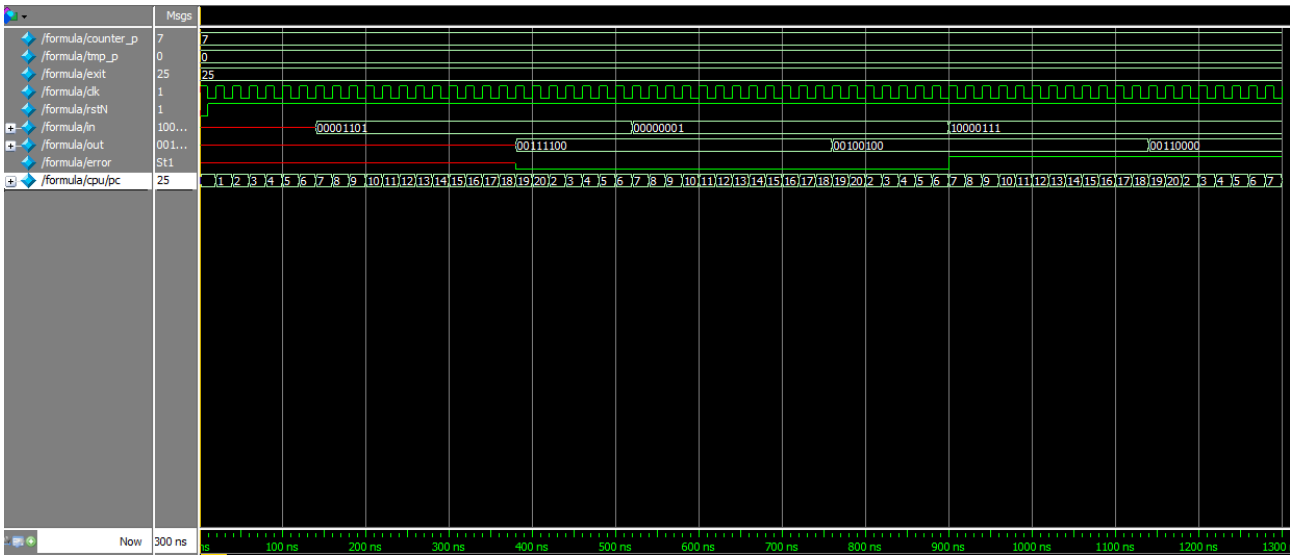
مقدار صفر را در خودش ذخیره می کند. همچنین آدرس هفت از حافظه را به $counter$ اختصاص می دهیم که تعداد دفعاتی است که می خواهیم ورودی بگیریم.
در مرحله بعد، در یک بلاک $initial$، دستورات لازم برای سه بار گرفتن ورودی و انجام محاسبات برای هر یک از ورودی ها وارد می شود. سپس با سه ورودی، کارکرد ماژول را بررسی می کنیم. کد این بخش نیز به صورت زیر می باشد:

```
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;
```

```
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
41      /* LOOP BODY */
42      `inst(9,    cpu.op_pushmem, cpu.in_addr);   // s_head  = x
43      `inst(10,   cpu.op_pushc,   23);            // s_head  = 23
44      `inst(11,   cpu.op_add);                    // s_head  = x + 23
45      `inst(12,   cpu.op_pop,     tmp_p);         // tmp     = x + 23
46      `inst(13,   cpu.op_pushmem, tmp_p);         // s_head  = x + 23
47      `inst(14,   cpu.op_pushmem, tmp_p);         // s_head  = x + 23
48      `inst(15,   cpu.op_add);                    // s_head  = (x + 23) * 2
49      `inst(16,   cpu.op_pushc,   12);            // s_head  = 12
50      `inst(17,   cpu.op_sub);                    // s_head  = ((X + 23) * 2) - 12
51      `inst(18,   cpu.op_pop,     cpu.out_addr);  // y       = ((X + 23) * 2) - 12
52
53      /* jump to start of loop */
54      `inst(19,   cpu.op_pushc,   2);             // s_head  = 2
55      `inst(20,   cpu.op_j);                      // goto    2
56  end
```

```
58  initial begin
59
60      #10 rstN = 1;
61      wait(cpu.pc == 7);
62      in = 13;
63      wait(cpu.pc == 19);
64      $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(in), $signed(out), error);
65
66      wait(cpu.pc == 7);
67      in = 1;
68      wait(cpu.pc == 19);
69      $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(in), $signed(out), error);
70
71      wait(cpu.pc == 7);
72      in = -121;
73      wait(cpu.pc == 19);
74      $display("((%d + 23) * 2) - 12 = %d, error = %b", $signed(in), $signed(out), error);
75
76      wait(cpu.pc == exit);
77      $stop;
78  end
79
80  endmodule
```

پس از اجرای این تست بنچ، به نتایج زیر می رسیم:




```
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
```

پایان