به نام خداوند بخشنده مهربان

به نام خداوند بخشنده مهربان

آز سیستم های دیجیتال

دکتر اجلالی

مبینا حیدری، عاطفه قندهاری، نیکا قادری
بهار ۱۴۰۳

تاریخ گزارش: ۱۰ خرداد ۱۴۰۳ | **اعداد مختلط** Alu | آزمایش هشتم

۱ مقدمه و شرح آزمایش

در این آزمایش می خواهیم یک سیستم محاسبات طراحی کنیم که ضرب و تقسیم اعداد مختلط را پشتیبانی کند. همچنین، بخش خوانش و اجرای دستورات را به صورت پایپلاین انجام می دهیم. در طراحی این حافظه، طول کلمات، ۱۶ بیت و تعداد کلمات (خانه های حافظه) ۳۲ در نظر گرفته شده است. از آنجایی که اعداد دو بخش حقیقی و موهومی دارند، به طور قراردادی ۸ بیت برای بخش حقیقی و ۸ بیت برای بخش موهومی در نظر گرفته شده است. قبل از این که به توضیح کد بپردازیم، لازم به ذکر است که ابعاد کلی برنامه به منظرو دسترسی بهتر و خواناتر شدن کد، در فایلی به نام $macros$ قرار گرفته اند. این فایل در تمامی ماژول ها اینکلود می شود تا بتوان از مقادیر تعریف شده استفاده کرد. محتویات آن نیز به صورت زیر می باشند:

```
1   `define WL          8
2   `define complex     [2*`WL-1:0]
3   `define Re(c)       c[2*`WL-1:`WL]
4   `define Im(c)       c[`WL-1:0]
5   `define sRe(c)      $signed(`Re(c))
6   `define sIm(c)      $signed(`Im(c))
```

شکل ۱: فایل macros

- WL: مقدار برابر با هشت.
- $Re(c)$: تابعی برای یافتن هشت بیت پرارزش تر ورودی که به عنوان بخش حقیقی در نظر گرته شده اند.
- $Im(c)$: تابعی که هشت بیت کم ارزش داده را به ما برمی گرداند که همان بخش موهومی عدد می باشند.
- $sRe(C)$: تابعی که بخش حقیقی اعداد مختلط علامت دار را به صورت $signd$ بر می گرداند.
- $sIm(c)$: تابعی برای برگرداندن قسمت موهومی اعداد مختلط علامت دار.
- $complex$: مشخص کننده ابتدا و انتهای عددی مختلط.

۲ ماژول ها و توضیحات

۱.۲ $Addsub$

۱.۱.۲ هدف

جمع و تفریق اعدا مختلط. به این صورت کار می کند که بخیش های مختلط را با هم، و بخش های حقیقی را با هم جفت می کند و بسته به ورودی کنترلی آن ها را با هم جمع یا تفریق می کند.

۲.۱.۲ ورودی ها

۱. a: عدد اول

۲. b: عدد دوم

۳. op: بیت کنترلی که مشخص می کند دو عدد باید با هم جمع شوند یا تفریق. یک به معنای تفریق و صفر به معنای جمع می باشد.

۳.۱.۲ خروجی ها

۱. s: حاصل نهایی

کد این بخش به صورت زیر می باشد:

```
`include "macros.v"

module addsub (
    input   `complex    a,
    input   `complex    b,
    input               op,     // 0 for addition, 1 for substraction
    output  `complex    s
);

assign `Re(s) = `sRe(a) + (op ? -1 : 1) * `sRe(b);
assign `Im(s) = `sIm(a) + (op ? -1 : 1) * `sIm(b);

endmodule
```

شکل ۲: ماژول جمع/تفریق کننده

Mul ۲.۲

۱.۲.۲ هدف

این ماژول قرار است بتواند اعداد مختلط را در هم ضرب کند. اگر دو عدد $a + bi$ و $c + di$ داشته باشیم، ضرب آن ها به صورت زیر تعریف می شود:

$$a + bi)(c + di) = (ac - bd) + (bc + ad)i$$

۲.۲.۲ ورودی ها

۱. a عدد اول

۲. b عدد دوم

۳.۲.۲ خروجی ها

۱. s حاصل ضرب دو عدد ورودی

در نتیجه این بخش به صورت زیر پیاده سازی می شود:

```
`include "macros.v"

module mul (
    input   `complex    a,
    input   `complex    b,
    output  `complex    s
);

assign `Re(s) = `sRe(a) * `sRe(b) - `sIm(a) * `sIm(b);
assign `Im(s) = `sRe(a) * `sIm(b) + `sIm(a) * `sRe(b);

endmodule
```

شکل ۳: ضرب کننده

Alu ۳.۲

۱.۳.۲ هدف

این واحد عملا مشخص می کند چه عملیاتی باید روی داده انجام بگیرد، که با استفاده از دو بیت ورودی انجام می گیرد. یک بیت مشخص می کند ضرب باید انجام شود یا خیر، بیت بعدی مشخص می کند اگر ضرب انجام نشود باید جمع شود یا تفریق.

۲.۳.۲ ورودی ها

۱. a: عملوند اول

۲. b: عملوند دوم

۳. op: مشخص کننده نوع عملیات

پیاده سازی واحد محاسبات به صورت زیر می باشد:

```verilog
`include "macros.v"

module alu (
    input   `complex    a,
    input   `complex    b,
    input   [1:0]       op,
    output  `complex    s
);

wire    `complex    addsub_res, mul_res;
addsub  ADDSUB (a, b, op[0], addsub_res);
mul     MUL (a, b, mul_res);

assign s = op[1] ? mul_res : addsub_res;

endmodule
```

شکل ۴: alu

۴.۲ $Memory$

۱.۴.۲ هدف

این ماژول حافظه ی فرضی کامپیوتر ساده ی ماست و از ۳۲ کلمه ی ۱۶ بیتی ساخته شده است.

۲.۴.۲ ورودی ها

۱. $Raddr$۱: آدرس اولین ورودی که میخواهیم مقدار آن را بخوانیم و چون حافظه ۳۲ کلمه ایست، به ۵ بیت برای مشخص کردن آن نیاز داریم.

۲. $Raddr$۲: آدرس دومین ورودی که میخواهیم مقدار آن را بخوانیم و چون حافظه ۳۲ کلمه ایست، به ۵ بیت برای مشخص کردن آن نیاز داریم.

۳. $Wdata$: مقداری که می خواهیم در حافظه ذخیره کنیم.

۴. $Waddr$: آدرس مقداری که می خواهیم ورودی $wdata$ در آن ذخیره شود.

در این ماژول همواره مقداری از حافظه که در خانه ی $raddr$۱ قرار دارد، در خروجی $rdata$۱ و مقداری از حافظه که در خانه ی $raddr$۲ قرار دارد در خروجی $rdata$۲ قرار می گیرد. ضمنا با هر تغییر ورودی های $wdata$ یا $waddr$، مقدار $wdata$ در خانه ای از حافظه با اندیس $waddr$ قرار می گیرد. کد وریلاگ این بخش از مدار به شرح زیر است:

```verilog
`include "macros.v"

module memory #(
    parameter   DEPTH = 32,
    parameter   A_LEN = 5
) (
    input   [A_LEN:1]       raddr1,
    input   [A_LEN:1]       raddr2,
    input   `complex        wdata,
    input   [A_LEN:1]       waddr,
    output  `complex    rdata1,
    output  `complex    rdata2
);

reg `complex    mem [DEPTH-1:0];

assign rdata1 = mem[raddr1];
assign rdata2 = mem[raddr2];
always @(*) mem[waddr] <= wdata;

endmodule
```

شکل ۵: حافظه

۵.۲ $Inst\_fetch$

۱.۵.۲ هدف

در این ماژول به واکشی دستورات ۱۷ بیتی می پردازیم. به این صورت که با استفاده از دستور ذخیره شده در حافظه، بخش های مختلف اطلاعات را جدا می کنیم و خروجی می دهیم.

۲.۵.۲ ورودی ها

۱. Clk: سیگنال کلاک.

۲. $rstN$: سیگنال ریست نات که اگر صفر شود باید مقدار pc صفر شود. در واقع انگار به دستور صفرم اشاره می کنیم.

۳.۵.۲ خروجی ها

۱. op: دو بیت که مشخص می کنند واحد محاسبات باید چه عملیات حسابی ای انجام دهد که همانطور که در بخش قبل توضیح داده شد سه حالت دارد: جمع، ضرب یا تفریق.

۲. $Waddr$: آدرس خانه ای از حافظه است که حاصل انجام دستور باید در آن قرار گیرد.

۳. $Raddr۱$: آدرس خانه ای از حافظه که ورودی اول در آن قرار دارد.

۴. $Raddr۲$: آدرس خانه ای از حافظه که ورودی دوم در آن قرار دارد.

ابتدا متغیر pc را برای پیمایش این حافظه مشخص می کنیم. در هر دستور که قرار گرفته باشیم، همواره مقدار خروجی های op، $waddr$، $raddr۱$ و $raddr۲$ به ترتیب برابر با بیت های اول تا دوم آن دستور، بیت های سوم تا هفتم آن دستور، بیت های هشتم تا دوازدهم آن دستور و بیت های سیزدهم تا هفدهم آن دستور می باشند. ضمنا با هر لبه بالارونده کلاک یا پایین رونده ریست، اگر مقدار ورودی $rstN$ صفر باشد، مقدار pc برابر با صفر و در غیر این صورت مقدار pc یک واحد افزایش می یابد.
کد وریلاگ این بخش به صورت زیر است:

```
module inst_fetch (
    input           clk,
    input           rstN,
    output  [1:0]   op,
    output  [4:0]   waddr,
    output  [4:0]   raddr1,
    output  [4:0]   raddr2
);

localparam  DEPTH = 32;
localparam  A_LEN = 5;

reg [1:17]      mem [DEPTH-1:0];
reg [A_LEN:1]   pc;

assign op       = mem[pc][1:2];
assign waddr    = mem[pc][3:7];
assign raddr1   = mem[pc][8:12];
assign raddr2   = mem[pc][13:17];

always @(posedge clk or negedge rstN)
    pc <= rstN ? (pc + 1) : 0;

endmodule
```

شکل ۶: واحد واکشی دستورات

۶.۲ $Pipeline$

۱.۶.۲ هدف

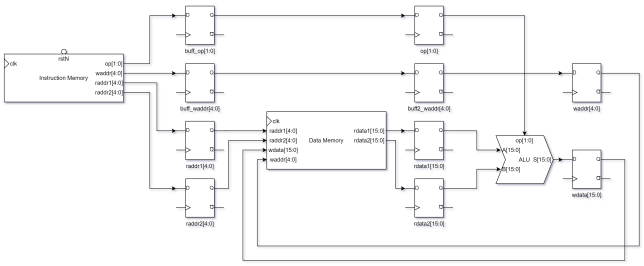
این ماژول، ماژول اصلی آزمایش می شود که از بخش های مختلف استفاده کرده تا عملکرد یک پردازنده که دستورات را به صورت پایپلاین اجرا می کند داشته باشد. مقادیر موجود در حافظه و دستورات در فایل هایی جدا ذخیره می شوند که این واحد به آن ها دسترسی دارد.

۲.۶.۲ ورودی ها

۱. Clk: سیگنال ساعت.

۲. $restN$: سیگنال ریست

دیاگرام این ماژول، به صورت زیر می باشد: ابتدا از هر یک از ماژول های Alu، $Memory$، $Inst\_fetch$ یک نمونه می گیریم:



شکل ۷: پایپلاین

- IF: نمونه گرفته شده از $Inst\_fetch$ می باشد که همان ورودی های ماژول $Pipeline$ را می گیرد و خروجی هایش را در چهار $Wire$ قرار می دهد.
- Mem: نمونه گرفته شده از ماژول $Memory$ که چهار ورودی اش را از چهار reg می گیرد و خروجی هایش را در دو $wire$ بر می گرداند.
- Alu: نمونه گرفته شده از ماژول Alu که سه ورودی اش را از سه رجیستر می گیرد و خروجی اش را در یک سیم قرار می دهد.

حال به یک $always\,block$ که حساس به لبه مثبت سیگنال کلاک یا لبه منفی ورودی ریست است وارد می شویم. در صورت یک بودن مقدار $restN$، در هر کلاک مقادیر خروجی سی نمونه ذکر شده که در $wire$ های مختلف ذخیره شده بودند، به reg منتقل می شوند.
خروجی i_op از IF در بافر $buff\_op$ قرار می گیرد، سپس مقدار Op که ورودی Alu می باشد برابر با $buff\_op$ می شود که با این که در کلاک جدید مقداردهی شده است، اما چون $non\_blocking$ می باشد، تا پایان کلاک مقدار سابق خود را حفظ می کند و باعث می شود کامپیوتر به صورت پایپلاین عمل کند. به طور مشابه خروجی i_waddr از IF ابتدا در بافر buf_waddr و سپس در بافر $buf۲\_waddr$ قرار می گیرد تا پس از انجام محاسبات در Alu، نتیجه صحیح بتواند در آن خانه از حافظه ذخیره شود و خللی در عملیات پایپلاین وارد نشود. در واقع به کمک این بافرها و رجیسترها، هر دستور در هر کلاک که در مرحله محاسبات و ذخیره در حافظه قرار داشته باشد، دستور پیشین آن در یک مرحله قبلتر یعنی خواندن از حافظه و دستور پیش از پیشین در دو مرحله قبلتر یعنی واکشی قرار دارد.
در نهایت، پیاده سازی این ماژول نیز به صورت زیر می باشد:

```
`include "macros.v"

module pipeline (
    input           clk,
    input           rstN
);

wire    [1:0]       i_op;
wire    [4:0]       i_waddr, i_raddr1, i_raddr2;
wire    `complex    m_rdata1, m_rdata2, alu_out;

reg     [1:0]       buff_op, op;
reg     [4:0]       buff_waddr, buff2_waddr, raddr1, raddr2, waddr;
reg     `complex    rdata1, rdata2, wdata;

inst_fetch  IF(clk, rstN, i_op, i_waddr, i_raddr1, i_raddr2);
memory      MEM(raddr1, raddr2, wdata, waddr, m_rdata1, m_rdata2);
alu         ALU(rdata1, rdata2, op, alu_out);

always @(posedge clk or negedge rstN) begin
    if (rstN) begin
        // IF
        buff_op <= i_op;
        buff_waddr <= i_waddr;
        raddr1 <= i_raddr1;
        raddr2 <= i_raddr2;

        // MEM
        rdata1 <= m_rdata1;
        rdata2 <= m_rdata2;
        op <= buff_op;
        buff2_waddr <= buff_waddr;

        // ALU
        waddr <= buff2_waddr;
        wdata <= alu_out;

        $display("#%d\tbuff_op=%b, buff_waddr=%d, raddr1=%d, raddr2=%d", $time,
            buff_op, buff_waddr, raddr1, raddr2);

        $display("#%d\top=%b, buff2_waddr=%d, rdata1=(%d, %d), rdata2=(%d, %d)", $time,
            op, buff2_waddr, `sRe(rdata1), `sIm(rdata1), `sRe(rdata2), `sIm(rdata2));

        $display("#%d\twaddr=%d, wdata1=(%d, %d)\n", $time,
            waddr, `sRe(wdata), `sIm(wdata));
    end
end

endmodule
```

شکل ۸: پایپلاین

۳ تست بنچ

در این ماژول که $pipeline\_TB$ نام دارد، عملکرد کلی مدار بررسی می شود. در ابتدا از ماژول اصلی که پایپلاین می باشد، یک نمونه می گیریم. سپس مقادیر موجود در دو فایل $inst\_mem$ و $initial\_mem$ که به ترتیب مجموعه دستورات و مقادیر اولیه ذخیره شده در حافظه می باشند در آرایه mem در بخش IF از ماژول $Pipeline$ و در آرایه mem در بخش MEM از ماژول $Pipeline$ قرار می گیرند.
مقادیر موجود در $initial\_mem$ عبارت اند از:

```
00011001_11100100        // (25, -28)
00001110_00001001        // (14, 9)
00000101_11110010        // (5, -14)
00010101_00000111        // (21, 7)
11001111_00010001        // (-49, 17)
00011101_11101001        // (29, -23)
00010111_00001111        // (23, 15)
11001101_11111010        // (-51, -6)
11110110_00001010        // (-10, 10)
11111110_11011101        // (-2, -35)
00000001_00000010        // (1, 2)
11111111_00100000        // (-1, 32)
01000101_10111010        // (69, -70)
11110111_10011111        // (-9, -97)
11111111_01100110        // (-1, 102)
00010111_11111001        // (23, -7)
00110100_00000000        // (52, 0)
01011001_00110110        // (89, 54)
11000100_11100101        // (-60, -27)
11110110_11001101        // (-10, -51)
11111111_10011001        // (-1, -103)
01100001_10101101        // (97, -83)
01101111_11100011        // (111, -29)
11111110_11111110        // (-2, -2)
01001011_10101110        // (75, -82)
11001111_10000110        // (-49, -122)
10011101_11111011        // (-99, -5)
10010001_01111000        // (-111, 120)
00111010_01011110        // (58, 94)
00101011_01111101        // (43, 125)
01011010_00110001        // (90, 49)
00111101_00000110        // (61, 6)
```

مقادیر موجود در $inst\_mem$ عبارت اند از:

```
10_00111_01010_10111    // mul $00111 $01010 $10111
10_00011_10111_00001    // mul $00011 $10111 $00001
10_01011_00001_10111    // mul $01011 $00001 $10111
10_11011_01010_01001    // mul $11011 $01010 $01001
00_00010_00111_10010    // add $00010 $00111 $10010
00_01000_11011_01111    // add $01000 $11011 $01111
00_00010_00111_01001    // add $00010 $00111 $01001
00_00101_01011_10001    // add $00101 $01011 $10001
01_00110_00001_01000    // sub $00110 $00001 $01000
01_10110_10011_01000    // sub $10110 $10011 $01000
01_11100_11111_00101    // sub $11100 $11111 $00101
01_11010_01111_00010    // sub $11010 $01111 $00010
01_01011_00100_00101    // sub $01011 $00100 $00101
01_01111_10001_01000    // sub $01111 $10001 $01000
01_00101_10100_00101    // sub $00101 $10100 $00101
```

با توجه به این که ۱۵ دستور داریم و روند اجرای دستورات به صورت پایپلاین است، حداقل به ۱۷ کلاک نیاز داریم. پس به محض رسیدن به کلاک ۱۸ ام، اجرا متوقف می شود.
نهایتا آرایه mem در بخش MEM از ماژول $Pipeline$ که اکنون برخی از خانه های آن مقادیر جدیدی دارند، به عنوان حافظه نهایی در فایل $final\_mem$ ذخیره می شود:

```
// memory data file (do not edit the following line - required for mem load use)
// instance=/pipeline_TB/PIPELINE/MEM/mem
// format=bin addressradix=h dataradix=b version=1.0 wordsperline=1 noaddress
0001100111100100
0000111000001001
0000000011010111
1111011011010010
1100111100010001
1011000010010001
1011001100110111
0000001011111010
0101101111010010
1111111011011101
0000000100000010
1000000000001001
0100010110111010
1111011110011111
1111111101100110
1111111001100100
0011010000000000
0101100100110110
1100010011100101
1111011011001101
1111111110011001
0110000110101101
1001101111111011
1111111011111110
0100101110101110
1100111110000110
0001011100100010
0100010011011001
1110111011111110
0010101101111101
0101101000110001
0011110100000110
```

مقادیر حقیقی و موهومی این اعداد به شکل زیر است:

```
0001100111100100      // (25, -28)
0000111000001001      // (14, 9)
0000000011010111      // (0, -41)
1111011011010010      // (-10, -46)
1100111100010001      // (-49, 17)
1011000010010001      // (-80, -111)
1011001100110111      // (-77, 55)
0000001011111010      // (2, -6)
0101101111010010      // (91, -46)
1111111011011101      // (-2, -35)


0000000100000010      // (1, 2)
1000000000001001      // (-128, 9)
0100010110111010      // (69, -70)
1111011110011111      // (-9, -97)
1111111101100110      // (-1, 102)
1111111001100100      // (-2, 100)
0011010000000000      // (52, 0)
0101100100110110      // (89, 54)
1100010011100101      // (-60, -27)
1111011011001101      // (-10, -51)
1111111110011001      // (-1, -103)
0110000110101101      // (97, -83)
1001101111111011      // (-101, -5)
1111111011111110      // (-2, -2)
0100101110101110      // (75, -82)
1100111110000110      // (-49, -122)
0001011100100010      // (23, 34)
0100010011011001      // (68, -39)
1110111011111110      // (-18, -2)
0010101101111101      // (43, 125)
0101101000110001      // (90, 49)
0011110100000110      // (61, 6)
```

کد وریلاگ این ماژول به شرح زیر است:

```
module pipeline_TB ();

reg rstN = 0, clk = 1;
pipeline PIPELINE(clk, rstN);

always #10 clk = ~clk;
initial begin
    $readmemb("data/inst_mem.txt", PIPELINE.IF.mem, 0, 32);
    $readmemb("data/initial_mem.txt", PIPELINE.MEM.mem, 0, 32);

    #40 rstN = 1;
    wait(PIPELINE.IF.pc == 18);
    $writememb("data/final_mem.txt", PIPELINE.MEM.mem);
    $stop;
end

endmodule
```

با شبیه سازی این ماژول توسط $modelsim$ می توان مشاهده کرد که طبق انتظار، در هر کلاک (به جز کلاک های ابتدایی و انتهایی که تنها یک دستور در حال جرا دارند) همزمان یک دستور در حال واکشی، یک دستور در حال خوانده شدن از حافظه و یک دستور در حال محاسبه و ذخیره است:

```
#    Time: 0 ns  Iteration: 0  Instance: /pipeline_TB
#                 40   buff_op=xx, buff_waddr= x, raddr1= x, raddr2= x
#                 40   op=xx, buff2_waddr= x, rdata1=(   x,    x), rdata2=(   x,    x)
#                 40   waddr= x, wdata1=(   x,    x)
#
#                 60   buff_op=10, buff_waddr= 7, raddr1=10, raddr2=23
#                 60   op=xx, buff2_waddr= x, rdata1=(   x,    x), rdata2=(   x,    x)
#                 60   waddr= x, wdata1=(   x,    x)
#
#                 80   buff_op=10, buff_waddr= 3, raddr1=23, raddr2= 1
#                 80   op=10, buff2_waddr= 7, rdata1=(   1,    2), rdata2=(  -2,   -2)
#                 80   waddr= x, wdata1=(   x,    x)
#
#                100   buff_op=10, buff_waddr=11, raddr1= 1, raddr2=23
#                100   op=10, buff2_waddr= 3, rdata1=(  -2,   -2), rdata2=(  14,    9)
#                100   waddr= 7, wdata1=(   2,   -6)
#
#                120   buff_op=10, buff_waddr=27, raddr1=10, raddr2= 9
#                120   op=10, buff2_waddr=11, rdata1=(  14,    9), rdata2=(  -2,   -2)
#                120   waddr= 3, wdata1=( -10,  -46)
#
#                140   buff_op=00, buff_waddr= 2, raddr1= 7, raddr2=18
#                140   op=10, buff2_waddr=27, rdata1=(   1,    2), rdata2=(  -2,  -35)
#                140   waddr=11, wdata1=( -10,  -46)
#
#                160   buff_op=00, buff_waddr= 8, raddr1=27, raddr2=15
#                160   op=00, buff2_waddr= 2, rdata1=(   2,   -6), rdata2=( -60,  -27)
#                160   waddr=27, wdata1=(  68,  -39)
#
#                180   buff_op=00, buff_waddr= 2, raddr1= 7, raddr2= 9
#                180   op=00, buff2_waddr= 8, rdata1=(  68,  -39), rdata2=(  23,   -7)
#                180   waddr= 2, wdata1=( -58,  -33)
#
#                200   buff_op=00, buff_waddr= 5, raddr1=11, raddr2=17
#                200   op=00, buff2_waddr= 2, rdata1=(   2,   -6), rdata2=(  -2,  -35)
#                200   waddr= 8, wdata1=(  91,  -46)
#
#                220   buff_op=01, buff_waddr= 6, raddr1= 1, raddr2= 8
#                220   op=00, buff2_waddr= 5, rdata1=( -10,  -46), rdata2=(  89,   54)
#                220   waddr= 2, wdata1=(   0,  -41)
```

```
#
#                240   buff_op=01, buff_waddr=22, raddr1=19, raddr2= 8
#                240   op=01, buff2_waddr= 6, rdata1=(  14,    9), rdata2=(  91,  -46)
#                240   waddr= 5, wdata1=(  79,    8)
#
#                260   buff_op=01, buff_waddr=28, raddr1=31, raddr2= 5
#                260   op=01, buff2_waddr=22, rdata1=( -10,  -51), rdata2=(  91,  -46)
#                260   waddr= 6, wdata1=( -77,   55)
#
#                280   buff_op=01, buff_waddr=26, raddr1=15, raddr2= 2
#                280   op=01, buff2_waddr=28, rdata1=(  61,    6), rdata2=(  79,    8)
#                280   waddr=22, wdata1=(-101,   -5)
#
#                300   buff_op=01, buff_waddr=11, raddr1= 4, raddr2= 5
#                300   op=01, buff2_waddr=26, rdata1=(  23,   -7), rdata2=(   0,  -41)
#                300   waddr=28, wdata1=( -18,   -2)
#
#                320   buff_op=01, buff_waddr=15, raddr1=17, raddr2= 8
#                320   op=01, buff2_waddr=11, rdata1=( -49,   17), rdata2=(  79,    8)
#                320   waddr=26, wdata1=(  23,   34)
#
#                340   buff_op=01, buff_waddr= 5, raddr1=20, raddr2= 5
#                340   op=01, buff2_waddr=15, rdata1=(  89,   54), rdata2=(  91,  -46)
#                340   waddr=11, wdata1=(-128,    9)
#
#                360   buff_op=xx, buff_waddr= x, raddr1= x, raddr2= x
#                360   op=01, buff2_waddr= 5, rdata1=(  -1, -103), rdata2=(  79,    8)
#                360   waddr=15, wdata1=(  -2,  100)
#
#                380   buff_op=xx, buff_waddr= x, raddr1= x, raddr2= x
#                380   op=xx, buff2_waddr= x, rdata1=(   x,    x), rdata2=(   x,    x)
#                380   waddr= 5, wdata1=( -80, -111)
#
```

با بررسی نتیجه هر دستور مشاهده می شود که پایپلاین به درستی کار می کند.