بسمه تعالی

گزارش کار هشتم آزمایشگاه طراحی سیستم های دیجیتال

ALU اعداد مختلط

استاد

دکتر انصاری

نویسنده

سیدعماد امام جمعه – ۴۰۰۱۰۸۷۷۴

آرش ضیایی رازبان – ۴۰۰۱۰۵۱۰۹

محمدعرفان سلیما – ۴۰۰۱۰۵۰۱۴

دانشگاه صنعتی شریف

تابستان 1402

مقدمه

در این آزمایش هدف طراحی یک ALU برای محاسبات اعداد مختلط است. که از 3 بخش اصلی جمع/تفریق کننده، ضرب کننده و پایپ لاین تشکیل شده. هرکدام از این بخشها جداگانه تست خواهند شد و در نهایت در کنار یکدیگر قرار خواهند گرفت.

پیاده سازی

هرکدام از بخش های ذکر شده در دستور کار را به صورت جداگانه بررسی میکنیم.

macros

ابتدا به یک سری از macro ها و متغیرهای ثابت در طول برنامه اشاره میکنیم که در فایل زیر آمده اند.

```
`define MEM_LEN 32 // length of memory
`define A_LEN 5 // length of address memory
`define W_LEN 8 // length of the imaginary and real parts of the numbers.
`define complex [2*`W_LEN-1:0]
`define Re(c) c[2*`W_LEN-1:`W_LEN]
`define Im(c) c[`W_LEN-1:0]
`define sRe(c) $signed(`Re(c))
`define sIm(c) $signed(`Im(c))
```

توضیحات اعداد ثابت به صورت کامنت آمده اند. Re و Im هم به ترتیب بخش حقیقی و موهومی عدد مختلط را خروجی میدهند.

Add/Sub

همانطور که از اسم ماژول پیداست، وظیفه این ماژول تولید جمع و تفریق اعداد مختلط است. که حاصل این عملیات با جمع و تفریق مستقل دو بخش عدد مختلط به دست می آید. کد Verilog این ماژول به شکل زیر است:

```
`include "macros.v"

module AddSub (
    input `complex a,
    input `complex b,
    input op, // 0 add - 1 sub
    output `complex c
);
    assign `Re(c) = (op ? (`sRe(a) - `sRe(b)) : (`sRe(a) + `sRe(b)));
    assign `Im(c) = (op ? (`sIm(a) - `sIm(b)) : (`sIm(a) + `sIm(b)));
endmodule
```

ورودی op مشخص میکند که خروجی جمع یا تفریق دو عدد باشد.

برای تست این مدار TestBench زیر را طراحی میکنیم:

4 عملیات جمع و تفریق در این تست بنچ تست میشود که حاصل همه به درستی تولید شده است.

```
`include "macros.v"

module AddSub_TB;
    reg `complex a, b;
    reg op;
    wire `complex c;

    AddSub addsub(a, b, op, c);

    initial begin

        `Re(a) = -11; `Im(a) = 15;
        `Re(b) = 13; `Im(b) = -14;
        op = 1;
        #10;
        $display("(%d, %d) - (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        `Re(a) = 14; `Im(a) = 3;
        `Re(b) = 21; `Im(b) = -64;
        op = 0;
        #10;
        $display("(%d, %d) + (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        `Re(a) = 20; `Im(a) = 18;
        `Re(b) = -40; `Im(b) = 32;
        op = 0;
        #10;
        $display("(%d, %d) + (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        `Re(a) = 43; `Im(a) = 0;
        `Re(b) = -1; `Im(b) = -37;
        op = 1;
        #10;
        $display("(%d, %d) - (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        $finish();
    end

endmodule
```

خروجی TestBench:

```
PS E:\University\Term 4.5\DSD az\8\Code> vvp a.out
( -11,   15) - (  13,  -14) = ( -24,   29)
(  14,    3) + (  21,  -64) = (  35,  -61)
(  20,   18) + ( -40,   32) = ( -20,   50)
(  43,    0) - (  -1,  -37) = (  44,   37)
AddSub_TB.v:32: $finish called at 40 (1s)
```

Multiply

این ماژول وظیفه ضرب دو عدد مختلط را به عهده دارد. رابطه ضرب 2 عدد مختلط به شکل زیر است:

$$(a\,c - b\,d) + i\,(a\,d + b\,c).$$

بر این اساس ماژول ضرب دو عدد را در Verilog پیاده سازی میکنیم:

```
`include "macros.v"

module Multiply (
    input `complex a,
    input `complex b,
    output `complex c
);
    assign `Re(c) = `sRe(a) * `sRe(b) - `sIm(a) * `sIm(b);
    assign `Im(c) = `sRe(a) * `sIm(b) + `sIm(a) * `sRe(b);
endmodule
```

همچنین برای تست این ماژول، TestBench زیر را استفاده میکنیم:

```
`include "macros.v"

module mul_TB;

    reg `complex a, b;
    wire `complex c;

    Multiply mul(a, b, c);

    initial begin
        `Re(a) = -10; `Im(a) = 5;
        `Re(b) = 3; `Im(b) = -8;
        #10;
        $display("(%d, %d) - (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        `Re(a) = 6; `Im(a) = 3;
        `Re(b) = 2; `Im(b) = -6;
        #10;
        $display("(%d, %d) - (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        `Re(a) = 2; `Im(a) = 8;
        `Re(b) = -0; `Im(b) = 2;
        #10;
        $display("(%d, %d) - (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        `Re(a) = 4; `Im(a) = 1;
        `Re(b) = -2; `Im(b) = -7;
        #10;
```

```
        $display("(%d, %d) - (%d, %d) = (%d, %d)", `sRe(a), `sIm(a), `sRe(b),
`sIm(b), `sRe(c), `sIm(c));
        $finish();
    end

endmodule
```

خروجی این TestBench به شکل زیر خواهد بود:

```
PS E:\University\Term 4.5\DSD az\8\Code> vvp a.out
( -10,    5) * (   3,   -8) = (  10,   95)
(   6,    3) * (   2,   -6) = (  30,  -30)
(   2,    8) * (   0,    2) = ( -16,    4)
(   4,    1) * (  -2,   -7) = (  -1,  -30)
Multiply_TB.v:27: $finish called at 40 (1s)
```

که تمامی خروجی ها به درستی تولید شده اند پس ماژول به درستی کار میکند.

ALU

این ماژول از کنار هم قرار دادن جمع/تفریق کننده و ضرب کننده درست شده است. با استفاده از op مشخص میشود که عملیات مورد نظر ما جمع یا تفریق یا ضرب است. (00 جمع – 01 تفریق – 10 و 11 ضرب)

کد وریلاگ این ماژول:

```
`include "macros.v"

module ALU (
    input `complex a,
    input `complex b,
    input [1:0] op,
    output `complex res
);
    wire `complex addsub_res, mul_res;

    AddSub addsub(a, b, op[0], addsub_res);
    Multiply multiply(a, b, mul_res);

    assign res = (op[1] ? mul_res : addsub_res);

endmodule
```

Memory

این ماژول معادل حافظه 32 Word ای موجود در کامپیوتر است. که قابلیت write و read از 2 آدرس همزمان را دارد. نکته قابل ذکر درباره این ماژول این است که بر اساس کلاک write نمیکند. و هر وقت که مقادیر ورودی تغییری کردند write میکند. مقدار اولیه آرایه mem هم از فایل initial_memory.txt خوانده میشود که در فولدر Data قرار گرفته.

```
`include "macros.v"

module Memory (
    input [`A_LEN-1:0] addr1,
    input [`A_LEN-1:0] addr2,
    input `complex write,
    input [`A_LEN-1:0] addr3,
    output `complex read1,
    output `complex read2
);
    reg `complex mem [`MEM_LEN-1:0];

    assign read1 = mem[addr1];
    assign read2 = mem[addr2];
    always @(*) mem[addr3] <= write;
endmodule
```

Instruction

این ماژول هم مشابه Memory یک حافظه است که از آن دستورهای لازم برای اجرا خوانده میشود. پس از هر کلاک pc یک واحد جلو میرود تا instruction بعدی خوانده شود. همچنین instruction ها در همین ماژول بخش بندی شده و بخش متناظر op و آدرس ها به صورت مجزا خروجی داده میشوند.

```
`include "macros.v"

module Instruction (
    input clk, rstN,
    output [1:0] op,
    output [4:0] addr1,
    output [4:0] addr2,
    output [4:0] addr3
);
    reg [0:16] mem [`MEM_LEN-1:0];
    reg [`A_LEN:1] pc;

    assign op = mem[pc][0:1];
    assign addr3 = mem[pc][2:6]; // where to write
    assign addr1 = mem[pc][7:11];
    assign addr2 = mem[pc][12:16];

    always @(posedge clk or negedge rstN) begin
        if(~rstN) begin
            pc <= 0;
        end
        else begin
            pc <= pc + 1;
        end
    end
endmodule
```

PipeLine

در این ماژول همه ماژول های قبلی به یکدیگر متصل میشوند و کل پردازنده به صورت pipeline فعالیت میکند. زیرماژول های این ماژول Memory و ALU و Instruction است.

همچنین پایپ لاین مورد استفاده در این پردازنده از 3 stage تشکیل شده است که برای اجرای آن به صورت پایپلاین میان اتصالات هر بخش یک رجیستر قرار گرفته تا با هر کلاک دیتا یک stage جلو برود:

1. Inst: در این stage دستور خوانده میشود.
2. Mem: در این stage مقادیر متناظر آدرس های موجود در دستور، از حافظه خوانده میشود.
3. ALU: در این stage حاصل اجرای عملیات بر روی ورودی ها که در خروجی ALU قرار گرفته نوشته میشود.

برای انتقال برخی از دیتاها میان استیجها بعضی از خروجی ها مثل buff_op هستند که صرفا برای این buffer شده اند که در استیج درستی مورد استفاده قرار بگیرند.

لازم به ذکر است که در این پردازنده hazard هایی مانند RAW شدنی هستند و برای جلوگیری از آنها کاری صورت نگرفته.

کد وریلاگ این ماژول که ماژول اصلی مدار است در زیر آمده است:

```
`include "macros.v"

module Pipeline (
    input clk, rstN
);

wire [4:0] inst_addr1, inst_addr2, inst_addr3;
wire [1:0] inst_op;
wire `complex mem_read1, mem_read2, alu_res;

reg [4:0] buff_addr3, buff2_addr3, addr1, addr2, addr3;
reg [1:0] buff_op, op;
reg `complex read1, read2, write;

Memory mem(addr1, addr2, write, addr3, mem_read1, mem_read2);
ALU alu(read1, read2, op, alu_res);
Instruction inst(clk, rstN, inst_op, inst_addr1, inst_addr2, inst_addr3);

always @(posedge clk or negedge rstN) begin
    if (rstN) begin
        // inst
        addr1 <= inst_addr1;
        addr2 <= inst_addr2;
        buff_op <= inst_op;
        buff_addr3 <= inst_addr3;

        // mem
        read1 <= mem_read1;
        read2 <= mem_read2;
```

```
        op <= buff_op;
        buff2_addr3 <= buff_addr3;

        // ALU
        addr3 <= buff2_addr3;
        write <= alu_res;

        $display("%dbuff_op=%b, buff_addr3=%d, addr1=%d, addr2=%d", $time,
buff_op, buff_addr3, addr1, addr2);
        $display("%dop=%b, buff2_addr3=%d, read1=(%d, %d), read2=(%d, %d)",
$time, op, buff2_addr3, `sRe(read1), `sIm(read1), `sRe(read2), `sIm(read2));
        $display("%daddr3=%d, write=(%d, %d)\n", $time, addr3, `sRe(write),
`sIm(write));
    end
end

endmodule
```

از دستورات $display برای بررسی درستی مدار استفاده میشود. (چون برداشت کردن از waveform در این مدار دشوارتر است)

برای تست ماژول بالا TestBench زیر نوشته شده است:

```
module pipeline_TB;

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

always #5 clk = ~clk;
initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, pipeline_TB);
    $readmemb("data/instruction_memory.txt", pipeline.inst.mem, 0, 31);
    $readmemb("data/initial_memory.txt", pipeline.mem.mem, 0, 31);

    #20 rstN = 1;
    wait(pipeline.inst.pc == 18);
    $writememb("data/final_memory.txt", pipeline.mem.mem);
    $finish();
end

endmodule
```

توجه کنید با وجود اینکه 16 دستور داریم، نیاز به 18 کلاک داریم چون پس از خواندن آخرین دستور به 3 کلاک برای اتمام کار آن دستور نیاز خواهیم داشت.

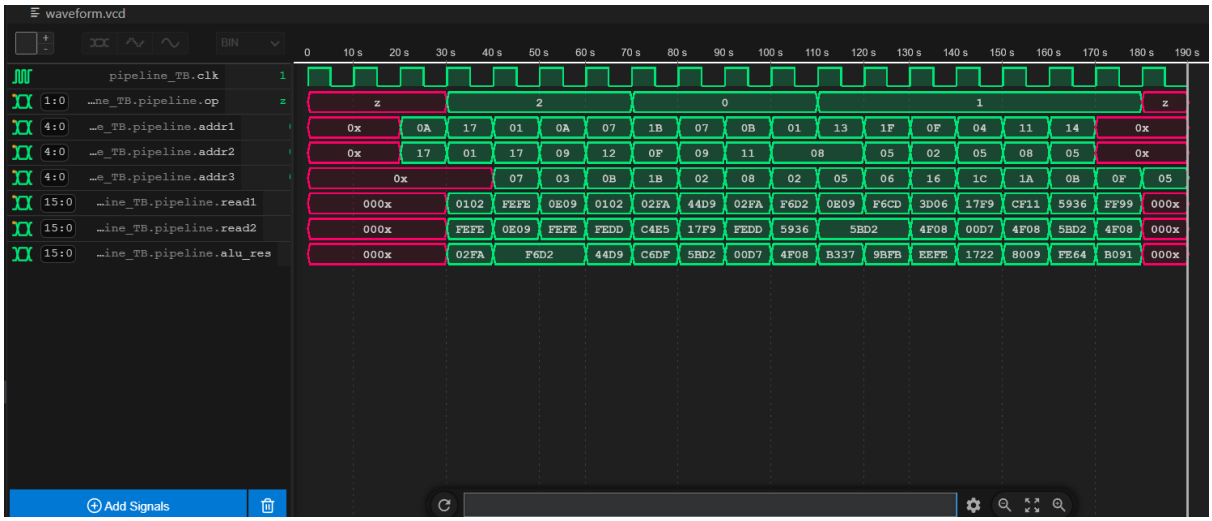
محتویات حافظه اولیه و حافظه دستورات نیز در صفحه بعد آورده شده.

data > instruction_memory.txt

```
1   10_00111_01010_10111    // mul $00111 $01010 $10111
2   10_00011_10111_00001    // mul $00011 $10111 $00001
3   10_01011_00001_10111    // mul $01011 $00001 $10111
4   10_11011_01010_01001    // mul $11011 $01010 $01001
5   00_00010_00111_10010    // add $00010 $00111 $10010
6   00_01000_11011_01111    // add $01000 $11011 $01111
7   00_00010_00111_01001    // add $00010 $00111 $01001
8   00_00101_01011_10001    // add $00101 $01011 $10001
9   01_00110_00001_01000    // sub $00110 $00001 $01000
10  01_10110_10011_01000    // sub $10110 $10011 $01000
11  01_11100_11111_00101    // sub $11100 $11111 $00101
12  01_11010_01111_00010    // sub $11010 $01111 $00010
13  01_01011_00100_00101    // sub $01011 $00100 $00101
14  01_01111_10001_01000    // sub $01111 $10001 $01000
15  01_00101_10100_00101    // sub $00101 $10100 $00101
```

```
1   00011001_11100100   // (25, -28)
2   00001110_00001001   // (14, 9)
3   00000101_11110010   // (5, -14)
4   00010101_00000111   // (21, 7)
5   11001111_00010001   // (-49, 17)
6   00011101_11101001   // (29, -23)
7   00010111_00001111   // (23, 15)
8   11001101_11111010   // (-51, -6)
9   11110110_00001010   // (-10, 10)
10  11111110_11011101   // (-2, -35)
11  00000001_00000010   // (1, 2)
12  11111111_00100000   // (-1, 32)
13  01000101_10111010   // (69, -70)
14  11110111_10011111   // (-9, -97)
15  11111111_01100110   // (-1, 102)
16  00010111_11111001   // (23, -7)
17  00110100_00000000   // (52, 0)
18  01011001_00110110   // (89, 54)
19  11000100_11100101   // (-60, -27)
20  11110110_11001101   // (-10, -51)
21  11111111_10011001   // (-1, -103)
22  01100001_10101101   // (97, -83)
23  01101111_11100011   // (111, -29)
24  11111110_11111110   // (-2, -2)
25  01001011_10101110   // (75, -82)
26  11001111_10000110   // (-49, -122)
27  10011101_11111011   // (-99, -5)
28  10010001_01111000   // (-111, 120)
29  00111010_01011110   // (58, 94)
30  00101011_01111101   // (43, 125)
31  01011010_00110001   // (90, 49)
32  00111101_00000110   // (61, 6)
```

Waveform حاصل پس از شبیه سازی به شکل زیر خواهد بود. که البته تحلیل آن به خصوص با توجه به اینکه اعداد مختلط به درستی نمایش داده نمیشوند دشوار است:



بنابراین از خروجی های display برای بررسی درستی عملکرد استفاده میکنیم.

خروجی $display پس از شبیه سازی test bench به طور کامل در فایل display_output.txt به طور کامل آمده. اما به دلیل طولانی بودن ما با آوردن چند کلاک اول عملکرد دستور اول در مدار را تست میکنیم.

- **خروجی 4 کلاک اول:**

20 buff_op=xx, buff_addr3= x, addr1= x, addr2= x

20 op=xx, buff2_addr3= x, read1=(x, x), read2=(x, x)

20 addr3= x, write=(x, x)

30 buff_op=10, buff_addr3= 7, addr1=10, addr2=23

30 op=xx, buff2_addr3= x, read1=(x, x), read2=(x, x)

30 addr3= x, write=(x, x)

40 buff_op=10, buff_addr3= 3, addr1=23, addr2= 1

40 op=10, buff2_addr3= 7, read1=(1, 2), read2=(-2, -2)

40 addr3= x, write=(x, x)

50 buff_op=10, buff_addr3=11, addr1= 1, addr2=23

50 op=10, buff2_addr3= 3, read1=(-2, -2), read2=(14, 9)

50 addr3= 7, write=(2, -6)

همانطور که میبینید دستور اول معادل است با:

mul $00111 $01010 $10111

که یعنی حاصل ضرب محتویات آدرس 10 و 23 را در آدرس 7 قرار بده. که میبینیم آدرس ها و op به درستی در stage اول خوانده شده است. (زمان 30 خط اول)

پس از آن محتویات دو آدرس به درستی خوانده شده که برابر است با 2i + 1 و 2i – 2- . (زمان 40 خط دوم)

در استیج بعدی حاصل آنها در write به درستی (6i – 2) محاسبه شده و در آدرس 7 قرارداده میشود. (زمان 50 خط سوم)

پس مدار به درستی روند اجرای این دستور را طی کرد.

منابع:

Mano, Morris. *Computer system architecture*. Prentice-Hall of India, 2003.