به نام خدا

گزارش آزمایش ۸

علیرضا سلیمیان 400105036

امیرحسین علمدار 400105144

پیام تائبی 400104867

با توجه به مشکلاتی که جلسه پیش در آزمایشگاه به آن برخورد کرده بودیم و کدی که زده شده بود، جوابگو نبود تصمیم گرفتیم که کد را کلا تغییر دهیم. مشکلاتی که در جلسه پیش به آن برخورد کرده بودیم این بود که باید میتوانستیم خروجی را روی FPGA نمایش دهیم و خب کدی که ما زده بودیم مناسب این کار نبود.

از چیز هایی که یاد گرفتیم این بود که به reg فقط در always میتوان مقدار داد و خارج از آن نمیتوان آنرا مقدار دهی کرد و این بخش بزرگی از مشکل ما بود، نیاز بود که به reg مقدار دهی کنیم ولی نمیتوانستیم.

در ادامه به توضیح کد جدید میپردازیم:

در این کد، فرض شده است که مقدار موهومی و حقیقی در دو رجیستر مجزا هستند و محاسبات برای هرکدام جدا انجام میشود. در ادامه کدی که برای جمع/تفریق زده شده است را بررسی میکنیم:

بخش جمع/تفریق:

ورودی ها:

- Rin1: بخش حقیقی ورودی اول
- Iin1: بخش موهومی ورودی اول
- Rin2: بخش حقیقی ورودی دوم
- Iin2: بخش موهومی ورودی دوم
- addSubN: مشخص کننده نوع عملیات

خروجی ها:

- Rout: مشخص کننده بخش حقیقی جواب
- Iout: مشخص کننده بخش موهومی جواب

```
1   module add_sub #(parameter n=4)
2             (input signed [n-1:0] Rin1,
3              input signed [n-1:0] Iin1,
4              input signed [n-1:0] Rin2,
5              input signed [n-1:0] Iin2,
6              input addSubN,
7              output signed [n-1:0] Rout,
8              output signed [n-1:0] Iout
9             );
10  assign Rout = addSubN ? (Rin1+Rin2) : (Rin1-Rin2);
11  assign Iout = addSubN ? (Iin1+Iin2) : (Iin1-Iin2);
12  endmodule
13
14
```

بخش های حقیقی با یکدیگر جمع یا تفریق میشوند و بخش های موهومی نیز با یکدیگر جمع یا تفریق میشوند.

بخش ضرب:

ورودی ها:

- Rin1: بخش حقیقی ورودی اول
- Iin1: بخش موهومی ورودی اول
- Rin2: بخش حقیقی ورودی دوم
- Iin2: بخش موهومی ورودی دوم

خروجی ها:

- Rout: مشخص کننده بخش حقیقی جواب
- Iout: مشخص کننده بخش موهومی جواب

```
1   module mult#(parameter n=4)
2     (           input signed [n-1:0] Rin1,
3                 input signed [n-1:0] Iin1,
4                 input signed [n-1:0] Rin2,
5                 input signed [n-1:0] Iin2,
6                 output signed[2*n-1:0] Rout,
7                 output signed[2*n-1:0] Iout
8     );
9   assign Rout = Rin1*Rin2 - Iin1*Iin2;
10  assign Iout = Rin1*Iin2 - Rin2*Iin1;
11  endmodule
12
```

فرض کنید دو عدد $a = (c + di)$ و $b = (e + fi)$ را در یکدیگر ضرب کنیم، میدانیم که i به معنی $\sqrt{-1}$ است. اگر دو عدد را در یکدگر ضرب کنیم به عبارت: $result = (ce - fd) + (cf + ed)i$ خواهیم رسید که قسمت حقیقی آن، بخش اول و قسمت موهومی آن بخش دوم آن است پس برای $result$ بخش های حقیقی و موهومی آنرا با عبارت های بالا قرار می‌دهیم.

بخش حافظه:

مشکل اصلی ما در کدی که قبلا زده شده بود، در این بخش بود. این بخش در حقیقت instruction memoryما است، میدانیم برای آنکه بتوانیم pipline داشته باشیم، باید instruction memory و data memory ما از یکدیگر جدا باشند تا به مشکل نخوریم.

ورودی ها:

- Pc: مشخص کننده آدرس جایی که باید دستور آن خوانده شود.

خروجی ها:

- Out: مشخص کننده مقدار خوانده شده از حافظه.

```
module memory#(parameter m =32)
                (       input [4:0]PC,
                        output [m-1:0]out
                );
            reg [m-1:0] Mem [31:0];
            assign out = Mem[PC];
            always@(PC) begin
                Mem[0]= {2'b00,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[1]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[2]= {2'b01,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[3]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[4]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[5]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[6]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[7]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[8]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[9]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[10]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[11]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[12]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[13]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[14]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[15]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[16]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[17]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[18]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[19]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[20]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[21]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[22]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[23]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[24]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[25]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[26]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[27]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[28]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[29]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[30]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};
                Mem[31]= {2'b10,5'b00100,5'b00101,5'b00000,5'b00001,5'b00010,5'b00011};


            end
endmodule
```

حال به توضیح decode کردن دستور میپردازیم:

دو بیت ابتدایی مشخص کننده کاری است که باید انجام شود، ما 00 را برای جمع، ۰۱ تفریق و ۱۰ را برای ضرب در نظر گرفته ایم.

و هرکدام از بخش های بعد به ترتیب به صورت زیر هستند:

- ۵ بیت اول مشخص کننده رجیستر مقصد برای بخش حقیقی
- ۵ بیت دوم مشخص کننده رجیستر مقصد برای بخش موهومی
- ۵ بیت سوم مشخص کننده بخش حقیقی عدد اول

- ۵ بیت چهارم مشخص کننده بخش موهومی عدد اول
- ۵ بیت پنجم مشخص کننده بخش حقیقی عدد دوم
- ۵ بیت ششم مشخص کننده بخش موهومی عدد دوم

چون باید دستورات را به صورت دستی وارد میکردیم، برای همین فقط دستورات اول تا سوم یکتا هستند و نتیجه آنها نیز در انتها در waveform نمایش داده میشود که به صورت pipline اجرا میشوند.

بخش pipline:

نکته ای که در پیاده سازی این بخش وجود دارد،انتقال مرحله به مرحله در pipline است. میدانیم که در pipline ها، کار را بخش های مختلف تقسیم میکنیم و در هر کلاک هر دستور یک بخش جلو میرود، در pipline حتی اگر یک دستور به یک بخش نیاز نداشته باشد باید آنرا طی کند. Pipline در دستورات به تعداد بالا بسیار به سرعت کمک میکند.

ورودی ها:

- Start: مشخص کننده شروع کار برای پردازنده
- Load: برای مقدار دهی اولیه به data memory
- Address: این بخش مشخص کننده جایی است که میخواهیم مقدار آنرا بدانیم.
- Clk: برای کلاک استفاده میشود.

خروجی ها:

- Out: مشخص کننده خروجی است که با استفاده از address مشخص میشود.

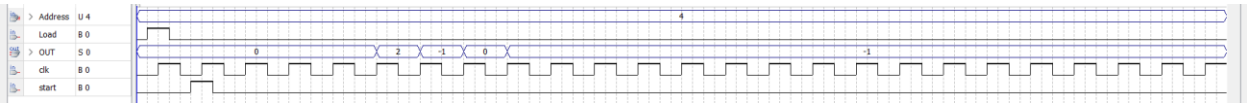
هنگامی که load یک باشد، مقدار دهی اولیه به data memory انجام میشود و در کلاک های بعد میتوان از آنها استفاده کرد.

```
module pipline#(parameter LogRFsize =5, parameter RegLength = 4, parameter RFsize = 32, parameter wordLength = 32)
                (input start,
                input [LogRFsize-1:0] Address,
                input Load,
                input clk,
                     output [RegLength-1:0] OUT
  );
reg [RegLength-1:0] RegFile [RFsize - 1:0];
reg[4:0] pc;
assign OUT = RegFile[Address];
wire [wordLength-1:0]PIPEWIRE1;
reg [wordLength-1:0]PIPEREG1;


reg [RegLength-1:0]PIPEREGO1;
reg [RegLength-1:0]PIPEREGO2;
reg [RegLength-1:0]PIPEREGO3;
reg [RegLength-1:0]PIPEREGO4;


reg [wordLength-1-6*LogRFsize:0]OPCODE1;


reg [LogRFsize-1:0]PIPEREGWON1;
reg [LogRFsize-1:0]PIPEREGWON2;


memory #(.m(wordLength))MEM_1(.PC(pc),.out(PIPEWIRE1));

wire [RegLength-1:0]MULTWIRER;// real part of output mult
wire [RegLength-1:0]MULTWIREI; // imaginary part of output mult
wire [RegLength-1:0]ADDSUBWIRER; // real part of output add sub
wire [RegLength-1:0]ADDSUBWIREI;// imaginay part of ouptut add sub

reg [RegLength-1:0]ALUREGR;
reg [RegLength-1:0]ALUREGI;

reg E1,E2,E3;

reg [LogRFsize-1:0]PIPEREGWOM1;
reg [LogRFsize-1:0]PIPEREGWOM2;

mult #(.n(RegLength)) MULT_1(.Rin1(PIPEREGO1),.Iin1(PIPEREGO2),.Rin2(PIPEREGO3),.Iin2(PIPEREGO4),.Rout(MULTWIRER),.Iout(MULTWIREI));
add_sub #(.n(RegLength)) ADDSUB_1(.Rin1(PIPEREGO1),.Iin1(PIPEREGO2),.Rin2(PIPEREGO3),.Iin2(PIPEREGO4),.Rout(ADDSUBWIRER),.Iout(ADDSUBWIREI),.addSubN(!OPCODE1[0]));
```

```
47  reg Flag = 0;
48  always@(posedge clk) begin
49                  if(Load) begin
50                      RegFile[0] <= 4'b0001;
51                      RegFile[1] <= 4'b0001;
52                      RegFile[2] <= 4'b0001;
53                      RegFile[3] <= 4'b0010;
54                  end
55                  else if(!Flag) begin
56                      if(start) begin
57                          pc <= 0;
58                          Flag <= 1;
59                          E1 = 0;
60                          E2 = 0;
61                          E3 = 0;
62                      end
63                  end
64                  else begin
65                     pc <= pc + 1;
66                      PIPEREG1 <= PIPEWIRE1;
67                      E1 <= 1;
68
69                      if(E1) begin
70                          PIPEREGO4 <= RegFile[PIPEREG1[LogRFsize-1:0]];
71                          PIPEREGO3 <= RegFile[PIPEREG1[2*LogRFsize-1:LogRFsize]];
72                          PIPEREGO2 <= RegFile[PIPEREG1[3*LogRFsize-1:2*LogRFsize]];
73                          PIPEREGO1 <= RegFile[PIPEREG1[4*LogRFsize-1:3*LogRFsize]];
74                          PIPEREGWON2 <= PIPEREG1[5*LogRFsize-1:4*LogRFsize];
75                          PIPEREGWON1 <= PIPEREG1[6*LogRFsize-1:5*LogRFsize];
76                          OPCODE1 <= PIPEREG1[wordLength-1:6*LogRFsize];
77                          E2 <= 1;
78                      end
```

```
79
80                      if(E2) begin
81                          if(OPCODE1[1]) begin
82                              ALUREGR <= MULTWIRER;
83                              ALUREGI <= MULTWIREI;
84                          end
85
86                          else begin
87                              ALUREGR <= ADDSUBWIRER;
88                              ALUREGI <= ADDSUBWIREI;
89                          end
90                          PIPEREGWOM1 <= PIPEREGWON1;
91                          PIPEREGWOM2 <= PIPEREGWON2;
92                          E3 <= 1;
93                      end
94
95                      if(E3) begin
96                          RegFile[PIPEREGWOM1] <= ALUREGR;
97                          RegFile[PIPEREGWOM2] <= ALUREGI;
98                      end
99                      if(pc == 31)
100                         Flag <= 0;
101                 end
102             end
103 endmodule
104
```

در ادامه تست مدار را مشاهده میکنید:



همانطور که مشاهده میکنید، سر کلاک چهارم نتایج اولین دستور آماده شده است و بعد از آن، در هر کلاک جواب دستورات بعدی آماده شده است.

البته در این بخش فقط بخش حقیقی تست شده است ولی بخش موهومی نیز به درستی کار میکند.