به نام خدا

گزارش پروژه نهایی درس معماری کامپیوتر دکتر عطارزاده

در این پروژه سعی به ساختن ماژول co-processor بودیم، که با منتظر ماندن فرمان پروسسور بتواند عملیات های gcd و lcm را روی دو عدد 8 بیت انجام داده و خروجی را به پروسسور بدهد، سپس پروسسور آن را در رجیستر مورد نظر یادداشت کند.

بخش صفر: قالب دستورات و نحوه ارتباط

قالب دستورات را از مانند R-type در نظر گرفتیم و opcode, func4, func7 غیر رزرو شده ای برایشان انتخاب کردیم:

که حروف C B A به ترتیب rd rs1 rs2 را نشان میدهند.

ب م م: **0000000A|AAAA|BBBB|B000|CCCC|C000|0000**

ک م م: **0000000A|AAAA|BBBB|B000|CCCC|C000|0001**

- قالب کلمه ای که به کوپروسسور میفرستیم: (WriteData)

Bit #	7:0	15:8	16	31:17
Function	X_0	Y_0	Op (GCD/LCM)	X

*** (op = 1) if gcd, 0 if lcm

- قالب کلمه ای که از کوپروسسور میخوانیم: (ReadData)

Bit #	7:0	8	31:9
Function	Result	Done	X

بخش اول: طراحی کلی پروسسور سینگل سایکل

از ریزمعاری سینگل سایکلی که قبلا ساخته بودیم استفاده میکنیم و آن را طبق نیاز خود آپدیت میکنیم.

پردازنده ما 4 بخش اصلی Program Counter, RegisterFile, Instruction Memory و Data Memory بود که این بخش ها همان باقی میمانند و فقط تغییراتی را روی آنها اعمال میکنیم

آپدیت واحد کنترل:

اضافه کردن سیگنال خروجی Start که به معنای این است که دستور مورد نظر gcd یا lcm است (طبیعتا فقط برای این دو دستور روشن خواهد شد و بعد خاموش میشود)

اضافه کردن سیگنال ورودی copDone که در اول هر کلاک سایکل منتظر روشن شدن این سیم و برداشت جواب نهایی است.

```
module controller(input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic funct7b5,
                  input logic Zero,
                  output logic [1:0] ResultSrc,
                  output logic MemWrite,
                  output logic PCSrc, ALUSrc,
                  output logic RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl,
                  output logic PCRControl,
                  output logic Start,
                  input logic copDone);
```

```
module maindec(input logic [6:0] op,
    always_comb
        case(op)
        // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump_PCRControl_Start
            7'b0000011: controls = 13'b1_00_1_0_01_0_00_0_0_0; // lw
            7'b0100011: controls = 13'b0_01_1_1_00_0_00_0_0_0; // sw
            7'b0110011: controls = 13'b1_xx_0_0_00_0_10_0_0_0; // R-type
            7'b1100011: controls = 13'b0_10_0_0_xx_1_01_0_1_0; // bne / beq
            7'b0010011: controls = 13'b1_00_1_0_00_0_10_0_0_0; // I-type ALU
            7'b1101111: controls = 13'b1_11_0_0_10_0_00_1_1_0; // jal
            7'b1100111: controls = 13'b1_00_1_0_00_0_10_1_0_0; // jalr
            7'b0000000: controls = 13'b1_xx_x_0_11_0_xx_0_0_1; // gcd
            7'b0000001: controls = 13'b1_xx_x_0_11_0_xx_0_0_1; // lcm

            default: controls = 13'bx_xx_x_x_xx_x_xx_x_1_0; // ???
    endcase
endmodule
```

آپدیت دیتاپف:

سیگنال WriteData که قبلا پروسسور برای ارتباط با موری میساخت با فرمت دستوری که حالا میخواهیم به کوپروسسور بدهیم فرق میکند، پس باید هم سیگنال جدید را بسازیم هم WriteDataFinal را انتخاب کنیم.

در دیتاپف با اسایِن کردن 8 بیت R2 وR1 خوانده شده از RF و برداشتن یک بیت از opCode آنها که که تفاوتشان را مشخص میکند دستور را میسازیم.

با استفاده از یک mux و سیگنال Start انتخاب میکنیم که رایت دیتا ما چه باشد.

```
    assign WDCop = {15'b0, Instr[0], WriteData[7:0], SrcA[7:0]};

    WDSel sel (WDCop, WriteData, Start, WDFinal);

    logic [31:0] ans;
    assign ans = {24'b0, copAns[7:0]};

    mux2 #(32) srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
    alu alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
    mux3 #(32) resultmux( ALUResult, ReadData, PCPlus4, ans, ResultSrc, Result);
endmodule
```

همچنین mux که ResultSrc را انتخاب میکرد را نیز آپدیت میکنیم و اپشن خواندن جواب از کوپروسسور را نیز به آن میدهیم.

آپدیت pcUpdate:

با توجه به اینکه کوپروسسور ما از کلاک خود پروسسور استفاده میکند قادتا بیشتر از یک کلاک سایکل (که زمانی است که پروسسور ما دستور بعدی را اجرا میکند) نیاز دارد.

برای جلوگیری از ایجاد تداخل ما یک سیگنال enable اضافه میکنیم که هرگاه یکی از دستورات gcd یا lcm را داشتیم تا زمان Done نشدن کوپروسسور دستور دیگری نخواند.

```
1  module flopenr #(parameter WIDTH = 8)
2                  (input logic clk, reset, en,
3                   input logic [WIDTH-1:0] d,
4                   output logic [WIDTH-1:0] q);
5      always_ff @(posedge clk, posedge reset)
6          if (reset) q <= 0;
7          else if (en) q <= d;
8  endmodule
```

همچنین در فایل ابتدایی خود یک اینستنس از کوپروسسور خود نیز میسازیم.

```
7      // instantiate processor and memories
8      riscvsingle rvsingle( clk, reset, PC, Instr, MemWrite,
9      DataAdr, WriteData, ReadData, WDFinal ,Start, AnsData);
10     imem imem(PC, Instr);
11     dmem dmem(clk, MemWrite, DataAdr, WDFinal, ReadData);
12     coprocessor cop(clk, Start, WDFinal, AnsData);
13 endmodule
```

بخش سوم: تست و بررسی عملکرد

با نوشتن برنامه اسمبلی و تست بنچ کار را شروع میکنیم.

فایل اسمبلی:

```
addi x6, x0, 24            // x6 = 24                01800313
addi x7, x0, 3             // x7 = 3                 00300393
addi x12, x0, 4            // x12 = 4                00400613
lcm x10, x7, x12           // x10 = 12               00C38501            lcm(3,4) = 12
addi x2, x0, 5             // x2 = 5                 00500113
lcm x8, x2, x10            // x8 = 10                00A10401            lcm(5,12) = 60
addi x13, x0, 60           // x13 = 60               03C00693
gcd x3, x6, x8             // x3 = 12                00830180            gcd(24, 60) = 12
or x4, x7, x2              // x4 = 7                 FF718393
and x5, x3, x4             // x5 = 4                 0041F2B3
add x5, x5, x4             // x5 = 11                004182B3
beq x5, x7, end            // shouldnt be taken
slt x4, x3, x4             // x4 = 0
beq x4, x0, around         // should be taken
addi x5, x0, 0             // shouldnt execute
around: slt x4, x7, x2     // x4 = 1
add x7, x4, x5             // x7 = 12
sub x7, x7, x2             // x7 = 7
sw x7, 84(x3)              // [96] = 7
lw x2, 96(x0)              // x2 = [96] = 7
add x9, x2, x5             // x9 = 18
addi x2, x0, 3             // x2 = 3
jal x3, end                // x3 = 92, jump to end
addi x2, x0, 1             // shouldnt execute
end: lcm x2, x2, x7        // x2 = 21                00710101            gcd(3,7) = 21
sw x2, 32(x3)              // [124] = 21
done: beq x2, x2, done     // infinite loop
```

تست نوشته شده تمام دستوراتی را که پردازنده ما ساپورت میکند را حداقل یکبار اجرا میکند تا از کاردکرد آن اطمینان پیدا کند.

ترجمه دستورات اسمبلی و برخی از ماشین کد ها را در گزارش آورده شده و فایل کامل آن نیز گذاشته میشود.

تست بنچ:

با چک کردن سیگنال های WriteData و DataAddress در تست بنچ خود با مقدار های پیش بینی شده چک میکنیم.

سپس آن را سیمیولیت میکنیم:

```
20      // check results
21      always @(negedge clk) begin
22          if(MemWrite)
23          begin
24              if(DataAdr === 124 & WDFinal === 21) begin
25                  $display("Simulation succeeded");
26                  $stop;
27              end
28              else if (DataAdr === 96) begin
29                  $display("Simulation running: DataAdr = %d", DataAdr);
30              end
31              else begin
32                  $display("Simulation failed: DataAdr = %d", DataAdr);
33                  $stop;
34              end
35          end
36      end
37  endmodule
```