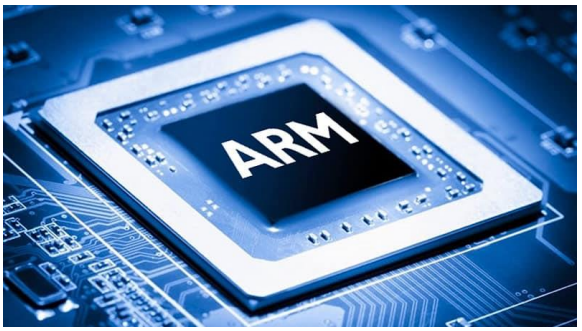
به نام خدا

دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



آزمایشگاه معماری کامپیوتر

گزارش دستور کار شماره 1

محمد صالح عرفاتی
810197543
علی پادیاو
810199388

اسفند 1401

مقدمه

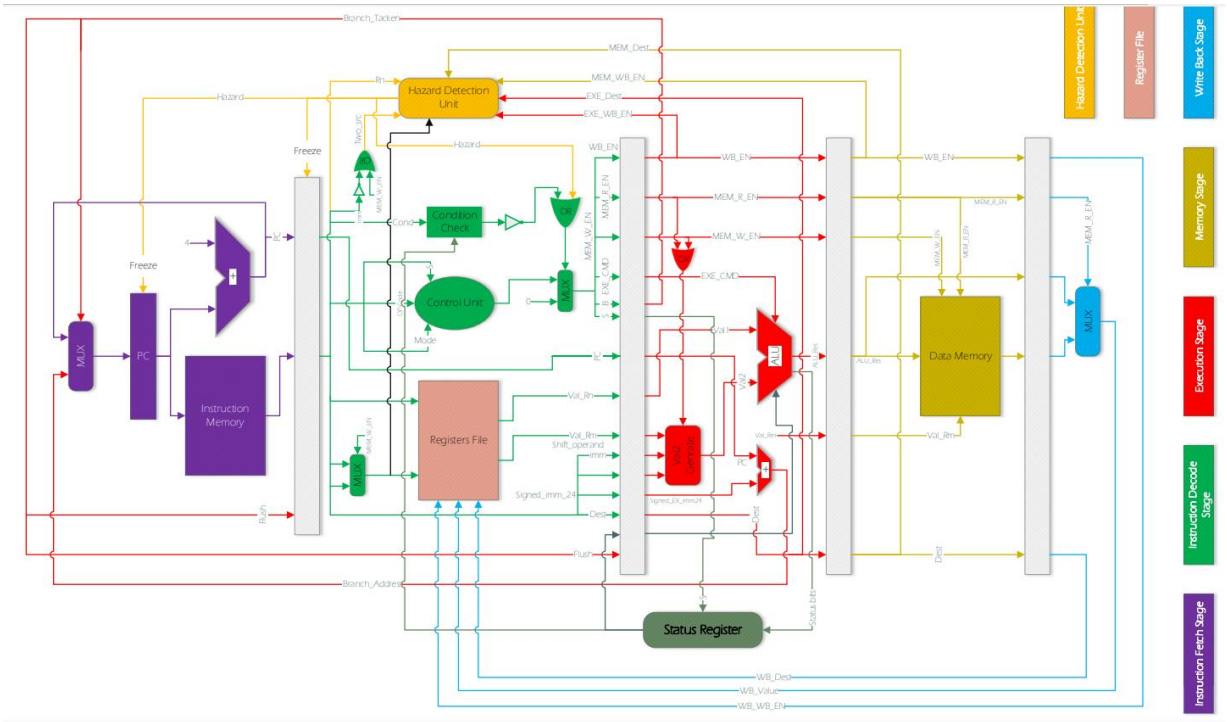
در این آزمایش با معماری پردازنده های ARM و نحوه اجرای دستوراتی که در این معماری پوشش داده می‌شوند را مشاهده کردیم.

این معماری شامل دستورات 32 بیتی می‌باشد و از معماری پایپ لاین برای اجرای دستورات استفاده می‌کند و شامل پنج مرحله (یا stage) می‌باشد که به ترتیب عبارت اند از:

- Instruction Fetch Stage (IF)
- Instruction Decode Stage (ID)
- Execution Stage (EXE)
- Memory Stage (MEM)
- Write Back Stage (WB)

علاوه بر این stage ها ماژول های hazard detection و status registers را هم داریم لازم به ذکر است که در جلسه اول فقط به پیاده سازی مرحله IF stage پرداختیم و بقیه مراحل را در جلسات آینده به پیاده سازی می‌پردازیم.

به طور کلی این معماری را می‌توان در بلوک دیاگرام زیر خلاصه نمود:

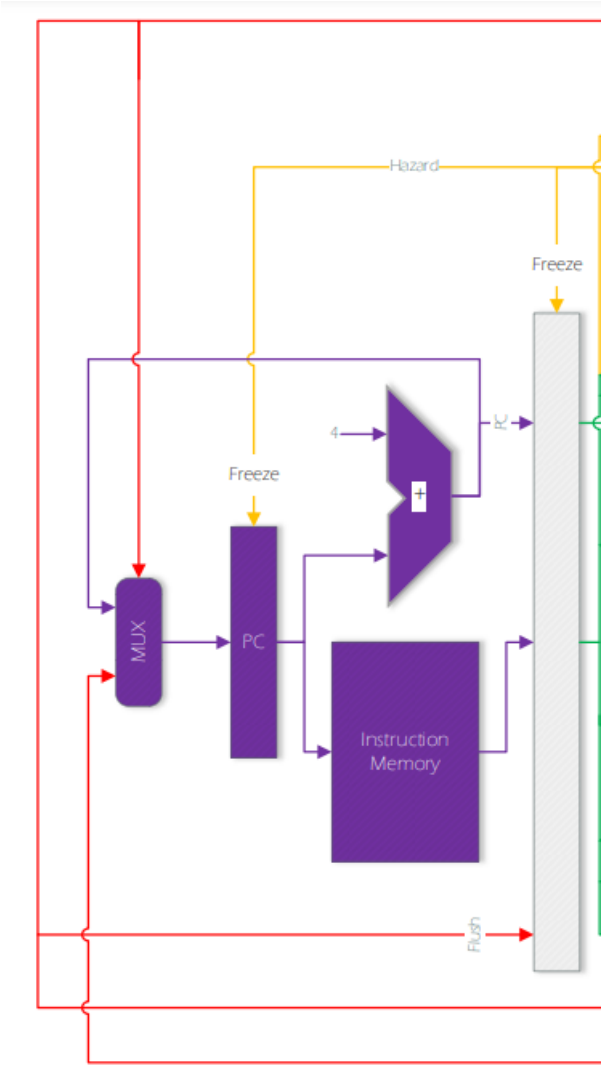


شکل 1

در جلسه اول به پیاده سازی IF stage پرداختیم.

شرح ماژول ها

در بخش IF stage دستور با شماره PC از instruction memory(`Ins_Mem`) خوانده می‌شود و اگر branch نداشتیم PC به تعداد چهار واحد زیاد می‌شود (قابل ذکر است که بخش برنچ در جلسات بعدی پیاده سازی میگردد) و در کلاک بعدی دستور بعدی خوانده شود. اگر از hazard detection سیگنال freeze مثبت بود، PC به مقدار چهار واحد زیاد نمی‌شود، چون پردازنده در stage های `IF, ID` نباید جلو برود (این عملیات نیز در جلسات بعد پیاده سازی می‌گردد) و بلوک دیاگرام آن به صورت زیر می‌باشد:



شکل 2

ماژول جمع کننده:

این ماژول وظیفه فراهم آوردن مقدار مرحله بعد PC را دارد. دو ورودی به صورت باینری (عدد صحیح) دارد.

یکی از آن دو ورودی عدد 4 و دیگری خروجی PC می‌باشد.(PC در هر مرحله در حالت عادی و بدون برنچ و ... به مقدار 4 واحد اضافه میگردد)

کد Adder ما به صورت زیر می‌باشد:

```
module Mux (
    input [31:0] a,
    b,
    input sel,
    output [31:0] c
);

    assign c = (sel ? b : a);

endmodule

module Adder (
    input  [31:0] a,
    b,
    output [31:0] res
);

    assign res = a + b;

endmodule

```

شکل 3

ماژول حافظه:

در اینجا برای پیاده سازی ماژول Ins_mem به صورت دستی مقدار دهی شده است و صرفا برای مثال 8 خانه 32 بیتی در نظر گرفته شده است(در مجموع 8 دستور) و با ورود هر کلاک به صورت آسنکرون (بدون سیگنال های کلاک و ریست) مقدار حافظه ای که PC به آن اشاره میکند را بر میگرداند. (پس یک ورودی PC و یک خروجی مقدار حافظه دارد که هر دو 32 بیتی می‌باشند)

کد مورد نظر به صورت زیر پیاده سازی شده است:

```
module Ins_Mem (
    input  [31:0] in,
    output [31:0] out
);

    reg [31:0] mem[6:0];

    initial begin
        mem[0] = 32'b000000_00001_00010_00000_00000000000;
        mem[1] = 32'b000000_00011_00100_00000_00000000000;
        mem[2] = 32'b000000_00101_00110_00000_00000000000;
        mem[3] = 32'b000000_00111_01000_00010_00000000000;
        mem[4] = 32'b000000_01001_01010_00011_00000000000;
        mem[5] = 32'b000000_01011_01100_00000_00000000000;
        mem[6] = 32'b000000_01101_01110_00000_00000000000;
    end

    assign out = mem[in>>2];

endmodule

```

شکل 4

ماژول مولتی پلکسر:

مولتی پلکسر برای سیگنال انتخاب بین خروجی جمع کننده (PC+4) و برنچ پیاده سازی شده است (در حال حاضر چون برنچ نداریم ورودی سلکتور آن صفر داده می‌شود) پس کد آن به صورت زیر می‌باشد:

```
module Adder (
    input  [31:0] a,
    b,
    output [31:0] res
);

    assign res = a + b;

endmodule

```

شکل 5

ماژول IF_Stage:

این ماژول که از ترکیب ماژول های بالا پیاده سازی شده است و بلوک دیگرام آن مانند شکل 2 می‌باشد. در این مرحله دستورات از مموری خانده شده و آماده برای خواندن دستور بعد می‌شود.

ورودی های آن علاوه بر سگنال کلاک و ریست، سیگنال های `freeze` و `branch_taken` و مقدار آدرس برنچ می‌باشد و خروجی آن نیز PC و دستور خوانده شده (instruction) می‌باشد. پس داریم:

```
module IF_Stage (
    input clk,
    rst,
    freeze,
    branch_taken,
    input [31:0] branch_address,
    output [31:0] PC,
    instruction
);

    wire [31:0] PC_reg_in;
    reg  [31:0] PC_reg_out;

    Mux mux (
        PC,
        branch_address,
        branch_taken,
        PC_reg_in
    );

    Adder pcAdder (
        PC_reg_out,
        4,
        PC
    );

    Ins_Mem instruction_mem (
        PC_reg_out,
        instruction
    );

    always @(posedge clk, posedge rst) begin
        if (rst) PC_reg_out <= 0;
        else if (~freeze) PC_reg_out <= PC_reg_in;
    end

endmodule

```

شکل 6

ماژول IF_Reg:

این ماژول وظیفه نگهداری از سیگنال ها و مقادیر خروجی از مرحله IF_Stage را دارا می‌باشد تا دیتا های مورد نیاز مرحله بعد یعنی ID_Stage فراهم باشد پس داریم:

```
module IF_Reg (
    input clk,
    rst,
    freeze,
    flush,
    input [31:0] PC_in,
    instruction_in,
    output reg [31:0] PC_out,
    instruction_out
);

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            PC_out <= 0;
            instruction_out <= 0;
        end else if (flush) begin
            PC_out <= 0;
            instruction_out <= 0;
        end else if (~freeze) begin
            PC_out <= PC_in;
            instruction_out <= instruction_in;
        end
    end

endmodule

```

شکل 7

حال مقادیر خروجی مورد نظرمان را از این مرحله در اختیار داریم ولی بهتر است این سیگنال ها را تا انتها (تا مرحله WB) اتقال دهیم به همین منظور چهار استیج دیگر نیز تعریف شده و صرفا وظیفه انتقال داده را بر عهده دارد.

```
module ID_Stage (
    input clk, rst,
    input[31:0] PC_in,
    output[31:0] PC_out
);

    always @(posedge clk, posedge rst) begin

    end

endmodule

```

شکل 8

و 4 مرحله دیگر نیز رجیسترهای خود برای ذخیره دیتا را نیز دارند برای مثال برای ID_reg داریم:

```
module ID_Reg (
    input clk,
    rst,
    freeze,
    flush,
    input [31:0] PC_in,
    instruction_in,
    output reg [31:0] PC_out,
    instruction_out
);

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            PC_out <= 0;
            instruction_out <= 0;
        end else if (flush) begin
            PC_out <= 0;
            instruction_out <= 0;
        end else if (~freeze) begin
            PC_out <= PC_in;
            instruction_out <= instruction_in;
        end
    end

endmodule

```

شکل 9

لازم به ذکر است که هر چهار مرحله به همین صورت پیاده سازی شده اند تا در جلسات بعد کامل شوند و صرفاوظیفه انتقال سیگنال ها را دارند. در گام بعد به پیاده سازی ماژول نهایی بعنی ماژول ARM خود پرداختیم تا این پنج استیج را به یک دیگر متصل کنیم و کد آن به شکل زیر نوشته شد:

```
module ARM (input clk, rst);
    wire branchTaken;
    wire[31:0] branchAddr;
    wire[31:0] PC_IF, Inst_IF;
    wire[31:0] PC_ID, Inst_ID;
    wire[31:0] PC_EXE, Inst_EXE;
    wire[31:0] PC_MEM, Inst_MEM;
    wire[31:0] PC_FINAL, Inst_FINAL;

    IF_Stage if_stage(.clk(clk), .rst(rst), .freeze(1'b0),
        .branchTaken(1'b0), .branchAddr(0), .PC(PC_IF),.instruction(Inst_IF));
    IF_Reg if_reg(.clk(clk), .rst(rst), .freeze(1'b0), .flush(1'b0),
        .PC_in(PC_IF), .instruction_in(Inst_IF), .PC(PC_ID),.instruction(Inst_ID));

    ID_Stage id_stage(.clk(clk), .rst(rst), .PC_in(PC_ID), .PC(PC_ID));
    ID_Reg id_reg( .clk(clk), .rst(rst), .freeze(1'b0), .flush(1'b0),
        .PC_in(PC_ID), .instruction_in(Inst_ID), .PC(PC_EXE), .instruction(Inst_EXE));

    EXE_Stage exe_stage( .clk(clk), .rst(rst), .PC_in(PC_EXE), .PC(PC_EXE));
    EXE_Reg exe_reg(.clk(clk), .rst(rst), .freeze(1'b0), .flush(1'b0), .PC_in(PC_EXE),
        .instruction_in(Inst_EXE), .PC(PC_MEM), .instruction(Inst_MEM));

    MEM_Stage mem_stage(.clk(clk), .rst(rst), .PC_in(PC_MEM), .PC(PC_MEM));

    MEM_Reg mem_reg(.clk(clk), .rst(rst), .freeze(1'b0), .flush(1'b0), .PC_in(PC_MEM),
        .instruction_in(Inst_MEM), .PC(PC_FINAL),.instruction(Inst_FINAL));
    WB_Stage wb_stage(.clk(clk), .rst(rst), .PC_in(PC_FINAL), .PC(PC_FINAL));
endmodule
```

شکل 10

همانطور که در شکل 10 نیز مشاهده میکند ماژول آرم ما از وصل سیگنال های پنج مرحله اشاره شده در بالا شکل میگیرد و نکته خاص دیگری ندارد.

SIMULATION:

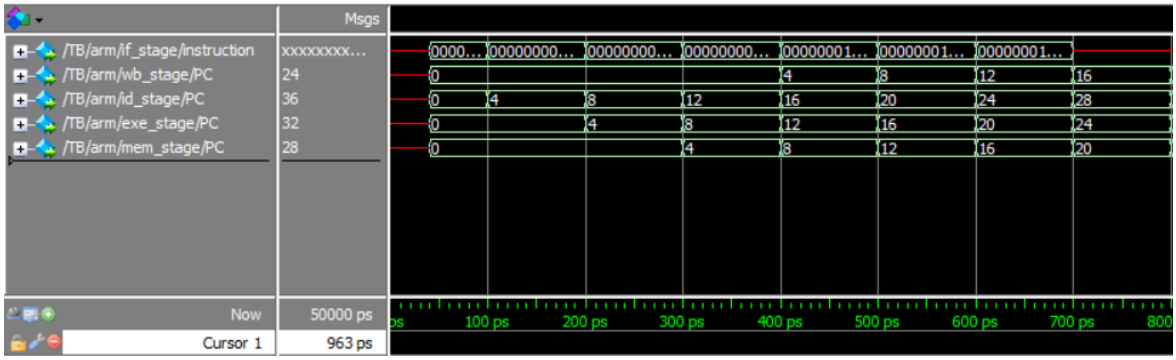
برای شبیه سازی نیاز بود تا یک تست بنچ بنویسیم و سیگنال کلاک را تولید کنیم و در ابتدا سیگنال ریست را نیز فعال کنیم(قابل ذکر است در همه ماژول ها سیگنال `reset` آسنکرون و بدون نیاز به کلاک پیاده سازی شده است). به همین منظور تست بنچ ما به صورت زیر می‌باشد که در آن یک اینستنس از ماژول arm خود که در شکل 10 نیز آن را مشاهده می‌کنیم ایجاد کردیم و کلاک و ریست ماژول آرم را نیز به آن دادیم و شروع به مشاهده سیگنال PC در هر استیج و همینطور instruction خوانده شده نیز کردیم.

کد تست بنچ:

```
module TB_DE2 ();

    reg clk, rst;

    ARM arm (
        .clk(clk),
        .rst(rst)
    );

    initial begin
        clk = 1;
        repeat (200) begin
            #50;
            clk = ~clk;
        end
    end

    initial begin
        rst = 0;
        #20 rst = 1;
        #10 rst = 0;
    end

endmodule

```

شکل 11

سپس شروع به اجرای simulation در مادل سیم کردیم:



شکل 12

همانطور که در تصویر خروجی نیز مشاهده میکنید مقدار PC در هر مرحله به اندازه چهار واحد اضافه میشود و مقدار دستور خوانده شده برای PC نیز مشاهده میگردد و همه چیز مطابق آن چه که انتظار داشتیم پیش رفت.

Synthesize on FPGA:

بعد از مرحله شبیه سازی در مادل سیم شروع به سنتز کردن کد بر روی FPGA(Intel) کردیم.

برای این کار ابتدا از فایلی که TA های کلاس در اختیارمان قرار داده بودند استفاده کردیم(فایل DE.V). در این فایل پایه های FPGA تعریف شده اند (شامل LED ها و SWITCH ها و ...).

در انتهای این فایل یک اینستنس از ماژول ARM خود تعریف کردیم و برای کلاک از سیگنال CLOCK_50 که یک سیگنا 50 مگاهرتز می‌باشد استفاده کردیم و برای ریست نیز از SWITCH[0] استفاده کردیم.

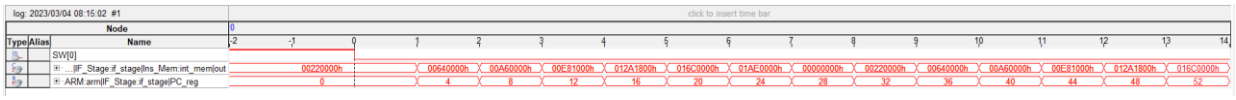
یعنی در انتهای این فایل بعد از تعریف GPIO ها داریم:

```
301  ////////////////////////    GPIO    ////////////////////////////////
302  inout   [35:0]  GPIO_0;                 //  GPIO Connection 0
303  inout   [35:0]  GPIO_1;                 //  GPIO Connection 1
304
305  ARM arm(
306          .clk(CLOCK_50),
307          .rst(SW[0])
308      );
309
```

شکل 13

سپس به کامپایل آن در نرم افزار Quartus پرداختیم و بعد از موفقیت آمیر بودن آن فایل ایجاد شده را بر روی FPGA آزمایشگاه سنتز کردیم.

برای بررسی عملکرد کد مربوطه از Signal Tab در داخل نرم افزار Quartus استفاده کردیم که به ما به راحتی این امکان را میداد تا خروجی های سیگنال های خود را داخل FPGA مشاهده کنیم و تقریبا به راحتی شبیه ساز در داخل مادل سیم این ابزار امکان دیدن سیگنال های هر مر حله را به ما داد:



شکل 14

تصویر بالا خروجی Signal Tab را به ما نشان میدهد که در آن مقدار PC و مقدار دستور خوانده شده نشان داده می‌شود. PC به مقدار چهار در هر کلاک بالا می‌رود و دستور خوانده شده نیز به ترتیب حافظه Ins_mem خوانده میشود و کاملا مطابق آن است که انتظار داشتیم.

جمع بندی:

در جلسه اول آزمایشگاه یک آشنایی کلی با معماری آرم و نحوه اجرای دستورات در آن پیدا کردیم و دیدیم که شامل 5 مرحاه اصلی برای اجرای یک دستور می‌باشد و از معماری پایپ لاین استفاده میکند و سپس شروع به ساختن مرحله اول آن یعنی IF کردیم. در این مرحله بعد از خواندن هر دستور PC چهار واحد اضافه میشود و اینستراکشن مربوطه به مرحله بعد فراستاده میشود. در این جلسه این مرحله ابتدا شبیه سازی شد(شکل 12) و سپس بر روی FPGA ازمایشگاه سنتز شد و با استفاده از Signal Tab (شکل 14) خروجی مورد نظر بررسی و صحت آن نیز تایید شد.

مشکلات:

یکی از مشکلاتی که حین آزمایش داشتیم، نگرفتن خروجی درست در Signal Tab بود. چون تریگر را pos edge ریست قرار دادیم، حین simulation اتفاقی نمی‌افتاد. که با تغییر تریگر، این مشکل حل شد.