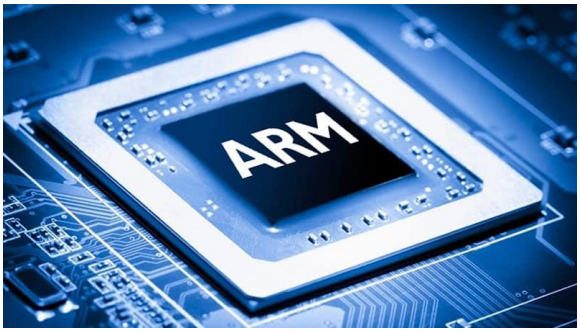
به نام خدا

دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



آزمایشگاه معماری کامپیوتر

گزارش دستور کار شماره 3

علی پادیاو
810199388
محمد صالح عرفاتی
810197543

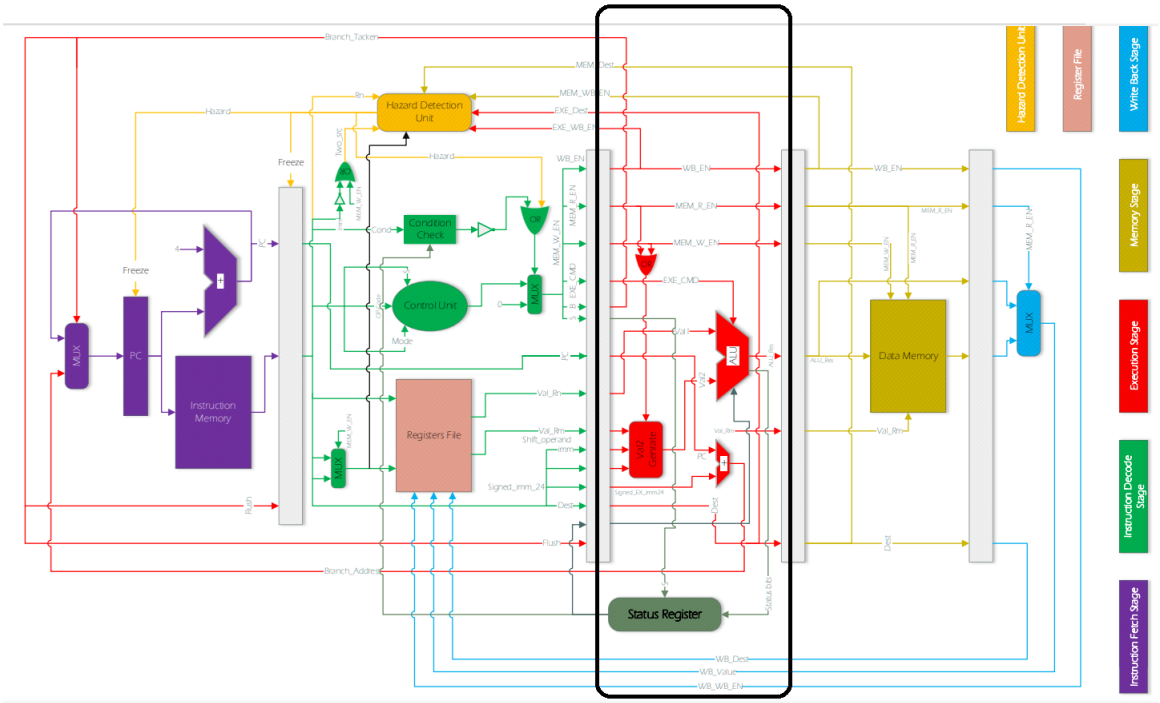
فروردین 1402

مقدمه

در آزمایشات قبلی مراحل IF و ID از معماری ARM9 پیاده سازی شد.

حال در این آزمایش به پیاده سازی مرحله EXE پرداختیم.

بلوک دیاگرام این مرحله به صورت زیر می باشد:



شکل 1

پس از واکشی دستورات در مرحله قبل و به دست آوردن رجیستر های ورودی و فهمیدن نوع دستور خواسته شده، نوبت آن رسیده است تا دستور مورد نظر اجرا گردد.

دستوراتی که در شرح آزمایش گفته شده اند با استفاده از ماژول ALU محاسبه میشوند که ورودی آن یکی RN از مرحله قبل و دیگری خروجی ماژول value2_generator که از خروجی مرحله ID به دست می آید، می باشد.

قابل ذکر است که اگر دستور از نوع برنچ باشد آدرس برنچ نیز باید محاسبه شود پس به همین دلیل یک ماژول Adder نیز داریم تا با استفاده از آن آدرس برنچ را محاسبه کنیم.

و در انتها Status Register را داریم که ورودی condition check را فراهم میکند.

شرح ماژول ها:

1- ALU

کد:

```
module ALU (
    input[31:0] input1, input2,
    input carry_in,
    input[3:0] command,

    output reg[31:0] out,
    output reg carry_out, V,
    output N, Z
);

    always @ (*) begin
        out = 32'b0;
        carry_out = 1'b0;
        case (command)
            4'b0001: out = input2;
            4'b1001: out = ~input2;
            4'b0010: {carry_out, out} = input1 + input2;
            4'b0011: {carry_out, out} = input1 + input2 + carry_in;
            4'b0100: {carry_out, out} = input1 - input2;
            4'b0101: {carry_out, out} = input1 - input2 - 1 + carry_in;
            4'b0110: out = input1 & input2;
            4'b0111: out = input1 | input2;
            4'b1000: out = input1 ^ input2;
            default: {carry_out, out} = 33'b0;
        endcase
    end

    assign N = out[31];
    assign Z = (out == 32'b0);

    always @(*) begin
        V = 1'b0;
        case (command)
            4'b0010: V = (input1[31] & input2[31] & (~N)) || ( (~input1[31]) & (~input2[31]) & N);
            4'b0011: V = (input1[31] & input2[31] & (~N)) || ( (~input1[31]) & (~input2[31]) & N);
            4'b0100: V = (input1[31] & (~input2[31]) & (~N)) || ( (~input1[31]) & input2[31] & N);
            4'b0101: V = (input1[31] & (~input2[31]) & (~N)) || ( (~input1[31]) & input2[31] & N);
            default: V = 1'b0;
        endcase
    end

endmodule
```

این ماژول بر اساس سیگنال ورودی 4 بیتی command نوع عملگر مربوطه را انتخاب میکند، به طور مثال اگر این سیگنال برابر 0010 بود، آن گاه عملگر جمع برروی دو ورودی داده شده انجام میشود.

4 سیگنال دیگر نیز در این ماژول تولید میشوند:

N: بیان گر آن است که آیا جواب نهایی منفی می باشد یا خیر (با بیت ساین متوجه میشویم و اگر بیت ساین یا به عبارت دیگر بیت 31 برابر یک باشد آن گاه N نیز برابر یک میشود).

Z: بیانگر آن است که آیا خروجی برابر عدد صفر می باشد یا خیر.

C: به معنای carry میباشد که تنها در عملیات جمع و تفریق تولید میشوند.

V: نشان دهنده رخ دادن overflow در سیستم می باشد.

قابل ذکر است که لزومی ندارد ماژول ALU ما با کلاک کار کند وتنها وظیفه آن اجرای عملگر می باشد، به همین دلیل ورودی reset و clk را دارا نمی باشد.

جدای از این ورودی و خروجی ها یک ورودی carry_in دارد که اگر باید علاوه بر جمع دو ورودی با کری نیز جمع میشد، این عملیات نیز پیاده سازی شود.

2- VALUE2_GENERATOR:

وظیفه این ماژول تهیه ورودی دوم ماژول ALU می باشد که از بین RM و سیگنال Immediate و یک offset 12 بیتی برای دستورات load و store، یک سیگنال توسط ورودی mem انتخاب میشوند.

در صورت یک بودن mem دستور load و یا store می باشد و به همین دلیل آفست 12 بیتی انتخاب میشود و ساین اکستند می شود.

حال اگر دستور لود و یا استور نباشد بر اساس بیت immediate انتخاب میشود که آیا مقدار immediate ویا مقدار RM توسط این ماژول انتخاب شود و به عنوان ورودی دوم به ماژول ALU فرستاده شود.

کد موردنظر به صورت زیر می باشد:

```
module VALUE2_Generator (
    input[11:0] Shift_operand,
    input[31:0] RM_value,
    input imm, mem,
    output reg[31:0] Val2
);

    wire[7:0] immed_8 = Shift_operand[7:0];
    wire[3:0] rotate_imm = Shift_operand[11:8];
    wire[4:0] shift_imm = Shift_operand[11:7];
    wire[1:0] shift = Shift_operand[6:5];
    reg[63:0] temp_64;

    always @ (*) begin
        temp_64 = 64'b0;
        Val2 = 32'b0;
        if (mem)
            Val2 = { {20{Shift_operand[11]}}, Shift_operand };
        else if (imm) begin
            temp_64[39:32] = immed_8;
            temp_64 = temp_64 >> (2*rotate_imm);
            Val2 = temp_64[31:0] | temp_64[63:32];
        end else begin
            temp_64[63:32] = RM_value;
            if (shift == 2'b11) begin
                temp_64[63:32] = RM_value;
                temp_64 = temp_64 >> shift_imm;
                Val2 = temp_64[31:0] | temp_64[63:32];
            end else if (shift == 2'b01)
                Val2 = RM_value >> shift_imm;
            else if (shift == 2'b10)
                Val2 = RM_value >>> shift_imm;
            else
                Val2 = RM_value << shift_imm;
        end
    end

endmodule
```

دقت شود که اگر RM انتخاب شود بر اساس جدول توسط 4 بیت shift_imm خالت شیفت انخاب می شود:

وضعیت شیفت	توضیحات	مقدار
LSL	Logical shift left	00
LSR	Logical shift right	01
ASR	Arithmetic shift right	10
ROR	Rotate right	11

3- ADDER:

ماژول Adder که در قبل (در مرحله IF) پیاده سازی شده بود در ا ین مرحله نیز مورد استفاده قرار گرفته است.

وظیفه این ماژول محاسبه branchAddress می باشد و ورودی آن Signed_EX_imm24 و ورودی دیگر آن PC می باشد.

کد مربوطه:

```
12  module Adder (
13      input  [31:0] a, b,
14      output [31:0] res
15    );
16
17    assign res = a + b;
18
19  endmodule
```

4- OR:

قابل ذکر است که ماژول OR نیز با صرفا یک خط ساده در ماژول EXE_stage پیاده سازی شده است.

```
17      wire mem = MEM_R_EN || MEM_W_EN;
```

5- Status Register:

این ماژول وظیفه نگهداری 4 بیت N,V,C,Z که توسط ماژول ALU تولید میشوند را دارد تا در کلاک بعدی به ماژول condition_check ارسال کند. قابل ذکر است که اگر ورودی S برابر یک باشد این ذجیستر تغییر می کند.

کد مربوطه:

```
module StatusRegister (
    input clk, rst,
    input [3:0] status_in,
    input S,
    output reg [3:0] status_out
  );

  always @(negedge clk, posedge rst)
  begin
    if (rst)
      status_out <= 0;
    else if (S)
      status_out <= status_in;
  end

endmodule
```

5-EXE Stage:

سپس به متصل کردن اجزای EXE Stage نمودیم.

کد مربوطه به صورت زیر می باشد:

```
module EXE_Stage (
    input clk, rst,
    input[3:0] EXE_CMD,
    input MEM_R_EN, MEM_W_EN,
    input[31:0] PC,
    input[31:0] Val_Rm, Val_Rn,
    input imm,
    input[11:0] Shift_operand,
    input[23:0] Signed_imm_24,
    input[3:0] status_IN,

    output[31:0] ALU_res, Br_addr,
    output[3:0] status
);

    wire[31:0] Signed_imm_32 = { {6{Signed_imm_24[23]}}, Signed_imm_24, 2'b00};
    wire mem = MEM_R_EN || MEM_W_EN;
    wire[31:0] Val2;

    VALUE2_Generator value2_generator(
        .Shift_operand(Shift_operand),
        .RM_value(Val_Rm),
        .imm(imm),
        .mem(mem),
        .Val2(Val2)
    );

    ALU alu(
        .input1(Val_Rn),
        .input2(Val2),
        .carry_in(status_IN[1]),
        .command(EXE_CMD),
        .out(ALU_res),
        .carry_out(status[1]),
        .V(status[0]),
        .Z(status[2]),
        .N(status[3])
    );

    Adder adder(
        .a(PC),
        .b(Signed_imm_32),
        .res(Br_addr)
    );

endmodule
```

6-EXE Reg:

سپس برای ذخیره رجیستر های مرحله ی EXE ماژول EXE_reg را ایجاد کردیم تا ورودی های مرحله بعد را فراهم کند.

کد مربوطه:

```
module EXE_Reg (
    input clk, rst, WB_en_in, MEM_R_EN_in, MEM_W_EN_in,
    input[31:0] ALU_res_in, ST_val_in,
    input[3:0] Dest_in,

    output reg WB_en, MEM_R_EN, MEM_W_EN,
    output reg[31:0] ALU_res, ST_val,
    output reg[3:0] Dest
);

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            ALU_res <= 32'b0;
            ST_val <= 32'b0;
            Dest <= 4'b0;
            WB_en <= 1'b0;
            MEM_R_EN <= 1'b0;
            MEM_W_EN <= 1'b0;
        end else begin
            ALU_res <= ALU_res_in;
            ST_val <= ST_val_in;
            Dest <= Dest_in;
            WB_en <= WB_en_in;
            MEM_R_EN <= MEM_R_EN_in;
            MEM_W_EN <= MEM_W_EN_in;
        end
    end

endmodule
```

6-WB Stage:

در انتها ماژول WB Stage را پیاده سازی کردیم که تنها دارای یک مالتی پلکسر می باشد:

```
module WB_Stage (
    input clk, rst, MEM_R_EN,
    input[31:0] ALU_res, mem_out,

    output[31:0] WB_value
);

    Mux mux(
        .a(ALU_res),
        .b(mem_out),
        .sel(MEM_R_EN),
        .c(WB_value)
    );

endmodule
```

دستورات برنامه:

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

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

  initial
  begin
    mem[0] = 32'b1110_00_1_1101_0_0000_0000_000000010100;
    mem[1] = 32'b1110_00_1_1101_0_0000_0001_101000000001;
    mem[2] = 32'b1110_00_1_1101_0_0000_0010_000100000011;
    mem[3] = 32'b1110_00_0_0100_1_0010_0011_000000000010;
    mem[4] = 32'b1110_00_0_0101_0_0000_0100_000000000000;
    mem[5] = 32'b1110_00_0_0010_0_0100_0101_000100000100;
    mem[6] = 32'b1110_00_0_0110_0_0000_0110_000010100000;
    mem[7] = 32'b1110_00_0_1100_0_0101_0111_000101000010;
    mem[8] = 32'b1110_00_0_0000_0_0111_1000_000000000011;
    mem[9] = 32'b1110_00_0_1111_0_0000_1001_000000000110;
    mem[10] = 32'b1110_00_0_0001_0_0100_1010_000000000101;
    mem[11] = 32'b1110_00_0_1010_1_1000_0000_000000000110;
    mem[12] = 32'b0001_00_0_0100_0_0001_0001_000000000001;
    mem[13] = 32'b1110_00_0_1000_1_1001_0000_000000001000;
    mem[14] = 32'b0000_00_0_0100_0_0010_0010_000000000010;
    mem[15] = 32'b1110_00_1_1101_0_0000_0000_101100000001;
    mem[16] = 32'b1110_01_0_0100_0_0000_0001_000000000000;
    mem[17] = 32'b1110_01_0_0100_1_0000_1011_000000000000;
  end

  assign out = mem[in>>2];

endmodule
```

طبق شرح آزمایش 18 دستور اول برنامه محک را داخل Ins_Mem ذخیره کردیم.

SIMULATION:

برای simulation از تست بنچ زیر استفاده شده است:

```
module TB();

  reg clk, rst;

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

  initial begin
      rst = 0;
      #40;
      rst = 1;
      #40
      rst = 0;
  end

  initial begin
      clk = 1;
      repeat(1000) begin
          #50;
          clk = ~clk;
      end
  end


endmodule
```

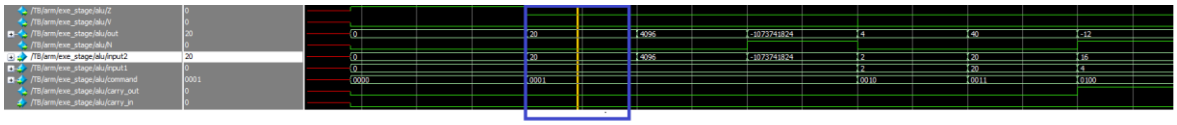
دستور 1:

در دستور اول باید مقدار 20 در رجیستر 0 نوشته شود:

MOV R0,#20

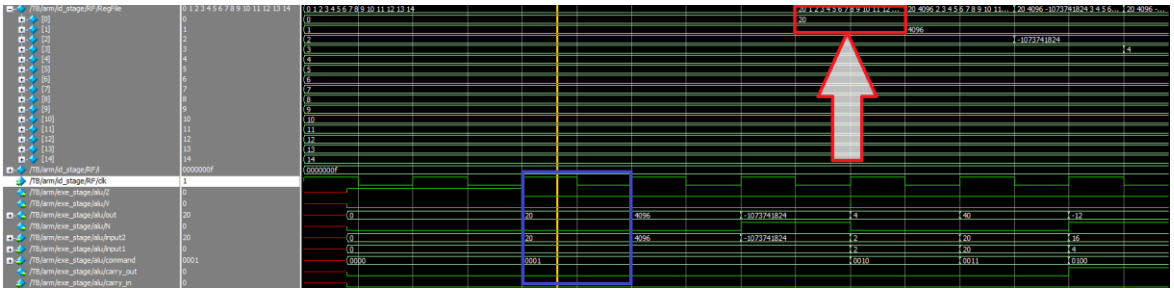
32'b1110_00_1_1101_0_0000_0000_000000010100

نگاهی به ماژول ALU می اندازیم:



همانطور که مشاهده میشود ورودی اول برابر صفر و ورودی دوم برابر 20 و کامند مربوطه نیز 0001 می باشد که بیانگر عملگر result = in2 می باشد.

پس همانطور که مشاهده نیز میشود خروجی نیز برابر عدد 20 می باشد. حال در سه کلاک بعد این مقدار باید در رجیستر شماره صفر نوشته شود پس در سه کلاک بعد نگاهی به register file می اندازیم:



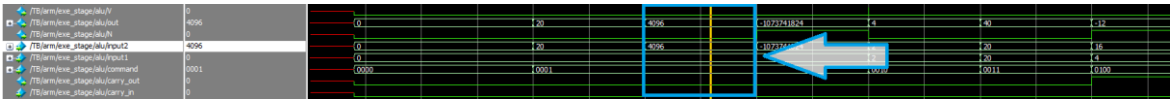
همانطور که مشاهده می شود مقدار 20 نیز در رجیستر شماره صفر نوشته میشود.

دستور 2:

MOV R1,#4096

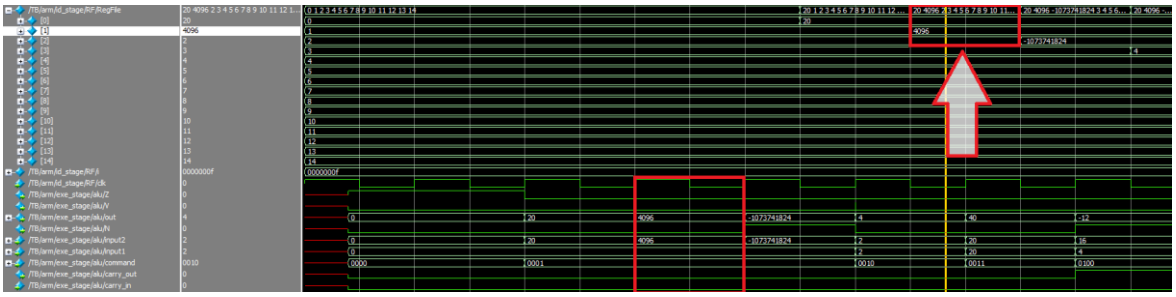
32'b1110_00_1_1101_0_0000_0001_101000000001

نگاهی به ماژول ALU می اندازیم:



همانطور که مشاهده میشود ورودی اول برابر صفر و ورودی دوم برابر 4096 و کامند مربوطه نیز 0001 می باشد که بیانگر عملگر result = in2 می باشد.

پس همانطور که مشاهده نیز میشود خروجی نیز برابر عدد 4096 می باشد. حال در سه کلاک بعد این مقدار باید در رجیستر شماره یک نوشته شود پس در سه کلاک بعد نگاهی به register file می اندازیم:



همانطور که مشاهده می شود مقدار 4096 نیز در رجیستر شماره یک نوشته میشود.

دستور 3:

MOV R2,# 0xC0000000 = > R2 = -1073741824

32'b1110_00_1_1101_0_0000_0010_000100000011

نگاهی به ماژول ALU می اندازیم:

همانطور که مشاهده میشود ورودی اول برابر صفر و ورودی دوم برابر 1073741824- و کامند مربوطه نیز 0001 می باشد که بیانگر عملگر result = in2 می باشد.

پس همانطور که مشاهده نیز میشود خروجی نیز برابر عدد 1073741824- می باشد. حال در سه کلاک بعد این مقدار باید در رجیستر شماره دو نوشته شود پس در سه کلاک بعد نگاهی به register file می اندازیم:

همانطور که مشاهده می شود مقدار 1073741824- نیز در رجیستر شماره دو نوشته میشود.

دستور 4:

//ADDS R3 ,R2,R2 => R3 = -2147483648

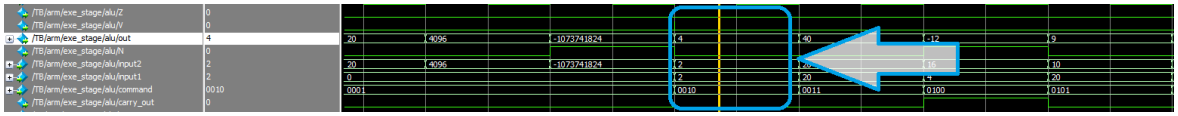
32'b1110_00_0_0100_1_0010_0011_000000000010

دقت شود که این دستور نیاز به freeze دارد زیرا از رجیستر R2 استفاده میکند و این رجیستر از دستور قبل باید

در داخل رجیستر فایل نوشته شود و چون در این فاز از پروژه hazard detection پیاده سازی نشده پس در این فاز به اشتباه مقدار قبلی این رجیستر یعنی مقدار دو خوانده میشود.

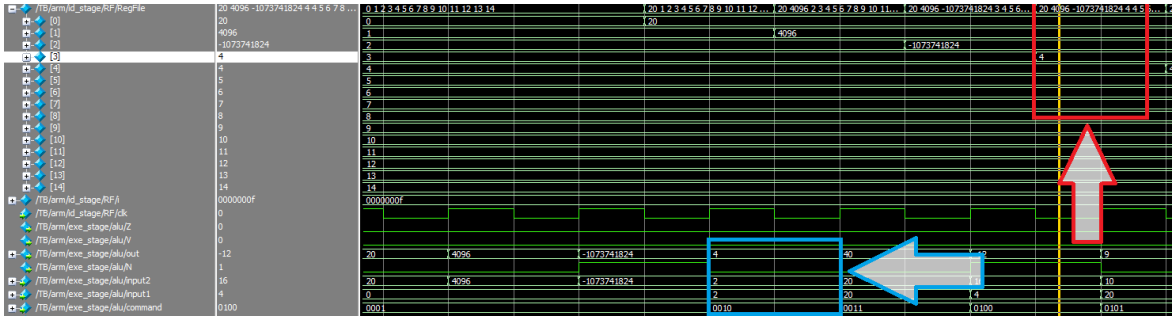
باید دو برابر مقدار رجیستر R2 در R3 نوشته شود:

نگاهی به ماژول ALU می اندازیم:



همانطور که مشاهده میشود چون ماژول hazard detection را در این فاز از پروژه در اختیار نداریم ورودی اول و ورودی دوم مقدار 2 و کامند مربوطه نیز 0010 می باشد که بیانگر عملگر result = in1 + in2 می باشد.

پس همانطور که مشاهده نیز میشود خروجی برابر عدد 4 می باشد. حال در سه کلاک بعد این مقدار باید در رجیستر شماره سه نوشته شود پس در سه کلاک بعد نگاهی به register file می‌اندازیم:



همانطور که مشاهده می شود مقدار 4 نیز در رجیستر شماره سه نوشته میشود.

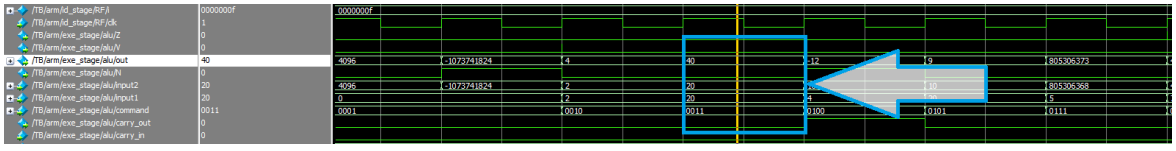
دستور 5:

//ADC R4 ,R0,R0 => R4 = 41

32'b1110_00_0_0101_0_0000_0100_000000000000

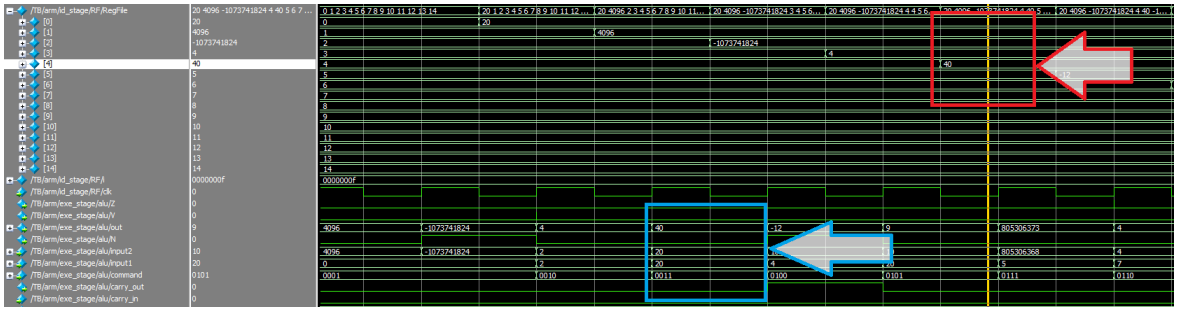
دقت شود که در دستور قبل باید سیگنال carry رخ میداد ولی چون hazard detection نداشتیم به اشتباه مقدار جواب بدست آمد و بیت carry به اشتباه صفر مقدار دهی میشود.

نگاهی به ماژول ALU می اندازیم:



مقدار هر دو ورودی ALU و یا به عبارتی R0 به درستی 20 خوانده میشود و کامند مربوطه نیز 0011 می باشد که بیانگر result = in1 + in2 + C می باشد.

حال در سه کلاک بعد این مقدار باید در رجیستر شماره چهار نوشته شود پس در سه کلاک بعد نگاهی به register file می اندازیم:

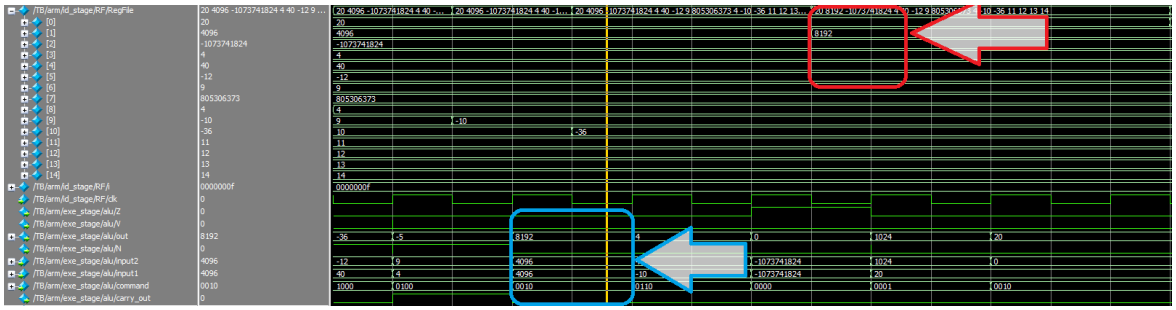


دستور 13:

//ADDNE R1 ,R1,R1 => R1 = 8196

32'b0001_00_0_0100_0_0001_0001_000000000001

نگاهی به خروجی بیاندازیم:



همانطور که مشاهده میشود ALU به درستی رجیستر یک را به عنوان هر دو ورودی استفاده کرده و چون کامند برابر 0010 می باشد، داریم:

result = in1 + in2

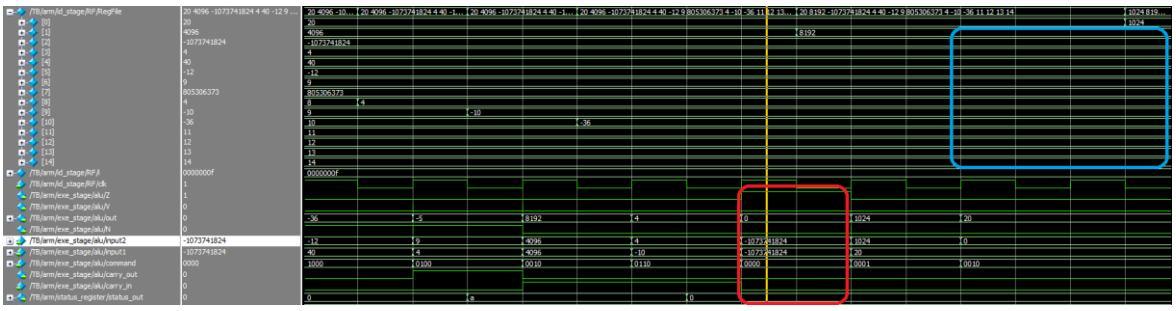
پس همانطور که مشاهده می شود به درستی مقدار 8196 در رجیستر R1 نوشته میشود.

دستور 15:

//ADDEQ R2 ,R2,R2 => R2 = - 1073741824

32'b0000_00_0_0100_0_0010_0010_000000000010

خروجی را مشاهده کنیم:

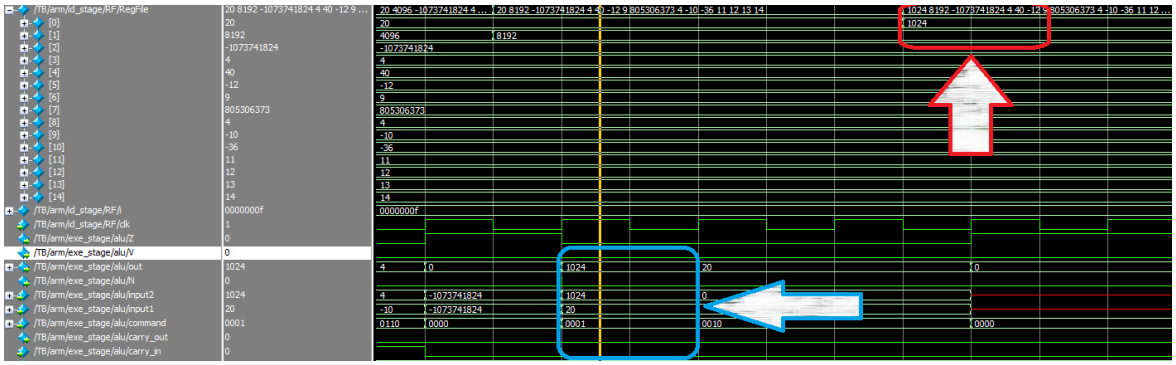


دستور 16:

//MOV R0 ,#1024 => R0 = 1024

32'b1110_00_1_1101_0_0000_0000_101100000001

خروجی:



همان طور که انتظار داشتیم بعد از 3 کلاک از محاسبه ALU مقدار 1024 در رجیستر R0 مینشیند.

پس همه دستوراتی بدون وابستگی داده ای بودند به طور کامل به درستی کار میکردند و آن دستوراتی نیز که وابستگی داده ای داشتند فاقد از hazard detection آن ها نیز عملکرد قابل پیش بینی داشتند.