علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

در این گزارش کار قصد داریم به بررسی جنبه های مختلف طراحی و پیاده سازی پردازنده ی ARM به طور عادی و با اضافه کردن Forwarding، SRAM و Cache و مقایسه ی نحوه ی عملکرد آن ها بپردازیم.

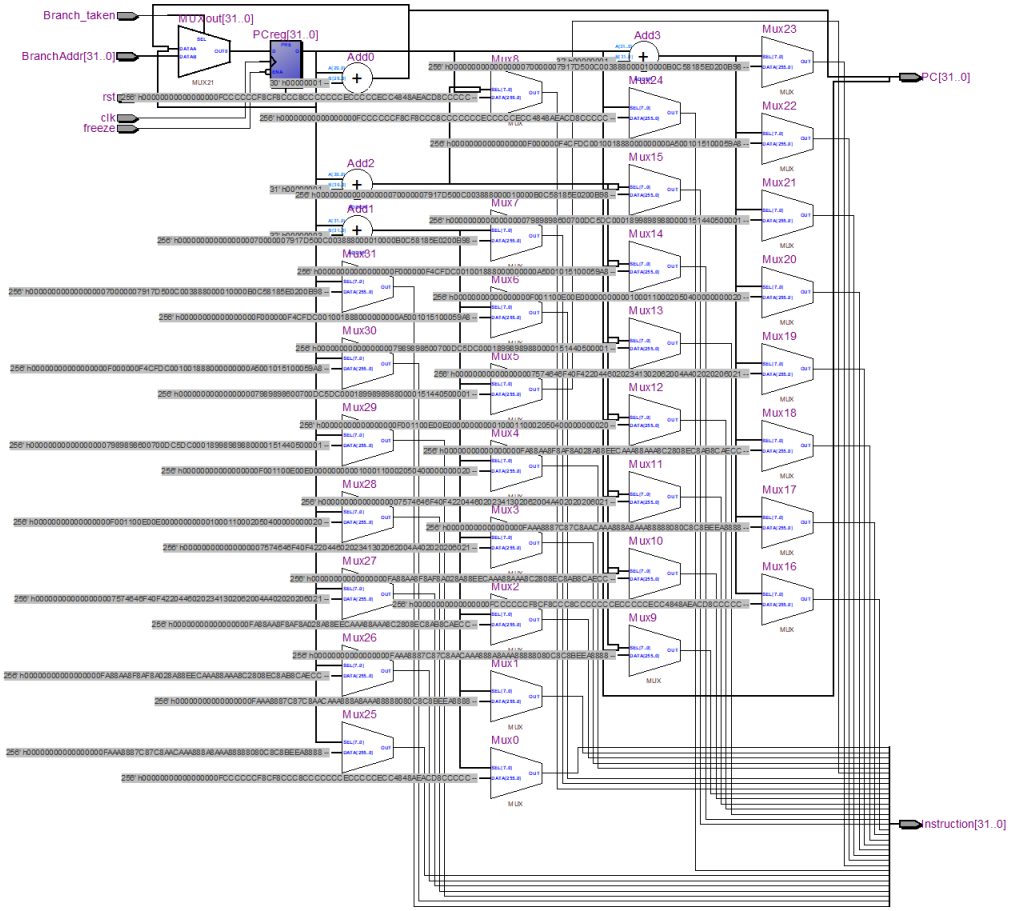
ARM

به منظور ارائه توضیحات پردازنده ARM، ابتدا به بررسی بخش های اصلی سازنده ی این پردازنده میپردازیم:

Instruction Fetch Stage (IF)

در این مرحله، یک شمارنده با نام PC (Program Counter) داریم که آدرس دستوری که باید از Instruction Memory واکشی شود را تولید میکند. در اجرای بدون پرش و اگر دستور کنترلی freeze صادر نشود، مقدار خروجی PC در هر clock cycle به میزان 4 واحد افزایش میابد. اگر دستور پرش داشته باشیم و شرط آن برقرار باشد، خروجی PC آدرس دستوری را به خود میگیرد که باید به آن پرش کنیم و اگر شرط پرش برقرار نباشد، مانند قبل، در هر کلاک 4 واحد افزایش میابد. با صدور دستور freeze، مقدار خروجی PC تغییر نمیکند.

در تصویر زیر، شمای RTL این بخش را مشاهده میکنید:



همانطور که میبینید، به آنکه instruction memory را فقط مقدار اولیه دادم و هیچ مکانیزم نوشتنی برای آن در کد قرار ندادم(که نیاز باشد با لبه ی کلاک در رجیستر بنویسد)، ابزار سنتز instruction memory را به صورت مالتیپلکسری پیاده سازی کرده است. من Instruction Memory را به صورت وایر تعریف کردم، اما اگر در کد به صورت reg هم تعریف کنید باز هم نتایجی مشابه با همین سنتز مشاهده خواهید کرد.

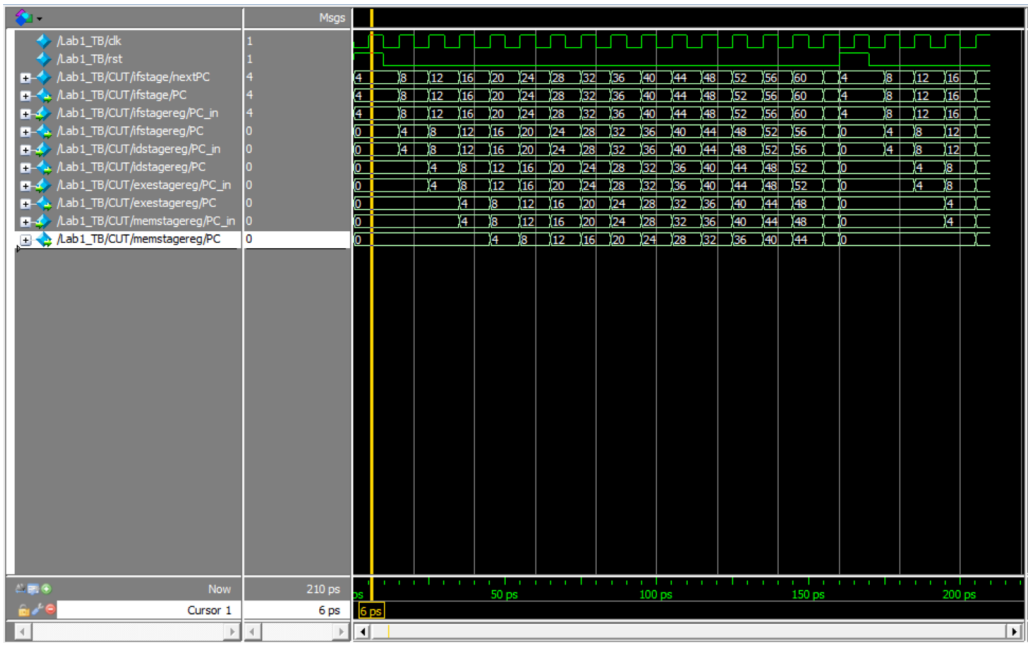
علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

کد آن را نیز در زیر مشاهده میفرمایید:

```
module IF_Stage(
  input clk,rst,freeze,Branch_taken,
  input[31:0] Branchaddr,
  output[31:0] PC, Instruction
);

  wire[7:0] InsMem [0:187];
  wire[31:0] nextPC,MUXout;
  reg[31:0]PCreg=0;
  always@(posedge clk, posedge rst)begin
    if(rst)
      PCreg<=0;
    else if(~freeze)
      PCreg<=MUXout;
  end
  assign PC=nextPC;
  assign nextPC=PCreg+4;
  assign MUXout=Branch_taken ? BranchAddr : nextPC;
  assign Instruction={InsMem[PCreg+3],InsMem[PCreg+2],InsMem[PCreg+1],InsMem[PCreg+0]};

          assign {InsMem[3], InsMem[2], InsMem[1], InsMem[0]} = 32'b1110_00_1_1101_0_0000_0000_000000010100;      //  MOV R0 = 20          R0 = 20
          assign {InsMem[7], InsMem[6], InsMem[5], InsMem[4]} = 32'b1110_00_1_1101_0_0000_0001_10100000000;      //  MOV R1 ,#4096        R1 = 4096
          assign {InsMem[11], InsMem[10], InsMem[9], InsMem[8]} = 32'b1110_00_1_1101_0_0000_0010_00010000011;   //  MOV R2 ,#0xC0000000    R2 = -1073741824
          assign {InsMem[15], InsMem[14], InsMem[13], InsMem[12]} = 32'b1110_00_0_0100_1_0010_0011_00000000010; //  ADDS R3 ,R2,R2       R3 = -2147483648
          assign {InsMem[19], InsMem[18], InsMem[17], InsMem[16]} =  32'b1110_00_0_0101_0_0000_0100_00000000000; //ADC R4 ,R0,R0 //R4 = 41
          assign {InsMem[23], InsMem[22], InsMem[21], InsMem[20]} = 32'b1110_00_0_0010_0_0100_0101_00010000100; //SUB R5 ,R4,R4,LSL #2 //R5 = -123
          assign {InsMem[27], InsMem[26], InsMem[25], InsMem[24]} = 32'b1110_00_0_0110_0_0000_0110_00001010000; //SBC R6 ,R0,R0,LSR #1//R6 = 10
          assign {InsMem[31], InsMem[30], InsMem[29], InsMem[28]} = 32'b1110_00_0_1100_0_0101_0111_00010100010; //ORR R7,R5,R2,ASR #2//R7 = -123
          assign {InsMem[35], InsMem[34], InsMem[33], InsMem[32]} = 32'b1110_00_0_0000_0_0111_1000_00000000011; //AND R8,R7,R3   R8 = -2147483648
          assign {InsMem[39], InsMem[38], InsMem[37], InsMem[36]} = 32'b1110_00_0_1111_0_0000_1001_00000000110; //MVN R9,R6//R9 = -11
          assign {InsMem[43], InsMem[42], InsMem[41], InsMem[40]} = 32'b1110_00_0_0001_0_0100_1010_00000000101; //EOR R10,R4,R5//R10 = -84
          assign {InsMem[47], InsMem[46], InsMem[45], InsMem[44]} = 32'b1110_00_0_1010_1_1000_0000_00000000110; //CMP R8,R6
          assign {InsMem[51], InsMem[50], InsMem[49], InsMem[48]} = 32'b0001_00_0_0100_0_0001_0001_00000000001; //ADDNE R1,R1,R1//R1 = 8192
          assign {InsMem[55], InsMem[54], InsMem[53], InsMem[52]} = 32'b1110_00_0_1000_1_1001_0000_00000001000; //TST R9,R8
          assign {InsMem[59], InsMem[58], InsMem[57], InsMem[56]} = 32'b0000_00_0_0100_0_0010_0010_00000000010; //ADDEQ R2 ,R2,R2   //R2 = -1073741824
          assign {InsMem[63], InsMem[62], InsMem[61], InsMem[60]} = 32'b1110_00_1_1101_0_0000_0000_10110000001; //MOV R0,#1024//R0 = 1024
          assign {InsMem[67], InsMem[66], InsMem[65], InsMem[64]} = 32'b1110_01_0_0100_0_0000_0001_00000000000; //STR R1,[R0],#0//MEM[1024] = 8192
          assign {InsMem[71], InsMem[70], InsMem[69], InsMem[68]} = 32'b1110_01_0_0100_1_0000_1011_00000000000; //LDR R11,[R0],#0//R11 = 8192
          assign {InsMem[75], InsMem[74], InsMem[73], InsMem[72]} = 32'b1110_01_0_0100_0_0000_0010_00000000100; //STR R2,[R0],#4//MEM[1028] = -1073741824
          assign {InsMem[79], InsMem[78], InsMem[77], InsMem[76]} = 32'b1110_01_0_0100_0_0000_0011_00000001000; //STR    R3 ,[R0],#8//MEM[1032] = -2147483648
          assign {InsMem[83], InsMem[82], InsMem[81], InsMem[80]} = 32'b1110_01_0_0100_0_0000_0100_00000001101; //STR    R4 ,[R0],#13    //MEM[1036] = 41
          assign {InsMem[87], InsMem[86], InsMem[85], InsMem[84]} = 32'b1110_01_0_0100_0_0000_0101_00000010000; //STR R5,[R0],#16//MEM[1040] = -123
          assign {InsMem[91], InsMem[90], InsMem[89], InsMem[88]} = 32'b1110_01_0_0100_0_0000_0110_00000010100; //STR R6,[R0],#20 //MEM[1044] = 10
          assign {InsMem[95], InsMem[94], InsMem[93], InsMem[92]} = 32'b1110_01_0_0100_1_0000_1010_00000000100; //LDR R10,[R0],#4 //R10 = -1073741824
          assign {InsMem[99], InsMem[98], InsMem[97], InsMem[96]} = 32'b1110_01_0_0100_0_0000_0111_00000011000; //STR R7,[R0],#24    //MEM[1048] = -123
          assign {InsMem[103], InsMem[102], InsMem[101], InsMem[100]} = 32'b1110_00_1_1101_0_0000_0001_00000000100; //MOV R1,#4//R1 = 4
          assign {InsMem[107], InsMem[106], InsMem[105], InsMem[104]} = 32'b1110_00_1_1101_0_0000_0010_00000000000; //MOV R2,#0//R2 = 0
          assign {InsMem[111], InsMem[110], InsMem[109], InsMem[108]} = 32'b1110_00_1_1101_0_0000_0011_00000000000; //MOV R3,#0//R3 = 0
          assign {InsMem[115], InsMem[114], InsMem[113], InsMem[112]} = 32'b1110_00_0_0100_0_0000_0100_00010000011; //ADD R4,R0,R3,LSL #2
          assign {InsMem[119], InsMem[118], InsMem[117], InsMem[116]} = 32'b1110_01_0_0100_1_0100_0101_00000000000; //LDR R5,[R4],#0
          assign {InsMem[123], InsMem[122], InsMem[121], InsMem[120]} = 32'b1110_01_0_0100_1_0100_0110_00000000100; //LDR R6,[R4],#4
          assign {InsMem[127], InsMem[126], InsMem[125], InsMem[124]} = 32'b1110_00_0_1010_1_0101_0000_00000000110; //CMP R5,R6
          assign {InsMem[131], InsMem[130], InsMem[129], InsMem[128]} = 32'b1100_01_0_0100_0_0100_0110_00000000000; //STRGT R6,[R4],#0
          assign {InsMem[135], InsMem[134], InsMem[133], InsMem[132]} = 32'b1100_01_0_0100_0_0100_0101_00000000100; //STRGT R5,[R4],#4
          assign {InsMem[139], InsMem[138], InsMem[137], InsMem[136]} = 32'b1100_00_1_0100_0_0011_0011_00000000001; //ADD R3,R3,#1
          assign {InsMem[143], InsMem[142], InsMem[141], InsMem[140]} = 32'b1110_00_1_1010_1_0011_0000_00000000011; //CMP R3,#3
          assign {InsMem[147], InsMem[146], InsMem[145], InsMem[144]} = 32'b1011_10_1_0_11111111111111111111110111  ; //BLT#-9
          assign {InsMem[151], InsMem[150], InsMem[149], InsMem[148]} = 32'b1110_00_1_0100_0_0010_0010_00000000001; //ADD R2,R2,#1
          assign {InsMem[155], InsMem[154], InsMem[153], InsMem[152]} = 32'b1110_00_0_1010_1_0010_0000_00000000001;//CMP R2,R1
          assign {InsMem[159], InsMem[158], InsMem[157], InsMem[156]} = 32'b1011_10_1_0_11111111111111111110011  ;//BLT#-13
          assign {InsMem[163], InsMem[162], InsMem[161], InsMem[160]} = 32'b1110_01_0_0100_1_0000_0001_00000000000;//LDR R1 ,[R0],#0//    R1 = -2147483648
          assign {InsMem[167], InsMem[166], InsMem[165], InsMem[164]} = 32'b1110_01_0_0100_1_0000_0010_00000000100;//LDR R2 ,[R0],#4//    R2 = -1073741824
          assign {InsMem[171], InsMem[170], InsMem[169], InsMem[168]} = 32'b1110_01_0_0100_1_0000_0011_00000001000;//LDR R3 ,[R0],#8//    R3 = 41
          assign {InsMem[175], InsMem[174], InsMem[173], InsMem[172]} = 32'b1110_01_0_0100_1_0000_0100_00000001100;//STR R4 ,[R0],#12//   R4 = 8192
          assign {InsMem[179], InsMem[178], InsMem[177], InsMem[176]} = 32'b1110_01_0_0100_1_0000_0101_00000010000;//STR R5 ,[R0],#16//   R5= -123
          assign {InsMem[183], InsMem[182], InsMem[181], InsMem[180]} = 32'b1110_01_0_0100_1_0000_0110_00000010100;//STR R6 ,[R0],#20//   R6 = 10
          assign {InsMem[187], InsMem[186], InsMem[185], InsMem[184]} = 32'b1110_10_1_0_11111111111111111111111;//B#-1

endmodule
```

به منظور اطمینان از نحوه ی عملکرد این بخش، تست بنچی طراحی شد که حرکت موج گونه ی خروجی PC در پایپ لاین مشاهده شود:



لذا عملکرد مرحله طراحی IF اطمینان حاصل شد و به ارائه ی توضیحات بخش بعدی میپردازیم.

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

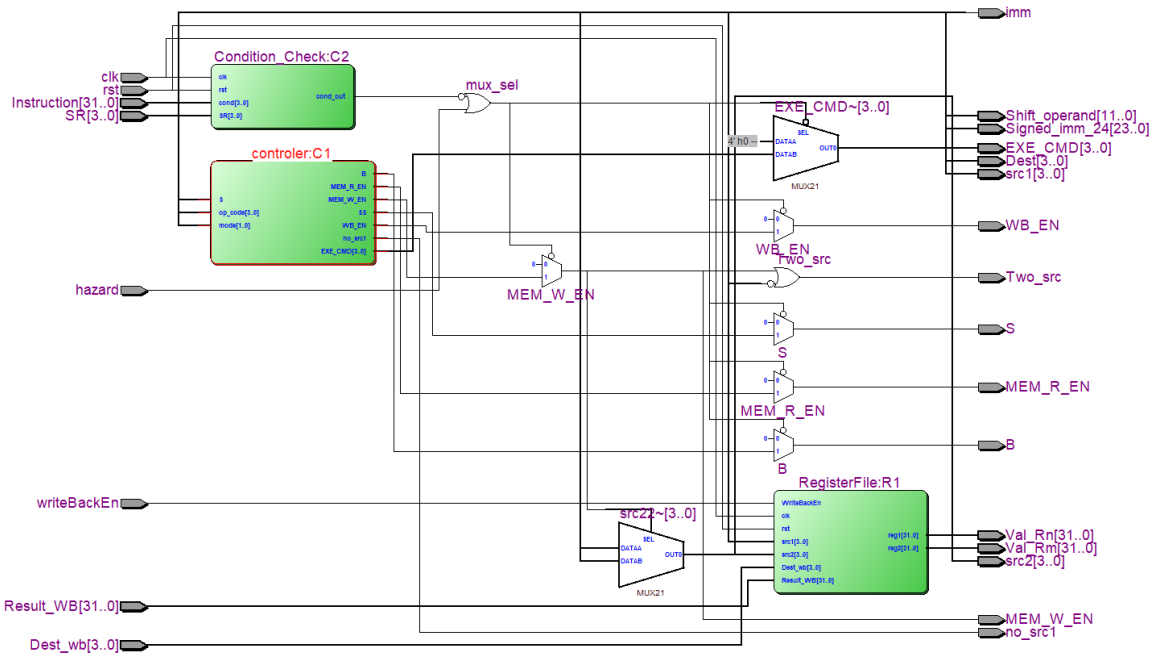
Instruction Decode Stage (ID)

در این بخش، دستوری که از استیج IF واکشی شده، decode میشود و سیگنال های کنترلی لازم برای اجرای آن دستور صادر میشود و دستوراتی که لازم است به استیج های دیگر بروند، به طور مستقیم یا از طریق پایپ لاین منتقل میشوند. همچنین register file در این مرحله وجود دارد که قابلیت خواندن و نوشتن دارد/

بخش های اصلی این استیج عبارتند از:

1. Control Unit: این ماژول وظیفه ی تولید سیگنال های کنترلی پردازنده به منظور انجام عملکرد های محاسباتی، حافظه و انجام پرش را بر عهده دارد.
2. Register File: این ماژول شامل تعدادی ثبات است که سریعترین حافظه هایی هستند که پردازنده با آن ها میتواند عملکرد های پردازشی داشته باشد. در طراحی پردازنده ی ARM بسیاری از بخش های حافظه را ایده آل در نظر گرفتیم و ممکن است با عنوان کردن عبارت "سریعترین حافظه های در دسترس پردازنده" به شک بیافتید. اما در پردازنده ی واقعی برای Register File این عبارت صادق است.
3. Condition Check: این ماژول برقراری شرط در دستورات شرطی را بررسی میکند و در صورت برقرار نبودن شرط اجازه نمیدهد که سیگنال های کنترلی control unit به بخش های دیگر منتقل شود به جای آن سیگنال ها 0 را قرار میدهد.

در تصویر زیر شمای RTL این بخش را مشاهده میفرمایید:



علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

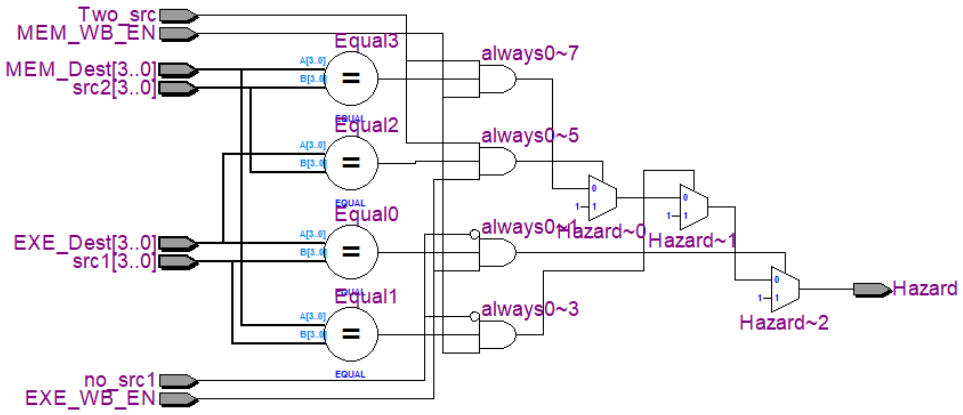
همچنین کد مربوط به این بخش را در زیر مشاهده میفرمایید:

```
module ID_Stage(input clk , rst,
                input [31:0] Instruction,
                input [31:0] Result_WB,
                input writeBackEn,
                input [3:0] Dest_wb,
                input [3:0] SR,
                input hazard,

                output no_src1,
                output WB_EN,MEM_R_EN,MEM_W_EN,B,S,
                output [3:0] EXE_CMD,
                output [31:0] Val_Rn,Val_Rm,
                output imm,
                output [11:0] Shift_operand,
                output [23:0] Signed_imm_24,
                output [3:0] Dest,src1,src2,
                output Two_src
                  );

wire mux_sel;
wire cond_out,SS1,B1,MEM_R_EN1,MEM_W_EN1,WB_EN1;
wire [3:0] EXE_CMD1,src22;

RegisterFile R1 (
  .clk(clk),.rst(rst),
  .src1(Instruction[19:16]),.src2(src22),.Dest_wb(Dest_wb),
  .Result_WB(Result_WB),
  .WriteBackEn(writeBackEn),
  .reg1(Val_Rn),.reg2(Val_Rm)
  );

controler C1 (.op_code(Instruction[24:21]),.mode(Instruction[27:26]),.S(Instruction[20]),.SS(SS1),
              .B(B1),.MEM_R_EN(MEM_R_EN1),.MEM_W_EN(MEM_W_EN1),
             .WB_EN(WB_EN1),.EXE_CMD(EXE_CMD1), .no_src1(no_src1));

Condition_Check C2(.cond(Instruction[31:28]),.clk(clk),.rst(rst),
                   .SR(SR),.cond_out(cond_out));


assign mux_sel      = ~((hazard)|(~cond_out));
assign S            = (mux_sel)? SS1 : 1'b0;
assign B            = (mux_sel)? B1 : 1'b0;
assign MEM_R_EN     = (mux_sel)? MEM_R_EN1    : 1'b0;
assign MEM_W_EN     = (mux_sel)? MEM_W_EN1    : 1'b0;
assign WB_EN        = (mux_sel)? WB_EN1       : 1'b0;
assign EXE_CMD  = (mux_sel)? EXE_CMD1 : 4'b0;
assign src22        = (MEM_W_EN)? Instruction[15:12] : Instruction[3:0] ;

assign imm                  = Instruction[25];
assign Shift_operand        = Instruction[11:0];
assign Signed_imm_24        = Instruction[23:0];
assign Dest                 = Instruction[15:12];
assign src1                 = Instruction[19:16];
assign src2                 = src22;

assign Two_src      =  ((MEM_W_EN)|(~Instruction[25]));

endmodule
```

به منظور مشاهده جزئیات بیشتر میتوانید به کدهای ارسال شده مراجعه کرده و هریک از ماژول های اینستنس گرفته شده را مشاهده فرمایید.

Hazard Detection Unit

در این ماژول مخاطره ی وابستگی داده ای حل میشود و هنگامی که به طور مثال یک دستور میخواهد در یک رجیستر بنویسد، در حالی که دستور بعدی میخواهد از همان رجیستر بخواند در حالی که در آن رجیستر مقدار جدید ذخیره نشده، مرحله ی واکشی دستور و رجیستر میانی آن را متوقف میکند تا دستورات جلو نروند تا وقتی که مقدار درست در رجیستر مذکور ذخیره شود. تصویر RTL آن را در شکل زیر مشاهده میکنید:



علیرضا جابری راد
810196438

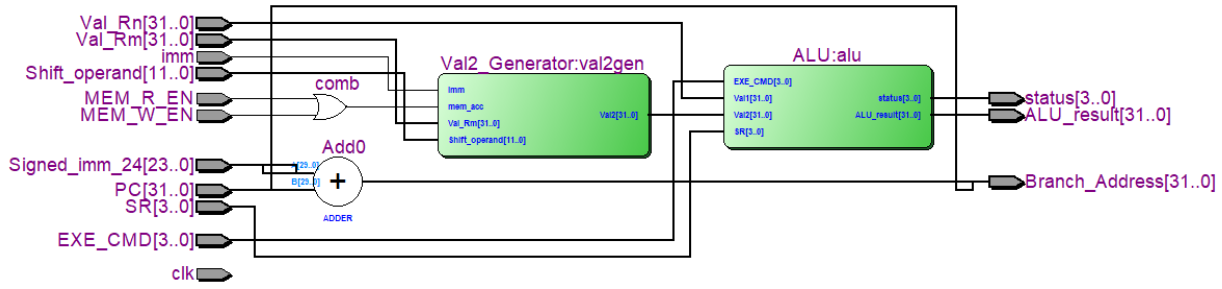
گزارش نهایی
آزمایشگاه معماری کامپیوتر

همچنین کد آنرا در زیر مشاهده میفرمایید:

```
module Hazard_Detection_Unit (
    src1,
    src2,
    EXE_Dest,
    MEM_Dest,
    Two_src,
    EXE_WB_EN,
    MEM_WB_EN,
    no_src1,

    Hazard
);

    input [3:0] src1;
    input [3:0] src2;
    input [3:0] EXE_Dest;
    input [3:0] MEM_Dest;
    input Two_src;
    input EXE_WB_EN;
    input MEM_WB_EN;
    input no_src1;

    output reg Hazard;

    always @(*)
    begin
      Hazard = 1'b0;
      if ((src1 == EXE_Dest) && (EXE_WB_EN == 1'b1) && ~no_src1)
        Hazard = 1'b1;
      else if ((src1 == MEM_Dest) && (MEM_WB_EN == 1'b1) && ~no_src1)
        Hazard = 1'b1;
      else if ((src2 == EXE_Dest) && (EXE_WB_EN == 1'b1) && (Two_src == 1'b1))
        Hazard = 1'b1;
      else if ((src2 == MEM_Dest) && (MEM_WB_EN == 1'b1) && (Two_src == 1'b1))
        Hazard = 1'b1;
      else
        Hazard = 1'b0;
    end

endmodule
```

Execute Stage (EXE)

این استیج مسئولیت انجام محاسبات را بر عهده دارد و برحسب سیگنال هایی که از مرحله ID به این مرحله انتقال میابند، عملیات ALU و مقدار ورودی دوم ALU توسط Val2 Generator تعیین میشود. همچنین آدرسی که در هنگام انجام دستور پرش لازم است به Program Counter داده شود در این استیج تولید میشود. تصویر RTL این استیج را در زیر مشاهده میفرمایید:



کد این بخش را در تصویر زیر میتوانید مشاهده کنید:

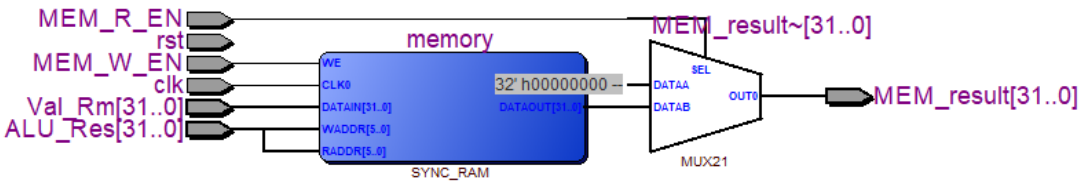
```
module EXE_Stage(
  input clk,
  input[3:0] EXE_CMD,
  input MEM_R_EN, MEM_W_EN,
  input[31:0] PC,
  input[31:0] Val_Rn, Val_Rm,
  input imm,
  input[11:0] Shift_operand,
  input[23:0] Signed_imm_24,
  input[3:0] SR,

  output[31:0] ALU_result, Branch_Address,
  output[3:0] status
);

  wire[31:0] Val2;

  ALU alu(
    .EXE_CMD(EXE_CMD),
    .Val1(Val_Rn),
    .Val2(Val2),
    .SR(SR),
    .status(status),
    .ALU_result(ALU_result)
  );

  Val2_Generator val2gen(
    .Val_Rm(Val_Rm),
    .Shift_operand(Shift_operand),
    .imm(imm),
    .mem_acc(MEM_R_EN|MEM_W_EN),

    .Val2(Val2)
  );

  assign Branch_Address= {{ {8{Signed_imm_24[23]}} , Signed_imm_24 }<<2} + PC;

endmodule
```

به منظور مشاهده جزئیات بیشتر میتوانید به کدهای ارسال شده مراجعه کرده و هریک از ماژول های اینستنس گرفته شده را مشاهده فرمایید.

Memory Stage (MEM)

در این استیج نوشتن در مموری فایل و خواندن از آن انجام میشود، آدرس خواندن و نوشتن در مموری توسط ALU تولید میشود و به این استیج می آید و مقداری که هنگام نوشتن نیاز است، به عنوان Rm از استیج ID به این استیج توسط پایپ لاین منتقل میشود.

تصویر RTL آن را در زیر مشاهده میکنید:



علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

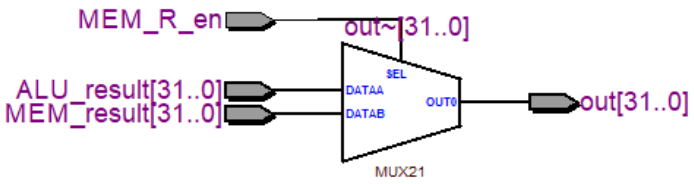
کد این بخش را در زیر مشاهده میفرمایید:

```
module MEM_Stage(
  clk,
  rst,
  Val_Rm,
  ALU_Res,
  MEM_W_EN,
  MEM_R_EN,
  MEM_result
);
  input clk;
  input rst;
  input[31:0] Val_Rm;
  input[31:0] ALU_Res;
  input MEM_W_EN;
  input MEM_R_EN;

  output[31:0] MEM_result;

  reg[31:0] memory[0:63];
  wire[31:0] address;

  assign address= (ALU_Res-32'd1024) >> 2;

  assign MEM_result = MEM_R_EN ? memory[address] : 32'b0;

  always@(posedge clk)begin//write
    if(MEM_W_EN)
      memory[address] = Val_Rm;
  end

endmodule
```

علت اینکه در RTL پس از مموری یک مالتیپلکسر گذاشته است اینست که من در کد نوشته ام که در صورتی که MEM_R_EN صفر باشد، صفر را در خروجی قرار دهد.

Write Back Stage (WB)

این استیج صرفا یک مالتیپلکسر است که مقدار رجیستری که باید بازنویسی شود را بر حسب اینکه دستور خواندن از مموری آمده است یا خیر، تعیین میکند. تصویر RTL آنرا در زیر میتوانید مشاهده کنید:



همچنین کد متناظر با آن را در تصویر زیر مشاهده میفرمایید:

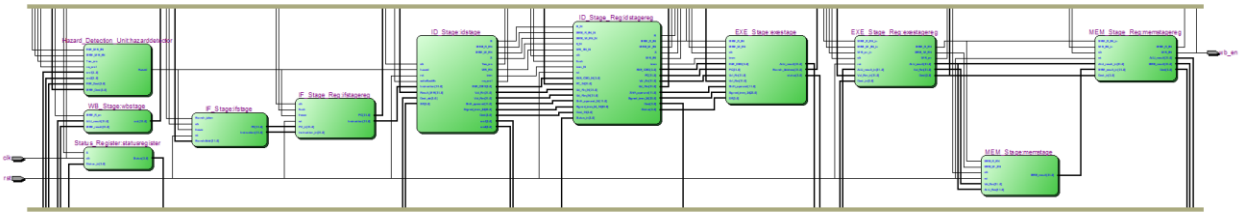
```
module WB_Stage(
  input[31:0] ALU_result, MEM_result,
  input MEM_R_en,
  output[31:0] out
);
  assign out= MEM_R_en ? MEM_result : ALU_result;
endmodule
```

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

بررسی پردازنده ی ساخته شده توسط استیج های معرفی شده

ابتدا RTL این مدار را مشاهده بفرمایید:



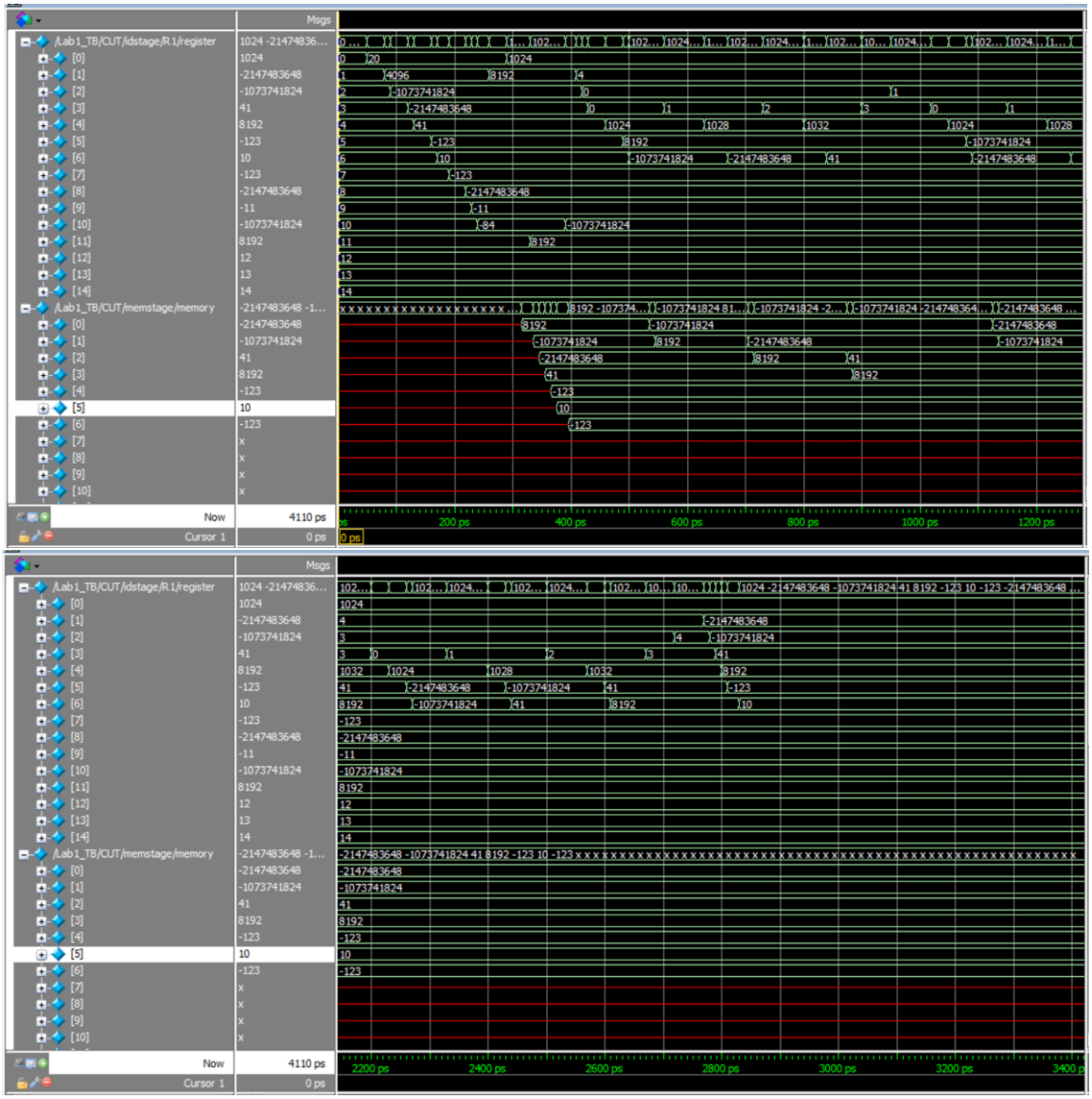
تنها تفاوتی که این RTL نسبت به RTL Design ای که در حین طراحی در نظر گرفتیم و در گزارش کار قرار داده شده، اینست که چینش ماژول ها یکسان نیست اما نکته ی مهم اینست که وایرینگ آنها یکسان است. همچنین بنده عمدا wb_en ای را که از رجیستر میانی بخش مموری می آمد را به عنوان خروجی تعریف کردم تا یک سیگنال خروجی وابسته به لبه ی کلاک داشته باشیم و ابزار سنتز logic element ها را در گزارش خود لحاظ کند.

این مدار را با استفاده از یک تست بنچ و دستوراتی که در گزارش کار قید شده بود تست کردم که کد تست بنچ به همراه نتایج تست را در تصاویر زیر مشاهده میفرمایید:

```
module Lab1_TB;

  reg clk,rst;
  Top_Module CUT(
    .clk(clk),
    .rst(rst)
  );

  always#5 clk=~clk;

  initial begin
    clk=0;
    rst=1;
    #10
    rst=0;
    #4000
    $stop;
  end
endmodule
```

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر



همچنین در گزارش زیر میزان استفاده از سخت افزار را میتوانید مشاهده بفرمایید:

Flow Summary	
Flow Status	Successful - Fri Dec 17 02:44:38 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Top_Module
Top-level Entity Name	Top_Module
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1,918 / 33,216 (6 %)
Total combinational functions	1,584 / 33,216 (5 %)
Dedicated logic registers	866 / 33,216 (3 %)
Total registers	866
Total pins	3 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	2,528 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

همانطور که در تصاویر تست بنچ بالا میبینید، دوره تناوب کلاک 10ps بوده و تعداد کلاکی که به منظور اجرای این برنامه زده شده تا وقتی که PC برای اولین بار به JMP -1 میرسد برابر است با:

$$(2785 - 10)/10 = 278.5$$

حال با اضافه کردن کد زیر تعداد Instruction ای که در این کد اجرا شده برابر 180 بدست آمد، یعنی مداری که در تست بنچ در زمان 2785ps به آن رسیده است (که چون عملکرد branch در دستورات داریم این مقدار تقریبی است):

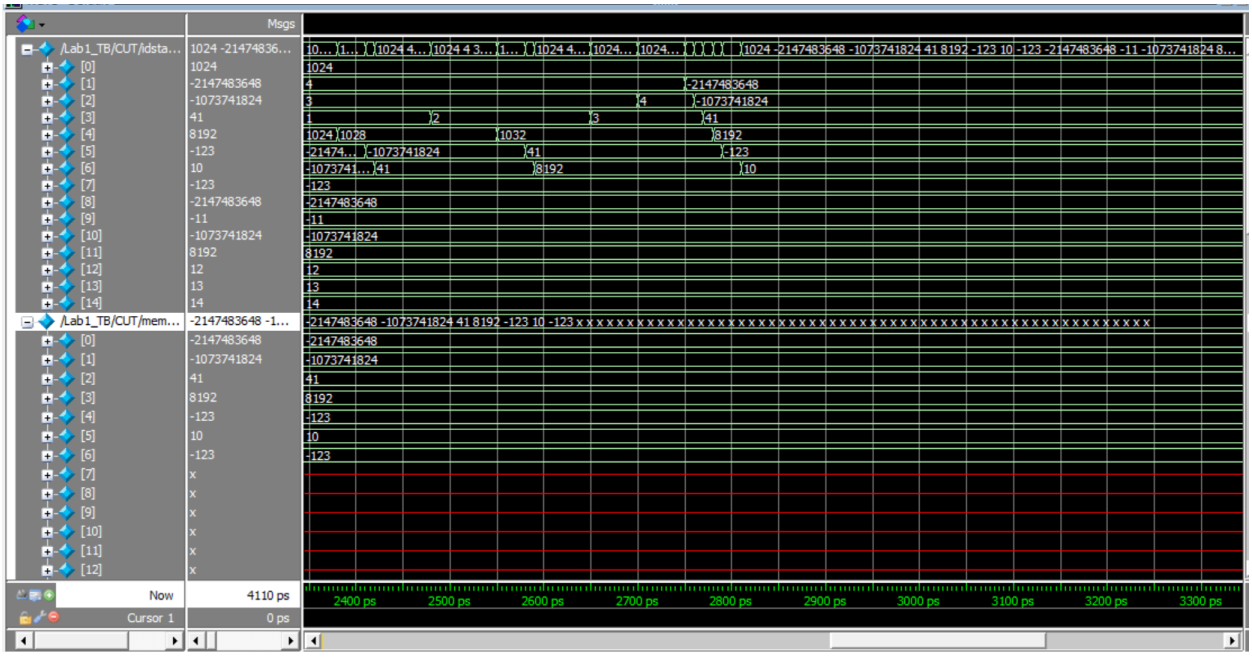
```
reg[7:0] instruction_cnt=0;

always@(CUT.IF_Reg_PC)
  instruction_cnt=instruction_cnt+1;
```

لذا CPI در این مرحله برابر خواهد بود با:

$$CPI = 278.5/180 = 1.547$$

بخش امتیازی ARM

با در نظر گرفتن MOV و MOVN به عنوان دستوراتی که وابستگی داده ای لازم نیست برای سورس اول آنها چک شود(چون در instruction آن ها سورس 1 صفر است در حالی که این دستور اصلا کاری با سورس 1 ندارد) عملکرد پردازنده در اجرای این کد به اندازه ی 2 کلاک سایکل بهبود یافت که نتیجه ی شبیه سازی را در تصویر زیر مشاهده میفرمایید:



لذا با بهبود اعمال شده CPI در این حالت برابر خواهد بود با:

$$CPI = 276.5/180 = 1.536$$

علیرضا جابری راد
810196438

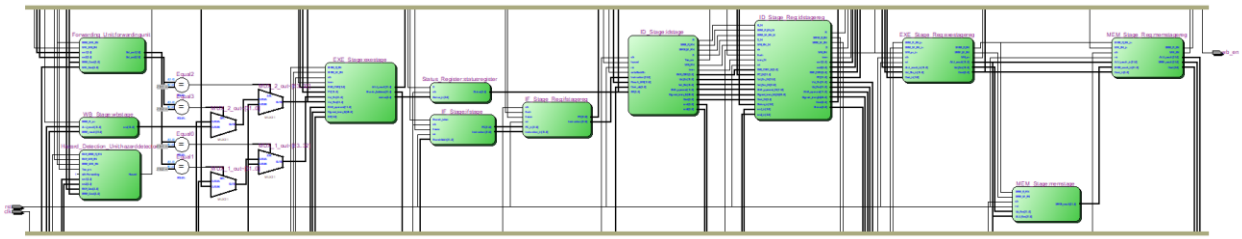
گزارش نهایی
آزمایشگاه معماری کامپیوتر

Forwarding

با اضافه شدن این ماژول توقف پردازنده در هنگام ایجاد وابستگی داده ای کاهش یافت و با مدیریت Forwarding Unit یکسری وایرها بین داده هایی که در استیج های جلوتر بودند به استیج های عقب تر انتقال داده شدند و در صورت لزوم با لحاظ کردن اولویت به آخرین تغییر(یعنی نزدیک ترین استیج، با استفاده از ایجاد سلسله مراتب با استفاده از if و else که در کد میتوانید مشاهده کنید) داده ها Forward میشدند. در تصویر زیر کد مربوط به Forwarding Unit را مشاهده میفرمایید:

```
module Forwarding_Unit(
  src1,
  src2,
  WB_WB_EN,
  MEM_WB_EN,
  MEM_Dest,
  WB_Dest,
  Sel_src1,
  Sel_src2
  );

  input[3:0] src1,src2,MEM_Dest,WB_Dest;
  input WB_WB_EN,MEM_WB_EN;

  output reg[1:0] Sel_src1, Sel_src2;

  always@(*)begin
    if(src1 == MEM_Dest && MEM_WB_EN)
      Sel_src1=2'd1;
    else if(src1 == WB_Dest && WB_WB_EN)
      Sel_src1=2'd2;
    else
      Sel_src1=2'd0;
  end

  always@(*)begin
    if(src2 == MEM_Dest && MEM_WB_EN)
      Sel_src2=2'd1;
    else if(src2 == WB_Dest && WB_WB_EN)
      Sel_src2=2'd2;
    else
      Sel_src2=2'd0;
  end

endmodule
```

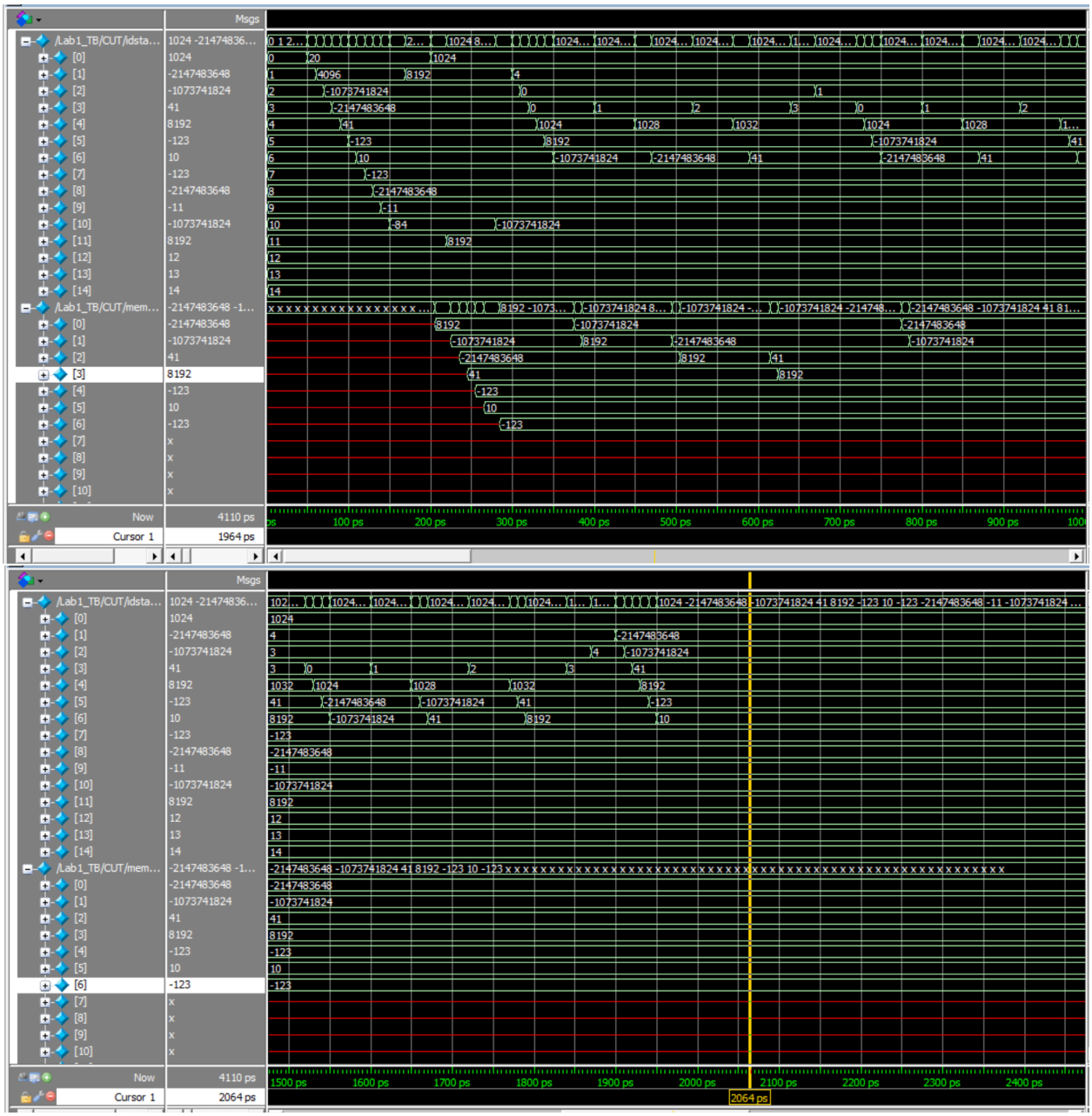
با اضافه کردن این ماژول و انجام یکسری تغییرات در کلیت مدار به منظور سازگاری با قابلیت جدید اضافه شده، RTL مدار به صورت زیر بدست آمد:



به منظور اطمینان از نحوه عملکرد پردازنده، از تست بنچی که در بخش گذشته از آن استفاده کردیم، در این بخش نیز استفاده کردیم و نتایج مطابق تصویر زیر بدست آمد و همانطور که میبینید عملکرد مدار صحیح است:

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر



طبق اندازه گیری انجام شده، در 1915ps خروجی PC به دستور JMP -1 میرسد. لذا در این حالت خواهیم داشت:

$$\text{clock cycles} = (1915 - 10) / 10 = 190.5$$

$$\Rightarrow \text{CPI} = 190.5/180 = 1.058$$

$$\Rightarrow \text{میزان افزایش کارایی نسبت به پردازنده آرم بدون فوروارد‌ینگ} = (1.536 / 1.058)*100 - 100 = 45.18\%$$

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

در ادامه با مشاهده نتایج سنتز، به بررسی هزینه ی سخت افزاری اضافه کردن بخش فورواردینگ میپردازیم:

Flow Summary	
Flow Status	Successful - Fri Dec 17 02:35:43 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Top_Module
Top-level Entity Name	Top_Module
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	1,946 / 33,216 (6 %)
Total combinational functions	1,702 / 33,216 (5 %)
Dedicated logic registers	771 / 33,216 (2 %)
Total registers	771
Total pins	3 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	2,528 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

در پردازنده ای که در بخش قبل طراحی کردیم، تعداد کل المان های منطقی برابر بود با 1918 در حالی که با اضافه کردن فورواردینگ، این تعداد به 1946 افزایش یافت. لذا:

میزان هزینه سخت افزاری(درصد افزایش استفاده از المان های منطقی) = 1946/1918 * 100 – 100 = 1.46%

=> Performance per Cost = 45.18/1.46 = 30.945

(البته در این روش محاسبه کارایی به هزینه، نسبت به اینکه باید میزان افزایش را به عنوان مبنا در نظر بگیرم یا کارایی نسبی به حالت گذشته، مطمئن نبودم و میزان افزایش را به عنوان مبنای محاسبه کارایی به هزینه لحاظ کردم)

به علت آنکه این عدد به طور قابل توجهی بزرگتر است 1 است، اضافه کردن Forwarding Unit بسیار تصمیم درست و مفیدی است.

بخش امتیازی Forwarding

به منظور افزایش کارایی مدار، پیشنهادی که دارم اینست که به جای آنکه عملکرد مقایسه در EXE انجام شود، یک مقایسه کننده در استیج ID قرار دهیم. در این صورت به هنگام اجرای دستور پرش، به ازای اشتباه بودن شرط پرش، نیازی نیست دستورات را از پایپ لاین (با flush کردن رجیستر های میانی IF و ID) حذف کنیم. زیرا نتیجه ی مقایسه در همان لحظه ای که دستور کدگشایی میشود تعیین میگردد و برقرار بودن یا نبودن شرط پرش بررسی میشود و لذا در سیکل بعدی، دستور صحیح وارد پایپ لاین میشود و نیازی به flush کردن نیست.

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

SRAM

بخش های قبلی به صورت عادی پیاده سازی کردیم، به طوری که به صورت combinational میتوانستیم از آن بخوانیم و در یک کلاک میتوانستیم در آن بنویسیم. اما در این بخش به منظور آشنایی با حافظه ی SRAM، این دو عملیات را در 6 کلاک سایکل انجام میدهیم و در طول انجام عملیات خواندن و نوشتن در SRAM با صدور سیگنال freeze از پیشروی دستورات در طول پایپ لاین جلوگیری کردیم. پیاده سازی SRAM به پیاده سازی دو ماژول SRAM و کنترلر آن تقسیم شده است که کد هریک را در دو تصویر زیر مشاهده میفرمایید:

```
timescale 1ns/1ns
module SRAM(
  CLK,
  RST,
  SRAM_WE_N,
  SRAM_ADDR,
  SRAM_DQ
  );

  input CLK,RST,SRAM_WE_N;
  input[16:0] SRAM_ADDR;

  inout[31:0] SRAM_DQ;

  reg[31:0] memory[0:511];

  assign #30 SRAM_DQ = SRAM_WE_N ? memory[SRAM_ADDR] : 32'bz;

  always@(posedge CLK)begin
    if(~SRAM_WE_N)
      memory[SRAM_ADDR] = SRAM_DQ;
  end

  endmodule
```

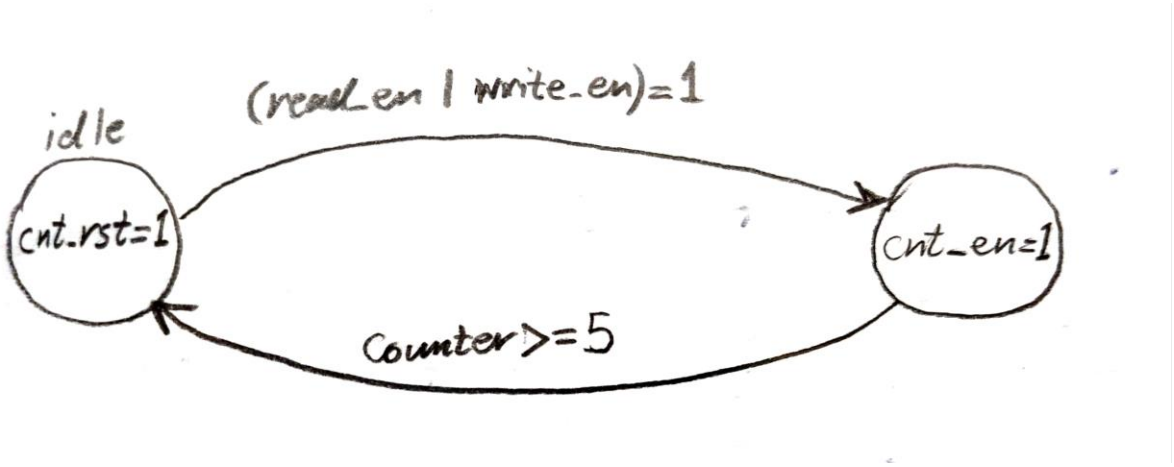
بخش اصلی کد کنترلر آن را در زیر مشاهده میکنید:

```
reg[2:0] clk_counter;
initial clk_counter=3'b0;
reg cnt_en, cnt_rst;
initial begin cnt_en=1'b0; cnt_rst=1'b0; end

reg ps,ns;

localparam  idle = 0, w_r_wait = 1;
assign SRAM_UB_N=1'b1;
assign SRAM_LB_N=1'b1;
assign SRAM_CE_N=1'b1;
assign SRAM_OE_N=1'b1;
assign SRAM_WE_N = ~write_en;
assign SRAM_DQ = (write_en) ? writeData : 32'bz;
assign readData = (read_en) ? SRAM_DQ : 32'bz;
assign SRAM_ADDR = ((address-32'd1024) >> 2);

always@(*)begin
  ready=1'b1;
  if(ns==w_r_wait)
    ready=1'b0;
end

always@(posedge clk, posedge rst) begin
  if(rst)
    ps<=1'b0;
  else
    ps<=ns;
end

always@(*)begin
  case(ps)
    idle: ns=(read_en | write_en) ? w_r_wait : idle;
    w_r_wait: ns=(clk_counter<3'd4) ? w_r_wait : idle;
  endcase
end

always@(ps)begin
  cnt_en=1'b0; cnt_rst=1'b0;
  case (ps)
    idle: cnt_rst=1'b1;
    w_r_wait: cnt_en=1'b1;
    default:begin cnt_en=1'b0; cnt_rst=1'b0; end
  endcase
end

always@(posedge clk, posedge cnt_rst, posedge rst)begin
  if(rst)
    clk_counter<=3'b0;
  else if(cnt_rst)
    clk_counter<=3'b0;
  else
    clk_counter<=clk_counter+3'd1;
end

endmodule
```

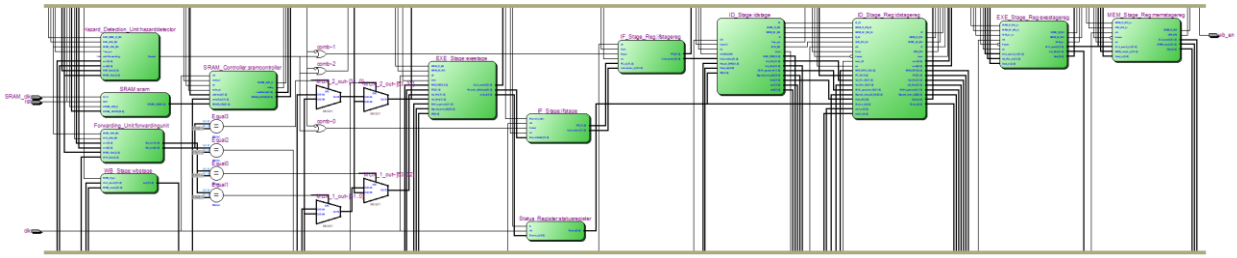
علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

استیت ماشینی که در این کد استفاده شده مطابق تصویر زیر است که کنترل میکند عملیات SRAM در 6 کلاک سایکل انجام شود:



همچنین در تصویر زیر نتیجه ی اضافه کردن این دو ماژول به پردازنده ی مرحله قبل را مشاهده میفرمایید:



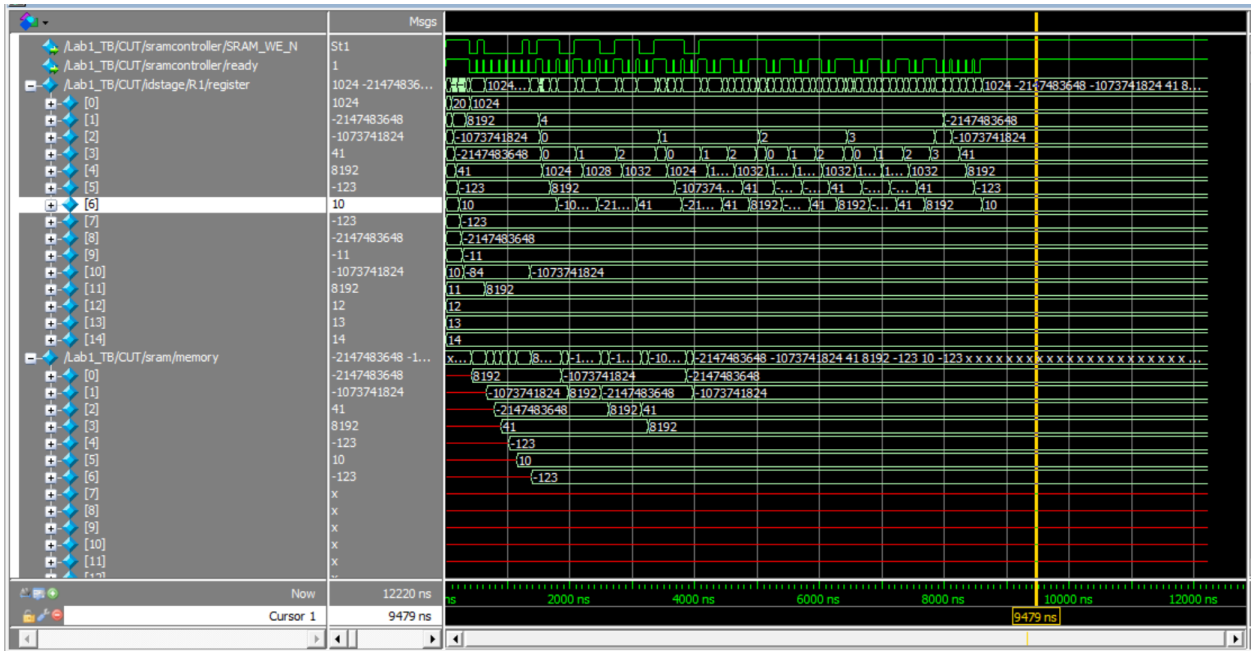
در ادامه با استفاده از تستی که در زیر مشاهده میکنید، عملکرد مدار را ارزیابی میکنیم:

```
`timescale 1ns/1ns
module Lab1_TB;

  reg clk,rst,SRAM_clk;
  Top_Module CUT(
    .SRAM_clk(SRAM_clk),
    .clk(clk),
    .rst(rst)
  );

  always#20 SRAM_clk=~SRAM_clk;
  always#10 clk=~clk;

  initial begin
    SRAM_clk=0;
    clk=0;
    rst=1;
    #20
    rst=0;
    #200
//    rst=1;
//    #10
//    rst=0;
    #12000
    $stop;
  end
endmodule
```

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

در تصویر زیر نتیجه ی تست بنچ این بخش را مشاهده میفرمایید:



مطابق روش های گذشته تعداد کلاک اجرای برنامه و CPI را محاسبه میکنیم:

clock cycles = (8230 – 20) / 20 = 410.5

=> CPI = 410.5/180 = 2.28

به وضوح کارایی این بخش نسبت به بخش قبل بسیار کاهش یافته است:

=> میزان تغییر کارایی نسبت به پردازنده آرم با فوروارد‌ینگ = (1.058 / 2.28)*100 – 100 = -53.596%

همانطور که محاسبات بالا نشان میدهد، کارایی این پردازنده نسبت به حالتی که از حافظه داخلی استفاده میکردیم، بیشتر از 50 درصد کاهش داشته است!

در ادامه تصویر نتایج سنتز را مشاهده میکنید:

Flow Summary	
Flow Status	Successful - Fri Dec 17 16:41:26 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Top_Module
Top-level Entity Name	Top_Module
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	21,836 / 33,216 (66 %)
Total combinational functions	13,244 / 33,216 (40 %)
Dedicated logic registers	17,191 / 33,216 (52 %)
Total registers	17191
Total pins	4 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	480 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

همانطور که میبینید، استفاده از منابع سخت افزاری در این بخش نسبت به بخش گذشته افزایش بسیار زیادی داشته است:

میزان هزینه سخت افزاری(درصد افزایش استفاده از المان های منطقی) = 21836/1946 * 100 – 100 = +1022.097%

همانطور که میبینید علاوه بر کاهش کارایی پردازنده، میزان استفاده از منابع سخت افزاری هم بسیار افزایش پیدا کرده است.

بخش امتیازی SRAM

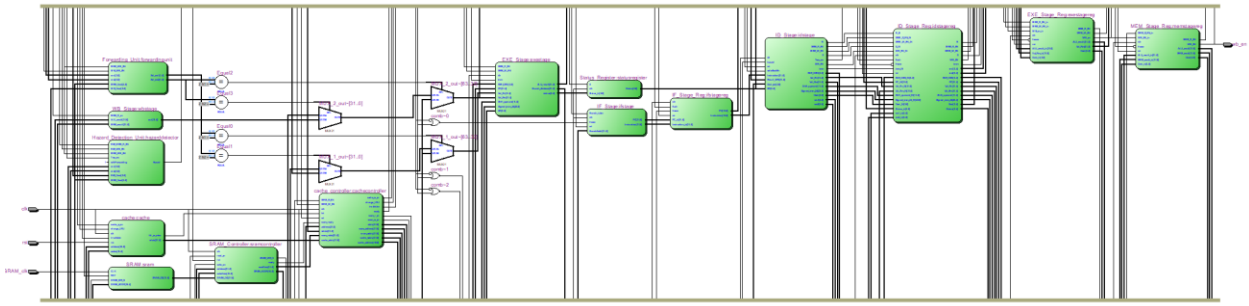
به منظور افزایش کارایی پردازنده ای که تا اینجا ساختیم، میتوان از رویکرد In memory computing بهره برد. in memory computing روش های مختلفی دارد، اما روشی که در این مورد مد نظر بنده است اینست که دستوراتی که نیاز است اطلاعات آن از مموری گرفته شود و پس از انتقال داده ی آن به رجیسترها محاسبات ریاضیاتی پایه روی آن انجام شود(یعنی به جای آنکه دو دستور را اجرا کنیم تا مثلا مقدار یک آدرس مموری را به علاوه ی مقدار یک رجیستر کنیم و در مموری ذخیره کنیم)، یک ALU ساده در کنار SRAM داشته باشیم که روندی که توضیح داده شده را به عنوان یک دستور واحد در زمانی کمتر اجرا کند. بدین منظور علاوه بر اضافه کردن دستور جدید به دستورات پردازنده، لازم است در کنترلر SRAM هم تغییراتی ایجاد شود.

علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

Cache

در تصویر زیر RTL مربوط به این بخش را مشاهده میفرمایید:



به منظور اضافه کردن cache به پردازنده، دو ماژول cache و کنترلر آن به پردازنده اضافه شدند که بخش اصلی کدهای آن ها را در تصاویر زیر مشاهده میکنید:

```
input clk,rst;
input[18:0] address;
input[31:0] wdata;
input cache_w_en;
input invalidate;
input change_LRU;

output hit_or_miss;
output[31:0] rdata;

reg[31:0] datablk[0:63][0:3];

reg[9:0] tag [0:63][0:1];

reg valid [0:63][0:1];

reg LRU [0:63];

wire hit_or_miss0, hit_or_miss1;
wire[9:0] tag_in;
wire[5:0] index;
wire LSB_or_MSB;

assign tag_in=address[18:9];
assign index=address[8:3];
assign LSB_or_MSB=address[2];

assign hit_or_miss0=((tag[index][0] == tag_in) && valid[index][0]);
assign hit_or_miss1=((tag[index][1] == tag_in) && valid[index][1]);
assign hit_or_miss=(hit_or_miss0 | hit_or_miss1);

assign rdata=datablk[index][{hit_or_miss1,LSB_or_MSB}];

always@(posedge clk)begin
  if(invalidate)
    valid[index][hit_or_miss1]<=1'b0;
  else if(cache_w_en)begin
    valid[index][LRU[index]]<=1'b1;
    datablk[index][{LRU[index],LSB_or_MSB}]<=wdata;
    if(change_LRU)
      LRU[index]<=~LRU[index];
  end
end

integer i;
initial begin
  for(i=0;i<64;i=i+1)begin
    valid[i][0]=1'b0;
    valid[i][1]=1'b0;
    tag[i][0]=10'b0;
    tag[i][1]=10'b0;
    LRU[i]=1'b0;
  end
end
```

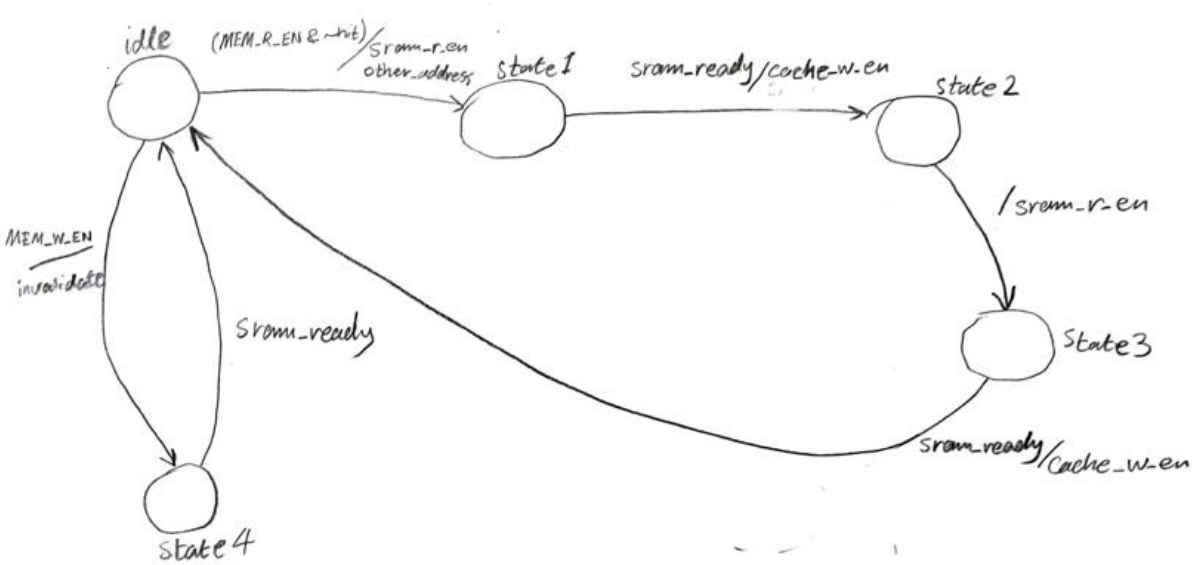
علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

کنترلر:

```
input clk;
input rst;

//memory stage unit
input[31:0] address;
input[31:0] wdata;
input MEM_R_EN;
input MEM_W_EN;

output[31:0] rdata;
output ready;

//SRAM controller
input[31:0] sram_rdata;
input sram_ready;

output[31:0] sram_address;
output[31:0] sram_wdata;
output sram_w_en;
output reg sram_r_en;

//Cache
input hit;
input[31:0] cache_rdata;

output reg cache_w_en;
output[31:0] cache_wdata;
output[18:0] cache_address;
output reg change_LRU;
output reg invalidate;

reg[2:0] ps,ns;
localparam  idle=0, state1=1, state2=2, state3=3, state4=4;
reg other_address;

assign ready= (hit | sram_ready) && ~(ps==state1 | ps==state2 | ps==state3);
assign rdata= (hit) ? cache_rdata : sram_rdata;
assign sram_address=address^({29'b0,other_address,2'b00});

assign sram_w_en = MEM_W_EN;
assign sram_wdata = (MEM_W_EN) ? wdata : 32'bz;
assign cache_wdata=sram_rdata;
assign cache_address=(address-32'd1024)^({29'b0,other_address,2'b00});

always@(posedge clk, posedge rst)begin
  if(rst)
    ps<=idle;
  else
    ps<=ns;
end

always@(*)begin
  case(ps)
    idle: ns = (MEM_R_EN & ~hit) ? state1 : (MEM_W_EN) ? state4 : idle;
    state1: ns = sram_ready ? state2 : state1;
    state2: ns=state3;
    state3: ns=sram_ready ? idle : state3;
    state4: ns=(sram_ready) ? idle : state4;
    default: ns=idle;
  endcase
end

always@(ns)begin
  sram_r_en=1'b0; other_address=1'b0; cache_w_en=1'b0; invalidate=1'b0; change_LRU=1'b0;
  case({ps,ns})
    {3'd0,3'd1}: begin sram_r_en=1'b1; other_address=1'b1; end
    {3'd1,3'd2}: begin cache_w_en=1'b1; end
    {3'd2,3'd3}: begin sram_r_en=1'b1; change_LRU=11'b1; end
    {3'd3,3'd0}: begin cache_w_en=1'b1; end
    {3'd0,3'd4}: invalidate=1'b1;
  endcase
end
```

کنترلر بالا بر مبنای استیت ماشین زیر طراحی شده است که عملکرد خواندن و نوشتن cache و SRAM در آن کنترل میشود:



علیرضا جابری راد
810196438

گزارش نهایی
آزمایشگاه معماری کامپیوتر

متاسفانه تستی که از طراحی انجام شده گرفتم، پاسخ درستی نداد و علت آنکه پاسخ درستی نداد و نتوانستم مشکل آنرا با وجود تلاش فراوان حل کنم، این بود که در صورت miss شدن داده، داده ی اول به درستی از SRAM به cache منتقل میشد، اما داده ی دوم در زمان نامناسبی به cache میرسید و 32 بیت دوم، در برخی موارد x بودند و در برخی موارد z بودند و در برخی موارد اعداد نادرستی بودند، همچنین در خروجی هم در صورت miss شدن عدد درستی قرار نمیگرفت. چون این استیت ماشین به گونه ای طراحی شده که 32 بیتی که miss شده به عنوان داده ی دوم در cache ذخیره شود و در پایان عملیات نوشتن در cache، آن داده توسط cache به خروجی منتقل شود در حالی که همانطور که توضیح داده شد، داده ی دوم داده ی غلطی بوده و منجر به اختلال در عملکرد پردازنده میشد.

اما با توجه به کدی که نوشته شده (بخش اصلی دو حلقه ی تودرتو است) اضافه کردن این بخش بین 60 تا 120 کلاک سایکل از تعداد کلاک سایکل مورد نیاز برای اجرای برنامه میکاهد. لذا فرض کنید 90 کلاک سایکل از تعداد کلاک سایکل های اجرای برنامه نسبت به بخش قبل کم شود. در اینصورت خواهیم داشت:

CPI = 320.5/180 = 1.78

=> میزان افزایش کارایی در مقایسه با بخش قبل (بدون کش) = 2.28/1.78 * 100 – 100 = 28.09%

به منظور محاسبه میزان افزایش استفاده از منابع سخت افزاری به تصویر زیر که گزارش سنتز آن است توجه فرمایید:

Flow Summary	
Flow Status	Successful - Fri Jan 14 23:21:41 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Top_Module
Top-level Entity Name	Top_Module
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	26,821 / 33,216 (81 %)
Total combinational functions	16,292 / 33,216 (49 %)
Dedicated logic registers	21,007 / 33,216 (63 %)
Total registers	21007
Total pins	4 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	480 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

لذا طبق تصویر بالا خواهیم داشت:

میزان هزینه سخت افزاری(درصد افزایش استفاده از المان های منطقی) = 26821/21836 * 100 – 100 = +22.829%

=> Performance per Cost = 28.09/22.829 = 1.23

چون مقدار بدست آمده ی Performance per Cost مقداری بزرگتر است یک است، پس اضافه کردن cache به پردازنده ی مرحله ی قبل تصمیم خوبی است.

جهت مقایسه ی سه حالت استفاده از حافظه داخلی، استفاده از SRAM و استفاده از Cache+SRAM به جدول زیر توجه فرمایید:

	internal memory	SRAM	Cache+SRAM
CPI	1.058	2.28	1.78
Total Logic Elements	1946	21836	26821

همانطور که مشاهده میشود هزینه ی سخت افزاری دو مورد آخر نسبت به استفاده از حافظه داخلی بسیار بیشتر و کارایی آن ها کمتر است. اما در مقایسه ی دو مورد آخر نسبت به یکدیگر، استفاده از cache با وجود آنکه سخت افزار بیشتری مصرف میکند، اما کارایی بهتری ارائه میدهد و میزان کارایی آن نسبت به هزینه سخت افزاری آن به گونه ای است که اضافه کردن آن تصمیم خوبی است.

بخش امتیازی Cache

به منظور افزایش کارایی این بخش، به جای آنکه در دو مرتبه داده ی 32 بیتی خوانده شود و در این فرآینده 12 کلاک سایکل مصرف کنیم، باس داده ی بین SRAM و Cache را 64 بیتی کنیم که این زمان نصف شود.

علاوه بر آن روش write through هنگامی که یک داده ی درون کش به دفعات زیاد تغییر میکند بسیار کارایی کش را کاهش میدهد. چون با هر بار تغییر لازم است مقدار درون SRAM هم تغییر کند که این کار انرژی زیادی مصرف میکند و حتی ممکن است سرعت اجرا را نیز(بسته به مدل طراحی کنترلر کش) کاهش دهد. اما روش write back فقط در هنگامی که میخواهیم داده ی جدیدی به جای یک داده قرار دهیم، مقدار موجود در آدرس متناظر با آن را در SRAM آپدیت میکند و در مثالی که به آن اشاره کردم، به جای آنکه چندین بار دسترسی به SRAM ایجاد کنیم و مقدار آنرا آپدیت کنیم، فقط یکبار اینکار را انجام دادیم و با این روش مشکلات مطرح شده حل میشود.

همچنین میتوان میزان حافظه ی cache را افزایش داد، به گونه ای که طول آن یا تعداد way آن را بیشتر کنیم و یا حتی تعداد word داده ای که در هر بلاک ذخیره می کنیم را افزایش دهیم. اما اینکار هزینه ی تولید سخت افزاری را به طرز چشم گیری زیاد میکند. چون تولید کش گران است.