به نام خدا

گزارش کار آزمایشگاه معماری کامپیوتر

کیمیا محمدطاهری (۸۱۰۱۹۸۵۳۵)

سارا اسعدی (۸۱۰۱۹۷۴۴۸)

پردازنده ARM

در بخش ابتدایی این آزمایش مدار پردازنده ARM پیاده سازی شده است. معماری این پردازنده به صورت خط لوله[1] است و شامل پنج مرحله است. به این صورت که در مرحله اول (Instruction Fetch) دستوری که لازم است اجرا شود از حافظه دستور خوانده می‌شود و آدرس دستور بعدی در رجیستری به نام PC[2] ذخیره می‌شود. پس از آن در مرحله دوم (Instruction Decode) ، این دستور تفسیر می‌شود و مشخص می‌شود که برای این دستور از کدام فرآیند عملیاتی پیاده سازی شده باید استفاده شود. در این آزمایش 12 دستور برای پردازنده تعریف شده است. بعد از مشخص شدن نوع دستور، سیگنال‌های کنترلی مربوط به آن تولید می‌شود و به مرحله بعد می‌رود. در مرحله سوم (Execution) دستورات محاسباتی با توجه به سیگنال‌های بخش قبل اجرا می‌شود. در مرحله چهارم (Memory) در صورتی که دستور نیازمند خواندن و نوشتن در حافظه باشد دسترسی به حافظه انجام می‌شود. در مرحله آخر (Write Back) در صورت لزوم، رجیسترها به روز می‌شوند.

مرحله واکشی دستور[3]

در این مرحله از چهار ماژول استفاده شده است. رجیستر PC آدرس دستور بعدی را ذخیره می‌کند، در ادامه از یک جمع کننده برای محاسبه آدرس دستور بعدی استفاده می‌شود. این جمع کننده خروجی رجیستر PC را با ۴ جمع می‌کند. یک multiplexer ورودی رجیستر را انتخاب می‌کند، یکی از ورودی‌های این multiplexer خروجی ماژول جمع‌کننده و ورودی دیگر آن آدرس branch است که از مرحله اجرا وارد این مرحله می‌شود. همچنین یک حافظه دستور وجود دارد که PC را به عنوان آدرس دریافت می‌کند و دستور متناظر را خروجی می‌دهد.

در ادامه کد مربوط به این بخش آمده است.

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

[1] Pipeline
[2] Program Counter
[3] Instruction Fetch

```
wire [31:0] adderOut,PC_in,PC_out;

register PC_reg(
	.clk(clk),
	.rst(rst),
	.ld(~freeze),
	.Qin(PC_in),
	.Q(PC_out)
);
//در این رجیستر که مقدار PC نگهداری می شود و با سیگنال های کنترلی مقدار آن قابل کنترل است

adder a(
	.a(PC_out),
	.b(32'd4),
	.res(adderOut)
);
//adder برای این است که هر بار PC با عدد 4 جمع شود تا آدرس دستور بعدی را نمایش دهد
inst_mem im(
	.Address(PC_out),
	.inst(Instruction)
);
// حافظه ی نگهداری دستورات است که هر بار به ازای یک آدرس ورودی دستور ذخیره شده را نمایش می دهد

mux2nton #32 m(
	.a(adderOut),
	.b(branchAddr),
	.o(PC_in),
	.s(Branch_taken)
);
// این قسمت برای کنترل کردن دستور branch در نظر گرفته شده است
//که در این صورت از آدرسی که از سمت حافظه می آید باری خواندن دستور استفاده شود
assign PC=PC_in;

endmodule
```

مرحله کدگشایی دستور[4]

این مرحله چهار ماژول اصلی دارد که مهمترین آن control unit است. کارکرد control unit به طور کامل در قسمت پایین تر توضیح داده شده است. در این قسمت multiplexer ای برای هندل کردن hazard وجود دارد به این صورت که در صورتی که دستور نیازمند چندین سایکل باشد و یا hazard اتفاق افتاده باشد ID دستور جدیدی را خروجی نمی دهد و در حالت غیر فعال باقی می ماند.

```
module ID_stage (
       input clk,rst,
       //from IF stage
       input [31:0] Instruction,
       //from WB stage
       input[31:0] Result_WB,
       input writeBackEn,
       input [3:0] Dest_wb,
       //from hazard detection module
       input hazard,
       //from status register
       input [3:0]SR,
       //to next stage
       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,
       //to hazard detect module
       output[3:0] src1, src2,
       output Two_src);

       wire condition_check_out;
       wire WB_EN_cn_out,MEM_R_EN_cn_out,MEM_W_EN_cn_out,B_cn_out,S_cn_out;
       wire[3:0]EXE_CMD_cn_out,src2in;

       wire mux1select;

       Control_unit cn(
       .mode(Instruction[27:26]),
       .op_code(Instruction[24:21]),
       .S(Instruction[20]),
```

[4] Instruction Decode

```
        .Execute_command(EXE_CMD_cn_out),
        .mem_read(MEM_R_EN_cn_out),
        .mem_write(MEM_W_EN_cn_out),
        .WB_enable(WB_EN_cn_out),
        .B(B_cn_out),
        .Update_SR(S_cn_out)
        );

        or r1(mux1select,hazard,~condition_check_out);
        mux2nton #9 mux1(

.a({WB_EN_cn_out,MEM_R_EN_cn_out,MEM_W_EN_cn_out,B_cn_out,S_cn_out,EXE_CMD_cn_out})
,
        .b(9'd0),
        .o({WB_EN,MEM_R_EN,MEM_W_EN,B,S,EXE_CMD}),
        .s(mux1select)
        );

        Condition_Check cc(
        .Cond(Instruction[31:28]),
        .SR(SR),
        .check_output(condition_check_out)
        );

        Register_file rf(
        .clk(clk),
        .rst(rst),
        .src1(Instruction[19:16]),
        .src2(src2in),
        .Dest_wb(Dest_wb),
        .Result_WB(Result_WB),
        .writeBackEn(writeBackEn),
        .reg1(val_Rn),
        .reg2(Val_Rm)
        );

        assign Dest=Instruction[15:12];
        assign imm=Instruction[25];
        assign Signed_imm_24=Instruction[23:0];
        assign Shift_operand=Instruction[11:0];
        assign Two_src=(~imm)|MEM_W_EN;
        assign src1=Instruction[19:16];
        assign src2=src2in;

        mux2nton #4 mux2(
        .a(Instruction[3:0]),
        .b(Instruction[15:12]),
```

```
        .s(MEM_W_EN),
        .o(src2in)
        );


endmodule
```

- ماژول Control Unit

در این قسمت همانطور که مشخص شده است دستور بر اساس mode و opcode تقسیم بندی شده اند و بر اساس این دو قسمت Execution command متفاوت میگیرد همچنین یکسری از سیگنال های کنترلی مقدار دهی می شود

```
module Control_unit(
        input [1:0] mode,
        input [3:0] op_code,
        input S,
        output reg [3:0] Execute_command,
        output reg mem_read,mem_write,WB_enable,B,Update_SR
);
parameter [3:0]
        MOV=4'b1101,MVN=4'b1111, ADD=4'b0100,ADC=4'b0101,SUB=4'b0010, SBC =4'b0110,
AND=4'b0000, ORR=4'b1100, EOR=4'b0001, CMP=4'b1010, TST=4'b1000, LDR_STR=4'b0100;
//OPCODES
parameter [1:0]
        COMPUTE=2'b00, MEMORY=2'b01, BRANCH=2'b10;
//MODES
always@(op_code,mode,S)begin
        {Execute_command,mem_read,mem_write,WB_enable,B,Update_SR}=9'b0000_00000;
        Update_SR = S;
        case(mode)
        COMPUTE:begin
                case(op_code)
                MOV:begin
                        Execute_command = 4'b0001;
                        WB_enable = 1'b1;
                end
                MVN:begin
                        Execute_command = 4'b1001;
                        WB_enable = 1'b1;
                end
```

```
            ADD:begin
                  Execute_command = 4'b0010;
                  WB_enable = 1'b1;
            end
            ADC:begin
                  Execute_command = 4'b0011;
                  WB_enable = 1'b1;
            end
            SUB:begin
                  Execute_command = 4'b0100;
                  WB_enable = 1'b1;
            end
            SBC:begin
                  Execute_command = 4'b0101;
                  WB_enable = 1'b1;
            end
            AND:begin
                  Execute_command = 4'b0110;
                  WB_enable = 1'b1;
            end
            ORR:begin
                  Execute_command = 4'b0111;
                  WB_enable = 1'b1;
            end
            EOR:begin
                  Execute_command = 4'b1000;
                  WB_enable = 1'b1;
            end
            CMP:begin
                  Execute_command = 4'b0100;
            end
            TST:begin
                  Execute_command = 4'b0110;
            end
            endcase
      end

      MEMORY:begin
            Execute_command = 4'b0010;
            mem_read = S;
            mem_write = ~S;
            WB_enable = S;

      end

      BRANCH:begin
            B=1'b1;
```

```
        end
    endcase
end
endmodule
```

- ماژول condition_check

این ماژول بر اساس مقدار رجیستر وضعیت در یک بلاک always، خروجی یک بیتی خود را مشخص می‌کند.

```
module Condition_Check(
        input[3:0] Cond,
        input[3:0] SR,
        output reg check_output);
//SR={N,Z,C,V}
parameter[1:0]
        N_=2'd3,Z_=2'd2,C_=2'd1,V_=2'd0;
always @(Cond,SR) begin
        case(Cond)
        4'd0: check_output=SR[Z_];
        4'd1: check_output=~SR[Z_];
        4'd2: check_output=SR[C_];
        4'd3: check_output=~SR[C_];
        4'd4: check_output=SR[N_];
        4'd5: check_output=~SR[N_];
        4'd6: check_output=SR[V_];
        4'd7: check_output=~SR[V_];
        4'd8: check_output=SR[C_]&~SR[Z_];
        4'd9: check_output=~SR[C_]|SR[Z_];
        4'd10: check_output=(SR[N_]==SR[V_])?1'b1:1'b0;
        4'd11: check_output=(SR[N_]==SR[V_])?1'b0:1'b1;
        4'd12: check_output=~SR[Z_]&((SR[N_]==SR[V_])?1'b1:1'b0);
        4'd13: check_output=SR[Z_]|((SR[N_]==SR[V_])?1'b1:1'b0);
        4'd14: check_output=1'b1;
        4'd15: check_output=1'b0;
        endcase
end
endmodule
```

- ماژول Register File

این ماژول در صورت وجود سیگنال rst مقدار اولیه رجیسترها که برابر شماره آن‌ها است را تعیین می‌کند و پس از آن بر اساس سیگنال writeBackEn تصمیم به نوشتن می‌گیرد. خروجی نیز با assign مشخص شده است.

```
module Register_file(
        input clk,rst,
        input [3:0]src1,src2,Dest_wb,
```

```
       input [31:0] Result_WB,input writeBackEn,
       output [31:0] reg1,reg2
);

reg [31:0] regFile[0:15];
integer i;

always@(negedge clk,posedge rst)begin
       if(rst)begin
       for(i=0;i<16;i=i+1)
              regFile[i]<=i;
       end
    else begin
       if(writeBackEn)
              regFile[Dest_wb]<=Result_WB;
    end
end

assign reg1=regFile[src1];
assign reg2=regFile[src2];

endmodule
```

مرحله اجرا[5]

در این مرحله خروجی دستورات اجرایی مشخص می‌شود. دو ماژول اصلی این مرحله ALU و Val2Generator هستند. ورودی دوم ALU توسط ماژول Val2Generator تعیین می‌شود.

همچنین آدرس barnch در این مرحله توسط ماژول جمع کننده مشخص می‌شود که ورودی اول آن PC و ورودی دوم آن Signed_imm_24 است که sign-extend شده است.

در ادامه کد مربوط به این مرحله آمده است.

```
module EXE_stage (
       input clk,
       input[3:0] EXE_CMD,
       input MEM_R_EN,MEM_W_EN,
```

[5] Execution

```
        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,Br_addr,
        output[3:0]status
);

wire [31:0] Val_2;

Val2Generator v2g(
        .MEM_CMD(MEM_R_EN|MEM_W_EN),
        .imm(imm),
        .Shift_operand(Shift_operand),
        .Val_Rm(Val_Rm),
        .Val_2(Val_2)
);

ALU alu(
        .EXE_CMD(EXE_CMD),
        .Val1(Val_Rn),
        .Val2(Val_2),
        .carry(SR[1]),
        .ALU_res(ALU_result),
        .status(status)
);

adder a(
        .a(PC),
        .b({8'd0,Signed_imm_24}),
        .res(Br_addr)
);

endmodule
```

- ماژول ALU

این ماژول اعمال محاسباتی را انجام می‌دهد. این ماژول علاوه بر عملیات مورد اجرا (EXE_CMD) مقادیر Val2, Val1 و carry را به عنوان ورودی دریافت می‌کند و با توجه دستور ورودی، مقدار خروجی را محاسبه می‌کند. همچنین وضعیت(status) را در قالب یک عدد چهار بیتی خروجی می‌دهد که بیت اول نشان دهنده وقوع

سرریز[6]، بیت دوم نشانه وجود carry، بیت سوم نشانه صفر بودن و بیت چهارم نشانه منفی بودن خروجی است. این ماژول به کمک always و case نوشته شده است. مقدار خروجی در یک بلاک always و مقدار عدد وضعیت در بلاک always دیگر محاسبه می‌شود. در ادامه کد مربوط به این ماژول آمده است.

```
module ALU(
	input carry,
	input[3:0] EXE_CMD,
	input[31:0]Val1,Val2,
	output reg[31:0]ALU_res,
	output reg[3:0]status
);

reg carry1,ncarry;

always@(*)begin
	ALU_res=32'd0;
	carry1=1'b0;
	ncarry=~carry;
	case(EXE_CMD)
	4'd1: ALU_res=Val2;
	4'd9: ALU_res=~Val2;
	4'd2: {carry1,ALU_res}=Val1+Val2;
	4'd3: {carry1,ALU_res}=Val1+Val2+carry;
	4'd4: ALU_res=Val1-Val2;
	4'd5: ALU_res=Val1-Val2-(ncarry);
	4'd6: ALU_res=Val1&Val2;
	4'd7: ALU_res=Val1|Val2;
	4'd8: ALU_res=Val1^Val2;
	endcase
end

always@(ALU_res)begin
	status=4'd0;
	status[0] =
((EXE_CMD==4'd2|EXE_CMD==4'd3)&((ALU_res[31]&~Val1[31]&~Val2[31])|(~ALU_res[31]&Val
1[31]&Val2[31])))|((EXE_CMD==4'd4|EXE_CMD==4'd5)&((ALU_res[31]&~Val1[31]&Val2[31])|
(~ALU_res[31]&Val1[31]&~Val2[31])));
	status[1] = carry1;
	status[2] = ~(|ALU_res);
	status[3] = ALU_res[31];
end

endmodule
```

[6] Overflow

- ماژول Val2Generator

این ماژول ورودی دوم ALU را تعیین می‌کند. پیاده سازی این ماژول با یک بلاک always انجام شده است. در صورتی که دستور ورودی یک دستور دسترسی به حافظه باشد، خروجی برابر عدد Shift_operand خواهد بود. در غیر این صورت با توجه به مقدار imm تصمیم گرفته می‌شود. کد مربوط به این قسمت در ادامه آمده است.

```
module Val2Generator(
	input MEM_CMD,imm,
	input [11:0]Shift_operand,
	input [31:0]Val_Rm,
	output reg[31:0]Val_2);

always@(Shift_operand, imm, Val_Rm, MEM_CMD)begin
	if(MEM_CMD==1'b1)
	Val_2 = Shift_operand;
	else begin
	case(imm)
		1'b1:begin
		Val_2 = {Shift_operand[7:0],24'd0}>>(Shift_operand[11:8]);
		Val_2 = Val_2>>(Shift_operand[11:8]);
		Val_2 = {Val_2[23:0],Val_2[31:24]};
		end
		1'b0:begin
		case(Shift_operand[6:5])
			2'b00: Val_2 = Val_Rm<<Shift_operand[11:7];
			2'b01: Val_2 = Val_Rm>>Shift_operand[11:7];
			2'b10: Val_2 = Val_Rm>>>Shift_operand[11:7];
			2'b11: Val_2 = {Val_Rm,Val_Rm}>>Shift_operand[11:7];
		endcase
		end

	endcase
	end
end
endmodule
```

مرحله حافظه

در این مرحله دسترسی به حافظه انجام می‌شود. تنها ماژول این مرحله حافظه اصلی است که با دریافت آدرس و داده ورودی، بسته به سیگنال کنترلی داده را در حافظه می‌نویسد یا از حافظه می‌خواند. در ادامه پیاده سازی ماژول این مرحله و ماژول حافظه آمده است.

```
module MEM_stage (
	input clk,MEMread,MEMwrite,
	input[31:0] address,data,
	output [31:0] MEM_result
);

data_mem dm(
	.clk(clk),
	.memWrite(MEMwrite),
	.memRead(MEMread),
	.Address(address),
	.writeData(data),
	.readData(MEM_result)
);

endmodule
```

```
module data_mem(input clk,memWrite,memRead,input [15:0]writeData,Address,output reg
[15:0]readData);
reg[15:0] mem[0:127];

initial begin

end

always@(memRead,Address)begin
	if (memRead ==1)
	readData <= mem[Address];
	else
	readData <=16'd0;
end

always@(posedge clk)begin
	if(memWrite==1)
	mem[Address]<=writeData;
end
```

```
endmodule
```

مرحله بازنویسی[7]

در این مرحله تنها یک multiplexer وجود دارد که از بین خروجی ALU و خروجی حافظه، مقداری که باید در رجیستر فایل نوشته بشود را مشخص می‌کند. انتخاب کنند این multiplexer سیگنال MEM_R_EN یا همان خواندن از حافظه است، چرا که در صورتی که دستور مرحله قبل خواندن از حافظه باشد، خروجی حافظه و در غیر این صورت خروجی ALU انتخاب می‌شود. در ادامه پیاده‌سازی این مرحله آمده است.

```
module WB_stage (
	input[31:0] ALU_result,MEM_result,
	input MEM_R_en,
	output [31:0] out
);

mux2nton #32 mux(
	.a(ALU_result),
	.b(MEM_result),
	.s(MEM_R_en),
	.o(out)
);
endmodule
```

ماژول‌های دیگر

- ماژول تشخیص مخاطره[8]

این ماژول وظیفه تشخیص مخاطره داده‌ای در پردازنده را به عهده دارد و به کمک assign پیاده‌سازی شده است. کد آن در ادامه آمده است.

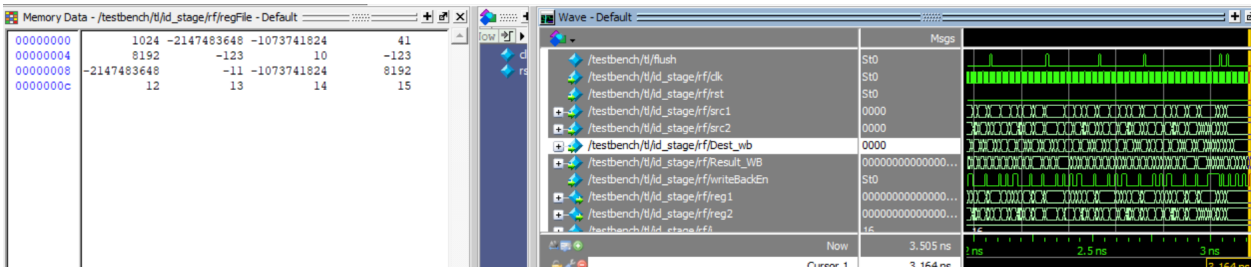
```
module hazard_Detection_unit(
	input [3:0]src1,
	input [3:0]src2,
	input [3:0]Exe_Dest,
```

[7] Write-Back
[8] Hazard Detection

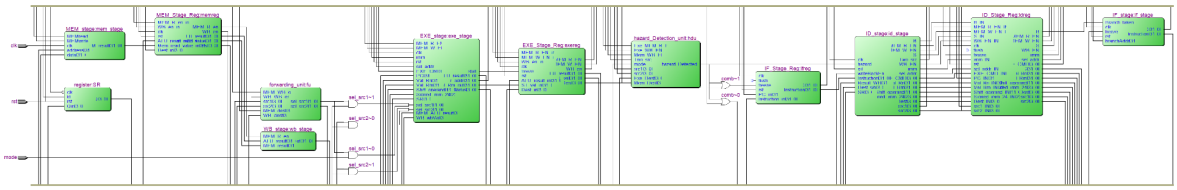
```
        input Exe_WB_EN,
        input [3:0]Mem_Dest,
        input Mem_WB_EN,
        input Two_src,
        output hazard_Detected);

        assign hazard_Detected=((src1==Exe_Dest)&Exe_WB_EN)?1'b1:
                               ((src1==Mem_Dest)&Mem_WB_EN)?1'b1:
                               ((src2==Exe_Dest)&Exe_WB_EN&Two_src)?1'b1:
                               ((src2==Mem_Dest)&Mem_WB_EN&Two_src)?1'b1:1'b0;

endmodule
```

سنتز و اجرا

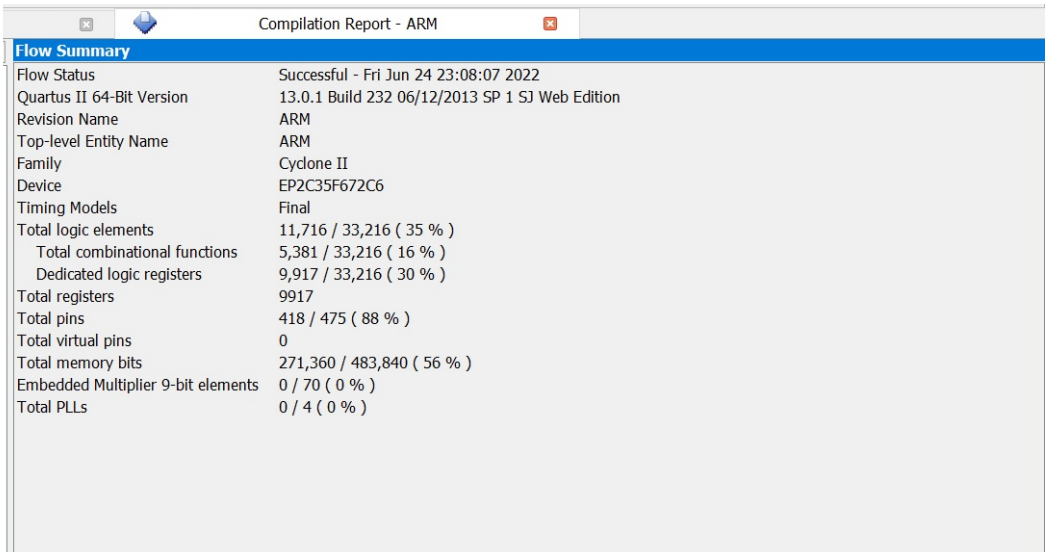
خروجی حاصل از شبیه‌سازی کد در ادامه آمده است و مشاهده می‌شود که مرتب سازی به درستی انجام شده است. زمان اجرای برنامه با کلاک 10ps حدود 3.164ns شد. همچنین در زمان اجرا روی برد در آزمایشگاه حدودا 310 سیکل زمان برد.



در ادامه خروجی RTL تولید شده توسط Quartus آمده است که شباهت قابل قبولی با مدل طراحی شده توسط ما دارد.



در ادامه report حاصل از سنتز آمده است:



تعداد سیکل‌های اجرای برنامه: 310

$$CPI = \frac{310}{49} = 6.32$$

افزودن تکنیک ارسال به جلو[9]

برای ایجاد توقف کمتر و افزایش سرعت پردازنده، از تکنیک ارسال به جلو استفاده می‌شود. در بخش دوم آزمایشگاه این تکنیک به پردازنده افزوده شده است، به این صورت که یک بیت mode به ورودی‌های پردازنده اضافه شده است که در صورتی که این بیت صفر باشد، پردازنده در حالت عادی و اگر یک باشد، با forwarding کار می‌کند. در ادامه تغییرات انجام شده بر روی کد توضیح داده می‌شود.

ماژول های اضافه شده

- ماژول forwarding_unit

در صورتی که بین دو دستور مخاطره داده‌ای وجود داشته باشد، در صورتی که داده مورد نظر دستور در مراحل جلوتر حاضر باشد، می‌توان به جای stall کردن، داده را به مرحله قبل منتقل کرد. از ماژول forwarding_unit

[9] Forwarding

برای تشخیص امکان پذیر بودن این موضوع استفاده می‌شود. در صورتی که داده مورد نیاز مرحله اجرا در مرحله حافظه موجود باشد خروجی برابر ۱، اگر داده مورد نیاز مرحله اجرا در مرحله بازنویسی موجود باشد خروجی برابر ۲ و در غیر این صورت برابر ۰ خواهد شد. از این خروجی در مرحله اجرا استفاده می‌شود.

```
module forwarding_unit(
	//From EXE Phase:
	input [3:0]src1,src2,
	//From MEM Phase:
	input [3:0]MEM_dest,
	input MEM_WB_en,
	//From WB Phase:
	input [3:0]WB_dest,
	input WB_WB_en,
	//To EXE Phase:
	output [1:0]sel_src1,sel_src2
);
assign sel_src1 = (src1==MEM_dest & MEM_WB_en)?2'b01:(src1==WB_dest &
WB_WB_en)?2'b10:2'b00;
assign sel_src2 = (src2==MEM_dest & MEM_WB_en)?2'b01:(src2==WB_dest &
WB_WB_en)?2'b10:2'b00;

endmodule
```

تغییرات بخش‌های قبلی

- مرحله اجرا

به کد این مرحله دو multiplexer اضافه کردیم که در صورت وجود وابستگی داده‌ای بین ورودی‌های اول و دوم ALU و مراحل بعدی، از داده‌های موجود در آن مراحل به عنوان ورودی استفاده می‌کند. در ادامه کد تغییر یافته آورده شده و دو ماژول اضافه شده مشخص شده‌اند.

```
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,

        //forwarding unit:
        input[1:0]sel_src1,sel_src2,
        input[31:0]MEM_ALU_result, WB_wbVal,

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

wire [31:0] src1,src2,Val_2;


mux4nton mux1(
        .a(Val_Rn),
        .b(MEM_ALU_result),
        .c(WB_wbVal),
        .d(32'd0),
        .s(sel_src1),
        .o(src1)
);

mux4nton mux2(
        .a(Val_Rm),
        .b(MEM_ALU_result),
        .c(WB_wbVal),
        .d(32'd0),
        .s(sel_src2),
        .o(src2)
);
assign Val_Rm_out=src2;

Val2Generator v2g(
        .MEM_CMD(MEM_R_EN|MEM_W_EN),
        .imm(imm),
        .Shift_operand(Shift_operand),
        .Val_Rm(src2),
        .Val_2(Val_2)
);

ALU alu(
        .EXE_CMD(EXE_CMD),
        .Val1(src1),
        .Val2(Val_2),
        .carry(SR[1]),
```

```
        .ALU_res(ALU_result),
        .status(status)
);

adder a(
        .a(PC),
        .b({{6{Signed_imm_24[23]}},Signed_imm_24,2'b00}),
        .res(Br_addr)
);

endmodule
```

- ماژول تشخیص مخاطره

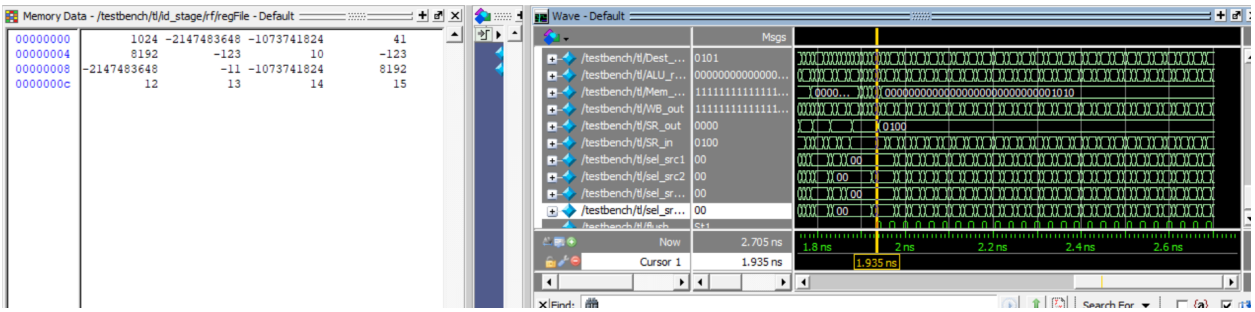
با توجه به این که این ماژول در دو حالت کار می‌کند، ورودی mode را به این ماژول اضافه کردیم. در صورتی که mode برابر صفر باشد خروجی مشابه بخش‌های قبل است. در صورت ۱ بودن این مقدار، خروجی برای حالت forwarding تولید می‌شود. تنها حالتی که امکان ارسال داده به جلو وجود ندارد و نیاز به stall داریم حالتی است که داده مورد نیاز مرحله اجرا باید از حافظه خوانده شود و دستور مربوط به این کار اکنون در مرحله حافظه قرار دارد. در این صورت داده هنوز آماده نیست و باید پردازنده stall بشود. در ادامه کد مربوط به این بخش آمده است.

```
module hazard_Detection_unit(
        input [3:0]src1,
        input [3:0]src2,
        input [3:0]Exe_Dest,
        input Exe_WB_EN,
        input [3:0]Mem_Dest,
        input Mem_WB_EN,
        input Exe_MEM_R_EN,
        input Two_src,
        input mode,
        output hazard_Detected);

        wire hazard_Detected_0,hazard_Detected_1;

        assign hazard_Detected_0=((src1==Exe_Dest)&Exe_WB_EN)?1'b1:
                                 ((src1==Mem_Dest)&Mem_WB_EN)?1'b1:
                                 ((src2==Exe_Dest)&Exe_WB_EN&Two_src)?1'b1:
```

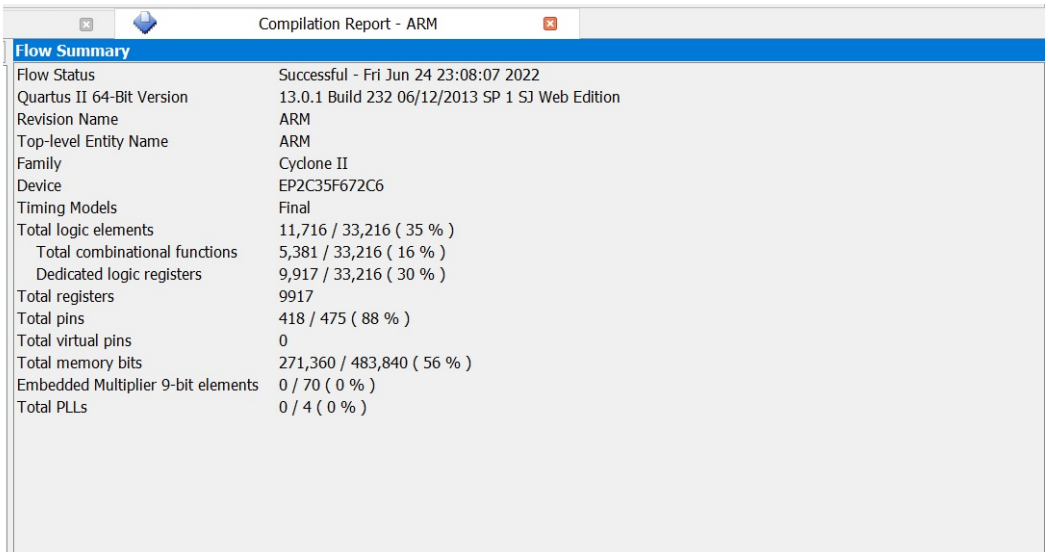
```
                        ((src2==Mem_Dest)&Mem_WB_EN&Two_src)?1'b1:1'b0;
      assign hazard_Detected_1=((src1==Exe_Dest)&Exe_WB_EN&Exe_MEM_R_EN)?1'b1:

((src2==Exe_Dest)&Exe_WB_EN&Two_src&Exe_MEM_R_EN)?1'b1:1'b0;

      assign hazard_Detected=(mode==0)?hazard_Detected_0:hazard_Detected_1;

endmodule
```

سنتز و اجرا

خروجی حاصل از شبیه‌سازی کد در ادامه آمده است و مشاهده می‌شود که مرتب سازی به درستی انجام شده است. زمان اجرای برنامه با کلاک 10ps حدود 1.635ns ثانیه شد. همچنین در زمان اجرا روی برد در آزمایشگاه حدود 190 سیکل زمان برد. مشاهده می‌شود که تعداد سیکل‌ها نسبت به حالت قبلی کاهش یافته و کارایی پردازنده افزایش یافته است.



در ادامه report حاصل از سنتز آمده است:


Compilation Report - ARM

Flow Summary

Flow Status	Successful - Fri Jun 24 23:08:07 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	11,716 / 33,216 (35 %)
Total combinational functions	5,381 / 33,216 (16 %)
Dedicated logic registers	9,917 / 33,216 (30 %)
Total registers	9917
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	271,360 / 483,840 (56 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

تعداد سیکل‌های اجرای برنامه: 190

$$CPI = \frac{190}{49} = 3.87$$

افزودن SRAM به عنوان حافظه اصلی

در بخش سوم آزمایشگاه، از SRAM به عنوان حافظه اصلی استفاده می‌کنیم. با توجه به این که SRAM یک حافظه خارجی است، برای ارتباط با آن به controller نیاز داریم و نیاز است که پردازنده برای چند سیکل متوقف بشود. دسترسی به SRAM در ۶ سیکل انجام می‌شود.

ماژول های اضافه شده

- ماژول SRAM_controller

این ماژول وظیفه ارتباط با SRAM و فرستادن سیگنال‌های مورد نیاز آن را به عهده دارد. این ماژول به صورت یک کنترلر با ۶ استیت طراحی شده است. در ابتدا در استیت 0 قرار داریم و تا زمانی که سیگنال خواندن یا نوشتن در حافظه دریافت نشود در همین استیت می‌مانیم. در صورتی که سیگنال نوشتن یا خواندن دریافت بشود به استیت‌های بعدی می‌رویم و پس از رسیدن به استیت 5 مجددا به استیت 0 باز می‌گردیم. با توجه به

این که SRAM دارای بلاک‌های ۱۶ بیتی و داده های ما ۳۲ بیتی است، در استیت اول ۱۶ بیت اول داده و در استیت دوم ۱۶ بیت دوم داده روی SRAM_DQ قرار داده می‌شود. در ادامه کد مربوط به این بخش آمده است.

```
module SRAM_controller(
       input clk,
       input rst,
       //From Memory Stage
       input wr_en,
       input rd_en,
       input[31:0]address,
       input [31:0]write_Data,
       //to next stage
       output[31:0] ReadData,
       //to freeze other stages
       output reg ready,

       inout [15:0]SRAM_DQ,
       output [17:0]SRAM_ADDR,
       output reg SRAM_UB_EN,
       output reg SRAM_LB_EN,
       output reg SRAM_WE_EN,
       output reg SRAM_CE_EN,
       output reg SRAM_OE_EN);

       reg [2:0] ps, ns;
       reg [17:0]addr;
       reg [15:0]writeDataOut;
       reg [31:0]rawReadData;

       assign SRAM_DQ=(SRAM_WE_EN==0)?writeDataOut:16'bzzzzzzzzzzzzzzzz;
       assign SRAM_ADDR=addr;
       assign ReadData=rawReadData;

       always@(posedge clk,posedge rst)begin
       if(rst) ps<=2'b00;
       else ps<=ns;
       end

       always@(ps,wr_en,rd_en)begin
       if(ps==3'b101)
              ns=3'b000;
       else if(wr_en|rd_en)
              ns=ps+1;
       else
              ns=3'b000;
       end
```

```
always@(ps,wr_en,rd_en)begin
{SRAM_WE_EN,SRAM_UB_EN,SRAM_LB_EN,SRAM_CE_EN,SRAM_OE_EN,ready}=6'b100000;
if(wr_en)
	case(ps)
	3'b000: begin
	ready=~(wr_en|rd_en);
	end
	3'b001: begin
	writeDataOut=write_Data[15:0];
	addr=((address>>2)<<2)-32'd1024;
	SRAM_WE_EN=1'b0;
	end
	3'b010: begin
	writeDataOut=write_Data[31:16];
	addr=(((address>>2)<<2)-32'd1024)+2;
	SRAM_WE_EN=1'b0;
	end
	3'b101: begin
	ready=1'b1;
	end
	endcase
else if(rd_en)
	case(ps)
	3'b000: begin
	ready=~(wr_en|rd_en);
	end
	3'b001: begin
	addr=((address>>2)<<2)-32'd1024;
	end
	3'b010: begin
	rawReadData[15:0]=SRAM_DQ;
	addr=(((address>>2)<<2)-32'd1024)+2;
	end
	3'b011: begin
	rawReadData[31:16]=SRAM_DQ;
	end
	3'b101: begin
	ready=1'b1;
	end
	endcase
else
	case(ps)
	3'b000: begin
	ready=~(wr_en|rd_en);
	end
	endcase
```

```
        end

endmodule
```

تغییرات بخش‌های قبلی

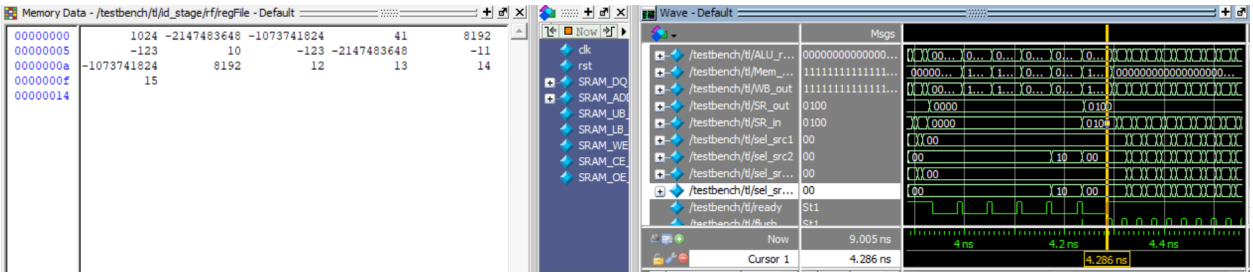
به تمام رجیسترهای بین مراحل تا پیش از مرحله حافظه سیگنال freeze اضافه شده و نقیض سیگنال ready به آن متصل شده است. همچنین ماژول SRAM_controller جایگزین ماژول حافظه اصلی در مرحله حافظه شده است.

```
module MEM_stage (
        input clk,rst,MEMread,MEMwrite,
        input[31:0] address,data,
        output [31:0] MEM_result,
        output ready,
        inout [15:0]SRAM_DQ,
        output [17:0]SRAM_ADDR,
        output SRAM_LB_EN,
        output SRAM_UB_EN,
        output SRAM_WE_EN,
        output SRAM_CE_EN,
        output SRAM_OE_EN);

SRAM_controller sc(
        .clk(clk),
        .rst(rst),
        .wr_en(MEMwrite),
        .rd_en(MEMread),
        .address(address),
        .write_Data(data),
        .ReadData(MEM_result),
        .ready(ready),

        .SRAM_DQ(SRAM_DQ),
        .SRAM_ADDR(SRAM_ADDR),
        .SRAM_UB_EN(SRAM_UB_EN),
        .SRAM_LB_EN(SRAM_LB_EN),
        .SRAM_WE_EN(SRAM_WE_EN),
        .SRAM_CE_EN(SRAM_CE_EN),
        .SRAM_OE_EN(SRAM_OE_EN)
        );
```

```
endmodule
```

سنتز و اجرا

خروجی حاصل از شبیه‌سازی کد در ادامه آمده است و مشاهده می‌شود که مرتب سازی به درستی انجام شده است. زمان اجرای برنامه با کلاک 10ps حدود 4.286ns شد. همچنین در زمان اجرا روی برد در آزمایشگاه حدود 420 سیکل زمان برد. مشاهده می‌شود که تعداد سیکل‌ها نسبت به حالت قبلی که 190 سیکل بود افزایش یافته و کارایی پردازنده کاهش یافته است.



در ادامه report حاصل از سنتز آمده است:

Flow Summary

Flow Status	Successful - Fri Jun 24 23:34:52 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	11,713 / 33,216 (35 %)
Total combinational functions	5,386 / 33,216 (16 %)
Dedicated logic registers	9,849 / 33,216 (30 %)
Total registers	9849
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	267,264 / 483,840 (55 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

مشاهده می‌شود که هزینه سخت افزاری اندکی کاهش یافته است که با توجه به حذف شدن حافظه داخلی مورد انتظار بود.

افزودن حافظه نهان[10]

با توجه به هزینه زمانی زیاد ارتباط به SRAM، در این فاز از آزمایش به پردازنده حافظه نهان اضافه کردیم تا در صورت موجود بودن داده مورد نیاز در آن، از آن داده استفاده کرده و مجبور به stall کردن پردازنده نشویم.

ماژول های اضافه شده

- ماژول Cache_controller

این ماژول سیگنال‌های کنترلی مناسب را به Cache و SRAM_controller ارسال می‌کند. طراحی آن به صورت یک کنترلر با ۴ استیت است. استیت 0 استیت شروع است. در صورتی که قصد خواند از حافظه را داشته باشیم و داده در cache نباشد به استیت 1 می‌رویم و سیگنال خواندن از حافظه به SRAM_controller و سیگنال ذخیره داده به cache داده می‌شود. در صورتی که قصد نوشتن در حافظه را داشته باشیم به استیت 2 می‌رویم و سیگنال خواندن از حافظه به SRAM_controller و cache داده می‌شود.

```
module cache_controller(
    input clk,
    input rst,

    input [31:0] address,
    input [31:0] wdata,
    input MEM_R_EN,
    input MEM_W_EN,
    output [31:0] rdata,
    output reg ready,

    output [31:0] sram_address,
    output [31:0] sram_wdata,
    output reg write,
    output reg read,
    input [63:0] sram_rdata,
    input sram_ready );

    wire hit;
    wire [31:0] cache_rdata;
```

[10] Cache

```
    reg [2:0] ns,ps;

    reg cache_write,cache_read,cache_mem_write;

    cache c(
        .clk(clk),
        .rst(rst),
        .cache_read(cache_read),
        .mem_write(cache_mem_write),
        .cache_write(cache_write),
        .write_data(sram_rdata),
        .address(address[18:0]),
        .hit(hit),
        .read_data(cache_rdata)
    );

    assign sram_wdata=wdata;
    assign sram_address=address;
    assign rdata=(hit)?cache_rdata:(address[2])?sram_rdata[63:32]:sram_rdata[31:0];

    always@(posedge clk,posedge rst)begin
        if(rst)
            ps<=3'b000;
        else
            ps<=ns;
    end

    always@(ps,sram_ready,MEM_R_EN,MEM_W_EN,hit)begin
        {cache_write,cache_read,cache_mem_write,write,read,ready}=6'b000000;
        case(ps)
            3'b000: begin cache_read=1'b1;
ready=(MEM_W_EN)?1'b0:(MEM_R_EN)?hit|sram_ready:1'b1; end
            3'b001: {read,cache_write}={1'b1,sram_ready};
            3'b010: {cache_mem_write,write}=2'b11;
            3'b100: ready=1'b1;
        endcase
    end

    always@(ps,MEM_R_EN,hit,MEM_W_EN,sram_ready)begin
        case(ps)
            3'b000: ns=(MEM_R_EN&~hit)?3'b001:(MEM_W_EN)?3'b010:3'b000;
            3'b001: ns=(sram_ready)?3'b100:3'b001;
            3'b010: ns=(sram_ready)?3'b100:3'b010;
            3'b100: ns=3'b000;
        endcase
    end
```

```
endmodule
```

- ماژول Cache

در ابتدا حافظه مورد نیاز این ماژول را تعریف کردیم. همچنین خروجی hit را تعیین کردیم که در صورتی اتفاق می‌افتد که tag داده با قسمت tag آدرس ورودی یکسان باشد و داده valid باشد.

```
module cache(
	input clk,rst,cache_read,mem_write,cache_write,
	input [18:0] address,
	input [63:0] write_data,
	output hit,
	output [31:0] read_data
);

	wire [9:0] tag;
	wire [5:0] index;
	wire [2:0] offset;

	wire [63:0] raw_read_data;

	assign {tag, index, offset} = address;
	assign read_data=(offset[2])?raw_read_data[63:32]:raw_read_data[31:0];

	reg [63:0] data_0 [0:63];
	reg [63:0] data_1 [0:63];

	reg [9:0] tag_0 [0:63];
	reg [9:0] tag_1 [0:63];

	reg valid_0 [0:63];
	reg valid_1 [0:63];

	reg lru [0:63];

	wire hit_0, hit_1;
	assign hit_0 = (tag_0[index] == tag) & valid_0[index];
	assign hit_1 = (tag_1[index] == tag) & valid_1[index];

	assign hit = hit_0 | hit_1;
```

سپس مقدار اولیه خانه‌های cache را تعیین می‌کنیم.

```
integer i;
```

```
initial for (i=0; i<64 ; i=i+1) begin
       valid_0[i] = 1'b0;
       valid_1[i] = 1'b0;
       tag_0[i] = 10'b0;
       tag_1[i] = 10'b0;
       lru[i] = 1'b0;
end
```

در ادامه بخش مربوط به خواندن از cache نوشته شده است. در این قسمت lru به روز می‌شود و همچنین داده خروجی با توجه به این که hit برای کدام ستون داده اتفاق افتاده است تعیین می‌شود.

```
always @(posedge clk)
       if (cache_read & hit)
              lru[index] <= hit_0 ? 1'b0: 1'b1;

mux2nton #64 mux1(
       .a(data_0[index]),
       .b(data_1[index]),
       .s(~hit_0),
       .o(raw_read_data)
);
```

در ادامه بخش مربوط به نوشتن در حافظه نوشته شده است. در این قسمت نیاز است که کد valid خانه مربوطه را برابر صفر قرار بدهیم و lru را به روز کنیم.

```
always @(posedge clk) begin
       if (mem_write & hit) begin
              if (hit_0) begin
              valid_0[index] <= 1'b0;
              lru[index] <= 1'b1;
              end

              else if (hit_1) begin
              valid_1[index] <= 1'b0;
              lru[index] <= 1'b0;
              end
       end
end
```

در نهایت بخش مربوط به نوشتن در cache آمده است. در این قسمت با توجه به مقدار lru تصمیم می‌گیریم که در کدام ستون cache بنویسیم.

```
always @(posedge clk) begin
       if (cache_write) begin
              if (lru[index]==0) begin
```

```
                data_1[index] <= write_data;
                tag_1[index] <= tag;
                valid_1[index] <= 1'b1;
                end

                else if (lru[index]==1) begin
                data_0[index] <= write_data;
                tag_0[index] <= tag;
                valid_0[index] <= 1'b1;
                end
        end
end
```

تغییرات بخش‌های قبلی

- ماژول SRAM_controller

با توجه به این که داده‌های cache از ۶۴ بیت استفاده می‌کنند، این ماژول را تغییر دادیم تا به جای ۳۲ بیت، ۶۴ بیت از حافظه خارجی بخواند. در ادامه قسمت تغییر کرده آورده شده است.

```
always@(ps,wr_en,rd_en)begin
        {SRAM_WE_EN,SRAM_UB_EN,SRAM_LB_EN,SRAM_CE_EN,SRAM_OE_EN,ready}=6'b100000;
        if(wr_en)
                case(ps)
                3'b001: begin
                writeDataOut=write_Data[15:0];
                addr=((address>>2)<<2)-32'd1024;
                SRAM_WE_EN=1'b0;
                end
                3'b010: begin
                writeDataOut=write_Data[31:16];
                addr=(((address>>2)<<2)-32'd1024)+2;
                SRAM_WE_EN=1'b0;
                end
                3'b101: begin
                ready=1'b1;
                end
                endcase
        else if(rd_en)
                case(ps)
                3'b001: begin
```

```
            addr=((address>>3)<<3)-32'd1024;
            end
            3'b010: begin
            rawReadData[15:0]=SRAM_DQ;
            addr=(((address>>3)<<3)-32'd1024)+2;
            end
            3'b011: begin
            rawReadData[31:16]=SRAM_DQ;
            addr=(((address>>3)<<3)-32'd1024)+4;
            end
            3'b100:begin
            rawReadData[47:32]=SRAM_DQ;
            addr=(((address>>3)<<3)-32'd1024)+6;
            end
            3'b101: begin
            rawReadData[63:48]=SRAM_DQ;
            ready=1'b1;
            end
            endcase
      else
            case(ps)
            3'b000: begin
            ready=1'b0;
            end
            endcase
      end
```

- مرحله حافظه

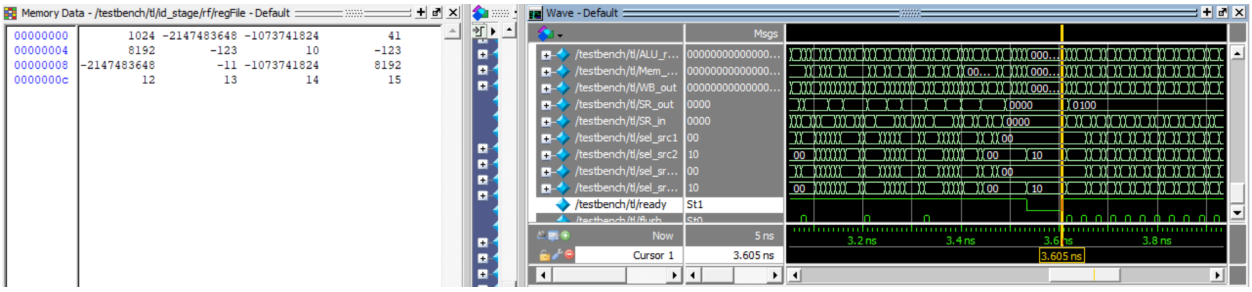
اتصالات این مرحله را با توجه به اضافه شدن cache تغییر دادیم و cache را واسط میان SRAM_controller و بیرون قرار دادیم، به طوری که تمام ارتباطات بین این دو از طریق cache_controller است.

```
module MEM_stage (
      input clk,rst,MEMread,MEMwrite,
      input[31:0] address,data,
      output [31:0] MEM_result,
      output ready,
      inout [15:0]SRAM_DQ,
      output [17:0]SRAM_ADDR,
      output SRAM_LB_EN,
      output SRAM_UB_EN,
      output SRAM_WE_EN,
      output SRAM_CE_EN,
      output SRAM_OE_EN);

wire [31:0]SRAM_addr,SRAM_wdata;
```

```
wire [63:0] SRAM_rdata;
wire SRAM_ready,SRAM_W_EN,SRAM_R_EN;

cache_controller cc(
       .clk(clk),
       .rst(rst),

       .address(address),
       .wdata(data),
       .MEM_R_EN(MEMread),
       .MEM_W_EN(MEMwrite),
       .rdata(MEM_result),
       .ready(ready),

       .sram_address(SRAM_addr),
       .sram_wdata(SRAM_wdata),
       .write(SRAM_W_EN),
       .read(SRAM_R_EN),
       .sram_rdata(SRAM_rdata),
       .sram_ready(SRAM_ready)
);

SRAM_controller sc(
       .clk(clk),
       .rst(rst),
       .wr_en(SRAM_W_EN),
       .rd_en(SRAM_R_EN),
       .address(SRAM_addr),
       .write_Data(SRAM_wdata),
       .ReadData(SRAM_rdata),
       .ready(SRAM_ready),

       .SRAM_DQ(SRAM_DQ),
       .SRAM_ADDR(SRAM_ADDR),
       .SRAM_UB_EN(SRAM_UB_EN),
       .SRAM_LB_EN(SRAM_LB_EN),
       .SRAM_WE_EN(SRAM_WE_EN),
       .SRAM_CE_EN(SRAM_CE_EN),
       .SRAM_OE_EN(SRAM_OE_EN)
       );

endmodule
```

سنتز و اجرا

خروجی حاصل از شبیه‌سازی کد در ادامه آمده است و مشاهده می‌شود که مرتب سازی به درستی انجام شده است. زمان اجرای برنامه با کلاک 10ps حدود 3.605ns شد. همچنین در زمان اجرا روی برد در آزمایشگاه حدود 360 سیکل زمان برد. مشاهده می‌شود که تعداد سیکل‌ها نسبت به حالت با SRAM و بدون حافظه که 420 سیکل بود کاهش یافته و کارایی پردازنده بهتر شده است، اما همچنان کارایی پردازنده از حالت حافظه داخلی که 190 سیکل بود کمتر است، که مورد انتظار ما نیز بود.



در ادامه report حاصل از سنتز آمده است:

Compilation Report - ARM

Flow Summary

Flow Status	Successful - Fri Jun 24 10:54:35 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	18,554 / 33,216 (56 %)
Total combinational functions	9,687 / 33,216 (29 %)
Dedicated logic registers	15,333 / 33,216 (46 %)
Total registers	15333
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	272,640 / 483,840 (56 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)