

گزارش کار آزمایشگاه معماری کامپیوتر

علی بنی هاشمی - محمد صادق عقیلی

۸۱.۱.۰۰۲۷۴ - ۸۱.۱.۰۰۲۴۵

آزمایش اول:

برای شروع باید برای هر ۵ بخش یک ماژول ساخته و رجیستر های اصلی داخل پایپ لاین را قرار دهیم. اصطلاحا با اینکار black box هایی ساختیم که می توانیم از طریق آنها جلو رفتن دستورات در هر کلاک داخل پایپ لاین را بررسی کنیم، مطابق صورت پروژه در ابتدا همه ی آنها صرفا ورودی را در هر کلاک به خروجی می فرستند.

پیاده سازی ماژول واکشی: وظیفه این ماژول صرفا انتخاب دستور بعدی و فرستادن آن به ماژول بعدی است.

دستور بعدی همواره یا دستور بعدی در برنامه است یا دستوری که پرش آن را مشخص می کند به همین دلیل شماره دستور را از طریق یک mux انتخاب می کنیم.

دلیل وجود adder ایجاد شماره دستور بعدی که برابر شماره دستور فعلی به علاوه ۴ است می باشد. شماره دستور انتخابی ما دستور مورد نظر را از یک حافظه استخراج کرده و دستور و شماره دستور داخل رجیستر پایپ لاین ذخیره می شوند.

```
Ln# 1 module ifetch(
2     input clk, rst,
3     input branchTaken, freeze,
4     input [31:0] branchAddr,
5     output [31:0] pc, instruction
6 );
7     wire [31:0] pcRegIn, pcRegOut, pcAdderOut;
8
9     Register #(32) pcReg(
10         .clk(clk),
11         .rst(rst),
12         .in(pcRegIn),
13         .ld(~freeze),
14         .clr(1'b0),
15         .out(pcRegOut)
16     );
17
18     Adder #(32) pcAdder(
19         .a(pcRegOut),
20         .b(32'd4),
21         .out(pcAdderOut)
22     );
23
24     Mux2To1 #(32) pcMux(
25         .a0(pcAdderOut),
26         .a1(branchAddr),
27         .sel(branchTaken),
28         .out(pcRegIn)
29     );
30
31     InstructionMemory instMem(
32         .pc(pcRegOut),
33         .inst(instruction)
34     );
35
36     assign pc = pcAdderOut;
37 endmodule
```

در این مازول ۴ مازول ساده دیگر طبق آنچه توضیح داده شد به همدیگر متصل شده اند.

```
1 module InstructionMemory #(parameter Count = 1024)
2   input [31:0] pc,
3   output reg [31:0] inst
4   );
5   wire [31:0] adr;
6   assign adr = {pc[31:2], 2'b00}; // Align address to the word boundary
7
8   always @(adr) begin
9     case (adr)
10       32'd0: inst = 32'b1110_00_1_1101_0_0000_0000_000000010100; // MOV R0,
11       32'd4: inst = 32'b1110_00_1_1101_0_0000_0001_101000000001; // MOV R1,
12       32'd8: inst = 32'b1110_00_1_1101_0_0000_0010_000100000011; // MOV R2,
13       32'd12: inst = 32'b1110_00_0_0100_1_0010_0011_000000000010; // ADDS R3,
14       32'd16: inst = 32'b1110_00_0_0101_0_0000_0100_000000000000; // ADC R4,
15       32'd20: inst = 32'b1110_00_0_0100_0_0100_0100_00000000100; // SUB R5,
16       32'd24: inst = 32'b1110_00_0_0110_0_0000_0110_000010100000; // SBC R6,
17       32'd28: inst = 32'b1110_00_0_1100_0_0101_0111_000101000010; // ORR R7,
18       32'd32: inst = 32'b1110_00_0_0000_0_0111_1000_000000000011; // AND R8,
19       32'd36: inst = 32'b1110_00_0_1111_0_0000_1001_0000000000110; // MVN R9,
20       32'd40: inst = 32'b1110_00_0_0001_0_0100_1010_0000000000101; // EOR R10,
21       32'd44: inst = 32'b1110_00_0_1010_1_1000_0000_00000000110; // CMP R8,
22       32'd48: inst = 32'b0001_00_0_0100_0_0001_0001_00000000001; // ADDNE R1,
23       32'd52: inst = 32'b1110_00_0_1000_1_1001_0000_000000001000; // TST R9,
24       32'd56: inst = 32'b0000_00_0_0100_0_0100_0010_000000000010; // ADDEQ R2,
25       32'd60: inst = 32'b1110_00_1_1101_0_0000_0000_101100000001; // MOV R0,
26       32'd64: inst = 32'b1110_01_0_0100_0_0000_0001_000000000000; // STR R1,
27       32'd68: inst = 32'b1110_01_0_0100_1_0000_1011_000000000000; // LDR R11,
28       32'd72: inst = 32'b1110_01_0_0100_0_0000_0010_0000000000100; // STR R2,
29       32'd76: inst = 32'b1110_01_0_0100_0_0000_0011_000000001000; // STR R3,
30       32'd80: inst = 32'b1110_01_0_0100_0_0000_0100_000000001101; // STR R4,
31       32'd84: inst = 32'b1110_01_0_0100_0_0000_0101_0000000010000; // STR R5,
32       32'd88: inst = 32'b1110_01_0_0100_0_0000_0110_0000000010100; // STR R6,
33       32'd92: inst = 32'b1110_01_0_0100_1_0000_1010_0000000000100; // LDR R10,
34       32'd96: inst = 32'b1110_01_0_0100_0_0000_0111_0000000011000; // STR R7,
```

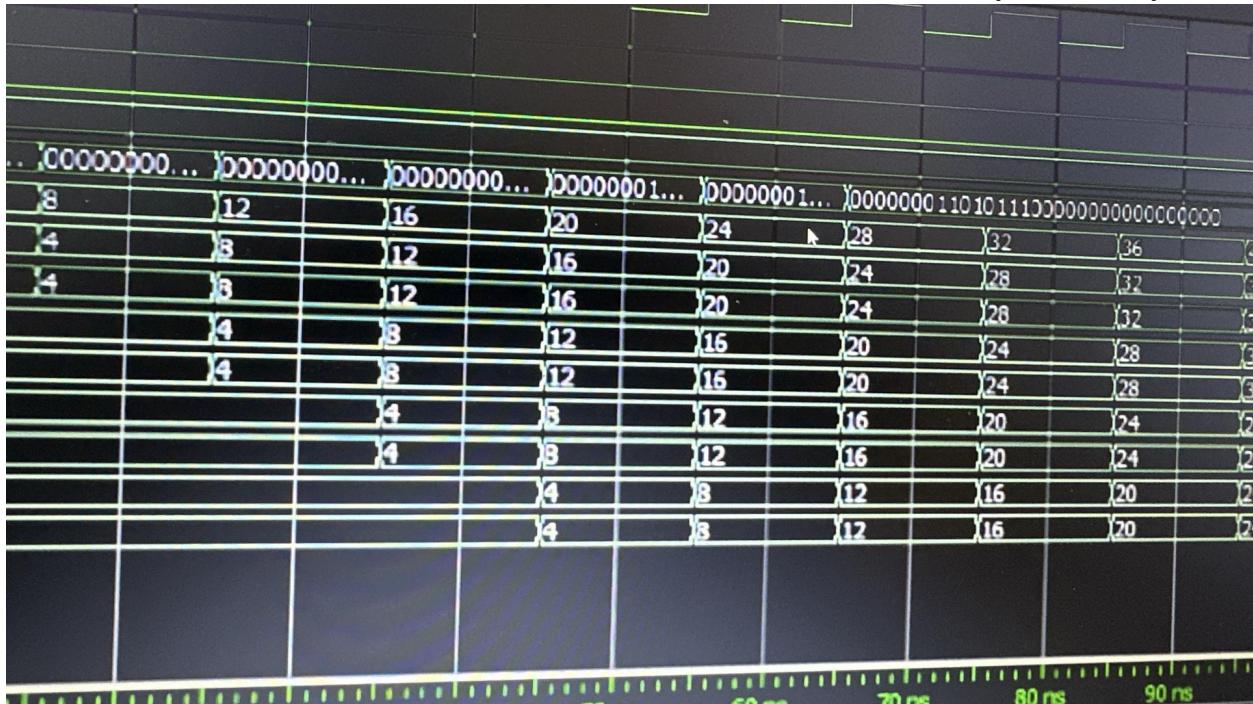
در مازول IMemory یک حافظه ساده داریم که شامل همه دستورات یک برنامه است. هر یک مربوط به یک شماره دستور مشخص هستند و از طریق آن شماره دستور PC به آنها دسترسی پیدا می‌کنیم.

مازوی های جمع کننده و رجیستر و ماکس هم پیاده سازی ساده ای دارند که برای ما آشنا است.

در نهایت رجیستر مربوط به این مازول در پایپ لاین که قرار است شماره دستور و دستور را در خود نگه دارد پیاده سازی می‌کنیم.

```
1 module firstreg(
2   input clk, rst,
3   input freeze, flush,
4   input [31:0] pcIn, instructionIn,
5   output [31:0] pcOut, instructionOut
6 );
7   Register #(32) pcReg(
8     .clk(clk), .rst(rst),
9     .in(pcIn), .ld(~freeze), .clr(flush),
10    .out(pcOut)
11  );
12
13  Register #(32) instReg(
14    .clk(clk), .rst(rst),
15    .in(instructionIn), .ld(~freeze), .clr(flush),
16    .out(instructionOut)
17  );
18 endmodule
```

حال یک مازول تاپ ایجاد کرده، دو مازول مربوط به واکشی را به همدیگر متصل می‌کنیم و باقی مازول‌ها را به حالت blackbox قرار می‌دهیم که علاوه بر بررسی صحت عملکرد واکشی پایپ لاین هم بررسی کنیم.



همانطور که می بینیم شماره دستورات در هر کلاک به یک مازول جلوتر می رود همچنین خود دستورات را می توانیم در رجیستر مربوط به ifetch ببینیم.

حال همین طراحی را وارد quartes یک اینستنس از تاپ مازول می سازیم و pin assignment کرده و مقادیر را بررسی کنیم.

آزمایش دوم:

در این آزمایش قسمت کدگشایی را ساده سازی می کنیم.

یکی از مهمترین بخش های این قسمت رجیستر فایل است که شامل رجیستر های پردازنه ماست. از بخش به صورت آسنکرون بک رجیستر را خوانده و به صورت سنکرون روی رجیستر می نویسد.

```

Ln#
1  module RegisterFile #(
2    parameter WordLen = 32,
3    parameter WordCount = 15
4  ) (
5    input clk, rst,
6    input [$clog2(WordCount)-1:0] readRegister1, readRegister2, writeRegister,
7    input [WordLen-1:0] writeData,
8    input regWrite, sclr,
9    output [WordLen-1:0] readData1, readData2
10   );
11   reg [WordLen-1:0] regFile [0:WordCount-1];
12
13   assign readData1 = regFile[readRegister1];
14   assign readData2 = regFile[readRegister2];
15
16   integer i;
17
18   initial begin
19     for (i = 0; i < WordCount; i = i + 1)
20       regFile[i] <= i;
21   end
22
23   always @ (posedge clk or posedge rst) begin
24     if (rst)
25       for (i = 0; i < WordCount; i = i + 1)
26         regFile[i] <= i;
27     else if (sclr)
28       regFile[writeRegister] <= {WordLen{1'b0}};
29     else if (regWrite)
30       regFile[writeRegister] <= writeData;
31   end
32 endmodule
33

```

بخش های اصلی دیگر این بخش، مازول های **Control unit** و **condition check** هستند.

مازول **Condition Check** بررسی می کند که آیا دستورالعمل شرطی باید اجرا شود یا خیر. این کار با مقایسه کد شرطی دستورالعمل (**Condition Code**) با فلگ های وضعیت (Z, N, C, V) از رجیستر **CPSR** انجام می شود. اگر شرط برقرار باشد، دستورالعمل به مرحله اجرا ارسال می شود؛ در غیر این صورت، پردازنده آن را نادیده می گیرد.

```

Ln#          ConditionCheck.sv - Default
1  module ConditionCheck(
2      input [3:0] cond,
3      input [3:0] status,
4      output reg result
5  );
6      wire n, z, c, v;
7      assign {n, z, c, v} = status;
8
9      always @(cond, n, z, c, v) begin
10         result = 1'b0;
11         case (cond)
12             4'b0000: result = z;           // EQ
13             4'b0001: result = ~z;        // NE
14             4'b0010: result = c;        // CS/HS
15             4'b0011: result = ~c;       // CC/LO
16             4'b0100: result = n;        // MI
17             4'b0101: result = ~n;       // PL
18             4'b0110: result = v;        // VS
19             4'b0111: result = ~v;       // VC
20             4'b1000: result = c & ~z;   // HI
21             4'b1001: result = ~c | z;   // LS
22             4'b1010: result = (n == v); // GE
23             4'b1011: result = (n != v); // LT
24             4'b1100: result = ~z & (n == v); // GT
25             4'b1101: result = z & (n != v); // LE
26             4'b1110: result = 1'b1;      // AL
27             default: result = 1'b0;
28         endcase
29     end
30 endmodule
31

```

ماژول Control Unit وظیفه رمزگشایی دستورالعمل و تولید سیگنال‌های کنترلی را دارد. این سیگنال‌ها نحوه عملکرد اجزای مختلف پردازنده (مانند ALU، رجیسترها، حافظه و ...) را تعیین می‌کنند. همچنین، کنترل جریان داده و مدیریت پرسش‌ها را بر عهده دارد تا دستورالعمل به درستی اجرا شود.

```

Ln#          ControlUnit.sv - Default
1  module ControlUnit(
2      input [1:0] mode,
3      input [3:0] opcode,
4      input sIn,
5      output reg [3:0] aluCmd,
6      output reg memRead, memWrite,
7      output reg wbEn, branch, sOut
8  );
9
10     always @ (mode, opcode, sIn) begin
11         aluCmd = 4'd0;
12         {memRead, memWrite} = 2'd0;
13         {wbEn, branch, sOut} = 3'd0;
14
15         case (opcode)
16             4'b1101: aluCmd = 4'b0001; // MOV
17             4'b1111: aluCmd = 4'b1001; // MVN
18             4'b0100: aluCmd = 4'b0010; // ADD, LDR, STR
19             4'b0101: aluCmd = 4'b0011; // ADC
20             4'b0010: aluCmd = 4'b0100; // SUB
21             4'b0110: aluCmd = 4'b0101; // SBC
22             4'b0000: aluCmd = 4'b0110; // AND
23             4'b1100: aluCmd = 4'b0111; // ORR
24             4'b0001: aluCmd = 4'b1000; // EOR
25             4'b1010: aluCmd = 4'b0100; // CMP
26             4'b1000: aluCmd = 4'b0110; // TST
27             default: aluCmd = 4'b0001;
28         endcase
29
30         case (mode)
31             2'b00: begin
32                 sOut = sIn;
33                 // no write-back for CMP and TST
34                 wbEn = (opcode == 4'b1010 || opcode == 4'b1000) ? 1'b0 : 1'b1;
35             end
36             2'b01: begin
37             end
38         endcase
39     end
40 endmodule
41

```

در نهایت علاوه بر این مازول ها چند ماکس ساده داریم و رجیستر پایپ لاین هم دقیقاً مانند مرحله قبل پیاده سازی می کنیم با این تفاوت که تعداد متغیر های که در آن رجیستر می شوند خیلی بیشتر است.

وظیفه مازول IDecode ایجاد خروجی های مانند:
 { خروجی های واحد کنترل: شامل سیگنال هایی مثل aluCmd، memRead، memWrite، branch wbEn،
 که در صورت برقرار نبودن شرطها صفر می شوند.
 مقادیر رجیسترها (Rd) و Rm یا Rn: مقدار خوانده شده از دو رجیستر مربوطه یا مقصد (برای دستور STR).

ثابت ها (24imm و shift operand): مقادیر عددی فوری و اپراتور شیفت.
 شماره رجیسترها (Rn و twoSrc): شماره رجیسترها که به Hazard Unit ارسال می شوند.
 بر اساس دستور است که در مرحله اجرا بتوانیم از آنها استفاده کنیم.

```

1 module idcode(
2     input clk, rst,
3     // From RegsIdId
4     input [31:0] pcIn, inst,
5     // From EX
6     input [3:0] status,
7     // From WB
8     input wbWbEn,
9     input [31:0] wbValue,
10    input [3:0] wbDest,
11    // From Hazard
12    input hazard,
13    // To RegsIdEx
14    output [31:0] pcOut,
15    output [3:0] aluCmd,
16    output memRead, memWrite, wbEn, branch, s,
17    output [31:0] val_Rn, val_Rm,
18    output imm,
19    output [11:0] shiftOperand,
20    output signed [23:0] imm24,
21    output [3:0] dest,
22    output [3:0] src1, src2,
23    // To Hazard
24    output hazardTwoSrc
25 );
26     assign pcOut = pcIn;
27     assign imm = inst[25];
28     assign shiftOperand = inst[11:0];
29     assign imm24 = inst[23:0];
30     assign dest = inst[15:12];
31     assign src1 = inst[19:16];
32
33     wire [3:0] aluCmdCU;
34     wire memReadCU, memWriteCU, wbEnCU, branchCU, scU;
35     wire [3:0] regfile2Inp;
36     wire cond, condFinal;
37     wire [31:0] regRn, regRm;
38     assign hazardTwoSrc = ~imm | memWriteCU;
39     assign condFinal = ~cond | hazard;
40
41 endmodule

```

(تصویر مربوط به کد نهایی است)

در نهایت برای تست باید تست بنچی اجرا کنیم و ببینیم هر دستور به چه صورتی کد گشایی می شود برای مثال دستور اول ما #20 R MOV O : R

کد آن به صورت :

1110_00_1_1101_0_0000_0000_000000010100

است.

از مقدار (imm) استفاده می‌شود و مقصد (rd) رجیستر شماره 0 است. واحد کنترل برای این دستور، سیگنال‌های memRead و branch را غیرفعال می‌کند، چون نیازی به خواندن از حافظه یا پرس وجود ندارد، اما سیگنال wbEn فعال است تا خروجی ALU در رجیستر مقصد نوشته شود. دستور MOV برای ALU کد 0001 را تولید می‌کند که به ALU می‌گوید مقدار فوري را بدون تغيير برگرداند. از رجیستر فایل هم مشخص است که rn برابر با 0 و rm برابر با 4 است، چون رجیسترها از قبل مقداردهی شده‌اند. علاوه بر اين، چون از مقدار فوري استفاده شده و دستور STR نیست، سیگنال twoSrc از Hazard Unit صفر می‌شود، یعنی این دستور تنها به يك منبع داده نياز دارد. به زبان ساده، اين دستور فقط عدد 20 را مي‌گيرد و بدون عمليات خاص در رجیستر شماره 0 ذخیره می‌کند.

تمام اين اطلاعات روی wave قابل بررسی است تا از صحت کد مطمئن شويم.

در نهايت همين تست رو به صورت مشابه در کوارتز و فضای سیگنال تب انجام ميدهيم و خروجي هر دستور را روی بورد نيز بررسی می‌کنيم.

آزمایش سوم:

بعد از پياده سازي واکشي و کدگشائي باید به سراغ مرحله اجرا برويم.

ماژول ALU برای انجام عمليات های اصلی استفاده می‌شود.

این ماژول با دریافت دو ورودی ۳۲ بیتی، یک دستور ۴ بیتی و یک بیت carryIn، عمليات‌های محاسباتی مختلفی را انجام داده و دو خروجی تولید می‌کند: یک خروجی ۳۲ بیتی به عنوان نتيجه عمليات و یک خروجی ۴ بیتی برای بهروزرسانی وضعیت رجیسترها (مانند صفر بودن نتيجه، منفی بودن، وجود کري، يا سريزن). ALU می‌تواند عمليات‌هایی مثل انتقال مقدار بدون تغيير (MOV)، محاسبه NOT اپرنده دوم (MVN)، جمع دو اپرنده (ADD)، جمع با اپرنده (ADC)، تغريق ساده (SUB) يا همراه با carryIn (SBC)، و عمليات منطقی بیت‌به‌بیت مثل AND، OR، XOR را انجام دهد. خروجی نهايی بر اساس دستور ۴ بیتی مشخص شده و وضعیت‌های مرتبط با عمليات نيز همزمان در خروجی وضعیت ثبت می‌شوند.

```

02:03:23 ...
02:12:26 ...
01:23:39 ...
02:02:23 ...
01:41:54 ...
11:13:48 ...
01:04:56 ...
01:59:53 ...
12:20:17 ...
12:15:42 ...
02:02:46 ...
11:27:58 ...
02:00:25 ...
01:59:24 ...
01:57:35 ...
03:27:58 ...
02:11:26 ...
02:03:07 ...
01:41:25 ...
01:58:19 ...
02:01:13 ...
02:01:54 ...
02:03:43 ...
01:58:42 ...
11:21:21 ...
02:16:07 ...
02:36:17 ...
Ln#
1   module ALU #( parameter N = 32
2   (
3     input [N-1:0] a, b,
4     input carryIn,
5     input [3:0] exeCmd,
6     output reg [N-1:0] out,
7     output [3:0] status
8   );
9
10   reg c, v;
11   wire z, n;
12   assign status = {n, z, c, v};
13   assign z = ~out;
14   assign n = out[N-1];
15
16   wire [N-1:0] carryExt, nCarryExt;
17   assign carryExt = {{(N-1){1'b0}}, carryIn};
18   assign nCarryExt = {{(N-1){1'b0}}, ~carryIn};
19
20   always @ (exeCmd or a or b or carryExt or nCarryExt) begin
21     out = {N{1'b0}};
22     c = 1'b0;
23
24     case (exeCmd)
25       4'b0001: out = b;                                // MOV
26       4'b1001: out = ~b;                               // MVN
27       4'b0010: (c, out) = a + b;                      // ADD
28       4'b0011: (c, out) = a + b + carryExt;          // ADC
29       4'b0100: (c, out) = a - b;                      // SUB
30       4'b0101: (c, out) = a - b - nCarryExt;         // SBC
31       4'b0110: out = a & b;                           // AND
32       4'b0111: out = a | b;                           // ORR
33       4'b1000: out = a ^ b;                           // EOR
34     default: out = {N{1'b0}};
35
36   endcase
37
38   v = 1'b0;
39   if (exeCmd[3:1] == 3'b001) begin                  // ADD, ADC
40     v = (a[N-1] == b[N-1]) && (a[N-1] != out[N-1]);
41
42 end

```

ماژول **Val2Generator** وظیفه تولید اپرند دوم ALU را بر عهده دارد و از چهار ورودی استفاده می‌کند: **MemInst**, که از ترکیب سیگنال‌های **EN_W_MEM** و **EN_R_MEM** مشخص می‌کند آیا دستور فعلی به حافظه مربوط است یا نه؛ **Imm**, بیت شماره 25 دستور که تعیین می‌کند اپرند دوم از نوع فوری (**Immediate**) است یا از رجیستر استفاده می‌کند؛ **ShifterOperand**, شامل 12 بیت سمت راست دستور که جزئیات شیفت و اپرند فوری را ارائه می‌دهد؛ و **ValRm**, مقدار 32 بیتی که از خروجی رجیستر فایل تأمین می‌شود. این ماژول یک خروجی 32 بیتی دارد که به عنوان ورودی دوم ALU عمل می‌کند.

رفتار مازلول به حالت دستور **bstg** دارد: اگر دستور مربوط به حافظه باشد، خروجی، مقدار **Imm** است که به صورت **Extend-Sign Operand Shifter** به 32 بیت گسترش یافته است. اگر **Imm** برابر 1 باشد، 8 بیت سمت راست **Operand Shifter** به اندازه دو برابر 4 بیت سمت چپ آن به صورت دایره‌ای از سمت راست شیفت می‌شود. در حالتی که اپرند دوم به رجیستر اشاره می‌کند (**Imm** برابر 0)، 4 بیت سمت راست **Operand Shifter** شماره رجیستر را مشخص می‌کند و مقدار **ValRm** با توجه به نوع شیفت (مشخص شده توسط بیت‌های 5 و 6 **ShifterOperand**) و اندازه شیفت (مشخص شده توسط 5 بیت سمت چپ **ShifterOperand**) تغییر داده می‌شود. یک حالت دیگر نیز وجود دارد که نیازی به پیاده‌سازی آن نیست.

```

Ln# 1 module Val2Generator(
2   input memInst, imm,
3   input [31:0] valRm,
4   input [11:0] shifterOperand,
5   output reg [31:0] val2
6 );
7   integer i;
8
9   always @{memInst or imm or valRm or shifterOperand} begin
10    val2 = 32'd0;
11    if (memInst) begin // LDR, STR
12      val2 = {{20{shifterOperand[11]}}, shifterOperand};
13    end
14    else begin
15      if (imm) begin // immediate
16        val2 = {24'd0, shifterOperand[7:0]};
17        for (i = 0; i < 2 * shifterOperand[11:8]; i = i + 1) begin
18          val2 = {val2[0], val2[31:1]};
19        end
20      end
21      else begin // shift Rm
22        case (shifterOperand[6:5])
23          2'b00: val2 = valRm << shifterOperand[11:7]; // LSL
24          2'b01: val2 = valRm >> shifterOperand[11:7]; // LSR
25          2'b10: val2 = $signed(valRm) >>> shifterOperand[11:7]; // ASR
26          2'b11: begin
27            val2 = valRm;
28            for (i = 0; i < shifterOperand[11:7]; i = i + 1) begin
29              val2 = {val2[0], val2[31:1]};
30            end
31          end
32          default: val2 = 32'd0;
33        endcase
34      end
35    end
36  end
37 endmodule
38

```

این دو مازول، مازول های اصلی مرحله اجرا هستند و در کنار یک جمع کننده این مازول را تکمیل می کنند. نتایج این مازول نیز داخل رجیستری در پایپ لاین رجیستر می شود.

مازول **WB** وظیفه انتخاب داده ای را دارد که باید به رجیستر فایل بازگردانده شود. این کار با استفاده از یک **Multiplexer** انجام می شود که سیگنال **EN_R_MEM** به عنوان سلکتور آن عمل می کند. اگر سیگنال **EN_R_MEM** فعال باشد (یعنی مقداری از حافظه خوانده شده باشد)، داده خوانده شده از مموری به رجیستر فایل ارسال می شود. در غیر این صورت، خروجی **ALU** به عنوان داده ای که باید در رجیستر فایل نوشته شود، انتخاب می شود. به طور خلاصه، این مازول بین داده های حاصل از مموری و **ALU** یکی را برای بازگرداندن به رجیستر فایل انتخاب می کند.

```

Ln# 1 module wbback(
2   input clk, rst,
3   input wbEnIn, memREn,
4   input [31:0] aluRes, mData,
5   input [3:0] destIn,
6   output wbEnOut,
7   output [31:0] wbValue,
8   output [3:0] destOut
9 );
10 assign wbEnOut = wbEnIn;
11 assign destOut = destIn;
12
13 Mux2To1 #(32) wbMux(
14   .a0(aluRes),
15   .a1(mData),
16   .sel(memREn),
17   .out(wbValue)
18 );
19 endmodule
20

```

در نهایت می توانیم برای بررسی عملکرد فعلی پردازنده مقدار رجیستر ها را با اجرای چند دستور و مشاهده تغییر مقدار رجیستر ها بررسی کنیم.

[0]	20	20	
[1]	8192	4096	8192
[2]	-1073741824	-1073741824	
[3]	4	4	
[4]	40	40	
[5]	-12	-12	
[6]	9	9	
[7]	805306373	805306373	

با بررسی مقادیر مشاهده می کنیم که برخی از مقادیر درست نیستند. دلیل این اشتباهات وجود نداشتن Hazardunit است که بتواند از این اشتباهات جلوگیری کند، در ادامه این مشکلات برطرف خواهند شد.

آزمایش چهارم:

در این مرحله باید پردازنده Arm اولیه را تکمیل کنیم. برای اینکار نیاز به بخش های HazardDetection و DataMemory داریم.

ماژول MEM وظیفه مدیریت عملیات خواندن و نوشتمن داده ها در حافظه را بر عهده دارد. این ماژول شامل یک حافظه داده (Memory Data) است که ورودی های آن شامل یک آدرس 32 بیتی، یک داده 32 بیتی، و بیت های memWrite و memRead است. اگر memWrite فعال باشد، داده ورودی همگام با کلاک در حافظه نوشته می شود و اگر memRead فعال باشد، داده به صورت غیرهمگام از حافظه خوانده می شود.

در ابتدا، آدرس ورودی 1024 واحد کاهش می یابد، زیرا خانه های 0 تا 1023 حافظه در پردازنده واقعی ARM به حافظه دستورالعمل اختصاص دارند، اما در این آزمایش این بخش به ماژول IF منتقل شده است. سپس آدرس به اندازه 2 بیت به سمت راست شیفت می شود تا خواندن از حافظه aligned باشد، زیرا حافظه از خانه های 32 بیتی تشکیل شده و این شیفت تضمین می کند که داده ها در بلوک های کامل 32 بیتی خوانده یا نوشته می شوند. در صورت نیاز به داده های جزئی تر (بایت های 0, 1, 2, یا 3)، حافظه مقادیر مناسب را در خروجی قرار می دهد.

```

1  module DataMemory(
2      input clk, rst,
3      input [31:0] memAddr, writeData,
4      input memRead, memWrite,
5      output reg [31:0] readData
6  );
7
8
9      reg [31:0] dataMem [0:63]; // 256B memory
10
11     wire [31:0] dataAddr, adr;
12     assign dataAddr = memAddr - 32'd1024;
13     assign adr = {2'b00, dataAddr[31:2]}; // Align
14
15     integer i;
16
17     always @(posedge clk, posedge rst) begin
18         if (rst)
19             for (i = 0; i < 64; i = i + 1) begin
20                 dataMem[i] <= 32'd0;
21             end
22         else if (memWrite)
23             dataMem[adr] <= writeData;
24     end
25
26     always @(memRead or adr) begin
27         if (memRead)
28             readData = dataMem[adr];
29     end
30 endmodule

```

ماژول **Hazard Detection Unit** وظیفه شناسایی **hazard data** را دارد و در صورتی که شرایط زیر رخ دهد، سیگنال **hazard** را فعال می‌کند: اگر سیگنال **wbEn** در مرحله **Exe** فعال باشد و مقدار **Rn** با **destEx** برابر باشد، یا اگر **wbEn** در مرحله **Exe** فعال و **twoSrc** نیز روشن باشد و مقدار **Rm** (ورودی دوم رجیستر فایل یا در دستور **STR**، مقدار **Rd**) با **destEx** برابر باشد. همچنین اگر **wbEn** در مرحله **Mem** فعال باشد و مقدار **Rn** با **destMem** برابر باشد، نشان‌دهنده این است که داده مورد نیاز هنوز تولید یا بهروز نشده است. این سیگنال فعال، پردازنده را از وقوع تعارض داده آگاه کرده و اقدامات لازم برای جلوگیری از مشکلات داده را ممکن می‌سازد.

```

1 module hazard(
2     input [3:0] rn, rdm,
3     input twoSrc,
4     input [3:0] destEx, destMem,
5     input wbEnEx, wbEnMem, memRen,
6     input forwardEn,
7     output reg hazard
8 );
9
10    always @(rn, rdm, destEx, destMem, wbEnEx, wbEnMem, memRen, twoSrc, forwardEn) begin
11        hazard = 1'b0;
12        if (forwardEn) begin
13            if (memRen) begin
14                if (rn == destEx || (twoSrc && rdm == destEx)) begin
15                    hazard = 1'bl;
16                end
17            end
18        end
19        else begin
20            if (wbEnEx) begin
21                if (rn == destEx || (twoSrc && rdm == destEx)) begin
22                    hazard = 1'bl;
23                end
24            end
25            if (wbEnMem) begin,
26                if (rn == destMem || (twoSrc && rdm == destMem)) begin
27                    hazard = 1'bl;
28                end
29            end
30        end
31    end
32
33 endmodule

```

تصویر مربوط به کد نهایی است.

حال پردازنده ARM ما تکمیل شده و می توانیم آن را تست کنیم. برای اینکار ابتدا از صحت نتایج در modelsim مطمئن شده سپس کد ها را در کوارتز سنتز می کنیم.

/dataMem[0]	-2147483643	8192	
/dataMem[1]	-1073741824	-1073741824	
/dataMem[2]	41	-2147483643	
/dataMem[3]	8192	41	
/dataMem[4]	-123	-123	
/dataMem[5]	10	10	
/dataMem[6]	-123	-123	

برای مثال با اجرای دستور STR مقدار خانه مموری را به -123 تغییر دادیم. به طریق مشابه می توانیم تک دستورات و نتایجشان را بررسی کنیم.

در نهایت بعد از دیدن مقادیر درست در رجیستر های modelsim کد را روی برد می بریم و نتایج را در سیگنال تب نیز بررسی می کنیم.

60	192	224	256	288	320	352	384
24							20
							-2147483648
2		3	4				-1073741824
0	1	2	3	0	1	2	3
1024	1028	1032	1024	1028	1032		8192
			41		41		-123
2		41	8192		41	8192	10
							-123
							-2147483648
							-11
							-1073741824
							8192

در نهایت مشاهده می کنیم که دستورات رو بورد نیز به درستی کار می کنند و رجیستر ها به درستی سنت می شوند.

آزمایش پنجم:

در آزمایش قبل یک پردازنده ARM بیاده سازی کردیم، در این پردازنده برای رفع مشکل وابستگی داده ای (هazard داده ای) از روش اضافه کردن توقف استفاده کردیم. به طوری که تا زمانی که وابستگی بین داده ها وجود دارد، باید دستور جدید را متوقف کرد.

در این آزمایش برای رفع این مشکل باید تکنیک ارسال به جلو (Forwarding) را به پردازنده اضافه کنیم.

واحد تشخیص هازاد همچنان در پردازنده وجود دارد، با افزودن این تکنیک پردازنده می تواند در دو حالت "اضافه کردن توقف" و "ارسال به جلو" کار کند. بنابراین یک سیگنال ورودی به پردازنده اضافه می شود که مشخص می کند پردازنده در چه حالتی کار میکند.

پس از افزودن این تکنیک باید مراحل تست همانند آزمایش قبل صورت گیرد تا از درستی عملکرد آن اطمینان حاصل شود که در ادامه نتایج را بررسی میکنیم و بهبود کارایی را محاسبه میکنیم.

ماژول Forwarding Unit با استفاده از ورودی forwardEn مشخص می کند که آیا باید فعال شود یا خیر.

در غیر این صورت، واحد hazard unit وظیفه دارد که تمامی تداخل‌ها را مدیریت کند. به عنوان نمونه، اگر دو دستور به یک رجیستر مشترک نیاز داشته باشند و دستور اول باعث تغییر مقدار رجیستر شود، این حالت توسط hazard unit شناسایی شده و وضعیت پایپ‌لاین در دو کلک استال قرار می‌گیرد.

در این شرایط، به جای توقف پایپ‌لاین، این مازول داده را مستقیماً از مراحل جلوتر پایپ‌لاین به مراحل قبلی انتقال می‌دهد. این مازول، دستوراتی که از مرحله EX عبور کرده‌اند و به رجیستری نیاز دارند که هنوز مقدار آن در حال تغییر است، شناسایی می‌کند و با استفاده از سیگنال‌های wbEn و dest مقدار مناسب را تأمین می‌کند.

برای اطمینان از این فرآیند، این مازول مقادیر موجود در رجیسترهاي 1src و 2src را بررسی می‌کند تا تشخیص دهد که آیا این مقادیر از دستورات قبلی که هنوز در مراحل MEM یا WB قرار دارند، نیاز است یا خیر. اگر چنین شرایطی برقرار باشد، مقادیر جدید از پایپ‌لاین دریافت شده و به رجیسترها ارسال می‌شود.

مازول با استفاده از سیگنال‌های 1selSrc و 2selSrc، داده‌ها را در مرحله EX به درستی از میان گزینه‌های موجود انتخاب می‌کند و در نهایت در صورت لزوم و با فعال شدن سیگنال‌های مربوطه، مقدار مناسب به مراحل بعدی منتقل می‌شود.

```

Ln# 1 module forward(
2   input forwardEn,
3   input [3:0] src1, src2,
4   input wbEnMem, wbEnWb,
5   input [3:0] destMem, destWb,
6   output reg [1:0] selSrc1, selSrc2
7 );
8   always @ (forwardEn, src1, wbEnMem, wbEnWb, destMem, destWb) begin
9     selSrc1 = 2'b00;
10    if (forwardEn) begin
11      if (wbEnMem && (destMem == src1)) begin
12        selSrc1 = 2'b01;
13      end
14      else if (wbEnWb && (destWb == src1)) begin
15        selSrc1 = 2'b10;
16      end
17    end
18  end
19
20  always @ (forwardEn, src2, wbEnMem, wbEnWb, destMem, destWb) begin
21    selSrc2 = 2'b00;
22    if (forwardEn) begin
23      if (wbEnMem && (destMem == src2)) begin
24        selSrc2 = 2'b01;
25      end
26      else if (wbEnWb && (destWb == src2)) begin
27        selSrc2 = 2'b10;
28      end
29    end
30  end
31 endmodule

```

در صورتی که forwarding غیرفعال باشد، این مازول تمامی تداخل‌ها را تشخیص داده و با استال کردن پایپ‌لاین، آن‌ها را مدیریت می‌کند. اما در صورت فعلی بودن forwarding، تنها مواردی را باید کنترل کند که امکان انتقال به جلو برای آن‌ها وجود نداشته باشد.

تنها دستوری که در این پردازندۀ چنین شرایطی دارد، دستور LDR است. این دستور مقداری را از حافظه خوانده و به یک رجیستر بازمی‌گرداند. از آنجاکه مقدار خوانده شده از حافظه فوراً آماده نمی‌شود (زیرا این مقدار از یک خروجی ترکیبی به دست می‌آید)، امکان انتقال آن به مراحل قبلی با forwarding وجود ندارد.

در نتیجه، کد hazard unit به‌گونه‌ای تغییر یافته که این شرایط را مدیریت کند.

```

1 module hazard(
2     input [3:0] rn, rdm,
3     input twoSrc,
4     input [3:0] destEx, destMem,
5     input wbEnEx, wbEnMem, memRn,
6     input forwardEn,
7     output reg hazard
8 );
9
10    always @ (rn, rdm, destEx, destMem, wbEnEx, wbEnMem, memRn, twoSrc, forwardEn)
11        hazard = 1'b0;
12        if (forwardEn) begin
13            if (memRn) begin
14                if (rn == destEx || (twoSrc && rdm == destEx)) begin
15                    hazard = 1'b1;
16                end
17            end
18        end
19        else begin
20            if (wbEnEx) begin
21                if (rn == destEx || (twoSrc && rdm == destEx)) begin
22                    hazard = 1'b1;
23                end
24            end
25            if (wbEnMem) begin
26                if (rn == destMem || (twoSrc && rdm == destMem)) begin
27                    hazard = 1'b1;
28                end
29            end
30        end
31    end
32 endmodule

```

در قسمت Top level نیز باید مازول Forwarding unit و سیگنال به آن اضافه کنیم

عکس نتایج سنتز:

log: 2024/12/23 15:00:38 #0		click to insert time bar	
Type	Alias	Node	Value
Top		0	179
SW[0]			
SW[1]			
Top:top:forwardEn			
↳	...erFile:friegFile[0]		00000400h
↳	...erFile:friegFile[1]		C0000000h
↳	...erFile:friegFile[2]		80000000h
↳	...erFile:friegFile[3]		00000290h
↳	...erFile:friegFile[4]	80000000h	X
		00000408h	00000200h

log: 2024/12/23 14:57:48 #0		click to insert time bar	
Type	Alias	Node	Value
Top		0	278
SW[0]			
SW[1]			
Top:top:forwardEn			
↳	...erFile:friegFile[0]		00000400h
↳	...erFile:friegFile[1]		80000000h
↳	...erFile:friegFile[2]		C0000000h
↳	...erFile:friegFile[3]		00000290h
↳	...erFile:friegFile[4]	00000408h	X
			00000200h

خروجی در دو حالت فوردارد خاموش و روشن:

همانطور که در تصاویر مشخص هست، در حالت فروارد جای ۲۸۰ به عدد ۱۸۰ رسیدیم و این نشان دهنده ی حدود ۳ درصد سرعت بیشتر است.

آزمایش ششم:

در آزمایش های قبل یک پردازنده ARM پیاده سازی کردیم که برای حافظه داده از حافظه داخلی استفاده میشد. در حافظه داخلی زمان خواندن و نوشتمن یک کلاک است. اما به علت محدودیت در استفاده از حافظه داخلی FPGA باید از حافظه خارجی بر روی برد استفاده شود. این نوع حافظه ها دارای تاخیر دسترسی بیش از یک کلاک هستند. که باعث ایجاد چالش در معماری پردازنده می گردد.

در این آزمایش از حافظه خارجی SRAM که روی برد 2DE وجود دارد به عنوان حافظه داده پردازنده ARM استفاده می کنیم. برای این کار یک مژول برای کنترل کردن آن طراحی می شود که دسترسی به حافظه داده با کلاک پردازنده همگام شود. همچنین خطوط داده برای خواندن حافظه و نوشتمن در آن را از هم جدا می کند.

حافظه SRAM موجود در برد 2DE دارای زمان دستیابی در یک کلاک برای هر کلمه ۱۶ بیتی می باشد، اما در پردازنده واقعی زمان دستیابی به حافظه خارجی بیش از یک کلاک است، بنابراین در اینجا زمان دستیابی به حافظه را چندین کلاک (مثلًا ۴ کلاک) در نظر می گیریم. هنگامی که زمان دستیابی به حافظه بیش از یک کلاک شود، آنگاه باید در طراحی خط لوله پردازنده تغییراتی صورت گیرد. به طوری که هنگام دستیابی به حافظه، خط لوله باید منتظر بماند تا عملیات مربوط به حافظه به اتمام رسد و سپس خط لوله به کارش ادامه می دهد.

در این آزمایش از SRAM به عنوان حافظه داده پردازنده استفاده می شود. حافظه SRAM که مورد استفاده قرار گرفته، از نوع ناهمگام است و دارای ظرفیت 512 کیلوبایت می باشد که کلمات آن ۱۶ بیتی هستند. این حافظه شامل خطوط داده ۱۶ بیتی و خطوط آدرس ۱۸ بیتی است. لازم به ذکر است که خطوط داده برای عملیات خواندن و نوشتمن به صورت مشترک استفاده می شوند، به این معنا که تنها یک گذرگاه ۱۶ بیتی برای داده وجود دارد. تعیین عملیات خواندن یا نوشتمن، با استفاده از سیگنال write انجام می شود.

برای استفاده از SRAM به عنوان حافظه داده پردازنده، به طراحی یک مژول کنترل SRAM نیاز است. این مژول وظیفه دارد تا عملیات خواندن و نوشتمن را در حافظه به صورت ناهمگام انجام دهد. همچنین، امکان جدا کردن خطوط داده برای عملیات خواندن و نوشتمن نیز وجود دارد.

```

module Sram(
    input clk, rst,
    input SRAM_WE_N,
    input [18:0] SRAM_ADDR,
    inout [15:0] SRAM_DQ
);
    reg [15:0] memory [0:511];
    assign SRAM_DQ = (SRAM_WE_N == 1'b1) ? memory[SRAM_ADDR] : 16'dz;

    always @(posedge clk or posedge rst) begin
        if (SRAM_WE_N == 1'b0) begin
            memory[SRAM_ADDR] = SRAM_DQ;
        end
    end
endmodule

```

با توجه به اینکه این مazzoل حافظه‌ای خارج از پردازنده دارد و خانه‌های آن به صورت 2 بایتی هستند، نمی‌توانیم در یک کلاک مانند حافظه قبلی عملیات حافظه را انجام دهیم و حداقل به 3 کلاک نیاز داریم. از طرفی، برای اینکه بتوانیم کارایی SRAM را در کنار حافظه نهان (بدون حافظه نهان) مقایسه کنیم، تعداد کلاک لازم برای انجام عملیات را 6 عدد در نظر می‌گیریم.

حال با توجه به اینکه نیاز داریم در این 6 کلاک پردازنده به طور کامل متوقف شود تا عملیات حافظه به پایان برسد، توقف یا عدم توقف پردازنده را با یک سیگنال ready مشخص می‌کنیم. زمانی که یک دستور حافظه (مانند load و store) وارد بخش MEM می‌شود، این سیگنال بررسی می‌گردد. در صورتی که این سیگنال freeze فعال باشد، پردازنده متوقف شده و تمامی رجیستر بلک‌های آن نیز ثابت می‌مانند.

اما در صورت وجود سیگنال ready در زمان فعال بودن، پردازنده اجازه ادامه کار و استفاده از رجیسترها را خواهد داشت.

```

always @(*) begin
    SRAM_ADDR = 18'b0;
    SRAM_WE_N = 1'bl;
    ready = 1'b0;
    case (ps)
        Idle: ready = ~ (wrEn | rdEn);
        New:;
        DataLow: begin
            SRAM_ADDR = sramLowAddr;
            SRAM_WE_N = ~wrEn;
            dq = writeData[15:0];
            if (rdEn)
                readData[15:0] <= SRAM_DQ;
        end
        DataHigh: begin
            SRAM_ADDR = sramHighAddr;
            SRAM_WE_N = ~wrEn;
            dq = writeData[31:16];
            if (rdEn)
                readData[31:16] <= SRAM_DQ;
        end
        Finish: begin
            SRAM_WE_N = 1'bl;
        end
        NoOp:;
        Done: ready = 1'b1;
    endcase
end
always @(posedge clk or posedge rst) begin
    if (rst) ps <= Idle;
    else ps <= ns;
end
endmodule

```

ابتدا مازول مربوط به حافظه قبل را حذف میکنیم و سپس یک کنترلر جدید به طراحی اضافه میکنیم. در این کنترلر، سیگنالی به نام `freeze` تعریف شده است که در صورت غیرفعال بودن سیگنال `ready`، پردازنده را متوقف کرده و تمام رجیستر بلاکها را ثابت نگه می‌دارد. در این کنترلر همچنین از سیگنال `hazard` که پیشتر طراحی شده بود، استفاده می‌شود. این سیگنال با سایر سیگنال‌های ترکیبی به صورت OR عمل می‌کند و نتیجهنهای آن در این بخش اعمال می‌شود.

علاوه بر این، یک `Multiplexer` نیز برای سیگنال `wbEn` در نظر گرفته شده است که داده‌های مرحله `MEM` را به مرحله `WB` انتقال می‌دهد. این طراحی تضمین می‌کند که هنگام نوشتن در رجیستر فایل، از انجام این عمل در زمانی که عملیات حافظه `SRAM` در حال اجراست، جلوگیری شود. سیگنال کنترلی این `Multiplexer` از همان سیگنال `freeze` استفاده می‌کند.

در نهایت، خروجی نهایی برای پردازنده از طریق ابزار `RTL Viewer` مطابق طراحی ارائه شده است.

یک مازول شبیه ساز SRAM مینویسیم که به توان با استفاده از آن تعداد کلک های دقیق را در نرم افزار ModelSim بدست بیاوریم.

با توجه به سیگنال تب حدود ۴۰۰ کلک تا اتمام برنامه گرفته شده است.

در حالتی که از مموری پردازنده استفاده می شد ۱۸۰ کلک نیاز بود که یعنی کارایی پردازنده کاهش یافته و ۲۳۰ کلک افزایش یافته است:

۱۸۰/۴۰۰ ≈ ۵۰٪ کاهش پرفورمنس

از طرف دیگر در مرحله سنتز نسبت به سنتر فورواردینگ حدود ۲۵ درصد المنت بیشتری استفاده شده (از ۷۵۰ به ۹۵۰) که این نشان دهنده هزینه سخت افزاری بیشتر است.