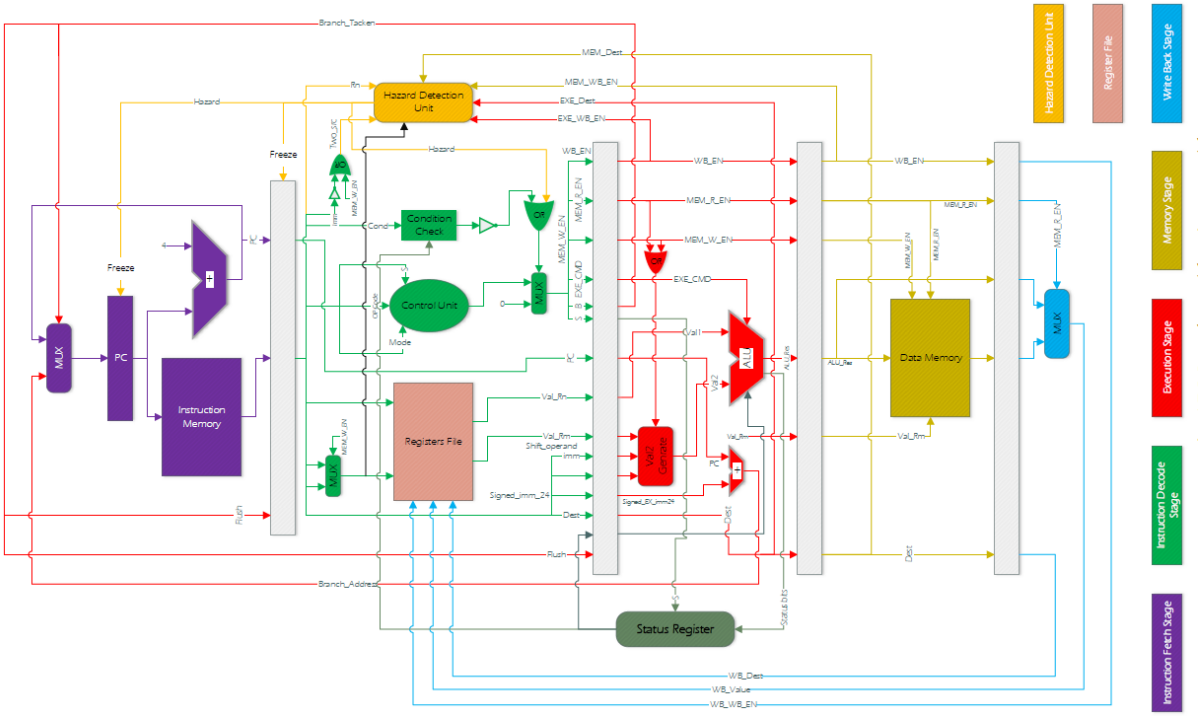
توضیحات آزمایش

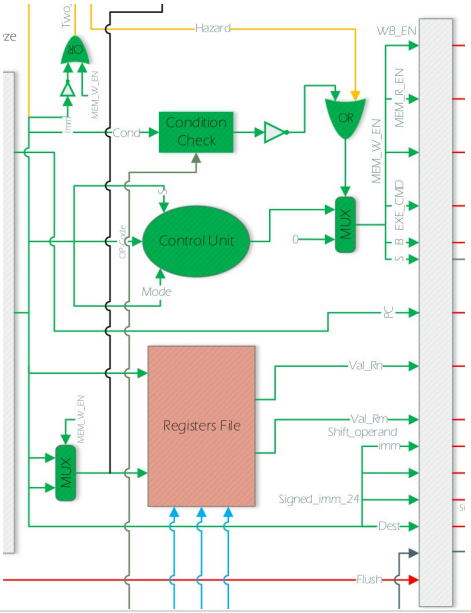
در این آزمایش پردازنده ARM به صورت پایپ‌لاین پیاده‌سازی می‌شود. دیاگرام این پردازنده به صورت زیر است:



این پردازنده دارای 13 دستور اصلی است. پیاده‌سازی باید در زبان وریلاگ باشد و در نهایت پس از شبیه‌سازی در نرم‌افزار ModelSim، با استفاده از نرم‌افزار Quartus سنتز می‌شود و روی FPGA قرار می‌گیرد. سپس، با استفاده از یک تست‌بنچ، پردازنده پیاده‌سازی شده تست می‌شود. از اهداف این آزمایش می‌توان به یادگیری نحوه عیب‌یابی و تست مدارهای سخت‌افزاری طراحی شده اشاره کرد.

جلسه دوم

در این جلسه ماژول کدگشایی دستور (ID) پیاده‌سازی می‌شود:



این ماژول از 3 بخش اصلی Register File، واحد کنترل و Condition Check تشکیل شده است.

1. رجیستر فایل

رجیستر فایل باید از 16 رجیستر تشکیل شده باشد که رجیستر آخر آن به عنوان PC استفاده می‌شود. ولی برای سادگی طراحی، این رجیستر به خارج از رجیستر فایل و به مرحله IF انتقال یافته و رجیستر فایل این مرحله فقط 15 رجیستر دارد که با شماره رجیستر مقداردهی اولیه شده‌اند.

```
module RegisterFile #(
    parameter WordLen = 32,
    parameter WordCount = 15
)(
    input clk, rst,
    input [$clog2(WordCount)-1:0] readRegister1, readRegister2,
                                  writeRegister,
    input [WordLen-1:0] writeData,
    input regWrite, sclr,
    output [WordLen-1:0] readData1, readData2
);
    reg [WordLen-1:0] regFile [0:WordCount-1];

    assign readData1 = regFile[readRegister1];
    assign readData2 = regFile[readRegister2];

    integer i;
    initial begin
        for (i = 0; i < WordCount; i = i + 1)
            regFile[i] <= i;
    end

    always @(negedge clk or posedge rst) begin
        if (rst)
            for (i = 0; i < WordCount; i = i + 1)
                regFile[i] <= i;
        else if (sclr)
            regFile[writeRegister] <= {WordLen{1'b0}};
        else if (regWrite)
            regFile[writeRegister] <= writeData;
    end
endmodule
```

این رجیستر فایل به طور async با تغییر آدرس read، مقدار رجیستر مد نظر را خروجی می‌دهد و به طور sync با کلاک می‌توان با فعال کردن `regWrite` مقدار رجیستر شماره `writeRegister` را تغییر داد.

2. واحد کنترل

واحد کنترل سه ورودی mode، opcode و s را می‌گیرد و با بررسی آنها سیگنال‌های مورد نیاز برای دستور کنونی را خروجی می‌دهد. این سیگنال‌ها شامل aluCmd، memRead و memWrite، wbEn، branch و s اند.

ورودی s:

s بیت 20 اینستراکشن است. این بیت مشخص می‌کند که آیا دستور مقادیر status register را تغییر می‌دهد یا خیر.

ورودی mode:

mode بیت 27:26 اینستراکشن است. در صورتی که 00 باشد، یعنی عملیات دستور محاسبات یا منطقیست، در صورتی که 01 باشد، دستور کار با حافظه است و اگر 10 باشد، دستور branch است.

در واحد کنترل، در صورتی که 00 باشد `s` خروجی همان بیت s اینستراکشن می‌شود.

تشخیص دو دستور LDR و STR از آنجا که mode و opcode یکسانی دارند، با بیت s است و خروجی‌های `memRead`، `memWrite` و `wbEn` تولید می‌شوند.

در صورتی که mode اینستراکشن 10 باشد خروجی `branch` فعال می‌شود.

ورودی opcode:

opcode بیت 24:21 اینستراکشن است. پس از مشخص شدن نوع کلی دستور با mode، زیردستورها با استفاده از opcode تشخیص داده می‌شوند.

برای دستورات محاسباتی (mode = 00) آپکد نشان می‌دهد که به طور مثال ADD است یا SUB.

با استفاده از opcode خروجی `aluCmd` که به ALU وصل می‌شود تولید می‌شود.

در mode = 00، در صورتی که دستور CMP یا TST باشد مقداری به رجیستر فایل برگردانده نمی‌شود پس `wbEn` صفر خواهد بود.

کد ماژول:

```
module ControlUnit(
    input [1:0] mode,
    input [3:0] opcode,
    input sIn,
    output reg [3:0] aluCmd,
    output reg memRead, memWrite,
    output reg wbEn, branch, sOut
);
    always @(mode, opcode, sIn) begin
        aluCmd = 4'd0;
        {memRead, memWrite} = 2'd0;
        {wbEn, branch, sOut} = 3'd0;

        case (opcode)
            4'b1101: aluCmd = 4'b0001; // MOV
            4'b1111: aluCmd = 4'b1001; // MVN
            4'b0100: aluCmd = 4'b0010; // ADD
            4'b0101: aluCmd = 4'b0011; // ADC
            4'b0010: aluCmd = 4'b0100; // SUB
            4'b0011: aluCmd = 4'b0101; // SBC
            4'b0000: aluCmd = 4'b0110; // AND
            4'b1100: aluCmd = 4'b0111; // ORR
            4'b0001: aluCmd = 4'b1000; // EOR
            4'b1010: aluCmd = 4'b0100; // CMP
            4'b1000: aluCmd = 4'b0110; // TST
            4'b0100: aluCmd = 4'b0010; // LDR
            4'b0100: aluCmd = 4'b0010; // STR
            default: aluCmd = 4'b0001;
        endcase

        case (mode)
            2'b00: begin
                sOut = sIn;
                // no write-back for CMP and TST
                wbEn = (opcode == 4'b1010 || opcode == 4'b1000) ?
                        1'b0 : 1'b1;
            end
            2'b01: begin
                wbEn = sIn;
                memRead = sIn;
                memWrite = ~sIn;
            end
            2'b10: branch = 1'b1;
            default:;
        endcase
    end
endmodule
```

3. بررسی شرط

بیت 31:28 اینستراکشن condition است. دستورهای ARM می‌توانند به طور شرطی اجرا شوند که شرط‌ها از روی حالات status register به وجود می‌آیند. این رجیستر 4 بیت N Z C V دارد که مخفف negative zero carry overflow اند.

در صورتی که شرط (که با توجه به آخرین مقادیر status register ارزیابی می‌شود) برقرار نبود، دستور اجرا نمی‌شود و یک NOP در پایپ‌لاین به جای دستور جلو می‌رود. (no operation در این پردازنده در اصل دستور AND رجیستر 0 با خودش است)

مقادیر مختلف cond را در کد ماژول می‌توان مشاهده کرد. مقدار 1110 نشان‌دهنده نبود شرط است و دستوری که مقدار cond آن برابر 1110 باشد همیشه اجرا می‌شود.

```
module ConditionCheck(
    input [3:0] cond,
    input [3:0] status,
    output reg result
);
    wire n, z, c, v;
    assign {n, z, c, v} = status;

    always @(cond, status) begin
        result = 1'b0;
        case (cond)
            4'b0000: result = z;             // EQ
            4'b0001: result = ~z;            // NE
            4'b0010: result = c;             // CS/HS
            4'b0011: result = ~c;            // CC/LO
            4'b0100: result = n;             // MI
            4'b0101: result = ~n;            // PL
            4'b0110: result = v;             // VS
            4'b0111: result = ~v;            // VC
            4'b1000: result = c & ~z;        // HI
            4'b1001: result = ~c | z;        // LS
            4'b1010: result = (n == v);      // GE
            4'b1011: result = (n != v);      // LT
            4'b1100: result = ~z & (n == v); // GT
            4'b1101: result = z & (n != v);  // LE
            4'b1110: result = 1'b1;          // AL
            default: result = 1'b0;
        endcase
    end
endmodule
```

4. حافظه دستورات

از آنجا که Quartus نمی‌تواند مستقیم فایلی را توسط `$readmemb` بخواند و باید از ماژول‌های ROM خودش استفاده شود، برای پیاده‌سازی `InstructionMemory`، از case statement استفاده شده که بنا بر آدرس ورودی، دستور متناظر در آن خانه حافظه را به صورت async خروجی می‌دهد.

18 دستور اول برنامه محک وارد حافظه شده‌اند:

```
module InstructionMemory #(
  parameter Count = 1024
)(
  input [31:0] pc,
  output reg [31:0] inst
);
  wire [31:0] adr;
  // Align address to the word boundary
  assign adr = {pc[31:2], 2'b00};

  always @(adr) begin
    case (adr)
      32'd0:  inst = 32'b1110_00_1_1101_0_0000_0000_000000010100;
      32'd4:  inst = 32'b1110_00_1_1101_0_0000_0001_101000000001;
      32'd8:  inst = 32'b1110_00_1_1101_0_0000_0010_000100000011;
      32'd12: inst = 32'b1110_00_0_0100_1_0010_0011_000000000010;
      32'd16: inst = 32'b1110_00_0_0101_0_0000_0100_000000000000;
      32'd20: inst = 32'b1110_00_0_0010_0_0100_0101_000100000100;
      32'd24: inst = 32'b1110_00_0_0110_0_0000_0110_000010100000;
      32'd28: inst = 32'b1110_00_0_1100_0_0101_0111_000101000010;
      32'd32: inst = 32'b1110_00_0_0000_0_0111_1000_000000000011;
      32'd36: inst = 32'b1110_00_0_1111_0_0000_1001_000000000110;
      32'd40: inst = 32'b1110_00_0_0001_0_0100_1010_000000000101;
      32'd44: inst = 32'b1110_00_0_1010_1_1000_0000_000000000110;
      32'd48: inst = 32'b0001_00_0_0100_0_0001_0001_000000000001;
      32'd52: inst = 32'b1110_00_0_1000_1_1001_0000_000000001000;
      32'd56: inst = 32'b0000_00_0_0100_0_0010_0010_000000000010;
      32'd60: inst = 32'b1110_00_1_1101_0_0000_0000_101100000001;
      32'd64: inst = 32'b1110_01_0_0100_0_0000_0001_000000000000;
      32'd68: inst = 32'b1110_01_0_0100_1_0000_1011_000000000000;
    endcase
  end
endmodule
```

5. ماژول ID

ورودی‌ها:

خروجی دیوار IF-ID که شامل اینستراکشن و PC+4 است.

مقادیر status register که از مرحله EX می‌آید.

مقدار write-back، رجیستر مقصد آن و enable بودنش که از مرحله WB می‌آید. (ورودی‌های `wbValue`، `wbDest` و `wbWbEn`)

ورودی `hazard` از Hazard Unit.

خروجی‌ها:

در ماژول کدگشایی، اینستراکشن شکسته می‌شود و پس از عبور از واحد کنترل و بررسی شرط، سیگنال‌ها و داده‌های مورد نیاز خروجی داده می‌شوند.

خروجی‌های واحد کنترل (aluCmd، memRead و memWrite، wbEn، branch) که می‌توانند در صورت برقرار نبود شرط همه 0 شوند.

مقدار PC+4 که به طور مستقیم خارج می‌شود.
خروجی‌های رجیستر فایل (مقدار دو رجیستر Rn و Rm (یا Rd در دستور STR))
بیت imm، شماره رجیستر Rd، shift operand و imm24.
علاوه بر اینها، دو خروجی شماره رجیستر Rn و `twoSrc` به Hazard Unit می‌روند.

6. دیوار رجیستر ID-EX

این دیوار رجیستر بین دو مرحله علاوه بر خروجی‌های ID (به جز آنهایی که به Hazard Unit می‌روند) دو ورودی دیگر هم می‌گیرد. یکی از آنها `flush` است که همان خروجی `branch` از این دیوار است. ورودی دیگر بیت carry رجیستر status است. این برای دستور ADC و SBC که به carry دستور قبل نیاز دارند نیاز است و از آن سمت دیوار به ALU می‌رود.

7. ماژول تاپ‌لول

در این ماژول، 9 ماژول پایپ‌لاین به یکدیگر متصل شده‌اند.
در ماژول اصلی ARM که Quartus سنتز می‌کند، یک اینستنس از این ماژول تاپ‌لول گرفته شده است که کلاک آن به `CLOCK_50`، و rst آن به یکی از سوییچ‌های FPGA، `SW[0]` وصل شده است.

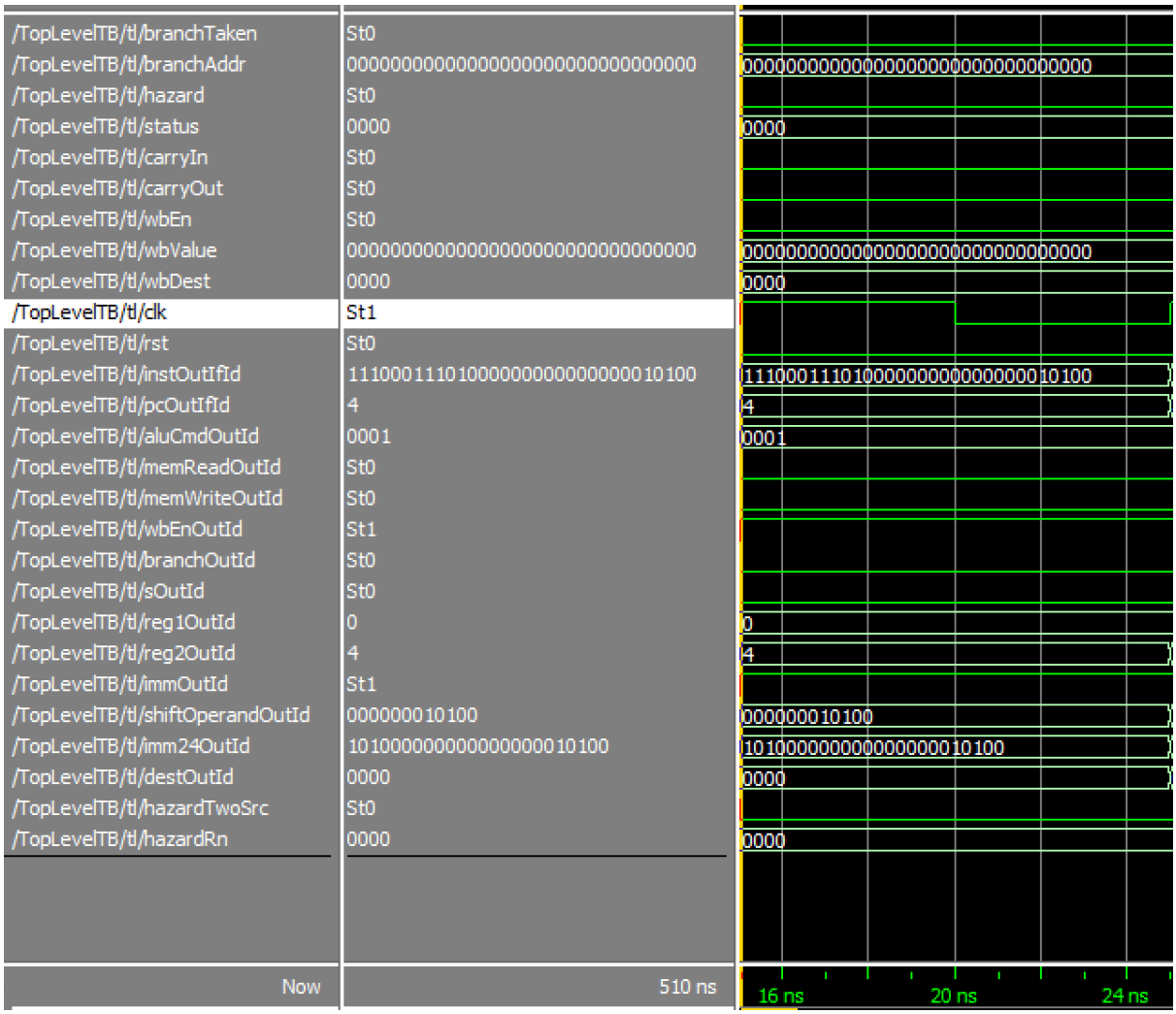
8. نتایج

تست‌بنچ:
Test-bench زیر نوشته شده است:

```
`timescale 1ns/1ns

module TopLevelTB();
    localparam HCLK = 5;

    reg clk, rst;
    TopLevel tl(clk, rst);

    always #HCLK clk = ~clk;

    initial begin
        {clk, rst} = 2'b01;
        #10 rst = 1'b0;
        #500 $stop;
    end
endmodule
```

برای دیدن نتایج 18 دستوری که در instruction memory قرار دارن نیاز به کامل شدن همه مراحل پایپ‌لاین داریم. چون مثلا دستورات محاسباتی باید در ALU حساب شده و سپس به رجیستر فایل برگردند.
به همین دلیل سیگنال‌های خروجی مرحله ID برای برخی از دستورات به طور مثال آورده شده تا صحت کارکرد این مرحله مشخص بشود.

دستور اول:



متغیرهای بالای clk هنوز پیاده‌سازی نشده‌اند و مقدار 0 دارند. زیر clk و rst، دو ورودی ID که اینستراکشن و PC+4 اند را می‌بینیم که از دیوار IF-ID آمده‌اند. زیر آنها خروجی‌های مرحله ID است.

دستور اول `MOV R0, #20` است که به صورت زیر شکسته می‌شود:

```
1110_00___1___1101___0_0000_0000_000000010100
cond_mode_imm_opcode_s_rn___rd___shiftoperand
```

از روی این خروجی‌های immOutId، shiftOperandOutId، imm24OutId و destOutId مشخص می‌شوند که در این دستور imm یک است و dest که همان rd است 0 است.

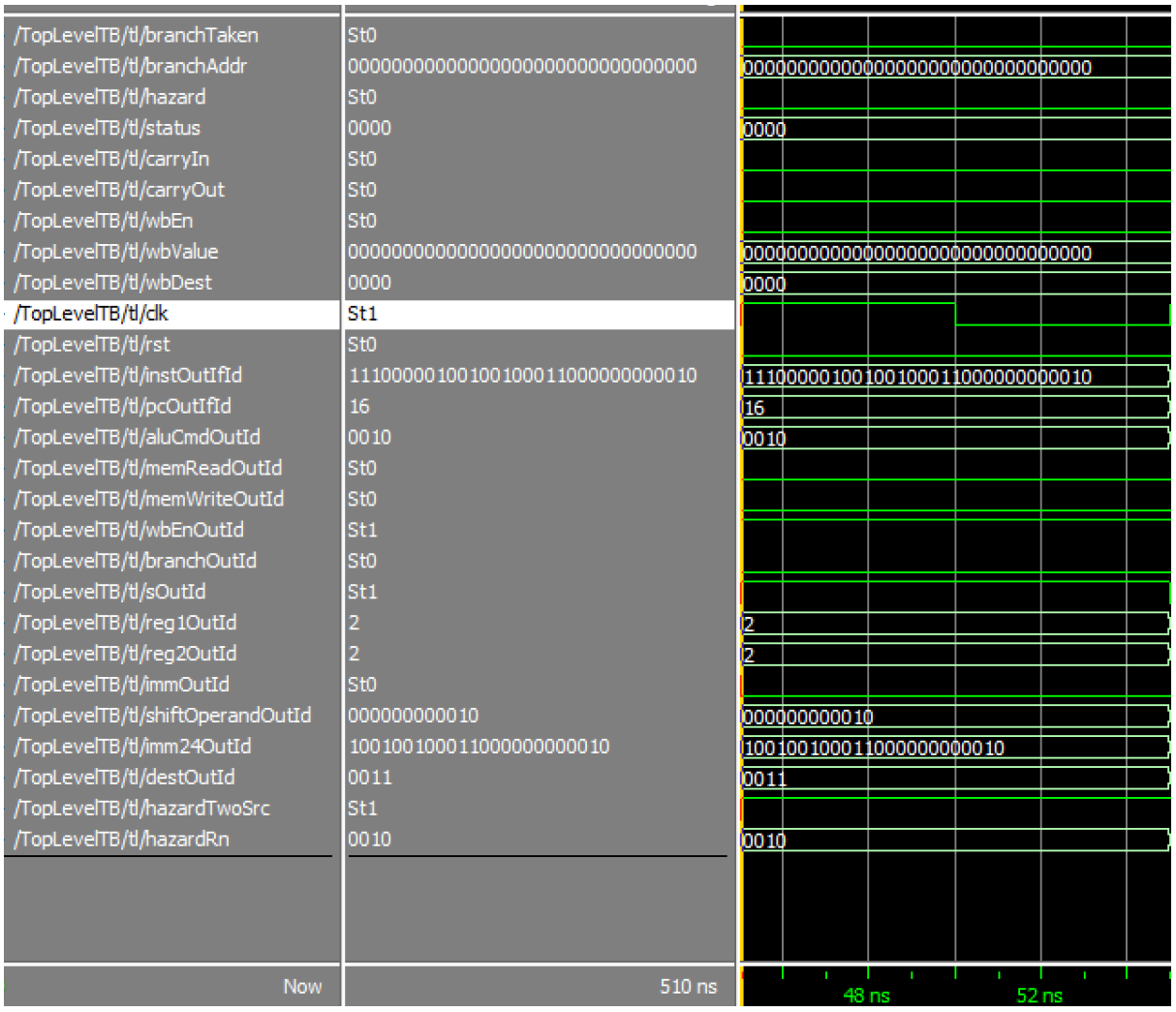
خروجی واحد کنترل برای دستور MOV، memRead/Write و branch غیرفعال اند و s همان s دستور است.

wbEn برای نوشتن خروجی MOV به رجیستر فایل فعال است و aluCmd = 0001 است که در ALU برای MOV است.

خروجی رجیستر فایل از آنجا که rn = 0 و rm = 4 است و رجیسترها با شماره شان مقداردهی اولیه شده‌اند، 0 و 4 است.

Rn به hazard رفته و از آنجا که imm صفر نیست و دستور STR هم نیست، مقدار TwoSrc صفر است.

دستور چهارم:



دستور چهارم `ADDS R3, R2, R2` است که به صورت زیر شکسته می‌شود:

```
1110_00___0___0100___1_0010_0011_000000000010
cond_mode_imm_opcode_s_rn___rd___shiftoperand
```

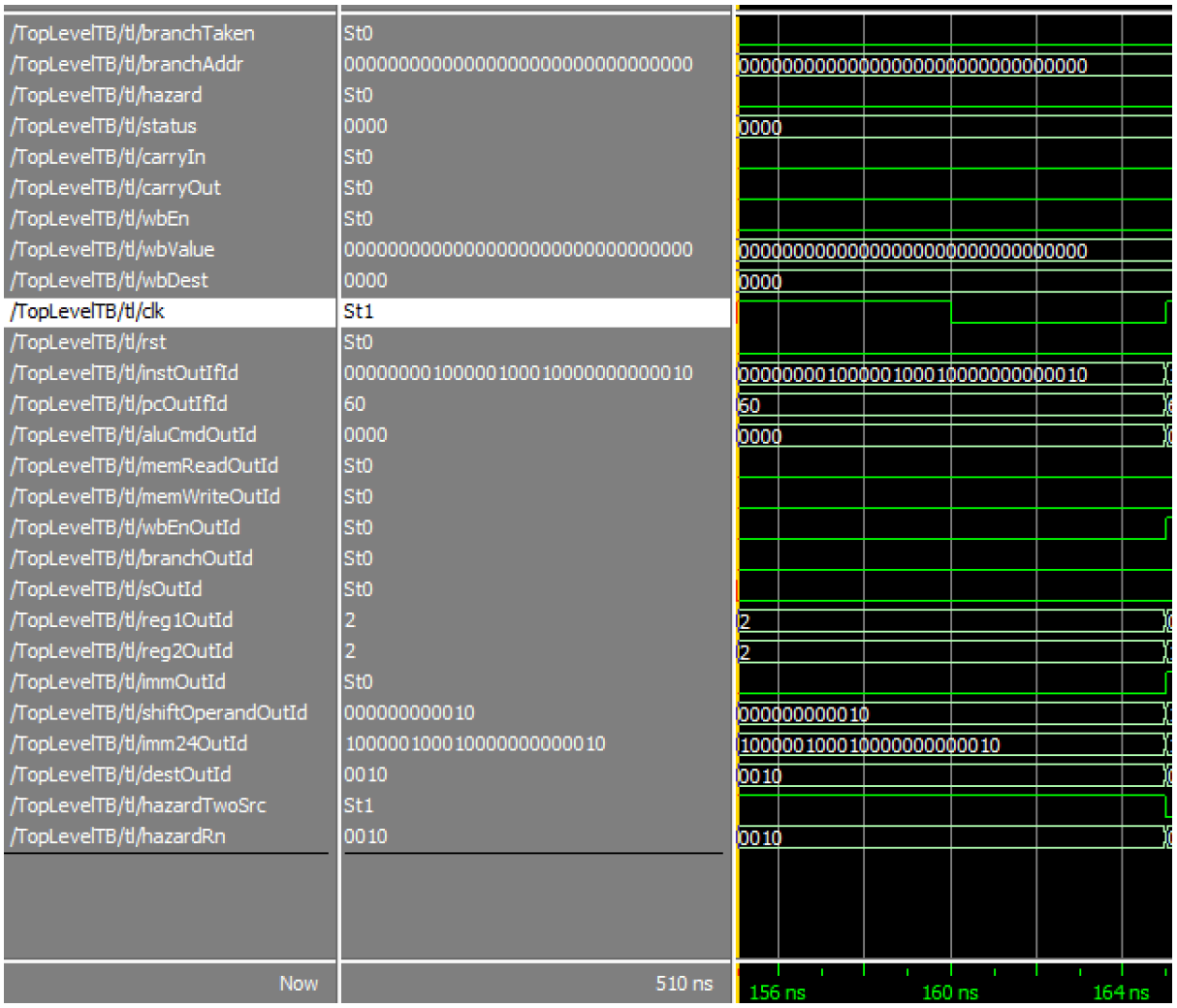
خروجی aluCmd چون که دستور جمع است 0010 بوده که در ALU جمع می‌بندد.

s خروجی برابر 1 است و status register باید آپدیت شود.

از آنجا که imm برابر 0 است TwoSrc روشن است.

rn و rm هر دو 2 اند پس جفت خروجی رجیستر فایل هم عدد 2 خواهد بود.

دستور پانزدهم:

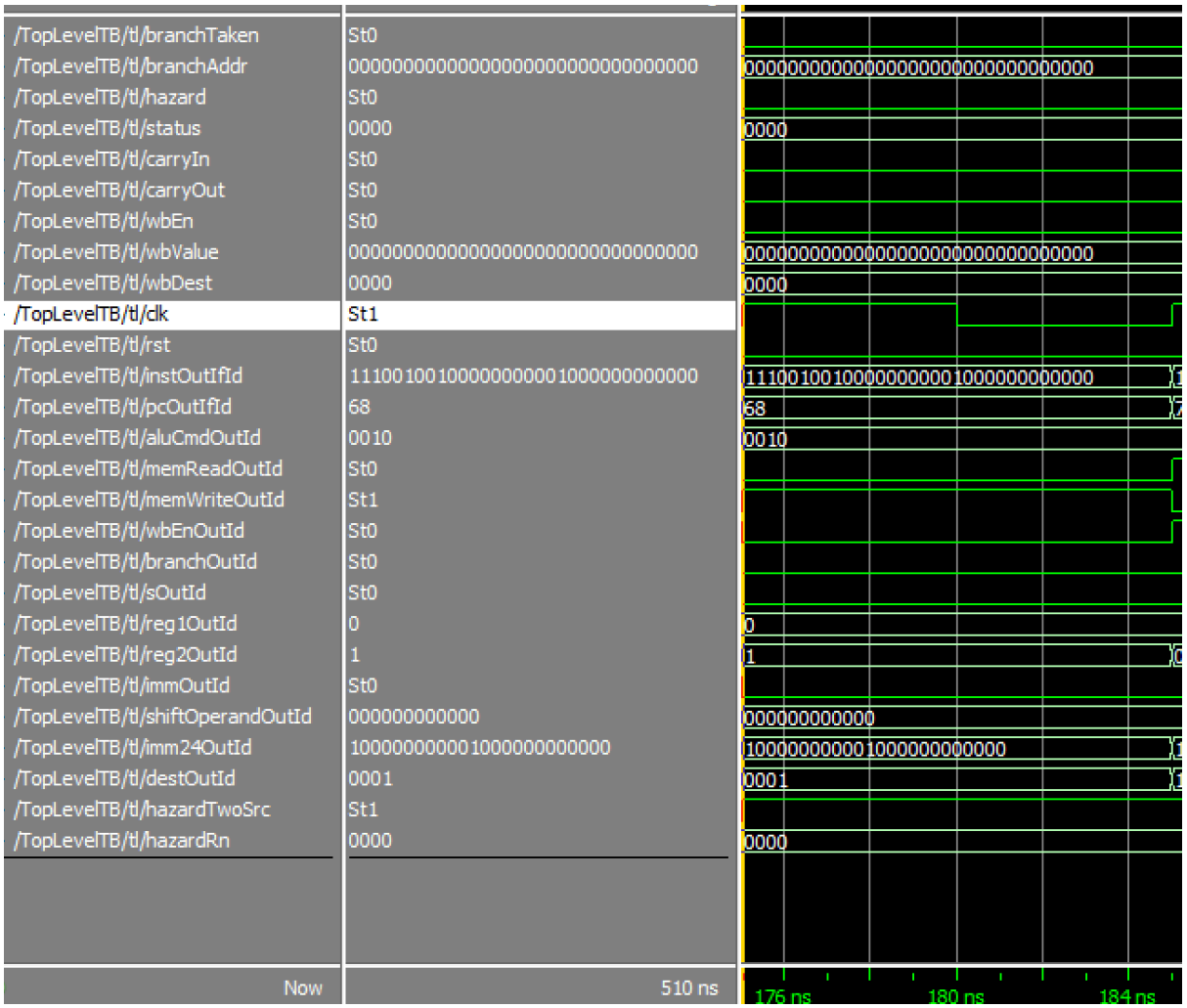


دستور 15-ام `ADDEQ R2, R2, R2` است که به صورت زیر شکسته می‌شود:

```
0000_00___0___0100___0_0010_0010_000000000010
cond_mode_imm_opcode_s_rn___rd___shiftoperand
```

این دستور به طور شرطی با شرط EQ باید اجرا بشود. ورودی cond = 0000 به ماژول بررسی شرط داده می‌شود و از آنجا که status register همواره مقدار 0 دارد و بیت Z (zero) هم 0 است، پس شرط (Z set) برقرار نیست و به جای خروجی واحد کنترل مقادیر 0 خروجی داده می‌شوند.

دستور هفدهم:



دستور 17-ام `STR R1, [R0], #0` است که به صورت زیر شکسته می‌شود:

```
1110_01___0___0100___0_0000_0001_000000000000
cond_mode_imm_opcode_s_rn___rd___offset12
```

طبق آپکد این دستور که همانند ADD است، aluCmd دستور متناظر با جمع کردن خواهد بود (چون rn با offset12 باید جمع شود).

خروجی اول رجیستر فایل همیشه rn است که اینجا 0 است. خروجی دوم آن در دستور STR مقدار rd یعنی 1 است چون rd به خانه rn+offset12 حافظه ریخته می‌شود.

نتیجه کامپایل Quartus:

Flow Summary	
Flow Status	Successful - Sat Apr 01 04:56:35 2023
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	0 / 33,216 (0 %)
Total combinational functions	0 / 33,216 (0 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

طبق گفته TA درس، نیازی به عکس از اجرا بر روی FPGA و نتیجه SignalTap نبوده و در کلاس دیده می‌شود.