

به نام خدا

گزارش آزمایشگاه معماری کامپیوتر

پاییز ۱۴۰۱

مریم ریاضی ۸۱۰۱۹۷۵۱۸

محمد تقی زاده ۸۱۰۱۹۸۳۷۳

ARM

در ابتدا به توضیح نحوه عملکرد هر یک از بخش های پردازنده ARM می پردازیم:

Instruction Fetch (IF):

هدف این بخش از پردازنده آپدیت کردن شمارنده PC و انتخاب دستور مناسب بر اساس مقدار آن است. در ابتدای کار و با ریست کردن، ابتدا مقدار PC صفر می شود و در المان جمع کننده در هر بار کلاک، ۴ تا به آن اضافه می شود. بر اساس سیگنال کنترلی المان MUX مقدار PC از خروجی جمع کننده و یا از آدرس دیگری می آید. آدرس دیگر منظور زمانی است که دستور branch اتفاق افتاده باشد و آدرس باید از مرحله EXE خوانده شود. بعد از مشخص شدن مقدار PC مقدار آن به حافظه دستورات داده می شود تا دستور متناظر با آن آدرس خوانده شود. لازم به ذکر است که مقدار PC در لبه بالارونده کلاک مقداردهی می شود. در این قسمت، سیگنال متوقف کننده freeeze نیز مشاهده می شود که در بخش hazard unit بیشتر به آن می پردازیم.

حافظه دستورات به صورت یک فایل تکست به پردازنده داده می شود که آدرس تمامی دستورات در آن قرار گرفته است.

Instruction Fetch Reg (IF Register):

رجیستر پایپ لاین مربوط به مرحله IF است که تمامی ورودی های آن ماژول به همراه دو سیگنال کنترلی flush , freeze را نیز دریافت می کند. سیگنال flush زمانی مقدار ۱ می گیرد که دستور branch داشته باشیم. یک شدن این سیگنال به معنی خالی شدن رجیستر است اما یک شدن سیگنال freeze به معنی متوقف شدن پردازنده و عدم ورود دستور جدید به مرحله بعدی است. در صورتی که هیچ کدام از این سیگنال ها فعال نشوند، مقادیر pc, instruction به مرحله بعدی می روند.

Instruction Decode (ID):

در این مرحله دستور وردی چند بخش می شود و از هر بخش آن اطلاعات لازم برای مراحل بعدی استخراج و سیگنال های کنترلی می شود.

(۱) در ابتدا ۲ آدرس وارد register file می شوند که یکی از آن ها آدرس نوشتن و دیگری آدرس خواندن است. می دانیم که عملیات خواندن به صورت combinational و عملیات نوشتن همراه با لبه پایین رونده کلاک رخ می دهد. برای اینکه رجیستر دیتایی را بنویسد باید سیگنال write enable فعال باشد و در این صورت دیتا را در write adress می نویسد.

(۲) ماژول مهم دیگر مرحله ID ماژول کنترل کننده پردازنده است که وظیفه دارد سیگنال های کنترلی پردازنده را تولید کند. سیگنال های تولیدی کنترل کننده عبارتند از :
cmd , mem read, mem write, write enable سیگنال cmd یک سیگنال ۴ بیتی است که نوع عملیات محاسباتی (جمع، تفریق و ..) را برای ماژول ALU آماده می کند. ورودی های کنترل کننده سیگنال های S, mode, opcode هستند که هر کدام در بخشی از دستور قرار گرفته اند. تمامی دستورات بجز load, store آپکودهای متمایز دارند. سیگنال mode برای مشخص کردن جنس عملیات (محاسباتی و یا کار با حافظه) است.

(۳) مرحله مهم دیگری که در ID اتفاق می افتد، خروجی های ماژول condition chek است. وظیفه این ماژول این است که بررسی کند آیا باید برای اجرای یک دستور شرطی برقرار شود یا خیر. این بررسی از طریق خروجی ماژول دیگری به نام status register صورت می گیرد که یک خروجی ۴ بیتی است. آنچه در ماژول condition check مقداردهی می شود یک سیگنال ۴ بیتی است که هر کدام از بیت ها به نام های N, Z, V, C اسم گذاری شده اند. N به معنای منفی شدن مقدار خروجی، Z به معنای صفر شدن، C به معنای داشتن carry، و V به معنای over flow است.

جدای از ماژول های توضیح داده شده در شماره های ۱ تا ۳، دو ماژول OR نیز در این مرحله وجود دارند. یکی از آن ها برای تولید سیگنال two source که ورودی hazard unit است استفاده می شود و دیگری نیز برای تولید سیگنال کنترلی multiplexer که خروجی آن، مقدار سیگنال های کنترلی را بر اساس وجود و یا عدم وجود مخاطره تعیین می کند.

Instruction Decode Register:

در این رجیستر از تمامی سیگنال های خروجی ID نمونه برداری می شود. لازم به ذکر است که محتوای این رجیستر در صورت یک بودن سیگنال flush، خالی می شود که به معنی برقرار شدن دستور branch است.

Execution (EXE):

در این محله از پردازنده، عملیات محاسباتی انجام می شوند. این مرحله نیز چند ماژول مهم دارد که هر کدام از آن ها را به ترتیب توضیح می دهیم:

(۱) مهم ترین ماژول این مرحله، ماژول val2 generator است. این ماژول وظیفه دارد که مقدار دوم ورودی ماژول ALU را تولید کند. تولید این مقدار حالت های مختلفی دارد: (۱) مقدار دوم، مقدار offset است که این حالت برای دستورهای store, load رخ می دهد. در این حالت ۲۰ بیت صفر به سمت چپ آن افزوده می شود تا به حالت استاندارد ۳۲ بیت برسد و عملیات انجام می شود.

(۲) حالت دوم، مربوط به حالت immediative value است. در این حالت برای تولید عملوند دوم نیاز به انجام چرخش به راست است. در ابتدا ۸ بیت پایین (بیت های ۰ تا ۷) Shifter Operand را با قرار دادن ۲۴ بیت صفر در سمت چپ آن به ۳۲ بیت گسترش می دهیم و سپس یک کپی از این عدد ۳۲ بیتی به دست آمده در کنار خودش قرار می دهیم تا یک باس ۶۴ بیتی تشکیل شود. در نهایت عملوند دوم، ۳۲ بیتی است که از $31 + (2 * \text{rotate immediate})$ شروع می شود تا ۳۲ بیت بعد از آن ادامه دارد. منظور از rotate immediate نیز، بیت های ۸ تا ۱۱ shifter operand است.

(۳) حالت سوم، شیف فوری است که خود به ۴ روش مختلف انجام می گیرد. شیفت منطقی به چپ یا راست با استفاده از مقدار Shift immediate صورت میگیرد. در صورتی که arithmetic داشته باشیم، باید داده خود را به اندازه ۳۲ بیت sign extend کنیم تا باس ۶۴ بیتی تولید شود. خروجی ما ۳۲ بیتی است که از بیت 31 (shift immediate) + شروع می شود و تا ۳۲ بیت دیگر ادامه دارد. در صورتی که نوع شیفت، rotate right باشد، داده ۳۲ بیتی را یک بار دیگر در کنار خودش قرار می دهیم تا باس ۶۴ بیتی تولید شود و ۳۲ بیت مورد نظر مانند روشی که رد arithmetic توضیح داده شد، انتخاب می شوند.

(۲) ماژول مهم دیگر در این مرحله، ماژول ALU است. ورودی های این ماژول، یکی خروجی val2 generator و دیگری، مقدار خوانده شده از رجیستر است. وظیفه این ماژول این است که با توجه به سیگنال ورودی cmd نوع دستور را تشخیص دهد و عملیات مناسب را روی آن ها انجام دهد. این ماژول، ورودی های ماژول status register را نیز تعیین میکند. بیت N، بیت آخر خروجی، بیت Z، مقدار or شده خروجی و بیت V, C نیز طی عملیات مقداردهی می شوند.

(۳) ماژول دیگر که تقریباً خارج از مرحله EXE قرار گرفته است، ماژول status register است. این ماژول وظیفه دارد که وضعیت دستور را نگهداری کند. در ابتدا مقادیر خروجی این ماژول با ریست، صفر می شوند و اگر سیگنال S که به آن می دهیم، یک شود، وضعیت را به روز کرده و نگهداری می کند تا به ماژول condition check بدهد. در نهایت نیز در این ماژول، از تمامی ماژول های ۱ تا ۳ instance گرفته می شود تا سیگنال های خروجی مناسب تولید شوند.

Execution Register:

مانند رجیسترهای پایپ لاین قبلی، خروجی های مرحله را در خود نگهداری میکند.

Memory (MEM):

حافظه دیتا نیز یک RAM است که آدرس دهی در آن برحسب بایت صورت می گیرد و هر بار ۴ بایت از حافظه خوانده یا نوشته می شود. نکته مهم در این مازول، align کردن آدرس ها است. در مرحله اول باید بدانیم که چون آدرس های ورودی مقداری بیشتر از ۱۰۲۳ دارند ابتدا باید عدد ۱۰۲۴ را از آدرس ها کم کنیم تا به آدرس های کوچکتر نیز دسترسی داشته باشیم. بعد از این کار، باید آدرس ها را بر ۴ تقسیم کنیم تا بتوانیم یکی یکی آدرس را زیاد کنیم.

Memory Register:

رجیستر پایپ لاین مرحله MEM است که تمام خروجی های آن را نگهداری میکند.

Write Back:

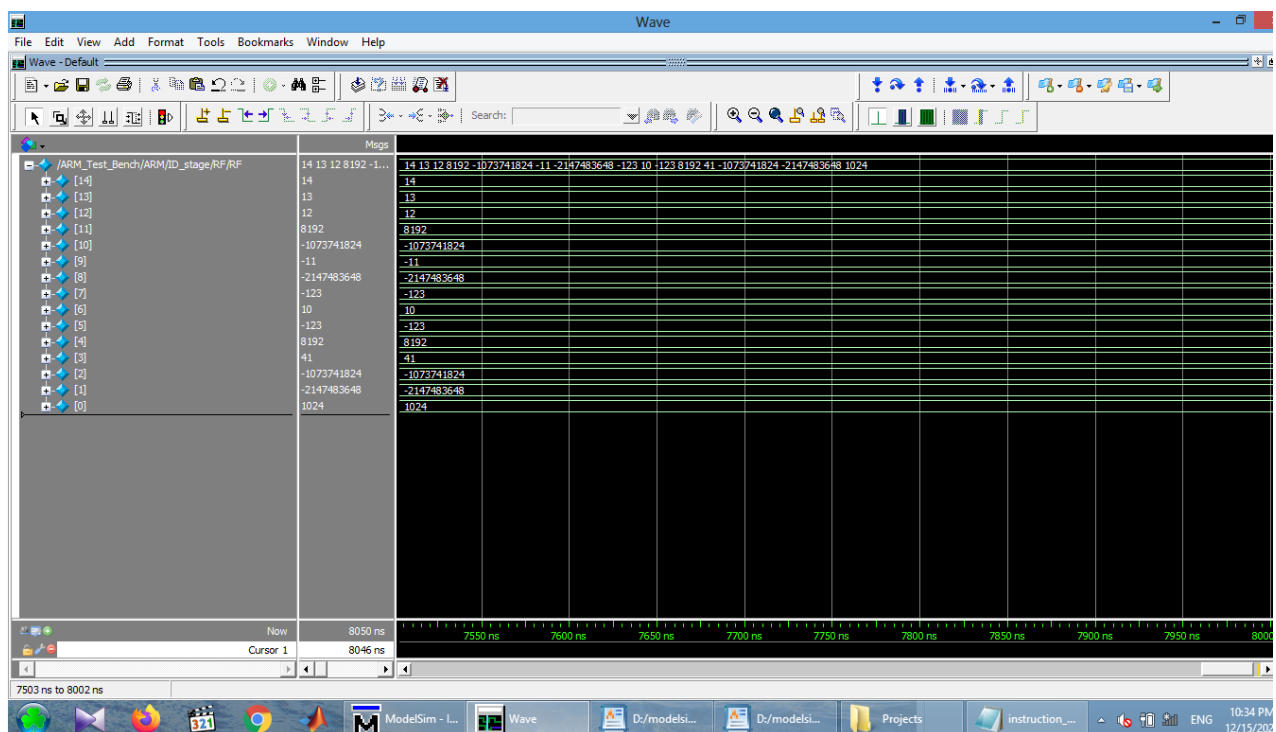
مرحله نهایی پردازنده است که وظیفه آن تولید دیتای مناسب برای نوشتن در رجیستر است. نکته مهم در این مرحله این است که این دیتا یا از خروجی ALU حاصل می شود و یا دیتایی است که طبق دستور load از حافظه دیتا خوانده شده است.

Hazard Unit:

وظیفه این مرحله، تشخیص مخاطره داده ای (خواندن بعد از نوشتن) و جلوگیری از آن است. این مخاطره زمانی اتفاق می افتد که در یک دستور، مقدار رجیستری بخواهد آپدیت شود و دستور بعدی بخواهد از مقدار آن رجیستر استفاده کند قبل از اینکه مقدار جدید در آن قرار گرفته باشد. در صورت رخ دادن این مخاطره، باید پردازنده متوقف شود تا مقدار رجیستر آپدیت شود و مقدار این توقف، حداکثر ۲ کلاک است.

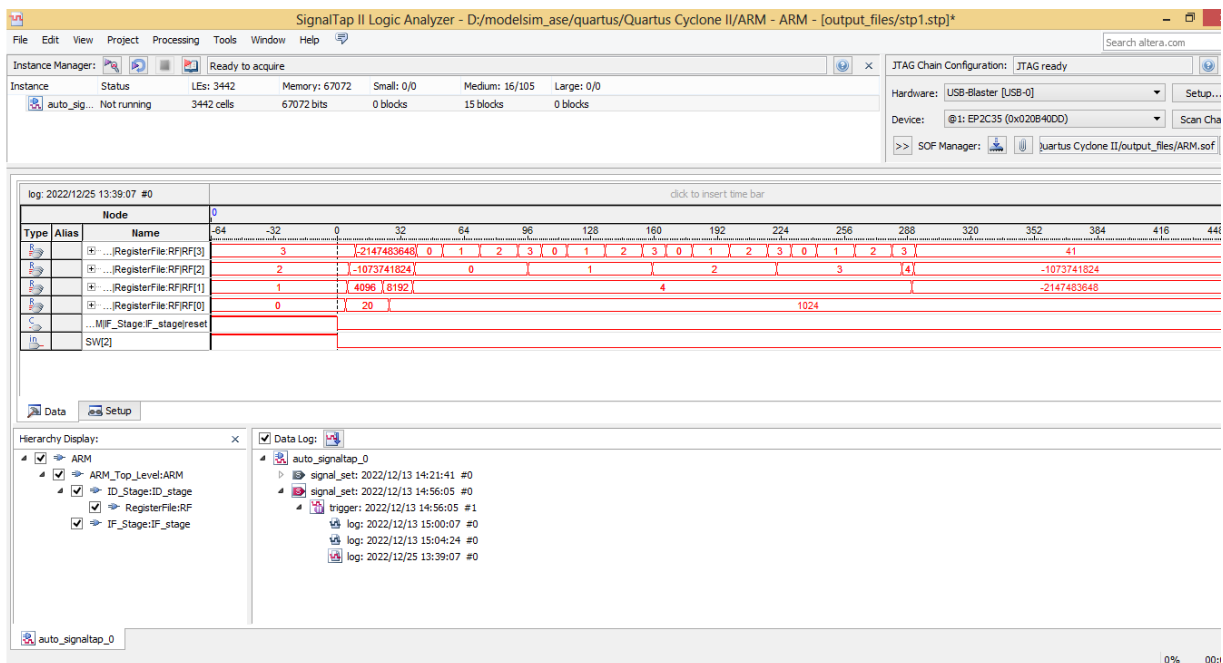
حالتی که را برای این مخاطره باید بررسی کنیم: یکسان بودن آدرس src2 یا src1 با آدرس مقصد مرحله EXE یا مرحله MEM. نکته مهم این است که اگر دستور، move یا movenot باشد، نیازی به بررسی src1 نیست زیرا در این دستورات استفاده نمی شود.

تصویر نتیجه شبیه سازی در Modelsim:



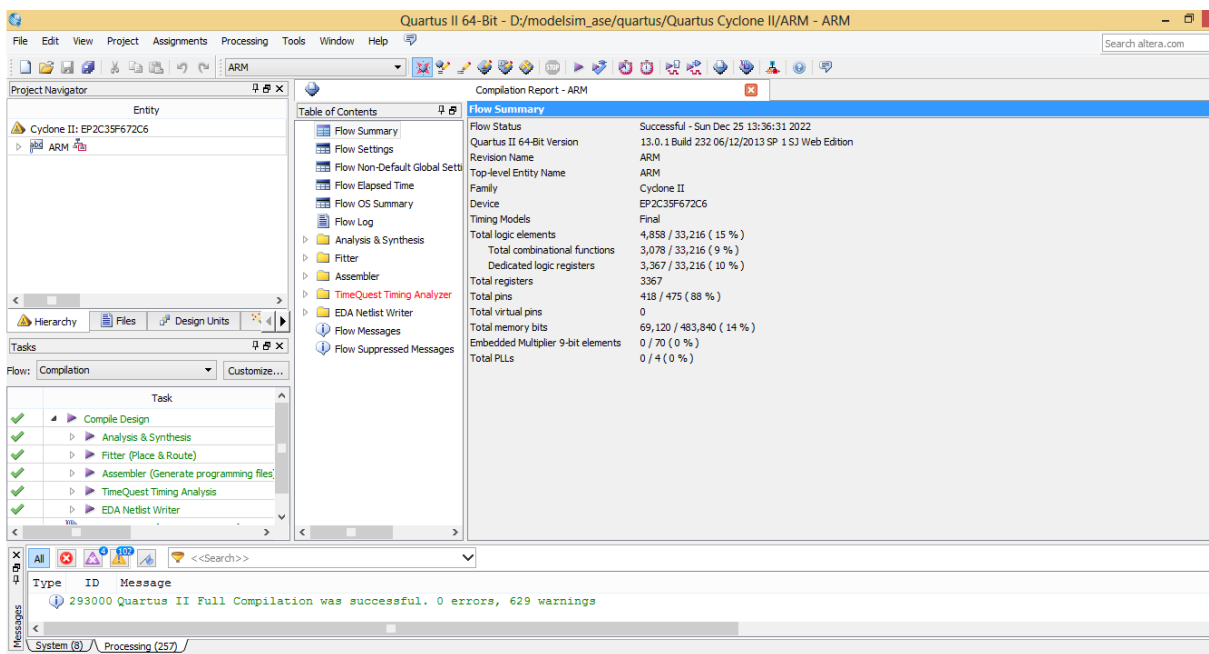
تصویر ۱: شکل موج خروجی رجیسترهای پردازنده ARM

خروجی Signal Tap:



تصویر ۲: خروجی رجیسترهای پردازنده ARM در Signal Tap

تعداد logic element های به کار رفته:



The screenshot shows the Quartus II 64-Bit IDE interface. The main window displays the 'Compilation Report - ARM' with the 'Flow Summary' tab selected. The report provides a detailed overview of the compilation process and resource usage for the ARM device.

Flow Status	Successful - Sun Dec 25 13:36:31 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	4,858 / 33,216 (15 %)
Total combinational functions	3,078 / 33,216 (9 %)
Dedicated logic registers	3,367 / 33,216 (10 %)
Total registers	3367
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	69,120 / 483,840 (14 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

The bottom of the window shows a message window with the following text:

```
293000 Quartus II Full Compilation was successful. 0 errors, 629 warnings
```

تصویر ۳: تعداد المان های به کار رفته در پردازنده ARM

Forwarding

هدف از ایجاد ماژول forwarding این است که کارایی پردازنده در صورت رخ دادن مخاطره، پایین نیاید و مجبور به توقف نشود؛ بلکه از داده های موجود در مراحل EXE یا MEM استفاده شود.

Forwarding Unit:

در واقع کار این ماژول، تولید سیگنال های کنترلی مربوط به دو MUX است که قرار است ورودی های ALU را تعیین کنند. سیگنال کنترلی select src1، برای تولید مقدار اول ALU و سیگنال کنترلی select src2 برای تولید مقدار دوم ALU است. لازم به ذکر است که هر دوی این سیگنال های کنترلی در ابتدا به صفر مقداردهی می شوند و در صورت برقراری شروط توضیح داده شده در پایین، مقدار می گیرند.

در این ماژول با استفاده از سیگنال های WB-MEM, WB-WB که در واقع سیگنال های Write Enable به ترتیب از ماژول های Memory, Write Back هستند استفاده می کند. در صورتی که src1 با رجیستر مقصد ماژول MEM یکسان و در عین حال WB-MEM فعال بود، سیگنال کنترلی select src1 مقدار ۱ به خود می گیرد که به این معنی است که باید داده مقدار اول ورودی ALU از دیتای حافظه خوانده شود. این مقایسه برای زمانی که سیگنال WB-WB هم یک باشد، رخ می دهد و در این حالت، سیگنال کنترلی select src1 مقدار ۲ به خود می گرد که به این معنی است که ورودی اول ALU باید از خروجی ماژول Write Back خوانده می شود.

این شروط را برای مقدار دوم ALU نیز به صورت مشابه چک کرده و سیگنال کنترلی select src2 را مشابه سیگنال قبلی مقداردهی می کنیم.

```

module Forwarding_Unit(WB_MEM, WB_WB, src_1_Register_File, src_2_Register_File, Reg_Dest_MEM, Reg_Dest_WB, select_src_1_alu, select_src_2_alu);

    input WB_MEM, WB_WB;
    input [3 : 0] src_1_Register_File, src_2_Register_File, Reg_Dest_MEM, Reg_Dest_WB;
    output reg [1 : 0] select_src_1_alu, select_src_2_alu;

    always@(*)begin
        select_src_1_alu = 2'b0;
        select_src_2_alu = 2'b0;

        if(WB_MEM & (src_1_Register_File == Reg_Dest_MEM))
            select_src_1_alu = 2'b01;
        else if(WB_WB & (src_1_Register_File == Reg_Dest_WB))
            select_src_1_alu = 2'b10;
        if(WB_MEM & (src_2_Register_File == Reg_Dest_MEM))
            select_src_2_alu = 2'b01;
        else if(WB_WB & (src_2_Register_File == Reg_Dest_WB))
            select_src_2_alu = 2'b10;

    end

endmodule

```

تصویر ۴: کد ماژول Forwarding Unit

```

module EXE_Stage(
    input clk, reset, WB_EN_in, MEM_R_EN_in, MEM_W_EN_in, Immediate_in, Status, Branch,
    input [1 : 0]select_src_1_alu, select_src_2_alu,
    input [3 : 0]EXE_CMD, Status_reg_in, Reg_Dest_in,
    input [11 : 0] shifter_operand,
    input [23 : 0] signed_immediate,
    input [31 : 0] PC, Val_Rn, Val_Rm_in, alu_result_MEM, alu_result_WB,
    output Status_En, Branch_En, MEM_R_EN_out, MEM_W_EN_out, WB_EN_out,
    output [3 : 0] Status_Reg_out, Reg_Dest_out,
    output [31 : 0] ALU_result, branch_address, Val_Rm_out);

    wire [31 : 0]Val_1, Val_2;

    assign Status_En = Status;
    assign Branch_En = Branch;
    assign branch_address = PC + 4 + ({8{signed_immediate[23]}}, signed_immediate) << 2);
    assign MEM_R_EN_out = MEM_R_EN_in;
    assign MEM_W_EN_out = MEM_W_EN_in;

    assign WB_EN_out = WB_EN_in;
    assign Reg_Dest_out = Reg_Dest_in;

    assign Val_Rm_out = (select_src_2_alu == 2'b00) ? Val_Rm_in :
        (select_src_2_alu == 2'b01) ? alu_result_MEM :
        (select_src_2_alu == 2'b10) ? alu_result_WB : 32'd0;

    assign Val_1 = (select_src_1_alu == 2'b00) ? Val_Rn :
        (select_src_1_alu == 2'b01) ? alu_result_MEM :
        (select_src_1_alu == 2'b10) ? alu_result_WB : 32'd0;

    Val_2_Generator VG2(Immediate_in, MEM_R_EN_in | MEM_W_EN_in, shifter_operand, Val_Rm_out, Val_2);

    ALU alu(Val_1, Val_2, Status_reg_in, EXE_CMD, Status_Reg_out, ALU_result);

endmodule

```

تصویر ۵: کد تغییر یافته ماژول EXE

با اضافه کردن ماژول Forwarding Unit، ماژول Hazard Unit نیز تغییر می کند که کد آن در تصویر زیر مشاهده می شود.

```
module hazard_unit(
    input two_src, move, WB_MEM, WB_EXE,
    input [3 : 0] RegDest_MEM, RegDest_EXE, Rn, Rm, input MEM_R_EN_EXE,
    output reg hazard
);

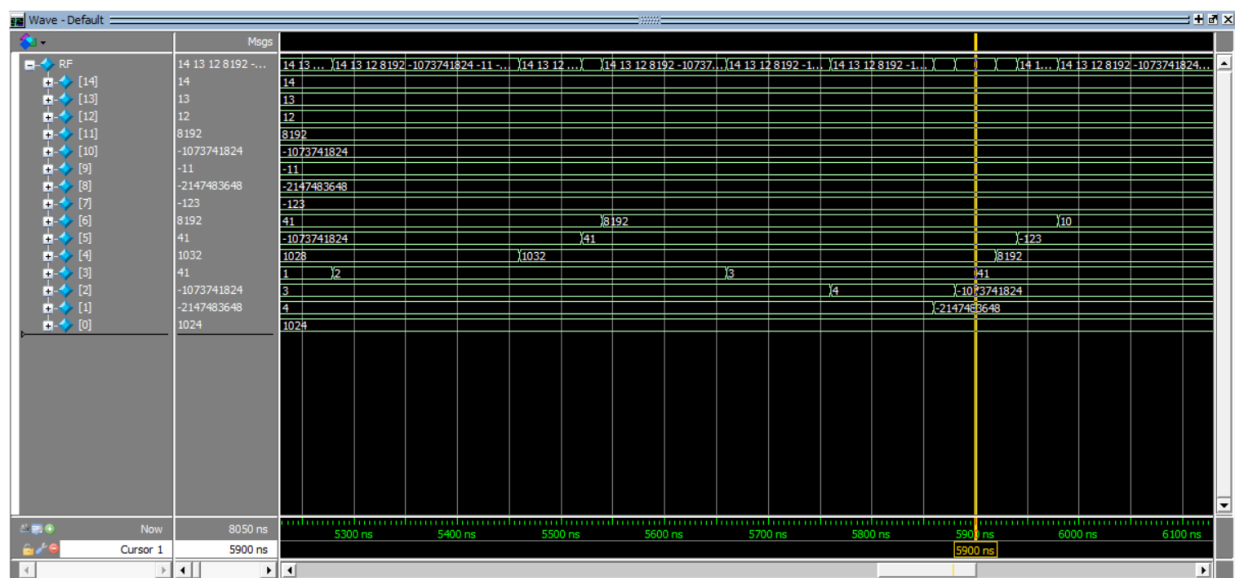
    always @* begin
        hazard = 1'b0;
        if(MEM_R_EN_EXE) begin
            if (two_src)
                if(WB_EXE & (RegDest_EXE == Rm))
                    hazard = 1'b1;
            if(WB_EXE & (RegDest_EXE == Rn))
                hazard = 1'b1;
        end
    end

endmodule
```

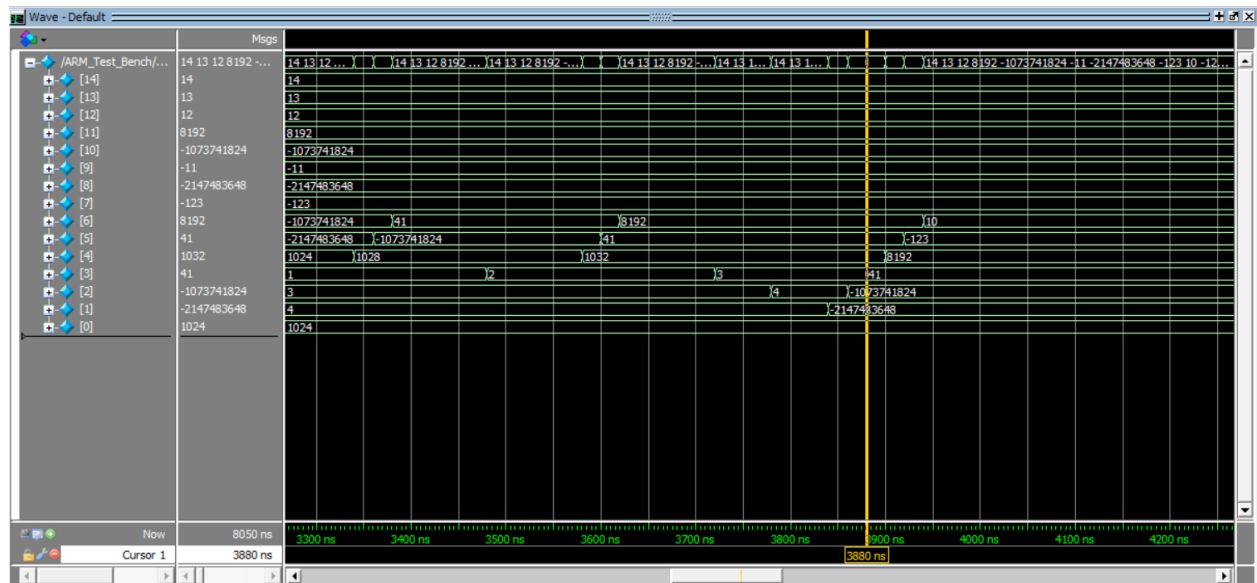
تصویر ۶: کد تغییر یافته ماژول Hazard Unit

باید در نظر داشت که ماژول Forwarding تمامی حالات مخاطره را تشخیص نمی دهد. تنها مخاطره ای که نمی تواند تشخیص دهد، زمانی است که یک دستور محاسباتی و پس از آن یک دستور load داشته باشیم. در این حالت به دلیل اینکه داده خروجی مرحله MEM از جنس عدد است؛ اما ما در دستور بعدی نیاز به آدرس داریم، نمی توانیم از این داده استفاده کنیم. در این حالت مجبور به stall کردن پردازنده هستیم. این استثنا در کد با چک کردن سیگنال MEM_R_EN_EXE که نشان دهنده دستور لود است، در نظر گرفته شده.

تصویر نتیجه شبیه سازی در Modelsim:



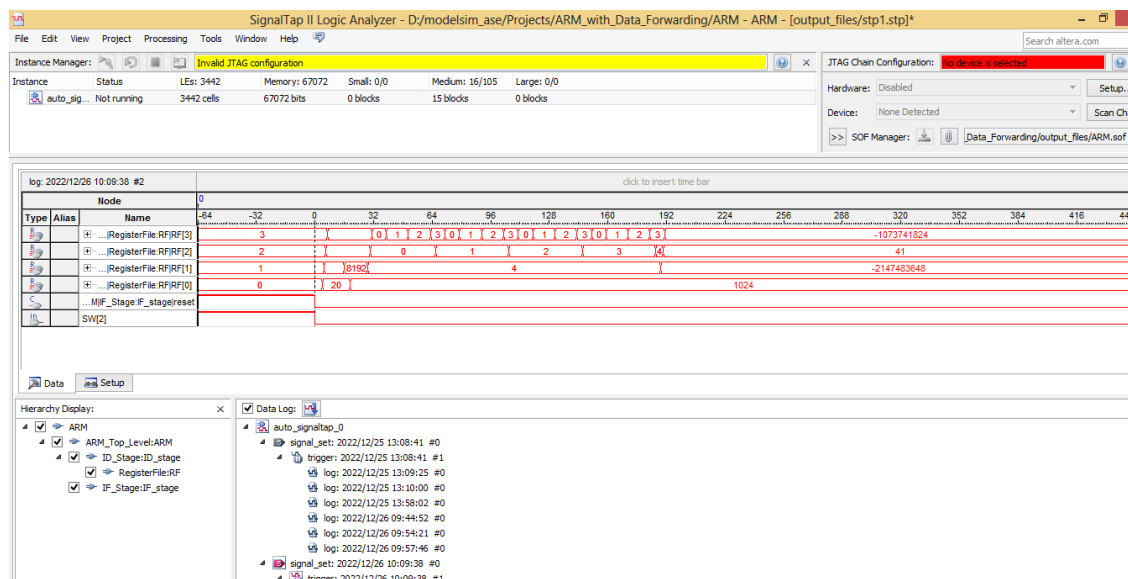
تصویر ۷: نتیجه شبیه سازی پردازنده ARM بدون ماژول Forwarding



تصویر ۸: نتیجه شبیه سازی پردازنده ARM به همراه ماژول Forwarding

همانطور که مشاهده می شود استفاده از ماژول Forwarding Unit باعث بهبود زمانی پردازنده شده است به طور یکه در حالت عادی ۵۹۰۰ نانوثانیه طول می کشید تا جواب نهایی حاصل شود؛ اما حالا ۳۸۸۰ نانو ثانیه طول می کشد.

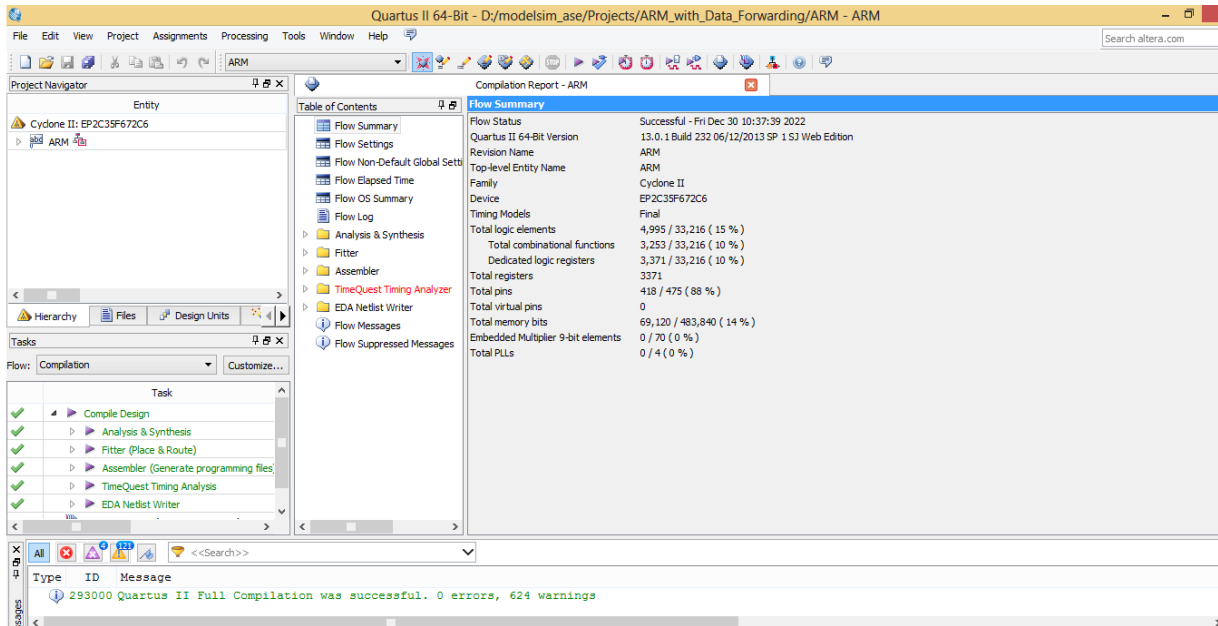
خروجی Signal Tap:



تصویر ۹: خروجی signal tap شبیه سازی پردازنده ARM به همراه Forwarding

همانطور که در تصویر signal tap مشاهده می شود، رجیسترها به درستی سورت نشده اند و مقادیر آن ها جابجا است. حدس ما پس از بررسی تمامی ماژول ها و در نظر گرفتن این نکته که خروجی در modelsim درست است، این است که نحوه عملکرد سخت افزارها در Quartus با Modelsim متفاوت است و همین موضوع ممکن است باعث چنین خطایی شود.

تعداد logic element های به کار رفته:



The screenshot shows the Quartus II 64-Bit IDE interface. The main window displays the 'Compilation Report - ARM' with the 'Flow Summary' tab selected. The report provides a detailed overview of the compilation process and resource usage for the ARM project.

Flow Summary	
Flow Status	Successful - Fri Dec 30 10:37:39 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	4,995 / 33,216 (15 %)
Total combinational functions	3,253 / 33,216 (10 %)
Dedicated logic registers	3,371 / 33,216 (10 %)
Total registers	3371
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	69,120 / 483,840 (14 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

The bottom status bar indicates: 293000 Quartus II Full Compilation was successful. 0 errors, 624 warnings.

تصویر ۱۰: تعداد المان های به کار رفته در پردازنده ARM به همراه Forwarding

SRAM

در مراحل قبلی حافظه پردازنده داخل خود آن بود؛ اما می دانیم که در عمل به این صورت نیست و هر پردازنده حافظه ای جداگانه دارد که به آن SRAM گفته می شود. در این بخش از آزمایش قصد داریم که این حافظه را ساخته، ارتباط آن را با خود پردازنده از طریق SRAM Controller برقرار کنیم و در نهایت تاثیر اضافه شدن این حافظه را بر سرعت عملکرد پردازنده مورد بررسی قرار دهیم. از آنجایی که حافظه SRAM برای هر بار عمل خواندن و یا نوشتن، به ۶ کلاک نیاز دارد، انتظار می رود که در نهایت، سرعت پردازنده بسیار کمتر شود. قدم اول برای افزودن SRAM به پردازنده، حذف Mempry Stage و قرار دادن SRAM Controller به جای آن است.

توضیح عملکرد SRAM:

کلاک این حافظه با کلاک پردازنده متفاوت است. عملیات خواندن در آن بدون نیاز به کلاک و عملیات نوشتن در آن با لبه بالارونده کلاک انجام می شود. نکته مهم در این حافظه سیگنال inout آن است که در واقع یک سیگنال ورودی و خروجی است. نام این سیگنال را SRAM-DQ گذاشته ایم. زمانی که داده ای از حافظه خوانده می شود، این سیگنال به عنوان خروجی و زمانی که داده ای در حافظه نوشته می شود، این سیگنال به عنوان ورودی در نظر گرفته می شود. سیگنال مهم دیگر در این ماژول، سیگنال SRAM_WE_N است که یک سیگنال active low است به این معنا که زمانی که مقدار آن صفر باشد، فعال است. فعال بودن آن به معنای نوشتن در حافظه و غیرفعال بودن آن به معنای خواندن از حافظه است.

```

module SRAM(
    input clk, reset, SRAM_WE_N,
    input [17 : 0] SRAM_ADDR,
    inout [15 : 0] SRAM_DQ
);

    reg [15 : 0] mem [511 : 0];

    assign SRAM_DQ = SRAM_WE_N ? mem[SRAM_ADDR] : 16'bz;

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

endmodule

```

تصویر ۱۱: کد ماژول SRAM

ماژولی که ارتباط بین حافظه SRAM و پردازنده را برقرار می کند، ماژول SRAM Controller است که در اینجا به توضیح آن می پردازیم:

کار این ماژول در ابتدا این است که تشخیص دهد آیا دستور از نوع store , load هست و اگر بود، با استفاده از یک شمارنده، تاخیری به اندازه ۶ کلاک تولید می کند. نکته مهم اما این است که تا زمانی که این ۶ کلاک به پایان نرسیده باشند، تمامی رجیسترهای پردازنده باید در حالت freeze قرار بگیرند و متوقف شوند به همین منظور سیگنالی به نام سیگنال ready داریم که در زمان عملکرد حافظه SRAM و در طول این ۶ کلاک، مقدار صفر به خود می گیرد. این سیگنال را به تمامی رجیسترهای پایپ لاین و همچنین رجیستر PC نیز می رود تا آن ها را freeze کند. بعد از اتمام ۶ کلاک، این سیگنال مقدار یک می گیرد و پردازنده می تواند به کار خود ادامه دهد. کار دیگری که در این ماژول انجام دادیم، align کردن آدرس ها بود. برای این کار ابتدا مانند گذشته، عدد ۱۰۲۴ را از تمامی آدرس ها کم کردیم و سپس آدرس ها را ۲ بیت به راست شیفت دادیم. از آنجایی که آدرس های SRAM ۱۶ بیتی هستند، برای هر بار خواندن یا نوشتن، ابتدا ۱۶ بیت کم ارزش داده و سپس ۱۶ بیت پر ارزش آن خوانده می شود.


```

module SRAM_controller(
    clk, reset, w_en, r_en, address, write_data, ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N,
    SRAM_CE_N, SRAM_OE_N, SRAM_ADDR, read_data, SRAM_DQ);

    input clk, reset, w_en, r_en;
    input [31 : 0] address, write_data;
    output reg ready;
    output SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N;
    output reg SRAM_WE_N;
    output reg [17 : 0] SRAM_ADDR;
    output reg [31 : 0] read_data;
    inout [15 : 0] SRAM_DQ;

    reg [15 : 0] DQ;

    reg [2 : 0] ps, ns, counter;
    reg init, count_en, ldu, ldd;

    wire co;

    assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'd0;

    always @(posedge clk, posedge reset) begin
        if(reset)
            ps <= 0;
        else
            ps <= ns;
        end

    always @(ps, w_en, r_en, co) begin
        case(ps)
            0: ns = (r_en) ? 1 : (w_en) ? 4 : 0;
            1: ns = 2;
            2: ns = 3;
            3: ns = 6;
            4: ns = 5;
            5: ns = 6;
            6: ns = (co) ? 7 : 6;
            7: ns = 0;
        endcase
    end
end

```

تصویر ۱۲

```

assign SRAM_DQ = DQ;

always @(ps, w_en, r_en) begin
    ready = 0; init = 0; DQ = 16'bz; count_en = 0; SRAM_WE_N = 1; ldu = 0; ldd = 0; SRAM_ADDR = 0;
    case(ps)
        0: begin
            ready = 1;
            if(w_en || r_en) begin init = 1; ready = 0; end
        end
        1: begin
            SRAM_WE_N = 1; count_en = 1;
            SRAM_ADDR = (address - 32'd1024) >> 1;
            ldd = 1;
        end
        2: begin
            SRAM_WE_N = 1; count_en = 1;
            SRAM_ADDR = ((address - 32'd1024) >> 1) + 1;
            ldu = 1;
        end
        3: begin
            count_en = 1;
        end
        4: begin
            SRAM_WE_N = 0;
            SRAM_ADDR = (address - 32'd1024) >> 1;
            DQ = write_data[15 : 0];
            count_en = 1;
        end
        5: begin
            SRAM_WE_N = 0;
            SRAM_ADDR = ((address - 32'd1024) >> 1) + 1;
            DQ = write_data[31 : 16];
            count_en = 1;
        end
        6: count_en = 1;
        7: ready = 1;
    endcase
end

always @(posedge clk, posedge reset) begin
    if(reset)

```

تصویر ۱۳

```

always @(posedge clk, posedge reset) begin
    if(reset)
        read_data = 0;
    else if(ldd)
        read_data[15 : 0] = SRAM_DQ;
    else if(ldu)
        read_data[31 : 16] = SRAM_DQ;
end

always@(posedge clk, posedge reset) begin
    if (reset)
        counter = 3'b0;
    else if(init)
        counter = 3'b0;
    else if (count_en)
        counter = counter + 3'd1;
end

assign co = (counter == 3'd3) ? 1 : 0;
endmodule

```

تصویر ۱۴

کد ماژول SRAM Controller در تصاویر ۱۲ تا ۱۴ مشاهده می شود.
در ادامه، تصویر کد ماژول هایی که تغییر کردند را مشاهده می کنیم.

```

module IF_Stage(
    input clk, reset, freeze, sram_ready, Branch_Taken,
    input [31 : 0] Branch_Address,
    output reg[31 : 0] PC,
    output [31 : 0]Instruction
);

reg [31 : 0] instruction_mem [46 : 0];

initial begin
    $readmemb("instruction_mem.txt", instruction_mem);
end

always @(posedge clk, posedge reset) begin

    if(reset)
        PC <= 0;
    else if(~freeze && sram_ready)
        PC <= (Branch_Taken) ? Branch_Address :
            (PC < 200) ? PC + 32'd4 : 200;

end

assign Instruction = instruction_mem[PC[31 : 2]];

endmodule

```

تصویر ۱۵: کد تغییر یافته ماژول IF Stage

```

module IF_Stage_Reg
(
    input clk, reset, freeze, sram_ready, flush,
    input [31 : 0] PC_in, instruction_in,
    output reg [31 : 0] PC_out, instruction_out
);

    always @(posedge clk, posedge reset) begin
        if (reset) begin
            PC_out = 32'b0;
            instruction_out = 32'b0;
        end
        else if (flush) begin
            PC_out = 32'b0;
            instruction_out = 32'b0;
        end
        else if (~freeze && sram_ready) begin
            PC_out = PC_in;
            instruction_out = instruction_in;
        end
    end

endmodule

```

تصویر ۱۶: کد تغییر یافته ماژول IF Stage Reg

```

module ID_Stage_Reg(
    input clk, reset, flush, sram_ready, Status_update_in, Branch_EN_in, MEM_R_EN_in, MEM_W_EN_in, WB_Enable_in, I_in,
    input [3 : 0] EXE_CMD_in, Reg_Dest_in, Status_Reg_in, Reg_File_src_1, Reg_File_src_2,
    input [11 : 0] shifter_operand_in,
    input [23 : 0] signed_immediate_in,
    input [31 : 0] PC_in, Rn_in, Rm_in,
    output reg Status_update_out, Branch_EN_out, mem_read, mem_write, WB_Enable, I,
    output reg [3 : 0] EXE_CMD, Reg_Dest_out, Status_Reg_out, src_1_reg_file, src_2_reg_file,
    output reg [11 : 0] shifter_operand,
    output reg [23 : 0] signed_immediate,
    output reg [31 : 0] PC_out, Rn_out, Rm_out
);

    always @(posedge clk, posedge reset) begin
        if (reset) begin
            Status_update_out = 1'b0;
            Branch_EN_out = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            WB_Enable = 1'b0;
            I = 1'b0;
            EXE_CMD = 4'b0;
            Reg_Dest_out = 4'b0;
            Status_Reg_out = 4'b0;
            src_1_reg_file = 4'b0;
            src_2_reg_file = 4'b0;
            shifter_operand = 12'b0;
            signed_immediate = 24'b0;
            PC_out = 32'b0;
            Rn_out = 32'b0;
            Rm_out = 32'b0;
        end
        else if (flush) begin
            Status_update_out = 1'b0;
            Branch_EN_out = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            WB_Enable = 1'b0;
            I = 1'b0;
            EXE_CMD = 4'b0;
            Reg_Dest_out = 4'b0;
            Status_Reg_out = 4'b0;
        end
    end

    reg_pcs_out = 4'b0;
    Status_Reg_out = 4'b0;
    src_1_reg_file = 4'b0;
    src_2_reg_file = 4'b0;
    shifter_operand = 12'b0;
    signed_immediate = 24'b0;
    PC_out = 32'b0;
    Rn_out = 32'b0;
    Rm_out = 32'b0;

    end
    else if (sram_ready) begin
        Status_update_out = Status_update_in;
        Branch_EN_out = Branch_EN_in;
        mem_read = MEM_R_EN_in;
        mem_write = MEM_W_EN_in;
        WB_Enable = WB_Enable_in;
        I = I_in;
        EXE_CMD = EXE_CMD_in;
        Reg_Dest_out = Reg_Dest_in;
        Status_Reg_out = Status_Reg_in;
        src_1_reg_file = Reg_File_src_1;
        src_2_reg_file = Reg_File_src_2;
        shifter_operand = shifter_operand_in;
        signed_immediate = signed_immediate_in;
        PC_out = PC_in;
        Rn_out = Rn_in;
        Rm_out = Rm_in;
    end

endmodule

```

تصاویر ۱۷ و ۱۸: کد تغییر یافته ماژول ID Stage Reg

```

module EXE_Stage_Reg(
    input clk, reset, sram_ready, MEM_R_EN_in, MEM_W_EN_in, WB_Enable_in,
    input [3 : 0] Reg_Dest_in,
    input [31 : 0] Rm_in, ALU_result_in,
    output reg mem_read, mem_write, WB_Enable,
    output reg [3 : 0] Reg_Dest_out,
    output reg [31 : 0] Rm_out, ALU_result
);

always @(posedge clk, posedge reset) begin
    if (reset) begin
        mem_write = 1'b0;
        mem_read = 1'b0;
        WB_Enable = 1'b0;
        Reg_Dest_out = 4'b0;
        Rm_out = 32'b0;
        ALU_result = 32'b0;
    end
    else if (sram_ready) begin
        mem_write = MEM_W_EN_in;
        mem_read = MEM_R_EN_in;
        WB_Enable = WB_Enable_in;
        Reg_Dest_out = Reg_Dest_in;
        Rm_out = Rm_in;
        ALU_result = ALU_result_in;
    end
end
endmodule

```

تصویر ۱۹: کد تغییر یافته ماژول EXE Stage Reg

```

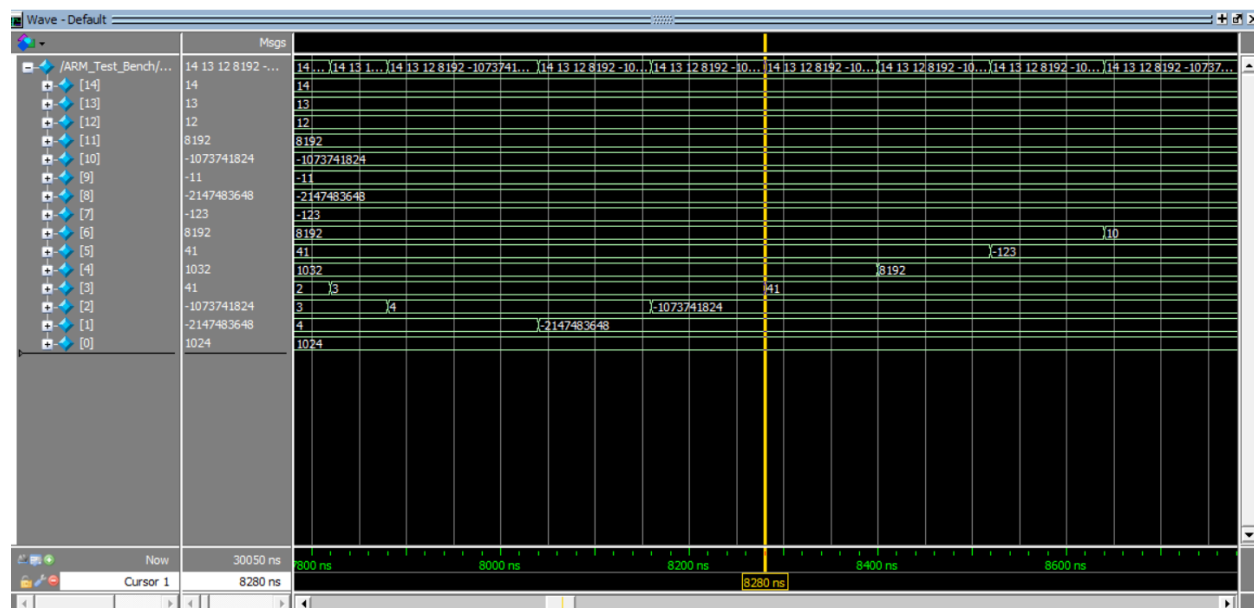
module MEM_Stage_Reg(
    input clk, reset,
    input MEM_R_EN_in, WB_Enable_in, sram_ready,
    input [3 : 0] Reg_Dest_in,
    input [31 : 0] mem_read_data_in, ALU_result_in,
    output reg mem_read, WB_Enable,
    output reg [3 : 0] Reg_Dest_out,
    output reg [31 : 0] mem_read_data, ALU_result
);

always @(posedge clk, posedge reset) begin
    if (reset) begin
        mem_read = 1'b0;
        WB_Enable = 1'b0;
        Reg_Dest_out = 4'b0;
        mem_read_data = 32'b0;
        ALU_result = 32'b0;
    end
    else if (!sram_ready) begin
        mem_read = mem_read;
        WB_Enable = WB_Enable;
        Reg_Dest_out = Reg_Dest_out;
        mem_read_data = mem_read_data;
        ALU_result = ALU_result;
    end
    else begin
        mem_read = MEM_R_EN_in;
        WB_Enable = WB_Enable_in;
        Reg_Dest_out = Reg_Dest_in;
        mem_read_data = mem_read_data_in;
        ALU_result = ALU_result_in;
    end
end
endmodule

```

تصویر ۲۰: کد تغییر یافته ماژول MEM Stage Reg

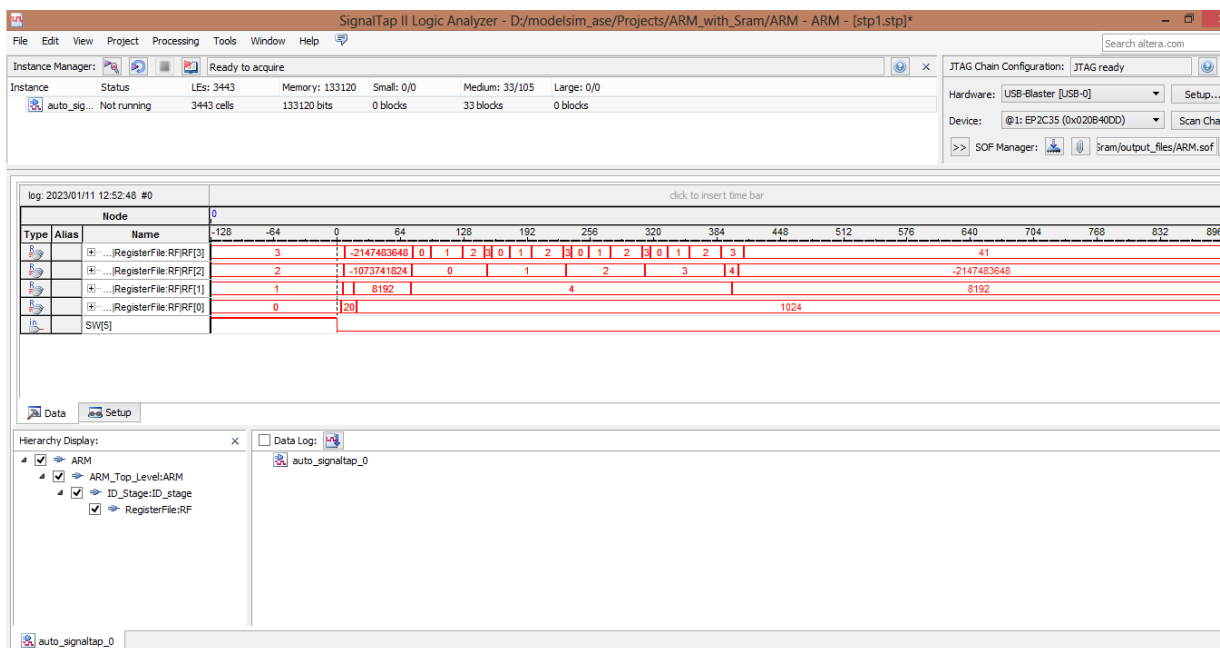
تصویر نتیجه شبیه سازی در Modelsim:



تصویر ۲۱: خروجی register file در پردازنده همراه با SRAM

همانطور که در تصویر ۲۱ مشاهده می شود، در این حالت ۸۲۸۰ نانوثانیه طول کشیده تا پردازنده تمامی دستورات را انجام دهد که از حالت های قبلی (ARM, Forwarding) زمان بیشتری است.

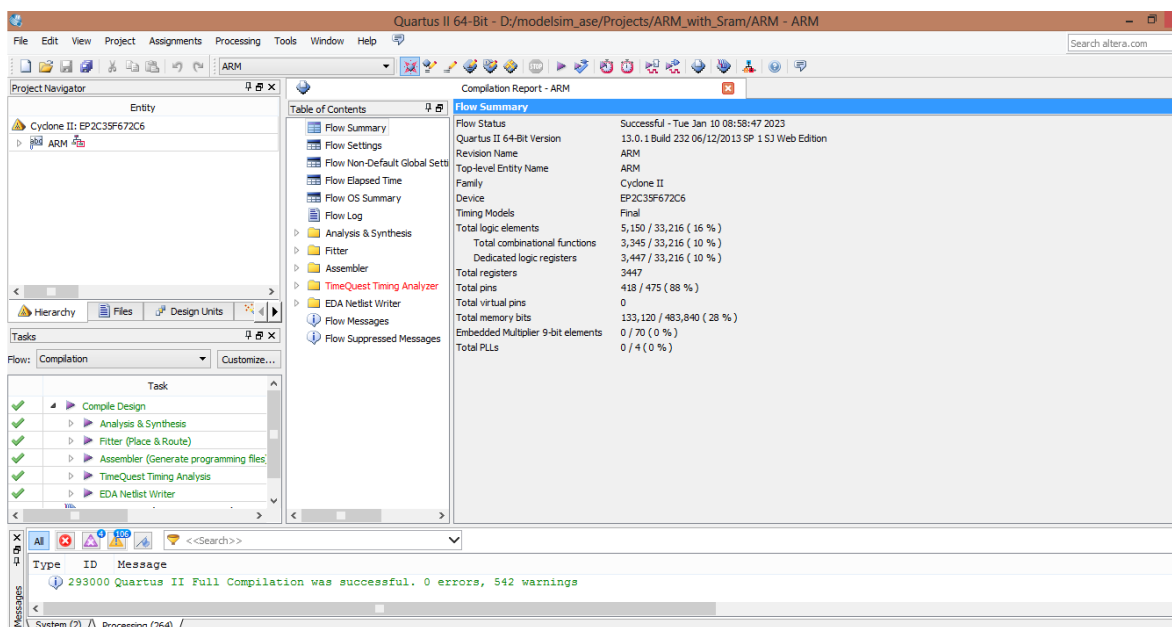
خروجی Signal Tap:



تصویر ۲۲: خروجی register file در پین‌دازنده همراه با SRAM در Signal Tap

همانطور که در تصویر ۲۲ مشاهده می شود، مقادیر موجود در رجیستر درست نیستند که مانند قسمت Forwarding حدس ما این است که تفاوت عملکرد Modelsim, Quartus باعث چنین شباهتی شده است.

تعداد logic element های به کار رفته:



تصویر ۲۳: تعداد المان های به کار رفته در SRAM

CACHE

در آزمایش قبل دیدیم که به کار بردن حافظه SRAM تاثیری زیادی بر زمان عملکرد پردازنده می گذارد. برای حل این مشکل از حافظه ای کوچکتر و سریعتر به نام cache استفاده می کنیم. نحوه عملکرد این حافظه به این صورت است که در هر دستوری اگر نیاز داشتیم داده ای را بخوانیم ابتدا به سراغ cache می رویم و اگر داده در آن ذخیره شده بود، خواندن از cache صورت می گیرد و در غیر این صورت مجبوریم که مجدداً به سراغ SRAM برویم. با هر بار خواندن از sram محتوای cache را نیز با ذخیره داده های قبل و بعد به روز می کنیم. نکته مهم این است که اگر در SRAM عمل نوشتن را انجام دادیم و آدرس آن در CACHE نیز وجود داشت باید به گونه ای CACHE را آپدیت کنیم که داده این آدرس دیگر معتبر نیست.

توضیح نحوه عملکرد CACHE:

در ابتدا برای هر کدام از بخش های کش (way0 , way1) یک سیگنال hit_way در نظر می گیریم.

شرط یک شدن این سیگنال این است که tag آدرس ورودی با tag مربوط به index کش برابر باشد و همچنین سیگنال valid نیز که از معتبر بودن داده داخل کش اطلاع می دهد، یک باشد. مرحله بعدی مشخص کردن محل دقیق داده مورد نیاز است که در کدام way و کدام word قرار دارد. برای مشخص کردن way از سیگنال hit_way و برای مشخص کردن word، از بیت سوم offset استفاده می کنیم.

برای نوشتن در SRAM: داده خوانده شده از کش را روی SRAM_DQ قرار می دهیم و آدرس SRAM را که همان سیگنال SRAM_ADDR است مقدار دهی می کنیم. خواندن از CACHE: با توجه به بیت سوم offset مقدار ادرس کوچکتر و بزرگتر را مشخص می کنیم.

نوشتن در cache: ابتدا باید سیگنال cache_write مقدار یک داشته باشد. اگر سیگنال last_used_block در index مقدار یک داشته باشد، باید در way1 بنویسیم و برعکس. برای نوشتن در کش باید چند سیگنال را مقدار دهی کنیم. اول سیگنال valid را در index مربوطه یک می کنیم. باید tag block در index مربوطه را با tag آدرس ورودی برابر قرار دهیم. اگر بیت سوم

offset مقدار یک داشته باشد، نوشتن داده اول در بیت های LSB تمام شده و باید داده دوم را در MSB بنویسیم و برعکس.

مرحله آخر: در صورتی که sram_ready, write_enable هر دو یک باشند، یعنی در کلاک ششم sram هستیم و کار نوشتن در آن تمام شده. در اینجا است که باید وضعیت اعتبار داده موجود در کش را به روز کنیم. برای این کار باید ببینیم آیا آدرسی که در sram در آن نوشته شده، در کش وجود دارد یا خیر که این در واقع به معنی یک یا صفر بودن سیگنال hit است. در صورت وجود این آدرس، باید سیگنال valid مربوط به آن را صفر کنیم که به معنای عدم اعتبار داده و تغییر آن است.

در ادامه در تصاویر ۲۴ و ۲۵، کد مربوط به مسیر داده کش مشاهده می شود.

```

integer i;
wire hit_way_0, hit_way_1;
wire [2:0] offset;
wire [5:0] index;
wire [8:0] tag;
wire [63:0] way_data;
reg least_used_block [63:0];
reg valid_block_0 [63:0];
reg valid_block_1 [63:0];
reg [8:0] tag_block_0 [63:0];
reg [8:0] tag_block_1 [63:0];
reg [63:0] data_block_0 [63:0];
reg [63:0] data_block_1 [63:0];

assign offset = address[2:0];
assign index = address[8:3];
assign tag = address[10:2];
assign hit_way_0 = (tag_block_0[index] == tag) & valid_block_0[index];
assign hit_way_1 = (tag_block_1[index] == tag) & valid_block_1[index];
assign hit = hit_way_0 | hit_way_1;
assign freeze_arm = (hit & ~write_enable) | sram_ready;
assign SRAM_w_en = write_enable;
assign SRAM_r_en = (~hit & read_enable);
assign SRAM_ADDR = address;
assign way_data = hit_way_0? data_block_0[index] : (hit_way_1? data_block_1[index] : 64'bz);
assign read_data = offset[2]? way_data[63:32] : way_data[31:0];
assign SRAM_W_DQ = write_data;

always @(posedge clk, posedge reset) begin: write_to_cache
  if (reset) begin
    for (i=0;i<64;i=i+1) begin
      valid_block_0[i] = 1'b0;
      valid_block_1[i] = 1'b0;
      tag_block_0[i] = 9'b0;
      tag_block_1[i] = 9'b0;
      least_used_block[i] = 1'b0;
    end
  end
  if (cache_write)
    if (least_used_block[index]) begin: write_to_way_1
      valid_block_1[index] = 1'b1;
      tag_block_1[index] = tag;
    end
  end
end

```

تصویر ۲۴

```

----
end
if (cache_write)
  if (least_used_block[index]) begin: write_to_way_1
    valid_block_1[index] = 1'b1;
    tag_block_1[index] = tag;
    if (offset[2])
      data_block_1[index][63:32] = SRAM_DQ;
    else
      data_block_1[index][31:0] = SRAM_DQ;
    end
  else begin: write_to_way_0
    valid_block_0[index] = 1'b1;
    tag_block_0[index] = tag;
    if (offset[2])
      data_block_0[index][63:32] = SRAM_DQ;
    else
      data_block_0[index][31:0] = SRAM_DQ;
    end
  end
  if (sram_ready & write_enable) begin
    if (hit_way_0)
      valid_block_0[index] = 1'b0;
    if (hit_way_1)
      valid_block_1[index] = 1'b0;
  end
  if (sram_ready & read_enable) begin
    if (hit_way_0)
      least_used_block[index] = 1'b1;
    if (hit_way_1)
      least_used_block[index] = 1'b0;
  end
end
end

endmodule

```

تصویر ۲۵

توضیح نحوه عملکرد cache controller :

در اینجا ما یک ماژول جداگانه برای کنترل کننده cache نداریم بلکه همان کنترل کننده sram است که یک خط به آن اضافه شده است. در cache ما سیگنال های کنترلی sram را جهت خواندن از آن و یا نوشتن در آن تعیین می کنیم.

در حالتی که می خواهیم از sram بخوانیم (فعال بودن read enable) و همچنین کنترل کننده در استیت ششم قرار داشته باشد، سیگنال cache write را یک می کنیم تا داده برای cache معتبر باشد تا داده ای که در cache نبود از sram خوانده شده و در cache نوشته شود.

```

module SRAM_controller(
    clk, reset, w_en, r_en, address, write_data, ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N,
    SRAM_CE_N, SRAM_OE_N, SRAM_ADDR, read_data, SRAM_DQ, cache_write);

    input clk, reset, w_en, r_en;
    input [31 : 0] address, write_data;
    output reg ready;
    output SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N;
    output reg SRAM_WE_N;
    output reg [17 : 0] SRAM_ADDR;
    output reg [31 : 0] read_data;
    inout [15 : 0] SRAM_DQ;
    output cache_write;

    reg [15 : 0] DQ;

    reg [2 : 0] ps, ns, counter;
    reg init, count_en, ldu, ldd;

    wire co;

    assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'd0;

    always @(posedge clk, posedge reset) begin
        if(reset)
            ps <= 0;
        else
            ps <= ns;
        end

    always @(ps, w_en, r_en, co) begin
        case(ps)
            0: ns = (r_en) ? 1 : (w_en) ? 4 : 0;
            1: ns = 2;
            2: ns = 3;
            3: ns = 6;
            4: ns = 5;
            5: ns = 6;
            6: ns = (co) ? 7 : 6;
            7: ns = 0;
        endcase
    end

```

تصویر ۲۶

```

assign SRAM_DQ = DQ;

always @(ps, w_en, r_en) begin
    ready = 0; init = 0; DQ = 16'bz; count_en = 0; SRAM_WE_N = 1; ldu = 0; ldd = 0; SRAM_ADDR = 0;
    case(ps)
        0: begin
            ready = 1;
            if(w_en || r_en) begin init = 1; ready = 0; end
        end
        1: begin
            SRAM_WE_N = 1; count_en = 1;
            SRAM_ADDR = (address - 32'd1024) >> 1;
            ldd = 1;
        end
        2: begin
            SRAM_WE_N = 1; count_en = 1;
            SRAM_ADDR = ((address - 32'd1024) >> 1) + 1;
            ldu = 1;
        end
        3: begin
            count_en = 1;
        end
        4: begin
            SRAM_WE_N = 0;
            SRAM_ADDR = (address - 32'd1024) >> 1;
            DQ = write_data[15 : 0];
            count_en = 1;
        end
        5: begin
            SRAM_WE_N = 0;
            SRAM_ADDR = ((address - 32'd1024) >> 1) + 1;
            DQ = write_data[31 : 16];
            count_en = 1;
        end
        6: count_en = 1;
        7: ready = 1;
    endcase
end

```

تصویر ۲۷

```

always @(posedge clk, posedge reset) begin
    if(reset)
        read_data = 0;
    else if(ldd)
        read_data[15 : 0] = SRAM_DQ;
    else if(ldu)
        read_data[31 : 16] = SRAM_DQ;
end

always@(posedge clk, posedge reset) begin
    if (reset)
        counter = 3'b0;
    else if(init)
        counter = 3'b0;
    else if (count_en)
        counter = counter + 3'd1;
end

assign cache_write = (r_en & (ps == 6)) ? 1 : 0;
assign co = (counter == 3'd3) ? 1 : 0;

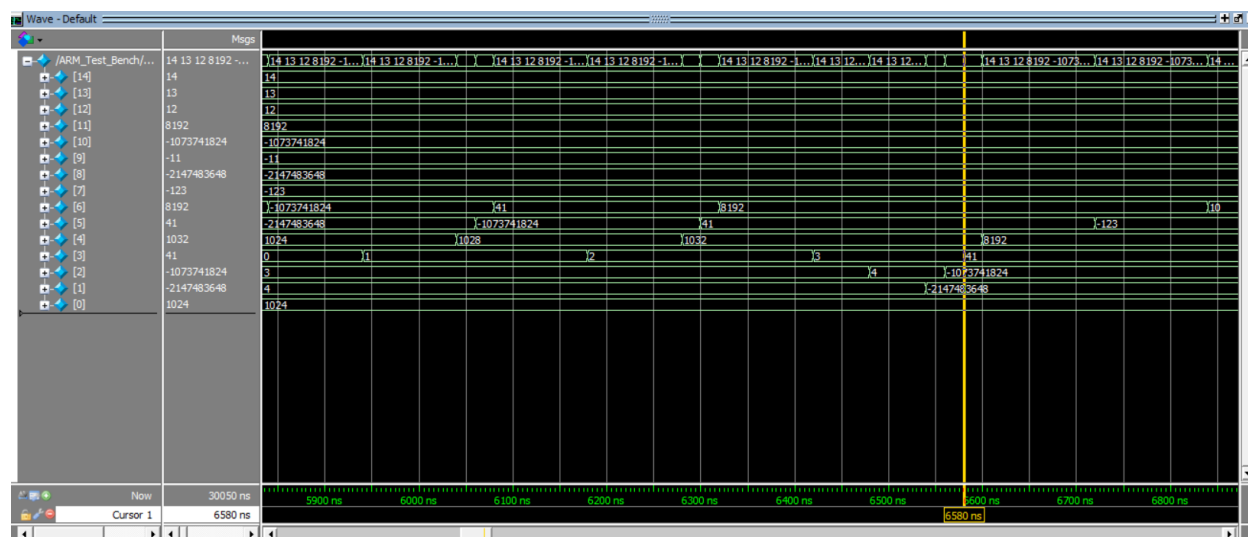
endmodule

```

تصویر ۲۸

کد مربوط به SRAM_Controller در تصاویر ۲۶ تا ۲۸ مشاهده می شود.

تصویر نتیجه شبیه سازی در Modelsim:



تصویر ۲۹: خروجی register file در پردازنده همراه با SRAM و CACHE

همانطور که در تصویر ۲۶ مشاهده می شود، زمان عملکرد پردازنده با افزودن cache بهبود پیدا کرده است و پس از ۶۵۸۰ نانو ثانیه به نتیجه می رسد که این زمان نسبت به حالتی که فقط SRAM داشتیم، ۱۷۰۰ نانوثانیه بهبود پیدا کرده است.