آزمایشگاه معماری کامپیوتر، بهار 1404

گزارش‌کار پیاده‌سازی پردازنده ARM

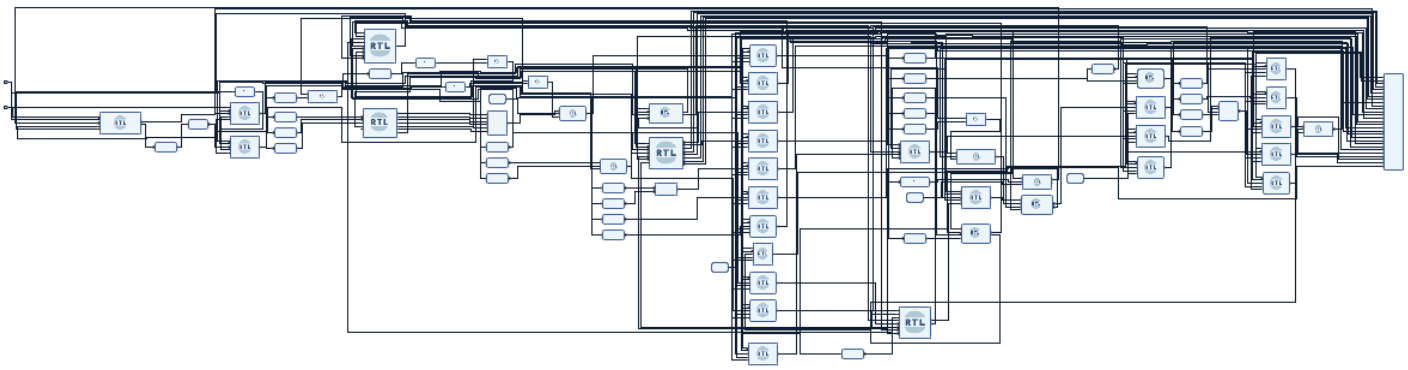
رضا چهرقانی *(810101401)*، سید عرشیا متقیان *(810101503)* و محمد مهدی صمدی *(810101465)*

مقدمه

در این گزارش، مراحل طراحی و پیاده‌سازی پردازنده مبتنی بر معماری ARM که به عنوان پروژه آزمایشگاه معماری کامپیوتر تعریف شده بود، شرح داده می‌شود. هدف اصلی این پروژه، به کارگیری مفاهیم تئوری درس معماری کامپیوتر (به خصوص طراحی Pipeline)، در عمل و مشاهده نتیجه روی برد است.

این پردازنده، ۳۲ بیتی با پنج استیج IF، ID، EXE، MEM و WB در پایپ‌لاین است که از ۱۳ دستورالعمل اصلی پشتیبانی می‌کند (یک دستور هم در امتحان پایانی افزوده شد). در این آزمایشگاه، مرحله به مرحله معماری کلی و اجزای مختلف پردازنده را که در RTL Level طراحی شده و در اختیارمان قرار گرفته بود را به زبان Verilog پیاده‌سازی کرده و برای اطمینان از صحت عملکرد، هر بخش را به صورت جداگانه و سپس کل پردازنده را به صورت یکپارچه در نرم‌افزار Vivado شبیه‌سازی و همچنین روی برد آزمایش کردیم.

دیاگرام پردازنده



توضیح کد

در این بخش، هر تکه از پردازنده را از نظر طراحی و کد پیاده‌سازی شده بررسی می‌کنیم.

- **طراحی Instruction Fetch**

مرحله‌ی Instruction Fetch در معماری ARM که در بخش بنفش رنگ تصویر مربوط به معماری کلی پردازنده ARM ساده شده در دستور آزمایش قرار دارد، وظیفه دارد دستورالعمل بعدی را از حافظه واکشی کند. در این مرحله، ابتدا واحد PC یا شمارنده برنامه آدرس دستورالعمل بعدی را تولید می‌کند. این آدرس به حافظه دستورالعمل داده می‌شود تا دستور مورد نظر خوانده شود. سپس این دستور برای مرحله بعدی یعنی Decode فرستاده می‌شود. در صورت وجود دستور پرش یا branch، مسیر اجرای برنامه ممکن است تغییر کند که با استفاده از یک مالتی‌پلکسر (MUX) آدرس جدید برای PC انتخاب می‌شود. همچنین سیگنال‌هایی مانند Freeze برای توقف موقت اجرای دستور در صورت بروز خطر (hazard) و Flush برای پاک‌سازی دستور اشتباه به کار می‌روند. به‌طور خلاصه، این مرحله مسئول شروع فرآیند اجرای هر دستور با واکشی آن از حافظه است.

- **طراحی Instruction Decode**

در مرحله‌ی Instruction Decode در معماری ARM که در بخش نارنجی رنگ تصویر مربوط به معماری کلی پردازنده ARM ساده شده در دستور آزمایش قرار دارد، پردازنده دستور واکشی‌شده را تفسیر می‌کند تا مشخص شود چه عملیاتی باید انجام شود و چه داده‌ها یا رجیستر هایی درگیر هستند. در این مرحله، دستورالعملی که از مرحله قبل آمده وارد Control Unit می‌شود. این واحد، سیگنال‌های کنترلی لازم برای سایر بخش‌های پردازنده را تولید می‌کند تا بدانند در مراحل بعدی چه کارهایی باید انجام دهند.

هم‌زمان، مقادیر مورد نیاز از رجیسترها از طریق Register File خوانده می‌شوند. در این بخش، آدرس دو رجیستر ورودی (که معمولاً در ساختار دستور مشخص هستند) به رجیستر فایل داده می‌شود و محتوای آن‌ها (به عنوان ورودی‌های عملیات بعدی) خوانده می‌شود. خروجی این رجیسترها به مرحله‌ی Execute فرستاده می‌شود. اگر دستور نیاز به مقدار Immediate داشته باشد، این مقدار توسط Sign Extend گسترش داده می‌شود تا به شکل صحیح برای استفاده در محاسبات آماده باشد.

در مجموع، در این مرحله، نوع دستور تشخیص داده می‌شود، داده‌های لازم از رجیسترها واکشی می‌شود، سیگنال‌های کنترلی تولید می‌شوند، و همه‌چیز برای اجرای دستور در مرحله بعدی آماده می‌گردد.

در این قسمت یکی از بخش های مهم Condition Check است که کد آن در بخش زیر مشخص است، در این بخش باید چهار فلگ به Zero, Overflow, Negative, Carry مشخص شوند. وقتی یک دستور مانند مقایسه یا عملیات ریاضی اجرا می‌شود، فلگ‌ها وضعیت را به‌روزرسانی می‌کند. سپس دستورهایی که به صورت شرطی نوشته شده‌اند (مانند پرش شرطی یا انتقال شرطی) با بررسی این‌ها تصمیم می‌گیرند که اجرا شوند یا نادیده گرفته شوند که برای نادیده گرفته شدن به نتیجه دقت می‌کنیم و اگر Hazard باشد یا در همین قسمت condition check هردو برقرار نباشند باعث میشود که تفسیر های که از مرحله ی قبل بدست آوردیم را به قسمت EXE ندهیم و به جای آن مقادیر صفر پاس داده خواهد شد.

```
module Condition_Check(
    input [3:0] ZCNV, input [3:0] cond,
    output reg cond_out
    );
    wire Z, C, N, V;
    assign Z = ZCNV[3];
    assign C = ZCNV[2];
    assign N = ZCNV[1];
    assign V = ZCNV[0];
    always @(*) begin
        case (cond)
            4'b0000: cond_out = Z;
            4'b0001: cond_out = ~Z;
            4'b0010: cond_out = C;
            4'b0011: cond_out = ~C;
            4'b0100: cond_out = N;
            4'b0101: cond_out = ~N;
            4'b0110: cond_out = V;
            4'b0111: cond_out = ~V;
            4'b1000: cond_out = C & ~Z;
            4'b1001: cond_out = ~C & Z;
            4'b1010: cond_out = (N == V);
            4'b1011: cond_out = (N != V);
            4'b1100: cond_out = (~Z) & (N == V);
            4'b1101: cond_out = ~((~Z) & (N == V));
            4'b1110: cond_out = 1'b1;
            default: cond_out = 1'b0;
        endcase
    end
endmodule
```

- **طراحی Execution**

بخش Execution در معماری پردازنده ARM وظیفه اصلی اجرای دستورالعمل‌ها را برعهده دارد و عملیات‌های محاسباتی، منطقی و انتقال داده‌ها را مدیریت می‌کند. پس از رمزگشایی دستور در مرحله قبل، مقادیر ورودی برای اجرای دستور آماده می‌شوند؛ دو مقدار اصلی تحت عنوان Val1 و Val2 دارد که Val1 از یکی از رجیسترهای عمومی گرفته می‌شود و Val2 توسط واحد Val2 Generate تولید می‌شود؛ که می‌تواند مقدار علامت‌دار immediate باشد (Signed_EX_imm24) یا مقدار رجیستر Val_Rm باشد. سپس این دو مقدار وارد ALU شده و بسته به نوع دستور، عملیات‌هایی مانند جمع، تفریق، AND، OR و... روی آن‌ها انجام می‌گیرد. فرمان انجام

عملیات از طریق سیگنال‌های EXE_CMD به ALU ارسال می‌شود تا نوع عمل را مشخص کند. خروجی ALU که تحت عنوان Val_ALU شناخته می‌شود، برای مراحل بعدی مثل MEM یا WB ارسال می‌گردد. سیگنال‌های کنترلی مانند EXE_WB_EN مشخص می‌کنند که آیا نتیجه باید در رجیستر ذخیره شود یا نه، و MEM_W_EN نیز عملیات نوشتن در حافظه را فعال می‌کند. در نهایت، بسته به اینکه دستور از نوع محاسباتی باشد یا حافظه‌ای، داده پردازش‌شده یا در حافظه ذخیره شده یا در یکی از رجیسترهای مقصد (که توسط MEM_Dest مشخص می‌شود) بازنویسی خواهد شد. این فرآیند باعث می‌شود که پردازنده بتواند به‌صورت دقیق، سریع و بهینه عملیات‌های مختلف را انجام دهد و دستورالعمل‌ها را مرحله‌به‌مرحله اجرا کند، در کد قسمت پایین که برای Val2Generate است که با توجه به اینکه کدام یک از حالت ها پیش می آید و کدام یک از شیفت ها مدنظر ما هست باید طبق آن عمل کنیم، برای مثال برای شیفت های چرخشی ما از یک کپی دیگر استفاده کردیم و از هر طرف که شیفت میدادیم به آن طرف اضافه می‌کردیم و به این صورت لاجیک قسمت شیفت چرخشی پیاده سازی شد.

```
module Val2Genrate(
    input [31:0] Val_Rm, input [12:0] Shift_operand_I,
    input MEM_EN, output reg [31:0] outt
    );
    wire [11:0] Shift_operand = Shift_operand_I[12:1];
    wire imm = Shift_operand_I[0];
    reg [31:0] tmp; reg [63:0] tmp2;

    always @(*) begin
        if (MEM_EN) begin
            outt = {{20{Shift_operand[11]}}, Shift_operand};
        end else if (imm) begin
            tmp = {24'b0, Shift_operand[7:0]};
            tmp2 = {tmp, tmp} >> (2 * Shift_operand[11:8]);
            outt = tmp2[31:0];
        end else begin
            case (Shift_operand[6:5])
                2'b00: outt = Val_Rm << Shift_operand[11:7];
                2'b01: outt = Val_Rm >> Shift_operand[11:7];
                2'b10: outt = Val_Rm >>> Shift_operand[11:7];
                2'b11: begin
                    tmp2 = {Val_Rm, Val_Rm} >> Shift_operand[11:7];
                    outt = tmp2[31:0];
                end
            endcase
        end
    end
endmodule
```

- **طراحی Memory**

در معماری پردازنده ARM، مرحله Memory وظیفه دسترسی به حافظه اصلی را برعهده دارد و معمولاً پس از اجرای عملیات توسط ALU وارد این مرحله می‌شویم. اگر دستورالعملی که پردازش شده load یا store باشد، مرحله Memory فعال خواهد شد. در این مرحله، مقدار آدرس حافظه که قبلاً توسط ALU تولید شده (Val_ALU)، به عنوان آدرس مقصد برای عملیات حافظه استفاده می‌شود. اگر عملیات از نوع بارگذاری باشد، داده مورد نظر از آن آدرس در حافظه خوانده شده و به مرحله بعدی یعنی Write Back فرستاده می‌شود؛ در حالی‌که اگر عملیات از نوع ذخیره‌سازی باشد، مقدار موجود در رجیستر مشخص (Val_Rm) در آن آدرس حافظه نوشته می‌شود.

کنترل این عملیات با سیگنال‌های خاصی انجام می‌شود، از جمله MEM_R_EN برای فعال‌سازی خواندن از حافظه و MEM_W_EN برای فعال‌سازی نوشتن در آن. همچنین MEM_Dest نقش تعیین‌کننده‌ای در مشخص‌کردن رجیستر مقصد دارد، مخصوصاً در عملیات بارگذاری که داده خوانده‌شده باید در رجیستری ذخیره گردد. در نهایت، داده‌ای که از حافظه خوانده یا در آن نوشته شده، به مرحله Write Back ارسال می‌شود تا اگر لازم بود در رجیسترهای عمومی ذخیره شود. این مرحله یکی از کلیدی‌ترین بخش‌ها در تعامل پردازنده با حافظه اصلی سیستم بوده و نقش تعیین‌کننده‌ای در اجرای موفق دستورات مرتبط با داده‌ها ایفا می‌کند.

- **طراحی Write Back**

در معماری پردازنده ARM، مرحله Write Back آخرین مرحله در چرخه اجرای دستورالعمل است و وظیفه آن نوشتن نتایج نهایی در رجیسترهای مقصد می‌باشد. پس از این‌که عملیات در مراحل Execution و Memory انجام شد، داده‌های حاصل باید در یکی از رجیسترهای عمومی ذخیره شوند تا پردازنده در مراحل بعدی بتواند به آن‌ها دسترسی داشته باشد. دو منبع اصلی برای داده‌هایی که در Write Back ذخیره می‌شوند، خروجی ALU یعنی Val_ALU و داده خوانده‌شده از حافظه یعنی MEM_Read_Value هستند. اینکه کدام‌یک انتخاب شود، توسط سیگنال WB_MUX_SEL مشخص می‌گردد.

سیگنال کنترلی EXE_WB_EN نیز تعیین می‌کند که آیا عملیات نوشتن در رجیستر باید انجام شود یا خیر. همچنین رجیستر مقصد با استفاده از ورودی Dest مشخص می‌شود تا داده نهایی در محل مناسب ذخیره گردد. این مرحله اهمیت زیادی دارد، زیرا بدون آن، نتایج عملیات پردازشی یا خواندن از حافظه در معماری پردازنده حفظ نمی‌شود و در چرخه‌های بعدی قابل استفاده نخواهد بود. مرحله Write Back در واقع نقطه اتصال پردازش داده با پایگاه‌داده رجیستر هاست و نقش کلیدی در پایداری و تداوم اجرای برنامه‌ها ایفا می‌کند.

- **طراحی Hazard Unit**

در معماری پردازنده ARM، واحد(Hazard Unit) نقش بسیار مهمی در حفظ پایداری و هماهنگی مراحل مختلف اجرای دستورالعمل‌ها ایفا می‌کند. این واحد وظیفه دارد تداخل‌های احتمالی بین مراحل متعدد اجرای دستور را شناسایی و مدیریت کند؛ به‌ویژه زمانی‌که چند دستور پشت‌سرهم در حال اجرا هستند و ممکن است منابع مشترک یا وابستگی داده‌ای داشته باشند.

در پردازنده‌هایی با معماری خط لوله (pipeline)، دستورها در مراحل مختلف مانند ،Fetch ،Decode Execution، Memory و Write Back به‌طور همزمان پردازش می‌شوند. اما در این شرایط ممکن است یک دستور هنوز نتیجه‌اش در دسترس نباشد و دستور بعدی به همان داده نیاز داشته باشد. به این حالت "Data Hazard" گفته می‌شود. واحد Hazard وظیفه دارد این شرایط را تشخیص دهد و برای جلوگیری از خطا یا اجرای نادرست، راه‌کارهایی ارائه دهد.

یکی از روش‌های مدیریت این تداخل‌ها، فعال‌سازی سیگنال‌هایی مانند Forwarding (ارسال پیشرفته داده از مرحله جلوتر به عقب‌تر) یا Stalling (ایست موقت یکی از مراحل پردازش) است. واحد Hazard بررسی می‌کند که آیا داده‌ای که قرار بوده از حافظه خوانده یا در رجیستر نوشته شود، مورد نیاز دستور بعدی هست یا نه. اگر باشد، جریان پردازش متوقف یا اصلاح می‌شود تا داده موردنظر آماده شود.

همچنین واحد Hazard در مدیریت کنترل‌های منطقی مثل انشعاب‌ها (Branch) و پرش‌ها نقش مهمی دارد؛ زیرا در شرایطی که مسیر اجرایی دستور تغییر می‌کند، ممکن است نیاز باشد مراحل خط لوله پاک‌سازی شوند تا از اجرای نادرست دستورالعمل‌ها جلوگیری شود.

در مجموع، Hazard Unit حکم هماهنگ‌کننده‌ی مرکزی در خط لوله را دارد؛ هم از بروز خطا جلوگیری می‌کند و هم کارایی سیستم را با حداقل وقفه حفظ می‌نماید. این واحد برای پردازنده‌هایی که عملکرد بالا و اجرای موازی دارند، یک عنصر حیاتی محسوب می‌شود، همان طور که در کد زیر مشخص است برای این قسمت که در ابتدا بدون Forwarding است ما باید دوتا مقدار رو چک کنیم تا با توجه به آن ها بفهمیم که Hazard بوده است یا خیر، در کل فلسفه ی Hazard است است که چک کنیم که آیا میخواهیم از مقداری در یک رجیستر استفاده کنیم که هنوز آپدیت نشده است و در دست آپدیت است یا خیر، زیرا اگر قرار باشد که آن مقدار آپدیت شود باید از مقدار جدید استفاده کنیم که وقتی هنوز Forwarding نداریم باید با استفاده از freeze کردن و Stall کردیم به این هدف برسیم تا مقدار جدید در رجیستر نوشته شود یا بتوانیم از مقدار جدید استفاده کنیم، برای اینکه بفهمیم Hazard داریم یا خیر از کد زیر استفاده کردیم که چک میکند که آیا آدرس های یکسانی در دستورات پشت سر هم داریم یا خیر.

```
module HazardUnit(
    input EXE_WB_EN, MEM_WB_EN, Two_Src, EXE_MEM_R_EN,
    input [3:0] Src1, Src2, MEM_DEST, EXE_DEST,
    output reg hazard
);
    always @ (*) begin
        hazard = 1'b0;
        if (MEM_WB_EN) begin
            hazard = (MEM_DEST == Src1) || (Two_Src && MEM_DEST == Src2);
        end
        if (EXE_WB_EN) begin
            hazard = (EXE_DEST == Src1) || (Two_Src && EXE_DEST == Src2);
        end
    end
endmodule
```

- **طراحی Forwarding Unit**

این ماژول کمک می کند که مقدار جدید محاسبه شده برای رجیستر ها را بدون نیاز به منتظر ماندن برای نوشته شدن در Register File مستقیم از واحد MEM و یا WB به واحد EXE به جلو ارسال کنیم و دیگر نیازی به stall نخواهد بود (به جزء یک حالت که در واحد Hazard توضیح خواهیم داد.). برای این کار، اول چک می کنیم که آیا مقادیر رجیستر ها قرار است که آپدیت بشوند یا خیر. اگر `MEM_WB_EN` و یا `WB_WB_EN` فعال باشند به این معناست که نتیجه آنها قرار است که در Register File نوشته شود. در صورتی که رجیستری که می خواهیم مقدار آن را بخوانیم قرار باشد آپدیت شود، دیگر مقداری که در Register File است منقضی شده است و باید با مقدار جدید محاسبات را پیش ببریم. برای آنکه مقدار جدید را داشته باشیم، رجیستر های ورودی واحد EXE را که همان `Src1` و `Src2` می شوند با رجیستر مقصد لایه های بعدی یعنی `MEM_DEST` و `WB_DEST` مقایسه می کنیم. در صورتی که هر کدام از آنها یکسان بود باید عمل Forwarding با اولویت لایه MEM (زیرا مقدار جدیدتر را دارد.) صورت بگیرد که این عمل به کمک MUX به ازای هر دو ورودی امکان پذیر است. ورودی اول این MUX ها همان مقداری است که از Register File خوانده شده است. ورودی دوم مقدار آپدیت شده از ابتدای لایه MEM و ورودی سوم از مقدار انتخاب شده در لایه WB است.

در کد عبارت `(MEM_DEST == Src1) || (MEM_DEST == Src2)` را مشاهده می کنیم. این عبارت برای زمانی است که `MEM_WB_EN` فعال می باشد اما مقصدش از هیچ کدام از ورودی های نیست و همزمان `WB_WB_EN` نیز فعال می باشد و مقصدش همان رجیستری است که از آن خوانده ایم.
همچنین برای همه این کار ها یک شرط فعال بودن سیگنال `forwardEn` قرار می دهیم.

```
module ForwardingUnit(
    input MEM_WB_EN, WB_WB_EN, forwardEn,
    input [3:0] Src1, Src2, MEM_DEST, WB_DEST,
    output reg [1:0] sel1, sel2
);
    always @ (*) begin
        sel1 = 2'b00; sel2 = 2'b00;
        if (MEM_WB_EN && ((MEM_DEST == Src1) || (MEM_DEST == Src2))) begin
            if (MEM_DEST == Src1 && forwardEn) sel1 = 2'b01;
            if (MEM_DEST == Src2 && forwardEn) sel2 = 2'b01;
        end
        else if (WB_WB_EN) begin
            if (WB_DEST == Src1 && forwardEn) sel1 = 2'b10;
            if (WB_DEST == Src2 && forwardEn) sel2 = 2'b10;
        end
    end
endmodule
```

- **طراحی Hazard Unit با واحد Forwarding**

کد بدون Forwarding را در داخل شرط `forwardEn == 0` قرار می دهیم تا حالت بدون Forwarding را هم داشته باشیم.

اما هنگامی که Forwarding داشته باشیم، فقط در حالتی که بخواهیم از حافظه بخوانیم (یعنی دستور Load Register) باید stall کنیم زیرا مقدار جدید در انتهای لایه MEM به دست می آید و ما نمی توانیم آن را به جلو ارسال کنیم چون در این صورت یک مسیر combinational طولانی از ابتدای لایه MEM تا انتهای آن و سپس از ابتدای لایه EXE تا انتهای آن به وجود می آید که موجب افزایش طول حداقل clock می شود که مطلوب نیست. بنابراین یک clock صبر می کنیم و مقدار جدید را از لایه WB ارسال می کنیم (دلیل آنکه در این حالت دیگر مشکل مسیر combinational طولانی را نداریم آن است که لایه WB فقط یک MUX دوتایی دارد که تاخیر زیادی اضافه نمی کند.).

برای ایجاد Hazard قصد داریم دستوری که به مقدار جدید از حافظه نیاز دارد را در لایه ID نگه داریم و یک دستور No Operation قبل آن اضافه کنیم. برای این کار هنگامی که دستور در ID است و دستور بعدی قصد آپدیت کردن رجیستر با خواندن مقدار از حافظه را دارد یعنی دو سیگنال `EXE_WB_EN` و `EXE_MEM_R_EN` همزمان فعال هستند در صورتی که رجیستر مقصد دستور بعدی با هر کدام از رجیستر های مبدا دستور فعلی یکسان باشد باید سیگنال Hazard فعال شود تا پردازنده stall شود.

```
module HazardUnit(
    input EXE_WB_EN, MEM_WB_EN, Two_Src, forwardEn, EXE_MEM_R_EN,
    input [3:0] Src1, Src2, MEM_DEST, EXE_DEST,
```

```
    output reg hazard
);
    always @ (*) begin
        hazard = 1'b0;
        if (forwardEn == 0) begin
            if (MEM_WB_EN) begin
                hazard = (MEM_DEST == Src1) || (Two_Src && MEM_DEST == Src2);
            end
            if (EXE_WB_EN) begin
                hazard = (EXE_DEST == Src1) || (Two_Src && EXE_DEST == Src2);
            end
        end
        else begin
            if (EXE_WB_EN && EXE_MEM_R_EN) begin
                hazard = (EXE_DEST == Src1) || (Two_Src && EXE_DEST == Src2);
            end
        end
    end
endmodule
```

مشکلات احتمالی

- **بدون واحد Hazard و Forward**

اگر برنامه‌ای وارد پردازنده شود که وابستگی داده‌ای بین دستورهای متوالی داشته باشیم، نتیجه غلط خواهد بود. سناریوی زیر را در نظر بگیرید:

```
ADD R1, R2, R3
ADD R4, R1, R2
```

وقتی دستور دوم به استیج ID می‌رسد محتوای `R1` و `R2` را از Register File می‌خواند در حالی که مقدار جدید R1 هنوز در Register File نوشته نشده است زیرا دستور قبلی هنوز در استیج EXE قرار دارد و دو سیکل بعد به استیج WB می‌رسد.

- **با واحد Hazard، اما بدون Forward**

در این حالت پردازنده از لحاظ عملکردی درست است، اما به علت احتیاط بیش از حد برای اطمینان از درستی عملکرد، سیکل‌های زیادی را در حالت استال می‌گذراند. در واقع همواره صبر می‌کند تا دیتای جدید به استیج WB

برسد و نوشته شود. در حالی که دیتای مدنظر احتمالاً جایی در یکی از استیج‌های قبلی آماده است و با Forward کردن سریع‌تر کار راه می‌افتد.

- **با واحد Hazard و Forward**

هنوز در دستوری مانند load word نیاز به استال دارد اما کمتر از حالت قبلی. سناریوی زیر را در نظر بگیرید:

```
LDR R1, [R2]
ADD R3, R1, R4
```

وقتی دستور ADD به استیج ID می‌رسد، هنوز محتوای R1 در Register File نوشته نشده است زیرا دستور LDR در استیج EXE است و اولین زمانی که دیتا آماده است استیج MEM می‌باشد. پس یک سایکل پردازنده استال می‌شود. در مقایسه با دو سایکل حالت قبل، پیشرفت است.

مشکل اصلی این مدل، پیچیدگی طراحی آن است که منجر به میزان استفاده از سخت‌افزار بیشتری می‌شود.

نتیجه اجرای پردازنده

برای تست، پردازنده را به روی کد باینری برنامه اسمبلی زیر اجرا می‌کنیم که توضیح آن بعد از کد داده شده است.

```
1.  MOV R0, #20
2.  MOV R1, #4096
3.  MOV R2, #0xC0000000
4.  ADDS R3, R2, R2
5.  ADC R4, R0, R0
6.  SUB R5, R4, R4, LSL #2
7.  SBC R6, R0, R0, LSR #1
8.  ORR R7, R5, R2, ASR #2
9.  AND R8, R7, R3
10. MVN R9, R6
11. EOR R10, R4, R5
12. CMP R8, R6
13. ADDNE R1, R1, R1
14. TST R9, R8
15. ADDEQ R2, R2, R2
16. MOV R0, #1024
17. STR R1, [R0], #0
18. LDR R11, [R0], #0
19. STR R2, [R0], #4
20. STR R3, [R0], #8
21. STR R4, [R0], #13
22. STR R5, [R0], #16
```

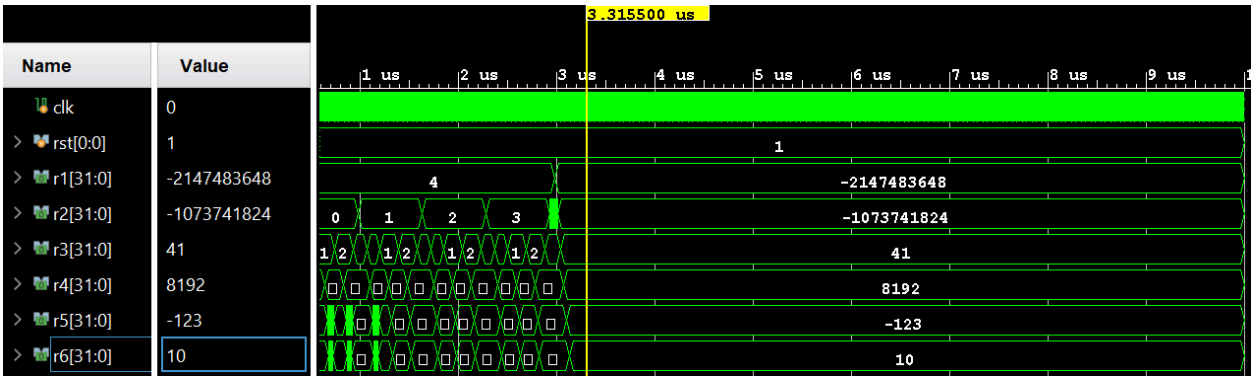
```
23. STR R6, [R0], #20
24. LDR R10, [R0], #4
25. STR R7, [R0], #24
26. MOV R1, #4
27. MOV R2, #0
28. MOV R3, #0
29. ADD R4, R0, R3, LSL #2
30. LDR R5, [R4], #0
31. LDR R6, [R4], #4
32. CMP R5, R6
33. STRGT R6, [R4], #0
34. STRGT R5, [R4], #4
35. ADD R3, R3, #1
36. CMP R3, #3
37. BLT #-9
38. ADD R2, R2, #1
39. CMP R2, R1
40. BLT #-13
41. LDR R1, [R0], #0
42. LDR R2, [R0], #4
43. LDR R3, [R0], #8
44. LDR R4, [R0], #12
45. LDR R5, [R0], #16
46. LDR R6, [R0], #20
47. B #-1
```

25 خط اول برنامه مجموعه‌ای از دستورات است که برای تست جداگانه و اطمینان از صحت عملکرد هر یک از دستورالعمل‌های پردازنده طراحی شده است. در این بخش، مقادیر مشخصی در رجیسترها بارگذاری شده و نتایج عملیات مختلف در حافظه ذخیره می‌شود.

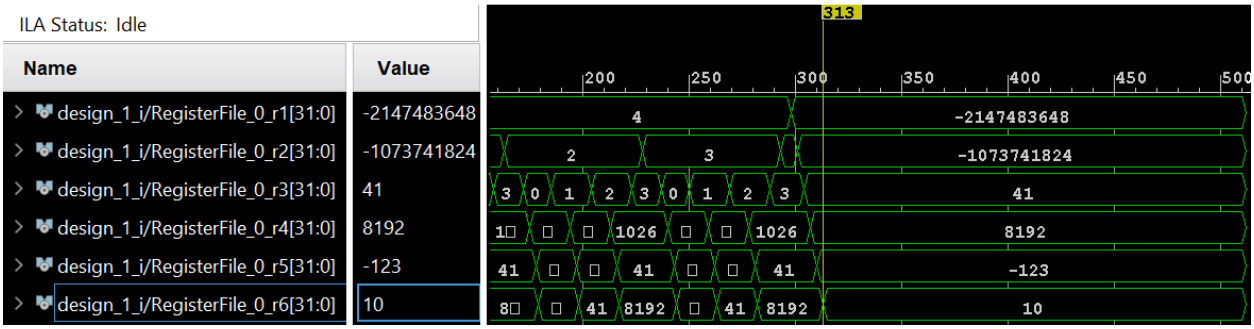
22 خط بعدی برنامه (26 تا 47) یک الگوریتم bubble sort را پیاده‌سازی می‌کند. این الگوریتم چند عدد را که در مرحله قبل در حافظه ذخیره شده بودند، به صورت صعودی مرتب می‌کند. هدف اصلی چک کردن صحت عملکرد دستورات branch مورد استفاده برای loop است.

- **پردازنده بدون واحد Forwarding**

نتیجه Simulation

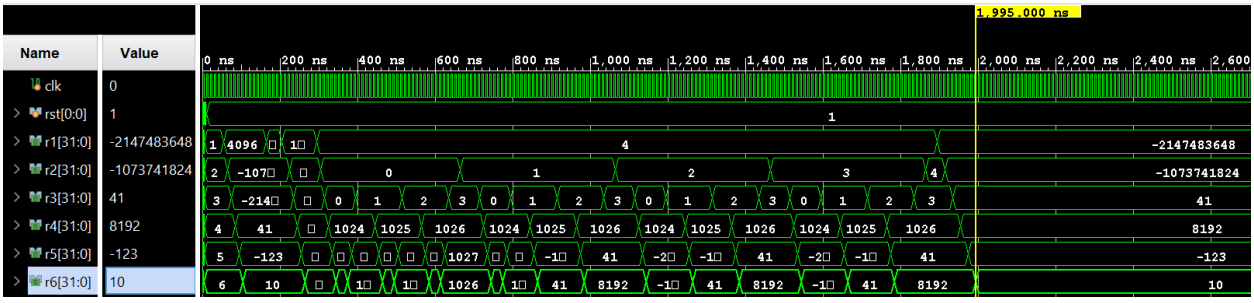


نتیجه Synthesis



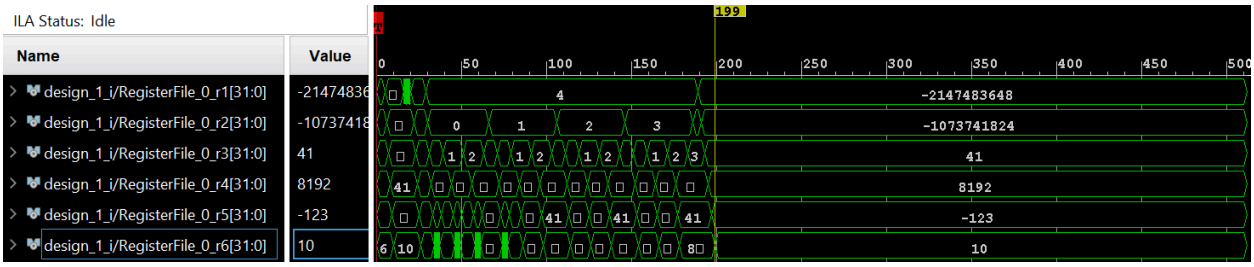
- **پردازنده با واحد Forwarding**

نتیجه Simulation



نتیجه Synthesis

ابتدا Waveform را می‌بینیم.



حال در فرم دو پنجره‌ای نتیجه‌ی هر دو حالت با و بدون واحد Forwarding را کنار هم می‌بینیم.

سنجش عملکرد پردازنده

در این بخش از معیارهای مختلفی برای سنجش پردازنده در هر دو حالت با و بدون Forwarding کمک می‌گیریم.

- **تعداد سیکل‌های اجرا**

همانطور که در تصاویر بخش "نتیجه اجرای پردازنده" مشخص است، بدون واحد Forwarding نیاز به 313 و با آن نیاز به 199 سیکل کلاک برای اتمام برنامه تست است. پردازنده روی این برنامه تست خاص بیش از 35 درصد بهبود عملکرد داشته است.

- **تعداد سیکل به ازای هر دستور – Clock per Instruction**

$$CPI = \frac{\#\ Clock\ Cycles}{\#\ Instructions}$$

با استفاده از اعداد بخش قبل و توجه به اینکه کل برنامه تست شامل 47 دستور بود، می‌توان گفت شاخص CPI بدون Forwarding برابر $\frac{313}{47} = 6.66 \simeq 7$ و با آن برابر $\frac{199}{47} = 4.23 \simeq 4$ است. این معیار هم معادل همان معیار قبل است که پیشرفت بیش از 35 درصدی را نشان می‌دهد.

- **میزان سخت‌افزار مورد استفاده**

Block Ram	F8 Muxes	F7 Muxes	Registers	LUTs	
5.5	193	455	2598	2915	بدون Forwarding
17.5	194	560	4181	3726	با Forwarding
3.18	1.00	1.23	1.61	1.27	نسبت به هم

- **کارایی بر هزینه - Performance per Cost**

برای محاسبه این بخش، به عنوان معیار ارزیابی کارایی، معکوس CPI و برای معیار ارزیابی هزینه، تعداد Lut ها را در نظر می‌گیریم. علت انتخاب این است که هرچه CPI کمتر باشد (معکوس CPI بیشتر باشد) پردازنده سریع‌تر است. همچنین هرچه تعداد LUTs -که بلاک اصلی پردازنده سنتز شده‌ست- بیشتر باشد هزینه‌ی ساخت آن بیشتر است.

بار دیگر این دو معیار را می‌بینیم:

معیار ارزیابی	بدون Forwarding	با Forwarding
Clock per Instruction	6.66	4.23
Number of LUTs	2915	3726

$$\text{Performance per Cost} = \frac{\frac{1}{4.23\times3726}}{\frac{1}{6.66\times2915}} = \frac{6.66\times2915}{4.23\times3726} \simeq 1.23$$

پس با معیار تعریف شده پردازنده در حالت دوم تقریباً 23 درصد بهتر از حالت اول عمل کرده است.