گزارش فاز اول پروژه‌ی معماری

رستا روغنی	۹۷۱۰۵۹۶۳	یگانه قره‌داغی	۹۷۱۰۶۲۱۶
مهرانه نجفی	۹۷۱۰۴۷۰۷	سیده صبا هاشمی	۹۷۱۰۰۵۸۱

طراحی مسیر داده‌ی این پردازنده مشابه پردازنده‌ی میپس می‌باشد.

در ابتدا بلاک‌های Register File، ALU و حافظه‌ها را طراحی کردیم که هر کدام به اختصار در زیر توضیح داده می‌شوند:

Register File:

وظیفه‌ی بخش RF یا Register file ، خواندن یا نوشتن در رجیسترها می‌باشد.

در Registers، سی و دو رجیستر ۳۲ بیتی داریم که کلاک و ریست‌شان (که آسنکرون است) از پردازنده گرفته می‌شود. و هرکدام یک سیگنال enable دارند که در زمانی که بخواهیم درون رجیستری بنویسیم، این سیگنال را برای آن فعال میکنیم.

در WriteandReadRegisters دو بخش داریم. بخش اول بدین صورت است که شماره ی رجیسترها را ورودی می‌گیرد (Read_Register_1 و Read_Register_2) و محتوای رجیسترهای مربوط را در خروجی (به ترتیب Read_Data_1 و Read_Data_2) قرار می‌دهد. برای این کار از دو mux که ورودی‌های آن‌ها مقادیر رجیسترها هستند استفاده شده است. درصورتی که سیگنال Write فعال باشد و بخواهیم در رجیستر مقداری بریزیم، سیگنال Write_Register شماره‌ی رجیستر مورد نظر را به یک دیکودر می‌دهد که سیگنال enable آن را فعال کرده و داده ی مدنظر که از طریق Write_Data داده می‌شود را درون رجیستر قرار می‌دهد.

فایل‌های مربوط به این بخش در فولدر RF قرار دارند.

ALU:

واحد ALU نتیجه‌ی مجموعه دستورالعمل های محاسباتی و منطقی و دستورات با عملوند صریح (و همچنین محاسبات جمع آدرس برای دسترسی به حافظه) را تعیین می‌کند. مقادیر ۳۲ بیتی Read_Data_1 ، Read_Data_2 و ۶ بیتی opcode به عنوان ورودی به ALU داده میشوند. در دستور های محاسباتی، مقادیر رجیسترهای مبدا در ورودی‌ها ریخته شده‌اند و در دستورهای با عملوند صریح در Read_Data_2 عدد صریح (immediate) به صورت extend شده (۳۲ بیتی) داده شده است. به همین دلیل برای محاسبات هر دو نوع دستور از یک سخت‌افزار استفاده می‌شود.

برای محاسبات از بلاک‌های آماده‌ی ویزارد استفاده شده است. برای دستورهای جمع و تفریق، ماکسیمم و مینیمم، شیفت‌ها و rotateهای راست و چپ سیگنال‌های کنترلی از opcodeها تعیین می‌شوند.

در نهایت در صورتی که دستورها مربوط به load کردن عدد صریح بودند، برای دستور LUI در Read_Data_1 مقدار رجیستر مقصد ریخته میشود و مقدار بالای رجیستر را immediate تشکیل داده و قسمت پایین تغییر نمیکند.
اما در LDI، در کل رجیستر مقصد، مقدار immediate قرار می‌گیرد. (یعنی به صورت sign extend)
در نهایت در ورودی mux نهایی تمامی مقدار های ممکن برای خروجی به ترتیب ۵ رقم پایین opcode قرار گرفته‌اند. مکان ۰ ام mux نیز برای به‌دست آوردن آدرس نهایی از offset و address reg از دستورات دسترسی به حافظه نگه داشته شده است. سپس طبق opcode پاسخ قرار گرفته شده در mux به عنوان result انتخاب شده و خروجی alu تعیین می‌شود.
فایل‌های مربوط به این بخش در فولدر ALU قرار دارند.

Memory:

بلاک حافظه برای این که بتوان هم در آن بایت و هم کلمه ذخیره کرد یا از آن خواند از چهار حافظه‌ی RAM تشکیل شده که کلمات آن ۸ بیتی هستند. در صورتی که بخواهیم بایت بخوانیم یا بنویسیم با استفاده از دو بیت کوچک‌تر آدرس بین این چهار حافظه حافظه‌ی مورد نظر را انتخاب می‌کنیم و سیگنال enable خواندن (rden) یا نوشتن (wren) را فقط برای آن فعال می‌کنیم. در این صورت هنگام نوشتن از دیتای ورودی ۸ بیت کم‌ارزش‌تر نوشته می‌شود و هنگام خواندن ۸ بیت sign extend شده و در خروجی داده می‌شود.
در صورتی که بخواهیم کلمه بخوانیم یا بنویسیم سیگنال‌های مربوط به چهار حافظه فعال می‌شوند. برای همین کلمات برای درست نوشته شدن در حافظه باید مضرب چهار باشند.

Instruction Memory:

از خانه‌های ۳۲ بیتی تشکیل شده است و کد ماشین برنامه‌هایی که زدیم را با استفاده از فایل mif. درون آن قرار داده و اجرا می‌کنیم.
فایل‌های دو بخش اخیر به نام‌های memory_asli2 و InstructionMemory در فولدر memory قرار دارند.

Control Unit:

واحد کنترل به زبان وریلاگ زده شده است و با گرفتن آپکد، سیگنال‌های command مربوط به انتخاب داده قبل از Register File و ALU، سیگنال‌های مربوط به پرش و سیگنال‌های مربوط به نوشتن در حافظه و رجیستر فایل را تولید می‌کند. هم چنین دو سیگنال مربوط به کمک پردازنده هم تولید می‌کند که در گزارش فاز ۲ توضیح داده شده است.
فایل‌های مربوط به سیگنال‌های کنترل در فولدر ControlUnit قرار دارد.

DataPath:

پس از طراحی بلاک‌های رجیستر فایل، ALU و حافظه‌ها آن‌ها را در DataPath به هم وصل کردیم.

فلوی کلی داده به این صورت است:
یک رجیستر PC داریم که آدرس دستوری که باید اجرا شود را در خود دارد و به ورودی InstructionMemory متصل است.
خروجی این حافظه دستور است که ۶ بیت اول آن به عنوان opcode به کنترل یونیت داده می‌شود تا سیگنال‌های کنترل را تولید کند. بیت ۱۶ تا ۲۰ به عنوان آدرس رجیستر اول به رجیستر فایل داده می‌شود. برای ورودی دوم رجیستر فایل یک mux گذاشته شده که یا بیت ۲۱ تا ۲۵ و یا ۱۱ تا ۱۵ را به آن بدهد چون برای بعضی از دستورات به یکی و برای بعضی به دیگری نیاز داریم. ۱۶ بیت کم‌ارزش آن نیز ساین اکستند شده و به عنوان immediate استفاده می‌شود. البته در دستورهای شیفت و rotate بیت‌های ۶ تا ۱۰ آن برای مقدار شیفت یا rotate در ALU استفاده می‌شوند.

بعد از این مرحله نوبت ALU است. قبل از دو ورودی ALU دو mux قرار داده شده تا با توجه به دستور بین Data1 و Data2 که مقادیر رجیسترها هستند که از رجیستر فایل آمده‌اند و داده‌ی immediate دو مقداری که به آن نیاز دارد به آن داده شود.

خروجی ALU به سمت حافظه می‌رود و در صورت فعال بودن سیگنال write حافظه که با توجه به opcode مشخص می‌شود آدرس خانه‌ی حافظه را مشخص می‌کند. مقداری که باید در حافظه نوشته شود هم از خروجی‌های رجیستر فایل گرفته می‌شود.

حال اگر لازم به نوشتن در رجیستر باشد یعنی write رجیستر فایل فعال باشد لازم است با توجه به آپکد یکی از مقادیر خروجی ALU و خروجی حافظه انتخاب شود که به این منظور یک mux قبل از ورودی write_data رجیستر فایل قرار می‌دهیم.

برای پیاده‌سازی دستورهای پرش قبل از رجیستر PC یک mux چهار ورودی قرار می‌دهیم که هر کدام از ورودی‌های آن یک حالت از دستورات پرش هستند. سلکت این PC با توجه به opcode توسط کنترل یونیت و با استفاده از سیگنال branch خروجی از ALU برای دو دستور پرش شرطی معین می‌شود.
بلاک دیتاپث در فولدر Datapath قرار دارد و بلاک کنترل یونیت در همین فایل به Datapath متصل شده است.
این بلاک که در واقع کل پردازنده‌ی ما است در فایل device در فولدری به همین نام به کمک‌پردازنده متصل شده است.

<u>پایپ لاین:</u>
برای پایپ‌لاین کردن پردازنده این مراحل را می‌توان در ۵ مرحله در نظر گرفت:

IF: Instruction Fetch
ID: Instruction Decode and Register read (RF)
EX: Execution or Calculate Address (ALU)

MEM: Memory Access
WB: Write back

بین داده‌های هر مرحله رجیستر قرار می‌دهیم. همچنین کامندهایی که کنترل یونیت در مرحله‌ی ID تولید می‌کند را نیز رجیستر می‌کنیم تا در هر کلاک داده‌ها و کامندهای مربوط به یک دستور در یکی از این مراحل باشند.

مخاطرات:
مخاطراتی که در این صورت در پردازنده رخ می‌دهد دو نوع خواهد بود؛ یکی برای دستورات پرش چرا که سیگنال branch توسط ALU تولید می‌شود و در صورت پرش لازم نیست دستورات بالافاصله بعد از این دستور اجرا شوند. دیگری نیز مشکل data hazards است که به این علت رخ می‌دهد که ممکن است دستوری مقدار رجیستری را نیاز داشته باشد که توسط دستورات قبل از آن که هنوز داخل پایپ‌لاین هستند و به اتمام نرسیده‌اند تغییر کند.

برای رفع مشکل اول یک hazard unit در پردازنده قرار دادیم. این واحد به این صورت عمل می‌کند که در مرحله‌ی ID آپکد را گرفته و اگر دستور پرش باشد enable رجیستر PC را که الآن آدرس دستور بعدی را دارد غیر فعال می‌کند تا مقدار آن تغییر نکند. همچنین مقدار خروجی instruction register را نیز flush می‌کند و به عنوان دستور مرحله‌ی بعد یک no operation می‌دهد. در کلاک بعدی دستور پرش در مرحله‌ی EX است و در مرحله‌ی ID یک no op قرار دارد. هم چنین در IF دستور بعد از دستور پرش قرار دارد. در این مرحله مشخص می‌شود که پرش انجام می‌شود یا خیر. واحد hazard، دستور مرحله‌ی EX را می‌گیرد و در صورتی که دستور پرش باشد و پرش انجام شده باشد یعنی سیگنال‌های سلکت مالتی‌پلکسر قبل از PC مقداری غیر از PC + 1 را به عنوان ورودی انتخاب کرده باشند دستور فعلی در مرحله‌ی IF را فلاش می‌کند و به جای آن یک no op به مرحله‌ی ID می‌فرستند چون دیگر اجرای آن لازم نیست.

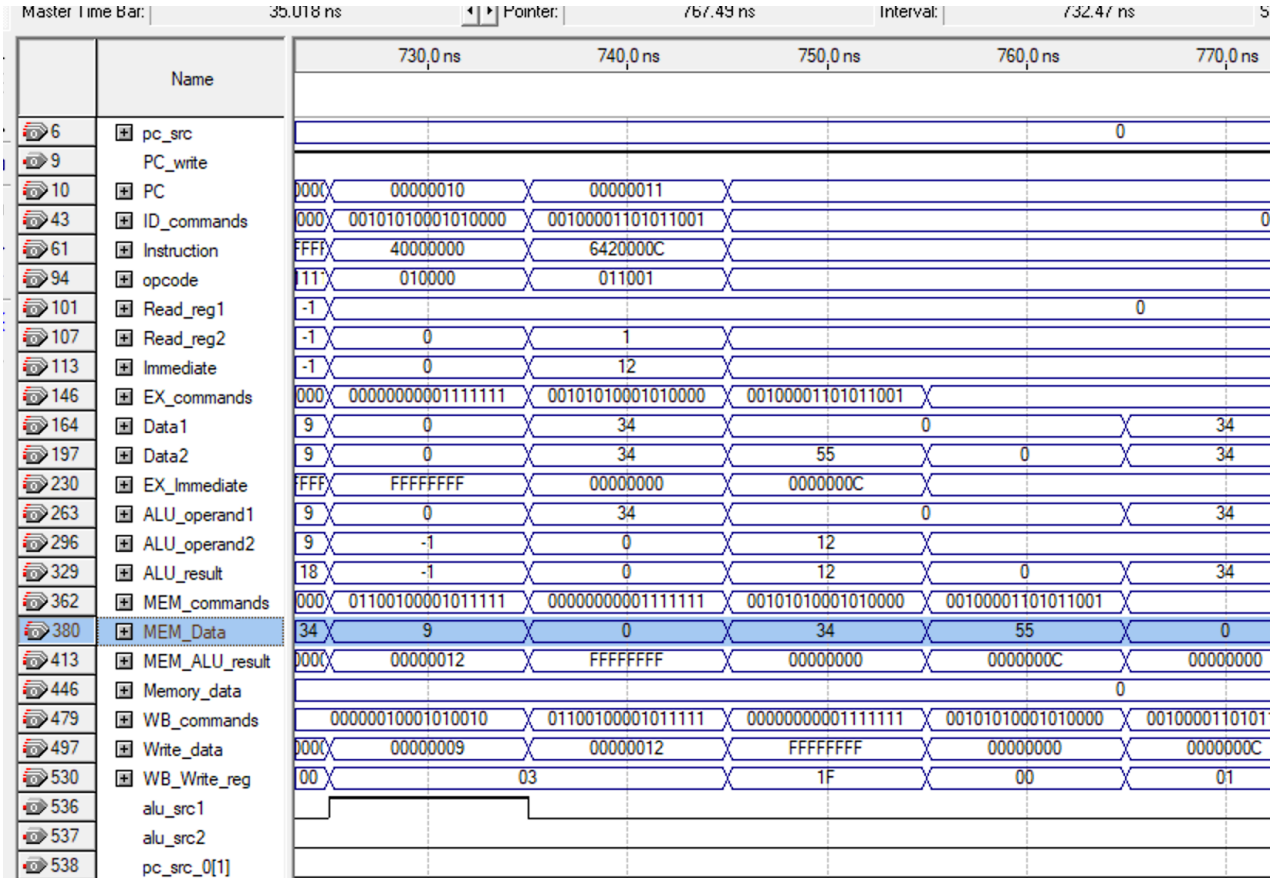
برای رفع data hazards یک forwarding unit قرار می‌دهیم. این واحد در بخش EX قرار دارد و وظیفه‌اش این است که اگر مقادیر رجیسترهای ورودی ALU هنوز در رجیسترهای مربوطه نوشته نشده‌اند مقادیر درست آن ها را از جایی که مقدار درست آن را دارد بگیرد. مقدار درست این رجیسترها یا در مرحله‌ی MEM و یا در مرحله‌ی WB است. به همین منظور forwarding unit شماره‌ی دو رجیستری که از رجیستر فایل خوانده شده اند را می‌گیرد. همچنین شماره ی رجیستری را که دستورهای مرحله‌ی MEM و WB می‌خواهند در آن بنویسند (MEM_Write_reg و WB_Write_reg) و commandهای مربوط به این دو مرحله را می‌گیرد. در کامندهای مربوط به هر مرحله یک بیت وجود دارد که بیت register write است و اگر دستور در مرحله‌ی WB در رجیستری می‌نویسد آن فعال می‌شود. حال اگر این بیت برای یکی از دو مرحله‌ی WB یا MEM فعال باشد و رجیستری از رجیسترهای خوانده شده با write reg آن مرحله یکی باشد forwarding unit مقداری که در رجیستر نوشته خواهد شد را به جای داده‌ای که از رجیستر فایل آمده به ALU می‌دهد. برای

انتخاب داده‌ی ورودی به ALU هم از دو عدد ماکس استفاده شده است و forwarding unit در واقع مقدار select این مالتی‌پلکسرها را تعیین می‌کند.
همچنین لازم است که عمل نوشتن در رجیستر فایل یا حافظه در لبه‌ی پایین‌رونده‌ی کلاک انجام شود.
این دو واحد نیز در کنار واحد کنترل در فایل ControlUnit قرار دارند.

تست:

برای تست کردن دستورات برنامه‌هایی به کد ماشین شامل همه‌ی دستورات نوشتیم و خروجی مراحل مختلف را بررسی کردیم. فایل mif. تست‌ها و توضیحات هر کدام در فایل Memory/testFiles موجود است.
پس از اطمینان از درستی همه‌ی دستورات خروجی‌های اضافی را حذف کردیم و تنها از مقدار ورودی و خروجی به حافظه برای تست استفاده کردیم.

در زیر نمایی از waveform یکی از تست‌ها را می‌بینید که شامل سیگنال‌های مراحل مختلف است:



نتیجه‌ی اجرای فیبوناچی:

برنامه‌ی نوشته شده برای فیبوناچی و کد ماشین آن به صورت فایل mif. نیز در Memory/testFiles موجود است.

در این برنامه رجیسترهای ۰ و ۱ در ابتدا به ترتیب ۰ و ۱ مقداردهی می‌شوند. رجیستر شماره‌ی ۲ مقدار n را نگه‌داری می‌کند و از رجیستر ۳ به عنوان counter استفاده می‌شود. در نهایت جواب در رجیستر ۱ قرار دارد که در خانه‌ی ۱۲ حافظه ریخته می‌شود.

تعداد دستورات برنامه ۱۳تاست که ۸تا از آن‌ها فقط یک بار اجرا می‌شوند. ۵ دستور دیگر دستورات داخل لوپ هستند. چهارتا از آن‌ها هر کدام یک کلاک می‌گیرند و n - 1 بار اجرا می‌شوند. دستور پنجم دستور پرش است و دستورات پرش اگر پرش انجام نشود ۲ کلاک و اگر انجام شود ۳ کلاک طول می‌کشند. دستورات دیگر در صورت پر بودن پایپ‌لاین هر کدام یک کلاک طول می‌کشند. اما چون در ابتدا پایپ‌لاین خالی است ۴ کلاک اضافی برای پر شدن آن لازم است.

در نتیجه تعداد کل کلاک‌های برنامه اگر فیبوناچی nام را بخواهیم برابر خواهد بود با:

$$4 + 8 + 4 * (n - 1) + 3 * (n - 2) + 2$$

$$= 4 + 7 * n$$

یعنی برای محاسبه‌ی فیبوناچی nام تقریبا به 7n کلاک نیاز داریم.

تعداد کل دستوراتی که اجرا می‌شود هم برابر است با:

$$8 + 5 * (n - 1) = 5 * n + 3$$

پس CPI این برنامه برابر 1.4 = 7/5 خواهد بود.

در زیر مقدار ذخیره شده در حافظه را برای n = 8 مشاهده می‌کنید: