معماری کامپیوتر

گزارش فاز اول پروژه

استاد سربازی

امیرمهدی کوششی

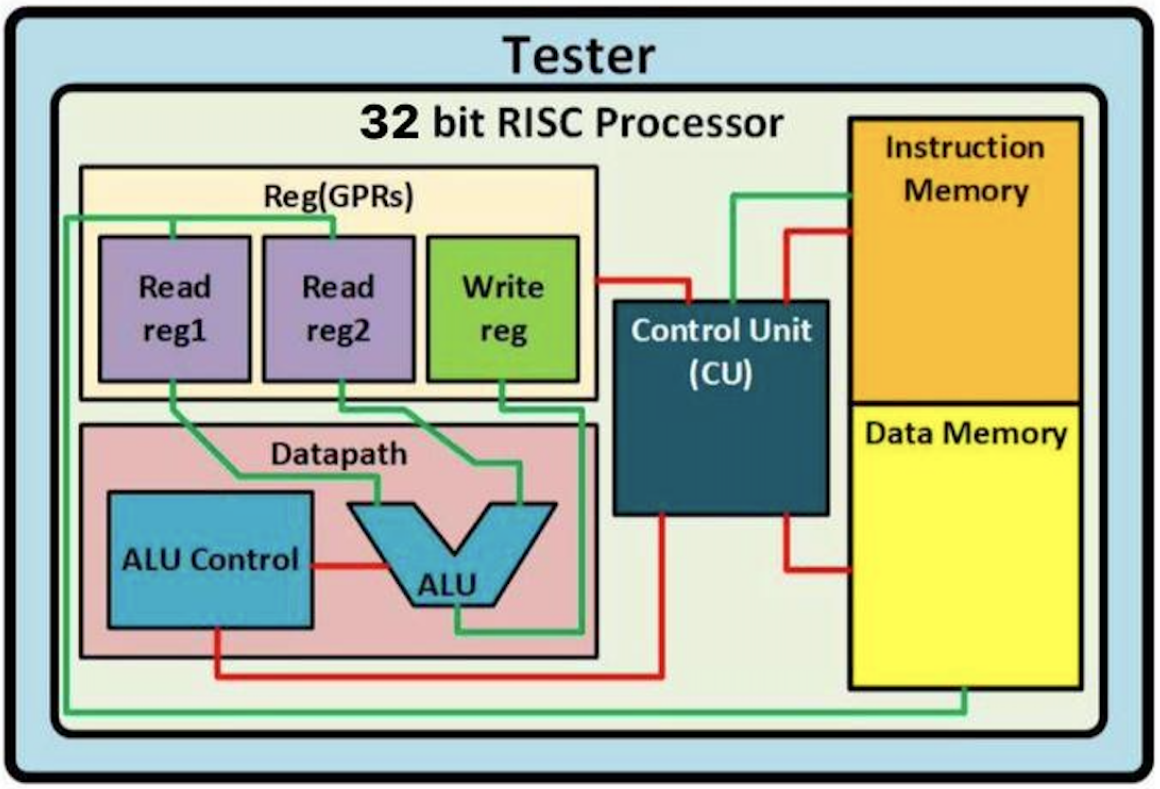
محمد صادق مجیدی

احسان موفق

پویا اسمعیلی آخوندی

طراحی

ساختار کلی ماشین میپس طراحی شده در پروژه ما مانند شکل زیر است که تفاوت‌هایی با پیاده‌سازی مدار ما دارد. در ادامه به توضیح هر یک و توضیح تفاوت ها خواهیم پرداخت.



توضیح مدار بالا:

دستورات وارد شده به کمک اینستراکشن

همانطور که در توضیحات داک پروژه نیز آمده بود، instruction memory و data memory به صورت آماده در اختیار ما قرار داده شده بود. طرز استفاده از آن‌ها برای پروژه ما نیز دقیقا مانند شکل بالا است.

تمامی کارهای مورد نیاز با registerها را (خواندن و نوشتن) از طریق یک instance از ماژول regfile کنترل میکنیم.

وظیفه نوشتن و خواندن اطلاعات در رجیسترها وظیفه Control Unit است. در واقع control unit پیشنیازهای مورد نیاز ALU را انجام میدهد. دیتاهایی که نیاز دارد تا روی آن‌ها عملیات انجام دهد را در اختیارش قرار داده. و نیز result نهایی را از ALU گرفته و در صورت نیاز در رجیسترها ذخیره می‌کند. Control Unit همچنین وظیفه ذخیره سازی و خواندن اطلاعات از memory را نیز دارد. به صورت کلی Control Unit هسته مرکزی مدار ما می‌باشد که وظایف

مختلف را بین ماژول‌های مختلف پخش کرده و آن‌ها را مدیریت می‌کند: چه از خواندن و نوشتن داده‌ها از رجیسترها و موری گرفته چه از مشخص کردن عملیات دستورها و اینکه چه خروجی‌ای را باید از ALU دریافت کرد.

یکی از تفاوت ها در همینجا است. در پروژه ما ALU Control در واقع در همان control unit قرار دارد، و ارتباط ما با datapath و ALU توسط همان Control Unit انجام می‌شود و ما ماژول جداگانه ای برای ALU Control در نظر نگرفته ایم. سیگنال های مورد نیاز ما، مانند عملیات (operation) ای که در ALU قرار است صورت بگیرد و همچنین سیگنال‌های مورد نیاز برای عملیات رجیسترها را هم نیز فراهم میکند.

یکی از مشکلاتی که در این حین برای ما ایجاد شد ایجاد latch هایی بود که به دلیل انجام همه عملیات ها بر posedge clock بود. ما در ابتدا کد را صورت کامل non-blocking در یک always زدیم و همین باعث ایجاد latch میشد. سپس کدمان را درست کردیم.

برای درست کردن آن ما ترکیبی از non-blocking و blocking در دو always متفاوت رفتیم.

```
always @(posedge clk) begin
    if(!rst_b) begin
        halted_signal <= 1'b0;
    end
    else begin
        halted_signal <= tmp_halted_signal;
```

```
always @(*) begin
    // R type
    tmp_pc_j_en = 1'b0;
    tmp_pc_branch_en = 1'b0;
    if (opcode == 6'b0) begin
        case(func)
```

در واقع با این عمل، ما همه دیتاهایی که نیاز داشتیم از instance های ماژول هایمان به دست بیاوریم و به کارکردن با آن ها بپردازیم را از وابستگی به کلاک جدا میکردیم. و زمانی که نیاز بود دیتاهای مورد نیاز سایر ماژول ها به آن داده شود، و یا خروجی ای ست گردد (مثل halted) با کلاک خوردن به صورت non-blocking و parralel دیتا ها بر روی سیم ها قرار میگرفتند.

همچنین در دستور های sw lw beq bne j مشکلات زیادی را داشتیم و وقت زیادی بابت دیباگ آن ها گذاشیم. دلیل اصلی مشکلات آن‌ها برای دستورات beq bne j تغییر PC بود. و مشکلات sw lw برای آدرس دهی های مموری بود. مشکلات ما به بیان ساده time delay هایی بود که داشتیم. در دستورات برنچینگ که pc مقدار کاملا جدیدی میگرفت، دوباره دچار مشکل latch میشدیم. زیرا مقادیر جدیدی که قرار بود با مقدار فعلی pc جمع شوند از CU به mips core میرفتند(تغییرات PC در mips core بود).

برای کار با memory نیز هنگامی که memory_write_en و mem_adr هارا ست میکردیم، به دلیل اینکه در memory نیز این اعمال با عوض شدن کلاک ست میشدند، کد ما دچار latch میشد.

** درنهایت همه این ارور ها درست شدند و همه تست ها پاس شدند.

روند کلی طراحی به شرح زیر می‌باشد:

پس از مشخص شدن دستور به کمک instrauction_memory داده‌های دستور به واحد کنترل‌کننده منتقل می‌شوند.

واحد کنترل‌کننده مشخص می‌کند دستور داده شده چه عملیاتی را انجام خواهد داد همچنین دیتای مورد نیاز برای پردازش را برای ALU فراهم می‌کند. دیتای فراهم شده یا از رجیسترها و یا از خود دستور که یا آدرس حافظه را مشخص می‌کند یا داده‌ای فوری است، به دست می‌اید. داده‌ها اولیه را به ALU داده و نتیجه به دست آمده از ALU را در رجیستر مورد نظر یا حافظه ذخیره می‌کند. برای دستوراتی که نیازی به ALU ندارند از خود CU استفاده میکنیم. صرفا CU مشخص می‌کند چه دستوری باید اجرا شود و پس از آن متغیرهایی را از خود خارج می‌کند که باعث تغییر مموری یا PC می‌شود.