گزارش پروژهٔ معماری کامپیوتر

اعضای گروه:

یاشار ظروفچی بنیسی **۹۷۱۰۶۱۱۹**

کسری عبدالله سروی **۹۷۱۰۶۱۲۱**

آتوسا چگینی **۹۷۱۰۶۲۵۱**

بهار ۹۹

طراحی ALU:

در طراحی این ماژول، سیگنال‌های کنترلی همان ۳ بیت کم‌ارزش opcode در نظر گرفته شده‌اند.

برای عملیات‌های مختلف بخش‌های زیر در نظر گرفته شده‌اند:

۱. مقایسه‌گر (طراحی شده)
۲. شیفت دهنده (طراحی شده)
۳. ماژول جمع/تفریق (lpm_add_sub)
۴. کامپوننت Nand (طراحی شده)

توضیح بخش‌ها:

۱. **مقایسه‌گر**:

دو ورودی را می‌گیرد و ورودی کوچکتر را به همراه دو سیگنال eq (به شرط برابری) و slt (به شرط کوچکتر بودن ورودی اول از دوم) خروجی می‌دهد. در طراحی این مقایسه‌گر از ماژول lpm_compare استفاده شده است.

۲. **شیفت‌دهنده**:

منطق طراحی این شیفت‌دهنده در ابتدا این بوده که به کمک یک Mux یکی از ۶۴ حالت مدهای شیفت‌دادن انتخاب شود (شیفت از ۰ تا ۳۱ هم به چپ و هم به راست). در نتیجه یک l_extender32 طراحی شد که قرار بود یک رشته بیت ۳۲ بیتی را به چپ شیفت دهد (برای شیفت به راست وارون همین رشته به چپ شیفت داده می‌شود و مجدد وارونه می‌شود)؛ اما به علت وجود محدودیت در تعداد ورودی و خروجی‌ها، یک l_extender16 طراحی شد. پس اتفاقی که افتاد این بود که رشته به دو قسمت تقسیم می‌شد. هر تکه راست و چپ رشته شیفت داده می‌شوند (در قالب رشته‌های ۳۲ بیتی که برای قسمت سمت راست ۱۶ بیت سمت چپ و در قسمت سمت چپ، ۱۶ بیت سمت راست صفر هستند). سپس اگر مقدار شیفت از ۱۶ کمتر بود دو رشته ۳۲ بیتی xor می‌شوند و اگر بیشتر یا مساوی بود نسخه شیفت‌داده شدهٔ سمت راست به تنهایی خروجی می‌شود (چون تمامی ۱۶ بیت سمت چپ عملا از رشته خارج شده‌اند).

۳. **ماژول جمع/تفریق:**

دو ورودی a و b و cin دارد که مشخص هستند و با سیگنال ورودی add_sub می‌توان تعیین کرد که جمع یا تفریق (به ترتیب با ۱ و ۰) انجام شود. همچنین در عملیات تفریق لازم است که مقدار cin ۱ باشد.

۴. **کامپوننت Nand:**

تعدادی ماژول nand2 هستند برای بیت‌های مختلف دو رشتهٔ دودویی.

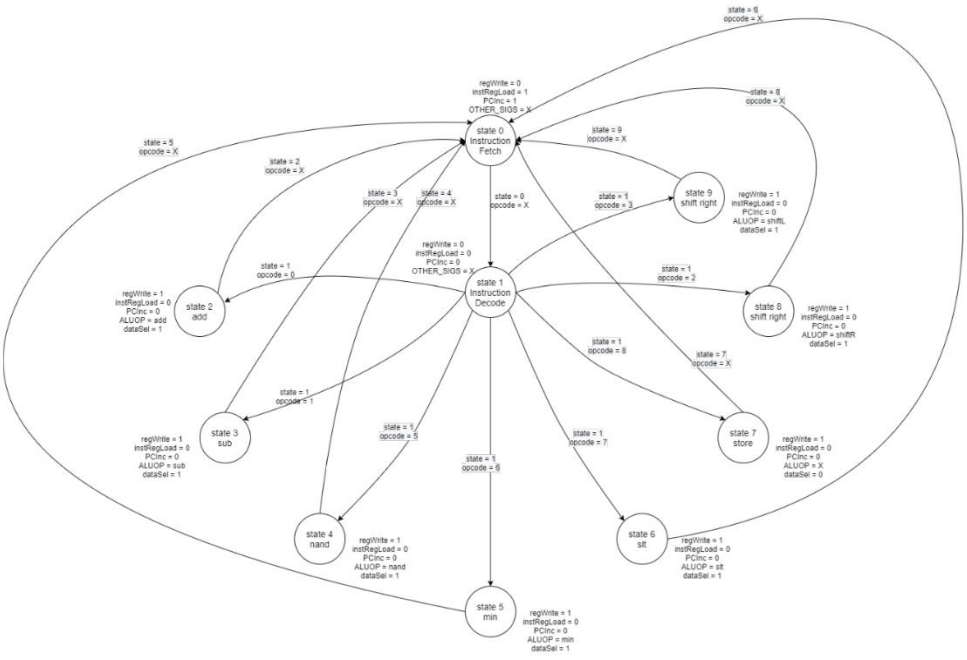
در نهایت، برای ALU، یک Mux با توجه به opcode تصمیم می‌گیرد که کدام خروجی را داشته باشد.

برای این بخش، دو تست موجود است.

1. Shifter-test که مستقل بخش شیفت‌دادن را تست می‌کند.
2. Alu-test که ماژول ALU رو به ازای همهٔ سیگنال‌های ممکن تست می‌کند.

طراحی CPU**:**

در طراحی اولیه، نخست نمودار FSM زیر به ذهن می‌رسد که مشتمل بر ۹ وضعیت است (پیوست complicated_fsm.png):

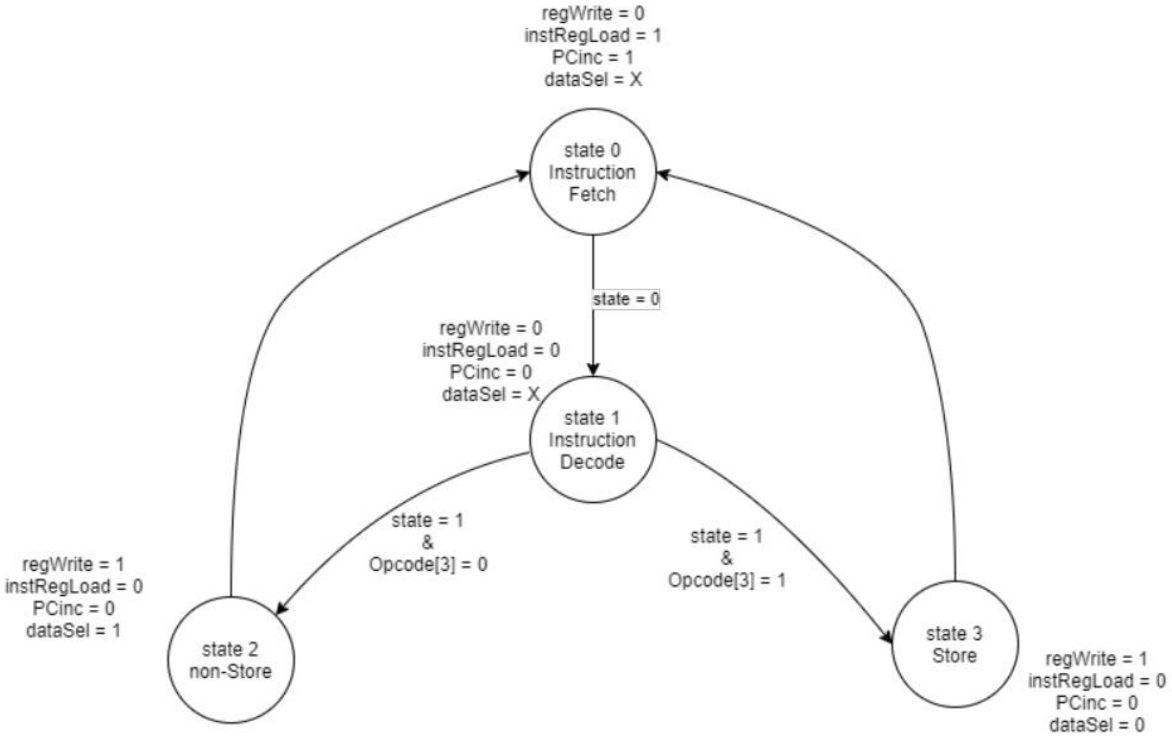


اما با دقت بیشتر درخواهیم یافت که می‌توان سیگنال‌های ALUOp را در تصمیمات واحد کنترل دخیل نکرد زیرا آن‌ها (که در حقیقت ۳ بیت کم‌ارزش opcode هستند) صرفا برای تعیین نوع عملیات محاسباتی به‌کار خواهند رفت. همچنین اگر نیک بنگریم، متوجه می‌شویم که دستورات کلا به دو صورت‌اند، یا مربوط به ذخیره‌سازی (store) هستند یا خیر. نتیجتا تمامی مشخصات دستورات دیگر متعاقبا از ۳ بیت کم‌ارزش opcode بدست خواهند آمد. پس می توانیم در کل به شمار وضعیت‌های نمودار بالا را از ۹ به ۴ کاهش داد؛ که به صورت زیر خواهند بود:

۰) واکشی دستور
۱) رمزگشایی دستور
۲) پردازش و اجرای دستورهای غیر ذخیره‌ای
۳) اجرای دستورهای ذخیره‌ای

حال اگر دقت کنیم می بینیم که تفاوت این دو نوع دستور با توجه به بیت پر ارزش opcode به راحتی مشخص می شود. پس در مجموع برای محاسبه ی سیگنال های کنترلی به ۳ بیت S1, S0, Opcode3 نیاز داریم که به ترتیب بیت سوم opcode و بیت های کم ارزش و پر ارزش state اند

در نهایت می‌توان نسخهٔ بالایی را به نمودار زیر فروکاست. (شکل این FSM ساده شده با نام simple_fsm.png به پیوست آمده است):



توجه کنید که در این طراحی سیگنال dataSel مشخص می کند که خروجی ای که به ثبات مقصد قرار است برود از نتیجهٔ محاسبه شده توسط alu باشد یا این که برای دستورات store این خروجی را از بیت ها ۵ تا ۹ (یعنی همان shift amount) بگیرد.

حال به سراغ محاسبه ی state ها و سیگنال های کنترلی بر حسب توابع منطقی به صورت combinational می رویم:

برای محاسبه ی NS0 و NS1 یعنی محاسبه ی state بعدی تنها به S0 و S1 و OP3 نیاز داریم (یعنی state فعلی و بیت پرارزش opcode)

Next state	State	Opcode[3]
01	00	X
10	01	0
11	01	1
00	10	X
00	11	X

پس جدول کارنوی NS0 و NS1 را می کشیم.

NS0:

S1, S0 / OP3	00	01	11	10
0	1	0	0	0
1	1	1	0	0

$$NS0 = S1'.S0' + OP3.S1'$$

NS1:

S1, S0 / OP3	00	01	11	10
0	0	1	0	0
1	0	1	0	0

$$NS1 = S1'.S0$$

حال برای سیگنال های register write enable, Instruction load, PC increment, data select بر اساس state مقدارشان را به دست می آوریم:

Register write enable	Instruction load	PC increment	Data select	State
0	1	1	X	00
0	0	0	X	01
1	0	0	1	10
1	0	0	0	11

پس به ترتیب معادلات زیر را خواهیم داشت:

$$RWE = S1$$

$$IL = S1'.S0'$$

$$PCI = S1'.S0'$$

$$DS = S0'$$

به این ترتیب واحد کنترلی ما به صورت ترکیبی ساخته می شود که می توانید برای دیدن آن به فایل cu.bdf مراجعه کنید

همچنین فایل کامل پردازنده ی طراحی شده با نام cpu.bdf به پیوست آمده است

برای تست این cpu نیز از یک rom استفاده شده است که ۲۵۶ سطر و ۲۰ ستون دارد. چون فضای آدرس دهی pc ۳۲ بیت است و فضای بسیار بزرگی می باشد، تنها از ۸ بیت کم ارزش pc استفاده شده که در مجموع ۲۵۶ دستور قابل ذخیره سازی در rom است.

دستورات مورد تست در فایل cpu_test_instructions.txt به پیوست آمده است که دستورات این فایل در rom مذکور به ترتیب از آدرس ۱ به بعد ریخته شده است. همچنین یک assembler ساده برای تولید دستورات با پایتون زده شده که می توانید برای تولید و تست دستورات دیگر از آن استفاده کنید. این assembler نیز با نام assembler.py به پیوست آمده است

فایل wave form تست انجام شده نیز با نام cpu_test_waveform.vwf موجود است.

دقت کنید که در پردازندهٔ طراحی شده برای آن که در تست ها مقادیر رجیستر های رجیستر فایل معلوم باشد ۸ ثبات‌های r0 تا r7 به عنوان خروجی موجود هستند (به علت محدودیت تعداد پورت های خروجی در کوارتوس امکان آن وجود نداشت که هر ۳۲ ثبات را در RF خروجی دهیم). لذا این ۸ مورد به خروجی های RF اضافه شدند و برای تست مجبور به ایجاد چنین تغییری در RF شدیم و تست‌ها نیز تنها با همین ۸ طراحی شده است که نتیجه مشخص باشد.

تست cpu در فایل cpu_test_waveform موجود است.