پروژه پایپ لاین سایه

در مرحله اول Datapath مورد نیاز برای طراحی پایپ لاین بدون سیگنال های کنترلی را رسم می کنیم.

سپس سیگنال های کنترلی مورد نیاز برای ماژول های اصلی و MUX های مورد نیاز را مشخص می کنیم.

در مرحله بعد انواع Hazard های موجود را بررسی می کنیم.

برای انواع Hazard کنترلی برای رفع آن ها ارائه می دهیم.

در پایان رجیسترهای مورد نیاز برای طبقات پایپ لاین را مشخص می نماییم.

- طبقات پایپ لاین sayeh شامل 5 طبقه

Instruction Fetch

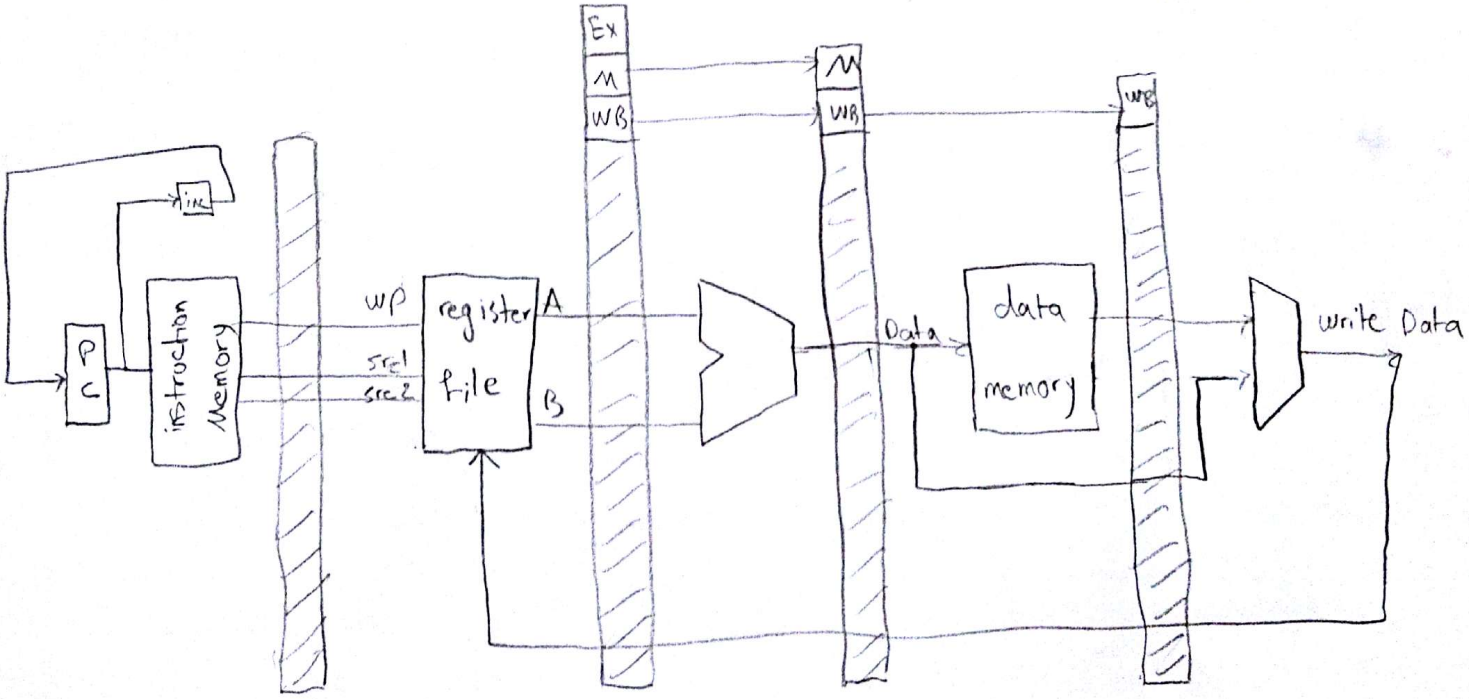
Instruction Decode

Execute

Memory

Write back

شکل کلی Datapath مطابق شکل زیر است.



رجیسترهای کنترلی مربوط به هر طبقه با استفاده از رجیسترهای طبقه بعد به مرحله بعد انتقال می یابند.

ورودی رجیستر فایل یا از Memory یا از خروجی ALU نوشته می شود.

برای مشخص کردن آدرس در رجیستر فایل از سه سیگنال wp ، src1 ، src2 استفاده می کنیم.

Data path را برای انواع دستورات کامل می‌کنیم

1. دستورات mvr ، and ، orr ، not ، shl ، shr ، add ، sub ، mul ، cmp :

نتیجه‌ی این نوع دستورات در Alu در طبقه Ex انجام شده و نتیجه‌ی آن در مرحله wb در رجیستر فایل ریخته می‌شود

---

2. دستورات LDA و STA : برای انجام این دو دستور نیاز داریم تا ALU مستقیماً با بیت کنترلی یکی از دو ورودی‌اش را در خروجی خود قرار دهد

برای دستور LDA : $R_D \Leftarrow (Rs)$ ، Rs را در خروجی Alu قرار می‌دهیم تا وارد آدرس MEM در مرحله بعد شود و خروجی MEM در مرحله WB وارد R_D رجیستر فایل می‌کنیم.

برای دستور STA ، $(R_D) \Leftarrow Rs$ ، R_D که آدرس مقصد حافظه است را در خروجی Alu قرار می‌دهیم تا مستقیماً وارد آدرس MEM شود. Rs را هم با کشیدن سیمی مستقیم از Rs (قبل از Alu) وارد ورودی MEM data می‌کنیم

---

3. برای دو دستور mil و mih ، قسمت immediate را به جای Rs وارد Alu می‌کنیم تا مشابه دستورات محاسباتی نتیجه در قسمت WB در رجیستر R_D نوشته شود

---

4. دستورات czf , szf و scf ، ccf .

رجیسترهای z و c پس در قسمت EX/MEM قرار می‌گیرند که برای 4 دستور بالا می‌توان با استفاده از بیت‌های کنترلی مقدارشان را set و clear می‌کنیم

در حالت عادی برای دستوراتی مانند cmp با استفاده از کنترلر می‌توان تعیین کرد که خروجی‌های z و c ، set شوند

---

5. دستورات inp و out مشابه با دو دستور LDA و STA انجام می‌شود

با این تفاوت که سخت‌افزاری مانند MEM در طبقه MEM استفاده می‌شود.

6- برای دستور awp ، cwp یک adder در طبقه ID تعریف می‌کنیم که ورودی‌های آن I و IF/ID (رجیستر wp) است که با استفاده از بیت کنترلی خروجی این adder یا مناسبت برای دستور cwp و یا جمع دو ورودی است برای دستور awp . درنهایت این خروجی در رجیستر wp مرحله ID/EX قرار می‌گیرد.

7- spc : PC از مرحله IF در رجیسترهای میانی PC منتقل می‌شود در مرحله WB با وجود یک MUX که ورودی‌های آن با این خروجی صوری و خروجی Alu بوده خروجی PC (MEM/WB هم) به آن اضافه می‌شود و با استفاده از بیت کنترلی در رجیستر فایل نوشته شده و ذخیره می‌شود.

8- دستورات JPA $PC \leftarrow R_D + I$

JPR $PC \leftarrow PC + I$

برای دستور JPA عمل جمع R_D, I را در مرحله EXE انجام می‌دهیم و خروجی آن را در ورودی MUX : PC تعریف می‌دهیم. اما برای دستور JPR با استفاده از یک adder در مرحله ID ، PC+I را در ورودی MUX PC تعریف می‌دهیم. علت تفاوت در انجام add در دو دستور بالا در دو مرحله به خاطر این است که در دستور JPA خواندن RD از رجیستر فایل زمان بر است و اگر add در همان طبقه انجام شود از طول کلاک بیش‌تر می‌شود. بنابراین در Hazard کنترلی برای JPA در JPR یک stall خواهد خورد.

9- دستورات brz ← branch if zero

brc ← branch if carry

این دو دستور مشابه دستور JPR ، PC+I آن در مرحله ID با استفاده از همان adder انجام می‌شود اما چون نتیجه‌ی branch در مرحله EXE مشخص می‌شود پس باید PC را با استفاده از رجیستر PC در ID/EXE به مرحله EXE انتقال دهیم و در آن جا کنترل تشخیص می‌دهد اگر branch باید انجام شود PC+I ذخیره شده در رجیستر PC، ID/EXE را به ورودی MUX ، PC انتقال دهد

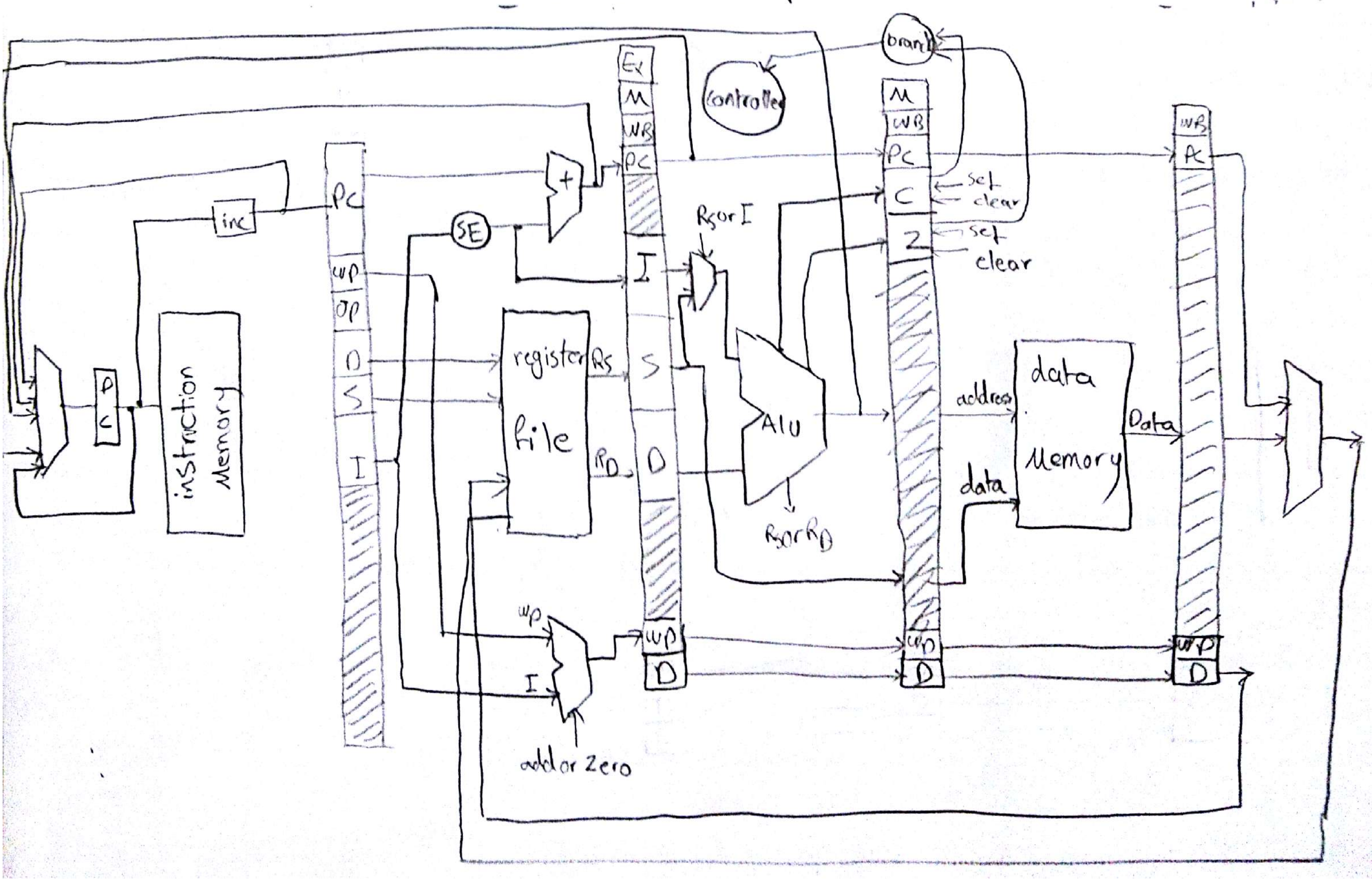
چون مشخص شدن branch برای این ۴ دستور حتماً باید به ALU انجام شود نمی‌توان سخت افزاری جداگانه برای آن در نظر گرفت تا همانند MIPS در مرحله ID انجام شود پس ناچاریم که آن را در مرحله EXE مشخص کنیم. پس فعلاً برای مشخص شدن branch و ورودی PC نیاز است که برای این دو دستور پایپ‌لاین 2 stall بخورد

10- دستور nop : به سخت افزار جدیدی ندارد زیرا می‌توان با استفاده از کنترلر همه سیگنال های کنترلی نوشتن در رجیسترفایل یا مموری را غیرفعال کند تا تأثیری در داده‌های موجود نباشد.

11- دستور HLT ← Fetching stops

چون نیاز داریم که HLT در همین مرحله Fetching تشخیص داده شود پس با سخت افزار جدیدی آن را در مرحله IF مشخص می‌کنیم که اگر دستور HLT بود PC قبل از این که تغییر کند مقدار کنونی‌اش در خودش ریخته شود

با توجه به توضیحات برای دستورات Datapath مطابق شکل زیر است.



تا این لحظه تعداد رجیسترهای موردنیاز مطابق زیر است

IF/ID: PC, wp, opcode, D, S, I → 6

ID/EXE: EXE, MEM, WB, PC, I, S, D, wp, D → 9

EXE / MEM: MEM, WB, PC, C, Z, wp, D → 7

MEM/WB: WB, PC, wp, D → 4

اکنون انواع Hazard ها وجود و نحوه ی رفع آن ها را بررسی می کنیم.

1- Structural: از دو حافظه ی مجزا برای دستورالعمل و داده استفاده می کنیم.

2- Data hazard:

زمانی اتفاق می افتد که رجیستر مقصد در دستور فعلی با رجیستر مبدأ در دستور بعدی برابر باشد. چون در طبقه wb نتیجه در رجیستر مقصد نوشته می شود پس این نوع Hazard اتفاق می افتد. برای دو دستور بعدی هم اگر همین شرط برقرار باشد باز هم این نوع Hazard اتفاق می افتد

	مقصد	مبدأ
add	(R1)	R2
add	R3	(R1)

add	(R1)	R2
add	R2	R3
add	R4	(R1)

برای حل این مشکل و جلوگیری از stall خوردن از forwarding استفاده می کنیم.

forwarding نیتی است که به pipeline اضافه می شود و در آن ما شرط های زیر با استفاده از رجیسترهای میانی که انتقال داده شدند چک می شوند که اگر برقرار بود forwarding انجام می شود

if EX/MEM Rd = ID/EX Rd or Rs

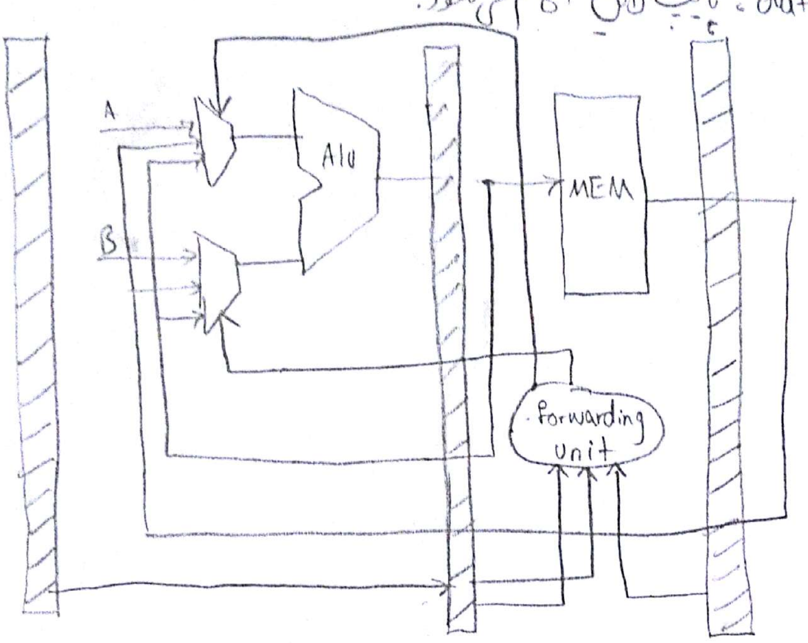
if MEM/WB Rd = EX/MEM Rd or Rs

برای پیاده سازی سخت افزاری آن ورودی ALU به جای آن که از Rs و Rd استفاده کنیم

از خروجی ALU در مراحل بعدی یعنی MEM و WB استفاده می کنیم تا مستقیماً در [illegible]

دستورات بعدی استفاده می شود

این تغییرات در datapath پایپ لاین انجام می شود.



اگر هر دو شرط صفحه ی قبل نیز همزمان با هم برقرار بود ورودی های ALU از مرحله MEM/WB انتقال می یابد

نوع دیگری از data hazard هم وجود دارد که نمی توان آن را با forwarding به صورت کامل حل کرد و

در دستور LDA این اتفاق رخ می دهد زیرا داده بر خلاف حالت قبل که در خروجی ALU ایجاد

می شد در این دستور در مرحله MEM آماده می شود بنابراین نیاز است تا pipe line یک stall

در ID بخورد

برای تشخیص آن در forwarding unit از شرط های زیر استفاده می کنیم.

if ID/E.MemRead and

ID/EXE R_d = IF/ID R_s or R_d

stall هم مشابه آن چه در datapath گفته شد انجام می شود یعنی PC همان مقدار

می ماند و رجیسترهای مرحله ID/EXE همگی ریست می شوند تا در واقع NOP انجام شود

control Hazard -3

مشاهده کردیم در دستورات JPA و JPR و brz و brc گفته شد نیاز به stall برای

مشخص شدن branch یا مشخص شدن مقدار جدید PC نیاز داریم

JPR → stall یک → PC ⇐ PC+I ⇐ درمرحله ID انجام می‌شود

JPA → 〃 دو → PC ⇐ R_D+I ⇐ درمرحله ID انجام می‌شود

اما تا مشخص شدن شرط

branch در EXE به PC انتقال می‌یابد

brz
brc } ⇒ stall دو → PC ⇐ PC+I ⇒

هر دو در مرحله EXE

مشخص و مقدار جدید PC به PC انتقال می‌یابد.

تشخیص این 4 دستور در مرحله decode انجام می‌شود و با سخت افزار موجود به تعداد مورد نیاز هر دستور

stall می‌خورد.

برای حل این مشکل از delayed branch استفاده می‌کنیم

که به کامپایلرنویس این اجازه را می‌دهد که دو دستور یا یک دستور بعد از دستورات پرش بدون توجه

به انجام یا عدم انجام پرش به وقوع بپیوندند.

در این صورت دیگر نیاز به 1 یا 2 stall نخواهیم داشت.

روش دیگر نیز استفاده از Dynamic branch prediction است