به نام خدا

گزارشکار فاز چهارم پروژه درس معماری کامپیوتر

پیاده‌سازی Branch Prediction

استاد

دکتر حمید سربازی آزاد

اعضای گروه

محمدپارسا بشری ۴۰۰۱۰۴۸۱۲

محسن قاسمی ۴۰۰۱۰۵۱۶۶

امیرحسین رازلیقی ۹۹۱۰۲۴۲۳

بهار ۱۴۰۲

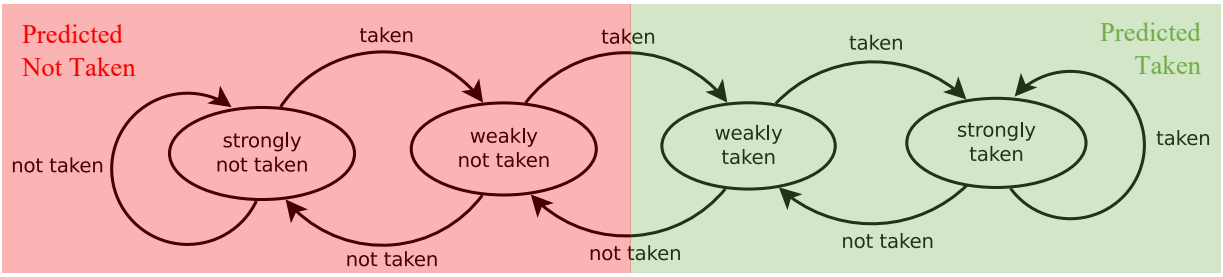
فهرست

مقدمه و هدف فاز چهارم

هدف کلی این پروژه، طراحی و پیاده‌سازی یک پردازنده MIPS است. در فاز اول Datapath و Control Unit این پردازنده را به صورت Single Cycle طراحی و پیاده‌سازی کردیم. در فاز دوم یک ماژول حافظه نهان (Cache) به پردازنده‌مان اضافه کردیم. در فاز سوم پردازنده خود را از حالت Single Cycle به حالت Pipelined تغییر دادیم. در این فاز می‌خواهیم با استفاده از branch prediction، مخاطرات کنترلی پایپلاین (control hazards) را برطرف کنیم. ایده اصلی این فاز پیش‌بینی انجام یا عدم انجام پرش قبل از مشخص شدن واقعی تصمیم پرش است. در این فاز از saturation counter استفاده می‌کنیم که توضیحات بیشتر در ادامه خواهد آمد.

طرز کار Saturation Counter

استفاده از saturation counter به ما کمک می‌کند که با توجه به پرش‌های قبلی، بتوانیم انجام یا عدم انجام پرش را پیش‌بینی کنیم. Saturation counter یک شمارنده دو بیتی است که طبق شکل زیر، با هر بار پرش مقدار خودش را آپدیت می‌کند.

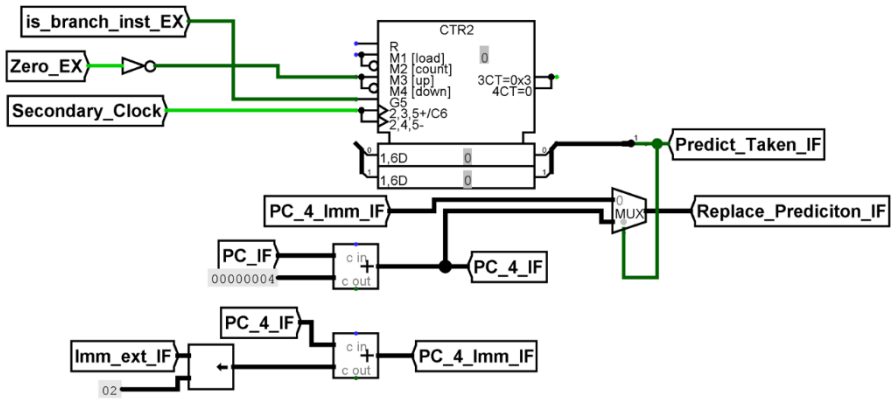


شکل ۱- نحوه کار saturation counter

برای استفاده از این شمارنده باید به حالت فعلی آن توجه کنیم. اگر در یکی از حالات weakly taken یا strongly taken قرار داشتیم، پیش‌بینی می‌کنیم که پرش بعدی انجام خواهد شد؛ اما اگر در حالات weakly not taken و یا strongly not taken بودیم، پیش‌بینی خواهیم کرد که پرش اتفاق نخواهد افتاد و خط بعدی برنامه اجرا خواهد شد (این پیش‌بینی به راحتی با نگاه کردن به بیت سمت چپ شمارنده قابل انجام است). سپس وقتی به استیج EX رسیدیم و واقعا نتیجه پرش مشخص شد، با توجه به اینکه پیش‌بینی‌مان درست بوده یا خیر، saturation counter را آپدیت می‌کنیم.

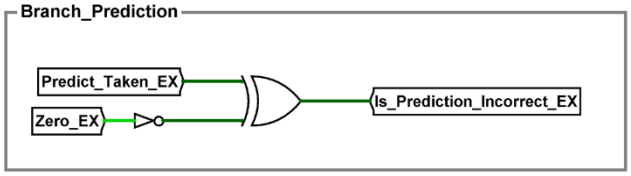
پیاده سازی Saturation Counter

در اولین قدم باید خود شمارنده را پیاده‌سازی کنیم. به جای شمارنده‌ای که برای دستورات پرش شرطی (branch) در فاز قبل گذاشته بودیم، یک شمارنده جدید قرار می‌دهیم و گزینه Action On Overflow آن را در حالت Stay at value می‌گذاریم تا عملکرد مورد نظرمان در شکل ۱ را داشته باشد. همچنین آدرس جایگزینی که تصمیم گرفته‌ایم به آن پرش نکنیم را نیز تولید کرده و به داخل پایپلاین می‌فرستیم.



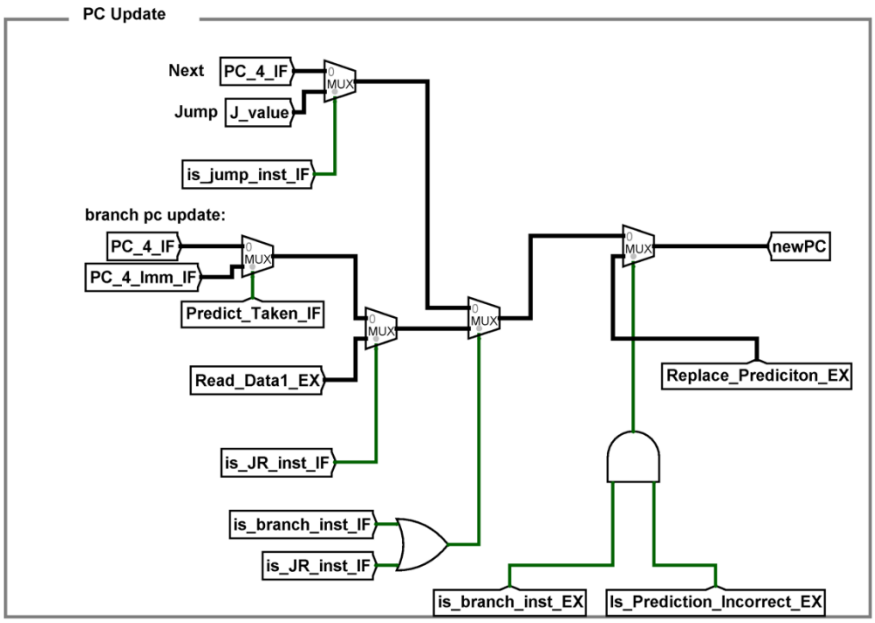
شکل ۲- پیاده‌سازی saturation counter

حالا در استیج EX سیگنال Zero_EX نشان می‌دهد که آیا باید پرش انجام شود یا خیر. در این استیج تصمیمی که پیش‌بینی کرده بودیم را با این سیگنال مقایسه می‌کنیم و تشخیص می‌دهیم که آیا پیش‌بینی‌مان درست بوده یا خیر.



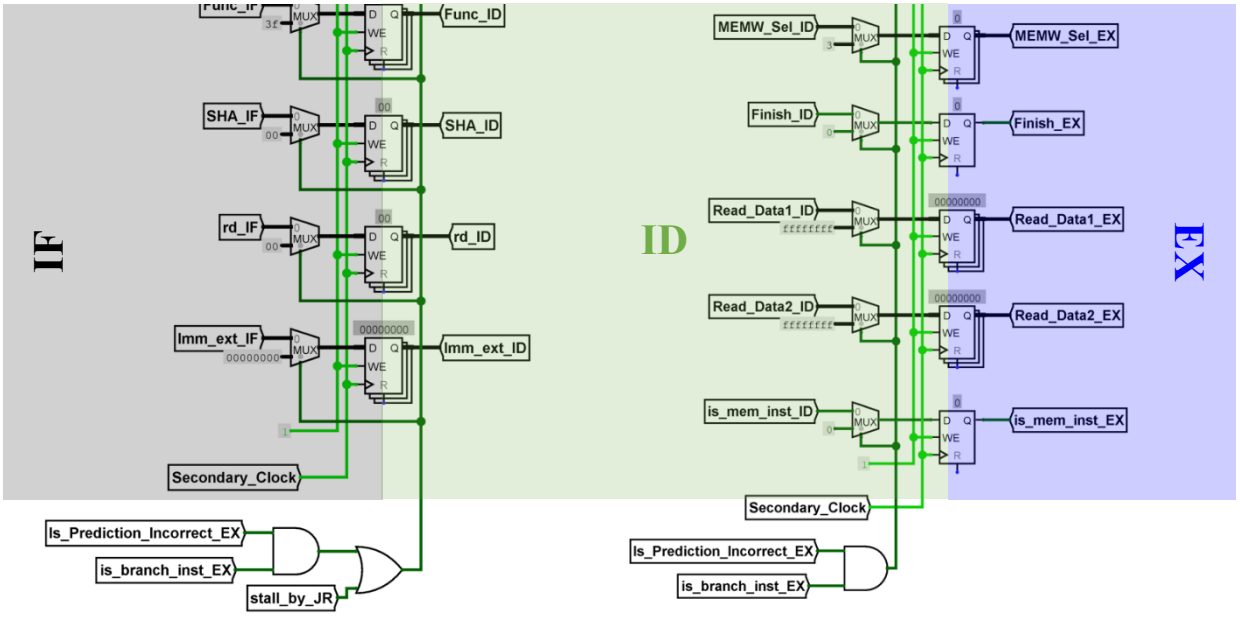
شکل ۳- بررسی درست یا نادرست بودن پیش‌بینی انجام شده

سپس با استفاده از این سیگنال، مقدار بعدی PC را مشخص می‌کنیم.



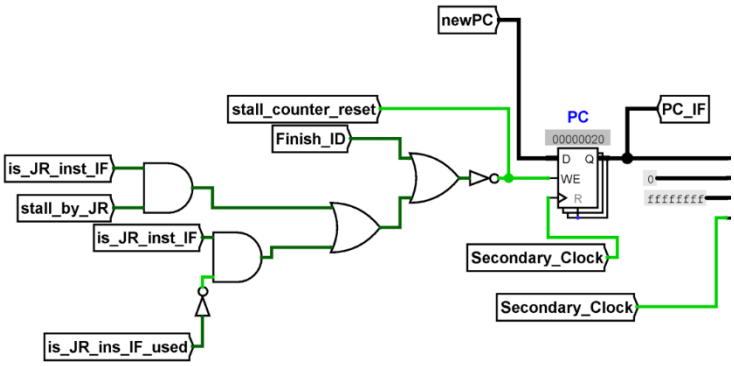
شکل ۴- منطق تولید مقدار جدید PC

همچنین در صورت پیش‌بینی اشتباه باید پایپلاین را خالی کنیم. به تقلید از فرایند stall فاز قبل، در اینجا علاوه بر بافرهای IF-ID جلوی بافرهای ID-EX هم مالتی‌پلکسر می‌گذاریم. سیگنال‌های کنترلی این مالتی‌پلکسرها نیز به شکل زیر تغییر می‌کنند.



شکل ۵- کنترل بافرهای میانی

همچنین با توجه به اینکه در این فاز فقط دستور JR باعث ایجاد stall می‌شود، مدار تولید WE برای PC را هم به شکل زیر تغییر می‌دهیم:



شکل ۶- منطق تولید WE برای PC

تست عملکرد Branch Predictor

در این فاز از branch prediction استفاده کردیم تا مخاطرات کنترلی را برطرف کنیم. برای بررسی عملکرد درست این ابزار، برنامه زیر را روی پردازنده‌مان اجرا می‌کنیم.

```
1  .text
2  and $zero, $zero, $zero
3  addi $s0, $zero, 10
4  and $zero, $zero, $zero
5  and $zero, $zero, $zero
6  and $zero, $zero, $zero
7  and $zero, $zero, $zero
8  and $zero, $zero, $zero
9  loop:
10 addi $s0, $s0, -1
11 and $zero, $zero, $zero
12 and $zero, $zero, $zero
13 and $zero, $zero, $zero
14 and $zero, $zero, $zero
15 and $zero, $zero, $zero
16 bne $s0, $zero, loop
17 addi $s1, $zero, 20
18 and $zero, $zero, $zero
19 and $zero, $zero, $zero
20 and $zero, $zero, $zero
21 and $zero, $zero, $zero
22 and $zero, $zero, $zero
```

شکل ۸- برنامه جهت تست branch prediction

```
00400000: 00000024 ; <input:7> and $zero, $zero, $zero
00400004: 2010000a ; <input:8> addi $s0, $zero, 10
00400008: 00000024 ; <input:9> and $zero, $zero, $zero
0040000c: 00000024 ; <input:10> and $zero, $zero, $zero
00400010: 00000024 ; <input:11> and $zero, $zero, $zero
00400014: 00000024 ; <input:12> and $zero, $zero, $zero
00400018: 00000024 ; <input:13> and $zero, $zero, $zero
0040001c: <loop> ; <input:14> loop:
0040001c: 2210ffff ; <input:15> addi $s0, $s0, -1
00400020: 00000024 ; <input:16> and $zero, $zero, $zero
00400024: 00000024 ; <input:17> and $zero, $zero, $zero
00400028: 00000024 ; <input:18> and $zero, $zero, $zero
0040002c: 00000024 ; <input:19> and $zero, $zero, $zero
00400030: 00000024 ; <input:20> and $zero, $zero, $zero
00400034: 1600fff9 ; <input:21> bne $s0, $zero, loop
00400038: 20110014 ; <input:22> addi $s1, $zero, 20
0040003c: 00000024 ; <input:23> and $zero, $zero, $zero
00400040: 00000024 ; <input:24> and $zero, $zero, $zero
00400044: 00000024 ; <input:25> and $zero, $zero, $zero
00400048: 00000024 ; <input:26> and $zero, $zero, $zero
0040004c: 00000024 ; <input:27> and $zero, $zero, $zero
```

شکل ۷- کد ماشین برنامه شکل ۷

برنامه بالا ۱۰ بار یک حلقه را تکرار می‌کند. اگر این برنامه را بدون استفاده از branch prediction اجرا کنیم، زمان اجرای برنامه به شکل زیر محاسبه می‌شود:

$$T_{normal} = 4 + 7 + (10 \times 9) + 6 = 107\ clocks$$

حالا اگر با استفاده از branch prediction برنامه را اجرا کنیم، در پیمایش اول و دوم و آخر پیش‌بینی اشتباه رخ می‌دهد که باعث می‌شود زمان اجرای برنامه به شکل زیر محاسبه شود:

$$T_{branch\ prediction} = 4 + 7 + (3 \times 9) + (7 \times 7) + 6 = 93\ clocks$$

بنابراین میزان speedup حاصل از branch prediction به شکل زیر محاسبه می‌شود:

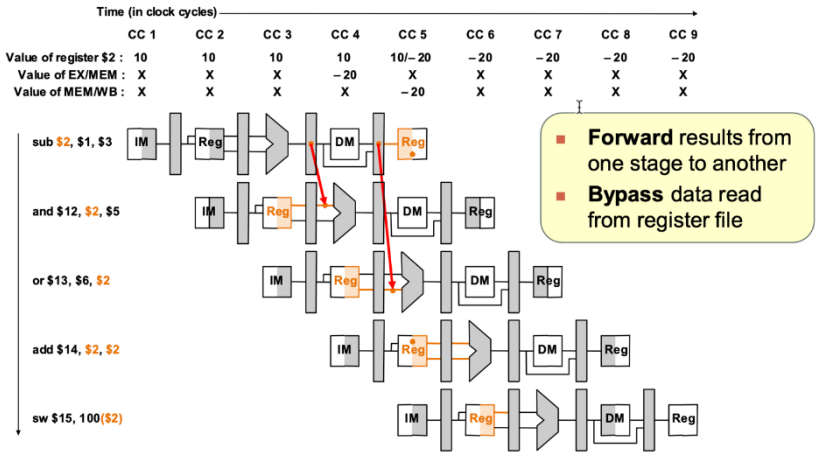
$$Speedup_{branch\ prediction} = \frac{107}{93} = 1.15$$

پس مشاهده شد که branch prediction به رفع مخاطرات کنترلی و افزایش سرعت پردازنده کمک می‌کند.

نمره امتیازی - Forwarding

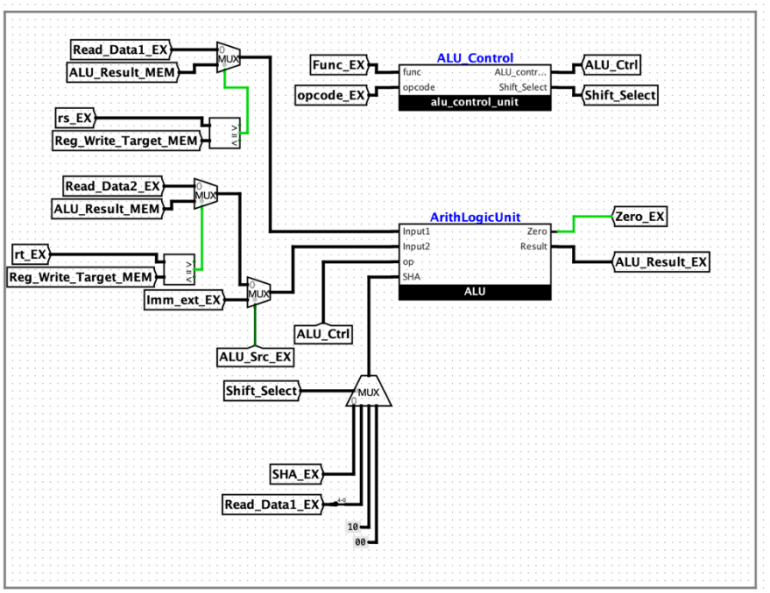
در این بخش به پیاده‌سازی Forwarding در pipeline می‌پردازیم. هدف از این‌کار این است که، فرض کنید در دستور اول که وارد pipeline می‌شود، محاسباتی را انجام می‌دهیم و آن را در رجیستر s2 ذخیره می‌کنیم. در دستور دوم، از حاصل s2 در یک محاسبه‌ی جدید استفاده می‌کنیم. بدیهی است که در دستور دوم، نیاز به حاصل s2 پس از محاسبات دستور اول داریم. اما چون در pipeline هستیم، وقتی دستور دوم به ورودی استیج ALU (یا همان EX) می‌رسد، هنوز دستور قبلی به MEM یا WB نرسیده است و در نتیجه، حاصل جدید در رجیستر مذکور ریخته نشده است! برای حل این مشکل (data hazard) از Forwarding استفاده می‌کنیم. به این صورت که خروجی ALU در دستور اول (که قرار است به استیج‌های بعدی برود) را به ورودی EX برای دستور دوم پاس می‌دهیم تا بتواند از آن استفاده کند. ضمن اینکه با جلو رفتن پایپ‌لاین و رد کردن استیج‌های MEM و WB، عملا داده‌ای که به آن پاس داده‌ایم در آن رجیستر قرار می‌گیرد و همه‌ی داده‌ها consistent باقی می‌مانند. تنها کاری که انجام می‌دهیم این است که دسترسی زود هنگام استیجی که به s2 نیاز داشته را به آن فراهم می‌کنیم، بدون اینکه bubble درون پایپ‌لاین قرار دهیم و stall رخ دهد، که بهبود قابل توجه و مهمی است!

تصویر زیر را درنظر بگیرید:



شکل ۹- مثال از کاربرد forwarding

همانطور که مشاهده می‌کنید، در دستور خط دوم، به حاصل تفریق خط اول نیاز داشته‌ایم درحالیکه آن حاصل هنوز در pipeline به انتها نرسیده تا write back صورت بگیرد. با forwarding، حاصل را از ALU مستقیما به ورودی stage بعدی پاس می‌دهیم تا بتوانیم قبل از writeback هم به آن دسترسی داشته باشیم. در پروژه‌ی ما، نحوه‌ی پیاده‌سازی اینگونه است:



شکل ۱۰- پیاده سازی forwarding

همانطور که مشحص است، در ALU و هنگام مشخص کردن ورودی، MUXهایی اضافه شده است که FORWARDING را انجام دهد. نحوه‌ی انجام بدین صورت است که در ابتدا، یک مقایسه کننده، مقدار register source که از استیج EX هست را با مقدار Reg_Write_Target_Mem که مقایسه می‌کند. در صورت تساوی، یعنی حاصل ALU یا همان ALU_Result_MEM را می‌توانیم به عنوان ورودی پاس بدهیم. اما اگر چنین نباشد (مساوی نباشند)، یعنی خروجی قبلی (که در استیج ALU هست) را در ورودی کنونی اصلا نیازی نداریم و باید بصورت عادی همان ورودی‌های ALU را به آن بدهیم و محاسبات را انجام بدهیم. در تصویر هم می‌بینید که ورودی 0 مالتی‌پلکسر به Read_Data وصل شده است. این ساختار ترکیبی (MUX + Comparator) را هم برای ورودی اول (Input1) و هم برای ورودی دوم (Input2) در ALU انجام می‌دهیم (بدیهی است، زیرا که برخی دستورات، در register source تغییرات را اعمال می‌کنند و برخی دیگر در register target و در نتیجه در هر دو این موارد باید Forwarding انجام شود).

تست بخش forwarding

در فاز قبلی مثالی برای Data hazard آورده بودیم که در فاز قبل جواب اشتباه را ذخیره می‌کرد. در اینجا با استفاده از forwarding آن مشکل را برطرف کرده‌ایم:

```
1 .text
2 and $zero, $zero, $zero
3 addi $s0, $zero, 10
4 addi $s1, $s0, 10
5 and $zero, $zero, $zero
6 and $zero, $zero, $zero
7 and $zero, $zero, $zero
8 and $zero, $zero, $zero
9 and $zero, $zero, $zero
```

شکل ۱۱- برنامه جهت تست forwarding

در نهایت مشاهده می‌شود که عدد ۲۰ در رجیستر ۱۷ ذخیره شده است. پس با استفاده از forwarding توانستیم data hazard را برطرف نماییم.

منابع و مراجع

- اسلایدهای درس
- https://alanhogan.com/asu/assembler.php
- https://www.researchgate.net/figure/Two-bit-saturating-counter_fig3_221219835
- https://en.wikipedia.org/wiki/Branch_predictor