به نام خدا

گزارشکار فاز سوم پروژه درس معماری کامپیوتر

طراحی و پیاده‌سازی Pipeline

استاد

دکتر حمید سربازی آزاد

اعضای گروه

محمدپارسا بشری ۴۰۰۱۰۴۸۱۲

محسن قاسمی ۴۰۰۱۰۵۱۶۶

امیرحسین رازلیقی ۹۹۱۰۲۴۲۳

بهار ۱۴۰۲

فهرست

مقدمه و هدف فاز سوم

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 می‌خواهیم پردازنده خود را از حالت Single Cycle به حالت Pipelined تغییر دهیم. ایده اصلی این فاز، تقسیم هر دستور به پنج مرحله (stage) و انجام هر مرحله در یک کلاک است به طوری با هر کلاک، یک دستور جدید وارد پایپلاین شده و مراحل مورد نیاز را طی می‌کند. بنابراین به طور معمول، پنج دستور به صورت همزمان در حال اجرا هستند و با هر کلاک، اجرای یک دستور خاتمه می‌یابد. در این گزارش (بر خلاف گزارش فازهای قبل که عملیات تست عملکرد پردازنده در انتهای گزارش آمده بود) پس از پیاده‌سازی هر قسمت از این فاز، تست عملکرد مربوط به آن مرحله آورده خواهد شد.

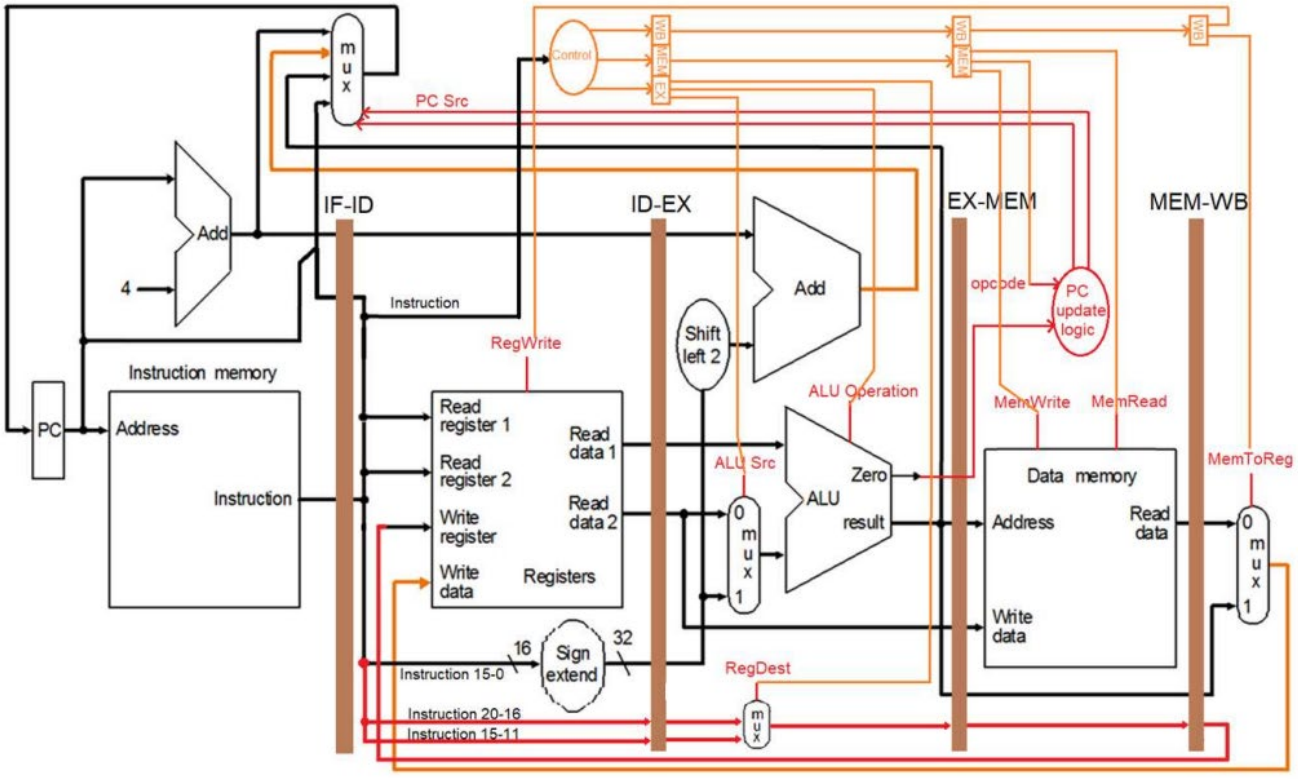
طراحی اولیه پایپلاین

پایپلاین در پردازنده MIPS دارای پنج stage است که در جدول ۱ نشان داده شده است:

Stage	Operation
IF	Instruction Fetch
ID	Instruction Decode and Register Read
EX	Execute an operation or calculate an address
MEM	Access an operand in data memory
WB	Write back the result into a register

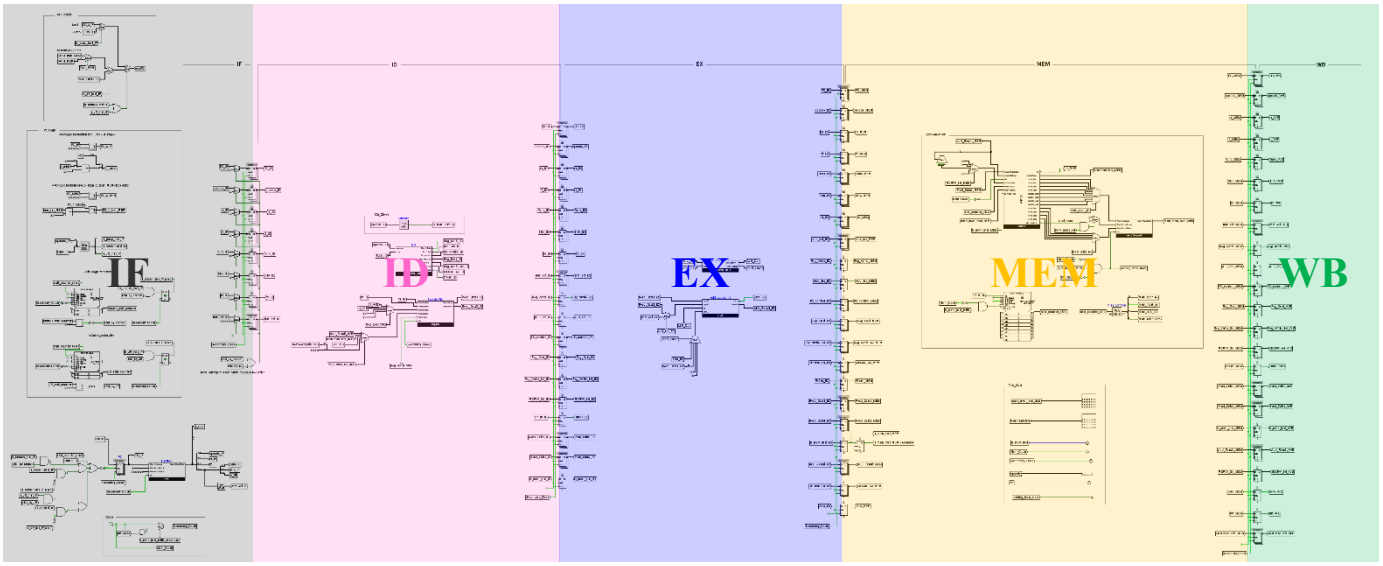
جدول ۱ - stage های پایپلاین در پردازنده MIPS

بنابراین اگر هیچ مشکلی در اجرای دستورات پیش نیاید (در ادامه این مشکلات را بررسی و حل می‌کنیم) و دستورات یکی پس از دیگری اجرا شوند، اجرای N دستور در N+4 کلاک امکان پذیر است. طراحی پایپلاین اولیه را طبق شماتیک زیر انجام می‌دهیم:



شکل ۱ - شماتیک اولیه پایپلاین

طراحی اولیه پایپلاین را دقیقا مطابق شکل ۱ انجام می‌دهیم (هدف از شکل زیر نمای کلی پردازنده است):



شکل ۲- طراحی کلی پردازنده به صورت pipeline

تست پایپلاین اولیه

تنها کاری که تا اینجا انجام دادیم، دسته‌بندی ماژول‌ها و قرار دادن رجیستر بین stage های پایپلاین بود. در ضمن PC را که در فازهای قبل (به دلیل single cycle بودن) حساس به لبه پایین طراحی کرده بودیم، برای اجرای درست در پایپلاین، حساس به لبه بالارونده کردیم. تا به اینجای کار پردازنده‌مان باید بتواند برنامه‌ای که شامل دستوراتی به جز دستورات حافظه و پرش باشد و همچنین data dependency نداشته باشد را به صورت پایپلاین اجرا کند.

به عنوان تست، برنامه زیر را که شامل **۱۹ دستور** است را اجرا می‌کنیم و مشاهده می‌کنیم که در **۲۴ کلاک** اجرای دستورات تمام می‌شود. فرایند تست کردن (تبدیل به کد ماشین و لود کردن روی ماژول instMem) دقیقا مشابه فازهای قبل است.

```
.text

addi $s0, $zero, 10
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
addi $s1, $zero, 20
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
addi $s2, $zero, 30
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
addi $s3, $zero, 40
```

شکل ۴- برنامه جهت تست پایپلاین ایده‌آل

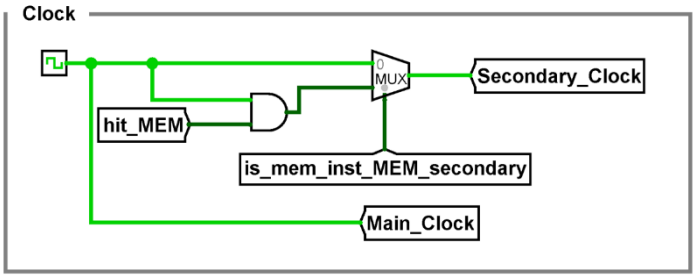
```
00400000: 2010000a ; <input:8> addi $s0, $zero, 10
00400004: 01084024 ; <input:9> and $t0, $t0, $t0
00400008: 01294824 ; <input:10> and $t1, $t1, $t1
0040000c: 014a5024 ; <input:11> and $t2, $t2, $t2
00400010: 016b5824 ; <input:12> and $t3, $t3, $t3
00400014: 018c6024 ; <input:13> and $t4, $t4, $t4
00400018: 20110014 ; <input:14> addi $s1, $zero, 20
0040001c: 01084024 ; <input:15> and $t0, $t0, $t0
00400020: 01294824 ; <input:16> and $t1, $t1, $t1
00400024: 014a5024 ; <input:17> and $t2, $t2, $t2
00400028: 016b5824 ; <input:18> and $t3, $t3, $t3
0040002c: 018c6024 ; <input:19> and $t4, $t4, $t4
00400030: 2012001e ; <input:20> addi $s2, $zero, 30
00400034: 01084024 ; <input:21> and $t0, $t0, $t0
00400038: 01294824 ; <input:22> and $t1, $t1, $t1
0040003c: 014a5024 ; <input:23> and $t2, $t2, $t2
00400040: 016b5824 ; <input:24> and $t3, $t3, $t3
00400044: 018c6024 ; <input:25> and $t4, $t4, $t4
00400048: 20130028 ; <input:26> addi $s3, $zero, 40
```

شکل ۳- کد ماشین برنامه تست

مشاهده می‌شود که بعد از گذشت ۲۴ کلاک، تمامی دستورات به درستی انجام شده و مقادیر ۱۰، ۲۰، ۳۰، ۴۰ در s0 تا s3 ذخیره شده و مقادیر اولیه t0 تا t3 حفظ شده است.

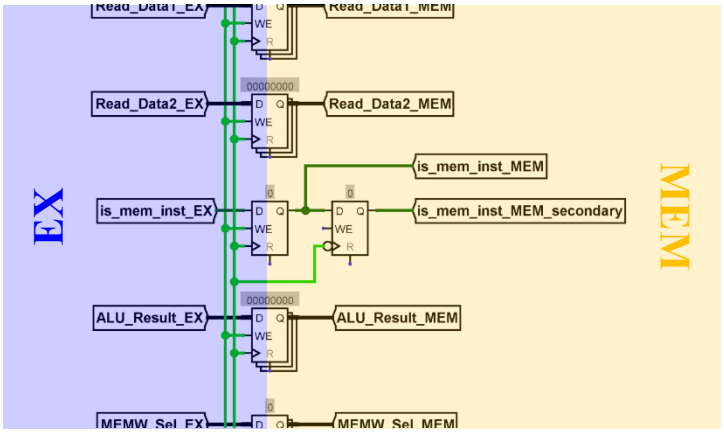
فرایند miss خوردن Cache

در صورتی که در استیج MEM در Cache میس رخ دهد، باید کل پایپلاین فریز شود. برای پیاده‌سازی این فرایند، همانند قسمت قبل عمل کرده و یک secondary clock می‌سازیم که به کل مدار (به جز ماژول dataMem) متصل است.



شکل ۵- منطق ایجاد کلاک فرعی

تنها تفاوتی که این مدار با فاز قبل دارد، سیگنال is_mem_inst_MEM_secondary است که بعد از تولید در استیج ID از طریق بافرها، حالا به استیج MEM رسیده و نشان می‌دهد که دستوری که در حال حاضر در استیج MEM است از نوع دستورات حافظه است یا خیر.



شکل ۶- تولید سیگنال is_mem_inst_MEM_secondary

علت وجود یک فلیپ‌فلاپ دیگر این است که اگر از خود سیگنال is_mem_inst_MEM استفاده می‌کردیم، کلاک Register File هم متوقف می‌شد (به علت حساسیت به لبه بالا) و دستوری که در حال حاضر در مرحله WB بود اجرا نمی‌شد. بنابراین یک رجیستر حساس به لبه پایین قرار دادیم که غیرفعال شدن کلاک، در لبه پایین رونده اتفاق بیفتد.

تست دستورات حافظه

برای اینکه از عملکرد پایپلاین حین کار با Cache مطمئن شویم، برنامه زیر را روی پردازنده اجرا می‌کنیم:

```
1  .text
2  and $zero, $zero, $zero
3  addi $s0, $zero, 64
4  addi $s1, $zero, 84
5  lw $s2, 0($zero)
6  addi $s3, $zero, 10
7  and $zero, $zero, $zero
8  and $zero, $zero, $zero
9  and $zero, $zero, $zero
10 and $zero, $zero, $zero
11 and $zero, $zero, $zero
12 addi $s2, $s2, 10
13 and $zero, $zero, $zero
14 and $zero, $zero, $zero
15 and $zero, $zero, $zero
16 and $zero, $zero, $zero
17 and $zero, $zero, $zero
18 sw $s2, 0($zero)
19 and $zero, $zero, $zero
20 and $zero, $zero, $zero
21 and $zero, $zero, $zero
22 and $zero, $zero, $zero
23 and $zero, $zero, $zero
```

شکل ۸- برنامه جهت تست دستورات حافظه‌ای در پایپلاین

```
00400000: 00000024 ; <input:7> and $zero, $zero, $zero
00400004: 20100040 ; <input:8> addi $s0, $zero, 64
00400008: 20110054 ; <input:9> addi $s1, $zero, 84
0040000c: 8c120000 ; <input:10> lw $s2, 0($zero)
00400010: 2013000a ; <input:11> addi $s3, $zero, 10
00400014: 00000024 ; <input:12> and $zero, $zero, $zero
00400018: 00000024 ; <input:13> and $zero, $zero, $zero
0040001c: 00000024 ; <input:14> and $zero, $zero, $zero
00400020: 00000024 ; <input:15> and $zero, $zero, $zero
00400024: 00000024 ; <input:16> and $zero, $zero, $zero
00400028: 2252000a ; <input:17> addi $s2, $s2, 10
0040002c: 00000024 ; <input:18> and $zero, $zero, $zero
00400030: 00000024 ; <input:19> and $zero, $zero, $zero
00400034: 00000024 ; <input:20> and $zero, $zero, $zero
00400038: 00000024 ; <input:21> and $zero, $zero, $zero
0040003c: 00000024 ; <input:22> and $zero, $zero, $zero
00400040: ac120000 ; <input:23> sw $s2, 0($zero)
00400044: 00000024 ; <input:24> and $zero, $zero, $zero
00400048: 00000024 ; <input:25> and $zero, $zero, $zero
0040004c: 00000024 ; <input:26> and $zero, $zero, $zero
00400050: 00000024 ; <input:27> and $zero, $zero, $zero
00400054: 00000024 ; <input:28> and $zero, $zero, $zero
```

شکل ۷- کد ماشین برنامه تست

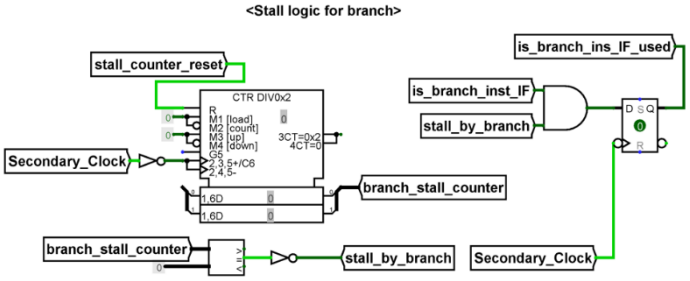
مشاهده می‌شود که هنگام miss شدن (خط ۵) کل پایپلاین به مدت ۱۸ کلاک فریز شده تا فرایند آوردن بلاک از ممووری کامل شود. بنابراین کل اجرای برنامه ۴۰ کلاک طول می‌کشد.

دستورات پرش در پایپلاین

دستورات پرش را به دو دسته پرش شرطی و غیرشرطی تقسیم می‌کنیم:

دستورات پرش شرطی (branch)

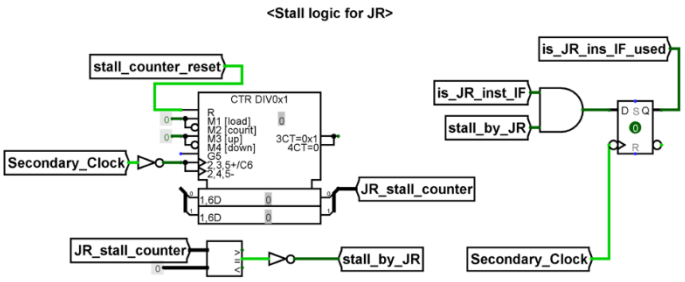
دستورات پرش شرطی (branch) نیاز به ۳ کلاک stall دارند زیرا تصمیم پرش در استیج MEM گرفته می‌شود. برای پیاده‌سازی این دستورات در پایپلاین، از یک شمارنده استفاده می‌کنیم که از ۰ تا ۲ بشمرد (برای ایجاد ۳ عدد stall). سیگنال reset این شمارنده فعال می‌ماند و به محض ورود یک دستور branch به پایپلاین، غیر فعال شده و شمارنده شروع به شمردن می‌کند. در حین شمردن، سیگنال stall_by_branch فعال می‌شود.



شکل ۹- منطق ایجاد stall برای دستورات branch

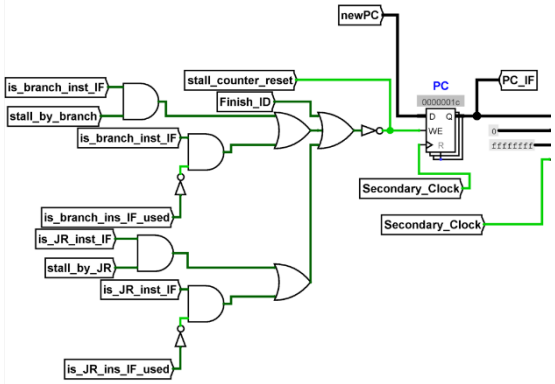
دستورات پرش غیرشرطی (jump)

اکثر دستورات پرش بدون شرط (jump) بدون نیاز به stall اجرا خواهند شد زیرا در هر صورت پرش انجام شده و آدرس پرش نیز در استیج IF تولید شده است. تنها دستوری که نیاز به stall دارد، دستور JR است؛ زیرا آدرس پرش در قالب دستور موجود نیست و در استیج ID تولید می‌شود. بنابراین نیاز به دو کلاک stall دارد. در این مورد هم از یک شمارنده استفاده می‌کنیم.



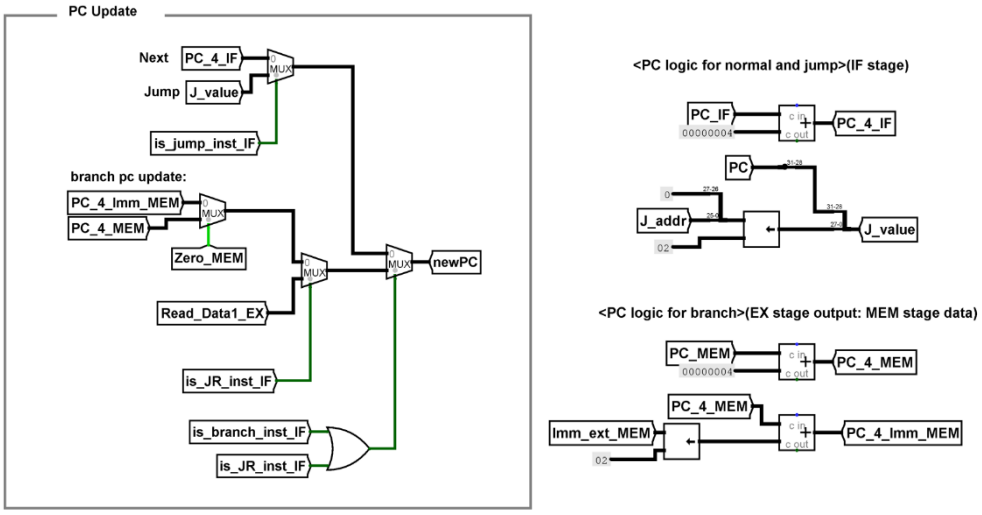
شکل ۱۰- منطق ایجاد stall برای دستور JR

در حین stall باید سیگنال write enable رجیستر PC غیر فعال باشد تا برنامه جلو نرود. علت تولید سیگنال is_branch_ins_IF_used و is_JR_ins_IF_used این است که در ابتدای کار که می‌خواهیم شروع به stall کنیم، مقدار داخل شمارنده صفر است، به همین دلیل برای غیرفعال کردن PC باید از سیگنال is_branch_inst_IF استفاده کنیم. اما این سیگنال فقط در کلاک اول قابل استفاده است زیرا در غیر این صورت، PC تا ابد غیر فعال خواهد ماند. بنابراین یک رجیستر قرار می‌دهیم که بعد از بار اول، PC دیگر از این سیگنال استفاده نکند. منطق تولید سیگنال WE رجیستر PC در شکل زیر نشان داده شده است. توجه کنید که از همین سیگنال به عنوان reset در شمارنده‌ها استفاده می‌کنیم.



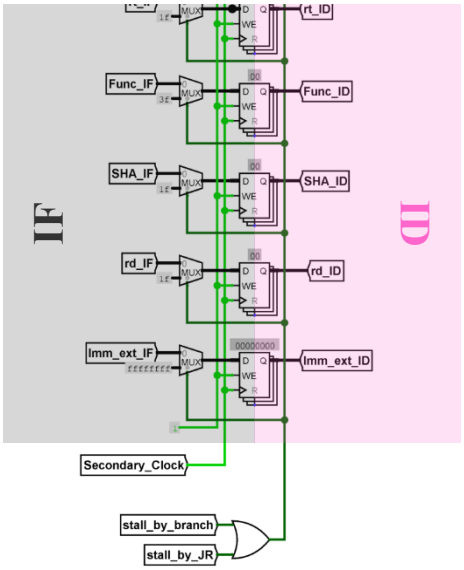
شکل ۱۱- منطق تولید سیگنال WE برای PC

مقدار جدید PC نیز توسط مدار زیر تولید می‌شود:



شکل ۱۲- تولید مقدار جدید PC

اتفاق دیگری که حین stall باید رخ دهد این است که محتوایی که وارد پایپلاین می‌شود، باید دستورات غیرقابل اجرا (nop) باشد. برای این کار در ورودی بافرهای بین استیج IF و ID مالتی‌پلکسر قرار می‌دهیم و در صورتی که در حال stall بودیم، ورودی nop به آنها می‌دهیم:



شکل ۱۳- ورودی nop در حین stall

تست دستورات پرش شرطی

برای تست عملکرد پایپلاین در دستورات پرش شرطی (branch) برنامه زیر را روی آن اجرا می‌کنیم:

```
.text
and $zero, $zero, $zero
addi $s0, $zero, 10
addi $s1, $zero, 10
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
and $t5, $t5, $t5
beq $s0, $s1, target1
addi $s5, $zero, 32
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
and $t5, $t5, $t5
target1:
beq $s0, $s2, target2
addi $s6, $zero, 64
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
and $t5, $t5, $t5
target2:
addi $s7, $zero, 128
and $t0, $t0, $t0
and $t1, $t1, $t1
and $t2, $t2, $t2
and $t3, $t3, $t3
and $t4, $t4, $t4
and $t5, $t5, $t5
```

```
00400000: 00000024 ; <input:7> and $zero, $zero, $zero
00400004: 2010000a ; <input:8> addi $s0, $zero, 10
00400008: 2011000a ; <input:9> addi $s1, $zero, 10
0040000c: 01084024 ; <input:10> and $t0, $t0, $t0
00400010: 01294824 ; <input:11> and $t1, $t1, $t1
00400014: 014a5024 ; <input:12> and $t2, $t2, $t2
00400018: 016b5824 ; <input:13> and $t3, $t3, $t3
0040001c: 018c6024 ; <input:14> and $t4, $t4, $t4
00400020: 01ad6824 ; <input:15> and $t5, $t5, $t5
00400024: 12110007 ; <input:16> beq $s0, $s1, target1
00400028: 20150020 ; <input:17> addi $s5, $zero, 32
0040002c: 01084024 ; <input:18> and $t0, $t0, $t0
00400030: 01294824 ; <input:19> and $t1, $t1, $t1
00400034: 014a5024 ; <input:20> and $t2, $t2, $t2
00400038: 016b5824 ; <input:21> and $t3, $t3, $t3
0040003c: 018c6024 ; <input:22> and $t4, $t4, $t4
00400040: 01ad6824 ; <input:23> and $t5, $t5, $t5
00400044: <target1> ; <input:24> target1:
00400044: 12120007 ; <input:25> beq $s0, $s2, target2
00400048: 20160040 ; <input:26> addi $s6, $zero, 64
0040004c: 01084024 ; <input:27> and $t0, $t0, $t0
00400050: 01294824 ; <input:28> and $t1, $t1, $t1
00400054: 014a5024 ; <input:29> and $t2, $t2, $t2
00400058: 016b5824 ; <input:30> and $t3, $t3, $t3
0040005c: 018c6024 ; <input:31> and $t4, $t4, $t4
00400060: 01ad6824 ; <input:32> and $t5, $t5, $t5
00400064: <target2> ; <input:33> target2:
00400064: 20170080 ; <input:34> addi $s7, $zero, 128
00400068: 01084024 ; <input:35> and $t0, $t0, $t0
0040006c: 01294824 ; <input:36> and $t1, $t1, $t1
00400070: 014a5024 ; <input:37> and $t2, $t2, $t2
00400074: 016b5824 ; <input:38> and $t3, $t3, $t3
00400078: 018c6024 ; <input:39> and $t4, $t4, $t4
0040007c: 01ad6824 ; <input:40> and $t5, $t5, $t5
```

شکل ۱۴- برنامه جهت تست دستورات پرش شرطی

مشاهده می‌شود که در خط ۱۱ و ۲۰ و ۲۹ سه stall وارد پایپلاین می‌شود.

تست دستورات پرش غیرشرطی

برای تست عملکرد پایپلاین در دستورات پرش غیرشرطی (jump) برنامه زیر را روی آن اجرا می‌کنیم:

```
1  .text
2  and $zero, $zero, $zero
3  addi $s0, $zero, 52
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  jr $s0
10 addi $s1, $zero, 20
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 addi $s2, $zero, 30
17 and $zero, $zero, $zero
18 and $zero, $zero, $zero
19 and $zero, $zero, $zero
20 and $zero, $zero, $zero
21 and $zero, $zero, $zero
```

```
00400000: 00000024 ; <input:7> and $zero, $zero, $zero
00400004: 20100034 ; <input:8> addi $s0, $zero, 52
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: 02000008 ; <input:14> jr $s0
00400020: 20110014 ; <input:15> addi $s1, $zero, 20
00400024: 00000024 ; <input:16> and $zero, $zero, $zero
00400028: 00000024 ; <input:17> and $zero, $zero, $zero
0040002c: 00000024 ; <input:18> and $zero, $zero, $zero
00400030: 00000024 ; <input:19> and $zero, $zero, $zero
00400034: 00000024 ; <input:20> and $zero, $zero, $zero
00400038: 2012001e ; <input:21> addi $s2, $zero, 30
0040003c: 00000024 ; <input:22> and $zero, $zero, $zero
00400040: 00000024 ; <input:23> and $zero, $zero, $zero
00400044: 00000024 ; <input:24> and $zero, $zero, $zero
00400048: 00000024 ; <input:25> and $zero, $zero, $zero
0040004c: 00000024 ; <input:26> and $zero, $zero, $zero
```

شکل ۱۵- برنامه جهت تست دستور JR

مشاهده می‌شود که در خط ۹ دو stall ایجاد می‌شود.

بررسی مخاطرات

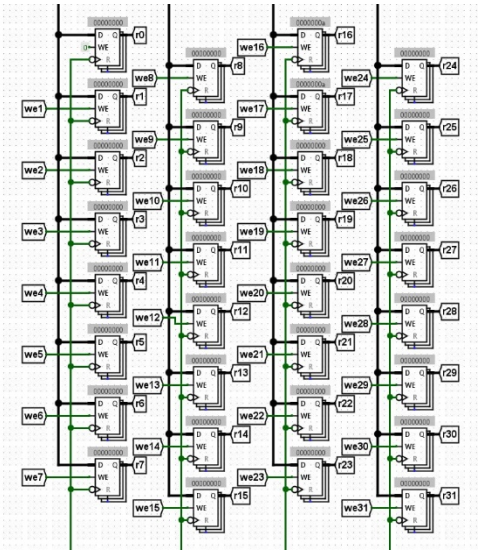
در معماری پایپلاین ممکن است مخاطراتی ایجاد شود که موجب اجرای دستورات غلط یا کاهش سرعت پایپلاین شوند. مخاطرات پایپلاین به سه دسته تقسیم می‌شوند:

۱) **مخاطرات ساختاری (Structural Hazards):** زمانی رخ می‌دهد که به صورت همزمان به یک منبع سخت‌افزاری نیاز داشته باشیم. به دلیل اینکه در پردازنده‌مان حافظه دستورات و حافظه داده از یکدیگر جدا هستند، این مخاطره رخ نمی‌دهد.

۲) **مخاطرات داده‌ای (Data Hazards):** زمانی رخ می‌دهد که بین دستورات وابستگی داده‌ای وجود داشته باشد. سه نوع مختلف مخاطره داده‌ای وجود دارد (RAW، WAR و WAW) که از بین این سه مخاطره، فقط RAW (Read After Write) در پردازنده میپس رخ می‌دهد(با استفاده از forwarding می‌توان این مخاطره را تا حدودی برطرف کرد) برای تست این مخاطره، قطعه کد زیر را روی پردازنده اجرا می‌کنیم:

```
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 ۱۶- برنامه‌ای جهت نشان دادن data hazard



شکل ۱۷- مقادیر رجیسترها بعد از اجرای برنامه شکل ۱۶

انتظار می‌رفت که پس از اجرای کد، در ثبات ۱۷ عدد ۲۰ ذخیره شده باشد اما این ثبات مقدار ۱۰ را نگه می‌دارد.

۳) **مخاطرات کنترلی (Control Hazards):** زمانی رخ می‌دهد که جریان اجرای برنامه تغییر کند. همانطور که مطرح شد، برای اینکه دستورات پرش شرطی و غیرشرطی به درستی کار کنند (دستور اشتباهی اجرا نکند) مجبور به انداختن stall شدیم که باعث کاهش سرعت پردازنده‌مان می‌شود. برای برطرف کردن این مخاطره از تکنیک‌هایی مثل early branch resolution، branch prediction و delayed branching استفاده می‌شود. در فاز بعدی با استفاده از branch prediction این مخاطره را برطرف خواهیم کرد.

منابع و مراجع

- اسلایدهای درس
- https://alanhogan.com/asu/assembler.php