به نام خدا

گزارشکار فاز دوم پروژه درس معماری کامپیوتر

طراحی و پیاده‌سازی Cache

استاد

دکتر حمید سربازی آزاد

اعضای گروه

محمدپارسا بشری ۴۰۰۱۰۴۸۱۲

محسن قاسمی ۴۰۰۱۰۵۱۶۶

امیرحسین رازلیقی ۹۹۱۰۲۴۲۳

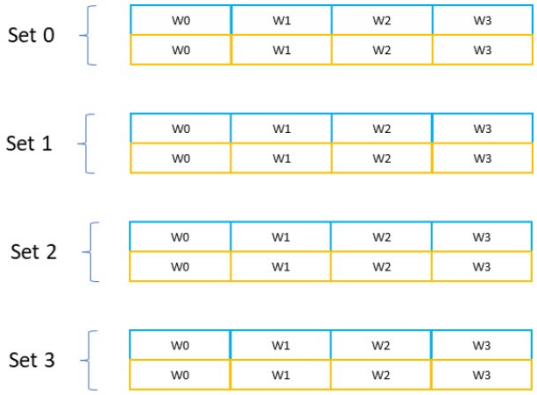
بهار ۱۴۰۲

فهرست

مقدمه و هدف فاز دوم ۲
طراحی Delayed Memory ۲
طراحی ساختار خارجی Cache ۳
طراحی ساختار داخلی Cache ۵
تست عملکرد پردازنده بعد از اضافه شدن Cache ۸
منابع و مراجع ۹

مقدمه و هدف فاز دوم

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 می‌خواهیم یک ماژول حافظه نهان (Cache) به پردازنده‌مان اضافه کنیم. در این فاز فرض می‌کنیم که مانند واقعیت، حافظه اصلی داده (Data Memory) تاخیر بیشتری نسبت به حافظه نهان دارد؛ بنابراین ماژول حافظه‌مان را با یک حافظه با تاخیر ۸ کلاک جایگزین می‌کنیم. ماژول cache شامل چهار set می‌شود که هر کدام دو block دارد. هر block نیز شامل چهار word خواهد بود. بنابراین طراحی ماژول cache به صورت 2-way set-associative است که ساختار کلی آن در شکل ۱ نشان داده شده است.

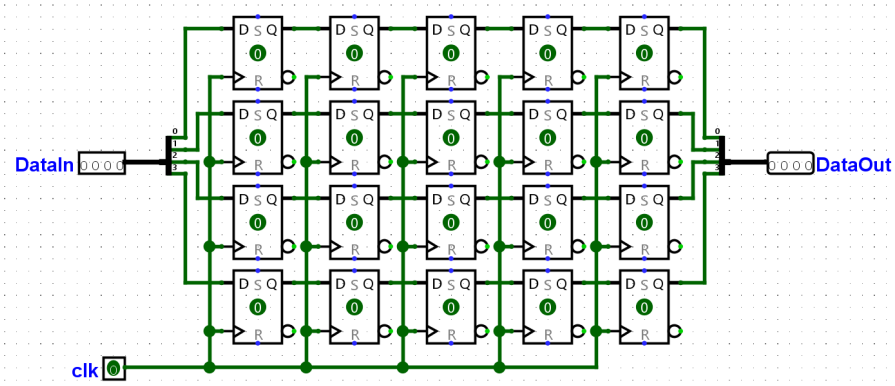


شکل ۱- ساختار کلی ماژول cache

همچنین به عنوان replacement policy از True LRU[1] استفاده کرده و به عنوان write scheme نیز از سیاست write-back استفاده خواهیم کرد.

طراحی Delayed Memory

برای طراحی ماژول Delayed Memory ابتدا یک ماژول کمکی به نام delayModule می‌سازیم که ساختاری به شکل زیر دارد:



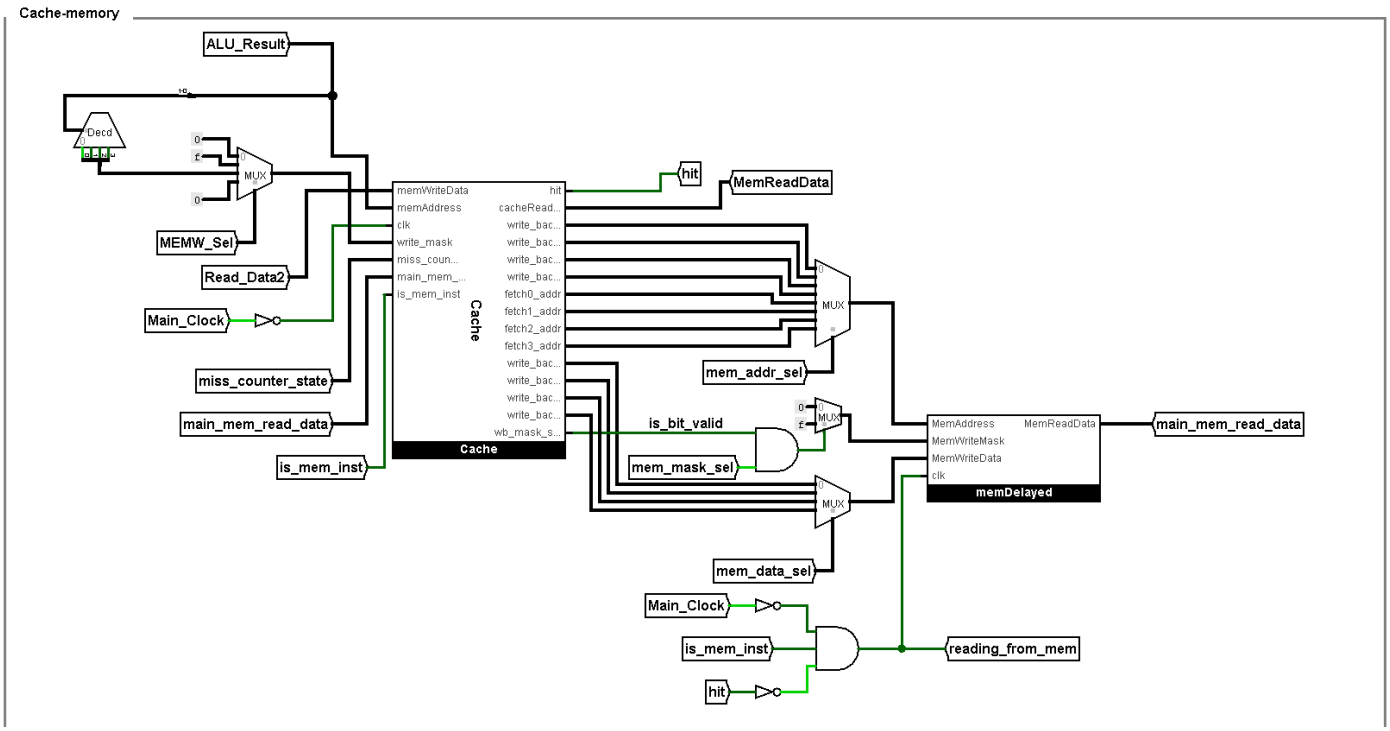
شکل ۲- طراحی داخلی ماژول کمکی delayModule

ماژول کمکی delayModuleOut نیز دقیقا مانند delayModule طراحی می‌شود، با این تفاوت که فقط ۴ کلاک تاخیر دارد. حالا به کمک این ماژول‌ها، Delayed Memory را می‌سازیم. به این صورت که جلوی هر ورودی یک delayModule و پشت هر خروجی یک delayModuleOut قرار می‌دهیم. بنابراین ۸ کلاک بعد از زمانی که آدرس را به حافظه می‌دهیم، خروجی آماده استفاده است.

[1] Least Recently Used Policy

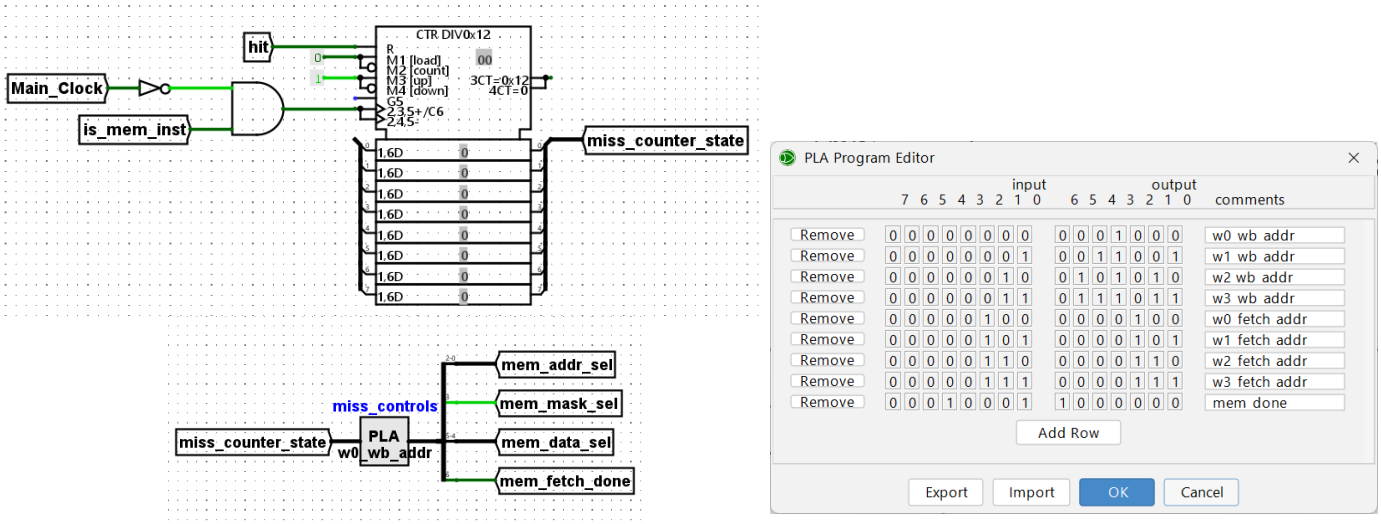
طراحی ساختار خارجی Cache

حافظه cache باید بین پردازنده و حافظه اصلی قرار بگیرد؛ به این صورت که همه‌ی ورودی‌ها و خروجی‌هایی که قبلا به حافظه اصلی متصل بودند، حالا باید به ماژول cache وصل شوند. همچنین تعدادی ورودی و خروجی برای ارتباط بین حافظه اصلی و cache باید وجود داشته باشد. در ضمن طراحی به صورتی است که کلاک memory فقط وقتی فعال می‌شود که دستور از نوع حافظه‌ای باشد و miss رخ داده باشد. رابطه بین cache و memory در شکل زیر آورده شده است:



شکل ۳- طراحی خارجی cache و رابطه آن با حافظه اصلی

همانطور که در شکل مشخص است، تعدادی سیگنال کنترلی مانند mem_addr_sel، mem_mask_sel و mem_data_sel وجود دارد که مشخص می‌کنند که کدام سیگنال‌های خروجی از cache به ورودی memory وصل شوند. توجه کنید که همه ارتباط بین cache و memory زمانی اتفاق می‌افتد که miss رخ بدهد. با رخ دادن miss، یک counter شروع به شمردن می‌کند و در هر state ترکیبی از سیگنال‌های کنترلی فعال شده و فرایند جابجایی اطلاعات بین memory و cache (شامل آوردن block جدید و فرایند write-back) طی ۱۸ کلاک انجام می‌شود. شمارنده و منطق بدست آوردن سیگنال‌های کنترلی از روی خروجی شمارنده در شکل زیر آورده شده است:



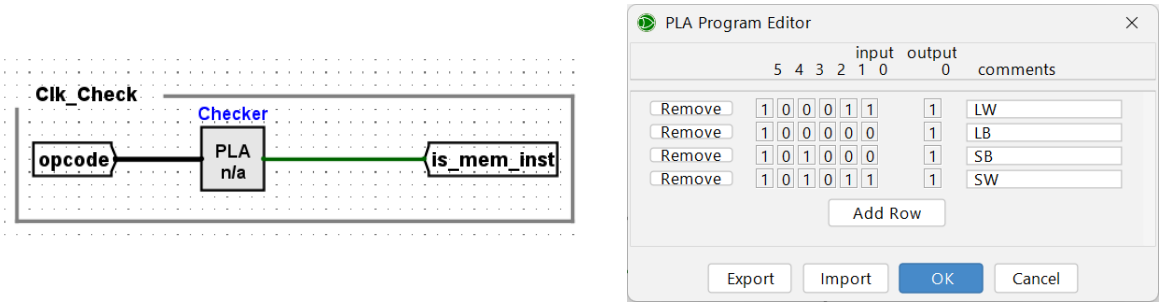
شکل ۴- شمارنده miss و تولید سیگنال‌های کنترلی

عملیاتی که در هر کلاک از فرایند miss انجام می‌شود در جدول زیر آورده شده است:

شماره کلاک (miss_counter_state)	عملیات
۰	آدرس و دیتای write-back اولین کلمه از بلاک جایگزین شده به مموری داده می‌شود
۱	آدرس و دیتای write-back دومین کلمه از بلاک جایگزین شده به مموری داده می‌شود
۲	آدرس و دیتای write-back سومین کلمه از بلاک جایگزین شده به مموری داده می‌شود
۳	آدرس و دیتای write-back چهارمین کلمه از بلاک جایگزین شده به مموری داده می‌شود
۴	آدرس اولین کلمه بلاک برای fetch به مموری داده می‌شود
۵	آدرس دومین کلمه بلاک برای fetch به مموری داده می‌شود + اولین کلمه از write-back در مموری نوشته می‌شود
۶	آدرس سومین کلمه بلاک برای fetch به مموری داده می‌شود + دومین کلمه از write-back در مموری نوشته می‌شود
۷	آدرس چهارمین کلمه بلاک برای fetch به مموری داده می‌شود + سومین کلمه از write-back در مموری نوشته می‌شود
۸	چهارمین کلمه از write-back در مموری نوشته می‌شود
۹	انتظار
۱۰	انتظار
۱۱	انتظار
۱۲	انتظار
۱۳	انتظار
۱۴	دیتای اولین کلمه fetch شده در کش نوشته می‌شود
۱۵	دیتای دومین کلمه fetch شده در کش نوشته می‌شود
۱۶	دیتای سومین کلمه fetch شده در کش نوشته می‌شود
۱۷	دیتای چهارمین کلمه fetch شده در کش نوشته می‌شود + مقدار LRU دو بلاک set آپدیت و valid bit برابر یک خواهد شد

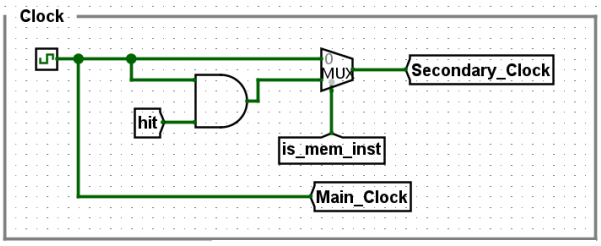
جدول 1 - مراحل فرایند miss

سیگنال is_mem_inst به سادگی نشان می‌دهد که دستورمان از دستورات حافظه هست یا خیر. برای ساختن این سیگنال از یک PLA استفاده می‌کنیم.



شکل ۵- ساخت سیگنال is_mem_inst

همانطور که می‌دانیم هنگام رخ دادن miss باید کلاک بقیه اجزای پردازنده (به طور خاص PC، instruction memory و register file) متوقف شود. برای این کار، با استفاده از یک مالتی‌پلکسر یک کلاک فرعی می‌سازیم و آن را به بقیه اجزای پردازنده می‌دهیم:



شکل ۶- ساخت کلاک فرعی

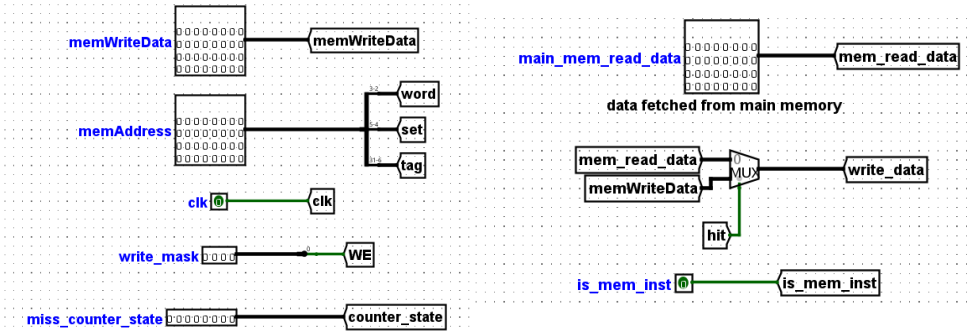
طراحی ساختار داخلی Cache

با توجه به اندازه حافظه اصلی و ساختار cache، آدرسی که در پردازنده تولید می‌شود به شکل زیر تقسیم می‌شود:

Tag (26 bits)	Set (2 bits)	Word (2 bits)	Byte (2 bits)

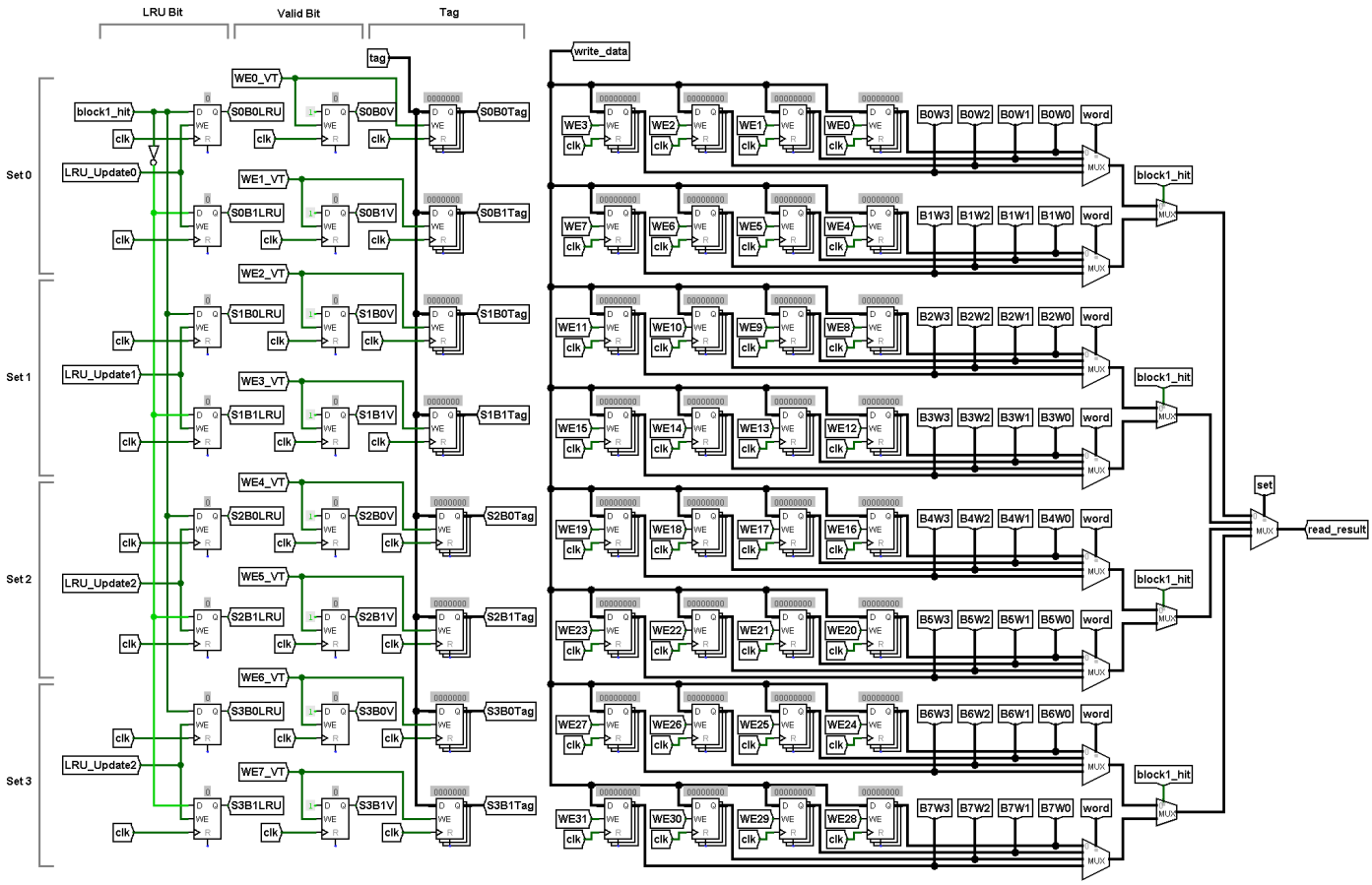
شکل 7 - آدرس تولید شده توسط پردازنده

با توجه به شکل ۷، پس از دریافت آدرس به عنوان ورودی، آن را به سه بخش word، set و tag تقسیم می‌کنیم. تصویر زیر، ورودی‌های حافظه نهان را نشان می‌دهد:



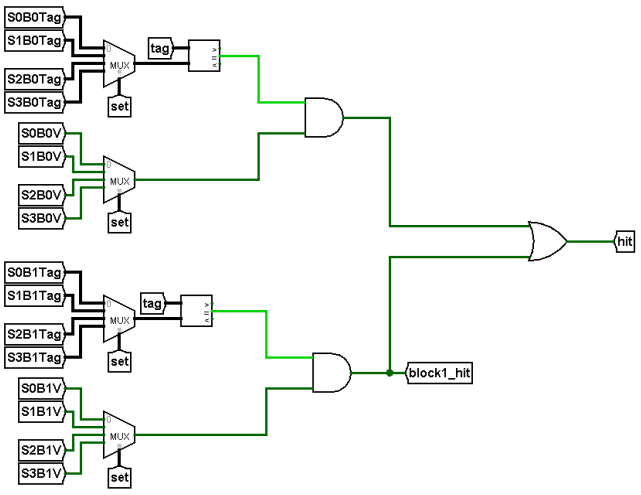
شکل ۸- ورودی‌های cache

بدنه اصلی حافظه cache با استفاده از تعدادی رجیستر طراحی می‌شود. این بدنه شامل ۸ بلاک است که هر block علاوه بر ۴ رجیستر ۳۲ بیتی (چهار کلمه)، شامل دو رجیستر ۱ بیتی (Valid bit و LRU bit) و یک رجیستر ۲۶ بیتی (Tag) خواهد بود.



شکل ۹- بدنه اصلی cache

برای تشخیص و تولید سیگنال hit باید از tag و valid bit بلاک‌های داخل set مربوطه استفاده کنیم. با استفاده از مدار زیر، سیگنال hit را تولید می‌کنیم:



شکل ۱۰- منطق تولید سیگنال hit

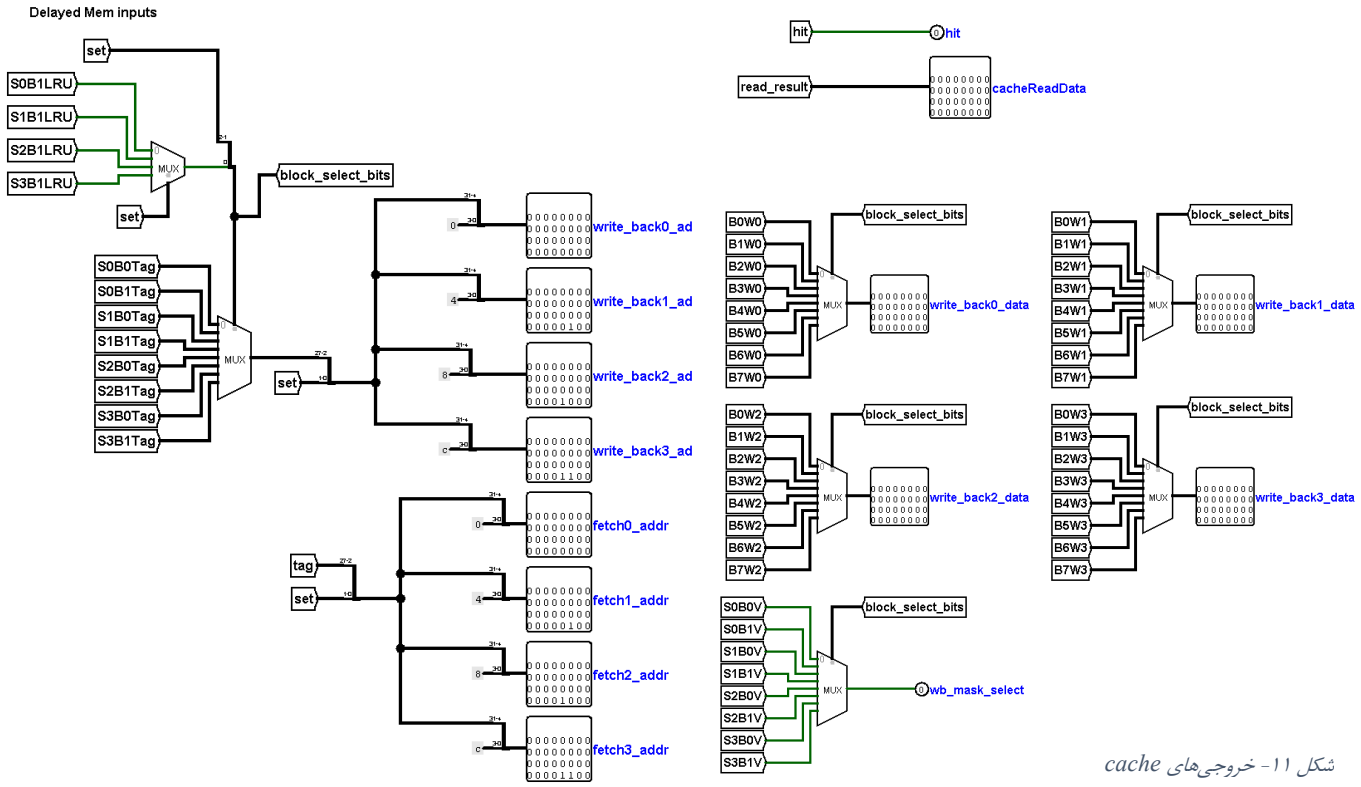
حالا چهار حالت مختلف را بررسی می‌کنیم:

۱) **hit در خواندن**: همانطور که در شکل ۹ مشخص است، کلمه مورد نظر بدون تاخیر انتخاب شده و به عنوان خروجی داده می‌شود.

۲) **hit در نوشتن**: باید سیگنال WE کلمه مورد نظر در cache فعال شود (مطابق شکل ۱۲). دیتایی که باید نوشته شود همانطور که در شکل ۸ مشخص است، بر اساس hit شدن یا نشدن انتخاب می‌شود.

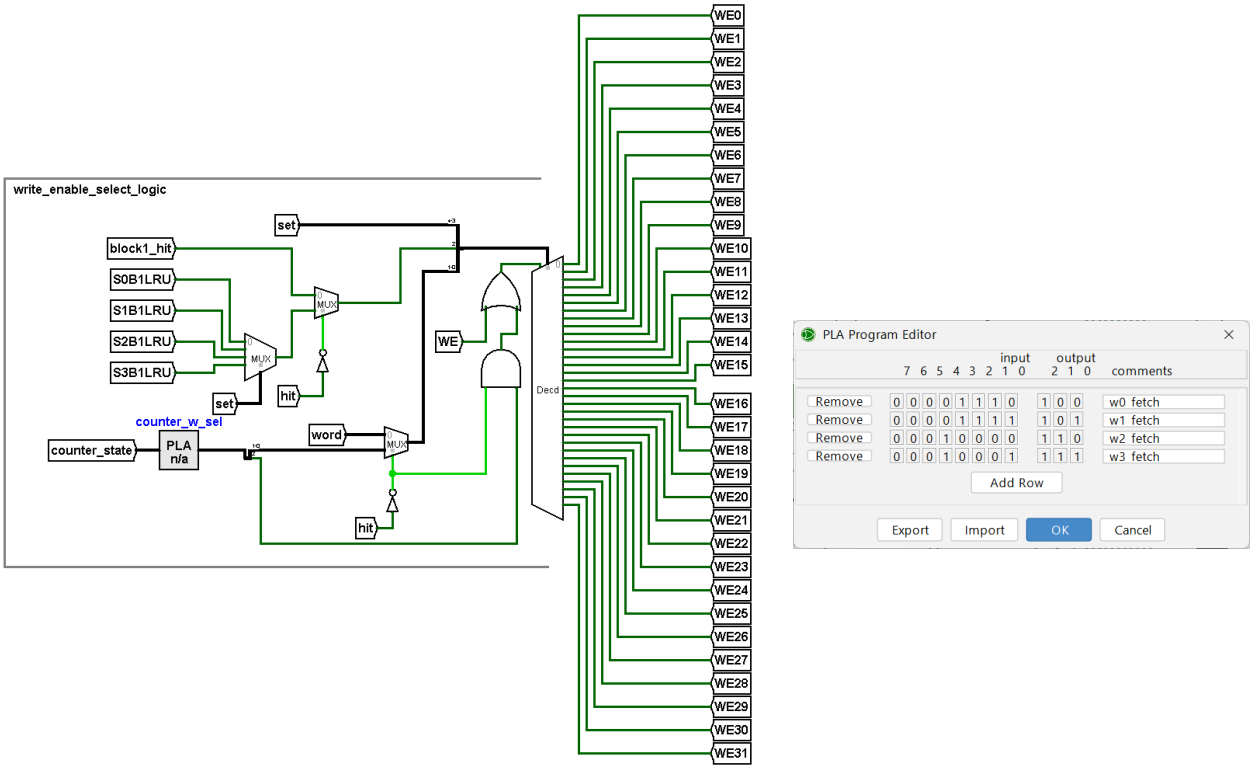
۳) **miss در نوشتن**: فرایند آوردن بلاک از حافظه اصلی، دقیقا مشابه حالتی که در خواندن miss رخ دهد اتفاق می‌افتد و بعد از اینکه بلاک در cache نوشته شد، مانند این رفتار می‌کنیم که هنگام نوشتن hit شده و سیگنال WE کلمه مورد نظر فعال می‌شود.

۴) **miss در خواندن**: در این حالت باید ۱۸ کلاک را مطابق جدول ۱ طی کنیم. از کلاک ۰ تا ۸ همانطور که در شکل ۳ و ۴ شرح دادیم، در خارج از cache اتفاق می‌افتد و ما در داخل cache کافی‌است همه‌ی خروجی‌های مورد نیاز را تولید کنیم تا در خارج cache با توجه شکل ۳ از بین آنها انتخاب شود. در شکل زیر خروجی‌های cache را نمایش می‌دهیم:



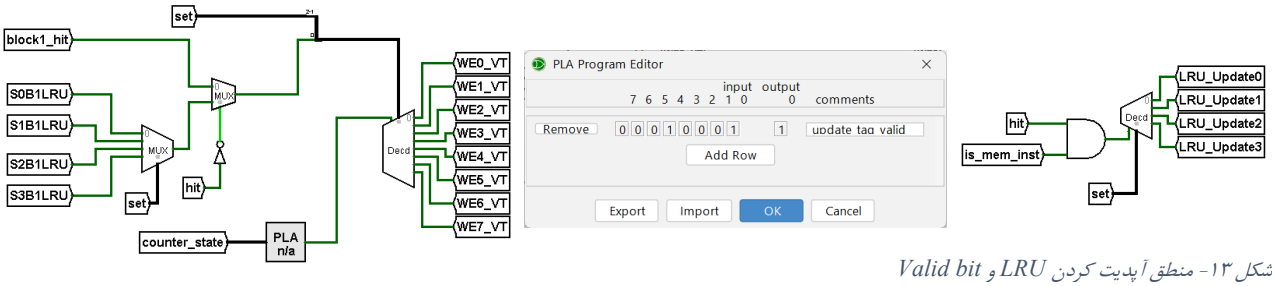
شکل ۱۱- خروجی‌های cache

اما کلاک‌های ۱۴ تا ۱۷ در جدول ۱ را باید در داخل cache پیاده‌سازی کنیم. برای نوشتن دیتای fetch شده در کلاک‌های ۱۴ تا ۱۷ از مدار زیر استفاده می‌کنیم که سیگنال‌های WE هر کلمه در cache را کنترل می‌کند:



شکل ۱۲- منطق کنترل ورودی‌های WE کلمات cache

برای آپدیت کردن مقدار LRU و Valid bit در کلاک آخر از مدار زیر استفاده می‌کنیم (نحوه اتصال این سیگنال‌ها در شکل ۹ نشان داده شده است):



شکل ۱۳- منطق آپدیت کردن LRU و Valid bit

یادآوری: توجه کنید که سیاست LRU در حالت 2-way set-associative به این صورت عمل می‌کند که هنگام eviction، LRU bit یکی از بلاک‌ها صفر و دیگری یک است و بلاکی که LRU bit یک داشته باشد جایگزین شده و سپس جای صفر و یک بلاک‌ها عوض می‌شود. دلیل ورودی‌های مالتی‌پلکسرهای شکل‌های ۱۱، ۱۲ و ۱۳ همین است.

تست عملکرد پردازنده بعد از اضافه شدن Cache

مانند فاز یک، برای تست عملکرد نهایی پردازنده بعد از اضافه شدن cache، باید یک برنامه را با آن اجرا کنیم. فرض کنید می‌خواهیم کد زیر را اجرا کنیم (فایل‌های تست این قسمت در `utils/cache-test/samples` قرار دارد):

```
.text
.globl main

addi $s0, $zero, 1
addi $s0, $s0, 1
lw $t0, 0($zero)
lw $t0, 0($zero)
addi $s0, $zero, 1
sw $s0, 0($zero)
addi $s0, $zero, 2
sw $s0, 4($zero)
addi $s0, $zero, 3
sw $s0, 8($zero)
addi $s0, $zero, 4
sw $s0, 12($zero)
lw $t0, 0($zero)
lw $t1, 4($zero)
lw $t2, 8($zero)
lw $t3, 12($zero)

add $s0, $zero, $zero
add $s0, $zero, $zero

lw $t0, 64($zero)
lw $t0, 64($zero)
addi $s0, $zero, 11
sw $s0, 64($zero)
addi $s0, $zero, 12
sw $s0, 68($zero)
addi $s0, $zero, 13
sw $s0, 72($zero)
addi $s0, $zero, 14
sw $s0, 76($zero)
lw $t0, 64($zero)
lw $t1, 68($zero)
lw $t2, 72($zero)
lw $t3, 76($zero)

add $s0, $zero, $zero
add $s0, $zero, $zero

lw $t0, 128($zero)
lw $t0, 128($zero)
```

شکل ۱۴- قطعه کد استفاده شده در تست عملکرد پردازنده

ابتدا با استفاده از سایت https://alanhogan.com/asu/assembler.php که یک اسمبلر آنلاین است، کد ماشین را تولید می‌کنیم:

```
00400000: 20100001 ; <input:8> addi $s0, $zero, 1
00400004: 22100001 ; <input:9> addi $s0, $s0, 1
00400008: 8c080000 ; <input:10> lw $t0, 0($zero)
0040000c: 8c080000 ; <input:11> lw $t0, 0($zero)
00400010: 20100001 ; <input:12> addi $s0, $zero, 1
00400014: ac100000 ; <input:13> sw $s0, 0($zero)
00400018: 20100002 ; <input:14> addi $s0, $zero, 2
0040001c: ac100004 ; <input:15> sw $s0, 4($zero)
00400020: 20100003 ; <input:16> addi $s0, $zero, 3
00400024: ac100008 ; <input:17> sw $s0, 8($zero)
00400028: 20100004 ; <input:18> addi $s0, $zero, 4
0040002c: ac10000c ; <input:19> sw $s0, 12($zero)
00400030: 8c080000 ; <input:20> lw $t0, 0($zero)
00400034: 8c090004 ; <input:21> lw $t1, 4($zero)
00400038: 8c0a0008 ; <input:22> lw $t2, 8($zero)
0040003c: 8c0b000c ; <input:23> lw $t3, 12($zero)
00400040: 00008020 ; <input:25> add $s0, $zero, $zero
00400044: 00008020 ; <input:26> add $s0, $zero, $zero
00400048: 8c080040 ; <input:28> lw $t0, 64($zero)
0040004c: 8c080040 ; <input:29> lw $t0, 64($zero)
00400050: 2010000b ; <input:30> addi $s0, $zero, 11
00400054: ac100040 ; <input:31> sw $s0, 64($zero)
00400058: 2010000c ; <input:32> addi $s0, $zero, 12
0040005c: ac100044 ; <input:33> sw $s0, 68($zero)
00400060: 2010000d ; <input:34> addi $s0, $zero, 13
00400064: ac100048 ; <input:35> sw $s0, 72($zero)
00400068: 2010000e ; <input:36> addi $s0, $zero, 14
0040006c: ac10004c ; <input:37> sw $s0, 76($zero)
00400070: 8c080040 ; <input:38> lw $t0, 64($zero)
00400074: 8c090044 ; <input:39> lw $t1, 68($zero)
00400078: 8c0a0048 ; <input:40> lw $t2, 72($zero)
0040007c: 8c0b004c ; <input:41> lw $t3, 76($zero)
00400080: 00008020 ; <input:43> add $s0, $zero, $zero
00400084: 00008020 ; <input:44> add $s0, $zero, $zero
00400088: 8c080080 ; <input:46> lw $t0, 128($zero)
0040008c: 8c080080 ; <input:47> lw $t0, 128($zero)
```

شکل ۱۵- خروجی اسمبلر

بقیه فرایند دقیقا مشابه تست فاز یک است: سپس با استفاده از دستور `cat code.txt | awk '{print $2}' | tail -n +2` کد باینری را از فایل جدا می‌کنیم. خروجی این دستور را در یک فایل ذخیره می‌کنیم (`binary.txt`) و سپس اسکریپت پایتونی که نوشتیم را اجرا می‌کنیم. این قطعه کد، بایت‌های هر دستور را جدا کرده، چهار فایل ایجاد می‌کند و هر بایت از هر دستور را در یک فایل می‌نویسد (`insMem0.txt`، … ، `insMem3.txt`).

در نهایت با راست کلیک روی هر ماژول حافظه (instruction memory) و انتخاب گزینه load image فایل مربوط به آن ماژول را در آن بارگذاری می‌کنیم. حالا با هر بار زدن کلاک، یک دستور اجرا می‌شود. نتیجه اجرای دستورات را می‌توانیم با توجه به تاثیرشان روی یکی از ماژول‌های PC، Data Memory و یا Register File ببینیم. با توجه به طولانی بودن این فرآیند و تعداد بالای screenshot مورد نیاز، از آوردن نتایج در این گزارش صرف نظر می‌کنیم.

منابع و مراجع

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