**به نام خدا**

گزارشکار فاز دوم پروژه درس معماری کامپیوتر

**طراحی و پیاده­سازی Cache**

استاد

**دکتر حمید سربازی آزاد**

اعضای گروه

**محمدپارسا بشری 400104812**

**محسن قاسمی 400105166**

**امیرحسین رازلیقی 99102423**

بهار 1402

فهرست

[مقدمه و هدف فاز دوم 2](#_Toc138341949)

[طراحی Delayed Memory 2](#_Toc138341950)

[طراحی ساختار خارجی Cache 3](#_Toc138341951)

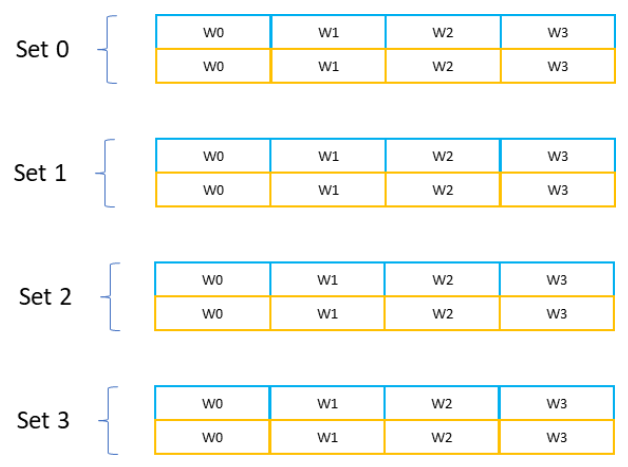
[تست عملکرد 4](#_Toc138341952)

[تست عملکرد اجزای مدار 5](#_Toc138341953)

[تست عملکرد نهایی پردازنده 6](#_Toc138341954)

[منابع و مراجع 7](#_Toc138341955)

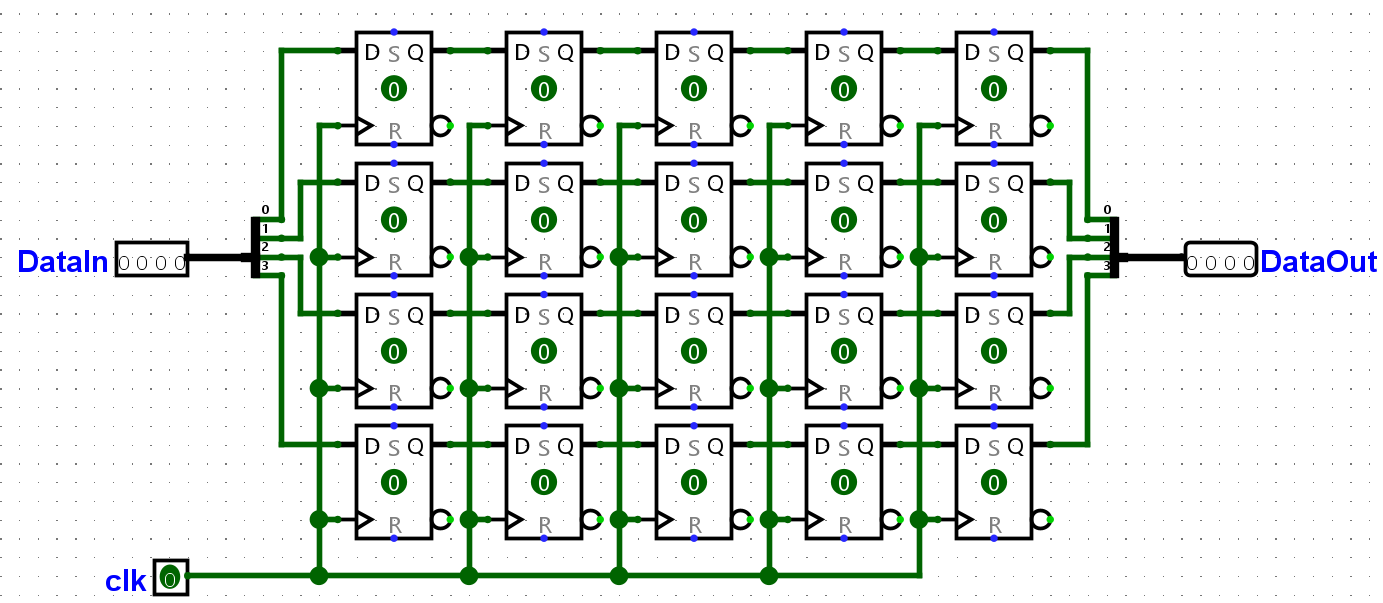
# مقدمه و هدف فاز دوم

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 8 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 1 نشان داده شده است.

شکل 1- ساختار کلی ماژول cache

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

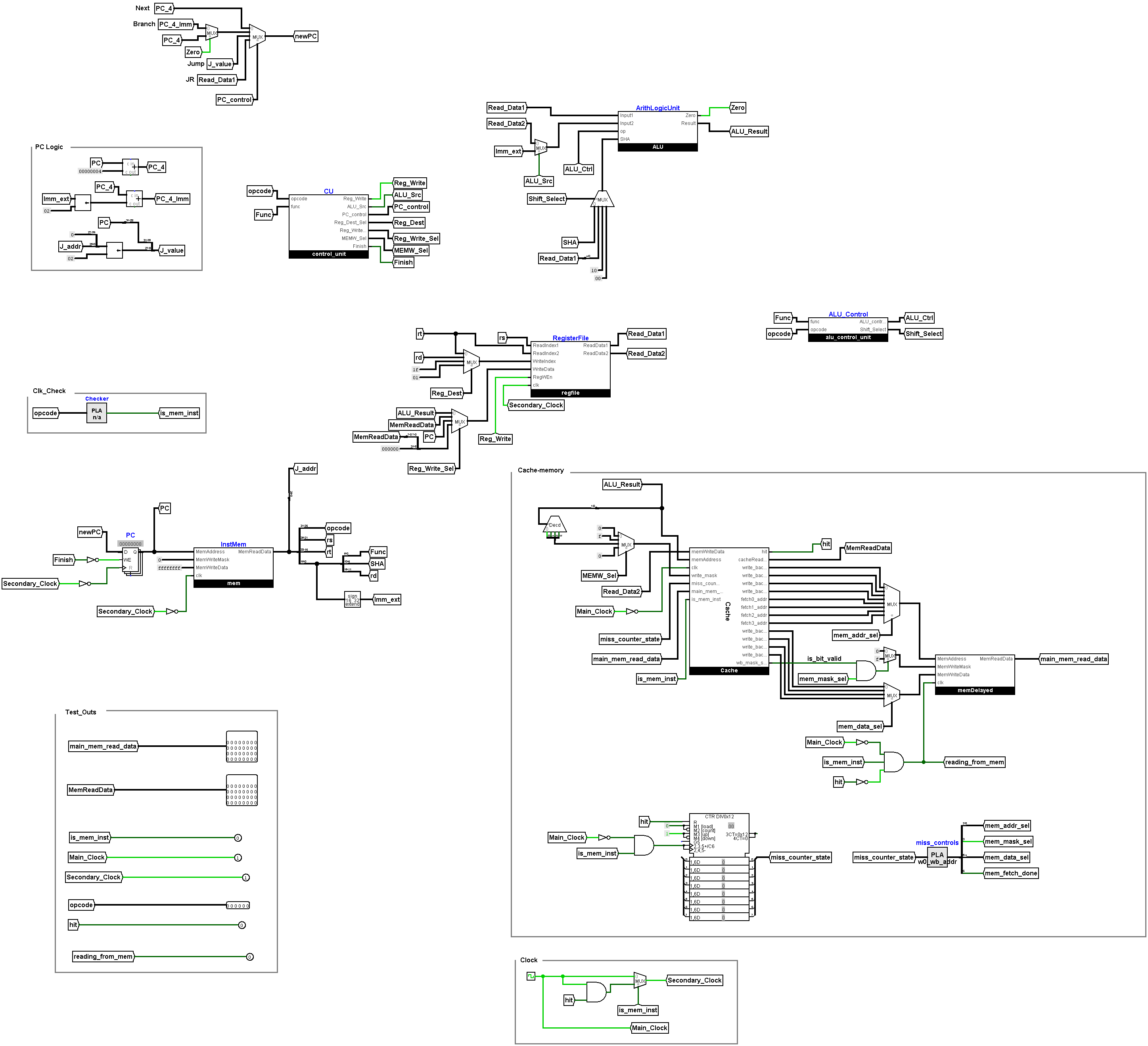
# طراحی Delayed Memory

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

شکل 2- طراحی داخلی ماژول کمکی delayModule

ماژول کمکی delayModuleOut نیز دقیقا مانند delayModule طراحی می­شود، با این تفاوت که فقط 4 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 8 کلاک بعد از زمانی که آدرس را به حافظه می­دهیم، خروجی آماده استفاده است.

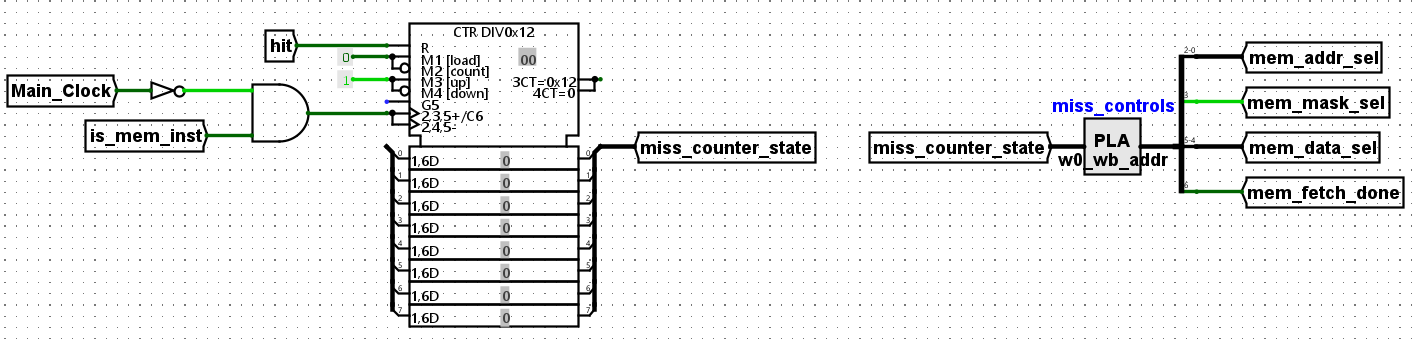
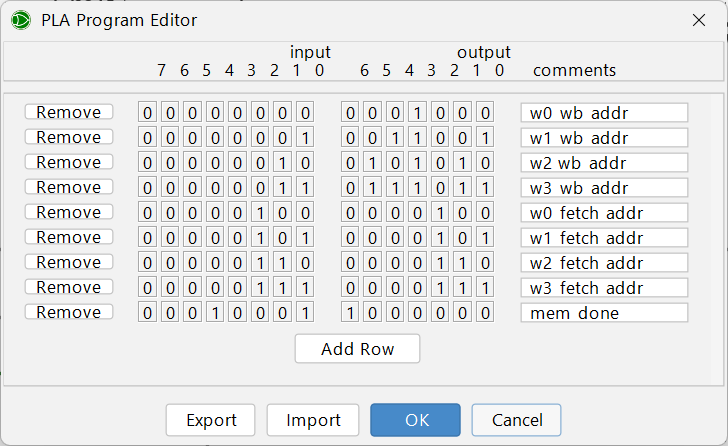
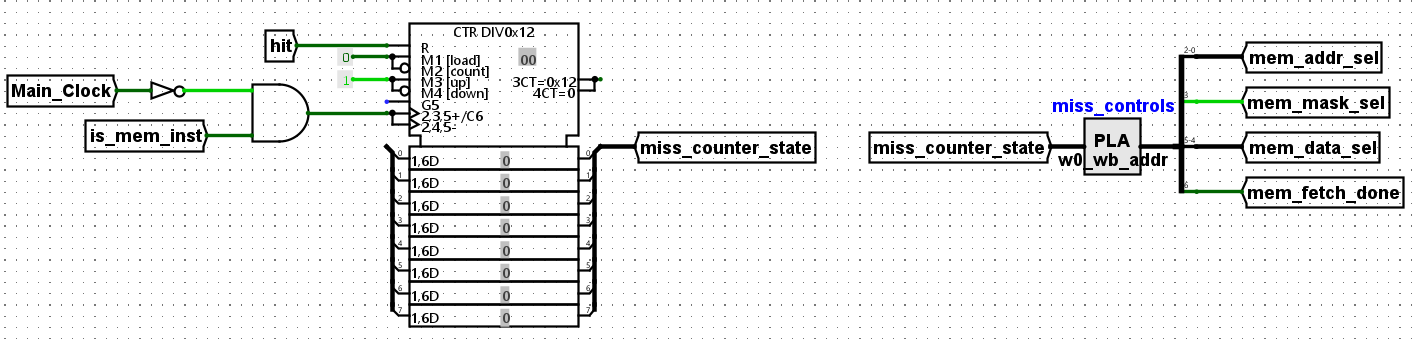
# طراحی ساختار خارجی Cache

 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 3- 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 18 کلاک انجام می­شود. شمارنده و منطق بدست آوردن سیگنال­های کنترلی از روی خروجی شمارنده در شکل زیر آورده شده است:

شکل 4- شمارنده miss و تولید سیگنال­های کنترلی



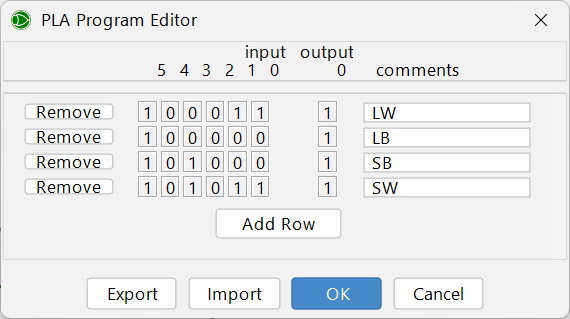
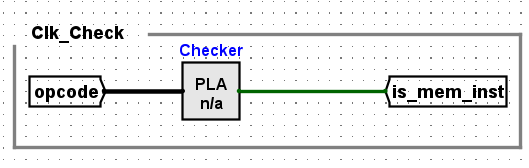
عملیاتی که در هر کلاک از فرایند miss انجام می­شود در جدول زیر آورده شده است:

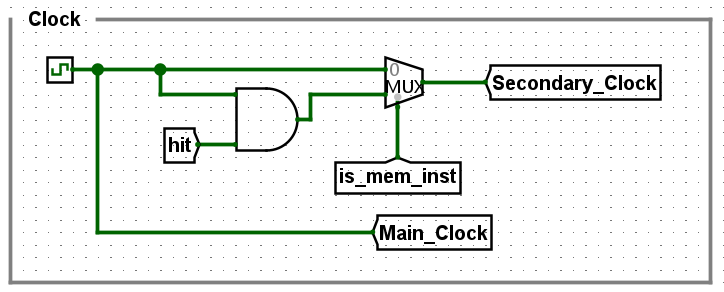
|  |  |
| --- | --- |
| شماره کلاک (miss\_counter\_state) | عملیات |
| 0 | آدرس و دیتای write-back اولین کلمه از بلاک جایگزین شده به مموری داده می‌شود |
| 1 | آدرس و دیتای write-back دومین کلمه از بلاک جایگزین شده به مموری داده می‌شود |
| 2 | آدرس و دیتای write-back سومین کلمه از بلاک جایگزین شده به مموری داده می‌شود |
| 3 | آدرس و دیتای write-back چهارمین کلمه از بلاک جایگزین شده به مموری داده می‌شود |
| 4 | آدرس اولین کلمه بلاک برای fetch به مموری داده می‌شود |
| 5 | آدرس دومین کلمه بلاک برای fetch به مموری داده می‌شود + اولین کلمه از write-back در مموری نوشته می‌شود |
| 6 | آدرس سومین کلمه بلاک برای fetch به مموری داده می‌شود + دومین کلمه از write-back در مموری نوشته می‌شود |
| 7 | آدرس چهارمین کلمه بلاک برای fetch به مموری داده می‌شود + سومین کلمه از write-back در مموری نوشته می‌شود |
| 8 | چهارمین کلمه از write-back در مموری نوشته می‌شود |
| 9 | انتظار |
| 10 | انتظار |
| 11 | انتظار |
| 12 | انتظار |
| 13 | انتظار |
| 14 | دیتای اولین کلمه fetch شده در کش نوشته می‌شود |
| 15 | دیتای دومین کلمه fetch شده در کش نوشته می‌شود |
| 16 | دیتای سومین کلمه fetch شده در کش نوشته می‌شود |
| 17 | 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 5- 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 6- ساخت کلاک فرعی

# طراحی ساختار داخلی Cache

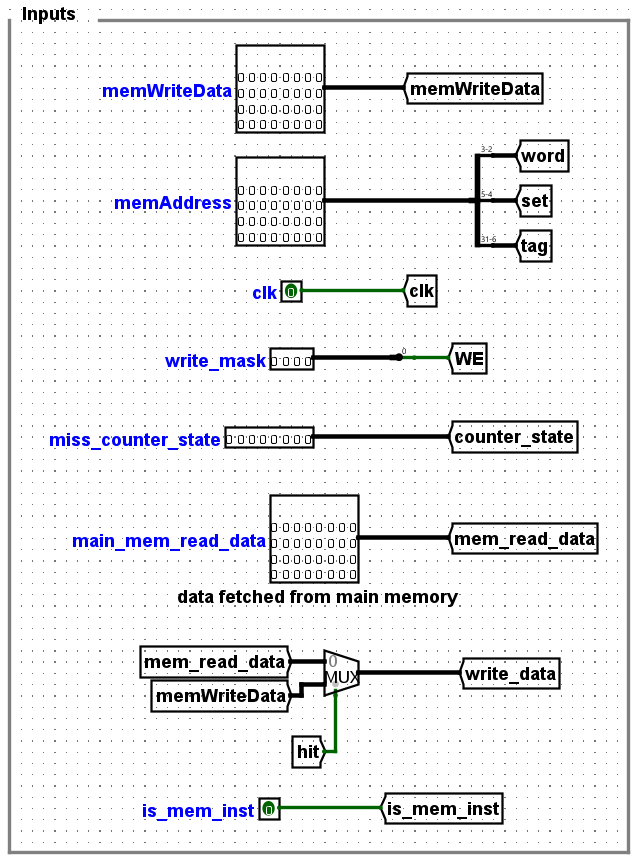
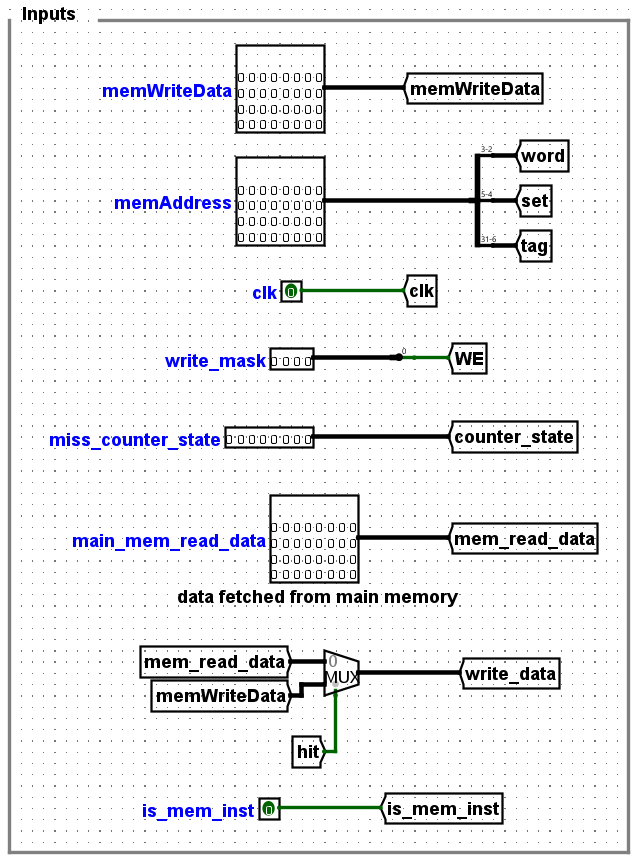
با توجه به اندازه حافظه اصلی و ساختار cache، آدرسی که در پردازنده تولید می­شود به شکل زیر تقسیم می­شود:

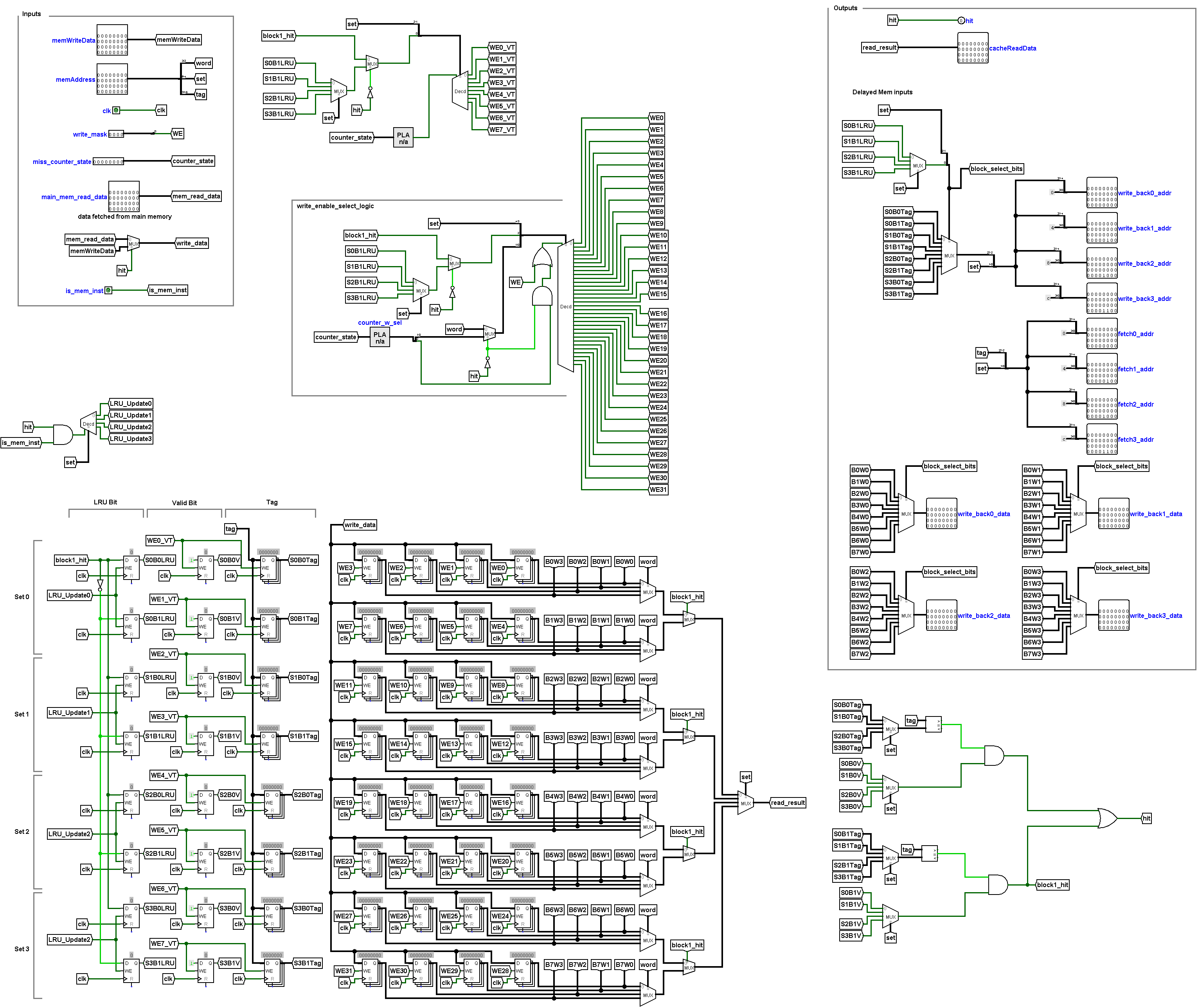
|  |  |  |  |
| --- | --- | --- | --- |
| Byte (2 bits) | Word (2 bits) | Set (2 bits) | Tag (26 bits) |

شکل 7- آدرس تولید شده توسط پردازنده

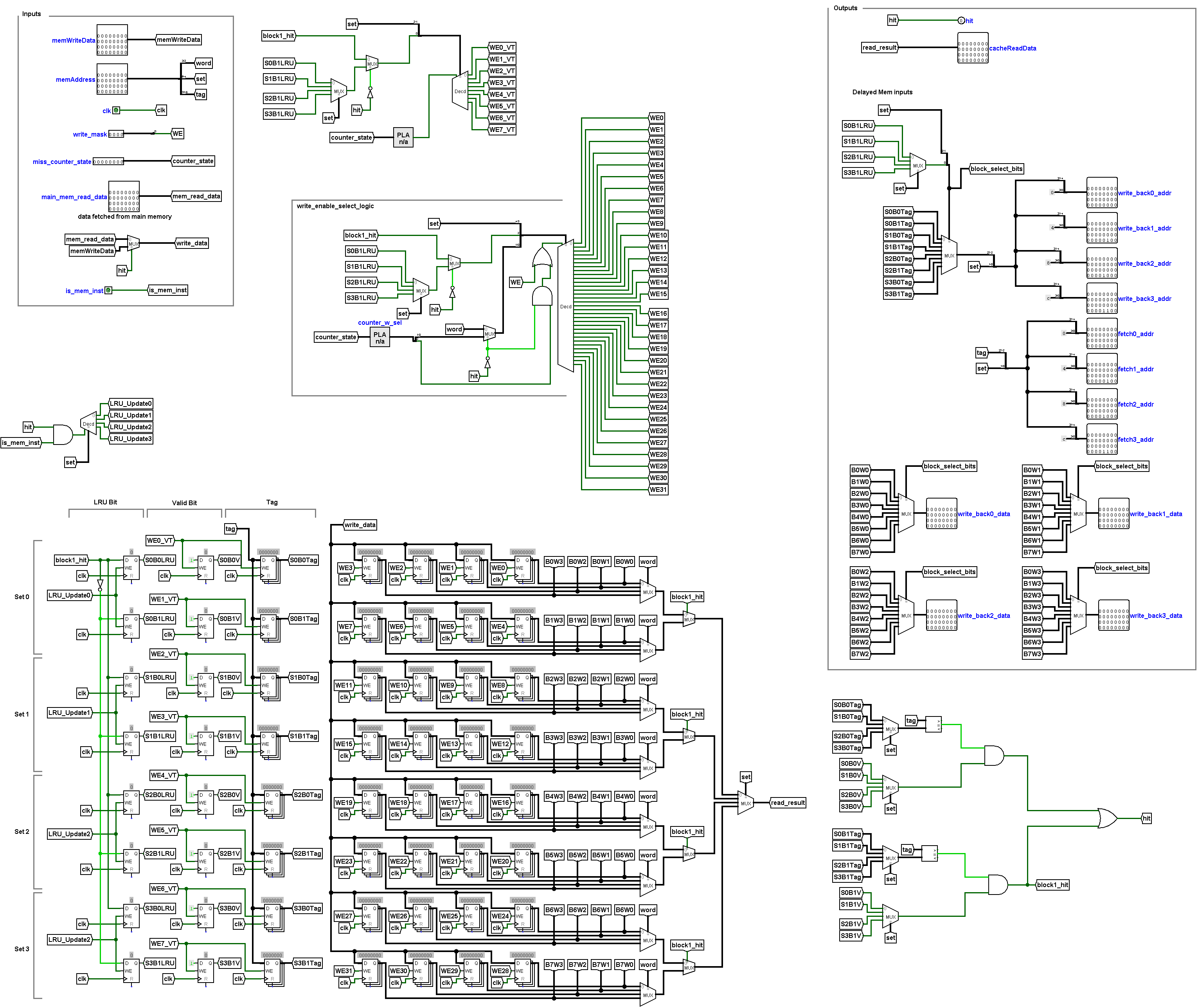
با توجه به شکل 7، پس از دریافت آدرس به عنوان ورودی، آن را به سه بخش word، set و tag تقسیم می­کنیم. تصویر زیر، ورودی­های حافظه نهان را نشان می­دهد:

شکل 8- ورودی­های cache



بدنه اصلی حافظه cache با استفاده از تعدادی رجیستر طراحی می­شود. این بدنه شامل 8 بلاک است که هر block علاوه بر 4 رجیستر 32 بیتی (چهار کلمه)، شامل دو رجیستر 1 بیتی (Valid bit و LRU bit) و یک رجیستر 26 بیتی (Tag) خواهد بود.

شکل 9- بدنه اصلی cache

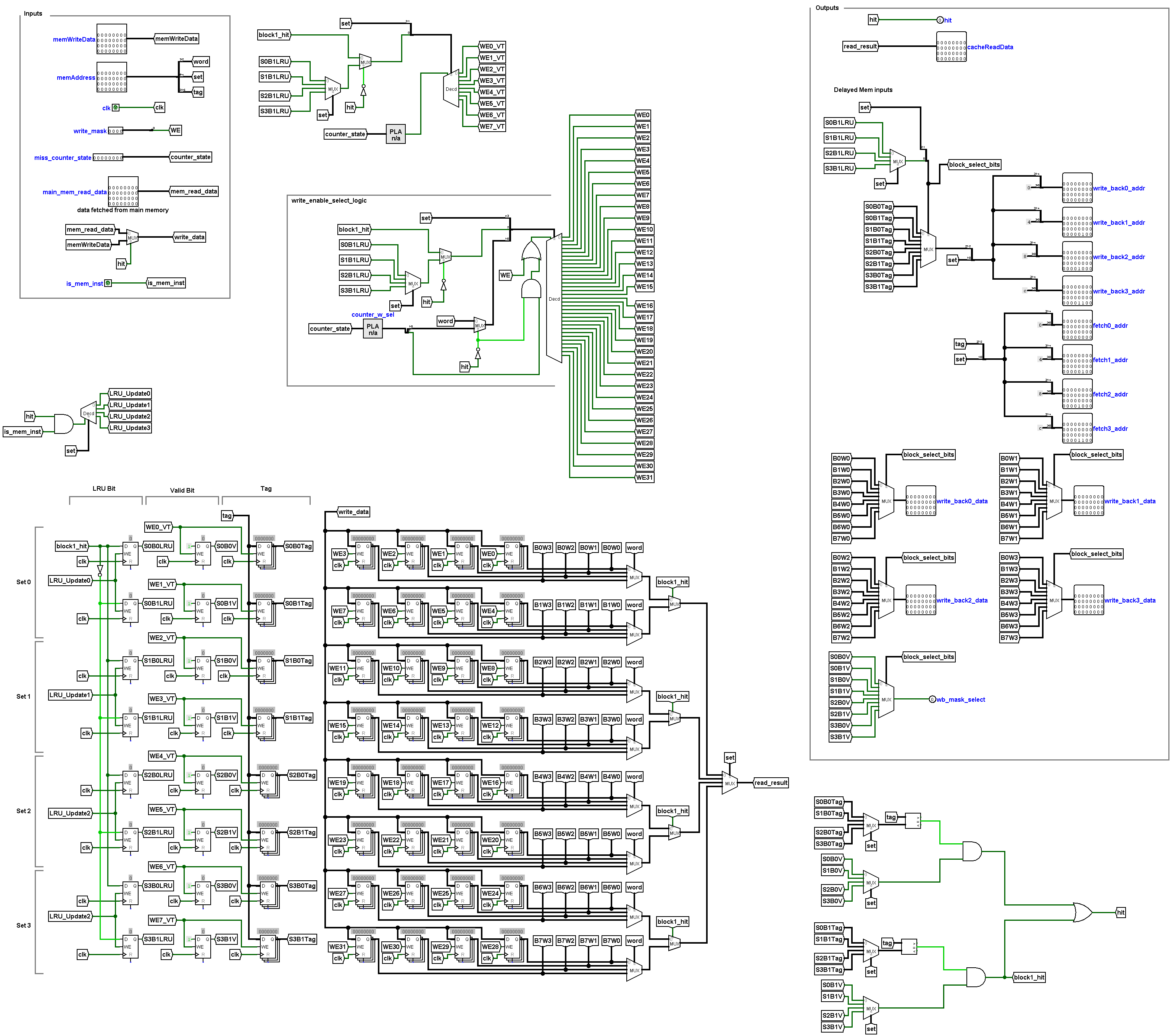
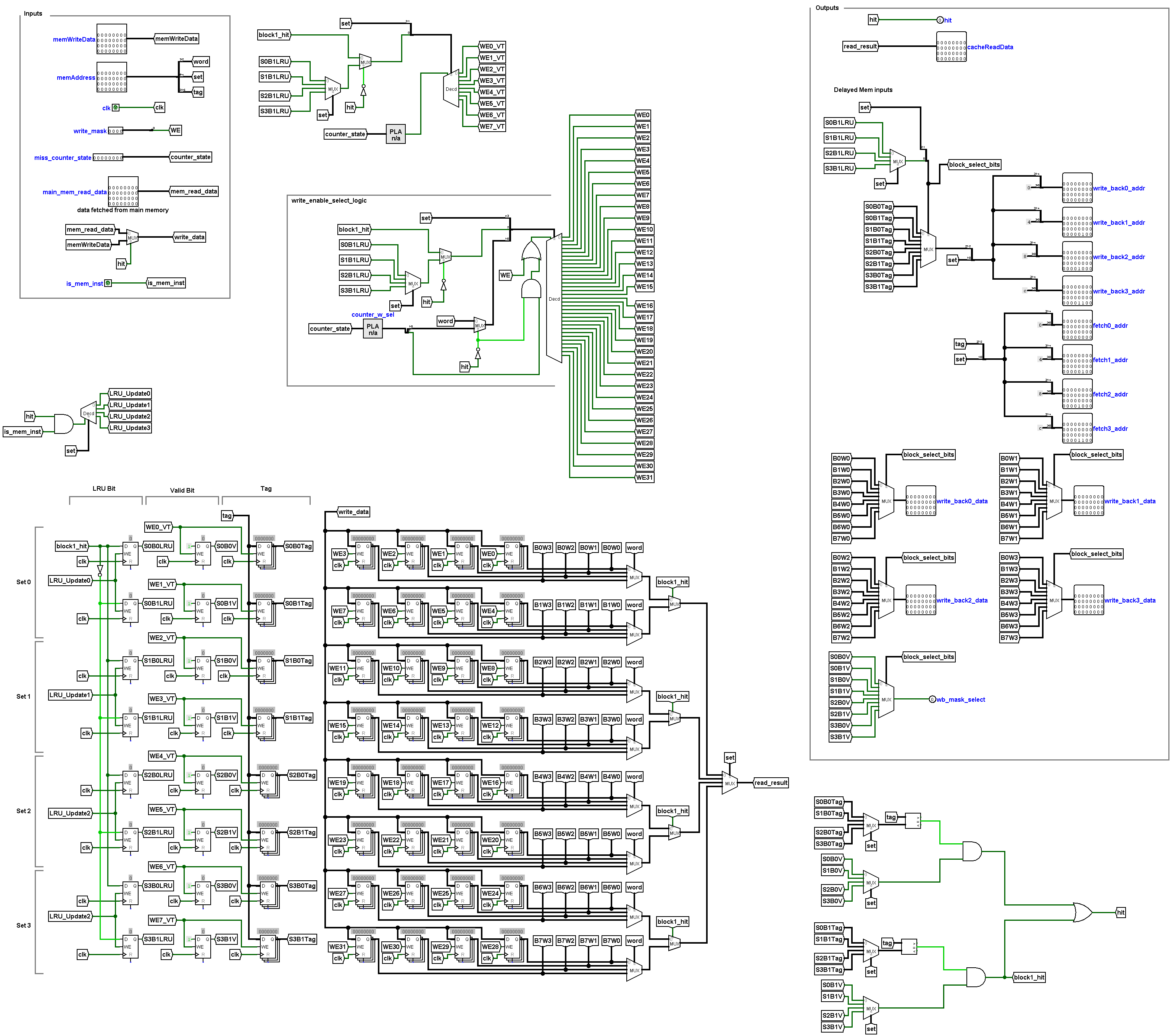
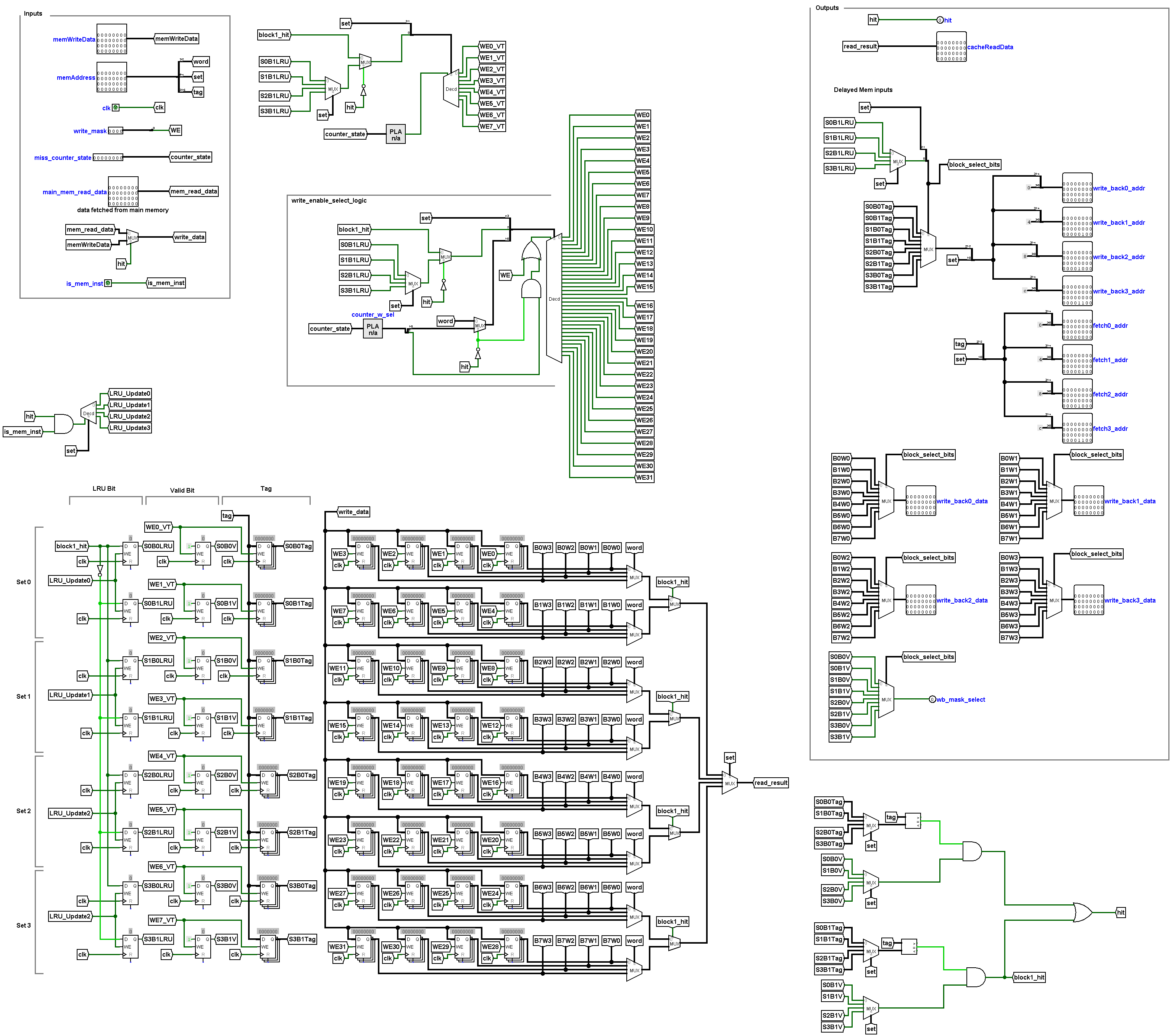
برای تشخیص و تولید سیگنال hit باید از tag و valid bit بلاک­های داخل set مربوطه استفاده کنیم. با استفاده از مدار زیر، سیگنال hit را تولید می­کنیم:

شکل 10- منطق تولید سیگنال hit

حالا چهار حالت مختلف را بررسی می­کنیم:

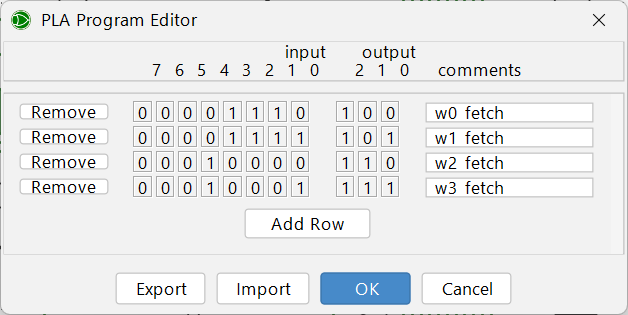
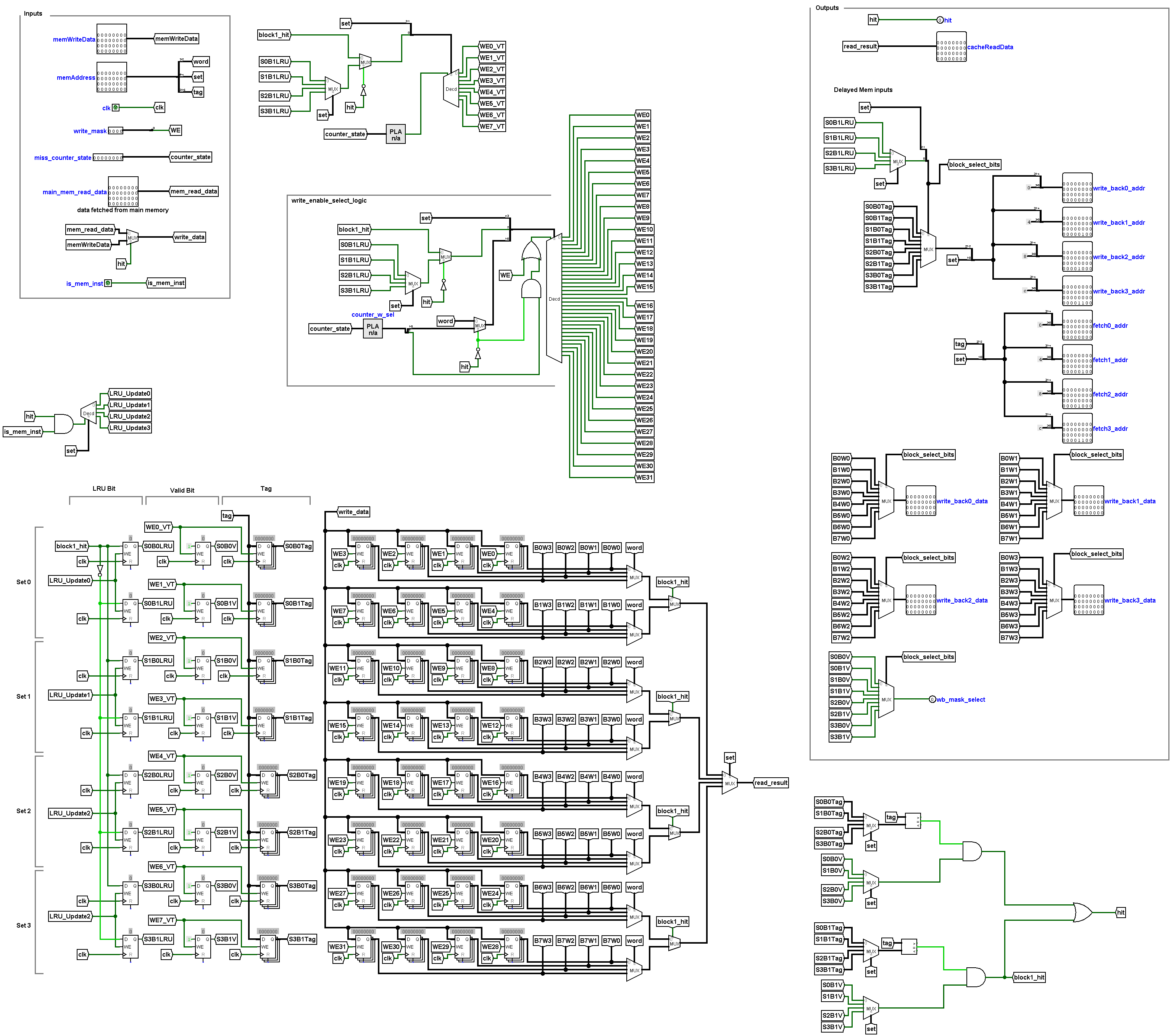
1. **hit در خواندن**: همانطور که در شکل 9 مشخص است، کلمه مورد نظر بدون تاخیر انتخاب شده و به عنوان خروجی داده می­شود.
2. **hit در نوشتن**: باید سیگنال WE کلمه مورد نظر در cache فعال شود (مطابق شکل 12). دیتایی که باید نوشته شود همانطور که در شکل 8 مشخص است، بر اساس hit شدن یا نشدن انتخاب می­شود.
3. **miss 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.
4. **miss در خواندن**: در این حالت باید 18 کلاک را مطابق جدول 1 طی کنیم. از کلاک 0 تا 8 همانطور که در شکل 3 و 4 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 3 از بین آنها انتخاب شود. در شکل زیر خروجی­های cache را نمایش می­دهیم:

شکل 11- خروجی­های cache



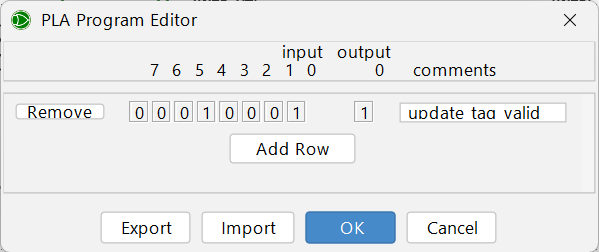
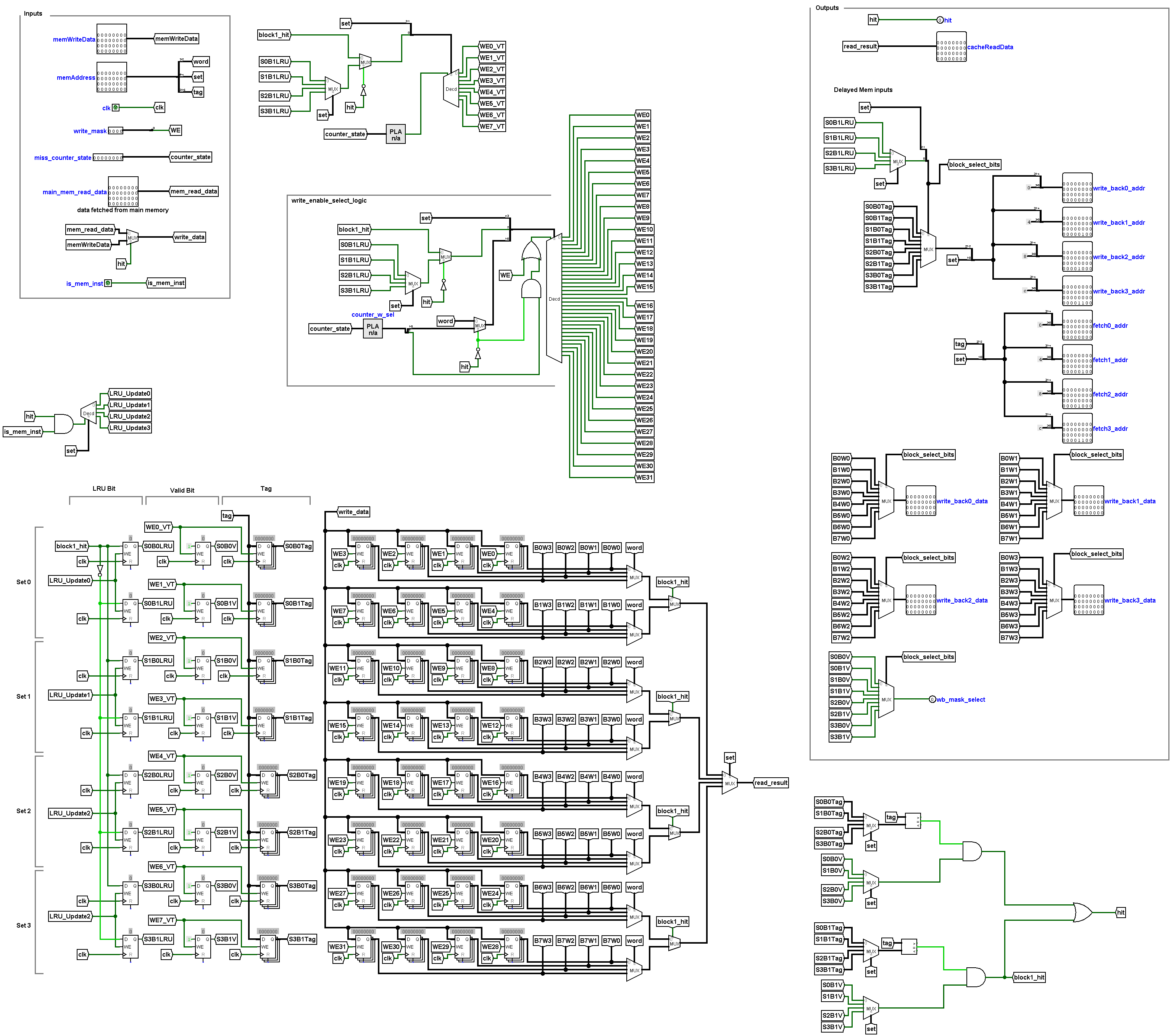
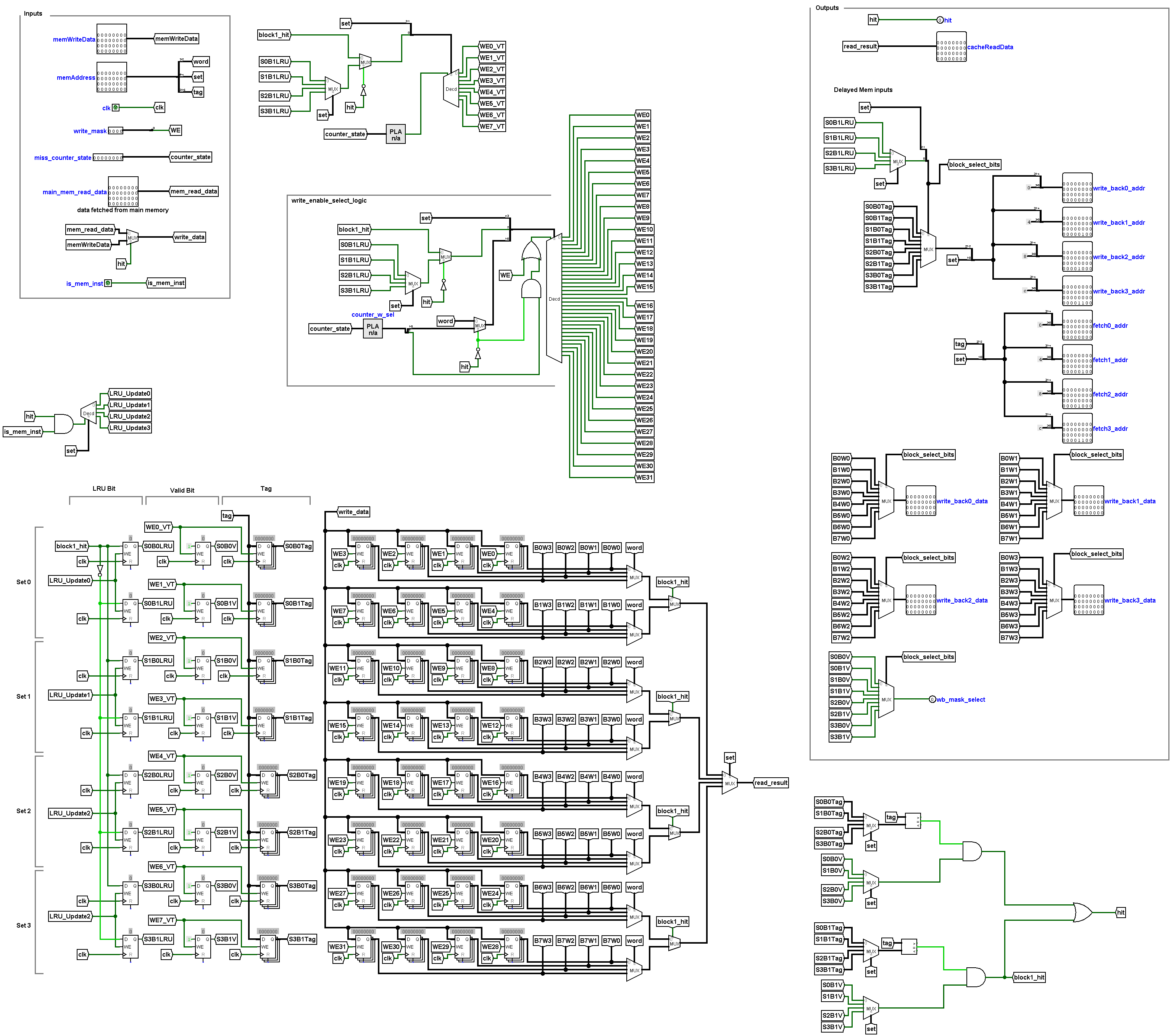
اما کلاک­های 14 تا 17 در جدول 1 را باید در داخل cache پیاده­سازی کنیم. برای نوشتن دیتای fetch شده در کلاک­های 14 تا 17 از مدار زیر استفاده می­کنیم که سیگنال­های WE هر کلمه در cache را کنترل می­کند:

شکل 12- منطق کنترل ورودی­های WE کلمات cache



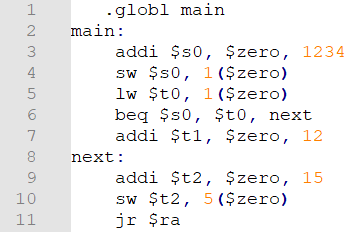
برای آپدیت کردن مقدار LRU و Valid bit در کلاک آخر از مدار زیر استفاده می­کنیم (نحوه اتصال این سیگنال­ها در شکل 9 نشان داده شده است):

شکل 13- منطق آپدیت کردن LRU و Valid bit

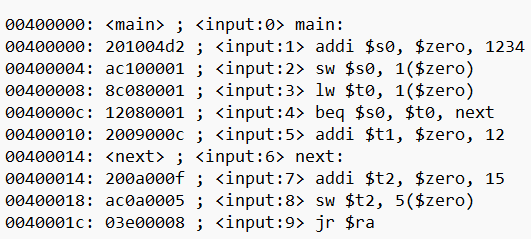


**یادآوری:** توجه کنید که سیاست LRU در حالت 2-way set-associative به این صورت عمل می­کند که هنگام eviction، LRU bit یکی از بلاک­ها صفر و دیگری یک است و بلاکی که LRU bit یک داشته باشد جایگزین شده و سپس جای صفر و یک بلاک­ها عوض می­شود. دلیل ورودی­های مالتی­پلکسرهای شکل­های 11، 12 و 13 همین است.

# تست عملکرد پردازنده بعد از اضافه شدن Cache

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:

شکل 18- 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:

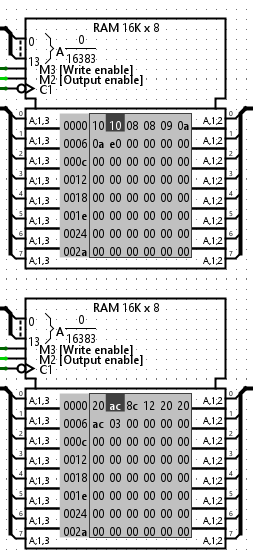
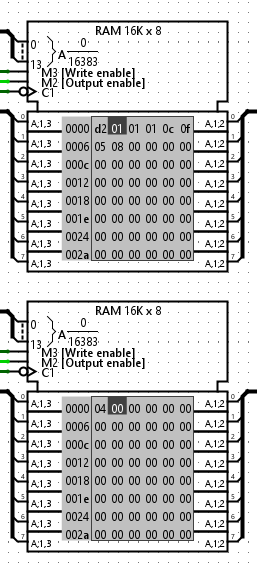
شکل 19- خروجی اسمبلر

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 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.

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.

شکل 20- چهار فایل ایجاد شده توسط کد پایتون

شکل 21- ماژول­های حافظه بعد از لود کردن دستورات



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://www.cs.fsu.edu/~zwang/files/cda3101/Fall2017/Lecture5_cda3101.pdf>
* <https://alanhogan.com/asu/assembler.php>
* <https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_help.html>

1. Least Recently Used Policy [↑](#footnote-ref-1)