
دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

# گزارش تمرین دوم شبیه سازی معماری کامپیوتر پیشرفته

## I. بخش اول : توضیح کدهای برنامه

دو شبه کد داده شده به زبان C++ نوشته شده اند. ابعاد ماتریس ها برای هر دو کد 150 * 150 در نظر گرفته شده است.

- Code a

کد اول که با نام فایل Step1.cpp در پوشه تمرین قرار گرفته، یک الگوریتم ضرب ماتریس معمولی را نشان می دهد. سه ماتریس A, B و C با ابعاد 150 * 150 تعریف شده اند.

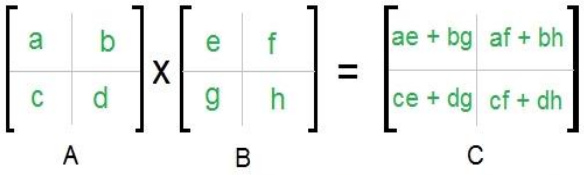
دو حلقه تو در تو برای پر کردن ماتریس ها با اعداد رندوم در بازه 1 تا 200 نوشته شده است.

سپس سه حلقه تو در تو برای ضرب دو ماتریس A و B و ذخیره نتایج در ماتریس C نوشته شده است.

در بخش آخر برنامه نیز با دو حلقه تو در تو محتوای ماتریس C نشان داده می شود.

- Code b

کد دوم با نام فایل step1.b.cpp در پوشه تمرین قرار گرفته، یک الگوریتم ضرب ماتریس به نام Strassen است.




A، B و C ماتریس‌های مربعی در اندازه N × N هستند.
a، b و c زیرماتریس‌های A با اندازه N/2 × N/2 هستند.
e، f و g زیرماتریس‌های B با اندازه N/2 × N/2 هستند.

این الگوریتم که یک الگوریتم تقسیم و غلبه است، برای ضرب دو ماتریس مربعی n*n از روش تقسیم ماتریس های A و B به 4 زیر ماتریس با ابعاد n/2 * n/2 استفاده می کند، سپس ماتریس نتیجه را به صورت بازگشتی از این زیر ماتریس ها محاسبه می کند. هدف این الگوریتم این است که به صورت بهینه تری از حافظه و کش استفاده کند.

در این کد، از حلقه های تو در تو برای تقسیم ماتریس ها به ماتریس های کوچکتر و ضرب آنها استفاده می کند.

هم چنین از دو حلقه تو در تو برای نمایش محتوای ماتریس C استفاده شده است.

## II. بخش دوم : گام های سوال 1 و بررسی نتایج

### 1. گام اول

**برای کد اول** با نام step1.cpp در شبیه ساز از دستور زیر استفاده شده :

```
build/X86/gem5.opt
configs/learning_gem5/part1/step1.py
homework /a.out
```

که step1.py مشابه فایل simple.py ولی با تغییر دایرکتوری به a.out برای کد اول است.

a.out فایل کامپایل شده کد با دستور {step1.cpp} g++ می باشد که در پوشه a تمرین گذاشته شده است.

عکس های مربوط به دستور و اجرای این بخش همراه stats آن و کد های استفاده شده در پوشه a --> step1 قرار داده شده است.

**برای کد دوم** با نام step1.b.cpp از دستور زیر در شبیه ساز استفاده شده است :

```
build/X86/gem5.opt
configs/learning_gem5/part1/step1_b.py
homework/step1_b/a.out
```

که step1_b.py مشابه فایل simple.py ولی با تغییر دایرکتوری به a.out برای کد دوم است.

a.out فایل کامپایل شده کد با دستور {step1.b.cpp} g++ می باشد که در پوشه b تمرین گذاشته شده است.

عکس های مربوط به دستور و اجرای این بخش همراه stats آن و کدهای استفاده شده در پوشه b --> step1 قرار داده شده است.

### 2. گام دوم

مقادیر پارامترهای خواسته شده با توجه به فایل های stats که در پوشه a برای کد اول و پوشه b برای کد دوم قرار داده شده، عبارتند از:

جدول 1 – مقادیر پارامترهای خواسته شده گام دوم

| | کد اول (a) | کد دوم (b) |
|---|---|---|
| IPC | 0.011034 | 0.010469 |
| Num busy cycle | 8362677783.999000 | 19096249973.999001 |

IPC (Instructions Per Cycle) ، یک معیار برای اندازه‌گیری تعداد دستورالعمل‌هایی است که یک پردازنده می‌تواند در یک چرخه کلاک اجرا کند. IPC بالاتر به این معناست که پردازنده می‌تواند دستورالعمل‌های بیشتری در ثانیه اجرا کند، که منجر به عملکرد بهتر می‌شود.

Busy Cycle (چرخه‌های مشغول) ، چرخه‌هایی از کلاک هستند که در آنها پردازنده هیچ دستورالعملی را اجرا نمی‌کند. این پدیده به دلایل مختلفی می تواند اتفاق بیفتد، مانند انتظار برای داده از حافظه یا انتظار برای اتمام اجرای یک دستورالعمل توسط پردازنده دیگری.

Number of Busy Cycle (تعداد چرخه‌های مشغول) بزرگتر به این معناست که پردازنده زمان کمتری را صرف اجرای دستورالعمل‌ها و زمان بیشتری را صرف انتظار می‌کند، که این می‌تواند منجر به عملکرد ضعیف‌تر شود.

با بررسی اطلاعات جدول 1 ، ملاحظه می شود که IPC با اختلاف بسیار کمی، تقریبا برای دو کد برابر است.

اما Num. busy cycle در حالت بدون حافظه نهان، برای کد دوم (الگوریتم ضرب Strassen) بیش از 2 برابر کد اول (الگوریتم ضرب معمولی) است.

یکی از دلایل اصلی بالا بودن Number of Busy Cycle (تعداد چرخه‌های مشغول) در الگوریتم ضرب ماتریس استراسن (کد دوم ) در حالت بدون حافظه نهان، نیاز به بارگذاری داده از حافظه است.

در سیستم بدون حافظه نهان، تمام داده‌های مورد نیاز برای اجرای الگوریتم ضرب ماتریس، باید از حافظه اصلی بارگذاری شوند. این امر می‌تواند منجر به چرخه‌های مشغول زیادی شود، زیرا پردازنده باید منتظر بماند تا داده‌ها از حافظه اصلی بارگذاری شوند.

در الگوریتم ضرب ماتریس معمولی (کد اول)، ماتریس‌ها به صورت خطی ضرب می‌شوند. این الگوریتم، نیاز کمتری به بارگذاری داده از حافظه اصلی دارد. بنابراین، تعداد چرخه‌های مشغول در این الگوریتم کمتر است.

در الگوریتم ضرب ماتریس استراسن، ماتریس‌ها به قطعات کوچک‌تر تقسیم می‌شوند و سپس این قطعات به صورت موازی ضرب می‌شوند. برای ضرب هر قطعه، الگوریتم نیاز به بارگذاری داده از حافظه اصلی دارد. این بارگذاری‌های داده می‌تواند منجر به چرخه‌های مشغول شود، زیرا پردازنده باید منتظر بماند تا داده‌ها از حافظه بارگذاری شوند.

دو ماتریس A و B داریم که هر کدام از آنها دارای n سطر و n ستون هستند. الگوریتم استراسن این دو ماتریس را به 4 قطعه کوچک‌تر تقسیم می‌کند:

```
A = A11 + A12 + A21 + A22
B = B11 + B12 + B21 + B22
```

سپس، الگوریتم این قطعات را به صورت موازی ضرب می‌کند:

```
C11 = A11 * B11 + A12 * B21
C12 = A11 * B12 + A12 * B22
C21 = A21 * B11 + A22 * B21
C22 = A21 * B12 + A22 * B22
```

به عنوان مثال، برای ضرب قطعه C11، الگوریتم نیاز به بارگذاری ماتریس A از حافظه دارد که این بارگذاری داده می‌تواند باعث انتظار پردازنده برای بارگذاری داده‌ها از حافظه شود.

## 3. گام سوم

سایز کش لایه اول برابر 128 kB و سایز کش لایه دوم برابر با 512 kB درنظر گرفته شده است ( این اندازه ها از جستجو در اینترنت به دست آمده اند).

**برای کد اول** در شبیه ساز از دستور زیر استفاده شده :

```
build/X86/gem5.opt
configs/deprecated/example/se.py
--cmd=homework/step2/a/a.out
--cpu-type= TimingSimpleCpu
--chaches -- L2cache
-- L1d_size= 128kB
-- L1i_size=64kB
-- L2_size= 512kB
-- L1d_assoc= 4
-- L1i_assoc = 4
-- L2_assoc= 8
--cacheline_size= 64
```

عکس های مربوط به دستور و اجرای این بخش به همراه فایل stats آن در پوشه a < -- step2 تمرین قرار گرفته است.

**برای کد دوم** از دستور زیر در شبیه ساز استفاده شده است :

```
build/X86/gem5.opt
configs/deprecated/example/se.py
--cmd=homework/step2/b/a.out
--cpu-type= TimingSimpleCpu
--chaches -- L2cache
-- L1d_size= 128kB
-- L1i_size=64kB
-- L2_size= 512kB
-- L1d_assoc= 4
-- L1i_assoc = 4
-- L2_assoc= 8
--cacheline_size= 64
```

عکس های مربوط به دستور و اجرای این بخش به همراه فایل stats آن در پوشه b < -- step2 تمرین قرار گرفته است.

## 4. گام چهارم

مقادیر پارامترهای خواسته شده با توجه به فایل های stats که در پوشه a برای کد اول و پوشه b برای کد دوم قرار داده شده، عبارتند از:

جدول 2 - مقادیر پارامترهای خواسته شده گام چهارم

| | **کد اول (a)** | **کد دوم (b)** |
|---|---|---|
| IPC | 0.268783 | 0.254877 |
| Num. busy cycle | 343292510.998000 | 784407296.998000 |
| L1i cache hit rate | 0.999986 | 0.999994 |
| L1i cache miss rate | 0.000014 | 0.000006 |
| L1d cache hit rate | 0.999564 | 0.999813 |
| L1d cache miss rate | 0.000436 | 0.000187 |
| L2 cache hit rate | 0.27575 | 0.263303 |
| L2 cache miss rate | 0.724275 | 0.736697 |

Hit rate درصد دفعاتی است که یک درخواست دسترسی به حافظه نهان (Cache)، در حافظه نهان یافت می‌شود. به عبارت دیگر، hit rate برابر است با نسبت تعداد درخواست‌های دسترسی به حافظه نهان که در حافظه نهان یافت می‌شوند، به تعداد کل درخواست‌های دسترسی به حافظه نهان.

Miss rate درصد دفعاتی است که یک درخواست دسترسی به حافظه نهان، در حافظه نهان یافت نمی‌شود. به عبارت دیگر، miss rate برابر است با نسبت تعداد درخواست‌های دسترسی به حافظه نهان که در حافظه نهان یافت نمی‌شوند، به تعداد کل درخواست‌های دسترسی به حافظه نهان.

Hit rate و Miss rate دو معیار مهم برای ارزیابی عملکرد حافظه نهان هستند. یک حافظه نهان با hit rate بالا، عملکرد بهتری خواهد داشت، زیرا درخواست‌های دسترسی به حافظه نهان را سریع‌تر می‌تواند پاسخ دهد.

رابطه بین hit rate و miss rate به صورت زیر است:

hit rate + miss rate = 1

## 5. گام پنجم

این گام را در دو بخش بررسی و نتایج را تحلیل خواهیم کرد:

بخش اول: مقایسه و تحلیل اختلاف مقادیر پارامترها در گام دو و چهار

بخش دوم: مقایسه و تحلیل مقادیر پارامترها برای کد اول و دوم در گام چهارم

- **بخش اول: مقایسه مقادیر در گام دوم و چهارم**

مقادیر IPC برای کد اول و کد دوم در گام دوم (فاقد حافظه نهان) و گام چهارم (دارای دو سطح حافظه نهان) نشان داده شده است:

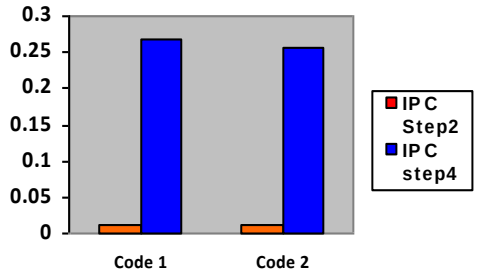
جدول 3 مقایسه مقادیر **IPC** برای دو کد در دو مرحله

| | **کد اول (a)** | **کد دوم (b)** |
|---|---|---|
| IPC Step 2 | 0.011034 | 0.010469 |
| IPC Step4 | 0.268783 | 0.254877 |

همانطور که ملاحظه می شود، مقادیر با میزان اختلاف کمی برای کد اول و کد دوم در یک گام، یکسان هستند.

اما طبق شکل 1 و جدول 3 ، مقدار پارامتر IPC برای هر کد، در حالت دو لایه کش ( گام چهارم ) نسبت به حالت فاقد کش ( گام دوم ) بیش از 20 برابر افزایش داشته است.

شکل 1 مقایسه مقادیر **IPC** برای دو کد در دو گام




در حالت بدون حافظه نهان، تمام داده‌های مورد نیاز برای اجرای یک دستورالعمل باید از حافظه اصلی بارگذاری شوند. این امر می‌تواند منجر به چرخه‌های مشغول زیاد و کاهش سرعت شود، زیرا پردازنده باید منتظر بماند تا داده‌ها از حافظه اصلی بارگذاری شوند.

در حالت دو لایه حافظه نهان، داده‌ها ابتدا در حافظه نهان سطح اول بارگذاری می‌شوند. اگر داده مورد نیاز در حافظه نهان سطح اول یافت شد، پردازنده می‌تواند به سرعت به آن دسترسی پیدا کند. در غیر این صورت، پردازنده باید داده را از حافظه اصلی بارگذاری کند.

در مجموع استفاده از حافظه نهان (Cache) می‌تواند نرخ IPC را زیاد کند به دلیل:

1. کاهش زمان دسترسی به داده:

- حافظه نهان به‌عنوان یک لایه سریع‌تر از حافظه اصلی عمل می‌کند. این به معنای کاهش زمان دسترسی به داده‌هاست.
- زمانی که یک دستور به داده‌ای نیاز دارد، اگر داده در حافظه نهان باشد، CPU نیازی به انتظار برای دسترسی به حافظه اصلی ندارد که زمان اجرای کد را کاهش می‌دهد و این می‌تواند به افزایش نرخ IPC منجر شود.

2. کاهش تعداد دستورات حافظه مرتبط

- استفاده از حافظه نهان می‌تواند تعداد دستورات حافظه مرتبط با دستورات اجرایی را کاهش دهد. اگر داده‌ها در حافظه نهان باشند، نیاز به دسترسی مکرر به حافظه اصلی برای هر دستور کمتر می‌شود.
- این کاهش تعداد دستورات حافظه مرتبط می‌تواند به معنای اجرای کارها با سرعت بیشتر و در نتیجه افزایش نرخ IPC باشد.

3. پیش‌بینی بهتر شاخه (Branch Prediction)

- برخی از حافظه‌های نهان دارای توانایی بهتری در پیش‌بینی شاخه‌ها هستند. اگر CPU بتواند به درستی پیش‌بینی کند که کدام شاخه اجرا خواهد شد، می‌تواند دستورات را به‌طور موثرتری اجرا کند و نرخ IPC را افزایش دهد.

4. استفاده بهینه از محلیت داده‌ها (Data Locality)

- حافظه نهان، داده‌هایی که به تازگی استفاده شده‌اند، ذخیره می‌کند. این به معنای افزایش محلیت داده‌ها است.

- استفاده بهینه از محلیت داده‌ها می‌تواند تأثیر مستقیمی بر کارآیی دستورات اجرایی داشته باشد و باعث افزایش نرخ IPC شود.

بنابراین، استفاده از حافظه نهان به عنوان یک فناوری بهینه‌سازی در ساختار حافظه CPU، می‌تواند به تسریع اجرای دستورات و افزایش نرخ IPC منجر شود.

---

مقادیر Number of Busy Cycle برای کد اول و کد دوم در گام دوم (فاقد حافظه نهان) و گام چهارم (دارای دو سطح حافظه نهان) نشان داده شده است:

جدول 4 مقایسه مقادیر #BusyCycle برای دو کد در دو مرحله

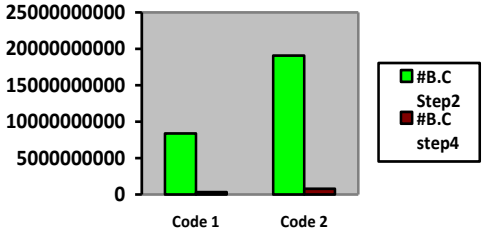
| | کد اول (a) | کد دوم (b) |
|---|---|---|
| #BusyCycle Step 2 | 8362677783.999000 | 19096249973.999001 |
| #BusyCycle Step4 | 343292510.998000 | 784407296.998000 |

در رابطه با علت بیشتر بودن Number Of Busy Cycle در کد دوم نسبت به کد اول، در گام دوم توضیح داده شده است.

با بررسی مقادیر جدول 4 ملاحظه می شود که تعداد چرخه های مشغول برای کد اول در گام دوم نسبت به گام چهارم حدودا 24 برابر بیشتر است. همینطور مقدار این پارامتر برای کد دوم در گام دوم نسبت به گام چهارم نیز تقریبا 24 برابر بالاتر است.

در نمودار زیر مقایسه بین مقادیر پارامتر Number Of Busy Cycle برای هر کد در دو مرحله نشان داده شده است.

شکل 2 - مقایسه مقادیر #BusyCycle برای دو کد در دو گام




حافظه نهان (Cache) یک حافظه سریع است که بین پردازنده و حافظه اصلی قرار دارد. هدف از استفاده از حافظه نهان، افزایش سرعت دسترسی پردازنده به داده‌ها و دستورالعمل‌ها است. در حالت بدون حافظه نهان، پردازنده برای دسترسی به داده‌ها و دستورالعمل‌ها باید مستقیماً به حافظه اصلی مراجعه کند. این کار می‌تواند زمان زیادی را صرف کند، زیرا حافظه اصلی نسبت به حافظه نهان کندتر است.

در حالت داشتن حافظه نهان، پردازنده ابتدا در حافظه نهان به دنبال داده‌ها و دستورالعمل‌ها می‌گردد، اگر داده یا دستورالعمل مورد نظر در حافظه نهان یافت شود، پردازنده می‌تواند به سرعت به آن دسترسی پیدا کند. در غیر این صورت، پردازنده باید به حافظه اصلی مراجعه کند.

بنابراین، در حالت داشتن حافظه نهان، تعداد چرخه‌های مشغول (#Busy Cycle) کمتر می‌شود، زیرا پردازنده کمتر به حافظه اصلی مراجعه می‌کند.

به طور خلاصه کاهش تعداد چرخه‌های مشغول در حالت داشتن حافظه نهان (Cache) نسبت به حالت بدون حافظه نهان می‌تواند به دلیل چندین عامل باشد:

۱. تقریب بهینه‌تر داده‌ها: حافظه نهان به برنامه کمک می‌کند تا داده‌های پراکنده در حافظه اصلی را به صورت تقریب بهینه‌تری در حافظه نهان ذخیره کند. این باعث کاهش زمان دسترسی به داده‌ها و اجرای دستورات می‌شود

و در نتیجه، تعداد چرخه‌های مشغول کمتری ممکن است مورد نیاز باشد.

۲. کاهش تعداد دستورات حافظه: حافظه نهان معمولاً مانع از نیاز به دسترسی مکرر به حافظه اصلی می‌شود. اگر داده‌های مورد نیاز توسط برنامه در حافظه نهان موجود باشند، دسترسی به آن‌ها سریع‌تر انجام می‌شود و تعداد دستورات حافظه کاهش می‌یابد.

۳. استفاده بهینه از خصوصیات مکانی داده: حافظه نهان دارای سطوح مختلف است که به صورت خصوصیات مکانی می‌تواند داده‌ها را ذخیره کند. این امکان به برنامه کمک می‌کند تا از خصوصیات مکانی حافظه نهان به صورت بهینه استفاده کند و داده‌های مورد نیاز را با کمترین تعداد دستورات دسترسی به حافظه بخواند.

۴. کاهش اثرات تاخیر حافظه: حافظه نهان باعث می‌شود تا اثرات تاخیر حافظه کاهش یابد. زمانی که داده‌ها در حافظه نهان قرار دارند، زمان دسترسی به آن‌ها به طور قابل توجهی کاهش می‌یابد و این باعث کاهش تعداد چرخه‌های مشغول ممکن است شود.

به طور کلی، حافظه نهان بهینه‌ترین استفاده از داده‌ها و دستورات را در زمان اجرای برنامه فراهم می‌کند که به کاهش تعداد چرخه‌های مشغول و بهبود عملکرد کلی سیستم منجر می‌شود.

- **بخش دوم: مقایسه و تحلیل مقادیر پارامترها برای کد اول و دوم در گام چهارم**

در این بخش پارامترهای Hit Rate و Miss Rate برای حافظه نهان لایه اول و لایه دوم بررسی خواهد شد. اما در ابتدا به تعریف مختصری از icache و dcache می پردازیم.

icache و dcache دو نوع حافظه نهان هستند که در پردازنده‌های مدرن استفاده می‌شوند.

icache برای ذخیره دستورالعمل‌ها و dcache برای ذخیره داده‌ها استفاده می‌شود.

icache مخفف Instruction Cache است. icache یک حافظه نهان کوچک است که دستورالعمل‌های مورد نیاز پردازنده را ذخیره می‌کند. این کار باعث می‌شود پردازنده بتواند به سرعت به دستورالعمل‌های مورد نیاز خود دسترسی پیدا کند.

dcache مخفف Data Cache است. dcache یک حافظه نهان کوچک است که داده‌های مورد نیاز پردازنده را ذخیره می‌کند. این کار باعث می‌شود پردازنده بتواند به سرعت به داده‌های مورد نیاز خود دسترسی پیدا کند.

پردازنده برای دسترسی به یک مقدار در حافظه نهان، ابتدا آدرس آن مقدار را در حافظه نهان جستجو می‌کند. اگر مقدار مورد نظر در حافظه نهان یافت شود، پردازنده می‌تواند به سرعت به آن دسترسی پیدا کند. در غیر این صورت، پردازنده باید به حافظه اصلی مراجعه کند.

در اینجا یک مقایسه بین icache و dcache آورده شده است:

| | ICACHE | DCACHE |
|---|---|---|
| **نوع داده** | دستورالعمل | داده |
| **اندازه** | کوچک | کوچک |
| **هدف** | افزایش سرعت دسترسی به دستورالعمل‌ها | افزایش سرعت دسترسی به داده‌ها |

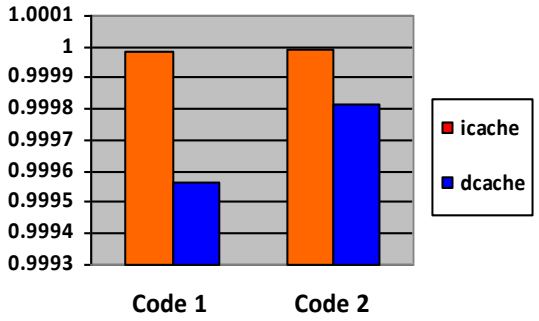
هردو icache و dcache نقش مهمی در بهبود عملکرد پردازنده‌ها دارند. این دو نوع حافظه نهان باعث می‌شوند پردازنده بتواند به سرعت به دستورالعمل‌ها و داده‌های مورد نیاز خود دسترسی پیدا کند. این امر باعث می‌شود پردازنده بتواند برنامه‌ها را سریع‌تر اجرا کند.

### ❖ بررسی Hit Rate و Miss Rate در Cache L1

در این بخش به بررسی Hit Rate در کش لایه اول و مقایسه این پارامتر برای کد اول و کد دوم می پردازیم.

در نمودار زیر مقادیر نرخ موفقیت برای کد اول و دوم در کش لایه اول ( به تفکیک icache و dcache ) آورده شده است:

شکل 3 مقایسه **Hit Rate** برای کش **L1**




با توجه به اطلاعات جدول و نمودار شکل 3، به بررسی دو نکته تفاوت در Hit Rate برای کد اول و کد دوم ، و همچنین تفاوت در Hit Rate برای icache و dcache می پردازیم.

<u>بررسی علت تفاوت Hit Rate در iCache و dCache:</u>

Hit rate در iCache نسبت به dCache بیشتر است، زیرا دستورالعمل‌ها معمولاً در برنامه‌ها تکرار می‌شوند. به عنوان مثال، یک برنامه ممکن است چندین بار از یک حلقه استفاده کند. در هر تکرار حلقه، پردازنده باید دستورالعمل‌های حلقه را از حافظه بارگیری کند. اگر این دستورالعمل‌ها در iCache ذخیره شده باشند، پردازنده می‌تواند بدون نیاز به دسترسی به حافظه اصلی، به آنها دسترسی پیدا کند.

در مقابل، داده‌ها معمولاً در برنامه‌ها تکرار نمی‌شوند. به عنوان مثال، یک برنامه ممکن است یک آرایه را بارگیری کند و سپس از آن آرایه استفاده کند. پردازنده فقط یک بار به آرایه دسترسی پیدا می‌کند. بنابراین، احتمال اینکه داده‌های مورد نیاز پردازنده در dCache یافت شوند، کمتر از احتمال این است که دستورالعمل‌های مورد نیاز پردازنده در iCache یافت شوند.

به طور کلی دلایل افزایش Hit Rate در iCache نسبت به dCache عبارت است از:

۱. ساختار دستورات و مدل اجرای برنامه: دستورات در برنامه‌ها معمولاً ساختارها و الگوهای خاصی دارند. این الگوها ممکن است باعث شود که دستورات به صورت مکرر استفاده شوند، و در نتیجه، احتمال حفظ آن‌ها در حافظه نهان دستورات بالاتر باشد که این منجر به افزایش Hit Rate می شود.

2. دستورالعمل‌ها معمولاً کوچکتر از داده‌ها هستند. بنابراین، احتمال اینکه یک دستورالعمل خاص در یک حافظه نهان یافت شود، بیشتر است.

3. دستورالعمل‌ها معمولاً به صورت خطی در حافظه ذخیره می‌شوند .این امر باعث می‌شود که الگوریتم‌های مدیریت حافظه نهان بتوانند دستورالعمل‌های مورد نیاز پردازنده را با دقت بیشتری پیش‌بینی کنند

در کل، بیشتر شدن Hit Rate در iCache نسبت به dCache به عوامل متعددی از جمله ساختار دستورات، الگوهای اجرایی برنامه، توزیع داده‌ها و دستورات، و ویژگی‌های Cache خود برمی‌گردد.

<u>بررسی علت تفاوت Hit Rate در کد اول و کد دوم :</u>

با بررسی اعداد جدول 2 مشاهده می شود که Hit Rate کش اول برای کد دوم ( الگوریتم استراسن) کمی بیشتر از الگوریتم ضرب معمولی است.

الگوریتم استراسن به دلیل ویژگی‌های خود که به بهینه‌سازی الگوهای دسترسی به حافظه کمک می‌کنند و تعداد Cache Miss ها را کاهش می‌دهد، به عنوان یک الگوریتم بهینه در استفاده از حافظه نهان شناخته می‌شود، به دلیل:

1. ساختار بازگشتی تقسیم و غلبه:

- الگوریتم استراسن از یک رویکرد بازگشتی تقسیم و غلبه استفاده می‌کند که ضرب ماتریس‌های بزرگتر را به ضرب زیرماتریس‌های کوچکتر تقسیم می‌کند.

- این ساختار بازگشتی تمایل دارد روی ماتریس های کوچکتر کار کند که به خوبی در حافظه نهان جا می گیرند. به همین دلیل، بهبود در محلی بودن داده و استفاده از حافظه نهان ایجاد می‌شود.

2. استفاده از متغیرهای محلی:

- الگوریتم استراسن از متغیرهای محلی برای ذخیره مقادیر میانی ماتریس‌ها استفاده می‌کند. این متغیرهای محلی معمولاً ماتریس‌های کوچکتر هستند.

- متغیرهای محلی منجر به کاهش نیاز به دسترسی به بخش‌های بزرگتر ماتریس‌ها در حافظه اصلی می‌شود و استفاده بهتر از حافظه نهان را تسهیل می‌دهد.

3. کاهش عملیات حسابی:

- تعداد کمتری عملیات حسابی به این معناست که حرکات داده نیز کمتر خواهد بود و داده‌های مشمول این عملیات احتمالاً در حافظه نهان باقی خواهند ماند.

4. دسترسی توالی به حافظه:

- الگوریتم به صورت متوالی به عناصر ماتریس دسترسی پیدا می‌کند که محلی بودن مکانی را بهبود می‌بخشد. دسترسی متوالی به حافظه با اندازه بلاک حافظه نهان همخوانی دارد و احتمال Cache Miss را کاهش می‌دهد.

- الگوهای دسترسی متوالی در کل عموماً Cache Friendly هستند، زیرا به پردازنده این امکان را می‌دهند که بلوک‌های متوالی حافظه را پیش‌بینی کند.

5. کاهش جابجایی داده:

- به دلیل ساختار بازگشتی الگوریتم، نتایج ماتریس میانی به صورت محلی محاسبه می‌شوند و فقط در مراحل بازگشتی بعدی از آن‌ها استفاده می‌شود.

این کاهش جابجایی داده به حداقل رساندن Cache Misses کمک می‌کند و با نزدیک نگه داشتن داده به واحدهای محاسبه، به بهبود عملکرد حافظه کمک می‌کند.

6. عملیا ت زیرماتریس:

الگوریتم استراسن عملیات را بر روی زیرماتریس‌ها انجام می‌دهد و این زیرماتریس‌ها معمولاً اندازه‌ای کوچک دارند که در حافظه نهان جا می‌شوند.

برای تحلیل نتایج Miss Rate در کش لایه اول، باتوجه به رابطه Miss Rate = 1 – Hit Rate ، به دلایل گفته شده Hit Rate در کش اول بسیار بالا بوده و طبیعتا عکس آن یعنی Miss Rate، بسیار اندک است. ( دلایل مذکور علاوه بر توجیه Hit Rate زیاد، برای توجیه Miss Rate پایین هم به کار می روند).

### ❖ بررسی Hit Rate و Miss Rate در Cache L2

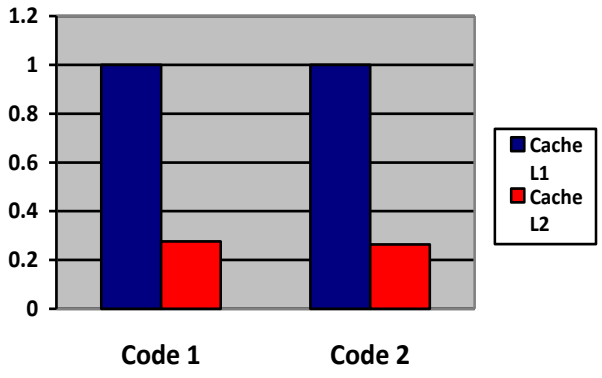
اطلاعات Hit/Miss Rate برای کد اول و کد دوم در جدول زیر آورده شده است:

جدول 5 مقادیر **Hit/Miss Rate** برای کش **L2**

| | Code 1 | Code 2 |
|---|---|---|
| L2 Cache HitRate | 0.27575 | 0.263303 |
| L2 Cache MissRate | 0.724275 | 0.736697 |

همچنین نمودار زیر مقایسه بین Hit Rate برای کش L1 و کش L2 را نشان می دهد.

شکل 4 مقایسه **Hit Rate** برای کش **L1** و **L2**



همانطور که در نمودار شکل 4 مشخص است، Hit Rate برای کش L2 نسبت به کش L1 برای هر دو کد تقریبا دو سوم کاهش داشته است.

برای توجیه این پدیده یعنی بیش از 90٪ بودن نرخ موفقیت برای کش اول ، و حدودا 30٪ بودن نرخ موفقیت در کش دوم می توان گفت که کش لایه اول از محلیت برنامه (Locality) آگاه است و این اطلاعات را برای بهبود عملکرد استفاده می کند. یعنی کش لایه اول ممکن است داده هایی را که اخیراً استفاده شده اند یا داده هایی که در نزدیکی داده های مورد استفاده قرار دارند، در اولویت قرار دهد.

اما کش لایه دوم از محلیت برنامه آگاه نیست. این بدان معناست که کش لایه دوم ممکن است داده هایی را در حافظه نهان ذخیره کند که به احتمال زیاد مورد استفاده قرار نمی گیرند. این امر می تواند منجر به افزایش Miss Rate و کاهش Hit Rate نسبت به کش لایه اول شود.

در مجموع بالا بودن Hit Rate در کش لایه اول نسبت به کش لایه دوم می‌تواند به عوامل مختلفی بازگردد که عبارتند از:

1. استفاده از کش لایه اول برای داده‌های محلی:

- داده‌هایی که توسط برنامه‌ها مکرراً استفاده می‌شوند و در کش لایه اول ذخیره می‌شوند، معمولاً مربوط به حلقه‌ها یا دستورات مکرر در برنامه‌ها هستند. این داده‌ها در کش لایه اول باقی مانده و احتمالاً hit rate بیشتری دارند.

2. الگوهای دسترسی به داده:

- الگوهای دسترسی به داده‌ها ممکن است باعث شود که داده‌ها به نحوی دسترسی شوند که hit rate در کش لایه اول بیشتر باشد. مثلاً اگر داده‌های مورد استفاده به صورت مکرر در حافظه فیزیکی متوالی باشند، احتمال hit در کش لایه اول بالاتر است.

3. الگوریتم های جایگذاری ممکن است تاثیرگذار باشند:

- الگوریتم‌های جایگذاری (Replacement Algorithms) ممکن است در کش لایه اول و دوم متفاوت باشند. اگر الگوریتم جایگذاری در کش لایه اول بهتر باشد و ترجیحاً به داده‌هایی که مکرراً استفاده می‌شوند اجازه hit دهد، hit rate افزایش می‌یابد.

## III. بخش سوم: سوال 2 تمرین

این تمرین شامل دو بخش با استفاده از کد sample.c می باشد.

- **گام اول**

برای این گام از دستور زیر در شبیه ساز استفاده شده است:

```
Build/X86/gem5.opt
configs/deprecated/example/se.py
–cmd=homework/Q2/a.out
–cpu-type=DerivO3CPU
--caches
```

برای نوع پردازنده Atomic نیز از دستور زیر استفاده شده است:

```
Build/X86/gem5.opt
configs/deprecated/example/se.py
–cmd=homework/Q2/a.out
–cpu-type=AtomicSimpleCPU
--caches
```

فایل Stats مربوط به هرنوع پردازنده در پوشه Q2 تمرین قرار گرفته است.

با بررسی فایل Stats، مقادیر پارامترهای IPC و Sim_Seconds برای این دو نوع پردازنده عبارتند از:

| | IPC | Sim_Second |
|---|---|---|
| AtomicSimpleCPU | 0.468535 | 0.002788 |
| DerivO3CPU | 0.674229 | 0.001938 |

همانطور که در جدول بالا مشخص است، مقدار IPC برای پردازنده DerivO3CPU نسبت به AtomicSimpleCPU بیشتر است.

برای توجیه علت این تفاوت می توان گفت در معماری‌های مختلف پردازنده، اجرای یک برنامه و عملیات‌های مختلف ممکن است به صورت متفاوتی پیاده‌سازی شود. AtomicSimpleCPU و DerivO3CPU دو نوع مختلف از مدل‌های پردازنده در شبیه‌سازهای معماری کامپیوتر می‌باشند که از هم تفاوت‌هایی دارند.

علت تفاوت مقدار IPC برای یک کد در دو نوع پردازنده AtomicSimpleCPU و DerivO3CPU، تفاوت در معماری و عملکرد این دو پردازنده است.

AtomicSimpleCPU یک پردازنده ساده و ابتدایی است که از یک مدل محاسباتی ساده برای اجرای دستورالعمل ها استفاده می کند. این پردازنده از ویژگی های پیشرفته ای مانند پیش بینی شاخه، بازیابی اطلاعات از حافظه، و اجرای موازی دستورالعمل ها پشتیبانی نمی کند.

DerivO3CPU یک پردازنده پیچیده تر است که از یک مدل محاسباتی پیشرفته برای اجرای دستورالعمل ها استفاده می کند. این پردازنده از ویژگی های پیشرفته ای مانند پیش بینی شاخه، بازیابی اطلاعات از حافظه، و اجرای موازی دستورالعمل ها پشتیبانی می کند.

به طور کلی، پردازنده هایی که از ویژگی های پیشرفته تری استفاده می کنند، می توانند IPC بالاتری داشته باشند. این امر به این دلیل است که این پردازنده ها می توانند دستورالعمل ها را به طور موثرتر و کارآمدتر اجرا کنند.

در اینجا برخی از دلایل خاص تفاوت IPC در این دو پردازنده آورده شده است:

- پیش بینی شاخه (BranchPrediction): AtomicSimpleCPU از پیش بینی شاخه پشتیبانی نمی کند. این بدان معناست که هر بار که یک شاخه در کد وجود داشته باشد، پردازنده باید تمام دستورالعمل های پس از شاخه را اجرا کند، حتی اگر شاخه اجرا نشود. در DerivO3CPU، از پیش بینی شاخه استفاده می شود. این بدان معناست که پردازنده می تواند شاخه ها را پیش بینی کند و فقط دستورالعمل های پس از شاخه هایی را که اجرا می شوند، اجرا کند. این امر می تواند باعث افزایش IPC در DerivO3CPU شود.
- بازیابی اطلاعات از حافظه: AtomicSimpleCPU از بازیابی اطلاعات از حافظه به صورت موازی پشتیبانی

نمی کند. این بدان معناست که پردازنده باید برای هر دسترسی به حافظه، یک چرخه را صرف کند. در DerivO3CPU، از بازیابی اطلاعات از حافظه به صورت موازی پشتیبانی می شود. این بدان معناست که پردازنده می تواند چندین دسترسی به حافظه را به صورت موازی انجام دهد. این امر می تواند باعث افزایش IPC در DerivO3CPU شود.

- اجرای موازی دستورالعمل ها: AtomicSimpleCPU از اجرای موازی دستورالعمل ها پشتیبانی نمی کند. این بدان معناست که پردازنده باید دستورالعمل ها را به صورت سریالی اجرا کند. در DerivO3CPU، از اجرای موازی دستورالعمل ها پشتیبانی می شود. این بدان معناست که پردازنده می تواند چندین دستورالعمل را به صورت موازی اجرا کند. این امر می تواند باعث افزایش IPC در DerivO3CPU شود.

با بررسی جدول و مقایسه مقادیر پارامتر Sim_Seconds برای دو نوع پردازنده، ملاحظه می شود که مقدار این پارامتر برای پردازنده AtomicSimpleCPU بیشتر از Derivo3CPU است.

Sim_seconds یک معیار زمانی است که در شبیه‌سازهای معماری کامپیوتر برای اندازه‌گیری زمان گذرانده در طول شبیه‌سازی استفاده می‌شود. این مقدار نشان‌دهنده زمان سیستم در واحد ثانیه به ازای شبیه‌سازی انجام شده است.

تفاوت در مقدار Sim_seconds بین دو نوع پردازنده AtomicSimpleCPU و DerivO3CPU ممکن است به دلیل موارد زیر باشد:

در اینجا برخی از دلایل خاص تفاوت Sim_Seconds در این دو پردازنده آورده شده است:

- پیش بینی شاخه: AtomicSimpleCPU از پیش بینی شاخه پشتیبانی نمی کند. این بدان معناست که هر بار که یک شاخه در کد وجود داشته باشد، پردازنده باید تمام دستورالعمل های پس از شاخه را اجرا کند، حتی اگر شاخه اجرا نشود. در DerivO3CPU ، از پیش بینی شاخه استفاده می شود .این بدان معناست که پردازنده می تواند شاخه ها را پیش بینی کند و فقط دستورالعمل های پس از شاخه هایی را که اجرا می شوند، اجرا کند. این امر می تواند باعث کاهش Sim_Seconds در DerivO3CPU شود.
- بازیابی اطلاعات از حافظه: AtomicSimpleCPU از بازیابی اطلاعات از حافظه به صورت موازی پشتیبانی نمی کند .این بدان معناست که پردازنده باید برای هر دسترسی به حافظه، یک چرخه را صرف کند .در DerivO3CPU، از بازیابی اطلاعات از حافظه به صورت موازی پشتیبانی می شود .این بدان معناست که پردازنده می تواند چندین دسترسی به حافظه را به صورت موازی انجام دهد .این امر می تواند باعث کاهش Sim_Seconds در DerivO3CPU شود.
- اجرای موازی دستورالعمل ها: AtomicSimpleCPU از اجرای موازی دستورالعمل ها پشتیبانی نمی کند .این بدان معناست که پردازنده باید دستورالعمل ها را به صورت سریالی اجرا کند .درDerivO3CPU ، از اجرای موازی دستورالعمل ها پشتیبانی می شود .این بدان معناست که پردازنده می تواند چندین دستورالعمل را به صورت موازی اجرا کند .این امر می تواند باعث کاهش Sim_Seconds در DerivO3CPU شود.

البته، مقدار Sim_Seconds برای یک کد در یک پردازنده خاص، به عوامل دیگری نیز بستگی دارد، مانند پیچیدگی کد، الگوهای دسترسی به حافظه، و اندازه و سرعت حافظه.

به طور کلی، می توان گفت که در صورت استفاده از کدی که از ویژگی های پیشرفته ای مانند پیش بینی شاخه، بازیابی اطلاعات از حافظه، و اجرای موازی دستورالعمل ها استفاده می کند، پردازنده DerivO3CPU می تواند عملکرد بهتری نسبت به پردازنده AtomicSimpleCPU داشته باشد. این امر باعث می شود که مقدار Sim_Seconds در پردازنده DerivO3CPU کمتر باشد.

- **گام دوم: تغییر در تابع**

برای بهبود پارامتر IPC در تابع reverse ، یک تغییر ساده ارائه شده است که از یک متغیر اضافی به نام prev برای نگه داشتن ارتباط قبلی استفاده می‌کند. در این تغییر، از یک حلقه while استفاده شده است که تا زمانی که لیست پیوندی به پایان نرسیده‌است، اجرا می‌شود و با تغییرات در ارتباطات next و prev لیست را معکوس می‌کند. این رویکرد باعث افزایش بهره‌وری و بهبود IPC می‌شود.

در تابع reverse ، استفاده از متغیر prev به این دلیل است که از یک حلقه while به جای یک تابع بازگشتی، برای تغییر ارتباطات در لیست پیوندی استفاده می‌شود. این حلقه به ازای هر گام، دو عمل اصلی را انجام می‌دهد:

next = current - -> next

next متغیری است که به عنوان نشانگر به گام بعدی در لیست پیوندی استفاده می‌شود. سپس:

current -->next = prev

این عمل باعث تغییر جهت لیست پیوندی می‌شود و current به عنوان نشانگر به گام قبلی (قبل از معکوس کردن) تنظیم می‌شود.

prev = current

متغیر prev به عنوان نشانگر به گام فعلی (بعد از معکوس کردن) تنظیم می‌شود.

current = next

متغیر current به عنوان نشانگر به گام بعدی در لیست پیوندی تنظیم می‌شود.

با این ترتیب، تغییرات در لیست پیوندی به صورت مستقیم و در محل انجام می‌شود و نیازی به تابع بازگشتی نیست. این ساختار کمک می‌کند تا عملیات معکوس کردن لیست پیوندی به صورت مؤثرتر و با بهره‌وری بیشتر انجام شود، که می‌تواند بهبود در IPC و سرعت اجرای الگوریتم ایجاد کند.

کد این تابع به نام فایل sample_prev.c در پوشه Q2 تمرین قرار داده شده است.

## IV. بخش چهارم: سوال 3

در شبیه سازی سیستم‌های حافظه، سه مد اصلی برای انجام عملیات روی حافظه وجود دارد:

Mode Timing: در این مد، عملیات‌های حافظه به صورت دقیق و بر اساس زمان‌بندی واقعی انجام می‌شوند. این مد برای شبیه‌سازی دقیق رفتار سیستم حافظه در دنیای واقعی استفاده می‌شود.

Mode Atomic: در این مد، عملیات‌های حافظه به صورت اتمی انجام می‌شوند. یعنی یا تمام عملیات انجام می‌شود یا هیچ عملیاتی انجام نمی‌شود. این مد برای شبیه‌سازی رفتار سیستم حافظه در حالت‌های بحرانی استفاده می‌شود.

Mode Functional: در این مد، عملیات‌های حافظه به صورت تابعی انجام می‌شوند. یعنی فقط نتیجه عملیات مهم است و زمان‌بندی یا نحوه انجام عملیات اهمیتی ندارد. این مد برای شبیه‌سازی عملکرد کلی سیستم حافظه استفاده می‌شود.

در ادامه به بررسی تفاوت‌های این سه مد و کاربردهای آنها در شبیه‌سازی می‌پردازیم.

- **Timing Mode**

در این مد، تمام عملیات‌های حافظه به صورت دقیق و بر اساس زمان‌بندی واقعی انجام می‌شوند. این بدان معناست که زمان‌های تأخیر بین مراحل مختلف عملیات حافظه، مانند زمان دسترسی به حافظه، زمان خواندن/نوشتن داده‌ها و زمان تأخیر درایورهای حافظه، به دقت محاسبه و در شبیه‌سازی لحاظ می‌شوند.

برای شبیه سازی دقیق زمان دسترسی به حافظه، باید مشخصات دقیق حافظه، مانند اندازه حافظه، زمان تأخیر در حافظه، و سرعت خط باس حافظه، در مدل شبیه سازی گنجانده شوند.

در حالت Timing mode، هر عملیات حافظه به صورت زیر شبیه سازی می شود:

- پردازنده درخواست دسترسی به حافظه را صادر می کند.
- درخواست دسترسی به حافظه از طریق گذرگاه حافظه به حافظه منتقل می شود.
- حافظه درخواست دسترسی را پردازش می کند.
- داده مورد نظر از حافظه به پردازنده منتقل می شود.

و زمان دسترسی به حافظه برابر است با مجموع زمان های زیر:

- زمان پردازش درخواست دسترسی در پردازنده
- زمان انتقال درخواست دسترسی از طریق گذرگاه حافظه
- زمان پردازش درخواست دسترسی در حافظه
- زمان انتقال داده از حافظه به پردازنده

زمان پردازش درخواست دسترسی در پردازنده معمولاً بسیار کوتاه است و می توان آن را نادیده گرفت. زمان انتقال درخواست دسترسی از طریق گذرگاه حافظه نیز معمولاً کوتاه است، اما می تواند در سیستم های حافظه با سرعت بالا مهم باشد.

زمان پردازش درخواست دسترسی در حافظه به عوامل مختلفی بستگی دارد، از جمله اندازه حافظه، نوع حافظه، و زمان تأخیر در حافظه.

زمان انتقال داده از حافظه به پردازنده به عوامل مختلفی بستگی دارد، از جمله اندازه داده، نوع حافظه، و سرعت خط Bus حافظه.

این مد برای شبیه‌سازی دقیق رفتار سیستم حافظه در دنیای واقعی استفاده می‌شود و این حالت برای شبیه سازی سیستم های حافظه با عملکرد بالا، مانند سیستم های کامپیوتری سطح بالا، ضروری است. این امر می تواند به مهندسان کمک کند تا سیستم های حافظه را بهینه کنند و عملکرد آنها را بهبود بخشند.

در اینجا چند مثال از کاربرد حالت Timing mode در شبیه سازی سیستم های حافظه آورده شده است:

- شبیه سازی عملکرد سیستم های حافظه با زمان دسترسی متفاوت
- شبیه سازی عملکرد سیستم های حافظه با اندازه حافظه متفاوت
- شبیه سازی عملکرد سیستم های حافظه با نوع حافظه متفاوت

مزایای Timing mode:

- دقت بالا
- امکان شبیه سازی دقیق الگوهای دسترسی متفاوت

معایب Timing mode:

- شبیه سازی کندتر
- پیچیدگی بیشتر

- **Atomic Mode**

در این مد، عملیات‌های حافظه به صورت اتمی انجام می‌شوند (زمان دسترسی به حافظه به صورت یک واحد فرض می شود). یعنی یا تمام عملیات انجام می‌شود یا هیچ عملیاتی انجام نمی‌شود. این بدان معناست که اگر در حین انجام یک عملیات حافظه، وقفه‌ای رخ دهد، عملیات به صورت کامل انجام شده یا اصلاً انجام نشده و به حالت اولیه خود باز می‌گردد.
این مد برای شبیه‌سازی رفتار سیستم حافظه در حالت‌های بحرانی استفاده می‌شود. به عنوان مثال، می‌توان از این مد برای شبیه‌سازی عملکرد یک سیستم حافظه در حالتی که چند پردازنده به طور همزمان سعی در دسترسی به یک مکان حافظه دارند استفاده کرد. استفاده از حالت Atomic mode شبیه سازی را سریع تر می کند، زیرا نیازی به محاسبه زمان دسترسی به حافظه برای هر عملیات حافظه نیست.

با این حال، دقت حالت Atomic mode کمتر از حالت Timing mode است.

در حالت Atomic mode ، هر عملیات حافظه به صورت زیر شبیه سازی می شود:

- پردازنده درخواست دسترسی به حافظه را صادر می کند.
- حافظه درخواست دسترسی را پردازش می کند.
- داده مورد نظر از حافظه به پردازنده منتقل می شود.

زمان دسترسی به حافظه هم برابر است با یک مقدار ثابت، که معمولاً برابر با زمان دسترسی متوسط حافظه است.

در اینجا چند مثال از کاربرد حالت Atomic mode در شبیه سازی سیستم های حافظه آورده شده است:

- شبیه سازی عملکرد سیستم های حافظه با اندازه حافظه متفاوت
- شبیه سازی عملکرد سیستم های حافظه با نوع حافظه متفاوت
- شبیه سازی عملکرد سیستم های حافظه با الگوهای دسترسی متفاوت

مزایای Atomic mode:

- شبیه سازی سریع تر
- سادگی بیشتر

معایب Atomic mode:

- دقت کمتر
- عدم امکان شبیه سازی دقیق الگوهای دسترسی متفاوت

- **Functional Mode**

در این مد، عملیات‌های حافظه به صورت تابعی انجام می‌شوند. یعنی فقط نتیجه عملیات و عملکرد کلی سیستم مهم است و زمان‌بندی یا نحوه انجام عملیات اهمیتی ندارد.
این مد ساده ترین حالت برای شبیه سازی سیستم های حافظه است که برای شبیه سازی سیستم های حافظه با عملکرد بسیار پایین، مانند سیستم های کامپیوتری کوچک و ساده، مناسب است.

به عنوان مثال، می‌توان از این مد برای شبیه‌سازی عملکرد یک سیستم حافظه در حالتی که فقط تعداد دسترسی‌ها به حافظه و زمان‌های تأخیر کلی اهمیت دارند استفاده کرد.

در حالت Functional mode ، هر عملیات حافظه به صورت زیر شبیه سازی می شود:

- پردازنده درخواست دسترسی به حافظه را صادر می کند.
- حافظه درخواست دسترسی را پردازش می کند.
- داده مورد نظر از حافظه به پردازنده منتقل می شود.

زمان دسترسی به حافظه برابر است با یک مقدار ثابت، که معمولاً برابر با زمان دسترسی متوسط حافظه است.

مزایای Functional mode:

- شبیه سازی بسیار سریع
- سادگی بسیار زیاد

معایب Functional mode:

- دقت بسیار کم
- عدم امکان شبیه سازی دقیق الگوهای دسترسی متفاوت

در جدول زیر، خلاصه‌ای از تفاوت‌های این سه مد ارائه شده است:

| ویژگی | MODE TIMING | MODE ATOMIC | MODE FUNCTIONAL |
|---|---|---|---|
| **زمان‌بندی** | دقیق | اتمی | تابعی |
| **اهمیت زمان‌بندی** | مهم | مهم | بی‌اهمیت |
| **کاربرد** | شبیه‌سازی دقیق | شبیه‌سازی رفتار بحرانی | شبیه‌سازی عملکرد کلی |

در مجموع انتخاب مد مناسب برای شبیه‌سازی سیستم‌های حافظه به عوامل مختلفی بستگی دارد، از جمله:

- دقت مورد نیاز
- نوع رفتاری که می‌خواهید شبیه‌سازی کنید
- زمان و منابع موجود برای شبیه‌سازی

به عنوان مثال، اگر نیاز به شبیه‌سازی دقیق رفتار یک سیستم حافظه چند کاناله دارید، باید از Mode Timing استفاده کنید. اگر نیاز به شبیه‌سازی رفتار یک سیستم حافظه در حالت‌های بحرانی دارید، باید از Mode Atomic استفاده کنید. و اگر نیاز به شبیه‌سازی عملکرد کلی یک سیستم حافظه دارید، می‌توانید از Mode Functional استفاده کنید.

### ❖ مقادیر پارامترهای خواسته شده در سوال 3

اطلاعات پارامترهای خواسته شده در 3 در جدول زیر آورده شده است:

جدول 6 مقادیر پارمترهای خواسته شده سوال 3

| | |
|---|---|
| **دامنه کلاک** | "1GHZ" |
| CLK_DOMAIN.CLOCK | |
| **مد شبیه سازی حافظه** | "timing" |
| mem_mode | |
| **سایز حافظه** | "512MB" |
| mem_ranges | |