Cache design part 2

1. ما هو عنوان الذاكرة؟

عنوان الذاكرة هو ببساطة رقم يُستخدم لتحديد مكان محدد في الذاكرة الرئيسية (RAM).

- الذاكرة الرئيسية (RAM) عبارة عن مجموعة ضخمة من الخلايا (كل خلية تخزن بيانات).
- كل خلية في الذاكرة لها عنوان (Address)، وهو مثل "رقم الشقة" أو "رقم صندوق بريد"، يُستخدم لتحديد أين تُخزن أو تُقرأ البيانات.

مثال:

- تخيل أن لديك ذاكرة بحجم 32 خلية.
- كل خلية لها رقم (0، 1، 2، 3، ... إلى 31).
- هذه الأرقام هي عناوين الذاكرة.

2. ما هو عنوان الكاش؟

عندما نستخدم ذاكرة الـ Cache (التي هي أسرع من RAM لكنها أصغر حجمًا)، نحتاج إلى آلية لتنظيم البيانات.

- لأن الكاش صغير، لا يمكنه تخزين كل البيانات من RAM.
- لذلك، نُخزن البيانات من RAM في الكاش بطريقة مُنظمة باستخدام "عنوان الكاش".

العلاقة بين ذاكرة RAM وذاكرة Cache:

1. كل عنوان في RAM يمكن أن يُخزن في الكاش.
2. العنوان في الكاش يتم تنظيمه باستخدام **أجزاء خاصة** (Tag، Index، Offset).

1. عنوان الذاكرة (Memory Address):

- عنوان الذاكرة في RAM يُستخدم لتحديد مكان كل خلية (أو بلوك) في الذاكرة الرئيسية.
- إذا كانت الذاكرة تحتوي على 2^n خلايا، فإن عنوان الذاكرة يتكون من n بت.

مثال:

إذا كان لديك ذاكرة بحجم 32 خلية (2^5)، فإن كل خلية سيكون لها عنوان مكوّن من 5 بتات:

- 00000، 00001، 00010، ... حتى 11111.

2. عنوان الكاش (Cache Address):

- عند استخدام الكاش، لا تتغير عدد البتات في العنوان.
- لكن يتم تقسيم هذه البتات إلى **Tag** و**Index** و**Offset**، وكل جزء له وظيفة معينة.

التقسيم:

- **Tag**: يحدد أي "مجموعة من البلوكات" في الذاكرة الرئيسية مرتبطة بهذا العنوان.
- **Index**: يحدد الخط (Line) داخل الكاش.
- **Offset**: يحدد الموضع داخل البلوك (إذا كان البلوك يحتوي على أكثر من كلمة).

DIRECT MAPPING أول طريقه للتخزين

ما هي طريقة الـ Direct Mapping؟

في طريقة الـ Direct Mapping:

1. يتم تقسيم الذاكرة الرئيسية إلى **بلوكات** (Blocks).
2. يتم تقسيم ذاكرة الـ Cache إلى **خطوط** (Lines).
3. كل بلوك من الذاكرة الرئيسية له مكان محدد واحد فقط في ذاكرة الـ Cache.

ما هي مشكلة التخزين؟

الـ Cache عادةً تكون صغيرة مقارنة بالذاكرة الرئيسية (RAM)، لذلك لا يمكنها تخزين كل شيء. هنا تأتي فكرة **Direct Mapping** لتنظيم البيانات التي تُخزن في الـ Cache.

كيف تعمل الطريقة؟

الخطوات:

1. **تقسيم العناوين:** عنوان الذاكرة (Memory Address) يُقسم إلى 3 أجزاء:
   - **Tag**: لتحديد البلوك.
   - **Index**: لتحديد موقع الخط (line) داخل الـ Cache.
   - **Block Offset**: لتحديد الكلمة (word) داخل البلوك (عادةً يتم تجاهلها إذا اعتبرنا البلوك كلمة واحدة).
2. **الحساب لتحديد موقع البلوك في الـ Cache:** يتم حساب **Index** من خلال معادلة بسيطة:

```
Copy code

Index = Block Number % Number of Cache Lines
```

بمعنى أن كل بلوك من الذاكرة الرئيسية يتم تخصيصه إلى موقع محدد داخل الـ Cache باستخدام باقي القسمة.

3. **التخزين:** عندما نحتاج إلى قراءة أو كتابة بيانات:
   - إذا كانت البيانات موجودة في الـ Cache (Cache Hit)، يتم التعامل معها مباشرة.
   - إذا لم تكن موجودة (Cache Miss)، يتم جلب البلوك من الذاكرة الرئيسية وتخزينه في الخط المحدد.

مثال بسيط لفهم العملية:

المعطيات:

- الذاكرة الرئيسية (RAM): 32 بلوك.
- ذاكرة Cache: 8 خطوط (Lines).
- كل بلوك يحتوي على كلمة واحدة.

العمليات:

1. **تحديد الـ Index:** لنفترض أننا نريد جلب البلوك رقم 12:

```
Index = Block Number % Number of Cache Lines
Index = 12 % 8 = 4
```

إذًا البلوك رقم 12 سيتم تخزينه في الخط رقم 4 داخل الـ Cache.

1. المعطيات:

- الذاكرة الرئيسية (RAM): 32 بلوك (2^5)، إذًا عنوان الذاكرة يتكون من 5 بت.
- ذاكرة Cache: 8 خطوط (2^3)، أي لدينا 8 مواقع (Lines) في الكاش.
- **كل بلوك يحتوي على كلمة واحدة:** لا يوجد Offset لأن البلوك يُعتبر كلمة واحدة.

---

2. تقسيم عنوان الذاكرة:

عنوان الذاكرة في هذه الحالة يتكون من **5 بت** ($2^5 = 32$).
يتم تقسيم العنوان إلى:

1. **Index (إندكس):** يحدد أي خط (Line) في الكاش سيتم تخزين البلوك فيه.
2. **Tag (تاج):** يحدد أي بلوك من الذاكرة الرئيسية مرتبط بهذا الخط.

Offset:

- لأن كل بلوك يحتوي على كلمة واحدة، لا يوجد حاجة لـ Offset.
- Offset = 0 بت.

3. حساب الـ Index:

- عدد الخطوط في الكاش = 8 خطوط (2^3).
- إذًا نحتاج إلى $\log_2(8) = 3$ بت لتحديد الـ Index.

---

4. حساب الـ Tag:

- عدد البتات الكلي لعنوان الذاكرة = 5 بت (لأن $2^5 = 32$).
- عدد بتات الـ Index = 3 بت.
- بتات الـ Tag هي الباقي:

$$\text{Tag} = \text{عدد البتات الكلية} - \text{Index}$$

$$\text{Tag} = 5 - 3 = 2 \text{ بت}$$

5. النتيجة النهائية:

- Index: 3 بت.
- Tag: 2 بت.
- Offset: 0 بت (لأن البلوك يحتوي على كلمة واحدة فقط).

مثال 2

المعطيات:

- RAM: 2048 بلوك (2^{11}).
- Cache: 128 خط (2^7).
- **كل بلوك يحتوي على 8 كلمات (2^3).

مثال 3

مثال لأرقام حقيقية:

المعطيات:

- الذاكرة الرئيسية (RAM): 8 كيلوبايت ($8\ \text{KB} = 2^{13}$ بايت).
- ذاكرة Cache: 1 كيلوبايت ($1\ \text{KB} = 2^{10}$ بايت).
- حجم كل بلوك: 16 بايت (2^4 بايت لكل بلوك).

الخطوات:

1. عدد البتات الكلية لعنوان الذاكرة:

- الذاكرة الرئيسية تحتوي على 2^{13} بايت.
- إذًا عنوان الذاكرة يحتاج إلى 13 بت لتمثيل جميع المواقع:
$\log_2(8192) = 13$ بت.

2. حساب الـ Offset:

- كل بلوك يحتوي على 16 بايت (2^4).
- إذًا عدد البتات اللازمة لتمثيل المواضع داخل البلوك (Offset):
$\log_2(16) = 4$ بت.

3. حساب الـ Index:

- ذاكرة الكاش حجمها 1 كيلوبايت (2^{10} بايت).
- وعدد خطوط الكاش = حجم الكاش ÷ حجم البلوك:
عدد خطوط الكاش $= \frac{2^{10}}{2^4} = 2^6$ خط.
- إذًا عدد البتات اللازمة لتمثيل خطوط الكاش (Index):
$\log_2(2^6) = 6$ بت.

4. حساب الـ Tag:

- عدد البتات الكلية = 13.
- عدد بتات الـ Index = 6.
- عدد بتات الـ Offset = 4.

$\text{Tag} = 13 - 6 - 4 = 3$ بت.

اختبار عنوان ذاكرة عملي:

لنفترض عنوان الذاكرة: 1011010010110

التقسيم:

- Tag: أول 3 بت → 101
- Index: البتات التالية 6 → 101001
- Offset: آخر 4 بت → 0110

تنويه مهم

نعم، حجم البلوك في الميموري هو نفس حجم البلوك في الكاش.

عندما نقوم بتقسيم الذاكرة الرئيسية (Main Memory) إلى بلوكات، فإن كل بلوك يتم نسخه أو نقله إلى الكاش بنفس الحجم. على سبيل المثال، إذا كان حجم البلوك في الميموري 16 بايت، فسيكون حجم البلوك في الكاش أيضًا 16 بايت.

لماذا يجب أن يكون حجم البلوك متطابقًا؟

- **الاتساق:** عند جلب البيانات من الميموري إلى الكاش، يتم جلبها على هيئة بلوكات كاملة.
- **بساطة المعالجة:** وجود بلوكات بنفس الحجم يسهل عملية إدارة الذاكرة والتخزين المؤقت (caching).
- **توافق التعيين:** البلوك الواحد من الميموري يأخذ مكانه بالكامل في الكاش دون تقسيم أو تجزئة.

إذا كانت أحجام البلوكات مختلفة، فستحدث مشكلات في عملية التخزين المؤقت وإدارة الكاش، مما يؤدي إلى تعقيدات إضافية.

امثله شامله لكل الأفكار (مهمه جدا)

Example 1: Simple Direct Mapping Calculation

Problem:

- Main Memory Size: 16 KB
- Cache Size: 1 KB
- Block Size: 16 Bytes

Solution (Detailed Explanation):

1. Calculate the number of cache lines:
   - Cache size is 1 KB = 1024 bytes.
   - Block size is 16 bytes.
   - Number of cache lines:

   This means the cache can hold 64 blocks at any given time.

2. Calculate the number of blocks in main memory:
   - Main memory size is 16 KB = 16,384 bytes.
   - Number of blocks:

   This indicates that main memory consists of 1024 blocks.

3. Apply the direct mapping formula:
   - Formula:

    - This means each block address is divided by the number of cache lines (64), and the remainder gives the cache line index.
4. Example Calculation (Block 200):
    - Block address = 200
    - Apply the formula:
    - This shows that block 200 will map to cache line 8.
5. Visualization:
    - If cache line 8 already holds another block (e.g., block 136), block 136 will be evicted, and block 200 will take its place.

---

3. Example 2: Address Breakdown

Problem:

- Address Size: 12 bits (Main Memory Size = 4 KB)
- Cache Size: 256 Bytes
- Block Size: 8 Bytes

Solution:

1. Calculate number of cache lines:
2. Offset (inside block):
3. Remaining bits for tag:
4. Address breakdown:
    - Tag: 4 bits
    - Index: 5 bits
    - Offset: 3 bits

---

4. Example 3: Cache Hit/Miss Calculation

Problem:

- Cache: 128 Bytes
- Block Size: 16 Bytes
- Memory Access Pattern: 0, 16, 32, 128, 144

Solution:

1. Cache lines:
2. Mapping:
3. Check accesses:
    - 0 (Block 0) \mod 8 = 0

    - 16 (Block 1) \mod 8 = 1
    - 32 (Block 2) \mod 8 = 2
    - 128 (Block 8) \mod 8 = 0 (Cache Line 0, Potential Conflict)}
    - 144 (Block 9) \mod 8 = 1 (Cache Line 1, Potential Conflict)}
4. Result:
    - Cache hit for addresses 0, 16, 32
    - Cache miss for 128, 144 (causing eviction)

---

5. Advanced Example: Larger Address Spaces

Problem:

- 24-bit Address Space (16 MB Main Memory)
- Cache Size: 4 KB
- Block Size: 32 Bytes

Solution:

1. Cache lines:
2. Offset:
3. Tag:
4. Address breakdown:
    - Tag: 12 bits
    - Index: 7 bits
    - Offset: 5 bits

Set Associative ثاني طريقه للتخزين

الـ **n-way set-associative** هو نوع من الـ **set-associative cache** حيث أن "n" يعني عدد الخطوط التي يمكن أن تحتوي عليها المجموعة الواحدة.

الفكرة الأساسية:

- الكاش مقسم إلى مجموعات (sets)، وكل مجموعة يمكن أن تحتوي على **n** خطوط.
- عندما نريد تخزين البيانات، نقوم بحساب المجموعة المناسبة باستخدام جزء من العنوان، ثم نضع البيانات في أحد الخطوط داخل هذه المجموعة.
- يمكن لكل مجموعة أن تحتوي على **n** خطوط، وبالتالي يمكن تخزين **n** بيانات مختلفة في نفس المجموعة.

مثال:

تخيل أن لديك كاش من 4 مجموعات (sets)، وكل مجموعة يمكن أن تحتوي على 2 خطوط (n = 2). هذا يعني أن:

- المجموعة الأولى (set 0) يمكن أن تحتوي على خطين.
- المجموعة الثانية (set 1) يمكن أن تحتوي على خطين.
- المجموعة الثالثة (set 2) يمكن أن تحتوي على خطين.
- المجموعة الرابعة (set 3) يمكن أن تحتوي على خطين.

عندما نريد تخزين البيانات، نأخذ جزء من العنوان ونحسب أين يجب أن نضع هذه البيانات بناءً على المجموعات المتاحة. فإذا كانت المجموعة 1 هي الأنسب للبيانات، يمكننا تخزين البيانات في أي من الخطين المتاحين في المجموعة 1.

5. كيف يتم تحديد مكان البيانات؟

يتم تقسيم العنوان إلى ثلاثة أجزاء رئيسية:

1. الـ **Tag**: يحدد إذا كانت البيانات في الكاش أم لا.
2. الـ **Index**: يحدد المجموعة (set) التي يجب أن تذهب إليها البيانات.
3. الـ **Block offset**: يحدد بالضبط مكان البيانات داخل الخط (line) الذي يقع في المجموعة.

كيفية عمل الـ n-way set-associative:

- أولًا، نأخذ العنوان بالكامل.
- نقسمه إلى:
- الـ **Tag**: جزء من العنوان الذي يحدد ما إذا كانت البيانات الموجودة في الكاش هي التي نبحث عنها.
- الـ **Index**: يحدد رقم المجموعة في الكاش.
- الـ **Block offset**: يحدد مكان البيانات داخل الخط.

6. المزايا والعيوب

المزايا:

- **مرونة أكبر** مقارنة بـ Direct Mapped، حيث أن هناك عدة خطوط في كل مجموعة يمكن أن تخزن البيانات.
- **أداء أفضل** مقارنة بـ Direct Mapped في بعض الحالات.

العيوب:

- يزيد التعقيد، لأننا بحاجة إلى مقارنة الـ Tag مع كل خط في المجموعة، وهذا قد يؤثر على السرعة.
- قد تحتاج الذاكرة إلى مزيد من المساحة للحفاظ على المزيد من البيانات.

مثال شامل

الفرضيات:

- **حجم الذاكرة**: 16 عنوان (نفترض أن العناوين هي من 0 إلى 15).
- **حجم الكاش**: 4 مجموعات (sets)، وكل مجموعة تحتوي على 2 خط (n = 2).
- **حجم كل خط**: 1 عنوان فقط (أي يمكن لكل خط تخزين 1 عنوان فقط).

1. تفاصيل الكاش:

- لدينا 4 مجموعات (sets) في الكاش.
- كل مجموعة يمكن أن تحتوي على 2 خط (n = 2).
- **إجمالي حجم الكاش** هو 4 مجموعات × 2 خطوط = 8 خطوط (أي 8 مواقع لتخزين البيانات).

2. تقسيم العنوان:

لنقسم العنوان إلى أجزاء بناءً على حجم الكاش:

- الـ **Index**: لأن لدينا 4 مجموعات في الكاش، نحتاج إلى 2 بت لتحديد المجموعة (لأن 4 مجموعات = 2^2).
- الـ **Tag**: باقي البتات ستكون مخصصة للـ Tag (لتحديد البيانات داخل المجموعة).
- الـ **Block offset**: بما أن كل خط يحتوي على 1 عنوان فقط، لا نحتاج إلى جزء خاص بـ Block offset.

تقسيم العنوان:

- العنوان: 4 بت (نظراً لأن لدينا 16 عنواناً، فنحتاج إلى 4 بت لتمثيلها).
- 2 بت للـ **Index** (لتحديد المجموعة).
- 2 بت للـ **Tag**.

إذن العنوان مقسم كالتالي:

- **Tag**: 2 بت
- **Index**: 2 بت
- **Block offset**: 0 بت (بما أن الخط يحتوي على عنوان واحد فقط).

نفترض أن العناوين هي من 0 إلى 15، ويتم تمثيلهم باستخدام 4 بت، كما في الجدول التالي:

العنوان	البتات (من 4 بت)	Tag (2 بت)	Index (2 بت)
0	0000	00	00
1	0001	00	01
2	0010	00	10
3	0011	00	11
4	0100	01	00
5	0101	01	01
6	0110	01	10
7	0111	01	11
8	1000	10	00
9	1001	10	01
10	1010	10	10
11	1011	10	11
12	1100	11	00
13	1101	11	01
14	1110	11	10
15	1111	11	11

5. استبدال البيانات:

إذا أردنا إضافة مزيد من العناوين (مثل العنوان 8)، فسيتم استبدال البيانات القديمة باستخدام أسلوب الاستبدال (مثل FIFO أو LRU) لأن كل مجموعة تحتوي فقط على 2 خط.

6. الوصول إلى البيانات:

إذا كان المعالج يريد الوصول إلى العنوان 2:

- يحسب الـ Index (10)، مما يعني أنه يبحث في **المجموعة 2**.
- يقارن الـ Tag (الذي هو 00) مع الـ Tag المخزن في خطوط المجموعة 2.
- إذا كانت الـ Tag متطابقة، يتم جلب البيانات من الكاش.

الفرق في الحسابات:

- Direct Mapped Cache:
- Index يتم حسابه فقط من العنوان لتحديد **المجموعة الوحيدة** التي سيُخزن فيها العنوان.
- لا يوجد **نطاق من الخطوط** داخل المجموعة؛ فكل مجموعة تحتوي على خط واحد فقط.
- n-way Set-Associative Cache:
- Index يُستخدم لتحديد **المجموعة** مثل Direct Mapped.
- لكن في هذا النوع، نقارن Tag مع **كل خط داخل المجموعة** لاختيار الخط الذي يحتوي على البيانات، لأن كل مجموعة تحتوي على أكثر من خط واحد (n خطوط).

Full Set Associative ثالث طريقه للتخزين

كيف يعمل الكاش Full Set Associative؟

في الذاكرة العادية، يتم تقسيم الكاش إلى مجموعات (sets) وتخزين البيانات داخل هذه المجموعات. في **Full Set Associative**، يُعتبر أن كل مجموعة تحتوي على جميع الأماكن المتاحة لتخزين البيانات (يُسمى هذا بوجود **عدد غير محدود من الأسطر** في المجموعة). هذا يعني أن كل عنوان في الذاكرة يمكن تخزينه في أي مكان في الكاش، بدلاً من أن يكون مرتبطًا بمجموعة معينة أو مكان ثابت كما في الأنواع الأخرى.

كيفية عمل البحث في الكاش:

عند وصول المعالج إلى عنوان في الذاكرة، يتم تقسيم هذا العنوان إلى عدة أجزاء:

- **جزء البلوك (Block Offset)**: لتحديد مكان البيانات داخل البلوك.
- **جزء المجموعات (Set Index)**: في Full Set Associative، لا يوجد جزء مخصص لهذا (لأنه يمكن وضع البيانات في أي مجموعة).
- **جزء التعرف (Tag)**: لتحديد البيانات داخل الكاش.

مميزات الكاش Full Set Associative:

1. **مرونة أكبر**: بما أن البيانات يمكن تخزينها في أي سطر من مجموعة الكاش، فإن هذا يقلل من عدد الإلغاءات (misses) في الكاش.
2. **زيادة الأداء**: يمكن الوصول إلى البيانات بسرعة أكبر في حال وجودها في الكاش، مما يحسن من سرعة المعالج.

عيوب الكاش Full Set Associative:

1. **تعقيد أكبر**: إدارة الكاش بهذا الشكل يتطلب أن يكون هناك المزيد من الحسابات والبحث في جميع الأسطر ضمن الكاش، مما يزيد من تعقيد الدوائر الكهربائية واحتياجات الطاقة.
2. **تكلفة أعلى**: بما أن هناك حاجة للبحث في كل الأسطر، فإن ذلك يزيد من تكلفة تصميم الكاش.
3. **البحث في الكاش**: عملية البحث في جميع الأسطر من أجل العثور على البيانات المطلوبة يمكن أن تكون بطيئة مقارنة ببعض أنواع الكاش الأخرى.

مثال توضيحي:

إذا كانت لديك مجموعة مكونة من 4 أسطر في الكاش، وفي حال كنت تستخدم نظام **Full Set Associative**، فيمكنك تخزين البيانات في أي من هذه الأسطر. في حالة حدوث تضارب (cache miss)، سيضطر النظام إلى البحث في جميع الأسطر لمعرفة إذا كانت البيانات موجودة أو لا، وهذا قد يبطئ الأداء قليلاً بالمقارنة مع الأنظمة الأخرى.