1 Caching

Een 64KiB, direct mapped cache heeft 16 byte blocks. Je mag veronderstellen dat het geheugen byte geadresseerd is (dus 1 woord = 1 byte). Als adressen 32 bits lang zijn, hoeveel bits zijn er dan voorzien voor de tag, de index, en de offset in deze cache?

Laten we het probleem stap voor stap oplossen.

1. Gegeven informatie:

- Cachegrootte: $64\,\text{KiB} = 64 \times 1024 = 65{,}536\,\text{bytes}$
- Blokgrootte: $16\,\text{bytes}$ per blok
- Adreslengte: $32\,\text{bits}$
- Cache is **direct mapped**.

2. Berekeningen:

a. Aantal blokken in de cache:

$$\text{Aantal blokken} = \frac{\text{Cachegrootte}}{\text{Blokgrootte}} = \frac{65{,}536}{16} = 4{,}096\,\text{blokken}.$$

b. Aantal bits voor de offset: De offset bepaalt welk byte binnen een blok wordt aangesproken. Omdat elk blok $16\,\text{bytes}$ bevat, zijn er:

$$\text{Offset-bits} = \log_2(16) = 4\,\text{bits}.$$

c. Aantal bits voor de index: De index bepaalt welk blok in de cache wordt aangesproken. Met 4,096 blokken zijn er:

$$\text{Index-bits} = \log_2(4{,}096) = 12\,\text{bits}.$$

d. Aantal bits voor de tag: De tag identificeert of een bepaald geheugenadres in de cache staat. Het aantal tag-bits wordt bepaald door het totale aantal adresbits (32) minus de offset- en index-bits:

$$\text{Tag-bits} = 32 - (\text{Index-bits} + \text{Offset-bits}) = 32 - (12 + 4) = 16\,\text{bits}.$$

3. Samenvatting:

- **Offset**: $4\,\text{bits}$
- **Index**: $12\,\text{bits}$
- **Tag**: $16\,\text{bits}$

Adresindeling:

Het 32-bits geheugenadres wordt als volgt opgedeeld:

[Tag (16 bits)] [Index (12 bits)] [Offset (4 bits)]

Pas je antwoord aan indien de cache 4-way set associatief is in plaats van direct.

Als de cache **4-way set associatief** is, verandert de manier waarop de cache wordt georganiseerd. Laten we de berekeningen herzien.

---

1. Gegeven informatie:

- Cachegrootte: 64 KiB = 65,536 bytes
- Blokgrootte: 16 bytes per blok
- Adreslengte: 32 bits
- **4-way set associatieve cache**.

---

2. Berekeningen:

a. Aantal blokken in de cache: De totale cachegrootte en blokgrootte zijn onveranderd, dus:

$$\text{Aantal blokken} = \frac{\text{Cachegrootte}}{\text{Blokgrootte}} = \frac{65{,}536}{16} = 4{,}096 \,\text{blokken}.$$

b. Aantal sets in de cache: In een 4-way set associatieve cache worden de blokken verdeeld over sets, met 4 blokken per set. Het aantal sets wordt:

$$\text{Aantal sets} = \frac{\text{Aantal blokken}}{\text{Aantal associatieve wegen}} = \frac{4{,}096}{4} = 1{,}024 \,\text{sets}.$$

c. Aantal bits voor de offset: De offset bepaalt welk byte binnen een blok wordt aangesproken. Dit is hetzelfde als bij de direct mapped cache:

$$\text{Offset-bits} = \log_2(\text{Blokgrootte}) = \log_2(16) = 4 \,\text{bits}.$$

d. Aantal bits voor de index: De index bepaalt naar welke set in de cache wordt verwezen. Aangezien er 1,024 sets zijn:

$$\text{Index-bits} = \log_2(\text{Aantal sets}) = \log_2(1{,}024) = 10 \,\text{bits}.$$

e. Aantal bits voor de tag: De tag identificeert welk blok in het geheugen overeenkomt met een cacheblok. Het aantal tag-bits wordt bepaald door het totale aantal adresbits (32) minus de offset- en index-bits:

$$\text{Tag-bits} = 32 - (\text{Index-bits} + \text{Offset-bits}) = 32 - (10 + 4) = 18 \,\text{bits}.$$

---

3. Samenvatting:

- **Offset**: 4 bits
- **Index**: 10 bits
- **Tag**: 18 bits

Adresindeling:

Het 32-bits geheugenadres wordt nu als volgt opgedeeld:

[Tag (18 bits)] [Index (10 bits)] [Offset (4 bits)]

Hoeveel bits hebben we voor de index nodig indien de cache volledig associatief is? Verklaar je antwoord.

In een **volledig associatieve cache** zijn er **geen sets of indices**. Elk blok in de cache kan worden toegewezen aan elk geheugenadres, zonder beperkingen van een index. Laten we dit uitleggen en berekenen:

---

1. Gegeven informatie:

- Cachegrootte: 64 KiB = 65,536 bytes
- Blokgrootte: 16 bytes per blok
- Adreslengte: 32 bits
- Cache is **volledig associatief**.

---

2. Volledig associatieve cache:

In een volledig associatieve cache zijn **alle blokken onderdeel van één grote set**. Hierdoor is er **geen aparte index nodig om een set te selecteren**. De bits van het geheugenadres worden als volgt verdeeld:

a. Offset-bits: De offset bepaalt welk byte binnen een blok wordt aangesproken. Dit is hetzelfde als in andere configuraties:

$$\text{Offset-bits} = \log_2(\text{Blokgrootte}) = \log_2(16) = 4 \text{ bits}.$$

b. Geen index: Omdat alle blokken deel uitmaken van één grote set, hebben we geen index nodig:

$$\text{Index-bits} = 0.$$

c. Tag-bits: De rest van het adres wordt gebruikt voor de tag, om unieke geheugenadressen te identificeren. Het aantal tag-bits is dus:

$$\text{Tag-bits} = 32 - (\text{Index-bits} + \text{Offset-bits}) = 32 - (0 + 4) = 28 \text{ bits}.$$

---

3. Samenvatting:

- **Offset**: 4 bits
- **Index**: 0 bits
- **Tag**: 28 bits

Verklaring:

In een volledig associatieve cache wordt de gehele cache beschouwd als één grote set. Daarom is er geen noodzaak voor een index om een specifieke set te selecteren, en alle bits behalve de offset worden toegewezen aan de tag.

Een 8 byte, 2-way set associatieve cache (met LRU replacement) met 2 byte blocks ontvangt requests voor de volgende (binaire) adressen:

0110,0000,0010,0001,0011,0100,1001,0000,1010,1111,0111. Geef na elke access de inhoud van de cache door de bijgevoegde tabel in te vullen, voor elke set moet je de aanwezige tags weergeven.

We zullen de gevraagde oefening stap voor stap oplossen, door te kijken hoe de cache-inhoud verandert na elke access.

Cache Details:

1. **Cache Size:** 8 bytes
2. **Block Size:** 2 bytes (4 blocks totaal in de cache)
3. **Associativity:** 2-way set associative → 2 sets met 2 blokken per set.
4. **Replacement Policy:** LRU (Least Recently Used).
5. **Address Breakdown:**
   - Block Offset (1 bit): bepaalt welk deel van het blok wordt aangesproken.
   - Index (1 bit): bepaalt de set (0 of 1).
   - Tag: wat overblijft na de offset en index bits.

De gegeven adressen zijn in binair:

- **Offset bit (1 bit):** Laatste bit.
- **Index bit (1 bit):** Eén-na-laatste bit.
- **Tag:** Rest van de bits.

Stap 1: Splits adressen in Tag, Index, Offset

Adres	Tag	Index	Offset
0110	01	1	0
0000	00	0	0
0010	00	1	0
0001	00	0	1
0011	00	1	1
0100	01	0	0
1001	10	0	1
0000	00	0	0
1010	10	1	0
1111	11	1	1
0111	01	1	1

Stap 2: Cache Invullen

Initieel:

- **Set 0:** Leeg.
- **Set 1:** Leeg.

Accesses:

1. **0110** (Tag: 01, Set: 1)
   - Miss: Voeg Tag 01 toe aan Set 1.
   - Cache:
     - **Set 0:** -
     - **Set 1:** 01/-
2. **0000** (Tag: 00, Set: 0)
   - Miss: Voeg Tag 00 toe aan Set 0.
   - Cache:

- **Set 0:** 00/-
- **Set 1:** 01/-

3. **0010** (Tag: 00, Set: 1)
   - Miss: Voeg Tag 00 toe aan Set 1.
   - Cache:
     - **Set 0:** 00/-
     - **Set 1:** 01/00
4. **0001** (Tag: 00, Set: 0)
   - Miss: Tag 00 al aanwezig in Set 0 (geen plaatswissel nodig).
   - Cache:
     - **Set 0:** 00/-
     - **Set 1:** 01/00
5. **0011** (Tag: 00, Set: 1)
   - Hit: Tag 00 al aanwezig in Set 1.
   - Cache blijft hetzelfde:
     - **Set 0:** 00/-
     - **Set 1:** 01/00
6. **0100** (Tag: 01, Set: 0)
   - Miss: Voeg Tag 01 toe aan Set 0 (LRU: verwijder oudste, dus hier geen conflict).
   - Cache:
     - **Set 0:** 01/00
     - **Set 1:** 01/00
7. **1001** (Tag: 10, Set: 0)
   - Miss: Voeg Tag 10 toe aan Set 0 (verwijder Tag 00 door LRU).
   - Cache:
     - **Set 0:** 10/01
     - **Set 1:** 01/00
8. **0000** (Tag: 00, Set: 0)
   - Miss: Voeg Tag 00 opnieuw toe aan Set 0 (verwijder LRU: Tag 01).
   - Cache:
     - **Set 0:** 00/10
     - **Set 1:** 01/00
9. **1010** (Tag: 10, Set: 1)
   - Miss: Voeg Tag 10 toe aan Set 1 (verwijder LRU: Tag 01).
   - Cache:
     - **Set 0:** 00/10
     - **Set 1:** 10/00
10. **1111** (Tag: 11, Set: 1)
    - Miss: Voeg Tag 11 toe aan Set 1 (verwijder LRU: Tag 00).
    - Cache:
      - **Set 0:** 00/10
      - **Set 1:** 11/10
11. **0111** (Tag: 01, Set: 1)
    - Miss: Voeg Tag 01 toe aan Set 1 (verwijder LRU: Tag 10).
    - Cache:
      - **Set 0:** 00/10
      - **Set 1:** 01/11

Tabel Invullen

Adres	Set 0	Set 1	Hit/Miss
0110	-	01/-	Miss
0000	00/-	01/-	Miss
0010	00/-	01/00	Miss
0001	00/-	01/00	Miss
0011	00/-	01/00	Hit
0100	01/00	01/00	Miss
1001	10/01	01/00	Miss
0000	00/10	01/00	Miss
1010	00/10	10/00	Miss
1111	00/10	11/10	Miss
0111	00/10	01/11	Miss

3M beslist Post-Its te maken door gele vierkanten op wit papier te printen, waarbij voor elk punt in het vierkant de CMYK (cyan, magenta, yellow, black) waarde moet worden ingevuld. 3M neemt je in dienst om drie implementaties te bekijken op een machine met een 2048-byte direct mapped data cache met 32 byte blokken/lijnen. Je krijgt volgende definities mee:

```
struct point_color {
    int c;
    int m;
    int y;
    int k;
};
struct point_color square[16][16];
register int i, j;
```

Veronderstel verder dat

- sizeof(int) = 4, square begint op memory address 0,
- de cache is initieel leeg,
- de enige memory accessen zijn naar de entries van de array square. De variabelen i en j zijn opgeslagen in de registers.
- het geheugen 4-byte geadresseerd wordt, dus 1 int = 1 woord.

Wat is het percentage van de writes in de volgende code die een hit veroorzaken?

Methode 1:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].y = 1;
        square[i][j].k = 0;
    }
}
```

Methode 2:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[j][i].c = 0;
        square[j][i].m = 0;
        square[j][i].y = 1;
        square[j][i].k = 0;
    }
}
```

Methode 3:

```
for (i=0; i<16; i++) {
    for (j=0; j<16; j++) {
        square[j][i].y = 1;
    }
}

for (i=0; i<16; i++) {
    for (j=0; j<16; j++) {
        square[j][i].c = 0;
        square[j][i].m = 0;
        square[j][i].k = 0;
    }
}
```

Laten we de details bekijken en berekenen waarom de gegeven hit rates voor elke methode kloppen. We analyseren de geheugenorganisatie en cachetoegangen.

Algemene Informatie

- Structuurgrootte: Een `point_color`-struct bevat 4 integers van elk 4 bytes. De grootte van de struct is dus $4 \times 4 = 16$ bytes.
- Array-organisatie: De `square`-array is een 16×16 tweedimensionale array. Elke rij bevat 16 structs, wat betekent dat een rij $16 \times 16 = 256$ bytes inneemt. Elke kolom bevat ook 16 structs, maar ze zijn verspreid met een stapgrootte van 256 bytes (afstand tussen structs in dezelfde kolom).
- Cacheconfiguratie:
  - Cachegrootte: 2048 bytes.
  - Cachelijn: 32 bytes (dat wil zeggen dat elk cacheblok 2 structs volledig kan opslaan, omdat 2 structs = 32 bytes).

De cache is direct mapped. Het aantal cachelijnen is $\frac{2048}{32} = 64$.

Belangrijk geheugenpatroon: Omdat struct `square` op adres 0 begint en de cache direct mapped is, zijn de adressen van struct-onderdelen die dicht bij elkaar liggen waarschijnlijk conflicterend in dezelfde cachelijn. Dit veroorzaakt cache misses bij minder voordelige toegangspatronen.

Methode 1

Codepatroon: We itereren rij-per-rij (`square[i][j]`), wat betekent dat we opeenvolgend door het geheugen lopen. Elke rij bevat 256 bytes, terwijl een cacheblok 32 bytes bevat. Dit resulteert in:

- $\frac{256}{32} = 8$ cachelijnen per rij.
- Elke rij heeft 16 structs. Per 2 structs is er een cache hit.

Cachetoegang per rij:

- De eerste struct (adres 0) van de rij veroorzaakt een miss.
- Voor de volgende 7 structs in dezelfde cachelijn zijn er hits.
- Dit patroon herhaalt zich 8 keer per rij.

Aantal hits per rij:

- $7 \times 8 = 56$ hits per rij (7 hits per cachelijn $\times$ 8 cachelijnen).

Totale toegang (writes):

- $16 \times 16 = 256$ writes.

Hitpercentage:

- $\frac{56 \times 16}{256} = \frac{7}{8}$.

Conclusie: Het hitpercentage is 7/8.

Methode 2

Codepatroon: We itereren kolom-per-kolom (`square[j][i]`), wat betekent dat we toegang hebben tot niet-contigu geheugen. Elke struct in een kolom ligt 256 bytes van de volgende af, wat cache-conflicten veroorzaakt omdat struct-geheugen van dezelfde kolom naar dezelfde cachelijn wordt gemapt.

Cachetoegang per kolom:

- De eerste struct van een kolom (bijv. `square[0][i]`) veroorzaakt een miss.
- Voor elke andere struct in de kolom ($16 - 1 = 15$) is er ook een miss, omdat $256 \mod 2048 = 256$. Alle struct-geheugens in dezelfde kolom mappen naar dezelfde cachelijn.

Aantal hits per kolom:

- Slechts de eerste write naar een nieuwe cachelijn veroorzaakt een miss.
- Voor de 16 kolommen is er $\frac{1}{4}$ hit rate.

Totale toegang (writes):

- $16 \times 16 = 256$.

Hitpercentage:

- $\frac{3}{4}$.

Laat T1 de access tijd voorstellen indien een woord zich in de L1 cache bevindt. Indien het woord enkel in de L2 cache aanwezig is en niet in de L1 cache, dan is er een tijd T2 nodig om het woord over te brengen naar de L1 cache. Tenslotte, is er een tijd TM nodig om een woord van het main geheugen naar de L2 cache te verplaatsen. Stel dat T1 = 20 nsec, T2 = 100 nsec en TM = 400 nsec.

Vraag 1: Indien we respectievelijk een hit rate van.9 en.5 hebben in de L1 en L2 cache, wat is dan de gemiddelde access time? Vraag 2 Indien de hit rate in de L2 cache.6 is, wat is dan de hit rate in de L1 cache indien de gemiddelde access time gelijk is aan 33 nsec?

Vraag 1: Gemiddelde toegangstijd berekenen

We berekenen de gemiddelde toegangstijd T_avg gegeven:

- Hit rate in L1 cache: $H_1 = 0.9$
- Hit rate in L2 cache: $H_2 = 0.5$
- Toegangstijd L1 cache: $T_1 = 20$ nsec
- Toegangstijd L2 cache (indien miss in L1): $T_2 = 100$ nsec
- Toegangstijd hoofdgeheugen (indien miss in L2): $T_M = 400$ nsec

De formule is als volgt:

$$\text{T\_avg} = H_1 \cdot T_1 + (1 - H_1) \cdot H_2 \cdot (T_1 + T_2) + (1 - H_1) \cdot (1 - H_2) \cdot (T_1 + T_2 + T_M)$$

Invullen van waarden:

$$\text{T\_avg} = 0.9 \cdot 20 + (1 - 0.9) \cdot 0.5 \cdot (20 + 100) + (1 - 0.9) \cdot (1 - 0.5) \cdot (20 + 100 + 400)$$

Stap voor stap uitwerken:

1. Eerste term: $0.9 \cdot 20 = 18$
2. Tweede term: $0.1 \cdot 0.5 \cdot (120) = 0.05 \cdot 120 = 6$
3. Derde term: $0.1 \cdot 0.5 \cdot (520) = 0.05 \cdot 520 = 26$

Totaal:

$$\text{T\_avg} = 18 + 6 + 26 = 50\,\text{nsec}$$

Vraag 2: Hit rate in L1 berekenen gegeven T_avg $= 33$ nsec

We lossen nu H_1 op, gegeven dat:

- T_avg $= 33$
- $H_2 = 0.6$

De formule blijft:

$$\text{T\_avg} = H_1 \cdot T_1 + (1 - H_1) \cdot H_2 \cdot (T_1 + T_2) + (1 - H_1) \cdot (1 - H_2) \cdot (T_1 + T_2 + T_M)$$

Invullen:

$$33 = H_1 \cdot 20 + (1 - H_1) \cdot 0.6 \cdot 120 + (1 - H_1) \cdot 0.4 \cdot 520$$

Vereenvoudigen:

$$33 = 20H_1 + (1 - H_1) \cdot 72 + (1 - H_1) \cdot 208$$

$$33 = 20H_1 + 72 - 72H_1 + 208 - 208H_1$$

$$33 = 280 - 260H_1$$

Los H_1 op:

$$260H_1 = 280 - 33$$

$$260H_1 = 247$$

$$H_1 = \frac{247}{260} \approx 0.95$$

Resultaten:

1. Gemiddelde toegangstijd in Vraag 1: **50 nsec**
2. Hit rate in L1 cache in Vraag 2: **0.95 (95%)**

Direct mapping versus set-associative mapping met random replacement

Beschouw het volgende stuk code uit de theorieslides.

```
float a[1024], b[1024];
for (i=0; i<1024; i++)
    sum += a[i] * b[i];
```

Veronderstel eveneens:

- Cache size: 4 KByte, Block size: 32 byte, Word size: 4 byte
- Float: 4 byte
- b zit vlak na a in het geheugen
- i en sum zijn registers

Wat is de hit rate als we direct-mapping veronderstellen ?

Analyse van de situatie Gegeven:

- **Cache size:** 4 KB $= 4 \times 1024 = 4096$ bytes
- **Block size:** 32 bytes
- **Word size:** 4 bytes
- **Float size:** 4 bytes
- **Arrays:** $a[1024]$ en $b[1024]$ bevinden zich opeenvolgend in het geheugen.

Afleidingen:

1. **Aantal cacheblocks:**

$$\text{Aantal cacheblocks} = \frac{\text{Cache size}}{\text{Block size}} = \frac{4096}{32} = 128 \text{ blocks.}$$

2. **Aantal floats per cacheblock:**

$$\text{Aantal floats per cacheblock} = \frac{\text{Block size}}{\text{Float size}} = \frac{32}{4} = 8 \text{ floats.}$$

3. **Direct-mapped cache:**
   - Elke cacheblock heeft een vaste plek in de cache, bepaald door:

$$\text{Index} = (\text{Memory address}/\text{Block size}) \mod \text{Aantal cacheblocks.}$$

   - Dit betekent dat floats met een afstand van 128 cacheblocks elkaar zullen verdringen (cache conflict).

4. **Gegeven over arrays a en b:**
   - De arrays a en b bevatten elk 1024 floats, wat overeenkomt met $1024 \times 4 = 4096$ bytes per array.
   - Omdat b direct na a in het geheugen zit, geldt:
     - $a[i]$ en $b[i]$ liggen steeds in opeenvolgende geheugenblokken.

Hit rate berekening Bij direct-mapped caching worden blokken van a en b mogelijk naar dezelfde cachelocatie gemapt.

1. **Blokken in het geheugen:**
   - a en b samen gebruiken:

$$\text{Geheugengrootte van } a \text{ en } b = 4096 \,(\text{voor } a) + 4096 \,(\text{voor } b) = 8192 \,\text{bytes.}$$

   - Aantal blokken in het geheugen:

$$\frac{\text{Geheugengrootte van } a \text{ en } b}{\text{Block size}} = \frac{8192}{32} = 256 \text{ blokken.}$$

2. **Conflict tussen $a[i]$ en $b[i]$:**
   - Omdat er slechts 128 cacheblocks zijn, zullen geheugenblokken i en $i + 128$ dezelfde cacheplek innemen.
   - Elk blok van a conflicteert met het corresponderende blok van b.

3. **Hit rate berekening:**
   - Bij de eerste toegang tot elk blok wordt dit in de cache geladen (miss).
   - De tweede toegang tot een element in hetzelfde blok is een hit, zolang er geen conflict is.
   - Vanwege directe mapping en de volgorde waarin $a[i]$ en $b[i]$ worden geaccessed, zal elk blok een miss veroorzaken bij $b[i]$, omdat de cacheplek inmiddels door een ander blok van a is overschreven.

Conclusie:

- Elke $a[i]$ toegang resulteert in een miss.
- Elke $b[i]$ toegang resulteert eveneens in een miss.
- Dit betekent dat de **hit rate = 0%.**

Als we een 4-way set-associatief geheugen met random replacement veronderstellen, bereken de (benaderde) hit rate.

Analyse van de situatie bij een 4-way set-associatieve cache

Gegeven:

- **Cache configuratie:**
  - Cache size = 4096 bytes
  - Block size = 32 bytes
  - 4-way set-associative cache:
    - Elke set kan 4 cacheblokken bevatten.
    - Aantal sets in de cache:

$$\text{Aantal sets} = \frac{\text{Cache size/Block size}}{\text{Aantal ways}} = \frac{4096/32}{4} = 32 \text{ sets.}$$

Afleidingen:

- Het geheugen wordt opgedeeld in blokken van 32 bytes.
- De **set-index** wordt bepaald door:

$$\text{Set-index} = (\text{Memory address/Block size}) \mod \text{Aantal sets.}$$

- Blokken $a[i]$ en $b[i]$ kunnen nu in dezelfde set vallen, maar doordat een set 4 slots bevat, kunnen blokken van zowel a als b samen in de cache passen zonder elkaar direct te vervangen.

Gedrag van de cache tijdens de iteraties

1. **Eerste 8 iteraties:**
   - **Geheugenstructuur:**
     - Blok $a[0] \dots a[7]$ en blok $b[0] \dots b[7]$ worden geaccessed.
   - **Cachetoestand:**
     - In de eerste toegang tot $a[0]$, is dit een miss. Het blok met $a[0] \dots a[7]$ wordt geladen in de cache.
     - In de volgende toegang tot $b[0]$, is dit een miss. Het blok met $b[0] \dots b[7]$ wordt geladen in de cache.
   - **Blokvervanging:**
     - De set kan maximaal 4 blokken bevatten. Bij de eerste iteraties is er nog voldoende ruimte voor zowel a als b zonder vervanging.
2. **Latere iteraties:**
   - Zodra de cache gevuld is met een set van $a[i]$ en $b[i]$, kan vervanging plaatsvinden door **random replacement**.
   - Random replacement veroorzaakt slechts sporadisch vervanging tussen a en b, waardoor de kans groot is dat blokken van a en b in de cache blijven.

Hit rate berekening

Om de benaderde hit rate te bepalen, bekijken we de **boomstructuur** van het gedrag:

De eerste 8 iteraties:

- Bij de eerste toegang tot een blok van a, treedt een miss op.
- Bij de eerste toegang tot een blok van b, treedt een miss op.

- Na deze iteraties bevinden zich zowel $a[i]$ als $b[i]$ in de cache. Verdere toegang tot a of b is een **hit**, tenzij er een random vervanging optreedt.

Boomstructuur:

- **Linkertak:** "Beide blokken (a en b) blijven in de cache."
- **Rechtertak:** "Een blok (a of b) wordt vervangen."

De kans dat een blok wordt vervangen is relatief klein omdat er meerdere vrije plekken zijn in de cache (tot 4 per set). Daarom is de hit rate na de eerste paar misses **hoog**.

Benadering van de hit rate

1. **Misses in de eerste iteraties:**
   - Elke blok wordt 1 keer ingeladen: 2 misses (voor $a[0]$ en $b[0]$).
   - Voor de eerste 8 iteraties: 16 elementen worden geaccessed, wat neerkomt op **4 blokken** (a) en **4 blokken** (b).

   $$\text{Aantal misses in eerste 8 iteraties} = 8 \text{ misses (4 voor } a \text{ en 4 voor } b).$$

2. **Hits en misses na de eerste iteraties:**
   - Kans op een miss per access wordt kleiner omdat beide blokken meestal in de cache aanwezig zijn.
   - Aangenomen dat er een kleine kans op vervanging is, schatten we:
     - Hit rate na eerste iteraties $\approx 90\%$.
3. **Totaal:**
   - In de lange termijn domineert de **hoge hit rate** van 90%.
   - De totale **gemiddelde hit rate** (rekening houdend met de eerste misses) wordt geschat op:

   $$\text{Hit rate} \approx 85\%.$$

Conclusie: De geschatte **hit rate is ongeveer 85%.**

Pas je antwoord op vorige 2 vragen aan, als je weet dat beide arrays elk slechts 256 elementen bevatten. Doe dit eveneens voor 1536.

Geval: 256 elementen per array

- **Relatieve positie in het geheugen:**
  - Elk array gebruikt $256 \times 4 = 1024$ bytes.
  - Samen gebruiken ze $2 \times 1024 = 2048$ bytes, wat gelijk is aan $\frac{1}{4}$ van de cachegrootte (4096 bytes).
  - De blokken van $a[i]$ en $b[i]$ worden dus **op dezelfde sets gemapt**.
- **Gedrag van de cache:**
  - Thrashing (conflicten) treedt **niet meer op** omdat $a[i]$ en $b[i]$ afzonderlijk in de sets passen.
  - Elk blok bevat 8 floats. Na de eerste miss in een blok volgen 7 hits bij toegang tot dezelfde array.
- **Hit rate:**
  - Per blok is de hit rate $\frac{7}{8}$ (7 hits, 1 miss).
  - Omdat er geen thrashing optreedt, geldt dit ook voor de gehele berekening.

  Conclusie voor 256 elementen bij direct mapping: Hit rate = $7/8$ = 87.5%.

Geval: 1536 elementen per array

- **Relatieve positie in het geheugen:**
  - Elk array gebruikt $1536 \times 4 = 6144$ bytes.
  - Samen gebruiken ze $2 \times 6144 = 12288$ bytes, wat gelijk is aan $\frac{6}{4}$ van de cachegrootte.
  - Dit is een veelvoud van $\frac{1}{4}$ van de cachegrootte, dus $a[i]$ en $b[i]$ worden opnieuw **op dezelfde sets gemapt**.
- **Gedrag van de cache:**
  - Net als bij 256 elementen is er geen thrashing omdat de cache voldoende groot is om de benodigde blokken te behouden.
- **Hit rate:**
  - Per blok is de hit rate $\frac{7}{8}$ (7 hits, 1 miss), hetzelfde gedrag als bij 256 elementen.

Conclusie voor 1536 elementen bij direct mapping: Hit rate = $7/8$ = 87.5%.

4-way set-associative cache Voor zowel 256 als 1536 elementen blijft het gedrag hetzelfde als in het oorspronkelijke scenario met 1024 elementen. Dit komt omdat:

1. Beide arrays worden op dezelfde sets gemapt, maar de 4-way associativiteit biedt genoeg ruimte om blokken van zowel $a[i]$ als $b[i]$ in dezelfde set te bewaren zonder conflicten.
2. Random replacement blijft zeldzaam, dus de **gemiddelde hit rate** na de initiële misses blijft hoog.

Conclusie voor 256 en 1536 elementen bij 4-way set-associative cache: Hit rate ≈ 85.

Samenvatting

Scenario	Direct Mapping Hit Rate	4-way Set-Associative Hit Rate
256 elementen per array	87.5%	85%
1024 elementen per array	0%	85%
1536 elementen per array	87.5%	85%