Εργασία Αρχιτεκτονική Υπολογιστών

Ονοματεπώνυμο: Ασημάκης Κύδρος
ΑΕΜ: 3881

8 Φεβρουαρίου 2022

1 Πρώτο Ζητούμενο

Από τα δεδομένα καταλαβαίνουμε ότι η cache θα έχει

$$\frac{32}{4} = 8 \quad words/block$$

καθώς και ότι θα έχει

$$\frac{256}{32} = 8 \quad blocks$$

άρα και 4 sets.

Συμπεραίνουμε λοιπόν πως ισχύουν:

$$L_{index} = log_2(number\_sets) = log_2(4) = 2 \quad bits \tag{1}$$

$$L_{offset} = log_2(block\_size) = log_2(32) = 5 \quad bits \tag{2}$$

$$00008000_{16} = 00000000000000001000000000000000_2 \tag{3}$$

$$=> L_{address} = 32 \quad bits$$

$$L_{tag} = L_{address} - L_{index} - L_{offset} = 32 - 2 - 5 = 25 \quad bits \tag{4}$$

Ισχύει γενικά πως, για στοιχείο πίνακα A[i][j], $address(A[i][j]) = Address(A) + i * N_{columns} * sizeof(element) + j * sizeof(element)$.
Οπότε, για imm1 = 8 και imm2 = 1 , έχουμε τον τύπο:

$address(A[i][0]) = 00000000000000001000000000000000_2 + 100_2 * i$

Αφού γνωρίζουμε πως οι πίνακες είναι ευθυγραμμισμένοι, καταλήγουμε στην παρακάτω αντιστοιχία:

```
/**
 *  Array slots and their addresses
 */
A[0][0]; //0000 0000 0000 0000 1000 0000 0000 0000 ,cache #1: index == 00, offset == 00000
A[1][0]; //0000 0000 0000 0000 1000 0000 0000 0100 ,cache #1: index == 00, offset == 00100
A[2][0]; //0000 0000 0000 0000 1000 0000 0000 1000 ,cache #1: index == 00, offset == 01000
A[3][0]; //0000 0000 0000 0000 1000 0000 0000 1100 ,cache #1: index == 00, offset == 01100
A[4][0]; //0000 0000 0000 0000 1000 0000 0001 0000 ,cache #1: index == 00, offset == 10000
A[5][0]; //0000 0000 0000 0000 1000 0000 0001 0100 ,cache #1: index == 00, offset == 10100
A[6][0]; //0000 0000 0000 0000 1000 0000 0001 1000 ,cache #1: index == 00, offset == 11000
A[7][0]; //0000 0000 0000 0000 1000 0000 0001 1100 ,cache #1: index == 00, offset == 11100

B[0][0]; //0000 0000 0000 0000 1000 0000 0010 0000 ,cache #1: index == 01, offset == 00000
B[1][0]; //0000 0000 0000 0000 1000 0000 0010 0100 ,cache #1: index == 01, offset == 00100
B[2][0]; //0000 0000 0000 0000 1000 0000 0010 1000 ,cache #1: index == 01, offset == 01000
B[3][0]; //0000 0000 0000 0000 1000 0000 0010 1100 ,cache #1: index == 01, offset == 01100
B[4][0]; //0000 0000 0000 0000 1000 0000 0011 0000 ,cache #1: index == 01, offset == 10000
B[5][0]; //0000 0000 0000 0000 1000 0000 0011 0100 ,cache #1: index == 01, offset == 10100
B[6][0]; //0000 0000 0000 0000 1000 0000 0011 1000 ,cache #1: index == 01, offset == 11000
B[7][0]; //0000 0000 0000 0000 1000 0000 0011 1100 ,cache #1: index == 01, offset == 11100
```

Η διαδικασία του κώδικα φαίνεται καλύτερα αν τον "ξεδιπλώσουμε":

```
/**
 *  Actual code ran
 */
A[1][0] = A[1][0] + A[0][0] + B[0][0]; //i == 0
A[2][0] = A[1][0] + B[0][0];           //i == 1
A[3][0] = A[3][0] + A[2][0] + B[2][0]; //i == 2
A[4][0] = A[3][0] + B[2][0];           //i == 3
A[5][0] = A[5][0] + A[4][0] + B[4][0]; //i == 4
A[6][0] = A[5][0] + B[4][0];           //i == 5
A[7][0] = A[7][0] + A[6][0] + B[6][0]; //i == 6
```

γνωρίζοντας παράλληλα πως σε επίπεδο assembly, οι θέσεις των πινάκων διαβάζονται όπως φαίνεται, δηλαδή στην πρώτη γραμμή θα διαβαστούν πχ A[1][0], A[0][0], B[0][0], A[1][0].

Αρχίζουμε λοιπόν την διαδικασία με την παρακάτω, αρχικά άδεια, cache:

index	block1	block2
00		
01		
10		
11		

Read: A[1][0], INDEX = 00, MISS
ALLOCATE

Η cache έπειτα από την πρώτη εντολή λοιπόν μοιάζει:

index	block1	block2
00	A[0][0] A[1][0] A[2][0] A[3][0] A[4][0] A[5][0] A[6][0] A[7][0]	
01		
10		
11		

Αφού κάθε block χωράει 8 λέξεις και οι 7 γειτονικές του A[1][0] έχουν το ίδιο index.

Read: A[0][0], INDEX = 00, HIT
Read: B[0][0], INDEX = 01, MISS
ALLOCATE

Έπειτα από την τρίτη εντολή μοιάζει:

index	block1	block2
00	A[0][0] A[1][0] A[2][0] A[3][0] A[4][0] A[5][0] A[6][0] A[7][0]	
01	B[0][0] B[1][0] B[2][0] B[3][0] B[4][0] B[5][0] B[6][0] B[7][0]	
10		
11		

Εύκολα φαίνεται πως από εδώ και πέρα όλα τα reads θα είναι hits, αφού και οι 2 πίνακες βρίσκονται ολάκεροι στην cache και δεν υπάρχει άλλη μεταβλητή για να τους διώξει

από εκεί. Έχουμε τελικά για το πρώτο ζητούμενο 23 hits και 2 miss.

2 Δεύτερο Ζητούμενο

Αφού τώρα το block έχει μέγεθος 16 bits, έχουμε:

$$\frac{16}{4} = 4 \quad words/block$$

$$\frac{256}{16} = 16 \quad blocks = 8 \quad sets$$

$$L'_{index} = log_2(8) = 3 \quad bits \tag{5}$$

$$L'_{offset} = log_2(16) = 4 \quad bits \tag{6}$$

$$L'_{tag} = 32 - 4 - 3 = 25 \quad bits \tag{7}$$

και αναγνωρίζουμε την καινούρια αντιστοιχία:

```
/**
 *  Array slots and their addresses
 */
A[0][0]; //0000 0000 0000 0000 1000 0000 0000 0000 ,cache #2: index == 000, offset == 0000
A[1][0]; //0000 0000 0000 0000 1000 0000 0000 0100 ,cache #2: index == 000, offset == 0100
A[2][0]; //0000 0000 0000 0000 1000 0000 0000 1000 ,cache #2: index == 000, offset == 1000
A[3][0]; //0000 0000 0000 0000 1000 0000 0000 1100 ,cache #2: index == 000, offset == 1100
A[4][0]; //0000 0000 0000 0000 1000 0000 0001 0000 ,cache #2: index == 001, offset == 0000
A[5][0]; //0000 0000 0000 0000 1000 0000 0001 0100 ,cache #2: index == 001, offset == 0100
A[6][0]; //0000 0000 0000 0000 1000 0000 0001 1000 ,cache #2: index == 001, offset == 1000
A[7][0]; //0000 0000 0000 0000 1000 0000 0001 1100 ,cache #2: index == 001, offset == 1100

B[0][0]; //0000 0000 0000 0000 1000 0000 0010 0000 ,cache #2: index == 010, offset == 0000
B[1][0]; //0000 0000 0000 0000 1000 0000 0010 0100 ,cache #2: index == 010, offset == 0100
B[2][0]; //0000 0000 0000 0000 1000 0000 0010 1000 ,cache #2: index == 010, offset == 1000
B[3][0]; //0000 0000 0000 0000 1000 0000 0010 1100 ,cache #2: index == 010, offset == 1100
B[4][0]; //0000 0000 0000 0000 1000 0000 0011 0000 ,cache #2: index == 011, offset == 0000
B[5][0]; //0000 0000 0000 0000 1000 0000 0011 0100 ,cache #2: index == 011, offset == 0100
B[6][0]; //0000 0000 0000 0000 1000 0000 0011 1000 ,cache #2: index == 011, offset == 1000
B[7][0]; //0000 0000 0000 0000 1000 0000 0011 1100 ,cache #2: index == 011, offset == 1100
```

Ξαναρχίζουμε την διαδικασία, με την καινούρια αρχικά άδεια cache:

index	block1	block2
000		
001		
010		
011		
100		
101		
110		
111		

Read: A[1][0], INDEX = 000, MISS
ALLOCATE

index	block1	block2
000	A[0][0] A[1][0] A[2][0] A[3][0]	
001		
010		
011		
100		
101		
110		
111		

Read: A[0][0], INDEX = 000, HIT
Read: B[0][0], INDEX = 010, MISS
ALLOCATE

index	block1	block2
000	A[0][0] A[1][0] A[2][0] A[3][0]	
001		
010	B[0][0] B[1][0] B[2][0] B[3][0]	
011		
100		
101		
110		
111		

Read: A[1][0], INDEX = 000, HIT
Read: A[1][0], INDEX = 000, HIT
Read: B[0][0], INDEX = 010, HIT
Read: A[2][0], INDEX = 000, HIT
Read: A[3][0], INDEX = 000, HIT
Read: A[2][0], INDEX = 000, HIT
Read: B[2][0], INDEX = 010, HIT
Read: A[3][0], INDEX = 000, HIT
Read: A[3][0], INDEX = 000, HIT
Read: B[2][0], INDEX = 010, HIT
Read: A[4][0], INDEX = 001, MISS
ALLOCATE

index	block1	block2
000	A[0][0] A[1][0] A[2][0] A[3][0]	
001	A[4][0] A[5][0] A[6][0] A[7][0]	
010	B[0][0] B[1][0] B[2][0] B[3][0]	
011		
100		
101		
110		
111		

Read: A[5][0], INDEX = 001, HIT
Read: A[4][0], INDEX = 001, HIT
Read: B[4][0], INDEX = 011, MISS
ALLOCATE

index	block1	block2
000	A[0][0] A[1][0] A[2][0] A[3][0]	
001	A[4][0] A[5][0] A[6][0] A[7][0]	
010	B[0][0] B[1][0] B[2][0] B[3][0]	
011	B[4][0] B[5][0] B[6][0] B[7][0]	
100		
101		
110		
111		

Για μια ακόμα φορά, οι πίνακες είναι ολάκεροι μέσα στην cache και δεν είναι εφικτό από την συνέχεια του προγράμματος να αντικατασταθούν, επομένως όλα τα υπόλοιπα reads θα είναι hits. Έχουμε τελικά για το δεύτερο ζητούμενο 16 hits και 4 miss.

Παρατηρούμε πως το hit ratio έπεσε. Θα προτιμήσουμε την πρώτη κρυφή μνήμη, καθώς:

1. έχει μικρότερο miss ratio
2. ασχολείται λιγότερες φορές με την κύρια μνήμη (κάνει μονάχα 2 φορές allocate), άρα είναι και πιο γρήγορη
3. αρκεί το μέγεθός της για τις ανάγκες μας, δεν μας χριάζονται οι περαιτέρω γραμμές/σύνολα της δεύτερης κρυφής μνήμης.