Πανεπιστήμιο Θεσσαλίας
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Lab #9

Ονοματεπώνυμο: Ζαχαριάδης Χαράλαμπος, Ραφτόπουλος Εμμανουήλ

Μάθημα: Οργάνωση και Σχεδίαση Η/Υ – ΑΕΜ: *03734,03735*
Ημερομηνία: *10/2/2024*

Θέμα **1**ο

Απάντηση. a) Αρχικά πρέπει να υπολογίσουμε το μέγεθος bits του offset. Επειδή το σύστημα εικονικής μνήμης έχει σελίδες των 4 KB (= 2^{12}) θα χρειαστούμε 12 bits από το virtual address που θα μας δείχνουν το page offset.

Έπειτα θα μετατρέψουμε τις δεκαδικές εικονικές διευθύνσεις σε δεκαεξαδικές για να συγκρίνουμε εύκολα τα bits τους:

Decimal	9452	30964	19136	46502	38110	16653	48480
HEX	0x24EC	0x77E6	0x4AC0	0xB5A6	0x94DE	0x410D	0xBD60

Τα υπόλοιπα bits πέρα του offset θα μπουν στο tag του TLB. Στην προσομοίωση που θα κάνουμε πάνω στην αποθήκευση εικονικών διευθύνσεων υπάρχουν τρεις πιθανές περιπτώσεις:

1. **TLB Hit:** Εδώ η διεύθυνση είναι ήδη στον TLB άρα δεν χρειάζεται να γίνει κάποια αλλαγή και η θέση του TLB που υπάρχει το Hit είναι η πιο πρόσφατα χρησιμοποιημένη.
2. **TLB Miss & Physical Page Table Hit:** Σε αυτήν την περίπτωση η διεύθυνση δεν είναι στον TLB άρα καταφεύγουμε στο Page Table όπου εκεί πηγαίνουμε στη θέση που μας υποδεικνύει το Tag και έχουμε Hit (Valid Bit = 1). Άρα μεταφέρουμε το PP number που δείχνει στη πρώτη κενή θέση του TLB (Valid Bit = 0). Σε περίπτωση που δεν υπάρχει κενή θέση (δηλαδή έχουμε σε όλον τον TLB Valid Bit = 1) τότε ακολουθούμε τον LRU αλγόριθμο κατά τον οποίο βάζουμε τη νέα διεύθυνση στη θέση που χρησιμοποιήθηκε πιο παλιά.
3. **TLB Miss & Physical Page Table Miss:** Παρόμοια περίπτωση με την πάνω αλλά με τη διαφορά ότι ο PP Number δεν είναι στο Page Table (άρα Valid Bit = 0 άρα Page Table Miss) αλλά στον σκληρό δίσκο.

Με βάση αυτούς τους κανόνες ακολουθεί παρακάτω η προσομοίωση:

Αρχική Κατάσταση:

Valid Bit	Tag	PP Number
1	11	12
1	7	4
1	3	6
0	4	9

Valid Bit	PP Number
1	5
0	Δίσκος
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(9452)_{10}$ ή καλύτερα $(0x24EC)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 2 αλλά δεν υπάρχει. Έπειτα πάμε στη θέση 2 (3η γραμμή) του πίνακα σελίδων όπου και εκεί δέν έχουμε valid bit άρα φέρνουμε απ' το δίσκο τη σελίδα που ψάχνουμε και στην τρίτη γραμμή του πίνακα σελίδων, καθώς και στην τελευταία γραμμή του TLB η οποία έχει valid bit = 0:

Valid Bit	Tag	PP Number
1	11	12
1	7	4
1	3	6
1	2	PA1

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(30964)_{10}$ ή καλύτερα $(0x77E6)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 7, που υπάρχει (TLB Hit). Άρα το μόνο που αλλάζει είναι ότι τώρα η πιο πρόσφατα χρησιμοποιημένη θέση του TLB είναι η θέση 1 που είχαμε το Hit:

Valid Bit	Tag	PP Number
1	11	12
1	7	4
1	3	6
1	2	PA1

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(19136)_{10}$ ή καλύτερα $(0x4AC0)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 4, που δεν υπάρχει (TLB Miss). Έπειτα πάμε στη θέση 4 (5η γραμμή) του πίνακα σελίδων όπου έχει valid bit και δείχνει στη φυσική σελίδα 9. Άρα θα το πάρουμε από εκεί και θα το γράψουμε και στο TLB στην πιο "παλιά" καταχώρηση, η οποία δίνεται από την εκφώνηση ότι είναι η 1η γραμμή:

Valid Bit	Tag	PP Number
1	4	9
1	7	4
1	3	6
1	2	PA1

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(46502)_{10}$ ή καλύτερα $(0xB5A6)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 11, που δεν υπάρχει (TLB Miss). Έπειτα πάμε στη θέση 11 (12η γραμμή) του πίνακα σελίδων όπου έχει valid bit και δείχνει στη φυσική σελίδα 12. Άρα θα το πάρουμε από εκεί και θα το γράψουμε και στο TLB στην πιο "παλιά" καταχώρηση, η οποία μετά από τις προηγούμενες διαπεράσεις είναι η 3η γραμμή:

Valid Bit	Tag	PP Number
1	4	9
1	7	4
1	11	12
1	2	PA1

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(38110)_{10}$ ή καλύτερα $(0x94DE)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 9 αλλά δεν υπάρχει. Έπειτα πάμε στη θέση 9 (10η γραμμή) του πίνακα σελίδων όπου και εκεί δέν έχουμε valid bit άρα φέρνουμε απ' το δίσκο τη σελίδα που ψάχνουμε και στην δέκατη γραμμή του πίνακα σελίδων, καθώς και στην τελευταία γραμμή του TLB η οποία είναι η πιο "παλιά" που διαπεράσαμε:

Valid Bit	Tag	PP Number
1	4	9
1	7	4
1	11	12
1	9	PA2

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
1	PA2
1	3
1	12

Για τη διεύθηνση $(16653)_{10}$ ή καλύτερα $(0x410D)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 4, που υπάρχει (TLB Hit). Άρα το μόνο που αλλάζει είναι ότι τώρα η πιο πρόσφατα χρησιμοποιημένη θέση του TLB είναι η θέση 0 που είχαμε το Hit:

Valid Bit	Tag	PP Number
1	4	9
1	7	4
1	11	12
1	9	PA2

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
1	PA2
1	3
1	12

Για τη διεύθηνση $(48480)_{10}$ ή καλύτερα $(0xBD60)_{16}$ Ψάχνουμε πρώτα στον TLB για tag = 11, που υπάρχει (TLB Hit). Άρα το μόνο που αλλάζει είναι ότι τώρα η πιο πρόσφατα χρησιμοποιημένη θέση του TLB είναι η θέση 2 που είχαμε το Hit, Άρα η τελική κατάσταση είναι η εξής:

Valid Bit	Tag	PP Number
1	4	9
1	7	4
1	11	12
1	9	PA2

Valid Bit	PP Number
1	5
0	Δίσκος
1	PA1
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
1	PA2
1	3
1	12

b) Με την αλλαγή του μεγέθους σελιδών απο 4KB σε 16KB(= 2^{14}) θα χρειαστούμε πλέον 14 bits από το virtual address που θα μας δείχνουν το page offset. Το μέγεθος του Page Table μένει ίδιο αφού η ποσότητα των physical pages είναι ίδια.

Μετατρέπουμε τις δεκαδικές εικονικές διευθύνσεις σε δυαδικές για να συμπεράνουμε τα page offsets:

Decimal	Binary
9452	00100100111101100
30964	01111000111110100
19136	01001010110000000
46502	10111010110100110
38110	10010100110111110
16653	01000000100001101
48480	10111101011000000

Ύστερα ακολουθώντας τους κανόνες που αναφέρθηκαν και στο υποερώτημα (a) πραγματοποιούμε την παρακάτω προσομοίωση:

Αρχική Κατάσταση:

Valid Bit	Tag	PP Number
1	11	12
1	7	4
1	3	6
0	4	9

Valid Bit	PP Number
1	5
0	Δίσκος
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(9452)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(00)_2 = (0)_{10}$ αλλά δεν υπάρχει (TLB Miss). Έπειτα πάμε στη θέση 0 (1η γραμμή) του πίνακα σελίδων όπου και εκεί έχουμε valid bit (Page Table Hit), άρα φέρνουμε απ' το Page Table τη σελίδα που ψάχνουμε στην τελευταία γραμμή του TLB η οποία έχει valid bit = 0:

Valid Bit	Tag	PP Number
1	11	12
1	7	4
1	3	6
1	0	5

Valid Bit	PP Number
1	5
0	Δίσκος
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(30964)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(01)_2 = (1)_{10}$ αλλά δεν υπάρχει, άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 1 (2η γραμμή) του πίνακα σελίδων όπου και εκεί δεν έχουμε valid bit (Page Table Miss και Page Fault) άρα φέρνουμε απ' τον δίσκο τη σελίδα στο Page Table (το valid bit γίνεται 1) και στη θέση 0 του TLB που είναι η λιγότερα πρόσφατα χρησιμοποιημένη, αφού όλες οι θέσεις του TLB είναι έγκυρες:

Valid Bit	Tag	PP Number
1	1	PA1
1	7	4
1	3	6
1	0	5

Valid Bit	PP Number
1	5
1	PA1
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(19136)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(01)_2 = (1)_{10}$ το οποίο υπάρχει στη θέση 0, άρα έχουμε TLB Hit και η πιο πρόσφατα χρησιμοποιημένη θέση είναι η θέση 0:

Valid Bit	Tag	PP Number
1	1	PA1
1	7	4
1	3	6
1	0	5

Valid Bit	PP Number
1	5
1	PA1
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(46502)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(10)_2 = (2)_{10}$ το οποίο δεν υπάρχει στον TLB , άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 2 (3η γραμμή) του πίνακα σελίδων όπου και εκεί δεν έχουμε valid bit (Page Table Miss και Page Fault) άρα φέρνουμε απ' τον δίσκο τη σελίδα στο Page Table (το valid bit γίνεται 1) και στη θέση 1 του TLB που είναι η λιγότερα πρόσφατα χρησιμοποιημένη, αφού όλες οι θέσεις του TLB είναι έγκυρες:

Valid Bit	Tag	PP Number
1	1	PA1
1	2	PA2
1	3	6
1	0	5

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(38110)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(10)_2 = (2)_{10}$ το οποίο υπάρχει στη θέση 1, άρα έχουμε TLB Hit και η πιο πρόσφατα χρησιμοποιημένη θέση είναι η θέση 1:

Valid Bit	Tag	PP Number
1	1	PA1
1	2	PA2
1	3	6
1	0	5

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(16653)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(01)_2 = (1)_{10}$ το οποίο υπάρχει στη θέση 0, άρα έχουμε TLB Hit και η πιο πρόσφατα χρησιμοποιημένη θέση είναι η θέση 0:

Valid Bit	Tag	PP Number
1	1	PA1
1	2	PA2
1	3	6
1	0	5

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(48480)_{10}$ ψάχνουμε πρώτα στον TLB για tag = $(10)_2 = (2)_{10}$ το οποίο υπάρχει στη θέση 1, άρα έχουμε TLB Hit και η πιο πρόσφατα χρησιμοποιημένη θέση είναι η θέση 1. Άρα, η τελική κατάσταση είναι η εξής:

Valid Bit	Tag	PP Number
1	1	PA1
1	2	PA2
1	3	6
1	0	5

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Σχολιάζοντας τα πλεονεκτήματα και μειονεκτήματα της αύξησης μεγέθους σελίδας καταλήξαμε στα εξής:

- Πλεονεκτήματα:
  1. Πιο αποτελεσματική χρήση του TLB: Τα TLB Hits αυξήθηκαν σε σχέση με το πρώτο πείραμα καθώς υπάρχουν περισσότερα κοινά tags.
  2. Μειωμένος αριθμός ενημερώσεων στον πίνακα σελίδων: Όσο μεγαλύτερη είναι η σελίδα, τόσο λιγότερες σελίδες θα χρειαστεί να διαχειριστεί ο πίνακας σελίδων, καθώς μια μεγαλύτερη σελίδα καλύπτει μεγαλύτερο εύρος μνήμης.
  3. Λιγότερες αναφορές στον πίνακα σελίδων & Λιγότερα Page Faults: Αυτό συμβαίνει εξαιτίας των παραπάνω TLB Hits.
  4. Μείωση μεγέθους του Page Table: Επειδή ο αριθμός των tag είναι λιγότερος μας επιτρέπει να χρησιμοποιούμε Page Table με λιγότερες θέσεις.
- Μειονεκτήματα:
  1. Απώλεια χώρου στη μνήμη: Εάν οι σελίδες είναι μεγάλες, αλλά οι λειτουργίες του λειτουργικού απαιτούν πολύ λιγότερο χώρο στη μνήμη, αυξάνεται η ποσότητα του άχρηστου καταλαμβανώμενου χώρου στη μνήμη.
  2. Μεγάλο Χρονικό Κόστος σε περιπτώσεις Miss: Σε περίπτωση που έχουμε Miss είτε στον TLB είτε στο Page Table είτε και στα δύο ειδικά, όπου έχουμε page fault, θα χρειαστεί πολύς χρόνος να μεταφερθούν δεδομένα ειδικά από τον Δίσκο αλλά και από το Page Table.

c) Για να μετατρέψουμε τον fully associative TLB σε direct mapped θα χρειαστούμε μια επιπλέον πληροφορία το block offset το οποίο θα βρίσκεται στα 2 επόμενα bits από τα 12 που χρειάζονται για το page offset. (2 γιατί έχουμε 4 εισόδους, δηλαδή 2^2)

Διάταξη εικονικών διευθύνσεων

Decimal	Tag	Block Offset	Page Offset
	Rest of the bits	2 bits	12 bits
9452	00	10	010011101100
30964	01	11	100011110100
19136	01	00	101011000000
46502	10	11	010110100110
38110	10	01	010011011110
16653	01	00	000100001101
48480	10	11	110101100000

Αρχική Κατάσταση:

Valid Bit	Tag	Block Offset	PP Number
0		00	
0		01	
0		10	
0		11	

Valid Bit	PP Number
1	5
0	Δίσκος
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(9452)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(10)_2$ αλλά έχουμε valid bit = 0, άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 0 (1η γραμμή) του πίνακα σελίδων όπου έχουμε valid bit (Page Table Hit) άρα φέρνουμε απ' τον πίνακα σελίδων στο TLB τη σελίδα 5 στο block offset 10 του TLB, κάνοντας το valid bit = 1 και βάζοντας στο tag $(00)_2$ και στο PP Number 5:

Valid Bit	Tag	Block Offset	PP Number
0		00	
0		01	
1	00	10	5
0		11	

Valid Bit	PP Number
1	5
0	Δίσκος
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(30964)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(11)_2$ αλλά έχουμε valid bit = 0, άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 1 (2η γραμμή) του πίνακα σελίδων όπου δεν έχουμε valid bit (Page Table Miss και Page Fault) άρα φέρνουμε απ' τον Δίσκο στο Page Table και στον TLB τη σελίδα PA1 στο block offset 11 του TLB, κάνοντας το valid bit = 1 και βάζοντας στο tag $(01)_2$ και στο PP Number PA1:

Valid Bit	Tag	Block Offset	PP Number
0		00	
0		01	
1	00	10	5
1	01	11	PA1

Valid Bit	PP Number
1	5
1	PA1
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(19136)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(00)_2$ αλλά έχουμε valid bit = 0, άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 1 (2η γραμμή) του πίνακα σελίδων όπου έχουμε valid bit (Page Table Hit) άρα φέρνουμε απ' το Page Table στον TLB τη σελίδα PA1 στο block offset 00 του TLB, κάνοντας το valid bit = 1 και βάζοντας στο tag $(01)_2$ και στο PP Number PA1:

Valid Bit	Tag	Block Offset	PP Number
1	01	00	PA1
0		01	
1	00	10	5
1	01	11	PA1

Valid Bit	PP Number
1	5
1	PA1
0	Δίσκος
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(46502)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(11)_2$ όπου έχουμε διαφορετικό tag, άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 2 (3η γραμμή) του πίνακα σελίδων όπου δεν έχουμε valid bit (Page Table Miss και Page Fault) άρα φέρνουμε απ' τον Δίσκο στο Page Table και στον TLB τη σελίδα PA2 στο block offset 11 του TLB, αφαιρώντας το ήδη υπάρχον στοιχείο στη θέση αυτή και βάζοντας στο tag $(10)_2$ και στο PP Number PA2:

Valid Bit	Tag	Block Offset	PP Number
1	01	00	PA1
0		01	
1	00	10	5
1	10	11	PA2

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(38110)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(01)_2$ όπου δεν έχουμε valid bit, άρα έχουμε TLB Miss. Έπειτα πάμε στη θέση 2 (3η γραμμή) του πίνακα σελίδων όπου έχουμε valid bit (Page Table Hit) άρα φέρνουμε απ' το Page Table στον TLB τη σελίδα PA2 στο block offset 01 του TLB, κάνοντας το valid bit 1 και βάζοντας στο tag $(10)_2$ και στο PP Number PA2:

Valid Bit	Tag	Block Offset	PP Number
1	01	00	PA1
1	10	01	PA2
1	00	10	5
1	10	11	PA2

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(16653)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(00)_2$ όπου έχουμε και valid bit και ίδιο tag, άρα έχουμε TLB Hit:

Valid Bit	Tag	Block Offset	PP Number
1	01	00	PA1
1	10	01	PA2
1	00	10	5
1	10	11	PA2

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

Για τη διεύθηνση $(48480)_{10}$ ψάχνουμε πρώτα στον TLB στο block offset $(11)_2$ όπου έχουμε valid bit και ίδιο tag, άρα έχουμε TLB Hit:

Valid Bit	Tag	Block Offset	PP Number
1	01	00	PA1
1	10	01	PA2
1	00	10	5
1	10	11	PA2

Valid Bit	PP Number
1	5
1	PA1
1	PA2
1	6
1	9
1	11
0	Δίσκος
1	4
0	Δίσκος
0	Δίσκος
1	3
1	12

- Πώς θα γινόταν ο χειρισμός της εικονικής μνήμης χωρίς την ύπαρξη του TLB;

  Χωρίς τον TLB η διαχείριση της εικονικής μνήμης θα είναι πολύ πιο αργή. Κάθε αίτηση πρόσβασης σε εικονική διεύθυνση θα απαιτεί την ανάτρεξη στον πίνακα σελίδων, ο οποίος βρίσκεται στην κύρια μνήμη. Αυτό είναι μια πολύ χρονοβόρα διαδικασία που απαιτεί δεκάδες έως και εκατοντάδες κύκλους, για να φέρουμε τα δεδομένα, σε σύγκριση με τα TLB Hits που χρειάζονται ελάχιστοι κύκλοι μηχανής (ακόμα λιγότερο και από 1 κύκλο).

d) Αρχικά θα απαριθμήσουμε τα νέα δεδομένα:

- Μέγεθος εικονικής διεύθυνσης 64 bits
- Μέγεθος σελίδας 16 KB = 2^{14} KB
- Μέγεθος κάθε εισόδου (entry) στον πίνακα σελίδων 8 bytes

Το page offset για τις εικονικές διευθύνσεις θα είναι 14 bits (όπως στο (b) υποερώτημα) άρα το virtual page number θα είναι $64 - 14 = 50$ bits. Οπότε, μπορούμε να έχουμε συνολικά 2^{50} εικονικές σελίδες. Και για να έχουμε κάθε μία από αυτές στον πίνακα σελίδων χρειαζόμαστε 8 $bytes * 2^{50} = 2^{53}$ bytes. Άρα αν για κάθε μία από τις 5 εφαρμογές φτιαχτεί ένα τέτοιο Page Table θα χρειαστούμε $5 * 2^{53}$ **bytes**.

e) Όπως και στο υποερώτημα (d) το virtual page number έχει 50 bits και το page offset 14 bits. Οπότε θα χωρίσουμε τα bits του VPN στα 2 επίπεδα του πίνακα σελίδων. Εφόσον γνωρίζουμε τα entries του πρώτου επιπέδου θα βρούμε και τα bits που χρειάζεται αυτό το επίπεδο. Οπότε, 256 entries = 2^8 entries, άρα θέλουμε τα 8 πρώτα bits του VPN για το πρώτο μόνο επίπεδο και τα υπόλοιπα 42 για το δεύτερο. Δηλαδή, το μέγεθος του πρώτου επιπέδου θα είναι $8 * 2^8 = 2^{11}$ bytes. Για να υπολογίσουμε το μέγεθος του δεύτερου επιπέδου θα πρέπει πρώτα να συλλογιστούμε, ότι για κάθε entry του πρώτου επιπέδου χρειαζόμαστε ένα πίνακα σελίδων δεύτερου και κάθε μοναδικό 2nd stage page table έχει μέγεθος $8 * 2^{42}$ bytes, και άρα συνολικά θα είναι $256 * 8 * 2^{42} = 2^{53}$ bytes. Για μία μόνο εφαρμογή το μέγεθος του πίνακα σελίδων 2 επιπέδων θα είναι $2^{11} + 2^{53}$ και για 5 εφαρμογές $5 * (2^{11} + 2^{53})$ bytes.

f) Όπως και στο υποερώτημα (e) το virtual page number έχει 50 bits και το page offset 14 bits. Η εκφώνηση μας λέει ότι κάθε cache block είναι 2 words, δηλαδή αν το μέγεθος του word είναι 4 bytes (όπως στα περισσότερα 32-bits συστήματα) θα έχουμε 8 byte ανά cache block.Συνολικά δηλαδή θα έχουμε:

$$\frac{16KB}{8bytes} = 2048 = 2^{11} \text{ cache blocks.}$$

Άρα από τα 14 διαθέσιμα bits του page offset για τη λειτουργία της VIPT cache στη συγκεκριμένη περίπτωση χρειαζόμαστε μόνο τα 11. Αυτό σημαίνει, ότι έχουμε 3 διαθέσιμα bits για να αυξήσουμε το μέγεθος της cache πάνω από 16KB επί 3 φορές το πολύ. Διαφορετικά αλλάζοντας την αρχιτεκτονική της είτε αυξάνοντας το associativity εώς $2^3 = 8$ way, είτε αυξάνοντας το μέγεθος των σελίδων για να έχουμε περισσότερα διαθέσιμα bits.

Θέμα **2**ο

Απάντηση**.** Το flag -O0 βοηθάει στο optimization του χρόνου μεταγλώττισης και στην ικανότητα debugging του προγράμματος. Οι περισσότερες μέθοδοι optimization είναι απενεργοποιημένες και αποτελεί default flag για τη gcc μεταγλώττιση.

Αποτελέσματα Εκτέλεσης των LOOP1 - LOOP2 (με -O0):

	LOOP1	LOOP2
Duration	0.96s	0.48s
Cycles	3.76B	1.88B
Stalled Cycles Per Instruction	2.12	0.99
Stalled Cycles Frontend	11k	62k
Stalled Cycles Backend	3.42B	1.6B
L1-Dcach Loads	1.3B	1.3B
L1-Dcache Misses	11k	7k
L1-Dcache Prefetches	71.2k	162.7k
L1-Icache loads	8k	5.9M
L1-Icache misses	4k	3k
Cache References	17k	6k
Cache Misses	9k	1k
Page Faults	59	55
DTLB Loads	1.6k	867
DTLB Misses	402	110
ITLB Loads	3 / 0	0 / 0
ITLB Misses	214 / 295	0 / 7
All TLBS Flushed	0	0

Σχολιασμός του κώδικα:

1. Allocations Καταχωρητών: Οι μεταγλωττιστές τείνουν να αποθηκεύουν συχνά χρησιμοποιούμενες μεταβλητές σε καταχωρητές για ταχύτερη πρόσβαση. Έτσι:
   - Στην περίπτωση της LOOP1 (όπου γίνεται το {x++; x++;}) και οι δύο προσθέσεις θα χρησιμοποιήσουν κατά πάσα πιθανότητα τον ίδιο καταχωρητή που σημαίνει πως μια τιμή πρέπει προσωρινά να αποθηκευτεί στη μνήμη (spilling) πριν γίνει η δεύτερη πρόσθεση της επανάληψης. Αυτή η διαδικασία (spilling overhead) οδηγεί στην επιβράδυνση της εκτέλεσης του προγράμματος.
   - Στη περίπτωση της LOOP2 (όπου γίνεται το {x++; y++;}) το πιο πιθανό σενάριο είναι τα x και y να χρησιμοποιούν ξεχωριστούς καταχωρητές που επιτρέπουν τη ταυτόχρονη εκτέλεση των προσθέσεων της επανάληψης χωρίς spills. Έτσι μπορεί να εξοικονομηθεί χρόνος κατά την εκτέλεση του προγράμματος.
2. Instruction Pipelining:
   - Στη LOOP1 όπως παρατηρήσαμε και από το vtune το pipeline καθυστερεί να εκτελέσει τις δύο προσθέσεις σε κάθε επανάληψη καθώς οι δύο προσθέσεις χρησιμοποιούν τον ίδιο καταχωρητή με αποτέλεσμα την αλληλοεξάρτηση των εντολών το οποίο καθυστερεί την λειτουργία των καταχωρητών λόγω των διπλάσιων stalls που δημιουργούνται, όπως συμπεράναμε από το perf.
   - Στη LOOP2 επειδή έχουμε διαφορετικές μεταβλητές είναι αρκετά λιγότερο πιθανό να έχουμε conflicts στη λειτουργία του pipeline, άρα έχουμε πιο γρήγορη εκτέλεση.
3. Memory Access:
   - Εφόσον οι x και y είναι δύο διαφορετικές μεταβλητές θα είναι αποθηκευμένες σε διαφορετικά σημεία της μνήμης ο κώδικας του LOOP2 διαβάζει από τη μνήμη 2 φορές για κάθε επανάληψη. Αυτό σημαίνει, ότι ενδέχεται να έχουμε περισσότερα misses στην cache και μείωση της απόδοσης του προγράμματος.
   - Παρόλα αυτά αυτό δεν συμβαίνει κατά κανόνα όπως είδαμε με τη χρήση του perf και δεν επισκιάζει την απόδοση που προσφέρουν τα προαναφερθέντα.

Ανακεφαλαιώνοντας:

Η ακριβής διαφορά στην απόδοση του προγράμματος εξαρτάται από διάφορους παράγοντες, όπως:

1. Η "στρατηγική" του μεταγλωτιστή, κυρίως κατά την οργάνωση των καταχωρητών που χρησιμοποιεί.
2. Η αρχιτεκτονική του επεξεργαστή και της cache (μέγεθος και είδος).
3. Η τιμή της μεταβλητής steps δηλαδή ο αριθμός των επαναλήψεων (μεγαλύτερες τιμές ευνοούν το LOOP2).

Εάν βάλουμε -O3 για fully optimized code, η διάρκεια εκτέλεσης και στις δύο περιπτώσεις είναι πολύ μικρή (Seconds = 0.000001).

Το flag -O3 αυτό που κάνει ώστε να έχουμε fully optimized code είναι να καθοδηγεί τον μεταγλωττιστή να είναι εξαιρετικά επιθετικός όσον αφορά τις τεχνικές βελτιστοποίησης που χρησιμοποιεί, και να χρησιμοποιεί όση μνήμη χρειάζεται για τη μέγιστη δυνατή βελτιστοποίηση.

Πιο συγκεκριμένα στην εφαρμογή μας η τεράστια διαφορά απόδοσης οφείλεται στο loop unrolling που κάνει το -O3 flag. Μεταφράζοντας τον κώδικα σε assembly αντί για ELF file παρατηρούμε, ότι ο compiler δίωχνει εντελώς τη for loop και βάζει απευθείας το αποτέλεσμα στον καταχωρητή επιστροφής eax γιαυτό και το πρόγραμμα τρέχει πολύ πιο γρήγορα έτσι. Βέβαια η μεταγλώττιση του προγράμματος με το -O3 flag είναι σχεδόν 5 φορές πιο αργή από ότι με το -O0, κάτι το οποίο όμως δεν μας ενδιαφέρει ιδιαίτερα αφού γίνεται μία μόνο φορά. Συγκεκριμένα:

	-O0	-O3
Duration	0,052 sec	0,245 sec

Με **-O0 flag**

```
.L3:
# ./lab9_program.c:15:   for (int i=0; i<steps; i++) { x++; x++;}
    .loc 1 15 33 discriminator 3
    addl    $1, -32(%rbp)   #, x
# ./lab9_program.c:15:   for (int i=0; i<steps; i++) { x++; x++;}
    .loc 1 15 38 discriminator 3
    addl    $1, -32(%rbp)   #, x
# ./lab9_program.c:15:   for (int i=0; i<steps; i++) { x++; x++;}
    .loc 1 15 26 discriminator 3
    addl    $1, -28(%rbp)   #, i
.L2:
# ./lab9_program.c:15:   for (int i=0; i<steps; i++) { x++; x++;}
    .loc 1 15 17 discriminator 1
    movl    -28(%rbp), %eax # i, tmp90
    cmpl    -24(%rbp), %eax # steps, tmp90
    jl      .L3
```

Με **-O3 flag**

```
movl        $536870912, %eax
```