Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εαρινό Εξάμηνο 2023-2024

ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Λύσεις Θεμάτων

Ιωάννης (Χουάν) Τσαντήλας
03120883

Contents

Επίσημες Απαντήσεις σε Θεωρητικές Ερωτήσεις 2
Κανονική 12 2
Θέμα 1 2
Θέμα 2 2
Κανονική 11 4
Θέμα 1 4
Θέμα 2 4
Θέμα 4 5
Επαναληπτική 10 6
Θέμα 1 6
Κανονική 09 7
Κανονική 08 9
Θέμα 1 9
Κανονική 23 11
Θέμα 1 11
Ερώτημα Α 11
Ερώτημα Β 11
Ερώτημα Γ 12
Θέμα 2 15
Κανονική 19 18
Θέμα 2 18
Θέμα 4 21
Κανονική 21 23
Θέμα 3 23

Επίσημες Απαντήσεις σε Θεωρητικές Ερωτήσεις

Παρακάτω, στα Θέματα των 12 έως και 08, είναι ερωτήσεις που τείνουν να ξαναβάζουν.

Κανονική 12

Θέμα 1

Ερώτημα Α: Εξηγήστε για ποιο λόγο οι αρχιτεκτονικές που χρησιμοποιούν τον αλγόριθμο Tomasulo χρειάζονται κάποιο μηχανισμό πρόβλεψης διακλαδώσεων.

Ο Tomasulo υλοποιεί out-of-order εκτέλεση εντολών προκειμένου να βελτιώσει την απόδοση του συστήματος. Αυτό επιτυγχάνεται δρομολογώντας εντολές που με βάση το program order ακολουθούν άλλες, οι οποίες όμως είναι blocked περιμένοντας το αποτέλεσμα άλλων προηγούμενων εντολών. Απαιτείται λοιπόν δρομολόγηση πολλών εντολών που βρίσκονται πιο μπροστά στο instruction stream. Έτσι, όταν συναντούν ένα branch απαιτείται ένας μηχανισμός πρόβλεψης ώστε να επιλεγεί το path από το οποίο θα έρθουν οι επόμενες εντολές. Διαφορετικά, αν δεν υπήρχε ο μηχανισμός πρόβλεψης, θα έπρεπε να περιμένουν το αποτέλεσμα του branch περιορίζοντας έτσι τα οφέλη της εκτέλεσης.

Ερώτημα Β: Σας αναθέτουν να σχεδιάσετε έναν επεξεργαστή που θα υλοποιεί ένα ISA το οποίο ορίζει K καταχωρητές (architecture registers). Για να βελτιώσετε την απόδοση, αποφασίζετε να σχεδιάσετε ένα out-of-order επεξεργαστή που θα υλοποιεί τον αλγόριθμο Tomasulo χρησιμοποιώντας ένα Reorder Buffer N θέσεων καθώς και M Reservation Stations. Ποιος είναι ο μέγιστος αριθμός φυσικών καταχωρητών (Physical Registers) που θα ενσωματώσετε στο σχέδιο σας;

Καταρχάς, χρειαζόμαστε τουλάχιστον όσους ορίζει το ISA, δηλαδή K. Εφόσον το σύστημα χρησιμοποιεί ένα ROB N θέσεων μπορούμε να έχουμε το πολύ N εντολές in-fight (να εκτελούνται ταυτόχρονα) οι οποίες γράφουν σε κάποιο καταχωρητή. Για να μπορούμε λοιπόν να κάνουμε το register renaming απαιτούνται N έξτρα φυσικοί καταχωρητές.

$$physical\ registers \leq N + K$$

Ερώτημα Γ: Υποθέστε μια αρχιτεκτονική που υλοποιεί τον αλγόριθμο Tomasulo με χρήση Reorder Buffer. Σε ποιο στάδιο πραγματοποιούνται τα loads (δηλ. έρχεται στον επεξεργαστή από τη μνήμη η τιμή που θα χρησιμοποιηθεί από κάποια επόμενη εντολή); Σε ποιο στάδιο πραγματοποιούνται τα store (δηλ. στέλνεται στην μνήμη η τιμή που έχει γράψει ο επεξεργαστής); Αν τα στάδια διαφορετικά εξηγήστε το γιατί.

Τα loads πραγματοποιούνται στο EX (όταν ολοκληρωθεί το EX η τιμή έχει έρθει στον επεξεργαστή). Αντίθετα τα stores πραγματοποιούνται στο CMT. Η διαφορά οφείλεται στις επιπτώσεις που έχουν οι εντολές αυτές κατά την υποθετική εκτέλεση (speculative execution). Τα loads δεν επηρεάζουν το architectural state και για αυτό το λόγο επιτρέπουμε να πραγματοποιούνται στο EX. Αντίθετα, τα stores μεταβάλλουν την κατάσταση του συστήματος και για αυτό το λόγο τα πραγματοποιούμε όταν ξέρουμε ότι η εντολή επιτρέπεται να εκτελεστεί με βάση τη σειρά του προγράμματος (χωρίς δηλαδή να αντιμετωπίζουμε κινδύνους λόγω exceptions/interrupts ή mispredicted branches.

Θέμα 2

Ερώτημα Α: Ποια η διαφορά μεταξύ snooping και directory πρωτοκόλλων; Ποιος από τους 2 παραπάνω τύπους πρωτοκόλλων κλιμακώνει καλύτερα; Εξηγήστε γιατί.

Στα snooping πρωτόκολλα απαιτείται οι controllers όλων των caches να παρακολουθούν τις λειτουργίες όλων. Αντίθετα, στα directory πρωτόκολλα, επειδή ο κατάλογος γνωρίζει ποιοι είναι οι shares για κάθε block, καθορίζει αυτός ποιες caches θα πρέπει να ενημερωθούν για κάποια λειτουργία μνήμης. Το γεγονός ότι δεν απαιτούνται broadcasts κάνει τα directory πρωτόκολλα να κλιμακώνουν καλύτερα από τα snooping πρωτόκολλα.

Ερώτημα Β: Πώς πλεονεκτεί μια υλοποίηση Test-and-Test-and-Set μιας λειτουργίας κλειδώματος (lock) έναντι μιας υλοποίησης Test-and-Set σε ένα πολυεπεξεργαστικό σύστημα που χρησιμοποιεί write-invalidate cache coherence protocol; Εξηγήστε την απάντησή σας.

Στην Test-and-Set υλοποίηση, κάθε φορά που ένας επεξεργαστής που δεν έχει το lock επιχειρεί να το πάρει, αναγκάζεται να γράψει στη θέση μνήμης που αντιστοιχεί στη διεύθυνση του lock. Αυτό συμβαίνει σε κάθε επανάληψη του Test-and-Set loop, με αποτέλεσμα όταν οι διεκδικητές του lock είναι πολλοί να προκαλούνται αλλεπάλληλα Invalidations της cache line που περιέχει το lock, και συνεπώς να εισάγεται αυξημένη κίνηση στο δίαυλο. Όσο μεγαλύτερος είναι ο αριθμός των διεκδικητών ή όσο περισσότερος ο χρόνος που το Lock είναι κρατημένο, τόσο μεγαλύτερη η συμφόρηση που υπάρχει στο δίαυλο.

Αντίθετα, στην Test-and-Test-and-Set υλοποίηση, σε κάθε επανάληψη του loop ο διεκδικητής του lock διαβάζει την τιμή του lock, και μόνο όταν αυτό φανεί ότι απελευθερώθηκε, επιχειρεί να το πάρει. Οι διεκδικητές, κατά συνέπεια, κάνουν spinning τοπικά στις caches τους, με αποτέλεσμα να μην εισάγεται άσκοπα κίνηση στον δίαυλο.

Κανονική 11

Θέμα 1

Ερώτημα Α: Ποια τεχνική χρησιμοποιεί ο αλγόριθμος Tomasulo για να αντιμετωπίσει τους WAR και WAW hazards; Γιατί δεν χρησιμοποιεί την ίδια τεχνική για να αντιμετωπίσει και τους RAW hazards;

Ο Tomasulo χρησιμοποιεί register renaming για να αντιμετωπίσει τους WAR, WAW hazards. Η τεχνική αυτή αλλάζει τον destination register, και έτσι οι 2 εντολές που προκαλούν το hazard δεν χρειάζεται πλέον να προσπελάσουν τον ίδιο καταχωρητή. Η αλλαγή αυτή όμως δεν επηρεάζει τις RAW εξαρτήσεις, μιας και όπως και να μετονομάσουμε τον destination καταχωρητή, η εντολή που διαβάζει θα εξακολουθήσει να χρειάζεται την τιμή που θα γράψει η προηγούμενη εντολή. Επομένως το register renaming δεν επιλύει τους RAW hazards.

Ερώτημα Β: Ποια η διαφορά μεταξύ Reservation Stations και Reorder Buffer;

Τα RS αποθηκεύουν μια εντολή και τα ορίσματα της μέχρι να μπορέσει η εντολή αυτή να εκτελεστεί. Αντίθετα, ο ROB αποθηκεύει την κατάσταση της εντολής (και μετά την εκτέλεση και το αποτέλεσμα της) μέχρι η εντολή να κάνει commit.

Ερώτημα Γ: Τι είναι ο Branch Target Buffer (BTB); Πώς βελτιώνει η χρήση του την απόδοση ενός επεξεργαστή; Γιατί δε θα δούλευε αποδοτικά για την πρόβλεψη των διευθύνσεων επιστροφής από ρουτίνες/συναρτήσεις;

Ο BTB αποθηκεύει το target address προηγούμενων αλμάτων που ήταν taken. Η απόδοση του επεξεργαστή βελτιώνεται, καθώς δε χρειάζεται να περιμένει τον υπολογισμό της διεύθυνσης του target και μπορεί να προχωρήσει κατευθείαν στη φόρτωση της επόμενης εντολής όταν το αποτέλεσμα του branch προβλεφθεί (ή γίνει γνωστό). Για αυτό το λόγο δε θα δούλευε αποδοτικά για τη πρόβλεψη των διευθύνσεων επιστροφής από ρουτίνες, μιας και η διεύθυνση επιστροφής δεν είναι πάντα ίδια και εξαρτάται από το σημείο του κώδικα (της εκτέλεσης) στο οποίο κλήθηκε η συγκεκριμένη ρουτίνα/συνάρτηση.

Θέμα 2

Ερώτημα Α: Αναφέρατε ένα πλεονέκτημα και ένα μειονέκτημα των χαλαρών μοντέλων συνέπειας μνήμης σε σχέση με το μοντέλο της ακολουθιακής συνέπειας.

Πλεονέκτημα: επιτρέπουν βελτιστοποιήσεις στην απόδοση (π.χ. bypassing των missing writes μέσω write-buffers).

Μειονέκτημα: τα προγράμματα απαιτούν σχολαστικό tuning από τον προγραμματιστή προκειμένου αφενός να τηρείται η σωστή σημασιολογία (που εξασφαλίζεται αυτομάτως από την SC), και αφετέρου να παίρνουμε το μέγιστο της απόδοσης (π.χ. εισαγωγή ελάχιστου αριθμού barriers/barriers με τους λιγότερους περιορισμούς).

Ερώτημα Β: i) Με ποιο τρόπο βελτιώνει την απόδοση του πρωτοκόλλου MESI το Exclusive state σε σχέση με το πρωτόκολλο MSI;

Η απόδοση βελτιώνεται μέσω της μείωσης του όγκου της κίνησης στο διάδρομο. Η μείωση αυτή επιτυγχάνεται, καθώς όταν ένα block είναι στο E δεν χρειάζεται να δημιουργήσει κάποιος bus transaction σε περίπτωση εγγραφής, μιας και είναι σίγουρο ότι η cache έχει το μοναδικό αντίγραφο.

ii) Στο πρωτόκολλο MESI μπορεί ένα block να μεταβεί από το S στο E, αν οι υπόλοιπες caches που έχουν το ίδιο block το διώξουν (evict); Γιατί;

Η μετάβαση αυτή δεν είναι εφικτή, καθώς δεν υπάρχει η πληροφορία πόσοι κόμβοι/caches μοιράζονται κάθε στιγμή ένα block. Έτσι όταν μια cache μείνει με το μοναδικό S block δεν το γνωρίζει και δεν μπορεί να μεταβεί στο E.

Θέμα 4

Ερώτημα Α: Για κάθε μία από τις ακόλουθες προτάσεις απαντήστε αν είναι σωστή ή λάθος, δικαιολογώντας την απάντηση σας. Υποθέστε ότι όλες οι παράμετροι της κρυφής μνήμης παραμένουν σταθερές εκτός αυτής στην οποίαν αναφέρεται η κάθε ερώτηση.

- Ο διπλασιασμός του associativity διπλασιάζει το πλήθος των tags στην cache. **ΛΑΘΟΣ**. Ο συνολικός αριθμός των cache lines παραμένει ίδιος, επομένως και ο αριθμός των tags.
- Ο διπλασιασμός της χωρητικότητας μιας direct-mapped cache συνήθως μειώνει τα conflict misses. **ΣΩΣΤΟ**. Ο διπλασιασμός της χωρητικότητας αυξάνει τις cache lines από L σε 2L. Ως εκ τούτου, οι διευθύνσεις i και i+L απεικονίζονται σε διαφορετικές θέσεις της cache, ενώ στην αρχική κατάσταση έκαναν conflict.
- Ο διπλασιασμός της χωρητικότητας μιας direct-mapped cache συνήθως μειώνει τα compulsory misses. **ΛΑΘΟΣ**. Διπλασιάζεται ο αριθμός των cache lines, όμως το μέγεθος της cache line παραμένει το ίδιο. Συνεπώς τα compulsory misses παραμένουν τα ίδια.
- Ο διπλασιασμός του μεγέθους του cache line συνήθως μειώνει τα compulsory misses. **ΣΩΣΤΟ**. Ο διπλασιασμός του Line size έχει σαν αποτέλεσμα τη φόρτωση περισσότερων δεδομένων κάθε φορά που έρχεται μία cache line στην cache. Αυτό αυξάνει την πιθανότητα δεδομένα που θα κληθούν για $1^η$ φορά σε επόμενες εντολές να βρίσκονται ήδη στη γραμμή που φορτώθηκε, μειώνοντας έτσι τα compulsory misses.

Επαναληπτική 10

Θέμα 1

Ερώτημα Α: Δώστε τους ορισμούς για τις RAW, WAR, WAW εξαρτήσεις καθώς και ένα παράδειγμα κώδικα για κάθε μια. Με ποιον τρόπο επιλύονται οι αντίστοιχοι hazards στις αρχιτεκτονικές που χρησιμοποιούν τον αλγόριθμο Tomasulo με ROB;

- **RAW**: Όταν μια εντολή παράγει μια τιμή που χρησιμοποιείται ως όρισμα από μια επόμενη.

```
add $r1, $r2, $r3
sub $r5, $r1, $r6
```

- **WAR**: Όταν μια μεταγενέστερη εντολή γράφει σε ένα καταχωρητή που πρέπει να διαβαστεί από μια προγενέστερη εντολή.

```
sub $r5, $r1, $r6
add $r1, $r2, $r3
```

- **WAW**: Όταν 2 εντολές γράφουν στον ίδιο καταχωρητή. Αν η εγγραφή γίνει out-of-order τότε ο καταχωρητής θα έχει τη λάθος τιμή.

```
add $r1, $r2, $r3
sub $r1, $r4, $r5
```

Ο Tomasulo επιλύει τους WAR και WAW hazards με register renaming. Οι RAW επιλύονται με stalls, μέσω των RS όπου οι εντολές περιμένουν να παραχθούν τα ορίσματα που χρειάζονται και forwarding των τιμών μέσω του CDB.

Ερώτημα Β: Αναφέρατε δύο λόγους για τους οποίους η τεχνική βελτιστοποίησης loop unrolling βελτιώνει την απόδοση και ένα λόγο για τον οποίο μπορεί να τη μειώσει.

Βελτίωση απόδοσης:

- Λιγότερα loop conditional evaluations.
- Λιγότερα branches/jumps.
- Μεγαλύτερη δυνατότητα για reordering εντολών που ανήκουν σε διαφορετικά iterations
- Δυνατότητα για συγχώνευση loads και stores που ανήκουν σε διαφορετικά iterations.

Μείωση απόδοσης:

- Αυξημένος φόρτος (pressure) στην I-cache.

Κανονική 09

Ερώτημα Α: Δίνεται επεξεργαστής με βαθιά σωλήνωση, ο οποίος χρησιμοποιεί Branch-Target Buffer (BTB) για την πρόβλεψη διακλαδώσεων. Ισχύουν τα εξής:

- 30% των εντολών είναι εντολές διακλάδωσης.
- Ο BTB έχει hit rate 80% ενώ η ακρίβεια πρόβλεψης του είναι 95%.
- Η ποινή σε περίπτωση λανθασμένης πρόβλεψης του BTB είναι 6 κύκλοι
- Σε περίπτωση αστοχίας του BTB, ο επεξεργαστής χρησιμοποιεί έναν two-level predictor (1,1). Η ποινή σε περίπτωση λανθασμένης πρόβλεψης του two-level predictor είναι 4 κύκλοι.
- Για κάθε εντολή διακλάδωσης η αποκωδικοποίηση και η πρόβλεψη γίνονται στο πρώτο στάδιο της σωλήνωσης. Ο έλεγχος ορθότητας της πρόβλεψης γίνεται στο τελευταίο στάδιο και σε περίπτωση λάθους η εκτέλεση της σωστής εντολής ξεκινά από τον επόμενο κύκλο.
- Οι εντολές που δεν προκαλούν stalls απαιτούν 1 κύκλο για να εκτελεστούν.

Υπολογίστε το ελάχιστο ποσοστό ακρίβειας πρόβλεψης του two-level predictor, ώστε ο επεξεργαστής αυτός να είναι πιο γρήγορος από έναν άλλο επεξεργαστή όπου δεν υπάρχει πρόβλεψη διακλάδωσης και κάθε εντολή διακλάδωσης κάνει stall την σωλήνωση για 2 κύκλους.

Για να συγκρίνουμε τους 2 επεξεργαστές θα πρέπει να υπολογίσουμε τον μέσο αριθμό κύκλων που απαιτεί η εκτέλεση κάθε εντολής.

$$CPI_A = 1 + 0.3 \cdot (0.8 \cdot 0.05 \cdot 6 + 0.2 \cdot (1 - h) \cdot 4) = 1.312 - 0.24h$$

$$CPI_B = 1 + 0.3 \cdot 2 = 1.6$$

Για να είναι πιο γρήγορος ο πρώτος υπολογιστής θα πρέπει να ισχύει: $1.312 - 0.24h \leq 1.6 \rightarrow h \geq -1.2$ που ισχύει ούτως ή άλλως αφού $h \geq 0$. Δηλαδή, ο πρώτος επεξεργαστής με αυτά τα χαρακτηριστικά είναι πάντα πιο γρήγορος από τον 2ο, ανεξάρτητα από το ποσοστό επιτυχίας του two-level predictor. Το πόσο πιο γρήγορος θα είναι προφανώς εξαρτάται από το h.

Ερώτημα Β: Δίνονται οι εξής 3 τεχνικές: α) Out-of-order issue with renaming, β) Branch prediction, γ) Superscalar. Για τα παρακάτω κομμάτια κώδικα, εξηγήστε ποια τεχνική θα επιλέγατε να εντάξετε στο σύστημά σας. Εξηγήστε επίσης τους λόγους για τους οποίους θα απορρίπτατε τις υπόλοιπες τεχνικές.

(i)	(ii)	(iii)
ADD F0, F1, F8	Loop: ADD R3, R4, R0	LD R1, 0(R2) # cache miss
ADD F2, F3, F8	LD R4, 8(R4)	ADD R2, R1, R1
ADD F4, F5, F8	BNEQZ R4, loop	LD R1, 0(R3) # cache hit
ADD F6, F7, F8		LD R3, 0(R4) # cache hit
		ADD R3, R1, R3
		ADD R1, R2, R3

i. Θα επιλέγαμε Superscalar μιας και οι εντολές δεν έχουν dependencies μεταξύ τους και μπορούν να εκτελεστούν παράλληλα εφόσον η αρχιτεκτονική διαθέτει πολλαπλά functional units. Το branch prediction δε θα προσέφερε κάτι μιας και δεν υπάρχει εντολή άλματος. Αντίστοιχα, και η τεχνική του out-of-order δε θα προσέφερε κάτι μιας και δεν υπάρχει κάποιο dependency, η επίλυση του οποίου να καθυστερεί την εκτέλεση κάποιας εντολής.

ii. Προφανώς θα επιλέγαμε να υλοποιήσουμε το branch prediction καθώς ο συγκεκριμένος κώδικας είναι ένα loop και η εντολή διακλάδωσης θα εκτελεστεί αρκετές φορές. Η superscalar αρχιτεκτονική δε θα βοηθούσε μιας και υπάρχουν dependencies μεταξύ των εντολών και άρα δεν μπορούν να

εκτελεστούν παράλληλα. Η out-of-order τεχνική θα έλυνε τα WAR αλλά τα RAW θα παρέμεναν. Ταυτόχρονα, καθώς δεν υπάρχει πρόβλεψη εντολή άλματος, η απόδοση θα περιοριζόταν καθώς θα έπρεπε να περιμένει να μάθει το αποτέλεσμα της διακλάδωσης. Έτσι δε θα βελτίωνε την απόδοση και ουσιαστικά οι εντολές θα κατέληγαν να εκτελεστούν in-order.

iii. Σε αυτή την περίπτωση θα επιλέγαμε out-of-order, καθώς θα μας επέτρεπε να εκτελέσουμε την 3η, 4η και 5η εντολή καθώς περιμένουμε να ικανοποιηθεί η αστοχία μνήμης της 1ης εντολής. Το branch prediction προφανώς δεν βοηθά καθώς δεν υπάρχουν branches, ενώ η superscalar αρχιτεκτονική δεν μπορεί να εκτελέσει παράλληλα τις εντολές λόγω των dependencies που υπάρχουν και οδηγούν σε hazards.

Ερώτημα Γ: Δίνεται επεξεργαστής με ένα επίπεδο κρυφής μνήμης. Πώς θα επηρεαστούν τα Compulsory, Conflict και Capacity misses, αν γίνει ξεχωριστά το καθένα από τα παρακάτω; Θα αυξηθούν, θα μειωθούν ή θα μείνουν αμετάβλητα; Εξηγήστε γιατί.

1. Διπλασιασμός του associativity, διατηρώντας σταθερό το μέγεθος του block και της cache.
    - Η συνέπεια της κίνησης αυτής είναι η μείωση στο μισό του αριθμού των sets. Τα compulsory misses παραμένουν σταθερά καθώς η αύξηση του associativity δεν επηρεάζει το πότε θα έρθουν για πρώτη φορά δεδομένα μέσα στην cache. Σταθερά παραμένουν και τα capacity misses, μιας και το capacity της cache παραμένει σταθερό. Τα conflict misses περιμένουμε όμως να μειωθούν, καθώς το μεγαλύτερο associativity σημαίνει ότι υπάρχει περισσότερος χώρος για να αποθηκευθούν στοιχεία που είναι mapped στο ίδιο set.
2. Υποδιπλασιασμός του μεγέθους του block, διατηρώντας σταθερό το associativity και τον αριθμό των sets.
    - Η συνέπεια της κίνησης αυτής είναι η μείωση στο μισό της χωρητικότητας της cache. Καθώς ο αριθμός των sets και ways παραμένει σταθερός τα conflict misses δεν επηρεάζονται. Η μείωση στο μισό όμως της χωρητικότητας προφανώς θα επιφέρει αύξηση των capacity misses. Επίσης, θα υπάρξει αύξηση των compulsory misses, καθώς μικρότερο block σημαίνει μικρότερη αξιοποίηση της spatial locality και "λιγότερο" prefetching μιας και τώρα θα έρχονται λιγότερα στοιχεία στην cache για κάθε miss.
3. Διπλασιασμός του αριθμού των sets διατηρώντας σταθερό το μέγεθος του block καθώς και το συνολικό μέγεθος της cache.
    - Η κίνηση αυτή συνεπάγεται μείωση στο μισό του associativity. Τα compulsory misses παραμένουν σταθερά καθώς δεν επηρεάζεται το πότε θα φορτωθούν δεδομένα στην cache. Αντίστοιχα σταθερά παραμένουν και τα capacity misses, μιας και η συνολική χωρητικότητα παραμένει σταθερή. Η μείωση όμως του associativity θα επιφέρει αύξηση των conflict misses.
4. Εφαρμογή προφόρτωσης δεδομένων (prefetching).
    - Το prefetching μπορεί να μειώσει τα compulsory misses καθώς αν λειτουργεί σωστά θα φορτώνει δεδομένα στην cache πριν αυτά ζητηθούν για πρώτη φορά. Τα υπόλοιπα misses παραμένουν σταθερά. Υπάρχει βέβαια περίπτωση σε κάποιο μη αποδοτικό prefetching, να έχουμε φαινόμενα cache pollution/thrashing, στην οποία περίπτωση θα αυξηθούν και τα conflict και τα capacity misses.
5. Εφαρμογή τεχνικής blocking.
    - Το blocking μειώνει τα capacity misses, ενώ αφήνει ανεπηρέαστα τα conflict misses καθώς δεν μπορεί να κάνει κάτι για τα στοιχεία εκείνα που είναι mapped στο ίδιο set. Αντίστοιχα, δεν επηρεάζει το πότε θα φορτωθούν για πρώτη φορά στοιχεία στην cache και άρα τα compulsory misses παραμένουν σταθερά.

Κανονική 08

Θέμα 1

Ερώτημα Α: Είστε υπεύθυνοι σχεδιασμού ενός out-of-order επεξεργαστή με δυναμική δρομολόγηση (dynamic scheduling) και υποθετική εκτέλεση (speculative execution) εντολών. Ο επεξεργαστής που σχεδιάσατε έχει 8 Integer Functional Units (IUs), 4 Floating Point Units (FPUs), 256KB on-chip caches, 4 reservation stations για integer και 2 για floating point. Ο Reorder Buffer έχει 8 θέσεις, ενώ το pipeline αποτελείται από 25 στάδια. Τέλος, στο σχέδιο σας έχετε συμπεριλάβει ένα μικρό 2-bit branch predictor, ο οποίος αποδίδει σχετικά καλά. Οι εφαρμογές για τις οποίες θα χρησιμοποιηθεί ο επεξεργαστής έχουν λίγο κώδικα και επεξεργάζονται δεδομένα μεγέθους περίπου 64KB. Την περισσότερη ώρα οι εφαρμογές εκτελούν loops, των οποίων οι επαναλήψεις είναι μεταξύ τους ανεξάρτητες, ενώ σε κάθε μία από αυτές υπάρχει περιορισμένη δυνατότητα εκμετάλλευσης του ILP. Μετά την αξιολόγηση του σχεδίου, σας ενημερώνουν ότι υπάρχουν ακόμα διαθέσιμα transistors τα οποία θα μπορούσατε να χρησιμοποιήσετε με κάποιον από τους παρακάτω τρόπους:

(i) Βελτίωση της ακρίβειας πρόβλεψης του branch predictor.

Ο predictor αποδίδει “σχετικά καλά” και γνωρίζουμε ότι ο κώδικας μας περιέχει πολλά branches. Ταυτόχρονα, το pipeline είναι αρκετά μεγάλο καθιστώντας τα λάθος προβλέψεις πολύ ακριβές. Θα επιλέγαμε επομένως να αυξήσουμε την ακρίβεια του predictor (και μέσω αυτής την απόδοση του συστήματος).

(ii) Αύξηση του μεγέθους της cache.

Δεν θα το επιλέγαμε. Το σύστημα έχει ήδη αρκετή μνήμη για να χωρέσει το working set των εφαρμογών, επομένως δε θα βλέπαμε σημαντική μείωση των miss rates.

(iii) Προσθήκη περισσότερων reservation stations.

Θα επιλέγαμε να προσθέσουμε περισσότερα RS, καθώς αυτό θα σήμαινε ότι το σύστημα θα είχε ένα μεγαλύτερο window στο οποίο θα έψαχνε για εντολές που θα μπορούσαν να εκτελεστούν παράλληλα. Θα έδινε πχ. τη δυνατότητα στο σύστημα μας να εκμεταλλευτεί τυχόν παραλληλισμό μεταξύ διαφορετικών επαναλήψεων ενός loop.

(iv) Προσθήκη περισσότερων IUs και FPUs.

Δεν θα το επιλέγαμε, τουλάχιστον σε σύγκριση με τις υπόλοιπες επιλογές. Αν ο επεξεργαστής μας δεν έχει τη δυνατότητα να ανακαλύπτει περισσότερο ILP τότε δεν έχει νόημα να προσθέσουμε περισσότερα execution units, καθώς θα μένουν ανενεργά.

(v) Προσθήκη περισσότερων θέσεων στον ROB.

Ο ROB έχει πολύ λίγες θέσεις. Θα επιλέγαμε λοιπόν να τον μεγαλώσουμε, καθώς θα βοηθούσε να κρυφτούν οι μεγάλες καθυστερήσεις (λόγω long latency instructions) και θα έδινε τη δυνατότητα στον επεξεργαστή να ψάξει για παραλληλισμό σε ένα μεγαλύτερο window εντολών (όμοια με την (iii)).

Ερώτημα Β: Για ποιο λόγο, οι αρχιτεκτονικές που χρησιμοποιούν τον αλγόριθμο Tomasulo χρειάζονται κάποιο μηχανισμό πρόβλεψης διακλάδωσης;

Ο αλγόριθμος Tomasulo χρησιμοποιείται για την υλοποίηση out-of-order εκτέλεσης εντολών. Τα συστήματα αυτά επιτυγχάνουν βελτίωση της απόδοσης τους εκτελώντας εντολές οι οποίες ακολουθούν άλλες, οι οποίες όμως είναι blocked περιμένοντας το αποτέλεσμα κάποιων προηγούμενων εντολών από τις οποίες

εξαρτώνται. Για να γίνει εφικτό αυτό, οι αρχιτεκτονικές αυτές απαιτείται να δρομολογούν πολλές εντολές, οι οποίες βρίσκονται πιο μπροστά στο instruction stream. Όταν λοιπόν συναντούν ένα branch απαιτείται ένας μηχανισμός πρόβλεψης, ώστε να ξέρουν ποιες εντολές πρέπει να δρομολογήσουν στη συνέχεια. Διαφορετικά, θα πρέπει να περιμένουν το αποτέλεσμα του branch χάνοντας έτσι το κέρδος της ooo execution.

Ερώτημα Γ: Υποθέστε ότι έχετε έναν επεξεργαστή ο οποίος υλοποιεί τον αλγόριθμο Tomasulo με τη βοήθεια ROB και ο οποίος έχει άπειρους επεξεργαστικούς πόρους (πχ. άπειρα reservation units και άπειρα execution units). Υποθέστε επίσης ότι ο μηχανισμός πρόβλεψης διακλάδωσης είναι ιδανικός και πως δεν υπάρχουν dependences μεταξύ των εντολών που εκτελούνται. Τέλος, το fetch στάδιο μπορεί να φορτώσει 12 εντολές ανά κύκλο, ενώ τα υπόλοιπα στάδια δεν έχουν περιορισμοί (πχ. αποκωδικοποίηση άπειρων εντολών ανά κύκλο).

(i) Αν όλες οι εντολές απαιτούν 1 κύκλο εκτέλεσης και ο ROB έχει άπειρες θέσεις ποιος είναι ο μέσος ρυθμός εκτέλεσης εντολών ανά κύκλο (IPC) του επεξεργαστή;

Το IPC περιορίζεται μόνο από το ρυθμό με τον οποίο οι εντολές γίνονται fetched. Δεν υπάρχουν data dependencies ή περιορισμοί στη δρομολόγηση ενώ έχουμε και άπειρους επεξεργαστικούς πόρους. Επομένως

$$IPC = 12$$

(ii) Αν για το σύστημα του ερωτήματος (i) έχετε κάθε 48 εντολές ένα load το οποίο ακολουθούν και τα miss χρειάζονται 500 κύκλους για να ικανοποιηθούν, πόσο θα είναι τώρα το IPC του επεξεργαστή;

Ο ROB είναι άπειρος, δηλαδή συνεχίζουμε να κάνουμε fetch και issue 12 εντολές ανά κύκλο. Το κάθε miss χρειάζεται 500 κύκλους από το fetch μέχρι το commit, αλλά αυτές οι καθυστερήσεις “κρύβονται” μέσω του ROB και το average throughput παραμένει 12.

(iii) Αν στο ερώτημα (ii) περιορίσετε τις θέσεις του ROB από άπειρες σε 48, το IPC του επεξεργαστή μεταβάλλεται; Αν το IPC είναι μικρότερο από 12, ποιο το μέγεθος του ROB που απαιτείται ώστε IPC = 12;

Προφανώς το IPC θα είναι μικρότερο του 12, μιας και λόγω του μεγέθους του ROB θα έχουμε stalls. Έστω ότι στον 1ο κύκλο έχουμε ένα load που προκαλεί miss. Το σύστημα θα συνεχίσει να φέρνει και να δρομολογεί εντολές μέχρι να γεμίσει ο ROB, δηλαδή 47 ακόμα εντολές θα γίνουν issued και μετά θα έχουμε stall. Στον κύκλο 500 το load που βρίσκεται στην κορυφή του ROB θα είναι έτοιμο να γίνει commit, μαζί βέβαια με τις υπόλοιπες 47 εντολές. Στο σημείο αυτό γίνεται issued το επόμενο load που προκαλεί miss και ο κύκλος επαναλαμβάνεται. Δηλαδή κάθε 500 κύκλους μπορούμε να κάνουμε commit 48 εντολές κι άρα

$$IPC = 48/500$$

Για να έχουμε IPC = 12, πρέπει το σύστημα να μπορεί να διατηρεί ρυθμό εκτέλεσης 12 εντολές ανά κύκλο κατά τη διάρκεια των 500 κύκλων όπου περιμένουμε το load (οι εντολές αυτές βέβαια θα γίνονται commit μαζί με το load). Επομένως απαιτείται το ROB να έχει τουλάχιστον 12 * 500 = 6000 θέσεις.

Κανονική 23

Θέμα 1

Ερώτημα Α

Απαντήστε συνοπτικά στις παρακάτω ερωτήσεις:

i. Τι ονομάζουμε register renaming και με ποιο τρόπο είναι βοηθητικό;
ii. Πως αντιμετωπίζει ο αλγόριθμος Tomasulo τους RAW hazards;
iii. Για ποιο λόγο χρειάζονται μηχανισμούς πρόβλεψης εντολών άλματος οι επεξεργαστές που υλοποιούν Tomasulo;
iv. Όταν μια εντολή ολοκληρώνει την εκτέλεση της, το αποτέλεσμα της αποθηκεύεται στο ROB αλλά και στα Value πεδία όσων Reservation Stations χρειάζεται. Για ποιο λόγο την αποθηκεύουμε στο R5 από τη στιγμή που η τιμή είναι διαθέσιμη στο ROB;

Λύση

Υποερώτημα (i)

Κανονική 11, Θέμα 1Α.

Υποερώτημα (ii)

Επαναληπτική 10, Θέμα 1Α.

Υποερώτημα (iii)

Κανονική 12, Θέμα Θ1Α.

Υποερώτημα (iv)

Κανονική 11, Θέμα 1Β.

Ερώτημα Β

Αναλαμβάνετε επικεφαλής σχεδίασης ενός καινούργιου out-of-order επεξεργαστή που υλοποιεί Tomasulo χρησιμοποιώντας ROB για in-order commit εντολών. Το προηγούμενο μοντέλο του επεξεργαστή έχει τα εξής χαρακτηριστικά:

- Βαθύ pipeline 25 σταδίων.
- 8 RS και 12 FUs για integer αριθμούς.
- 4 RS και 6 FUs για floating point αριθμούς.
- 8 θέσεις στον ROB.
- 256KB on chip caches.
- 2-bit saturating counter branch predictor με σχετικά καλή ακρίβεια.

Διαπιστώνετε πως οι εφαρμογές που θα εκτελούνται στον καινούργιο επεξεργαστή έχουν μικρό κώδικα, επεξεργάζονται μικρά datasets της τάξης των 64KB και περνούν σχεδόν την περισσότερη ώρα μέσα σε loops, οι επαναλήψεις των οποίων είναι ανεξάρτητες μεταξύ τους, ενώ η κάθε επανάληψη εμφανίζει περιορισμένο ILP. Για κάθε μία από τις παρακάτω αλλαγές, εξηγήστε συνοπτικά αν θα την επιλέγατε για να βελτιώσετε την επίδοση του επεξεργαστή σε σχέση με το προηγούμενο μοντέλο.

i. Αντικατάσταση του branch predictor με μηχανισμό υψηλότερης ακρίβειας.
ii. Αύξηση του μεγέθους της cache.
iii. Προσθήκη περισσότερων FUs.
iv. Αύξηση των θέσεων του ROB.

Λύση

Κανονική 08, Θέμα 1Α.

Ερώτημα Γ

Εργάζεστε στη σχεδίαση του AVP (ACME Vector Processor). Η εφαρμογή ενδιαφέροντος είναι το SAXPY (a*x[i]+y[i]) για διανύσματα διπλής ακρίβειας (double) μεγάλου μεγέθους. Η καρδιά του διανυσματικού κώδικα είναι η παρακάτω, όπου για λόγους απλότητας αγνοούμε το κόστος περιφερειακού κώδικα (ενημέρωση δεικτών και διακλαδώσεις):

LV V1, Rx	; load vector X
MULVS.D V2, V1	; vector-scalar multiply
LV V3, Ry	; load vector Y
ADDVV.D V4, V2, V3	; add two vectors
SV Ry, V4	; store the sum

Η αρχική σχεδίαση έχει τα εξής χαρακτηριστικά:

- Vector length: 128 στοιχεία.
- Μνήμη: 8 banks, καθυστέρηση πρόσβασης 8 κύκλοι, πλάτος 64 bit, με ξεχωριστές πόρτες ανάγνωσης/εγγραφής.
- Αθροιστής: Πλήρως pipelined, καθυστέρηση 2 κύκλοι.
- Πολλαπλασιαστής: Πλήρως pipelined, καθυστέρηση 4 κύκλοι.
- Πλάτος έκδοσης: 1 εντολή ανά κύκλο, υποστηρίζει chaining, 1 lane.

i. Υπολογίστε αναλυτικά τον χρόνο εκτέλεσης για αυτή τη βασική σχεδίαση και για μήκος διανύσματος 128 στοιχεία (ένα iteration του loop). Υπόδειξη: ένα διάγραμμα όπως αυτό του χρονισμού pipeline μπορεί να είναι χρήσιμο – αν δεν είναι απόλυτα απαραίτητο.
ii. Προτείνεται η βελτίωση του συστήματος με την προσθήκη ενός ακόμα lane (σύνολο 2 lanes) για υπολογισμούς αλλά και στο σύστημα μνήμης. Υπολογίστε τον νέο χρόνο εκτέλεσης και εκτιμήστε το κόστος και την πολυπλοκότητα υλοποίησης της νέας σχεδίασης. Ποιο κομμάτι της σχεδίασης θα γίνει πιο πολύπλοκο/ακριβό/κλπ. και με ποιο τρόπο;

Λύση

Υποερώτημα (i)

Ανάλυση λειτουργίας SAXPY:

1. **Load Vector X** (LV V1, Rx):
   - Φορτώνει 128 στοιχεία στο V1.
   - Καθυστέρηση μνήμης: 8 κύκλοι.

- Μήκος vector: 128 στοιχεία, πλάτος μνήμης: 64 bits.
- Κάθε φόρτωση φέρνει 64 bits = 8 bytes (αφού διπλή ακρίβεια σημαίνει ότι κάθε στοιχείο είναι 8 bytes).
- Συνεπώς, 128 στοιχεία * 8 bytes/στοιχείο / 64 bits/πλάτος τράπεζας = 16 πράξεις μνήμης.
- Με 8 memory banks, κάθε λειτουργία παραλληλίζεται πλήρως στις τράπεζες.

2. **Vector-scalar multiply** (MULVS.D V2, V1, F0):
   - Πολλαπλασιάζει κάθε στοιχείο του V1 με το κλιμάκιο F0.
   - Καθυστέρηση του πολλαπλασιαστή: 4 κύκλοι.
   - Fully pipelined, οπότε η απόδοση είναι 1 στοιχείο ανά κύκλο μετά την αρχική καθυστέρηση.
3. **Load Vector Y** (LV V3, Ry):
   - Παρόμοια με το load vector X, απαιτεί 16 πράξεις μνήμης.
   - Κάθε φόρτωση απαιτεί επίσης 8 κύκλους καθυστέρησης λόγω της διαμόρφωσης της μνήμης.
4. **Vector Sum** (ADDVV.D V4, V2, V3):
   - Προσθέτει τα στοιχεία των V2 και V3 για να προκύψει το V4.
   - Καθυστέρηση του αθροιστή: 2 κύκλοι.
   - Fully pipelined, η απόδοση είναι 1 στοιχείο ανά κύκλο μετά την αρχική καθυστέρηση.
5. **Store Sum** (SV Ry, V4):
   - Αποθηκεύει 128 στοιχεία από το V4 πίσω στη μνήμη.
   - Παρόμοιος χρόνος πρόσβασης στη μνήμη: 16 πράξεις, καθυστέρηση 8 κύκλων.

Διάγραμμα χρονισμού αγωγού:

Operation	**Cycle**	**Comment**
LV V1, Rx	1-16	16 operations over 8 banks, 8 cycles latency
MULVS.D V2, V1	17-20	Initial 4 cycles latency, then 1 element per cycle
LV V3, Ry	21-36	16 operations over 8 banks, 8 cycles latency
ADDVV.D V4, V2, V3	37-38	Initial 2 cycles latency, then 1 element per cycle
SV Ry, V4	29-54	16 operations over 8 banks, 8 cycles latency

Κάθε λειτουργία ολοκληρώνεται διαδοχικά λαμβάνοντας υπόψη τις εξαρτήσεις και το chaining.

- Η Load V1 ολοκληρώνεται στον κύκλο 16.
- Η MULVS.D ξεκινά στον κύκλο 17 και είναι fully pipelined, οπότε ολοκληρώνει το πρώτο στοιχείο στον κύκλο 20 και το τελευταίο στοιχείο στον κύκλο 147 (130 κύκλοι για όλα τα στοιχεία).
- Η Load V3 ξεκινά στον κύκλο 21 (λόγω chaining με το MULVS.D), ολοκληρώνεται στον κύκλο 36.
- Η ADDVV.D ξεκινά στον κύκλο 37, ολοκληρώνει το πρώτο στοιχείο στον κύκλο 38 και το τελευταίο στοιχείο στον κύκλο 166 (130 κύκλοι για όλα τα στοιχεία).
- Η Load V4 ξεκινά στον κύκλο 39 (λόγω chaining με την ADDVV.D), ολοκληρώνεται στον κύκλο 54 και ολοκληρώνει το τελευταίο στοιχείο στον κύκλο 184 (130 κύκλοι για όλα τα στοιχεία).

Συνολικός χρόνος εκτέλεσης: 184 κύκλοι για μία επανάληψη του βρόχου με μήκος διανύσματος 128 στοιχείων.

Υποερώτημα (ii)

Νέα χαρακτηριστικά: Δύο lanes για υπολογισμό και μνήμη. Αυτό σημαίνει ότι:

- Οι λειτουργίες Load και Store μπορούν να μειωθούν κατά το ήμισυ σε χρόνο, επειδή τώρα μπορούν να επεξεργαστούν ταυτόχρονα δύο στοιχεία.
- Αριθμητικές πράξεις (MULVS.D και ADDVV.D) μπορούν επίσης να μειωθούν στο μισό λόγω των δύο lanes.

Operation	**Cycle**	**Comment**
LV V1, Rx	1-8	16 operations halved, each cycle does 2 operations
MULVS.D V2, V1	9-12	Initial 4 cycles latency, then 2 elements per cycle
LV V3, Ry	13-20	16 operations halved, each cycle does 2 operations
ADDVV.D V4, V2, V3	21-22	Initial 2 cycles latency, then 2 elements per cycle
SV Ry, V4	23-30	16 operations halved, each cycle does 2 operations

Συνολικός χρόνος εκτέλεσης με 2 lanes: 30 κύκλοι. Το νέο κόστος και πολυπλοκότητα υλοποίησης της νέας σχεδίασης:

- *Memory System*:
  - **Κόστος**: Ο διπλασιασμός των lanes θα απαιτούσε διπλασιασμό του εύρους ζώνης της μνήμης, πράγμα που σημαίνει προσθήκη πρόσθετων θυρών ή τραπεζών μνήμης για την υποστήριξη ταυτόχρονων αναγνώσεων και εγγραφών. Αυτό αυξάνει την πολυπλοκότητα του ελεγκτή μνήμης.
  - **Πολυπλοκότητα**: Η διαχείριση της ταυτόχρονης πρόσβασης στη μνήμη και η εξασφάλιση της συνοχής γίνεται πιο πολύπλοκη.
- *Functional Units*:
  - **Κόστος**: Ο διπλασιασμός του αριθμού των FUs αυξάνει την επιφάνεια του πυριτίου και την κατανάλωση ισχύος.
  - **Πολυπλοκότητα**: Η λογική ελέγχου για την αποστολή, τον συγχρονισμό και την αλυσιδωτή σύνδεση εντολών σε πολλαπλές λωρίδες γίνεται πιο πολύπλοκη. Η εξασφάλιση της αποτελεσματικής χρήσης και των δύο lanes χωρίς bottlenecks προσθέτει επίσης πολυπλοκότητα.
- *Pipeline*:
  - **Κόστος**: Πρόσθετα pipelines και μηχανισμοί ελέγχου για τη διαχείριση δύο lanes.
  - **Πολυπλοκότητα**: Αυξημένη πολυπλοκότητα στην ανίχνευση κινδύνων, την προώθηση δεδομένων και τον συνολικό έλεγχο του pipeline για την εξασφάλιση ομαλής εκτέλεσης σε όλα τα lanes.

Θέμα 2

Δίνεται αρχιτεκτονική η οποία υλοποιεί τον αλγόριθμο Tomasulo χρησιμοποιώντας ROB για in-order commit εντολών. Το pipeline του επεξεργαστή περιέχει τα στάδια Issue (IS), Execute (EX), Write Result (WR) Commit και (CMT), αγνοούμε δηλαδή τα IF και ID. Ισχύουν επίσης τα ακόλουθα:

- Τα IS, WR, CMT απαιτούν 1 κύκλο.
- Το σύστημα διαθέτει περιορισμένο αριθμό από reservation stations (RS). Συγκεκριμένα, περιέχει 2 RS για προσθέσεις/αφαιρέσεις και 2 RS για πολλαπλασιασμούς/διαιρέσεις floating point αριθμών. Αντίστοιχα, για integer αριθμούς περιλαμβάνονται 4 RS για εντολές διακλάδωσης, αριθμητικές και λογικές εντολές καθώς και 1 RS για πολλαπλασιασμούς/διαιρέσεις.
- Το σύστημα περιλαμβάνει 4 non-pipelined functional unit για πράξεις integer αριθμών. Όλες οι εντολές μεταξύ integer αριθμών διαρκούν 2 κύκλους.
- Το σύστημα περιλαμβάνει 2 non-pipelined floating point functional units, 1 για ADD/SUBD και 1 για MULD/DIVD. Οι εντολές πρόσθεσης/αφαίρεσης διαρκούν 3 κύκλους, ενώ οι εντολές πολλαπλασιασμού/διαίρεσης 5 κύκλους.
- Για τις εντολές αναφοράς στη μνήμη, στο στάδιο EX γίνεται τόσο ο υπολογισμός της διεύθυνσης αναφοράς όσο και η προσπέλαση στη μνήμη. Το σύστημα περιλαμβάνει ένα Load και ένα Store Queue καθένα από τα οποία διαθέτει 2 θέσεις. Οι εντολές χρησιμοποιούν ένα ξεχωριστό pipelined functional unit για τον υπολογισμό της διεύθυνσης και διαρκούν 1 κύκλο στην περίπτωση Hit στην cache και 5 κύκλους σε περίπτωση Miss.
- Οι εντολές διακλάδωσης υπό συνθήκη χρησιμοποιούν τα κατάλληλα RS και FU για αφαίρεση, προκειμένου να υπολογίσουν αν ισχύει η συνθήκη. Η πρόβλεψη για μια εντολή διακλάδωσης υπό συνθήκη γίνεται ταυτόχρονα με τη δρομολόγηση της εντολής. Ο έλεγχος της πρόβλεψης γίνεται αμέσως μόλις γίνει γνωστό το αποτέλεσμα της εντολής, δηλαδή στο στάδιο WR (κύκλος k). Σε περίπτωση σφάλματος, σταματά η εκτέλεση των εντολών του miss-predicted execution path και στον επόμενο κύκλο (κύκλος k+1) δρομολογείται η σωστή εντολή.
- Ο ROB έχει 9 θέσεις.
- Το σύστημα περιλαμβάνει 1 CBD. Σε περίπτωση που παραπάνω από 1 εντολές θέλουν να το χρησιμοποιήσουν, τότε προτεραιότητα αποκτά η παλαιότερη εντολή (αυτή που έγινε issued πρώτη). Θεωρήστε ότι τα branches **δεν** χρησιμοποιούν το CDB κατά τη διάρκεια του WR σταδίου.
- Για τις εντολές διακλάδωσης υπό συνθήκη, το σύστημα χρησιμοποιεί έναν (2,2) global history predictor με συνολικά 8 entries, τα οποία φαίνονται στον παρακάτω πίνακα. Ο BHR έχει τιμή ίση με 0. Δίνεται επίσης το FSM διάγραμμα του 2-bit predictor, ο οποίος προβλέπει T για τιμές ≥ 2 και NT για τις υπόλοιπες.

	0 = 00	**1 = 01**	**2 = 10**	**3 = 11**
0	00	00	01	01
1	~~01~~ 00	11	10	00

$$00 \underset{N}{\overset{T}{\rightleftarrows}} 01 \underset{N}{\overset{T}{\rightleftarrows}} 10 \underset{N}{\overset{T}{\rightleftarrows}} 11$$

- Η δεικτοδότηση του πίνακα γίνεται χρησιμοποιώντας τον BHR καθώς και τον κατάλληλο αριθμό low order bits από το PC της εντολής. Ο επεξεργαστής υλοποιεί το ISA του MIPS.
- Το σύστημα περιλαμβάνει μια fully associative cache μεγέθους 32B με block size 16 bytes και πολιτική αντικατάστασης LRU. Αρχικά, η cache είναι άδεια.

- Οι καταχωρητές R1, R2 περιέχουν τις διευθύνσεις των $1^{ων}$ στοιχείων των πινάκων A και B αντίστοιχα, στους οποίους έχουν αποθηκευτεί αριθμοί διπλής ακρίβειας (μήκους 8B ο καθένας).
- Οι πίνακες είναι ευθυγραμμισμένοι.
- Δίνεται R8=1. Εκτελέστε τον κώδικα και δώστε τους χρόνους δρομολόγησης, εκτέλεσης και ολοκλήρωσης των εντολών σε έναν πίνακα όπως ο παρακάτω. ***Ποια τα τελικά περιεχόμενα της cache;***

OP	IS	EX	WR	CMT	Comment
LD F0,0(R1)					

Στο πεδίο «Σχόλιο» δικαιολογήστε τυχόν καθυστερήσεις μεταξύ IS-EX, EX-WR, WR-SMT καθώς και ακυρώσεις και ακυρώσεις εντολών.

Λύση

- IS, WR, CMT: 1 cycle
- ROB: 9 θέσεις
- 1 CBD:
  - Προτεραιότητα η παλαιότερη εντολή.
  - Το branches δεν χρησιμοποιούν το CDB στο WR.
- Πρόβλεψη διακλάδωσης ταυτόχρονη με δρομολόγηση εντολής.
- Έλεγχος πρόβλεψης: WR
- Δρομολόγηση σωστής εντολής: WR+1
- BHR = 00
- R8 = 1
- Cache: Fully Associative
  - Μέγεθος: 32 B
  - Block Size: 16 B
- Πολιτική Αντικατάστασης:
  - R1 → A[0] → 8 bytes
  - R2 → B[0] → 8 bytes

	RS	FU	Cycles
ADDD/SUBD	2	1	3
MULD/DIVD	2	1	5
BR/ADD/SUB/AND	4	4	2
MUL/DIV	1		2
LOAD	2		H:1, M: 5
STORE	2		H:1, M: 5

- IS: Χώρος στο FU + Χώρος στο RS
- EX: Χώρος στο FU + Εξαρτήσεις
- WR: Χώρος στο CBD
- CMT: Μετά από το προηγούμενο

	8 bytes	8 bytes
0	~~A[0]~~ B[4]	~~A[1]~~ B[5]
1	~~B[0]~~ A[2]	~~B[1]~~ A[3]

Εντολή	IS	EX	WR	CMT	Σχόλια
LD F0, 0(R1)	1	2-6	7	8	Miss A[0], LRU=1
ADDD F4, F4, F0	2	8-10	11	12	RAW F0
LD F1, 8(R2)	3	4-8	9	13	Miss B[1], LRU=0
ADDI R1, R1, 0x10	4	5-6	8	14	R1=A[2], CDB Conflict
MULD F4, F4, F1	5	12-16	17	18	RAW F1, F4
ADDI R2, R2, 0x10	6	7-8	10	19	R2=B[2], CDB Conflict
SUBI R8, R8, 0x1	7	8-9	12	20	R8=0, CDB Conflict
ANDI R9, R8, 0x1	8	13-14	15	21	RAW R8, R9=0
BNEZ R9, REP ***(Dir: 4)***	9	16-17	18	22	RAW R9, Pred=NT, Res=NT, (4=0**1**00, BHR=00), BHR'=00
ADDI R2, R2, 0x10	11	12-13	14	23	R2=B[4]
LD F1, 8(R2)	13	15-19	20	24	RAW R2, Miss B[5], LRU=1
MULD F1, F1, F1	14	17-21	22	25	RAW F1
MULD F4, F4, F1	18	23-27	28	29	RAW F1, F4
ADDD F4, F4, F4	19	29-31	32	33	RAW F4
BEZ R8, END ***(Dir: C)***	20	21-22	23	34	Pred=NT, Res=T (C=1**1**00, BHR=00)
X *LD F0, 8(R1)*	21	22-X	X	X	Miss A[3], LRU=0
X *MULD F4, F4, F0*	22	X	X	X	RAW F0, F4
X *SUBI R8, R8, 0x1*	23	X	X	X	Flush
SD F4, 0(R1)	24	25	26	35	Hit, LRU=0

Κανονική 19

Θέμα 2

Δίνεται αρχιτεκτονική η οποία υλοποιεί τον αλγόριθμο Tomasulo χρησιμοποιώντας ROB για in-order commit εντολών. Το pipeline του επεξεργαστή περιέχει τα στάδια Issue (IS), Execute (EX), Write Result (WR) Commit και (CMT), αγνοούμε δηλαδή τα IF και ID. Ισχύουν επίσης τα ακόλουθα:

- Τα IS, WR, CMT απαιτούν 1 κύκλο.
- Το σύστημα διαθέτει περιορισμένο αριθμό από reservation stations (RS). Συγκεκριμένα, περιέχει 3 RS για προσθέσεις/αφαιρέσεις και 2 RS για πολλαπλασιασμούς/διαιρέσεις floating point αριθμών. Αντίστοιχα, για integer αριθμούς περιλαμβάνονται 3 RS για εντολές διακλάδωσης, αριθμητικές και λογικές εντολές καθώς και 1 RS για πολλαπλασιασμούς/διαιρέσεις.
- Το σύστημα περιλαμβάνει 3 pipelined functional unit για πράξεις integer αριθμών. Όλες οι εντολές μεταξύ integer αριθμών διαρκούν 2 κύκλους.
- Το σύστημα περιλαμβάνει 2 non-pipelined floating point functional units, 1 για ADD/SUBD και 1 για MULD/DIVD. Οι εντολές πρόσθεσης/αφαίρεσης διαρκούν 3 κύκλους, ενώ οι εντολές πολλαπλασιασμού/διαίρεσης 5 κύκλους.
- Για τις εντολές αναφοράς στη μνήμη, στο στάδιο EX γίνεται τόσο ο υπολογισμός της διεύθυνσης αναφοράς όσο και η προσπέλαση στη μνήμη. Το σύστημα περιλαμβάνει ένα Load και ένα Store Queue καθένα από τα οποία διαθέτει 2 θέσεις. Οι εντολές χρησιμοποιούν ένα ξεχωριστό pipelined functional unit για τον υπολογισμό της διεύθυνσης και διαρκούν 1 κύκλο στην περίπτωση Hit στην cache και 4 κύκλους σε περίπτωση Miss.
- Οι εντολές διακλάδωσης υπό συνθήκη χρησιμοποιούν τα κατάλληλα RS και FU για αφαίρεση, προκειμένου να υπολογίσουν αν ισχύει η συνθήκη. Η πρόβλεψη για μια εντολή διακλάδωσης υπό συνθήκη γίνεται ταυτόχρονα με τη δρομολόγηση της εντολής. Ο έλεγχος της πρόβλεψης γίνεται αμέσως μόλις γίνει γνωστό το αποτέλεσμα της εντολής, δηλαδή στο στάδιο WR (κύκλος k). Σε περίπτωση σφάλματος, σταματά η εκτέλεση των εντολών του miss-predicted execution path και στον επόμενο κύκλο (κύκλος k+1) δρομολογείται η σωστή εντολή.
- Ο ROB έχει 10 θέσεις.
- Το σύστημα περιλαμβάνει 1 CBD. Σε περίπτωση που παραπάνω από 1 εντολές θέλουν να το χρησιμοποιήσουν, τότε προτεραιότητα αποκτά η παλαιότερη εντολή (αυτή που έγινε issued πρώτη). Θεωρήστε ότι τα branches δεν χρησιμοποιούν το CDB κατά τη διάρκεια του WR σταδίου.
- Για τις εντολές διακλάδωσης υπό συνθήκη, το σύστημα χρησιμοποιεί έναν (2,2) global history predictor με συνολικά 8 entries, τα οποία φαίνονται στον παρακάτω πίνακα. Ο BHR έχει τιμή ίση με 1. Δίνεται επίσης το FSM διάγραμμα του 2-bit predictor, ο οποίος προβλέπει T για τιμές ≥ 2 και NT για τις υπόλοιπες.

	0 = 00	**1 = 01**	**2 = 10**	**3 = 11**
0	00	00	01	01
1	01	~~11~~ 10	~~10~~, 01	00

$$00 \underset{N}{\overset{T}{\rightleftarrows}} 01 \underset{N}{\overset{T}{\rightleftarrows}} 10 \underset{N}{\overset{T}{\rightleftarrows}} 11$$

- Η δεικτοδότηση του πίνακα γίνεται χρησιμοποιώντας τον BHR καθώς και τον κατάλληλο αριθμό low order bits από το PC της εντολής. Ο επεξεργαστής υλοποιεί το ISA του MIPS.

- Το σύστημα περιλαμβάνει μια fully associative cache μεγέθους 32B με block size 16 bytes και πολιτική αντικατάστασης LRU. Αρχικά, η cache είναι άδεια.
- Οι καταχωρητές R1, R2 περιέχουν τις διευθύνσεις των $1^{ων}$ στοιχείων των πινάκων A και B αντίστοιχα, στους οποίους έχουν αποθηκευτεί αριθμοί διπλής ακρίβειας (μήκους 8B ο καθένας).
- Οι πίνακες είναι ευθυγραμμισμένοι.
- (Έβαλα τον κώδικα παρακάτω)

Λύση

- IS, WR, CMT: 1 cycle
- ROB: 10 θέσεις
- 1 CBD:
  - Προτεραιότητα η παλαιότερη εντολή.
  - Το branches δεν χρησιμοποιούν το CDB στο WR.
- Πρόβλεψη διακλάδωσης ταυτόχρονη με δρομολόγηση εντολής.
- Έλεγχος πρόβλεψης: WR
- Δρομολόγηση σωστής εντολής: WR+1
- BHR = ~~01~~, ~~10~~, 01
- R8 = ~~1~~, ~~0~~, ~~1~~, 0
- Cache: Fully Associative
  - Μέγεθος: 32 B
  - Block Size: 16 B
- Πολιτική Αντικατάστασης:
  - R1 → A[0] → 8 bytes
  - R2 → B[0] → 8 bytes

	RS	**FU**	**Cycles**
ADDD/SUBD	3	1	3
MULD/DIVD	2	1	5
BR/ADD/SUB/AND	3	3	2
MUL/DIV	1	3	2
LOAD	2		H:1, M: 4
STORE	2		H:1, M: 4

- IS: Χώρος στο FU + Χώρος στο RS
- EX: Χώρος στο FU + Εξαρτήσεις
- WR: Χώρος στο CBD
- CMT: Μετά από το προηγούμενο

	8 bytes	**8 bytes**
0	~~A[0]~~ B[2]	~~A[1]~~ B[3]
1	~~B[0]~~ ~~B[2]~~ B[0]	~~B[1]~~ ~~B[3]~~ B[1]

C = 1**(1)**00

Εντολή	IS	EX	WR	CMT	Σχόλια
LD F0, 0(R1)	1	2-5	6	7	Cache Miss, LRU = 1
ADDD F4, F4, F0	2	7-9	10	11	RAW F0, (WAR F4)
LD F1, 0(R2)	3	4-7	8	12	Cache Miss, LRU = 0
MULD F4, F4, F1	4	11-15	16	17	RAW F1, F4, (WAR F4)
ANDI R9, R8, 0x2	5	6-7	9	18	CDB Conflict
BNEZ R9, NEXT	6	10-11	12	19	RAW R9, Pred = 1, Res = NT
X *LD F5, 8(R1)*	7	8	11	X	Cache Hit, CDB Conflict, LRU = 1
X *ADDD F4, F4, F5*	8	12-X	X	X	RAW F5, F4, (WAR F4)
X *ADDI R1, R1, 0x8*	9	10-11	12	X	(WAR R1)
X *SUBI R8, R8, 0x1*	10	11-12	X	X	(WAR R8)
X *BNEZ R8, LOOP*	11	12-X	X	X	RAW R8
LD F2, 16(R2)	13	14-17	18	20	Cache Miss, LRU = 0,
MULD F2, F2, F5	14	19-23	24	25	RAW F2, (WAR F2)
ADDD F4, F4, F2	15	25-27	28	29	RAW F2, (WAR F4)
LD F5, 8(R1)	16	17	19	30	Cache Hit, LRU = 1, (WAR F5) , CDB Conflict
ADDD F4, F4, F5	17	29-31	32	33	RAW F4, F5, (WAR F4)
ADDI R1, R1, 0x8	18	19-20	21	34	Cache Hit, LRU = 0
SUBI R8, R8, 0x1	19	20-21	22	35	
BNEZ R8, LOOP	20	23-24	25	36	RAW R8, Pred T, Res NT
X *LD F0, 0(R1)*	21	22	23	37	Cache Hit, LRU = 1
X *ADDD F4, F4, F0*	22	X	X	X	RAW F0, (WAR F4)
X *LD F1, 0 (R2)*	23	24-X	X	X	Cache Miss, LRU = 0
X *MULD F4, F4, F1*	24	X	X	X	ROB Full
SD F4, 16(R2)	26	33-36	37	38	Cache Miss, LRU = 1, RAW F4

Θέμα 4

Θεωρήστε ένα σύστημα παράλληλης επεξεργασίας 2 επεξεργαστών κοινής μνήμης, το οποίο χρησιμοποιεί το πρωτόκολλο συνάφειας MESI επιτρέποντας cache-to-cache transfers και υλοποιεί το μοντέλο της ακολουθιακής συνέπειας. Κάθε επεξεργαστής διαθέτει μία μόνο L1 data cache μεγέθους 32B, direct-mapped, write-back αποτελούμενη από 4 blocks. Αρχικά, όλα τα blocks σε όλες τις caches είναι σε κατάσταση 1. Στους επεξεργαστές εκτελείται παράλληλα το εξής C πρόγραμμα:

Λύση

	P_1	**P_2**	**Cycles**
	IIII	IIII	
RD x[0].a	EIII	IIII	2+8
RD x[0].c	EIII	IEII	2+8
RD x[0].b	EIII	IEII	1
RD x[0].d	EIII	IEII	1
WR x[0].a	MIII	IEII	1
WR x[0].c	MIII	IMII	1
RD x[1].a	MSII	ISII	2+8
RD x[1].c	MSII	ISEI	2+8
RD x[1].b	MSSI	ISSI	2+8
RD x[1].d	MSSI	ISSI	1
WR x[1].a	MMSI	IISI	1+1
WR x[1].c	MMII	IIMI	1+1
RD x[2].a	MMIE	IIMI	2+8
RD x[2].c	MMIE	EIMI	2+8
RD x[2].b	MMIE	EIMI	1
RD x[2].d	MMIE	EIMI	1
WR x[2].a	MMIM	EIMI	1
WR x[2].c	MMIM	MIMI	1
RD x[3].a	SMIM	SIMI	2+8+8
RD x[3].c	SMIM	SEMI	2+8
RD x[3].b	SSIM	SSMI	2+8+8
RD x[3].d	SSIM	SSMI	1
WR x[3].a	MSIM	ISMI	1+1
WR x[3].c	MIIM	IMMI	1+1
RD x[4].a	MIEM	IMMI	2+8
RD x[4].c	MIEM	IMME	2+8
RD x[4].b	MIEM	IMME	1
RD x[4].d	MIEM	IMME	1
WR x[4].a	MIMM	IMME	1
WR x[4].c	MIMM	IMMM	1
RD x[5].a	MIMS	IMMS	2+8+8
RD x[5].c	MIMS	EMMS	2+8
RD x[5].b	SIMS	SMMS	2+8+8
RD x[5].d	SIMS	SMMS	1
WR x[5].a	SIMM	SMMI	1+1
WR x[5].c	IIMM	MMMI	1+1

Total			**209**

B0	x[0].a x[0].b
B1	x[0].c x[0].d x[1].a
B2	x[1].b x[1].c x[1].d
B3	x[2].a x[2].b
B0	x[2].c x[2].d x[3].a
B1	x[3].b x[3].c x[3].d
B2	x[4].a x[4].b
B3	x[4].c x[4].d x[5].a
B0	x[5].b x[5].c x[5].d

Block	L1	L2
0	~~x[0].a x[0].b~~ ~~x[2].c x[2].d x[3].a~~ x[5].b x[5].c x[5].d	~~x[2].c x[2].d x[3].a~~ x[5].b x[5].c x[5].d
1	~~x[0].c x[0].d x[1].a~~ x[3].b x[3].c x[3].d	~~x[0].c x[0].d x[1].a~~ x[3].b x[3].c x[3].d
2	~~x[1].b x[1].c x[1].d~~ x[4].a x[4].b	x[1].b x[1].c x[1].d
3	~~x[2].a x[2].b~~ x[4].c x[4].d x[5].a	x[4].c x[4].d x[5].a

Κανονική 21

Θέμα 3

- IS, WR, CMT: 1 cycle
- ROB: 8 θέσεις
- 1 CBD:
  - Προτεραιότητα η παλαιότερη εντολή.
  - Το branches δεν χρησιμοποιούν το CDB στο WR.
- Πρόβλεψη διακλάδωσης ταυτόχρονη με δρομολόγηση εντολής.
- Έλεγχος πρόβλεψης: WR
- Δρομολόγηση σωστής εντολής: WR+1
- BHR (T, >=2) = 1,
- R8 = 1
- Cache: Fully Associative
  - Μέγεθος: 32 B
  - Block Size: 16 B
- Πολιτική Αντικατάστασης:
  - R1 → A[0] → 8 bytes
  - R2 → B[0] → 8 bytes

	RS	**FU**	**Cycles**
ADDD/SUBD	3	1	3
MULD/DIVD	2	2	7
BR/ADD/SUB/AND	3	3	1
MUL/DIV	1	3	1
LOAD	2		H:1, M: 3
STORE	2		H:1, M: 3

- IS: Χώρος στη ROB + Χώρος στο RS
- EX: Χώρος στο FU + Εξαρτήσεις
- WR: Χώρος στο CBD
- CMT: Μετά από το προηγούμενο

	00	**01**
0	~~10~~ 01	01
1	00	~~11~~ 10

	8 bytes	**8 bytes**
0	~~A[0]~~ A[2]	~~A[1]~~ A[3]
1	~~B[2]~~ B[0]	~~B[3]~~ B[1]

Εντολή	**IS**	**EX**	**WR**	**CMT**	**Σχόλια**
LD F0, 0(R1)	1	2-4	5	6	Miss, LRU=1
ADDD F4, F4, F0	2	6-8	9	10	RAW F0, (WAR F4)
LD F1, 16(R2)	3	4-6	7	11	Miss, LRU=0

MULD F4, F4, F1	4	10-16	17	18	RAW F1, F4, (WAR F4)
ANDI R9, R8, 0x2	5	6	8	19	R8=1, R9=0, CDB Conflict
BNEZ R9, NEXT	6	9	10	20	RAW R9, Pred T, Res NT, (4_{16} = 0**1**00, BHR = 1 => 11), BHR=0
X *LD F5, 16(R1)*	7	8-10	X	X	Miss A[2], LRU=1, CDB Conflict
X *ADDD F4, F4, F5*	8	X	X	X	RAW F4, F5, (WAR F4)
X *ADDI R1, R1, 0x8*	9	10	X	X	R1=A[1]
					ROB Full, Flush
X *SUBI R8, R8, 0x1*	X	X	X	X	
LD F2, 16(R2)	11	12	13	21	Hit B[3], LRU=0
MULD F4, F2, F5	12	13-19	20	22	RAW F2
LD F5, 16(R1)	13	14	15	23	Hit A[3], LRU=1, (WAR F5)
ADDD F4, F4, F5	14	21-23	24	25	RAW F4, F5, (WAR F4)
ADDI R1, R1, 0x8	15	16	18	26	R1=A[2]
SUBI R8, R8, 0x1	19	20	21	27	R8=0, ROB Full
BNEZ R8, LOOP	20	22	23	28	RAW R8, Pred T, Res NT (0_{16}=0, BHR=0 => 10), BHR=1
X *LD F0, 0(R1)*	21	22	X	X	Hit A[2], LRU=1
X *ADDD F4, F4, F0*	22	X	X	X	RAW F0, (WAR F4)
X *LD F1, 0 (R2)*	23	X	X	X	Flush
SD F4, 8(R2)	24	25-27	28	29	Miss B[1], LRU=0