ΚΕΦΑΛΑΙΟ 2

12)Ποιες είναι οι διαφορετικές κατηγορίες αρχιτεκτονικής Instruction set και τα χαρακτηριστικά τους;

α) **stack architecture** (όλα τα operands μπαίνουν στην κορυφή μιας στοίβας)(+)είναι απλό μοντέλο εκτίμησης έκφρασης και οι μικρές εντολές μπορούν να παράγουν πολύ καλό κώδικα (πυκνότητα). (-)Δεν υπάρχει random access, (-)δυσκολία στην παραγωγή αποδοτικού κώδικα και δυσκολία στην αποδοτική λειτουργία λόγω συνωστισμού.

β) **accumulator architecture** (το implicit operand είναι ο accumulator) (+)ελαχιστοποιεί την εσωτερική κατάσταση της μηχανής και έχει μικρές εντολές. (-)έχουμε μεγαλύτερο traffic γιατί ο acc είναι μόνο για προσωρινή αποθήκευση.

γ) **general purpose register architecture** : (+)Τα περισσότερα γενικά μοντέλα για παραγωγή κώδικα (-)όλα τα operands πρέπει να έχουν όνομα άρα έχω μεγαλύτερες εντολές

register – memory: (1,2) (1 memory operant και 2 total operands) (+) τα δεδομένα μπορούν να προσπελαστούν χωρίς να χρειάζεται προηγουμένως φόρτωση της εντολής και η δομή των εντολών μπορεί να κωδικοποιηθεί εύκολα. (-) Οι τελεστές δεν είναι ίσοι. (-) Κωδικοποιώντας ένα καταχωρητή και μια διεύθυνση μνήμης για κάθε εντολή μπορεί να έχει ως αποτέλεσμα την μείωση των καταχωρητών. (-) Ο χρόνος ανά εντολή ποικίλει.

register – register: (0,3) (+) απλή και σταθερού μήκους κωδικοποίηση εντολών. Οι εντολές χρειάζονται περίπου τον ίδιο χρόνο για να εκτελεστούν (+) δημιουργία απλού κώδικα.(-) Περισσότερες γραμμές κώδικα.

memory – memory: (κρατά όλα τα operands στην κύρια μνήμη) (2,2) ή (3,3) (+)παράγει συμπιεσμένο κώδικα και δεν σπαταλάει καταχωρητές για προσωρινή αποθήκευση βοηθώντας έτσι το έργο των compiler (+) το instruction set έχει καλή πυκνότητα (-) μεγάλες διακυμάνσεις στο μέγεθος εντολών κυρίως για εντολές με 3 έντελα. (-)η πολλές προσπελάσεις της μνήμης δημιουργούν συμφόρηση με αποτέλεσμα μειωμένη απόδοση του συστήματος (-)Μεγάλες διακυμάνσεις στην εργασία που απαιτείται για κάθε εντολή → έχει καταργηθεί.

Σήμερα χρησιμοποιείται το RR γιατί οι καταχωρητές είναι γρηγορότεροι από την ΚΜ και οι compilers μπορούν να τους χρησιμοποιούν πιο εύκολα και επίσης γιατί έχει μειωθεί σημαντικά η τιμή του υλικού.

13)Τρόποι διευθυνσιοδότησης της μνήμης

(α)Άμεσος (β1)Κατευθείαν τρόπος διευθυνσιοδότησης θέσης μνήμης (β2)Κατευθείαν τρόπος διευθυνσιοδότησης καταχωρητή (γ1)έμμεσος τρόπος διευθυνσιοδότησης με χρήση καταχωρητή (γ2)έμμεσος τρόπος διευθυνσιοδότησης με χρήση κύριας μνήμης.

14) Τρόποι διευθυνσιοδότησης για DSP

(α)modulo ή circular addressing mode→περίπτωση συνεχούς ροής δεδομένων που χρησιμοποιούν circular buffers (β) bit reverse addressing mode

15)Ποιες οι εντολές επεξεργασίας πολυμέσων

(α) Single Instruction Multiple Data Instructions- εντολές οι οποίες μπορούν να ενεργήσουν σε πολλά data items ταυτόχρονα. (β) paired single operations – με μια εντολή εκτελούμε ταυτόχρονα δύο floating Point 32-bit πράξεις. (γ) MAC εντολές- για υπολογισμό του εσωτερικού γινομένου

16)Εντολές ροής ελέγχου: Εντολές αλλαγής της ροής του προγράμματος άνευ συνθήκης και εντολές αλλαγής της ροής του προγράμματος υπό συνθήκη. Συγκεκριμένα διακρίνονται σε Conditional Branches, Jumps, Procedure Calls/Returns Για τη διεύθυνση προορισμού είτε γίνεται πρόσθεση στον PC είτε αυτή περιέχεται σε κάποιο καταχωρητή

17)Κωδικοποίηση του συνόλου εντολών (Instruction set): Το πώς θα κωδικοποιηθεί στην εντολή η πληροφορία που αφορά τον τρόπο διευθυνσιοδότησης, εξαρτάται από το πλήθος των διαφορετικών τρόπων διευθυνσιοδότησης και το βαθμό εξάρτησης των τρόπων διευθυνσιοδότησης από μια συγκεκριμένη λειτουργία. Στόχοι του σχεδιαστή για το instruction set είναι να υπάρχουν πολλοί καταχωρητές και πολλοί τρόποι διευθυνσιοδότησης, μικρό μέσο μήκος εντολής και απλότητα κωδικοποίησης για εύκολη διαχείριση από pipelined αρχιτεκτονικές. Οι 3 βασικοί τρόποι κωδικοποίησης του Instruction Set είναι: (α)variable length, (b)fixed length (c)hybrid

18) Μεταγλωττιστές: **Στόχοι** των μεταγλωττιστών είναι: (α)ορθότητα, (β)ταχύτητα του μεταγλωττισμένου κώδικα, (γ)γρήγορη μεταγλώττιση, debug support
Λειτουργία: Οι μεταγλωττιστές συνήθως πραγματοποιούν από 2-4 περάσματα. Οι καλύτερα optimized αλγόριθμοι κάνουν περισσότερα περάσματα. Αυτή η δομή αυξάνει την πιθανότητα ότι ένα πρόγραμμα που μεταγλωττίζεται σε διάφορα επίπεδα θα παράξει την ίδια έξοδο για την ίδια είσοδο. Αν απαιτούμε ταχύτητα και είναι ανεκτός χαμηλής ποιότητας κώδικας, μπορούμε να παραλείψουμε περάσματα. Ένα πέρασμα είναι απλώς μια φάση στην οποία ο μεταγλωττιστής διαβάζει και μετασχηματίζει όλο το πρόγραμμα. Εφαρμόζοντας κατάλληλα optimizations στη μεταγλώττιση αυξάνεται κατακόρυφα η απόδοση.

ΚΕΦΑΛΑΙΟ 3

19)ILP – Dependences: Instruction Level Parallelism- (ILP) έχουμε επικάλυψη εντολών στοχεύοντας στην παράλληλη εκτέλεση τους. Με το να βρούμε κατά πόσο μια εντολή εξαρτάται από μια άλλη είναι ουσιώδες για να αποφασίσουμε πόσος παραλληλισμός υπάρχει σε ένα πρόγραμμα και πως ο παραλληλισμός αυτός μπορεί να επιτευχθεί. Αν δυο εντολές είναι παράλληλες τότε μπορούν να εκτελεστούν παράλληλα σε μια pipeline χωρίς να προκαλούν stalls, ενώ όταν δυο εντολές είναι dependent τότε δεν είναι παράλληλες και πρέπει να εκτελεστούν με σειρά, αν και μπορούν να είναι και μερικώς επικαλυπτόμενες. Τα dependences είναι ιδιότητα ενός προγράμματος και χωρίζονται σε 3 κατηγορίες.

(1)<u>Data dependences</u>- Μια εντολή για να εκτελεστεί χρειάζεται το αποτέλεσμα μιας άλλης. Μπορούν να προκαλέσουν RAW hazards.Η σημασία τους είναι ότι ένα dependence (α)υποδεικνύει την πιθανότητα ενός hazard (β)αποφασίζει τη σειρά με την οποία τα δεδομένα πρέπει να υπολογίζονται (γ)θέτει ένα όριο στο επίπεδο του παραλληλισμού που μπορεί να επιτευχθεί.

(2)<u>Name dependences</u>: Δυο εντολές χρησιμοποιούν το ίδιο όνομα αλλά δεν ανταλλάζουν δεδομένα. Μπορούν να προκαλέσουν WAR και WAW hazards.Αν η εντολή i προηγείται της εντολής j υπάρχουν 2 τύποι name dependence.
(a)antidependence- προκύπτει όταν η εντολή j γράφει σε ένα καταχωρητή ή μια διεύθυνση μνήμης που η εντολή i διαβάζει. Η κανονική σειρά εκτέλεσης των εντολών πρέπει να τηρηθεί έτσι ώστε η εντολή i να διαβάζει τη σωστή τιμή. (b)Output dependences- Η εντολή i και η εντολή j γράφουν στον ίδιο καταχωρητή ή στην ίδια διεύθυνση μνήμης. Η σειρά μεταξύ των εντολών πρέπει να τηρηθεί έτσι ώστε η τιμή που αποθηκεύεται τελικά να αντιστοιχεί στην εντολή j.
Λύση μπορεί να δοθεί και οι εντολές να εκτελούνται παράλληλα αν αλλάξουμε των καταχωρητή ή τη διεύθυνση μνήμης σε μια από τις δυο εντολές και αυτό μπορεί να γίνει είτε στατικά από τον compiler είτε δυναμικά από το HW.

(3)<u>Control dependences:</u> Αποφασίζει τη σειρά μιας εντολής, i, σε σχέση με μια εντολή διακλάδωσης, έτσι ώστε η εντολή i να εκτελεστεί σωστά και μόνο όταν πρέπει.

<u>Διευκρίνηση:</u> Ένα hazard δημιουργείται όταν υπάρχει dependence μεταξύ εντολών και υπάρχει κίνδυνος η επικάλυψη που γίνεται από το pipeline θα αλλάζει τη σειρά προσπέλασης των τελεστών που εμπλέκονται στο dependence. Λόγω του dependence πρέπει να διασφαλίσουμε αυτό που καλούμε program order.

20)Ποια η σχέση των dependencies με τα hazards ; Ονοματίστε και συσχετίστε τα είδη των data hazards με τα true, anti & output dependencies αντίστοιχα :
Ένα hazard δημιουργείται κάθε φορά που υπάρχει μια εξάρτηση μεταξύ εντολών, και αυτές είναι αρκετά κοντά έτσι ώστε η πιθανή επικάλυψη που προκαλείται από το pipeline, ή από κάποια άλλη επαναδιάταξη των εντολών, να αλλάζει τη σειρά της προσπέλασης στην τιμή που εμπλέκεται με την εξάρτηση.
Τα data hazards μπορούν σε ταξινομηθούν σε 3 κατηγορίες, ανάλογα με τη σειρά των προσπελάσεων για διάβασμα ή εγγραφή στις εντολές. Θεωρούμε 2 εντολές i και j όπου η i προηγείται στη σειρά του προγράμματος. Τα πιθανά data hazards είναι :
RAW (read after write) : η j προσπαθεί να διαβάσει μια τιμή πριν η i τη γράψει. Δηλαδή η j διαβάζει λάθος τιμή. Αυτός ο τύπος hazard είναι ανάλογος με μια true dependence.
WAW (write after write) : η j προσπαθεί να γράψει μια τιμή πριν η i γράψει στην ίδια θέση. Δηλαδή η τελική τιμή είναι λανθασμένα αυτή που γράφει η i. Αυτός ο τύπος hazard είναι ανάλογος με μια output dependence.

WAR (write after read) : η j προσπαθεί να γράψει μια τιμή πρίν η i διαβάσει από την ίδια θέση. Δηλαδή η i λαμβάνει λάθος τιμή. Αυτός ο τύπος hazard είναι ανάλογος με μια anti dependence.

Hazards: Καταστάσεις οι οποίες εμποδίζουν την εκτέλεση της επόμενης σε σειρά εντολής, στην ακολουθία εντολών ενός προγράμματος, κατά τη διάρκεια του προσχεδιασμένου κύκλου ρολογιού της. Τα hazards μειώνουν την ιδανική απόδοση που μπορεί να επιτύχουμε με το speedup. 3 κατηγορίες hazards.
(a)structural hazards: Εμφανίζονται από resource conflicts (σύγκρουση στα δεδομένα εισόδου) όταν το HW δεν μπορεί να υποστηρίξει όλους τους δυνατούς συνδυασμούς εντολών ταυτόχρονα σε επικαλυπτόμενη εκτέλεση.
(β)data hazards: εμφανίζονται όταν μια εντολή εξαρτάται από το αποτέλεσμα μιας προηγούμενης, με ένα τρόπο που καθορίζεται από την επικάλυψη εντολών στο pipeline. Τρόπος μείωσης: forwarding(1.ειδικό HW περνάει τα αποτελέσματα της ALU πίσω στην είσοδο της, 2.δεν επιτρέπει την χρήση παλιών τιμών).Ταξινομούνται σε RAW, WAW, WAR.
(γ)flow hazards: Εμφανίζονται στις διακλαδώσεις (branch, jump). Τρόποι μείωσης των χαμένων κύκλων από branch: 1.flush, 2.predict-not-taken, 3.predict-taken, 4.delayed branch.

21)Τι είναι το Dynamic Scheduling και πια η βασική ιδέα υλοποίησης του;
Dynamic Scheduling είναι η διαδικασία κατά την οποία το HW αναδιοργανώνει την εκτέλεση των εντολών με στόχο τη μείωση των stalls διατηρώντας τη ροή των δεδομένων και την συμπεριφορά των exceptions. (+)μπορούμε να αντιμετωπίσουμε καταστάσεις όπου τα dependences δεν είναι γνωστά σε compile time(+)Απλοποιείται ο compiler (+)κώδικας που έχει μεταγλωττιστεί σε συγκεκριμένο pipeline μπορεί να τρέξει και σε διαφορετικό pipeline.
Η βασική ιδέα υλοποίησης του έχει ως εξής: (1)Χωρίζουμε την αποκωδικοποίηση των εντολών σε 2 στάδια (α)issue: Γίνεται αποκωδικοποίηση και έλεγχος για structural hazards (β)read operands: Έλεγχος για data hazards και διάβασμα τελεστών. (2)ακολουθεί το στάδιο της αποκωδικοποίησης όπου υπάρχει ουρά εντολών και οι εντολές πριν εκτελεστούν πρέπει να εξασφαλιστεί ότι δεν υπάρχουν hazards. Έπειτα ακολουθεί in-order issue και out-of-order execution και completion.

22)Dynamic Scheduling με Tomasulo approach: Η προσέγγιση αυτή προσπαθεί να αυξήσει το ILP. Υλοποιήθηκε για IBM 360 αρχιτεκτονική, δεν χρησιμοποιεί cache και έχει ελάχιστους καταχωρητές. Χρησιμοποιεί control & buffers κατανεμημένα με Function Units (FU). Αυτά τα FU ονομάζονται reservation stations (RS). Η μετονομασία παρέχεται από τα RS που έχουν τα pending operands. Η βασική ιδέα είναι ότι τα RS παίρνουν και αποθηκεύουν ένα operand μόλις αυτό είναι διαθέσιμο εξαλείφοντας έτσι την ανάγκη να πάρει το operand από τον καταχωρητή. Στη συνέχει οι καταχωρητές στις εντολές αντικαθίστανται με τιμές η δείκτες στα RS πετυχαίνοντας έτσι register renaming. Αφού μπορούμε να έχουμε περισσότερα RS από καταχωρητές, η τεχνική μπορεί να εξαλείψει hazards από name dependences που δεν μπορούν να εξαλειφθούν από ένα compiler. Η ανακάλυψη hazards και ο έλεγχος λειτουργίας είναι κατανεμημένα και αυτό γιατί η πληροφορία που κρατείται στα RS σε κάθε FU αποφασίζει πότε μια εντολή μπορεί να εκτελεστεί στη μονάδα. Τα αποτελέσματα μεταφέρονται απευθείας στα FU από τα RS όπου και αποθηκεύονται χωρίς τη χρήση καταχωρητών. Αυτό γίνεται με τη χρήση του Common Data Bus (CDB) που στέλνει τα αποτελέσματα στις FUs. Τα Load και Store χρειάζονται 2 βήματα στη διαδικασία εκτέλεσης. Έτσι θεωρούνται ως FUs με RSs.

23)Περιγραφή του Tomasulo αλγόριθμου και επισκόπηση.

Ο αλγόριθμος Tomasulo αποτελείται από 3 στάδια:

(1)issue: Παίρνει την εντολή από την κορυφή της ουράς με τις εντολές. Αν το RS είναι ελεύθερο (δεν υπάρχει δηλαδή structural hazard) ξεκινά την εντολή και στέλνει τα operands. Σε αυτό το στάδιο γίνεται μετονομασία αποφεύγοντας WAR και WAW hazards. Αν το RS δεν είναι ελεύθερο, υπάρχει structural hazard και η εντολή stalls μέχρι ένα RS ή ένα buffer ελευθερωθεί.

(2)execution: Όταν όλα τα operands είναι διαθέσιμα γίνεται εκτέλεση της εντολής. Ν δεν είναι διαθέσιμα, τότε έχουμε αναμονή στο CDB για δεδομένα. Με αυτό τον τρόπο αποφεύγουμε τα RAW hazards.

(3)Write result: Όταν το αποτέλεσμα είναι διαθέσιμο το στέλνουμε στο CDB προς όλες τις μονάδες που περιμένουν και το RS σημειώνεται ως ελεύθερο.

Επισκόπηση: Ο αλγόριθμος αυτός εντοπίζει πότε είναι διαθέσιμα τα operands για τις εντολές για ελαχιστοποίηση των RAW hazards. Κάνει μετονομασία καταχωρητών για να ελαχιστοποιήσει τα WAW και RAW hazards. Επιτρέπει Loop unrolling σε HW. Δεν περιορίζεται σε basic blocks (εφόσον υπάρχει branch prediction). Συνεισφέρει σε dynamic Scheduling, register renaming και load/store disambiguation.

24)Τι είναι το Branch Prediction Buffer (ή Branch History) και πιο το πρόβλημα του 1-bit BHT;

Είναι μια μικρή μνήμη, ουσιαστικά είναι τα τελευταία bits του PC address index. Αυτή η μικρή μνήμη περιέχει ένα bit που δηλώνει κατά πόσο το branch έχει εκτελεστεί ή όχι, χωρίς όμως να κάνει έλεγχο διευθύνσεων.

Το πρόβλημα του 1bit BHT είναι ότι σε ένα loop θα προκαλέσει 2 μη σωστές προβλέψεις (1 στο τέλος του loop όταν βγαίνει αντί να συνεχίσει την επανάληψη όπως προηγουμένως και 1 στην πρώτη φορά του loop, στην επόμενη εκτέλεση προβλέπει έξοδο και όχι επανάληψη). Λύση με 2-bit prediction όπου αλλάζει η πρόβλεψη μόνο μετά από 2 αποτυχημένες προβλέψεις (-)χρειάζεται παράλληλα και read και write port και το update γίνεται πιο συχνά από το 1bit. Για να πετύχουμε καλύτερη πρόβλεψη απ΄ αυτή που προκύπτει από το απλό 2bit BHT χρησιμοποιούμε ***συσχετισμό των branches.*** Υποθέτουμε ότι τα πρόσφατα branches σχετίζονται, δηλαδή η συμπεριφορά των πρόσφατα εκτελεσθέντων branches επηρεάζει την πρόβλεψη του τρέχοντος branch. Η όλη ιδέα στηρίζεται στην καταγραφή των m πιο πρόσφατα εκτελεσθέντων branches ως taken ή not taken και χρήση του pattern αυτού για την επιλογή του κατάλληλου branch history table. Γενικά, (m,n) predictor σημαίνει καταγραφή των τελευταίων m branches για επιλογή ανάμεσα σε 2^m history tables με n-bit counter το καθένα. (άρα 2-bit BHT είναι ένας (0,2) predictor). Για (2,2) predictor, η συμπεριφορά των πρόσφατων branches επιλέγει μεταξύ 4 προβλέψεων για το επόμενο branch, ενημερώνοντας απλά αυτή την πρόβλεψη.

Γίνεται χρήση **branch target buffer** που είναι branch prediction cache που αποθηκεύει τις προβλεπόμενες διευθύνσεις για την εντολή μετά το branch. Η predicated execution μπορεί να μειώσει τον αριθμό των branches και τον αριθμό των μη επιτυχώς προβλεφθέντων branches.

25)Μπορούμε να πετύχουμε περισσότερο ILP με multiple issue

Στόχος μας σε κάθε περίπτωση είναι να επιτύχουμε CPI μικρότερο της μονάδας.

(i)Superscalar: έχουμε περισσότερες από μια εντολές ανά κύκλο ρολογιού και αποφασίζουν on the fly πόσες εντολές να εκδώσουν.

(a)static: Ο compiler αποφασίζει τη σειρά που θα εκτελεστούν οι εντολές. Πρέπει να ελέγχει για dependences τόσο μεταξύ εντολών που έχουν εκδοθεί, όσο και εντολών που είναι υποψήφιες προς έκδοση, όσο και εντολών που βρίσκονται ήδη στο pipeline. Χρειάζεται την αποκλειστική βοήθεια του compiler για να πετύχει καλά απόδοση.

(b)dynamic: το HW προσπαθεί να εκτελέσει πολλαπλές εντολές ανά κύκλο ρολογιού με τη χρήση αλγορίθμου Tomasulo. Χρειάζονται λιγότερη βοήθεια από τον compiler αλλά έχουν σημαντικό HW κόστος.

(ii)Very Long Instruction Word (VLIW): Εκδίδουν σταθερό αριθμό εντολών. Long Instructions που περιέχουν πολλές λειτουργίες και ο compiler προγραμματίζει τις λειτουργίες σε αυτές τις εντολές.

26)Υποστήριξη HW για περισσότερο ILP

Υπάρχουν διάφοροι μηχανισμοί για την υποστήριξη του speculation από τον compiler και ένας τέτοιος είναι το HW speculation το οποίο επεκτείνει την ιδέα του dynamic scheduling.

Η αντιμετώπιση του control dependence γίνεται υποθέτοντας τις εξόδους των branches και εκτελούμε την εντολή χωρίς συνέπειες. Πρέπει όμως να υπάρχουν και μηχανισμοί που να μπορούμε να χειριστούμε το γεγονός ότι η υπόθεση μας μπορεί να ήταν λανθασμένη.(boosting)

Το HW based speculation συνδυάζει 3 ιδέες.

(α)dynamic branch prediction για να διαλέξουμε ποιες εντολές να εκτελέσουμε.

(β)speculation: για να επιτρέψουμε την εκτέλεση των εντολών πριν να αποφασιστούν

(γ)dynamic scheduling

Το HW που εισάγει τον Tomasulo αλγόριθμο μπορεί να εμπλουτιστεί έτσι ώστε να υποστηρίζει speculation. Για να γίνει αυτό πρέπει να διαχωρίσουμε τα speculative bypassing αποτελέσματα από τα real bypassing αποτελέσματα. Κάνοντας αυτό το διαχωρισμό μπορούμε να επιστρέψουμε σε μια εντολή να εκτελεστεί και μετά να κάνουμε bypass τα αποτελέσματα της σε άλλες εντολές χωρίς να επιτρέπουμε στην εντολή να κάνει οτιδήποτε updates που δεν μπορούν να αναιρεθούν, μέχρι να ξέρουμε ότι η εντολή δεν είναι πλέον speculative. Όταν η εντολή δεν είναι πια speculative, αποθηκεύουμε τα boosted αποτελέσματα (instruction commit).

Το κλειδί στην όλη ιδέα είναι ότι επιτρέπουμε στις εντολές να εκτελούνται out-of-order αλλά τις αναγκάζουμε να commit-in-order προς αποφυγή αμετάκλητης ενέργειας μέχρι μια εντολή commits.

Για να γίνουν τα πιο πάνω απαιτούνται HW buffers για την αποθήκευση των αποτελεσμάτων των uncommitted εντολών (reorder buffers-ROB). Τα ROB είναι πηγή operands για τις εντολές, (όπως τα RS περιείχαν τα operands στον Tomasulo αλγόριθμο). Σε αυτή την περίπτωση γίνεται χρήση των ROB αντί των RS όταν η εκτέλεση ολοκληρώνεται και η προμήθεια των operands γίνεται ανάμεσα στη φάση που η εντολή έχει εκτελεστεί αλλά δεν έχει γίνει ακόμη commit. Όταν ένα operand commits τότε το αποτέλεσμα αποθηκεύεται σε καταχωρητή. Κάθε καταχωρητής στο ROB περιέχει 3 πεδία: (a)Instruction Type-δηλώνει κατά πόσο η εντολή είναι branch, a store, ή a register operation, (b) destination- περιέχει ή τον αριθμό του καταχωρητή ή τη διεύθυνση μνήμης που το αποτέλεσμα της εντολής πρέπει να γραφτεί και (c)value: χρησιμεύει για να κρατάει την τιμή που επιστρέφει η εντολή μέχρι αυτή να γίνει commit.

27)Ποια τα 4 στάδια του speculative Tomasulo αλγόριθμου;
(1)issue: Παίρνει την εντολή από την κορυφή της ουράς με τις εντολές. Αν υπάρχει ελεύθερο RS και κενό ROB slot, τότε issue the instruction, στείλε τα operands, κάνε update τα control entries έτσι ώστε να δηλώνουμε ότι οι buffers είναι σε χρήση. Επίσης στο RS στέλνεται ο αριθμός του ROB που δεσμεύεται για το αποτέλεσμα. Αν είτε τα RS είτε η ROB είναι γεμάτα the instruction issue is stall μέχρι να ελευθερωθούν θέσεις και στα δύο.
(2)execution: Όταν όλα τα operands είναι διαθέσιμα γίνεται εκτέλεση της εντολής. Αν δεν είναι διαθέσιμα, τότε έχουμε αναμονή στο CDB για δεδομένα. Αν τα operands είναι διαθέσιμα σε RS τότε έχω εκτέλεση. Με αυτό τον τρόπο αποφεύγουμε τα RAW hazards.
(3)Write result: Όταν το αποτέλεσμα είναι διαθέσιμο το στέλνουμε στο CDB και απ΄ εκεί στους ROB και στα FU που περιέχουν το αποτέλεσμα. Το RS σημειώνεται ως ελεύθερο.
(4)Commit: Ο επεξεργαστής κάνει update τον καταχωρητή με το αποτέλεσμα. Όταν η εντολή βρίσκεται στην κορυφή του ROB και τα αποτελέσματα της είναι παρόντα, γίνεται ενημέρωση του register με το αποτέλεσμα και απομακρύνεται η εντολή από το ROB. Αν έχω λάθος πρόβλεψη branch αδειάζει ο ROB και η εκτέλεση ξαναξεκινά.

28)Περιορισμοί στο ILP: Για να ισχύουν όλα τα πιο πάνω υποθέτουμε HW ιδανικό ώστε να καταφέρουμε να κατασκευάσουμε την τέλεια μηχανή.
(1)register renaming: υποθέτουμε απεριόριστους virtual registers και όλα τα WAW & WAR hazards αποφεύγονται.
(2)branch prediction –τέλειο. Δεν έχω δηλαδή λάθος προβλέψεις.
(3)jump prediction: όλα τα jumps προβλέπονται τέλεια άρα έχω μηχανή με τέλειο speculation και απεριόριστο buffer εντολών διαθέσιμο.
(4)memory address alias analysis- οι διευθύνσεις είναι γνωστές και ένα store μπορεί να μετακινηθεί πριν από ένα Load εφόσον οι διευθύνσεις δεν είναι οι ίδιες.
Με το (2)&(3) εξασφαλίζουμε ότι δεν θα υπάρχουν control dependences και με τα (1)&(4) ότι δεν θα υπάρχουν αληθινά data dependences.
Έτσι έχω 1 cycle latency για όλες τις εντολές και απεριόριστο αριθμό εντολών issued per clock cycle.

29)Περιγράψτε την αρχιτεκτονική P6
Περιέχει Dynamically Scheduled επεξεργαστή που μεταφράζει κάθε IA-32 εντολή σε μια σειρά μικρολειτουργιών που εκτελούνται από το pipeline. Έχει μέχρι 3 εντολές IA-32 οι οποίες φορτώνονται, αποκωδικοποιούνται και μεταφράζονται σε μικρολειτουργίες σε κάθε κύκλο ρολογιού. Οι μικρολειτουργίες εκτελούνται από ένα out-of-order speculative pipeline που χρησιμοποιεί register renaming και reorder buffer. Το pipeline αποτελείται από 14 στάδια, 8 στάδια για in-order instruction fetch, decode και dispatch, 3 στάδια για out-of-order εκτέλεση σε κάποιο από τα 5 functional units και 3 στάδια για instruction commit.

ΚΕΦΑΛΑΙΟ 4

30)Εκμετάλλευση pipeline: Για να διατηρούμε ένα pipeline γεμάτο πρέπει να εκμεταλλευόμαστε τον παραλληλισμό μεταξύ των εντολών έτσι ώστε να βρίσκουμε ακολουθίες από μη συσχετισμένες εντολές που μπορεί να επικαλύπτει η μια την άλλη στο pipeline. Για την αποφυγή ενός pipeline stall πρέπει μια εξαρτώμενη εντολή να επέχει από την εντολή πηγή τόσο clock cycles όσα και το pipeline latency αυτής.

31)Που χρησιμεύει το loop unrolling και ποια τα προβλήματα που έχει;
Το loop unrolling χρησιμεύει για να βελτιώσει την δρομολόγηση. Επειδή εξαλείφει τα branches επιτρέπει σε εντολές από διάφορες επαναλήψεις να δρομολογούνται μαζί. Επίσης με τη χρήση του unrolling μειώνονται τα dependences.
Τα προβλήματα του είναι (1)δεν γνωρίζουμε το loop count σε compile time, δηλ δεν ξέρουμε αν το loop count είναι πολλαπλάσιο του αριθμού που χωρίζουμε τον κώδικα. (2)δημιουργεί πολύ μεγάλο κώδικα (3)register pressure-χρειάζονται αρκετοί registers και μπορεί να φτάσουμε στο σημείο να έχουμε έλλειψη από καταχωρητές. Γι αυτό το λόγο πρέπει να γίνει έξυπνη χρήση των registers για να μειωθεί ο αριθμός τους. (4)Είναι δύσκολο για τον compiler να γνωρίζει αν μια μετακίνηση είναι επιτρεπτή.

32)Static Branch Prediction
Υπάρχει ανάγκη στατικής πρόβλεψης των εντολών διακλάδωσης κατά την μεταγλώττιση. Οι μέθοδοι για να γίνει αυτό είναι: (α)Predict as taken (b)Predict on the basis of branch prediction (c)Predict branches on the basis of profile information collected from earlier runs.
Το static branch behavior χρησιμεύει για δρομολόγηση εντολών βοηθώντας τους dynamic predictors και αποφασίζοντας ποια code paths χρησιμοποιούνται πιο συχνά.

33)Περιγραφή του VLIW και πια τα προβλήματα του.
Very Large Instruction Word. Είναι τεχνική με την οποία προσπαθούμε να επιτύχουμε CPI μικρότερο της μονάδας. Χρησιμοποιεί πολλαπλά ανεξάρτητα FU και οι εντολές έχουν μέγεθος 64–128 bits ή και μεγαλύτερο. Σε κάθε εντολή μπορεί να περιλαμβάνονται πολλαπλές λειτουργίες. Είναι τεχνική που στηρίζεται στον compiler όχι μόνο για να ελαχιστοποιήσει τα ουσιώδη data hazards stalls αλλά και για να διατάσει τις εντολές σε ένα βασικό πακέτο εντολών έτσι ώστε το HW να μην χρειάζεται για να ελέγχει dependences. Έτσι έχουμε πιο απλό HW (σε σχέση με τα superscalar) και ταυτόχρονα πετυχαίνουμε καλή απόδοση με extensive compiler optimization. Το VLIW όμως για να λειτουργεί αποδοτικά πρέπει να υπάρχει αρκετή παραλληλία σε μια λίστα κώδικα για να χρησιμοποιούνται όλα τα διαθέσιμα slots λειτουργιών. Επιπρόσθετα χρειάζεται χρήση πιο πολύπλοκου γενικού αλγόριθμου scheduling.
Τα προβλήματα του VLIW είναι τόσο τεχνικά –αυξημένος κώδικας λόγο των unrolling loops και των σπαταλούμενων μη χρησιμοποιημένων bits που προκύπτουν όταν οι εντολές δεν είναι γεμάτες, περιορισμοί σε lockstep operation- ένα stall σε κάποιο FU έχει σαν αποτέλεσμα να μείνει ανενεργός ολόκληρος ο επεξεργαστής και αυτό γιατί όλα τα FUs έπρεπε να μένουν συγχρονισμένα- όσο και λογιστικά- Binary code compatibility όπου ο διαφορετικός αριθμός units και unit latencies απαιτούν διαφορετικό κώδικα.(Λύση σε αυτό το πρόβλημα με object-code translation or emulation).

34)Loop-Level analysis: η διαδικασία για να αποφασίσουμε το είδος των dependences που υπάρχουν μεταξύ των operands ενός loop κατά τις επαναλήψεις αυτού του loop. Προσπαθούμε να επιλύσουμε τα data dependences – ένα operand γράφεται σε ένα σημείο και διαβάζεται σε μεταγενέστερο στάδιο- και αυτό γιατί ενώ υπάρχουν και name dependences αυτές μπορούν να αντιμετωπιστούν με τεχνικές μετονομασίας.

35)Loop-carrier dependences: Κατά πόσο οι προσπελάσεις δεδομένων σε μεταγενέστερες επαναλήψεις εξαρτώνται από τιμές που παράχθηκαν σε προγενέστερη επανάληψη. Αν δεν υπάρχει loop- carrier dependence τότε έχουμε loop-level parallelism.

36)Γιατί είναι σημαντικό να βρούμε τα dependences σε ένα πρόγραμμα, πως τα βρίσκουμε και πως τα ελαχιστοποιούμε;
Είναι σημαντικό να βρούμε τα dependences σε ένα πρόγραμμα γιατί μπορούμε να κάνουμε καλό scheduling του κώδικα, μπορεί να γίνει καθορισμός των loops που μπορούν να παραλληλοποιηθούν, και ελαχιστοποίηση των name dependences. Αυτή η διαδικασία είναι απλή για αναφορές σε μονές μεταβλητές αλλά είναι δύσκολο για pointers arrays κτλ. Η εστίαση για την εύρεση των dependences είναι στα loops όπου γίνεται unroll όσον περισσότερων resources επιτρέπονται καθώς και χρήση διαφορετικών ονομάτων registers για κάθε επανάληψη και επαναπρογραμματισμός για μεγιστοποίηση του παραλληλισμού.

37)Software pipeline: Είναι μια τεχνική παρόμοια με το loop unrolling. Ονομάζεται και symbolic loop unrolling και χρειάζεται λιγότερο κώδικα από το κλασσικό loop unrolling. Είναι τεχνική αναδιοργάνωσης των loop κατά την οποία κάθε επανάληψη αποτελείται από εντολές επιλεγμένες από διαφορετικές επαναλήψεις του αρχικού loop. Σε σχέση με το loop unrolling παράγει πιο συμπαγές κώδικα μεγιστοποιεί την απόσταση result-use. Επιπρόσθετα fill & drain μόνο μια φορά ανά loop σε σχέση με το loop unrolling που χρειάζεται μια φορά για κάθε unrolled επανάληψη. Παρ όλα αυτά το software loop unrolling είναι πολύ δύσκολο να εφαρμοστεί και χρειάζεται υποστήριξη HW

38) Ποιες είναι οι τεχνικές του compiler για αύξηση του ILP
Οι τεχνικές του compiler για αύξηση του ILP είναι:
(1)Loop Unrolling όπου χρησιμοποιούνται διάφορες επαναλήψεις του loop για να βρεθεί η παραλληλία. Σε αυτή την περίπτωση δημιουργούμε ακολουθίες ενιαίου κώδικα.
(2)Software pipelining: Τροποποίηση του loop body για εύρεση της παραλληλίας.
(3)Trace Scheduling: Γίνεται πρόβλεψη των branches και συνδυασμός block.
(4)Speculation: Χρησιμοποιείται υποστήριξη HW για υποθέσεις σχετικά με την κατάληξη του branch.

39)Global Scope Scheduling: Στοχεύει στο να ενσωματώσει ένα μέρος κώδικα σε μια εσωτερική δομή ελέγχου στη συντομότερη πιθανή ακολουθία που διατηρεί τα data και control dependencies. Απαιτεί πολύπλοκα trade-offs για αποφάσεις μετακίνησης κώδικα τα οποία εξαρτώνται από πολλούς παράγοντες.
Το global code scheduling είναι ένα εξαιρετικά πολύπλοκο πρόβλημα και υπάρχουν 2 μέθοδοι που απλοποιούν τη διαδικασία. Και οι 2 μέθοδοι επιλέγουν και δρομολογούν το πιο συχνά χρησιμοποιημένο μονοπάτι.
(1)Trace Scheduling: Είναι ένας τρόπος να οργανώσουμε τη διαδικασία global code motion και είναι σημαντικό για υπολογιστές που εκδίδουν ένα μεγάλο αριθμό εντολών per clock. Το trace scheduling έχει 2 βήματα.(α) Trace Selection: Αναζητά ακολουθία βασικών block που οι λειτουργίες τους θα ενσωματωθούν σε ένα μικρότερο αριθμό εντολών. Αυτή η ακολουθία ονομάζεται trace. Γίνεται χρήση loop unrolling για την δημιουργία μεγάλων traces. Αφότου επιλεχθούν τα traces εκτελείται η δεύτερη λειτουργία. (β)Trace compaction: προσπαθεί να συμπτύξει το trace σε όσο το δυνατό μικρότερο αριθμό ευρέων εντολών.
(+)απλοποιεί τις αποφάσεις που αφορούν το global code motion (-)πρόβλημα σε περίπτωση εξόδου από τη μέση του trace.(-)Οι είσοδοι και οι έξοδοι στη μέση του trace προκαλούν σοβαρά προβλήματα που απαιτούν από τον compiler να παράξει κώδικα διόρθωσης το οποίο είναι πολύ δύσκολο να γίνει. Δεν είναι σίγουρο αν μπορεί να εφαρμοστεί σε όχι και τόσο απλά προβλήματα.
(2)Superblocks: Παράγονται από μια διαδικασία παρόμοια με τα traces αλλά είναι μια μορφή από extended basic blocks τα οποία έχουν μοναδική είσοδο και πολλαπλές εξόδους.

40)Τι σημαίνει pipeline, superpipeline & superscalar; Δώστε σχηματικά τα διαγράμματα σε κάθε περίπτωση.
Η pipeline είναι τεχνική που εκμεταλλεύεται το γεγονός ότι κάθε εντολή αποτελείται από διάφορα κομμάτια τα οποία είναι ανεξάρτητα μεταξύ τους και άρα μπορεί η εκτέλεση τους να αλληλοκαλύπτεται. Έτσι η εκτέλεση της εντολής χωρίζεται σε αυτά τα κομμάτια και η οργάνωση της CPU χωρίζεται σε στάδια τα οποία αναλαμβάνουν την εκτέλεση αυτών των κομματιών. Η αλληλοκάλυψη δεν αφορά την ίδια την εντολή αλλά διαφορετικές. Μόλις η εκτέλεση ενός κομματιού τελειώσει τα αποτελέσματα προωθούνται στο επόμενο στάδιο και το τρέχον περιλαμβάνει του προηγούμενου. Π.χ. στην DLX έχουμε

I1	IM	ID	EX	MEM	WB		
I2		IM	ID	EX	MEM	WB	
I3			IM	ID	EX	MEM	WB

Η superpipeline είναι περαιτέρω αποσύνθεση διαφόρων σταδίων του pipeline σε μικρότερα. Ένα παράδειγμα είναι το pipeline του R4000 όπου τα IM και MEM έχουν χωριστεί σε 2 και 3 στάδια αντίστοιχα.
Τέλος το superscalar είναι η τεχνική στην οποία ο επεξεργαστής εκδίδει περισσότερες της μιας εντολές ανά κύκλο ρολογιού. Ο αριθμός των εντολών δεν είναι συγκεκριμένος αλλά ποικίλει.

41)Γιατί όταν αυξηθεί ο αριθμός των σταδίων pipeline, αυξάνεται η απόδοση του επεξεργαστή; Πότε πρέπει να σταματήσουμε να προσθέτουμε στάδια;
Όταν αυξηθεί ο αριθμός των σταδίων Pipeline αυξάνεται η απόδοση του επεξεργαστή γιατί καθώς μικραίνει το clock cycle με την προϋπόθεση ότι κάθε στάδιο έχει όμοιο φόρτο εργασίας έτσι ώστε τα το ρολόι του συστήματος να είναι μικρότερο δυνατό. Με αυτόν τον τρόπο πετυχαίνουμε στην ιδανική περίπτωση εκτέλεση μιας εντολής ανά κύκλο ρολογιού. Στην πραγματικότητα η αύξηση της απόδοσης επιβαρύνεται από staff που μπορεί να προκύψει στο pipeline.

Δυσκολίες υλοποίησης μιας pipeline: (1)dealing with exceptions(over/underflow) (2)stopping restarting execution (3)instruction set complications(οι πολλαπλασιασμοί χρησιμοποιούν την ALU για πολλούς κύκλους).

ΚΕΦΑΛΑΙΟ 5

42) Ποια είναι τα επίπεδα ιεραρχίας της μνήμης και ποιοι οι λόγοι που επιβάλλουν την χρήση ιεραρχικής μνήμης σε ένα υπολογιστικό σύστημα;

Τα επίπεδα ιεραρχίας της μνήμης είναι 4. Οι καταχωρητές που βρίσκονται μέσα στη CPU και διαχειρίζονται από τον compiler, η κρυφή μνήμη που βρίσκεται κοντά στη CPU και διαχειρίζεται από το υλικό, η κύρια μνήμη του συστήματος όπου είναι αποθηκευμένο το πρόγραμμα και τα δεδομένα και διαχειρίζεται από το λειτουργικό σύστημα και η δευτερεύουσα μνήμη του συστήματος που μπορεί να είναι δίσκος ή άλλα μέσα και διαχειρίζεται από το λειτουργικό σύστημα και τον χρήστη. Στη δευτερεύουσα μνήμη τα δεδομένα αποθηκεύονται με σκοπό τη διατήρηση τους και μετά την παύση της λειτουργίας του υπολογιστικού συστήματος.

Οι δύο κύριοι λόγοι είναι η βελτίωση της απόδοσης και η μείωση του κόστους υλοποίησης. Οι σημερινές απαιτήσεις μας οδηγούν στη χρήση όλο και μεγαλύτερων ποσοτήτων μνήμης. Αυτό σημαίνει χρήση μεγάλων και άρα αργών μνημών. Η χρήση ιεραρχικής μνήμης και η κατανομή των δεδομένων ανάλογα με τη χρησιμότητα τους στα κατάλληλα επίπεδα της ιεραρχίας οδηγεί σε βελτίωση της απόδοσης. Η ιεραρχία αυτή βασίζεται στο temporal και spatial locality.

Καθώς οι επιδόσεις των CPU αυξάνουν απαιτείται γρήγορη επικοινωνία της με τα δεδομένα που συνεπάγεται χρήση γρήγορης μνήμης. Όμως χρήση τέτοιας μνήμης οδηγεί σε διόγκωση του κόστους λόγω της χρήσης μεγάλων ποσοτήτων. Με την ιεραρχική μνήμη χρησιμοποιείται η κατάλληλη ποσότητα γρήγορης μνήμης που οδηγεί σε αποδεκτό κόστος. Αν έχουμε μια σειρά από μνήμες $M_1M_2...M_N$ η ΚΜΕ επικοινωνεί άμεσα μόνο με το πρώτο μέλος M_1 της ιεραρχίας μνήμης.

43)Ονομάστε τα 4 επίπεδα στην ιεραρχία της μνήμης και δώστε το τυπικό μέγεθος, χρόνο προσπέλασης και bandwidth;

Τύπος	Καταχωρητές	Κρυφή μνήμη	Κύρια μνήμη	Δίσκοι
Μέγεθος	< 1KB	< 16MB	< 16GB	> 100GB
Τεχνολογία Υλοποίησης	Τυπική μνήμη με πολλαπλά ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Μαγνητικοί δίσκοι
Χρόνος προσπέλασης	0,25-0,5ns	0,5-25ns	80-250ns	5ms
Bandwidth MB/sec	20000-100000	5000-10000	1000-5000	20-150
Διαχειρίζονται από	Compiler	Hardware	Operating system	Operating system/ opertator

44)Μπορούν ποτέ να καταργηθούν τα ιεραρχικά επίπεδα και να υπάρχει μόνο ένα;

Τα ιεραρχικά επίπεδα μνήμης δεν μπορούν να καταργηθούν όσο υπάρχει μεγάλη διαφορά μεταξύ των τεχνολογιών μνήμης και μικροεπεξεργαστών. Προς το παρόν η τεχνολογία που αφορά μνήμες με επιδόσεις συγκρίσιμες με τον επεξεργαστή παραμένει πολύ ακριβή για να χρησιμοποιείται σε μεγάλες ποσότητες. Επίσης η επιθυμία μόνιμης αποθήκευσης δεδομένων αποτελεί εμπόδιο για την κατάργηση της ιεραρχίας, αν και έχουν γίνει βήματα προς τη μεριά της επίτευξης non volatility σε αυτές τις μνήμες. Όμως παραμένει το πρόβλημα του μεγέθους που πρέπει να έχει η μνήμη αυτή. Έτσι μάλλον είναι αδύνατο να καταργηθούν όλα τα επίπεδα και να μείνει μόνο ένα.

45)cache hit, cache miss, block και τοπικότητα των αναφορών: Όταν η CPU βρει τα δεδομένα που ζητάει στην cache τότε έχουμε **cache hit**. Όταν η CPU δεν βρει τα δεδομένα που ζητάει στην cache τότε έχουμε **cache miss**. Ένα σταθερό μέγεθος δεδομένων που περιέχει τη ζητούμενη λέξη και ονομάζεται **block** καλείται από την κύρια μνήμη και τοποθετείται στην cache. Σύμφωνα με την **τοπικότητα των αναφορών** η πληροφορία που χρησιμοποιήθηκε πρόσφατα είναι πιθανόν να ξαναχρησιμοποιηθεί στο άμεσο μέλλον και η πληροφορία που βρίσκεται κοντά στη πληροφορία που χρησιμοποιείται τώρα είναι πιθανόν να χρησιμοποιηθεί στο άμεσο μέλλον. Σε κάθε οργάνωση της κρυφής μνήμης το πλήθος των λέξεων ανά πλαίσιο όπως και το πλήθος των πλαισίων της κρυφής μνήμης και το πλήθος των block της κύριας μνήμης είναι δυνάμεις του 2.

46)Που μπορεί να τοποθετηθεί ένα Block σε μια cache; Τρόποι οργάνωσης της κρυφής μνήμης.

Υπάρχουν 3 διαφορετικοί τρόποι οργάνωσης της κρυφής μνήμης.

(α)Άμεση οργάνωση: Κάθε block της κύριας μνήμης μπορεί να τοποθετηθεί σε ένα συγκεκριμένο πλαίσιο της κρυφής μνήμης. Η διεύθυνση του πλαισίου της κρυφής μνήμης στο οποίο θα τοποθετηθεί το block της κύριας μνήμης με διεύθυνση Μ δίνεται από το υπόλοιπο της διαίρεσης του Μ δια του πλήθους των πλαισίων της κρυφής μνήμης.

(β)Οργάνωση πλήρους συσχέτισης: Κάθε block της κύριας μνήμης μπορεί να τοποθετηθεί σε οποιοδήποτε πλαίσιο της κρυφής μνήμης.

(γ)Set associative- Οργάνωση συνόλου συσχέτισης: Η κρυφή μνήμη θεωρείται ότι αποτελείται από ομάδες των τ πλαισίων που καλούνται σύνολα και κάθε block της κύριας μνήμης μπορεί να τοποθετηθεί σε οποιοδήποτε πλαίσιο ενός συγκεκριμένου συνόλου. Σε κάθε σύνολο αντιστοιχεί μια διεύθυνση. Η διεύθυνση του συνόλου της κρυφής μνήμης στο οποίο θα τοποθετηθεί το block της κύριας μνήμης με διεύθυνση Μ δίνεται από το υπόλοιπο της διαίρεσης του Μ δια του πλήθους των συνόλων της κρυφής μνήμης.

Η άμεση οργάνωση έχει το πλεονέκτημα ότι επιτρέπει ταυτόχρονη προσπέλαση των επιθυμητών δεδομένων και της ετικέτας. Αυτό έχει σαν συνέπεια, μεταξύ κρυφών μνημών με διαφορετική οργάνωση η κρυφή μνήμη άμεσης οργάνωσης έχει μικρότερο χρόνο προσπέλασης. Επειδή όμως από όλα τα block της κύριας μνήμης που αντιστοιχούν σε ένα πλαίσιο της κρυφής μνήμης μόνο ένα μπορεί να βρίσκεται κάθε χρονική στη κρυφή μνήμη, η κρυφή μνήμη άμεσης οργάνωσης έχει το μεγαλύτερο ρυθμό αποτυχιών. Για τον ίδιο λόγο ο ρυθμός αποτυχίας της κρυφής μνήμης άμεσης οργάνωσης αυξάνεται απότομα αν δυο ή περισσότερα block της κύριας μνήμης που αντιστοιχούν στο ίδιο πλαίσιο της κρυφής μνήμης χρησιμοποιούνται το ένα μετά το άλλο εναλλάξ. Επειδή μια μνήμη με οργάνωση πλήρους συσχέτισης μπορεί να περιέχει οποιοδήποτε συνδυασμό από block της κύριας μνήμης, έχει μικρότερο ρυθμό αποτυχιών από τις άλλες 2. Ο χρόνος προσπέλασης όμως και το κόστος υλοποίησης είναι μεγαλύτερο από τις άλλες 2. Όλα αυτά ισχύουν υπό την προϋπόθεση ότι όλα τα υπόλοιπα χαρακτηριστικά είναι ίδια.

47)Πώς βρίσκουμε ότι ένα block βρίσκεται στη cache;

Οι κρυφές μνήμες έχουν ένα address tag σε κάθε block frame το οποίο δίνει τη διεύθυνση του block. Όταν ο επεξεργαστής παράγει μια διεύθυνση για να διαβάσει την πληροφορία που είναι αποθηκευμένη στη διεύθυνση αυτή, τότε τα δυαδικά ψηφία του πεδίου Index χρησιμοποιούνται για να διευθυνσιοδοτηθεί η κρυφή μνήμη. Ταυτόχρονα τα δυαδικά ψηφία του πεδίου block address χρησιμοποιούνται για να διευθυνσιοδοτηθεί η κύρια μνήμη. Επίσης τα δυαδικά ψηφία του πεδίου block offset επιλέγουν τη λέξη του πλαισίου που θα εμφανιστεί στην έξοδο του πολυπλέκτη. Το tag κάθε cache block που μπορεί να περιέχει την επιθυμητή πληροφορία ελέγχεται κατά πόσο ταιριάζει με το block address που ζητήθηκε από τη CPU. Επειδή ο χρόνος είναι πολύτιμος όλα τα tags ελέγχονται παράλληλα. Επίσης στο tag προσθέτουμε και ένα valid bit για να δείχνει αν η πληροφορία είναι έγκυρη.

Δεν είναι απαραίτητο ο έλεγχος να γίνει σε όλο το address tag παρά μόνο στο tag και αυτό γιατί το offset δεν είναι απαραίτητο να χρησιμοποιηθεί γιατί ολόκληρο το block είτε θα βρίσκεται ή όχι στη cache. Επίσης ο έλεγχος του index είναι πλεονασμός αφού χρησιμοποιήθηκε για το διάλεγμα του set που θα ελέγχεται.

48)Ποιο block πρέπει να αντικατασταθεί σε ενδεχόμενο cache miss;

Υπάρχουν 3 στρατηγικές:

(α)random Το block επιλέγεται τυχαία. Έχει εύκολη υλοποίηση σε υλικό

(β)Least Recently Used (LRU): Απομακρύνεται το block που έχει μείνει αχρησιμοποίητο για περισσότερη ώρα. Βασίζεται στην αρχή της τοπικότητας των αναφορών.

(γ)First in first out (FIFO): Απομακρύνεται το block που προσκομίστηκε πρώτο στην cache.

50)Τι συμβαίνει σε ενδεχόμενο write;

Αν το block στο οποίο ανήκει η πληροφορία που θα γραφεί από την CPU στην μνήμη βρίσκεται στην κρυφή μνήμη υπάρχουν 2 τεχνικές για να γίνει αυτό.

Write through: Η πληροφορία γράφεται τόσο στο block της cache όσο και στο block του χαμηλότερου επιπέδου μνήμης. (+)έχει μικρό κόστος σε υλικό και επιπρόσθετα η κύρια μνήμη έχει το πλέον πρόσφατο αντίγραφο των δεδομένων.

Write back:Η πληροφορία γράφεται μόνο στο block της cache και το τροποποιημένο block γράφεται στη κύρια μνήμη μόνο όταν αντικαθιστάται. Το dirty bit δείχνει ότι τροποποιήθηκε.(+)η εγγραφή γίνεται με την ταχύτητα εγγραφής στη κρυφή μνήμη και πολλαπλές εγγραφές μέσα σε ένα block απαιτούν μόνο μια εγγραφή στη κύρια μνήμη.

Αν το block στο οποίο ανήκει η πληροφορία που θα γραφεί από την CPU στην μνήμη δεν βρίσκεται στην κρυφή μνήμη άρα έχω write miss υπάρχουν 2 τεχνικές για να γίνει αυτό.

Write allocate: Το block προσκομίζεται από την κύρια μνήμη στη κρυφή μνήμη.

No-write allocate: Τα write misses δεν επηρεάζουν την cache. Το block τροποποιείται μόνο στη χαμηλότερου επιπέδου μνήμη.

51)Πως βελτιστοποιούμε την απόδοση της cache;

Με μείωση του miss penalty, μείωση του miss rate, μείωση του χρόνου για την επίτευξη hit στην cache.

$$CPU_{execution\ time} = IC \times (CPI_{execution} + \text{Miss rate} \times \frac{Mem\ \ accesses}{Instruction} \times \text{Miss penalty}) \times \text{Clock cycle time}$$

Μείωση του miss penalty: (α)Με χρήση Multi level caches. Το πρώτο επίπεδο μπορεί να είναι αρκετά μικρό για να ταιριάζει με το clock cycle μιας γρήγορης CPU, ενώ το δεύτερο μπορεί να είναι αρκετά μεγάλο για να αποθηκεύει πολλά blocks που σ' αντίθετη περίπτωση θα προσπελαύνονταν από την κύρια μνήμη. Χρειάζονται επιπρόσθετο HW.

(β)Critical Word First and Early Restart: Βασίζεται στην ιδέα ότι η CPU συνήθως χρειάζεται μόνο μια λέξη από το block για κάθε στιγμή. Έτσι δεν χρειάζεται όλο το block για να ξεκινήσει η CPU. Αυτή η μέθοδος ευνοεί υλοποιήσεις με μεγάλα cache blocks. Υπάρχουν 2 τεχνικές.

(i)Early Restart: Μόλις φτάσει η λέξη που ζητήθηκε από το block στέλνεται στη CPU. Η CPU εξακολουθεί να δουλεύει ενώ συνεχίζεται η φόρτωση των υπόλοιπων λέξεων του block.

(ii)Critical word first: Ζητείται από τη μνήμη πρώτα το word που απαιτείται και μόλις φτάσει στέλνεται στη CPU ενώ συνεχίζεται η φόρτωση των υπόλοιπων λέξεων του block.

(γ)Giving Priority to Read Misses over Writes. Εξυπηρετεί το διάβασμα πριν το γράψιμο να έχει ολοκληρωθεί. Το write through με χρήση write buffers προκαλεί RAW conflicts με main memory reads σε cache misses. Αν περιμένουμε να αδειάσει το write buffer ίσως αυξηθεί το read miss penalty. Κάνουμε έλεγχο των περιεχομένων των write buffers πριν το read και αν δεν υπάρχουν conflicts τότε το memory access συνεχίζεται. Το κόστος γραψίματος από τον επεξεργαστή σε μια write-back cache μπορεί επίσης να μειωθεί. Γίνεται μεταφορά του dirty block σε write block, εκτέλεση του read και μετά εκτέλεση του write.

(δ)Merging Write Buffer: Γίνεται βελτίωση της αποτελεσματικότητας των buffer. Εάν ο buffer περιέχει κάποια τροποποιημένα blocks, ελέγχεται η διεύθυνση των νέων δεδομένων αν συμπίπτει με αυτή των έγκυρων περιεχομένων του buffer. Αν συμπίπτουν, τα νέα δεδομένα συγχωνεύονται με τα υπάρχοντα.

(ε)Victim cache: Προσθέτουμε buffer όπου τοποθετούνται δεδομένα που απορρίφθηκαν προηγουμένως από την cache, με σκοπό να έχουμε γρήγορο hit time όπως σε direct mapped.

Μείωση του Miss Rate: (α)Large Block Size: Αυξάνουμε το μέγεθος του block και ως αποτέλεσμα έχουμε μείωση των compulsory misses λόγω spatial τοπικότητας. Επίσης, για μικρό cache size, αυξάνονται τα conflict και capacity misses, γιατί μειώνεται ο αριθμός των blocks στην cache. Αυξάνεται το miss penalty, αφού η μεταφορά μεγαλύτερων blocks απαιτεί πολλαπλάσιο χρόνο. Η επιλογή του μεγέθους του block εξαρτάται τόσο από το latency όσο και από το bandwidth της χαμηλότερού επιπέδου μνήμης.

(β)Larger Caches: Αυξάνουμε την χωρητικότητα της cache. Η τεχνική αυτή είναι δημοφιλής σε off-chip caches. Τα μειονεκτήματα της είναι ότι έχουμε μεγαλύτερο hit time και υψηλότερο κόστος.

(γ)Higher Associativity: μείωση του miss rate αλλά αύξηση του hit time. Η eight way associative cache είναι εξίσου αποτελεσματική με τη fully associative. Με βάση τον κανόνα 2:1 για caches, η direct mapped cache μεγέθους N έχει ίδιο miss rate με μια two way set associative μεγέθους N/2.

(δ)Way prediction & Pseudo-Associative: Στο Way prediction φυλάγονται extra bits στην cache για να προβλέπουν το set του επόμενου cache access. Στο Pseudo-

Associative έχω διαίρεση της cache και σε κάθε miss γίνεται έλεγχος του άλλου μισού της cache αν είναι εκεί και αν ναι έχω pseudo-hit (slow hit). Με αυτό τον τρόπο πετυχαίνω γρήγορο hit time όπως σε DM με χαμηλά conflict misses όπως σε 2-way SA cache. Η συγκεκριμένη υλοποίηση είναι καλύτερη για L2 caches.
(ε)Compiler Optimizations: Γίνεται βελτιστοποίηση του compiler και δεν απαιτείται καθόλου HW. Όσο αφορά τις εντολές γίνεται αναδιάταξη των procedures στη μνήμη για μείωση conflict misses καθώς και Profiling έλεγχος για conflicts. Όσο αφορά τα δεδομένα στόχος είναι η βελτίωση των spatial and temporal locality. Αυτό μπορεί να γίνει με Merging Arrays (βελτίωση spatial locality με μονό array συμπαγών στοιχείων αντί για 2 arrays), Loop Interchange (αλλαγή nesting στα loops για data access στη σειρά που είναι αποθηκευμένα), Loop Fusion (συνδυασμός 2 ανεξάρτητων loops με ίδιο looping και μερικό variable overlap), Blocking (βελτίωση temporal locality προσπελαύνοντας data blocks κατ' επανάληψη αντί για σάρωση γραμμών η στηλών).
Μείωση μεσώ παραλληλισμού:(α)Non-blocking Caches για μείωση stalls σε misses. (β)HW prefetching για instructions & data. (γ)Compiler-Controlled prefetching
Μείωση Hit time: (α)Μικρές & Απλές caches: Έχουμε μικρή cache ώστε να χωράει στο ίδιο chip με επεξεργαστή για αποφυγή του time penalty για off-chip access και επειδή έχουμε λιγότερο hardware είναι γρηγορότερο. Επίσης η cache είναι απλή σαν να χρησιμοποιούμε direct mapping. Με κέρδος ότι μπορώ να έχω overlap μεταξύ tag check και data transmission το οποίο μειώνει αισθητά το χρόνο. Για L2 caches προτείνεται να κρατούνται τα tags on-chip για γρήγορο έλεγχο και τα data off-chip.
(β)Αποφυγή address translation κατά το indexing της cache: • Χρήση Virtual caches που χρησιμοποιούν virtual addresses για την cache (hits πιο πιθανά από misses). Δεν χρησιμοποιούνται μόνο virtual caches (i)για προστασία (ii) οι virtual addresses αναφέρονται σε πολλές physical addresses, απαιτώντας από τη cache να κατακλύζεται (λύση: αύξηση του cache address tag width με ένα process-identifier tag (PID), (iii)δύο διαφορετικές virtual addresses για την ίδια physical address (synonyms ή aliases), με αποτέλεσμα δύο πανομοιότυπα στην cache, (iv)Το I/O χρησιμοποιεί physical addresses, οπότε θα απαιτούσε mapping της virtual address σε physical address.
(γ)Pipelined Cache Access**:** Πετυχαίνει το latency ενός first-level cache hit να είναι πολλαπλοί κύκλοι ρολογιού, δίνοντας fast cycle time και slow hits.
(δ)Trace Caches: Μια trace cache βρίσκει μια δυναμική ακολουθία εντολών, συμπεριλαμβανομένων και αυτών σε περίπτωση που γίνει branch, για να φορτώσει σε ένα cache block . Το branch prediction αναδιπλώνεται μέσα στην cache και πρέπει να εξασφαλίζεται όλες οι διευθύνσεις να έχουν έγκυρο fetch. Η τεχνική αυτή απαιτεί πιο πολύπλοκους μηχανισμούς address mapping.

Average memory access time=AMAT για cache 2 επιπέδων.
AMAT= $HitTime_{L1}$+$MissRate_{L1}$ x $MissPenalty_{L1}$
$MissPenalty_{L1}$= $HitTime_{L2}$+$MissRate_{L2}$ x $MissPenalty_{L2}$
Local miss rate: Είναι τα misses στη cache δια του συνολικού αριθμού των memory accesses στην cache αυτή.
Global miss rate: Είναι τα misses στη cache δια του συνολικού αριθμού των memory accesses που εκτέλεσε η CPU. Το Global miss rate είναι σημαντικό γιατί δείχνει τι κλάσμα των memory accesses που φεύγουν από τη CPU πάνε μέχρι τη κύρια μνήμη.

52)Ποιες είναι οι κατηγορίες που κατατάσσονται τα misses;

-Compulsory: Στην πρώτη προσπέλαση ένα block δεν είναι ποτέ στην cache και έτσι είναι απαραίτητο να το φέρουμε. (ακόμα και σε άπειρη σε μέγεθος cache).
-Capacity: Αν κατά την εκτέλεση ενός προγράμματος δεν χωρούν όλα τα blocks στην cache θα συμβούν capacity misses επειδή ένα block μπορεί να απορριφθεί και να επανέλθει αργότερα.
-Conflict: Σε set associative ή direct mapped caches, συμβαίνουν misses επειδή αντικαθίστανται block όταν άλλα διαφορετικά αποθηκεύονται στον ίδιο χώρο.

53)Ποιες είναι οι βασικές παράμετροι που καθορίζουν μια cache; Δώστε τυπικές τιμές για κάθε παράμετρο

Οι βασικές παράμετροι που χαρακτηρίζουν μια cache είναι το μέγεθος του block (16-128 bytes), το hit time (1-2 cycles), το miss penalty (8-100 cycles), το miss rate (0,5-10%) και τέλος το μέγεθος της ίδιας της cache (0,016-1MB)

54)Ποιοι είναι οι διαφορετικοί τρόποι μέτρησης της απόδοσης της cache;

•Οι διαφορετικοί τρόποι μέτρησης της απόδοσης είναι:
Ο νόμος του Ambahl (απλή εφαρμογή του)
Speed up = Performance for entire task using enhancements / Performance for entire task without using enhancements
•Το miss rate που μας δίνει μια ιδέα για το πόσο καλά συμπεριφέρεται η cache στις προσπελάσεις.
Ο μέσος χρόνος προσπέλασης μνήμης. Συνδέεται με το miss rate και τι χάνουμε όταν εμφανίζεται ένα miss. Ισχύει η σχέση:
Average memory access time =hit time+Miss rate * Miss penalty
Αυτή η σχέση συνδέει τα διάφορα στοιχεία της ιεραρχίας μνήμης και μας δίνει μια σφαιρικότερη άποψη της απόδοσης της cache.
Μια πιο ολοκληρωμένη σχέση που δείχνει την επίδραση της cache είναι αυτή που έχει σχέση με το CPU time.
CPU time= (CPI + memory accesses/instruction*miss rate*miss penalty)*IC*cct
Αυτή η σχέση δείχνει την επίδραση της cache με βάση τη χρονική εκτέλεση ενός προγράμματος. Έτσι μπορεί να αφορά διαφορετικά προγράμματα με διαφορετικές παραμέτρους.

55)Σε ποιες περιπτώσεις η cache δεν βοηθάει την απόδοση ενός Η/Υ;

Όταν η κρυφή μνήμη είναι πολύ μικρή οπόταν χρειάζονται πολλές μεταφορές σελίδων. Έτσι η απόδοση του υπολογιστή είναι μικρότερη από έναν ίδιο Η/Υ χωρίς cache.

56)Γιατί γίνεται οργάνωση της μνήμης;

Ο σκοπός οργάνωσης της μνήμης είναι για επίτευξη μεγαλύτερου memory bandwidth (ο αριθμός των bytes που διαβάζονται ή γράφονται ανά χρονική μονάδα) καθώς και μείωση του latency. Οι περισσότερες **τεχνικές** που εφαρμόζονται είναι **για αύξηση του bandwidth.** (α) Wider main Memory: Με την αύξηση του bus έχουμε ανάλογη μείωση του miss penalty. (β) Simple Interleaved Memory: η οποία προσφέρει παραλληλισμό. (γ) Independent Memory Banks: Χρήση πολλαπλών memory controllers που επιτρέπουν σε banks να λειτουργούν ανεξάρτητα.

Μια cache ονομάζεται pseudo-associative όταν ισχύουν οι παρακάτω προϋποθέσεις: (α)στην περίπτωση hit η cache συμπεριφέρεται ως direct-mapped (b)στην περίπτωση miss πρέπει να πάει στο χαμηλότερο επίπεδο μνήμης και αναζητά

τα δεδομένα σε μια δεύτερη θέση της cache. Οι pseudo-associative caches έχουν ένα γρήγορο και ένα αργό hit time που αντιστοιχούν σε ένα κανονικό και ένα pseudo hit αντίστοιχα.

ΚΕΦΑΛΑΙΟ 6ο

57)Κατηγορίες υπολογιστών:(a)*Single instruction stream, single data stream (SISD)*-Uniprocessor **(b)***Single instruction stream, multiple data streams (SIMD)*-Vector architectures **(c)***Multiple instruction streams, single data stream (MISD)*-Special-purpose stream processors. **(d)***Multiple instruction streams, multiple data streams (MIMD)*-Off-the-shelf microprocessors

58)Thread- Level Parallelism: Πολλαπλές διεργασίες μοιράζονται κώδικα και δεδομένα -> threads (Threads hundreds to millions of instructions at a time)
Instruction- Level Parallelism one instruction at a time (basically hardware)

59)Centralized Shared-Memory Αντικαθιστώντας το bus με πολλαπλά ή με ένα switch μπορούμε να αυξήσουμε τον αριθμό των επεξεργαστών. Παραδείγματα είναι: SMPs (Symmetric multiprocessors), UMA (Uniform memory access) architecture.

60)Distributed-memory Cost effective τρόπος κλιμάκωσης του memory bandwidth Μειώνει το latency των accesses στην local memory Key disadvantage Επικοινωνία μεταξύ επεξεργαστών πολύ πολύπλοκη

61)Επικοινωνία επεξεργαστών: 1. Shared address space: Αναφορά στη μνήμη μπορεί να πραγματοποιηθεί απόοποιονδήποτε επεξεργαστή σε οποιαδήποτε θέση μνήμης, υπό την προϋπόθεση ότι ο επεξεργαστής έχει δικαίωμα πρόσβασης (DSM architectures or NUMAs) **2.**Clusters (Message Passing multiprocessors) Ανεξάρτητα processor-memory modules.

62)Challenges of Parallel Processing:Περιορισμένη παραλληλία των προγραμμάτων, Υψηλό κόστος επικοινωνίας Αποτέλεσμα: Δύσκολη η επίτευξη καλού speedup σε κάποιον παράλληλο υπολογιστή Πχ. Για να πετύχουμε speedup 80 με 100 επεξεργαστές από τον Amdahl's law υπολογίζουμε ότι μόνο το 0.25% των υπολογισμών μπορεί να πραγματοποιηθούν ακολουθιακά.

63)Όταν συγκρίνουμε Παράλληλες Αρχιτεκτονικές:1. Πρέπει να υπάρχει αρκετός παραλληλισμός στο πρόγραμμα που χρησιμοποιούμε (ανάλογα με τον αριθμό των επεξεργαστών) **2.** Πρέπει να γνωρίζουμε καλά τα benchmarks για την σωστή εξαγωγή συμπερασμάτων **3.** Να είναι της ίδιας τεχνολογίας

64)Καταστάσεις του cache block:1. Shared: Ένας ή περισσότεροι επεξεργαστές έχουν στην cache το block και η τιμή στη μνήμη και σε όλες τις caches είναι ενημερωμένη. **2.**Uncached: Κανένας επεξεργαστής δεν έχει αντίγραφο του cache block **3.**Exclusive: Ακριβώς ένας επεξεργαστής έχει αντίγραφο του cache block και το έχει γράψει, άρα και το αντίγραφο της μνήμης δεν είναι ενημερωμένο. Ο επεξεργαστής αυτός ονομάζεται *ιδιοκτήτης* του block.

65)Multithreading: Πολλαπλά threads μπορούν να μοιράζονται τα functional units ενόςεπεξεργαστή με επικάλυψη. Μεταφορά από thread σε thread πολύ γρηγορότερα από ότι αν έχουμε processes. Δύο βασικές προσεγγίσεις: **(α)**Fine-grained multithreading. Μεταφορά μεταξύ thread σε κάθε εντολή. (+) Κρύβονται τα stalls (-) έτοιμα threads χωρίς stalls καθυστερούν από άλλα. **(β)**Coarse-grained multithreading: Αλλαγές threads συμβαίνουν μόνο σε περίπτωση μεγάλων stall όπως level 2 cache misses

66)Simultaneous Multithreading: Παραλλαγή του multithreading που χρησιμοποιεί τους πόρους ενός multiple-issue, dynamically scheduled processor για να επιτύχει thread-level parallelism ταυτόχρονα με instructionlevel parallelism.

67)Παράλληλες Μηχανές Πρέπει: Να εκτελούν σειριακό κώδικα όσο γρήγορα εκτελείται σε σειριακούς υπολογιστές και παράλληλο κώδικα πολύ πιο γρήγορα
(1)Να μειωθεί το communication overhead (2)Το επιλεγμένο granularity να ταιριάζει στην αρχιτεκτονική (3)Να φροντίσουμε για καλό καταμερισμό εργασίας (load balancing) (4)Να αποφύγουμε deadlocks (5)Να φροντίσουμε για το data coherency

68)Πως ταξινομούνται οι παράλληλες αρχιτεκτονικές; Τι σημαίνει speed up σε παράλληλες αρχιτεκτονικές;

Οι παράλληλες αρχιτεκτονικές ταξινομούνται βάση την οργάνωση της μνήμης τους. Έτσι υπάρχουν δυο κύριες κατηγορίες. Στην πρώτη που ονομάζεται centralized shared memory uniform οι επεξεργαστές μοιράζονται μια κεντρική μνήμη. Οι επεξεργαστές είναι συνδεδεμένοι σε ένα bus και έχουν ομογενή προσπέλαση στη μνήμη. Στη δεύτερη κατηγορία ανήκουν τα συστήματα που η μνήμη είναι διαμοιρασμένη στους επεξεργαστές, οι οποίοι τώρα είναι περισσότεροι. Χαρακτηριστικό είναι ότι χρειάζεται ένας πιο πολύπλοκος τρόπος διασύνδεσης των επεξεργαστών.

Speed up = Χρόνος εκτέλεσης σε μια CPU / Χρόνος εκτέλεσης σε N CPU

Δείχνει πόσες φορές ο παράλληλος αλγόριθμος είναι πιο γρήγορος . Το ιδανικό speedup είναι αυτό που δεν έχει overhead.

69)Ποιες είναι οι δύο βασικές κατηγορίες παράλληλων αρχιτεκτονικών και ποια τα χαρακτηριστικά τους;

Οι δύο βασικές κατηγορίες παράλληλων συστημάτων είναι: η SIMD και MIMD. Η SIMD κατηγορία αφορά συστήματα στα οποία όλοι οι επεξεργαστές εκτελούν την ίδια εντολή αλλά πάνω σε διαφορετικά δεδομένα. (single instruction stream-multiple data streams). Κάθε επεξεργαστής έχει δικιά του μνήμη στην οποία αποθηκεύονται τα δεδομένα που χειρίζεται. Οι εντολές όμως βρίσκονται σε μια μοναδική μνήμη και υπάρχει ένας επεξεργαστής που αναλαμβάνει να παρέχει τις εντολές σε όλα τα επεξεργαστικά στοιχεία. Ένα άλλο χαρακτηριστικό είναι ότι οι επεξεργαστές είναι ειδικού σκοπού. Η MIMD κατηγορία περιλαμβάνει συστήματα στα οποία ο κάθε επεξεργαστής εκτελεί τις δικές του εντολές και λειτουργεί πάνω στα δικά του δεδομένα (multi instructions streams, multiple data streams). Οι επεξεργαστές που χρησιμοποιούνται συνήθως είναι αυτοί που κυκλοφορούν στο εμπόριο. Χαρακτηριστικά αυτών των συστημάτων είναι η ευελιξία. Μπορούν να χρησιμοποιούνται για την επίλυση ενός προγράμματος που απαιτεί υψηλή απόδοση, για την εκτέλεση πολλών διεργασιών ή για ένα συνδυασμό τους. Επίσης είναι ένας καλός συνδυασμός κόστους απόδοσης. Σχεδόν όλοι οι multiprocessors χρησιμοποιούν επεξεργαστές που συναντάμε σε workstations και σε servers με μονό επεξεργαστή. Επίσης παρέχουν ευελιξία. Με την κατάλληλη υποστήριξη σε hardware και software οι MIMD υπολογιστές μπορούν να υποστηρίξουν τόσο μία μονή εφαρμογή ενός χρήστη όσο και πολλαπλά tasks ταυτόχρονα.

70)Γιατί όταν αυξηθεί ο αριθμός των επεξεργαστών σε ένα παράλληλο υπολογιστή δεν αυξάνεται ανάλογα και η απόδοση του συστήματος; Ποιο είναι το ανώτατο όριο αύξησης της απόδοσης;

Δυο κύριοι λόγοι δεν προκαλούν ανάλογη αύξηση της απόδοσης ενός υπολογιστικού συστήματος με την αύξηση των επεξεργαστών. Η αύξηση του αρ. των επεξεργαστών οδηγεί σε αυξημένες απαιτήσεις επικοινωνίας. Όταν το προσφερόμενο bandwidth αδυνατεί να καλύψει τις ανάγκες των επεξεργαστών τότε παρουσιάζονται καθυστερήσεις και οι επεξεργαστές είναι σε κατάσταση αναμονής των δεδομένων. Ο δεύτερος λόγος είναι το ποσοστό παραλληλοποίησης του εκάστοτε προγράμματος. Όσο πιο μικρό είναι τόσο μικρότερη η συμβολή των περισσότερων επεξεργαστών στην απόδοση. Το ανώτατο όριο απόδοσης δίνεται από τον νόμο του Amdahl. Αν το ρ% του προγράμματος είναι παραλληλοποιήσιμο το (1-ρ)% δεν είναι και η μέγιστη απόδοση είναι: Speed up = 1 /(ρ/κ + (1-ρ)).

η παραπάνω σχέση δίνει το θεωρητικό μέγιστο και δεν περιλαμβάνει το κόστος επικοινωνίας. Λαμβάνοντας υπόψη το κόστος επικοινωνίας από ένα σημείο και μετά

δεν υπάρχει αύξηση της απόδοσης με αύξηση του κ, δηλαδή υπάρχει κάποιο μέγιστο στη συνάρτηση speed up(k).

71)Ποιοι είναι οι λόγοι που δεν έχουν διαδοθεί ευρέως οι παράλληλοι υπολογιστές(π.χ. παράλληλα PC); Ποιες εξελίξεις θα οδηγήσουν στη διάδοση των παράλληλων υπολογιστών;

1) το μεγάλο κόστος ανάπτυξης τέτοιων συστημάτων που τα περιόρισε σε ειδικές εφαρμογές. Επίσης τα αποτελέσματα χρήσης τους δύσκολα δικαιολογούν το κόστος απόκτησης τους..

2) η έλλειψη προγραμμάτων που εκμεταλλεύονται την παραλληλία. Επίσης το γεγονός ότι όλα τα προβλήματα δεν μπορούν να παραλληλοποιηθούν ή δεν είναι κατάλληλα για συγκεκριμένα συστήματα

3) το γεγονός ότι η όλη τεχνολογία βρίσκεται ακόμη στα σπάργανα με αποτέλεσμα τα πράγματα να είναι πολύ ρευστά και πολλές προσεγγίσεις να αποδεικνύονται ασύμφορες.

Η εξάπλωση της χρήσης τους θα επωφεληθεί από τα εξής:

α) Καθώς οι μικροεπεξεργαστές θα παραμείνουν ο κύριος τρόπος υλοποίησης επεξεργαστών, ο πιο λογικός τρόπος αύξησης της απόδοσης είναι η διασύνδεση πολλών επεξεργαστών. Αυτό θα είναι και πιο συμφέρον από πλευράς κόστους.

β) Δεν γνωρίζουμε για πόσο καιρό ακόμη τα επιτεύγματα της αρχιτεκτονικής θα συντηρήσουν την αλματώδη πρόοδο της απόδοσης των μικροεπεξεργαστών. Ήδη μοντέρνοι επεξεργαστές με έκδοση πολλών εντολών έχουν γίνει τόσο πολύπλοκοι που η αύξηση στην απόδοση φαίνεται να έχει ισοσκελιστεί από την πολυπλοκότητα τους και το κόστος

γ)Αρχίζει να διαφαίνεται μια κινητικότητα και πρόοδος στο λογισμικό, τον τομέα δηλαδή που αποτελεί την τροχοπέδη της εξάπλωσης των παράλληλων συστημάτων

Τέλος οι εξελίξεις ωθούν την τεχνολογία στα όρια της και οι μικροεπεξεργαστές πλησιάζουν όλο και περισσότερο στο όριο της ταχύτητας του φωτός.

72)Πως ταξινομούνται οι παράλληλες αρχιτεκτονικές και τι σημαίνει speedup;