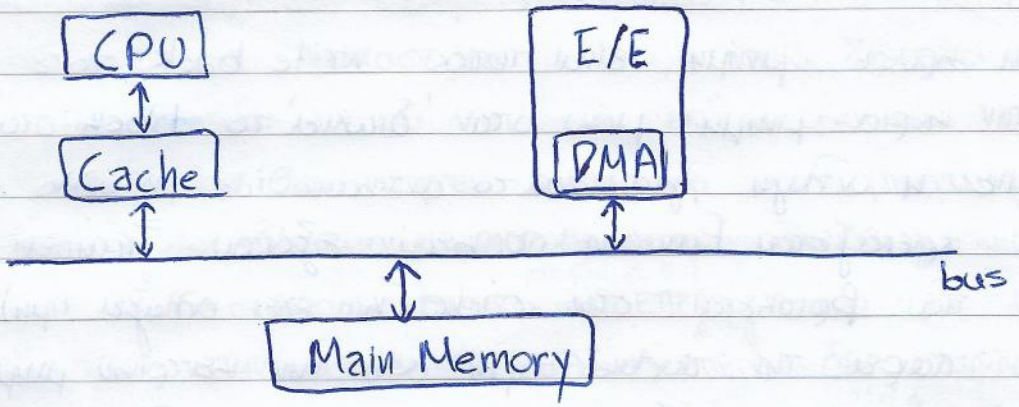
Ιωάννης Δάβανος ΑΜ4622

Σειρά 14η

Άσκηση 14.1



~~α) Υποθέτουμε πως η περιφερειακή συσκευή είναι συσκευή εισόδου (Input device). Ο επεξεργαστής θα διαβάσει τα σωστά δεδομένα, αν η cache δεν έχει τα δεδομένα (επομε~~

~~α) Εαν στην cache υπάρχουν τα παλιά δεδομένα (πριν γίνει η αλλαγή στην μ~~

α) Υποθέτουμε πως η περιφερειακή συσκευή είναι συσκευή εισόδου (Input device). Εαν η Cache έχει τα δεδομένα που ψάχνει η CPU πριν η περιφερειακή συσκευή αλλάξει τα δεδομένα, ~~τότε δεν θα~~ τότε η CPU θα διαβάσει τις παλαιές και επομένως λάθος τιμές. Από την άλλη, αν τα δεδομένα που ψάχνει η CPU δεν βρίσκονται στην Cache, τότε θα τα αποζητήσει από την κεντρική μνήμη, και επομένως τότε θα διαβάσει τις σωστές, αλλαγμένες από την περιφερειακή συσκευή τιμές.

Β) Εαν η κρυφή μνήμη του επεξεργαστή είναι τύπου write-through, ό,τι αλλάζει η CPU στη cache γίνεται και η αλλαγή στην main memory. Έτσι δεν αφήνει κανένα περιθώριο, να υπάρχει η παλαιά τιμή στην main και η καινούρια στην cache. Αυτο σημαίνει ότι η main memory έχει παντα τις ενημερωμένες τιμές των

δεδομένων. Συμπερασματικά, η συσκευή εξόδου δεν έχει κάποιο τρόπο να πάρει τα λανθασμένα δεδομένα διότι τα καινούργια επικαλύπτουν πάντα τα παλαιά.

γ) Εαν η κρυφή μνήμη είναι τύπου write back, τοτε γράφει στην κύρια μνήμη μόνο όταν 'διώχνει' το block στο οποίο είχε η αλλαγή, για να το αντικατασθίσει με άλλο block που χρειάζεται. Εαν η συσκευή εξόδου ζητήσει κάτι που βρίσκεται στη cache (και έχει αλλάξει τιμή) τοτε θα πάρει την παλαιά τιμή από την κεντρική μνήμη. Αντίθετα, αν η συσκευή ζητήσει κάτι που έχει 'διώξει' η cache, θα πάρει την σωστή τιμή.

Άσκηση 14.3

α) Εαν δεν υπάρχει πρωτόκολλο συνοχής στην αρτηρία και στις κρυφές μνήμες, τοτε ο κάθε επεξεργαστής θα πάρει την αρχική τιμή της κοινής μεταβλητής sharedVar=0 και θα κάνει πράξεις με αυτή, χωρίς να συνεργάζεται, αφού η νέα τιμή της μεταβλητής θα βρίσκεται στη cache της κάθε CPU (η τιμή της sharedVar μπορεί να διαφέρει από cache σε cache). Αυτο δείχνει ότι δεν υπάρχει συνεργασία. Επειδή τα caches είναι τύπου write back και δεδομένου ότι στο βήμα iv) η cache A δεν 'διώχνει' το block που έχει την sharedVar τοτε στο βήμα v) η CPU B θα διαβάσει την παλαιά τιμή της sharedVar από την main memory, δηλαδή 0. Αν οι κρυφές μνήμες ήταν τύπου write-through, στο βήμα iv) η νέα τιμή της shareVar=1 θα γραφόταν και στην main memory, με αποτέλεσμα στο βήμα v) η CPU B να διαβάζει την νέα ενημερωμένη και σωστή τιμή της κοινόχρηστης μεταβλητής.

*αφού δεν την βρήκε στην cache

β) Αρχικά, η κοινόχρηστη μεταβλητή βρίσκεται μόνο στη μνήμη και έχει τιμή 0. Στο βήμα ii) ο επεξεργαστής Α διαβάζει την κοινόχρηστη μεταβλητή (από την κεντρική μνήμη)* και μπαίνει σε κατάσταση S, ~~όπου έχουμε καταστάσεις MSI~~. Έπειτα Αυτό συμβαίνει επειδή έχουμε καταστάσεις MSI, η οποία δεν έχει την κατάσταση E, επομένως θα πάρει την S μαθώς είναι και clean και ~~δεν~~ ~~γνωρίζουμε~~ μπορεί να υπάρχει αντίγραφο της τιμής αυτής σε άλλη cache (δεν υπάρχει στην περίπτωση αυτή, αλλά δεν γίνεται να μην του βάλουμε μια κατάσταση). Έπειτα στο βήμα iii) ο επεξεργαστής Β μαίνει της SharedVar στην cache της, αλλά επειδή δεν υπάρχει εκεί, το ζητά από την main memory. Ο επεξεργαστής (cache) Α όμως έχει ~~αυτό που ζητά~~ την μεταβλητή SharedVar στην κεντρική του μνήμη, και αφού βλέπει το αίτημα της Β στο bus, επειδή να το δώσει πρώτη αυτή πριν το κάνει η κεντρική μνήμη (επομένως ο επεξεργαστής Β "παίρνει" την sharedVar από την cache του Α επεξεργαστή). Στο βήμα iv) ο επεξεργαστής Α αλλάζει την τιμή της μεταβλητής SharedVar σε 11, και για αυτό μπαίνει σε κατάσταση Μ και κάνει broadcast στο bus ότι άλλαξε την τιμή της και να κάνουν invalidate οι υπόλοιπες κρυφές μνήμες, το αντίγραφο της μεταβλητής. Τέλος, στο βήμα v) ο επεξεργαστής Θα δει ότι η τιμή της sharedVar είναι invalid στην cache της και θα το ζητήσει από το bus. Επειδή η κρυφή μνήμη του Α επεξεργαστή, έχει την σωστή τιμή, θα την 'δώσει' στην cache Β. Ως εκ τούτου η μεταβλητή θα πάει σε κατάσταση Shared (S).

Αν είχαμε τώρα και την κατάσταση Exclusive θα είχαμε τις εξής διαφορές:

- Στο βήμα ii) Η μεταβλητή θα έπαιρνε την κατάσταση Exclusive (αντί για Shared)
- Στο βήμα iii) Η μεταβλητή θα άλλαζε την κατάσταση της σε Shared, αφού η CPU Α (ή cache καλύτερα) το έδωσε στην cache Β

γ) Εαν αλλάζαμε το παραπάνω σενάριο όπως μας λέει η εκφώνηση αυτή, η διαφορα μεταξύ των 2 πρωτοκόλλων είναι ότι στην κατάσταση MSI (που δεν έχουμε την κατάσταση Ε) η μεταβλητή θα είχε την κατάσταση S εξαρχής στο βήμα ii) και στο iv) θα γινόταν Μ όταν άλλαξε. Στο βήμα v) η μεταβλητή θα γινόταν S ξανά.
Από την άλλη, στην κατάσταση MESI (που έχουμε την κατάσταση Ε) η μεταβλητή θα είχει την κατάσταση Ε στο βήμα ii) και στο iv) θα γινόταν Μ όταν άλλαξε.
Στο βήμα v) η μεταβλητή θα γινόταν S.
Αρα στο MSI αρχικα είναι S μετα είναι Μ
Ενω στο MESI είναι Ε αρχικά και μετα Μ.

Άσκηση 14.4

α)
```
lw   t0, 120(x0)
addi t0, t0, 2
sw   t0, 128(x0)
lw   t0, 124(x0)
addi t0, t0, -1
sw   t0, 132(x0)
```

β)
```
lw   t0, 120(x0)
lw   t1, 124(x0)
addi t0, t0, 2
sw   t0, 128(x0)
addi t1, t1, -1
sw   t1, 132(x0)
```

Όταν το πρόγραμμα α) εκτελείται στην "κλασική pipeline" των 5 βαθμίδων, χάνονται 2 κύκλοι ρολογιού (μετα από κάθε load χάνουμε 1 κύκλο, εφόσον βρίσκαμε αυτό που θέλουμε στη cache). Αυτό συμβαίνει λόγω αλληλεξαρτήσεων, δεν είχει προλάβει να γίνει load της διεύθυνσης που ζητάμε ωστε να χρησιμοποιηθεί στην επόμενη εντολή. Με την αναδιάταξη των εντολών καταφέρνουμε να μην χάνουμε κανένα κύκλο ρολογιού ενω παράλληλα να μην αλλάζουμε την σημασιολογία του προγράμματος. Αυτο επιτυγχάνεται με το να χρησιμοποιούμε έναν ακόμη καταχωρητή και κάνουμε τα load στην αρχή του 'προγράμματος', στη θέση του 'χαμένου' κύκλου, καλύπτωντας τον και λύνοντας έτσι το πρόβλημα.

γ)
```
lw   t0, 0(x14)
addi t0, t0, 2
sw   t0, 0(x16)
lw   t0, 0(x15)
addi t0, t0, -1
sw   t0, 0(x17)
```

Αν οι pointer px και pb έχουν διαφορετικές τιμές, τότε μπορεί να γίνει instruction scheduling όπως στο ερώτημα β). Απλά θα περιστατούμε έναν ως καταχωρητή, αποθηκεύουμε τις τιμές στους 2 πλέον διαφορετικούς καταχωρητές και κάνουμε τις πράξεις χωρίς να χάσουμε κύκλο ρολογιού. Από την άλλη αν οι τιμές των px και pb είναι ίδιες τότε το πρόγραμμα υπολογίζει αυτοστικά το $py = px - 1 \Rightarrow py = pa + 2 - 1 \Rightarrow \boxed{py = pa + 1}$. Για να συμβεί αυτό πρέπει πρώτα να υπολογισθεί το p~~x~~) και μετά το py, πράγμα που είναι αδύνατο με instruction scheduling, αφού ο compiler δεν μπορεί να ξέρει την τιμή που θα έχουν οι pointers. Στην πρώτη περίπτωση το αποτέλεσμα που υπολογίζει το 'πρόγραμμα' είναι παρόμοιο με το β) ερώτημα ενώ στη δεύτερη περίπτωση το 'πρόγραμμα' αυτοστικά υπολογίζει το $*py = *pa + 1$.

δ) Στο 1% των περιπτώσεων που οι pointers px και pb έχουν ίση τιμή τότε η ~~επόμενη εντολή~~ επόμενη εντολή load θα υποχρεούται να περιμένει, ενώ στις άλλες 99% των περιπτώσεων μπορεί να εκτελεστεί ανεξάρτητα. Η επόμενη εντολή load μπορεί να εξαρτάται από την προηγούμενη εντολή store, αν βρισκόμαστε στο 1% των περιπτώσεων όπου px=pb αρχικά. Το hardware, λόγω της ικανότητας του, να μπορεί να ελέγχει τις τιμές των δεικτών (πράγμα που δεν μπορεί να κάνει ο compiler), θα μπορούσε να κάνει το λεγόμενο 'instruction scheduling'. Στο 1% των περιπτώσεων έχουμε px=pb, άρα και αλληλοεξαρτήσεις, που σημαίνει ότι θα χάνουμε χρόνο, ενώ στο υπόλοιπο 99% των περιπτώσεων δεν έχουμε αλληλοεξαρτήσεις και επομένως δεν θα χάνουμε χρόνο.

Άσκηση 14.5

α) Εαν ο επεξεργαστής δεν είχε multithreading, και αφού βρίσκει εντολές που θα τον κρατήσουν απασχολημένο για 8 από τους 80, θα χρώνταν 80 - 8 = 72 κύκλοι ρολογιού (72 κύκλοι που η CPU θα 'περιμένει' χωρίς να κάνει κάτι παραγωγικό).

β) Στο ερώτημα α) είδαμε πως το πρόγραμμα Α χωρίς multithreading χάνει 72 κύκλους ρολογιού. Αν είχαμε multithreading, μόλις ο επεξεργαστής διαπιστώνει ότι η εντολή load του Α προκαλεί αστοχία, αρχίζει να κάνει fetch εντολές του Β προγράμματος μέσω του PCB και να τις εκτελεί χρησιμοποιώντας τους καταχωρητές RFB. Αυτό σημαίνει πως δεν ψάχνει να βρει εντολές του Α τον θα τον κρατήσουν (παραγωγικά) απασχολημένο για 8 κύκλους ρολογιού, κάνει αμέσως την 'αλλαγή' στο Β πρόγραμμα. Ως εκ τούτου το πρόγραμμα Α τωρα θα χάσει 80 - 0 = 80 κύκλους, γεγονός που σημαίνει ότι το πρόγραμμα Α χρειάζεται περισσότερους κύκλους για να εκτελεστεί, αρα είναι πιο αργό από ότι πριν

γ) Αν έχουμε multithreading, και το πρόγραμμα Β δεν έχει καποια αστοχία για τουλάχιστον αυτούς τους 80 κύκλους, τότε δεν θα χαθούν κύκλοι ρολογιού. Όσο το πρόγραμμα Α θα 'περιμένει' 80 κύκλους λόγω της αστοχίας, ο επεξεργαστής θα είναι απασχολημένος αυτους τους 80 κύκλους εκτελώντας εντολές του Β προγράμματος (που σημαίνει δεν θα περιμένει → όχι χάσιμο κύκλων). Συμπερασματικά, δηλαδή συνολικά δεν θα χαθεί κανένας κύκλος. Όσο το Α θα περιμένει χωρίς να έχει κάτι να κάνει, θα τρέχει ο Β κάνοντας κάτι χρήσιμο.