ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Μελέτη επεξεργαστών διπλού πυρήνα, πρωτοκόλλων συνοχής μνήμης και μελέτη υλοποίησης σε FPGA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δρασίδης Γεώργιος

Βόλος, Μάρτιος 2008

Ευχαριστίες

Καταρχήν, θα ήθελα να ευχαριστήσω τον κ. Γ. Σταμούλη για την πολύτιμη καθοδήγηση και τις συμβουλές του, όπως και για την επίλυση των επιμέρους προβλημάτων πριν καν δημιουργηθούν.
Τον κ. Γ. Δημητρίου για όλη την υποστήριξη, τις γνώσεις και τις πολλές ώρες δουλειάς που μου αφιέρωσε για να ολοκληρωθεί η εργασία αυτή.
Επίσης τον Δ. Συρίβελη για την αμέριστη βοήθεια σε θέματα υλοποίησης και όλο τον χρόνο που κατανάλωσε στην επίλυση των επιμέρους προβλημάτων.
Ευχαριστώ όλους τους φίλους και συναδέλφους στο γραφείο Ε5. Αντώνη, Βασίλη Δημήτρη ευχαριστώ για τις ώρες χαλάρωσης. Τον Δ. Καραμπατζάκη για όλη τη βοήθεια να ξεκινήσω την προσπάθεια.

Ιδιαίτερα Αλέξανδρε, Ντίνη, Μιχάλη, Ελένη ευχαριστώ για όλα

Τέλος θα ήθελα να αφιερώσω την εργασία αυτή στους γονείς μου και την αδερφή που έχουν κάνει τα πάντα για μένα

Περιεχόμενα

1. Εισαγωγή

Αντικείμενο αυτής της διπλωματικής εργασίας είναι η μελέτη αρχιτεκτονικών πολυεπεξεργαστικών συστημάτων και των κατηγοριών στις οποίες αυτές κατατάσσονται. Θα γίνει μια επισκόπηση και θεωρητική προσέγγιση των πρωτοκόλλων συνοχής κρυφής μνήμης, τα οποία χρησιμοποιούνται για την υποστήριξη και υλοποίηση των συστημάτων αυτών και ειδικότερα των αναγκαίων συνθηκών για την υλοποίηση και ορθή λειτουργία πολυεπεξεργαστικών συστημάτων. Επίσης θα δούμε πως γίνεται να εφαρμόσουμε την θεωρητική αυτή προσέγγιση σε ένα πραγματικό soft-core επεξεργαστή. Δηλαδή θα γίνει μελέτη ενός μονοεπεξεργαστικού συστήματος όπως είναι αυτό του μικροεπεξεργαστή LEON2 της European Space Agency με σκοπό την επέκταση της αρχιτεκτονικής του σε πολυεπεξεργαστή δύο πυρήνων, παρέχοντας έτσι την υποδομή για την επεξεργασία προγραμμάτων για δύο πυρήνες. Επίσης, θα γίνει η μελέτη και ανάλυση της ροής πληροφορίας για την διαδικασία υλοποίησης του μικροεπεξεργαστή αυτού σε επαναπρογραμματιζόμενη διάταξη FPGA.

Σκοπός αυτής της διπλωματικής εργασίας είναι η εμβάθυνση και η εξοικείωση με αρχιτεκτονικές συστημάτων πολλαπλών επεξεργαστών που παίζουν, και θα συνεχίσουν να παίζουν ένα αυξανόμενα κεντρικό ρόλο στην επεξεργασία δεδομένων. Αυτή η άποψη δεν βασίζεται μόνο στο γεγονός ότι η απόδοση ενός μονοεπεξεργαστή έχει φτάσει σχεδόν σε ένα επίπεδο το όποίο είναι πολύ δύσκολο να ξεπεραστεί λόγω περιορισμών στις περαιτέρω βελτιώσεις αλλά και στην εκτίμηση ότι το επόμενο επίπεδο σχεδιασμού συστημάτων, αυτό των πολυεπεξεργαστών, θα είναι πολύ πιο ελκυστικό εξαιτίας των δραματικών αυξήσεων στην πυκνότητα των τσιπ[20]. Ιδιαίτερα σε ενσωματωμένα συστήματα που χρησιμοποιούνται για μια συγκεκριμένη και εξειδικευμένη λειτουργία, η προσθήκη επεξεργαστικής ισχύος με την μέθοδο των πολυεπεξεργαστών είναι πολύ δημοφιλής.

2. Πολυεπεξεργαστές

2.1. Γενικά

Στο κεφάλαιο αυτό θα αναλύσουμε ένα από τα μοντέλα παράλληλων αρχιτεκτονικών, το οποίο και χρησιμοποιήσαμε στην εργασία αυτή για την μετατροπή του μικροεπεξεργαστή LEON2 σε πολυεπεξεργαστή δύο πυρήνων[1][2].

Γενικότερα η ανάγκη για πολυεπεξεργαστές προήλθε από την μείωση του ρυθμού προόδου των μονοεπεξεργαστών. Η ιδέα να χρησιμοποιήσουμε πολλούς επεξεργαστές για την αύξηση της απόδοσης χρονολογείται από τους πρώτους ηλεκτρονικούς υπολογιστές. Σύμφωνα με την κατηγοριοποίηση του Flynn(1966) ως προς τον παραλληλισμό των ακολουθιών εντολών και δεδομένων, οι ηλεκτρονικοί υπολογιστές χωρίζονται στις εξής κατηγορίες:

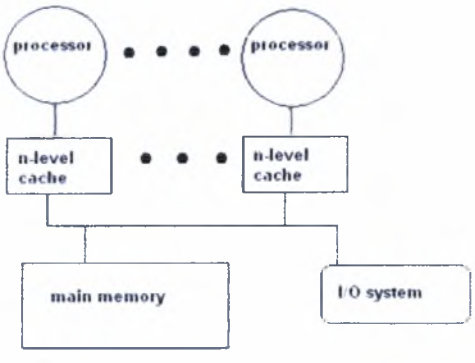
1. ***S**ingle **I**nstruction **S**ingle **D**ata (SISD):* Σε αυτή την κατηγορία ανήκουν οι μονοεπεξεργαστές.
2. ***S**ingle **I**nstruction **M**ultiple **D**ata (SIMD):* Η ίδια εντολή εκτελείται από πολλούς επεξεργαστές χρησιμοποιώντας διαφορετικές ακολουθίες δεδομένων. Κάθε επεξεργαστής έχει την δική του μνήμη δεδομένων, αλλά υπάρχει μια μνήμη εντολών. Πρόκειται ουσιαστικά για διανυσματικούς επεξεργαστές που εκμεταλλεύονται τον παραλληλισμό δεδομένων.
3. ***M**ultiple **I**nstruction **S**ingle **D**ata (MISD):* Θεωρητική κατηγορία. Δεν έχει κατασκευαστεί ηλεκτρονικός υπολογιστής αυτού του τύπου.
4. ***M**ultiple **I**nstruction **M**ultiple **D**ata (MIMD):* Κάθε επεξεργαστής έχει την δική του ακολουθία εντολών και τα δικά του δεδομένα. Εκμετάλλευση παραλληλισμού σε επίπεδο νημάτων ή και διεργασιών.

Η κατηγορία, με την οποία θα ασχοληθούμε είναι οι MIMD κοινής μνήμης, δηλαδή οι πολυεπεξεργαστές. Σε αυτήν την κατηγορία έχουμε πολλούς επεξεργαστές που εκτελούν ένα μόνο πρόγραμμα ή πολλαπλά προγράμματα και μοιράζονται τον ίδιο κώδικα και μεγάλο μέρος του χώρου διευθύνσεων της μνήμης.

Οι MIMD πολυεπεξεργαστές χωρίζονται σε 2 κατηγορίες με την σειρά τους, ανάλογα με τον αριθμό των επεξεργαστών τους, την οργάνωση της μνήμης και τον τρόπο διασύνδεσης τους.

Η πρώτη κατηγορία είναι οι κεντρικοποιημένοι πολυεπεξεργαστές κοινής μνήμης(centralized shared-memory multiprocessors), όπου ένας μικρός αριθμός επεξεργαστών (< 30) μοιράζεται μια κεντρική μνήμη και συνδεέται με μία αρτηρία (bus). Λόγω της ύπαρξης της κεντρικής μνήμης ο χρόνος προσπέλασης της από κάθε επεξεργαστή είναι ομοιόμορφος, γι' αυτό και οι πολυεπεξεργαστές αυτοί ονομάζονται *συμμετρικοί πολυεπεξεργαστές κοινής μνήμης*(*Symmetric MultiProcessors ή SMP*)

σχήμα 1: **SMP**



Η δεύτερη κατηγορία αποτελείται από συστήματα με φυσικά κατανεμημένη μνήμη, που ονομάζονται *κατανεμημένοι επεξεργαστές κοινής μνήμης*(Distributed Shared-Memory Multiprocessors ή DSM)(βλέπε σχήμα2).

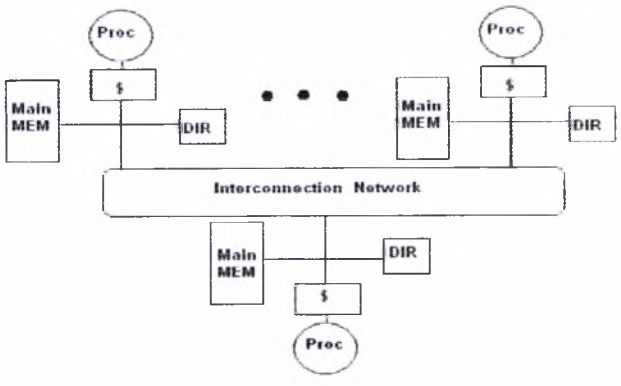
Σε αυτή την κατηγορία οι επεξεργαστές αποτελούν ξεχωριστούς κόμβους, όπου κάθε κόμβος έχει την δική του κρυφή μνήμη, δική του κύρια μνήμη(cache) και σύστημα I/O. Ο κάθε κόμβος μπορεί να έχει ένα μικρό αριθμό επεξεργαστών σε SMP διάταξη, οπότε και έχουμε υβριδική αρχιτεκτονική. Οι κόμβοι συνδέονται μεταξύ τους με ένα σύστημα διασύνδεσης που αποτελείται από αρτηρίες ή μεταγωγείς(switches). Ο χρόνος προσπέλασης της μνήμης είναι διαφορετικός για κάθε κόμβο ανάλογα με τα ποια λέξη δεδομένων θέλει να προσπελάσει, και η επικοινωνία μεταξύ των κόμβων γίνεται με ανταλλαγή μηνυμάτων αφού δεν υπάρχει ενιαίος χώρος διευθύνσεων μνήμης.

Σε κάθε συνδιαλλαγή εμπλέκονται τρεις οντότητες:

1) Ο *τοπικός κόμβος*, που ξεκινάει την συνδιαλλαγή

2) Ο *κόμβος ιδιοκτήτης*, όπου η φυσική θέση μιας διεύθυνσης μνήμης
3) Ο *απομακρυσμένος κόμβος*, που έχει ένα αντίγραφο του μπλοκ της cache

σχήμα2: **DSM**



Εξαιτίας του στόχου αυτής της διπλωματικής εργασίας για την υλοποίηση ενός συστήματος δύο πυρήνων(dual-core), η αρχιτεκτονική η οποία και επιλέχθηκε είναι η SMP η οποία είναι και η δημοφιλέστερη για σύστημα με λίγους πυρήνες, που στην περίπτωση μας θα είναι δύο.

Ο λόγος γι' αυτήν την επιλογή θα προσπαθήσουμε να γίνει ξεκάθαρος μέσα από αυτό το κεφάλαιο, όπου θα αναλυθεί η αρχιτεκτονική SMP, όπως και τα προβλήματα που προκύπτουν για την ορθή λειτουργία ενός συστήματος κατά την προσπάθεια επέκτασης της αρχιτεκτονικής από έναν σε δύο επεξεργαστές και πως αυτά επιλύονται. Επίσης θα αναλυθεί ο σχεδιασμός ενός τέτοιου συστήματος και οι επιμέρους αποφάσεις που πρέπει να πάρουμε, αλλά και η εφαρμογή τους στον μικροεπεξεργαστή LEON2 και οι ιδιαιτερότητες που εμφανίζονται κατά την υλοποίηση.

2.2. Συμμετρικοί πολυεπεξεργαστές κοινής-μνήμης(SMP)

Η πλέον επικρατούσα μορφή παράλληλων αρχιτεκτονικών είναι οι πολυεπεξεργαστές μικρού ή μεσαίου μεγέθους που παρέχουν καθολικό φυσικό χώρο διευθύνσεων και συμμετρικού κόστους πρόσβαση στη κύρια μνήμη από όλους τους επεξεργαστές και ονομάζονται *SMP*. Κάθε επεξεργαστής έχει μια cache, ενός ή περισσότερων επιπέδων, και συνδέονται στο ίδιο μέσο διασύνδεσης που μπορεί να είναι ένα διαμοιραζόμενο κοινό bus ή ένα σύστημα με switches[2].

Ο αποδοτικός διαμοιρασμός των πόρων, όπως η μνήμη και οι επεξεργαστές από αυτό το σχήμα κάνει τα συστήματα αυτά να χαρακτηρίζονται ως υψηλής απόδοσης. Επίσης η δυνατότητα να έχουμε πρόσβαση σε κοινά δεδομένα με την ίδια απόδοση από κάθε επεξεργαστή καθιστά το μοντέλο αυτό ιδιαιτέρως ελκυστικό για παράλληλο προγραμματισμό.

Ένα από τα πλεονεκτήματα της αρχιτεκτονικής SMP είναι ότι το μοντέλο προγραμματισμού κοινού χώρου διευθύνσεων υποστηρίζεται απευθείας από το υλικό και άρα είναι σαφώς πιο απλό για ένα προγραμματιστή υψηλού επιπέδου. Οι διεργασίες του χρήστη μπορούν να διαβάζουν και να γράφουν κοινές εικονικές διευθύνσεις μέσω απλών λειτουργιών φόρτωσης-αποθήκευσης. Αντιθέτως ένα κατανεμημένο σύστημα όπου η επικοινωνία γίνεται με το μοντέλο της ανταλλαγής μηνυμάτων πρέπει να υποστηρίζεται από ένα ενδιάμεσο επίπεδο λογισμικού, μια run-time βιβλιοθήκη για παράδειγμα.

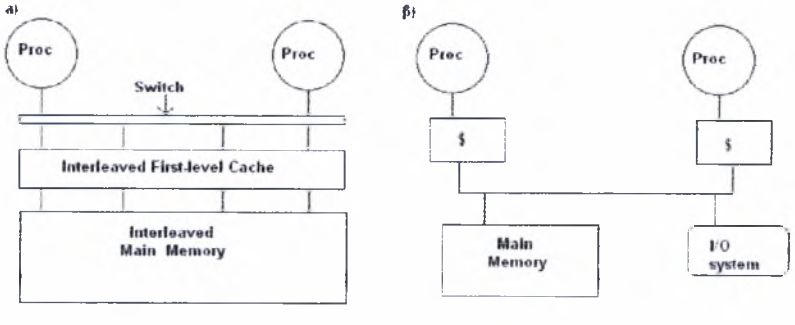
Γενικά, σε ένα σύστημα MP κάθε τοπικός υπολογισμός και επικοινωνία παράγει προσπελάσεις μνήμης σε ένα κοινό χώρο διευθύνσεων. Από την οπτική ενός αρχιτέκτονα το ζήτημα κλειδί είναι η οργάνωση της ιεραρχίας μνήμης. Η ιεραρχία μνήμης στους πολυεπεξεργαστές μπορεί να σχεδιαστεί σύμφωνα με μια από τις εξής κατηγορίες: (σχήμα 3)

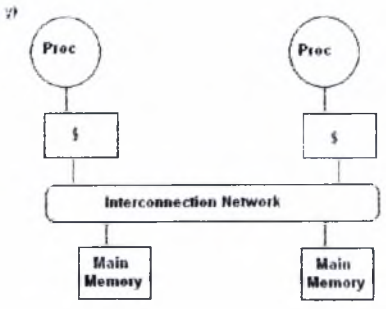
α)*Κοινή κρυφή μνήμη,*

β)*Κοινή κύρια μνήμη και κοινά διαμοιραζόμενο bus,*

γ)*Dancehall*

σχήμα 3:





Στην προσέγγιση του σχήματος 3-α υπάρχει διασύνδεση του επεξεργαστή με την κρυφή μνήμη πρώτου επιπέδου και με την σειρά της, της κρυφής μνήμης με την κύρια μνήμη. Η κρυφή μνήμη μπορεί να είναι διαφυλλωμένη(interleaved) για να πετύχουμε μεγαλύτερο εύρος ζώνης(bandwidth). Αυτό το σχήμα ισχύει μόνο σε μικρή κλίμακα, διότι ακόμη και με λίγους επεξεργαστές, η κρυφή μνήμη πρέπει να μας παρέχει πάρα πολύ μεγάλο εύρος ζώνης, αφού όλοι οι επεξεργαστές θα προσπελαύνουν την ίδια κρυφή μνήμη, αλλά και να εγγυάται μικρή καθυστέρηση για την προσπέλαση της.

Στο σχήμα 3-β η διασύνδεση υλοποιείται με ένα bus κοινά διαμοιραζόμενο που βρίσκεται ανάμεσα στην κρυφή μνήμη Ν επιπέδων, που είναι ξεχωριστή για κάθε επεξεργαστή, και την μοναδική κοινή κύρια μνήμη. Η προσέγγιση αυτή είναι η πιο διαδεδομένη για τα παράλληλα συστήματα λόγω της απλότητας υλοποίησης της και του χαμηλού κόστους. Οι περιορισμοί σε αυτό το μοντέλο οφείλονται κυρίως στα όρια του εύρους ζώνης του κοινού bus και του υποσυστήματος της μνήμης.

Τέλος, το dancehall το οποίο δεν βασίζεται σε κοινό bus αλλά σε ένα δίκτυο διασυνδέσεων σημείο-προς-σημείο και όπου η μνήμη χωρίζεται σε λογικά κομμάτια που συνδέονται σε διαφορετικά σημεία του δικτύου διασύνδεσης. Παρόλο που με την πρώτη ματιά δεν φαίνεται και αυτή η προσέγγιση είναι συμμετρική αφού το κόστος προσπέλασης της μνήμης είναι το ίδιο για κάθε επεξεργαστή, όμως το πρόβλημα είναι ότι η κύρια μνήμη είναι πολύ μακριά απο τους επεξεργαστές, αφού θα πρέπει να γίνουν πολλά ΄΄hops΄΄ μέσα στο δίκτυο διασύνδεσης για την προσπέλαση της κύριας μνήμης.

Σε όλες τις παραπάνω περιπτώσεις οι κρυφές μνήμες παίζουν τον βασικότερο ρόλο στη μείωση του μέσου χρόνου προσπέλασης δεδομένων και στη μείωση των απαιτήσεων εύρους ζώνης κάθε επεξεργαστή για το κοινό bus και υποσύστημα μνήμης. Η μείωση της απαίτησης για εύρος ζώνης στο bus οφείλεται στο ότι οι

προσπελάσεις δεδομένων του κάθε επεξεργαστή που εξυπηρετούνται στην κρυφή μνήμη δεν χρειάζεται να περάσουν απο το bus.

Σε όλες τις προσεγγίσεις εκτός απο την πρώτη κάθε επεξεργαστής διαθέτει ένα ή περισσότερα επίπεδα κρυφής μνήμης *ιδιωτικής*, που μπορεί να προσπελαστεί δηλαδή μόνο από τον συγκεκριμένο επεξεργαστή που συνδέεται με αυτή. Το πρόβλημα που δημιουργείται από αυτό το γεγονός είναι ότι, όταν είναι παρόντα αντίγραφα του ίδιου μπλοκ μνήμης στις κρυφές μνήμες δύο ή περισσοτέρων επεξεργαστών, υπάρχει ο κίνδυνος αν ο ένας επεξεργαστής γράψει το μπλοκ μνήμης, οι υπόλοιποι να συνεχίσουν να βλέπουν το παλιό αντίγραφο που έχουν στις κρυφές τους μνήμες αν δεν γίνει ο κατάλληλος χειρισμός της κατάστασης. Το πρόβλημα αυτό είναι γνωστό ως το ***πρόβλημα συνοχής της κρυφής μνήμης*** **(cache coherence)**.

Στην εργασία αυτή θα ασχοληθούμε με την πιο δημοφιλή περίπτωση SMP αρχιτεκτονικής, αυτής με το κοινό bus και θα αναλύσουμε τον τρόπο αντιμετώπισης του προβλήματος γενικά αλλά και το πως αυτή η τεχνική μπορεί να υλοποιηθεί σε ένα soft-core επεξεργαστή όπως είναι ο LEON2.

*2.3. Πρόβλημα Συνοχής Κρυφής Μνήμης (**Cache coherence**)*

Σε μια αρχιτεκτονική SMP υπάρχουν στις κρυφές μνήμες δύο είδη δεδομένων. Τα κοινά δεδομένα, που είναι προσπελάσιμα από όλους τους επεξεργαστές και ιδιωτικά δεδομένα, που είναι προσπελάσιμα μόνο απο τους επεξεργαστές στους οποίους ανήκει η κρυφή μνήμη. Για την επικοινωνία των επεξεργαστών χρησιμοποιούνται τα κοινά δεδομένα, τα οποία και προκαλούν το πρόβλημα της συνοχής της κρυφής μνήμης(*cache coherence*), ενώ για τα ιδιωτικά δεδομένα αντιμετωπίζονται όπως θα αντιμετωπίζονταν σε έναν απλό μονοεπεξεργαστή χωρίς καμιά επιπλοκή.

Για να γίνει καλύτερα αντιληπτό το πρόβλημα, θα πρέπει να σκεφτούμε τις ιδιότητες μιας μνήμης. Η βασική ιδιότητα είναι η μνήμη να επιστρέφει πάντα την τελευταία-πιο πρόσφατη τιμή που έχει γραφεί σε μια διεύθυνσή της. Έτσι, όταν έχουμε μια κοινή μνήμη για περισσότερους του ενός επεξεργαστή δημιουργείται ο κίνδυνος διαφορετικοί επεξεργαστές να βλέπουν διαφορετικά δεδομένα για το ίδιο μπλοκ μνήμης και άρα κάποιοι να χρησιμοποιούν την τελευταία τιμή ενώ

άλλοι ένα παλιό αντίγραφο, γεγονός που θα οδηγήσει προφανώς σε λανθασμένες εκτελέσεις.

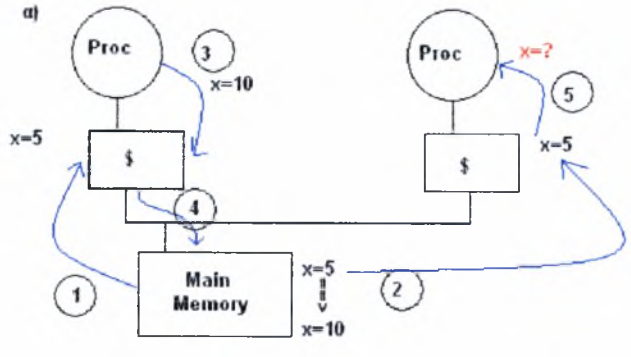
Ας δούμε ένα <u>σενάριο</u> εκτέλεσης:

Έστω δύο επεξεργαστές όπως στην *εικόνα 6*, αρχικά ο επεξεργαστής P1 διαβάζει από την κύρια μνήμη το μπλοκ που περιέχει την μεταβλητή x με τιμή 5 και την φορτώνει στην κρυφή της μνήμη. Ακολούθως διαβάζει την x=5 και ο P2 και την φορτώνει στην δική του κρυφή μνήμη. Έστω τώρα ο P1 γράφει την x και αλλάζει την τιμή της από 5→10. Άρα τώρα η τιμή της x στην κρυφή μνήμη της P1 είναι 10. Αν η κρυφή μνήμη ακολουθεί *write through(σχήμα4.α)* πολιτική, τότε γίνεται και η ενημέρωση της κύριας μνήμης για την αλλαγή άμεσα και άρα και εκεί η x=10. Αν τώρα ο P2 διαβάσει την x, θα διαβάσει τα παλιά δεδομένα που είχε φορτώσει πριν την αλλαγή, δηλαδή ότι x=5. Αυτό είναι μη αποδεκτό σενάριο και αποτελεί τον ορισμό του προβλήματος συνοχής της μνήμης.

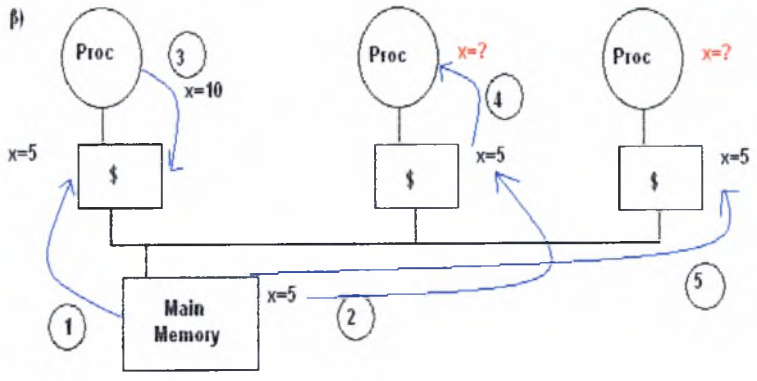
Το ίδιο πρόβλημα φυσικά ισχύει και για κρυφές μνήμες *write back(σχήμα4.β)* και μάλιστα είναι πιο έντονο, αφού για κάθε εγγραφή που γίνεται στην κρυφή μνήμη δεν ενημερώνεται άμεσα και η κύρια μνήμη, αλλά αυτό συμβαίνει αργότερα, όταν το μπλοκ πρέπει να αντικατασταθεί. Έστω τώρα ότι έχουμε και έναν P3 ο οποίος διαβάζει την x μετά την αλλαγή από την P1. Αφού η κρυφή μνήμη είναι *write back*, ο P1 δεν θα ενημερώσει την κύρια μνήμη για την αλλαγή πριν γίνει κάποιο cache miss και χρειαστεί να αντικατασταθεί το μπλοκ της x. Άρα τώρα και ο P3 θα διαβάσει τα παλιά δεδομένα, παρόλο που διάβασε την x μετά την εγγραφή απο τον P1.

σχήμα 4:

<u>Write-through cache</u>:



<u>Write-back cache:</u>



Το παραπάνω σενάριο δείχνει το πως μπορεί να παραβιαστεί η βασικότερη ιδιότητα ορθής λειτουργίας της μνήμης σε μια οργάνωση πολυεπεξεργαστή, να μην επιστρέφει δηλαδή η κρυφή μνήμη την πιο ενημερωμένη τιμή μιας διεύθυνσης μνήμης.

Στην πραγματικότητα όμως κάτι ανάλογο μπορεί να συμβεί και σε μονοεπεξεργαστές, όταν εκτελούνται λειτουργίες I/O. Οι περισσότερες από αυτές εκτελούνται από συσκευές DMA(Direct Memory Access) που μεταφέρουν (διαβάζουν ή γράφουν) δεδομένα μεταξύ της μνήμης και των περιφερειακών χωρίς την ανάμιξη του επεξεργαστή. Όταν οι συσκευές DMA γράφουν μια θέση μνήμης, ο επεξεργαστής μπορεί να συνεχίσει να βλέπει την παλιά τιμή αν είχε φορτώσει ήδη αυτή την θέση μνήμης στην κρυφή του μνήμη, ενώ αυτή έχει αλλάξει στην κύρια μνήμη. Ακόμα, αν οι κρυφές μνήμες είναι write back τότε μπορεί μια συσκευή DMA να διαβάσει ένα παλιό αντίγραφο, το οποίο έχει γραφεί από τον επεξεργαστή στην κρυφή μνήμη και δεν έχει ενημερώσει ακόμη την κύρια μνήμη. Μια λύση σε αυτό το ζήτημα είναι κομμάτια της κύριας μνήμης, τα οποία χρησιμοποιούνται από συσκευές DMA να χαρακτηρίζονται ως uncacheable, δηλαδή να μην φορτώνονται στην κρυφή μνήμη του επεξεργαστή όταν προσπελαστούν. Αυτή όπως και άλλες λύσεις που έχουν δοθεί δεν είναι οι καλύτερες, γι' αυτό οι τεχνικές που επιλύουν το παρεμφερές πρόβλημα της συνοχής της κρυφής μνήμης για πολυεπεξεργαστές, χρησιμοποιούνται για την επίλυση και αυτού του προβλήματος. Τεχνικές όπως αυτή που χρησιμοποιήσαμε για τις προσπελάσεις από DMA όπου οι κοινές μεταβλητές θα είναι uncacheable προφανώς δεν μπορεί να λειτουργήσουν στους πολυεπεξεργαστές, για όλα τα δεδομένα, καθώς εκμηδενίζουν τις δυνατότητες επικοινωνίας των επεξεργαστών μεταξύ τους. Ουσιαστικά, αν υπάρχουν uncacheable κομμάτια μνήμης για την λύση του προβλήματος συνοχής για τους διαφορετικούς επεξεργαστές, το μόνο

που καταφέρνουμε είναι να επιβραδύνουμε τόσο πολύ την επικοινωνία των επεξεργαστών ώστε στην ουσία αφαιρούμε την έννοια των κοινών μεταβλητών και να καταστρέφουμε την λειτουργικότητα και τα οφέλη που προσπαθούμε να επιτύχουμε με την χρήση πολυεπεξεργαστών.

Πριν μελετήσουμε τις τεχνικές επίλυσης του προβλήματος συνοχής θα πρέπει να ορίσουμε ξανά το πρόβλημα, αυτή τη φορά με ορθότερο τρόπο και μεγαλύτερο βάθος.

Αναλυτικότερα, το σύστημα μνημών των πολυεπεξεργαστών διατηρεί την *συνοχή* του, εάν τα αποτελέσματα κάθε εκτέλεσης ενός προγράμματος είναι τέτοια ώστε, για κάθε θέση μνήμης, να είναι δυνατό να κατασκευαστεί ένα υποθετικό σενάριο σειριακής εκτέλεσης όλων των λειτουργιών γι' αυτή την θέση, δηλαδή να τοποθετηθούν όλες οι εγγραφές και οι αναγνώσεις όλων των επεξεργαστών σε απόλυτη σειρά, που να είναι *συνεπές* με τα αποτελέσματα της εκτέλεσης και να ισχύει ότι:

Α) Λειτουργίες που δρομολογούνται από οποιαδήποτε συγκεκριμένο επεξεργαστή εμφανίζονται στο σύστημα μνημών με την απόλυτη σειρά με την οποία και δρομολογήθηκαν.

Β) Η τιμή που επιστρέφεται από κάθε λειτουργία ανάγνωσης είναι η τιμή που γράφτηκε από την τελευταία λειτουργία εγγραφής στην συγκεκριμένη θέση μνήμης σε απόλυτη σειρά.

Στον ορισμό του coherence δύο ιδιότητες, που πρέπει να τηρούνται,- υποννοούνται: α) *Διάδοση εγγραφής*, β) *Σειριοποίηση εγγραφής* .

Διάδοση εγγραφής σημαίνει ότι οι εγγραφές γίνονται ορατές στους υπόλοιπους επεξεργαστές. *Σειριοποίηση εγγραφής* σημαίνει ότι όλες οι εγγραφές σε μια θέση μνήμης από τον ίδιο η διαφορετικούς επεξεργαστές, γίνονται ορατές με την ίδια σειρά από όλους τους επεξεργαστές.

Τα αποτελέσματα ενός προγράμματος μπορούν να θεωρηθούν ότι είναι οι τιμές που επιστρέφονται από μία λειτουργία ανάγνωσης. Από τα αποτελέσματα, δεν μπορούμε να προσδιορίσουμε την σειρά με την οποία εκτελέστηκαν στην πραγματικότητα οι λειτουργίες από τον υπολογιστή, αλλά μπορούμε να προσδιορίσουμε την σειρά με την οποία εμφανίζονται ότι εκτελούνται. Αυτό

βέβαια είναι ακριβώς αυτό που μας ενδιαφέρει, αφού αυτό ''βλέπουν'' και οι επεξεργαστές.

*2.4. Συνέπεια Μνήμης (Memory **Consistency**)*

Θα κάνουμε μία παρένθεση πριν προχωρήσουμε σε πιο αναλυτικά θέματα πάνω στη συνοχή, για να αναλύσουμε την έννοια της συνέπειας που είναι πολύ στενά συνδεδεμένη με τη συνοχή[2][21].

Η συνοχή, είναι η ιδιότητα που πρέπει να έχει ένα πολυεπεξεργαστικό σύστημα για να μπορεί κάποιος επεξεργαστής που εκτελεί μια λειτουργία ανάγνωσης να διαβάζει την τελευταία ενημέρωση της τιμής και όχι κάποια παλιά τιμή. Εξασφαλίζει δηλαδή ότι πολλοί επεξεργαστές βλέπουν μια συνεκτική εικόνα της μνήμης. Παρόλα αυτά η ιδιότητα αυτή δεν διασφαλίζει το πότε μια εγγραφή είναι ορατή, δηλαδή σε παράλληλα προγράμματα πρέπει να μπορούμε να διασφαλίσουμε ότι μια ανάγνωση επιστρέφει ως αποτέλεσμα την τιμή μιας συγκεκριμένης εγγραφής. Αυτό επιτυγχάνεται με κάποιο είδος συγχρονισμού και ονομάζεται συνέπεια(***consistency***).

Για να γίνει πιο κατανοητή η έννοια της συνέπειας θα μελετήσουμε ένα απλό παράδειγμα:

Έστω ο κώδικας	P1	P2
	Data = 5;	while (Flag = = 0) {}
	Flag = 1;	X = Data;

Προφανώς ο προγραμματιστής θα περιμένει ότι πρώτα θα εκτελεστεί η ανάθεση στο Data και μέχρι ο P1 να κάνει 1 το Flag, ο P2 θα έχει κολλήσει στο βρόχο while. Κανένας δεν μπορεί να εγγυηθεί ότι το ο P1 θα εκτελέσει το Flag = 1 μετά το Data = 5, αφού σε επίπεδο υλικού η πρώτη ανάθεση μπορεί για διάφορους λόγους να διαρκέσει περισσότερους κύκλους μηχανής από την δεύτερη και έτσι η τιμή της να γίνει διαθέσιμη μετά την τιμή της επόμενης ανάθεσης, και άρα υπάρχει περίπτωση ο P2 να μπορέσει να περάσει το while πριν εκτελεστεί το Data = 5 από τον P1, και έτσι το αποτέλεσμα του X να μην είναι 5. Το να μην εκτελεστούν οι εντολές με διαφορετική σειρά από αυτή που περιμένουμε προγραμματιστικά δεν μπορεί να το εγγυηθεί η συνοχή, το οποίο για παράδειγμα διασφαλίζει ότι η νέα τιμή του Data θα γίνει σίγουρα ορατή στο P2, αλλά όχι απαραίτητα πριν την νέα τιμή του Flag. Γι αυτό υπάρχει η έννοια της συνέπειας.

Γενικά, ένα μοντέλο συνέπειας μνήμης για κοινό χώρο διευθύνσεων εισάγει περιορισμούς, που μπορεί να επιτυγχάνονται με μηχανισμούς συγχρονισμού όπως είναι τα barriers, στην σειρά με την οποία οι λειτουργίες μνήμης γίνονται ορατές στον κάθε επεξεργαστή, χωρίς να εισάγει μια απόλυτη σειρά σε όλες τις εντολές του προγράμματος. Αυτό περιλαμβάνει λειτουργίες στην ίδια διεύθυνση ή διαφορετικές διευθύνσεις και από το ίδιο η διαφορετικό process, δηλαδή η έννοια *συνέπειας* συμπληρώνει αυτή της συνοχής. Με λίγα λόγια η συνέπεια μπορεί να εξασφαλίσει κάποιου είδους σειρά στο πρόγραμμα ώστε ο προγραμματιστής να πάρει αναμενόμενα αποτελέσματα.

Ένα από τα πιο διαδεδομένα μοντέλα *συνέπειας* είναι το μοντέλο ακολουθιακής συνέπειας(***sequential consistency***). Ένας διαισθητικός ορισμός γι αυτό το μοντέλο έχει δοθεί από τον Lamport (1976):

Ένα πολυεπεξεργαστικό σύστημα είναι ακολουθιακά συνεπές αν το αποτέλεσμα οποιασδήποτε εκτέλεσης είναι το ίδιο όπως αν οι λειτουργίες όλων των επεξεργαστών εκτελούνταν σε ορισμένη ακολουθιακή σειρά, και οι λειτουργίες του κάθε ξεχωριστού επεξεργαστή προκύπτουν σε αυτήν την ακολουθία με την σειρά που καθορίζεται από το πρόγραμμα του.

Αφού είδαμε τον ορισμό της ακολουθιακής συνέπειας θα πρέπει να ορίσουμε κάποιες επαρκείς συνθήκες που θα εγγυώνται την διατήρηση της συνέπειας, είτε έχουμε συμμετρικούς πολυεπεξεργαστές, είτε κατανεμημένους με συνοχή ή χωρίς. Τρεις είναι οι βασικές συνθήκες:

1)Κάθε διεργασία δρομολογεί λειτουργίες μνήμης σύμφωνα με την σειρά του προγράμματος.

2)Αν δρομολογηθεί λειτουργία εγγραφής, τότε περιμένουμε την ολοκλήρωση της εγγραφής πριν την δρομολόγηση νέας εντολής.

3)Αφού δρομολογηθεί μια λειτουργία ανάγνωσης, η διεργασία που την δρομολόγησε σταματάει και περιμένει την ολοκλήρωση της ανάγνωσης, δηλαδή περιμένει μέχρι να γίνει εμφανής η ολοκλήρωση της εγγραφή της οποίας την τιμή θα επιστρέψει η συγκεκριμένη ανάγνωση σε όλους τους επεξεργαστές του συστήματος.

Η τρίτη συνθήκη είναι αυτή που διατηρεί την ατομικότητα εγγραφής. Η υλοποίηση της είναι πολύπλοκη, αφού κάθε ανάγνωση πρέπει να περιμένει κάθε εγγραφή που προηγείται λογικά, να ολοκληρωθεί και να γίνει καθολικά ορατή. Η τήρηση αυτών των συνθηκών επιβάλλει μεγάλη σειριοποιήση, όμως δεν είναι

απαραίτητο για την διατήρηση της ακολουθιακής συνέπειας να ακολουθούνται πλήρως όλες οι συνθήκες.

Επιπλέον για την διατήρηση της ακολουθιακής συνέπειας δεν πρέπει να γίνονται κάποιες από τις κλασσικές βελτιώσεις των μεταφραστών στον κώδικα, όπως η αναδιάταξη κώδικα, η δέσμευση καταχωρητών, οι μετατροπές στους βρόχους.

Για πολυεπεξεργαστές που βασίζονται για την επικοινωνία τους σε bus, η σειριοποίηση που παρέχει το bus βοηθάει στον να διατηρήσουμε την ακολουθιακή συνέπεια. Έτσι για παράδειγμα όταν συμβεί μία ανάγνωση σε μια διεύθυνση, τότε είναι εγγυημένο ότι κάποια εγγραφή στην ίδια διεύθυνση που έχει δρομολογηθεί πριν την ανάγνωση θα έχει ολοκληρωθεί, αφού θα έχει χρησιμοποιήσει το bus νωρίτερα, γεγονός που εγγυάται ατομικότητα εγγραφής.

*2.5. Υποκλέπτοντα Πρωτόκολλα (**Snoop-Based Protocols**)*

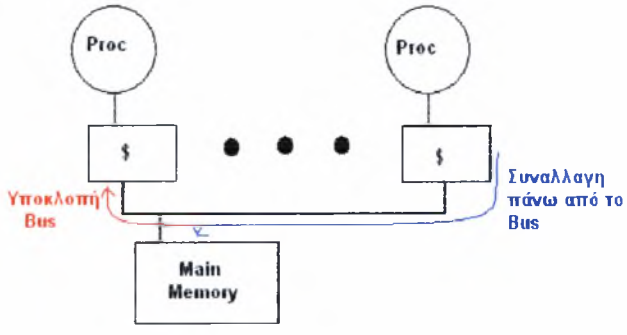
Επιστρέφουμε τώρα για να εξετάσουμε πως μπορούμε να διατηρήσουμε τη συνοχή κρυφής μνήμης χρησιμοποιώντας την τεχνική της υποκλοπής της κοινής αρτηρίας(***Bus Snooping***).

Το bus είναι ένα σύνολο από καλώδια που ενώνει τις διαφορετικές συσκευές ενός μικροεπεξεργαστή. Κάθε μια από τις συσκευές μπορεί να παρακολουθεί κάθε συναλλαγή που εκτελείται στο bus, όπου με τον όρο συναλλαγή εννοούμε κάθε λειτουργία ανάγνωσης ή εγγραφής που για την εκτέλεσή της δεσμεύει και χρησιμοποιεί το bus. Όταν ο επεξεργαστής δρομολογεί μια αίτηση στην κρυφή του μνήμη, ο ελεγκτής της κρυφής μνήμης εξετάζει την κατάσταση της κρυφής μνήμης και ενεργεί ανάλογα για να ικανοποιήσει την αίτηση που δέχθηκε. Για την διατήρηση της συνοχής λοιπόν θέλουμε να μπορούν όλοι οι ελεγκτές κρυφής μνήμης να υποκλέπτουν (***snoop***) το bus (σχήμα 6) και να παρακολουθούν τις συναλλαγές που γίνονται.

Ο ελεγκτής κρυφής μνήμης μπορεί να αποφασίσει να δράσει,να εκτελέσει κάποια λειτουργία, αν υποκλέψει συναλλαγή στο bus που να τον αφορά, δηλαδή αν η συναλλαγή αφορά κάποιο μπλοκ μνήμης του οποίου αντίγραφο έχει αυτός στην κρυφή μνήμη. Για παράδειγμα αν ο P1 έχει αντίγραφο μιας θέσης μνήμης στην κρυφή μνήμη και ο P2 γράφει την θέση μνήμης, τότε ο P1 βλέποντας την

εγγραφή μπορεί να δράσει ακυρώνοντας ή ανανεώνοντας το αντίγραφο στην κρυφή του μνήμη. Η συναλλαγές αυτές γίνονται σε επίπεδο μπλοκ κρυφής μνήμης, δηλαδή είτε ένα ολόκληρο μπλοκ μνήμης είναι σε έγκυρη κατάσταση ή είναι σε άκυρη κατάσταση, παρόλο που μπορεί μόνο μια λέξη αυτού του μπλοκ να είναι άκυρη.

σχήμα6: **Bus Snooping**



Οι βασικές ιδιότητες για την υποστήριξη της συνοχής είναι:

1)Οι συναλλαγές που εμφανίζονται στο bus είναι ορατές σε όλους τους ελεγκτές κρυφής μνήμης.

2)Είναι ορατές στους ελεγκτές κρυφής μνήμης με την ίδια σειρά, αυτήν με την οποία εμφανίστηκαν στο bus.

Γενικά, ένα πρωτόκολλο συνοχής μνήμης πρέπει να διασφαλίζει ότι όλες οι συναλλαγές που είναι απαραίτητες εμφανίζονται στο bus, και όλοι οι ελεγκτές εκτελούν τις ανάλογες ενέργειες που χρειάζονται.

Σε ένα τέτοιο πρωτόκολλο, κάθε λειτουργία εγγραφής προκαλεί μια συναλλαγή εγγραφής να εμφανιστεί στο bus, ώστε κάθε ελεγκτής κρυφής μνήμης να μπορεί να δει την λειτουργία εγγραφής. Αν μια *snooping* κρυφή μνήμη έχει αντίγραφο του μπλοκ μνήμης, τότε θα πρέπει είτε να ακυρώσει είτε να ανανεώσει το μπλοκ αυτό. Πρωτόκολλα τα οποία ακυρώνουν σε μια τέτοια περίπτωση το αντίγραφο ονομάζονται *πρωτόκολλα ακύρωσης*(***invalidation-based protocols***), ενώ τα πρωτόκολλα που απλώς ανανεώνουν το μπλοκ με την τελευταία τιμή ονομάζονται *πρωτόκολλα ενημέρωσης*(***update-based protocols***). Και στα δύο πρωτόκολλα ο επεξεργαστής την επόμενη φορά που θα διαβάσει το συγκεκριμένο μπλοκ, θα δει την τελευταία ενημερωμένη τιμή είτε μέσω αστοχίας

κρυφής μνήμης για πρωτόκολλο ακύρωσης είτε μέσω ενημέρωσης για πρωτόκολλο ενημέρωσης.

Ένα πρωτόκολλο συνοχής συνδέεται στενά με δύο έννοιες: α)τις συναλλαγές στο bus β)το διάγραμμα μετάβασης καταστάσεων του κάθε μπλοκ κρυφής μνήμης. Οι συναλλαγές στο bus αποτελούνται βασικά από τρεις φάσεις:

α)επιλογή αίτησης προς εξυπηρέτηση(διαιτησία - ***arbitration***)

β)εντολή/διεύθυνση

γ)δεδομένα

Στην φάση *arbitration,* σε μονοεπεξεργαστή, οι συσκευές που πρόκειται να ξεκινήσουν μια συναλλαγή εκδίδουν μια αίτηση για να πάρουν το bus, και το bus μέσω του διαιτητή επιλέγει μια από όλες τις αιτήσεις, σύμφωνα με τα κριτήρια που έχουν οριστεί, και απαντά σε μια από αυτές με το σήμα κατοχύρωσης του bus (**bus grant**). Με το που λάβει μια συσκευή το σήμα *grant* τοποθετεί την εντολή της, εγγραφή ή ανάγνωση, και την σχετική διεύθυνση στο bus. Όλες οι συσκευές βλέπουν την διεύθυνση που τοποθετήθηκε και μια από αυτές αναγνωρίζει ότι είναι αυτή υπεύθυνη, έχει, αυτή τη διεύθυνση. Αν έχω συναλλαγή ανάγνωσης, τη φάση αυτή ακολουθεί η φάση μεταφοράς δεδομένων. Αν τώρα έχω συναλλαγή εγγραφής για τα περισσότερα είδη bus ισχύει ότι η συσκευή που θα απαντήσει με δεδομένα μπορεί να θέσει ένα σήμα ώστε να καθυστερήσει η μεταφορά δεδομένων μέχρι να είναι η συσκευή έτοιμη για την μεταφορά Αυτό το σήμα συνήθως είναι ένα wired-OR κατά μήκος όλου του επεξεργαστή, το οποίο τίθεται σε λογικό 1 αν το θέσει οποιαδήποτε από τις συσκευές. Η συσκευή που έκανε την αίτηση δεν χρειάζεται να ξέρει ποια συσκευή θα απαντήσει, αλλά ότι υπάρχει τέτοια συσκευή και πότε είναι έτοιμη για την συναλλαγή.

Για την δεύτερη έννοια τώρα, γνωρίζουμε ότι κάθε μπλοκ κρυφής μνήμης έχει μια κατάσταση συσχετισμένη με αυτό, εκτός από την ετικέτα(**tag**) και τα δεδομένα, που καταδεικνύει αν το μπλοκ είναι για παράδειγμα έγκυρο(**valid**), άκυρο(**invalid**) ή τροποποιημένο(**dirty**). Η πολιτική της κρυφής μνήμης καθορίζεται από το διάγραμμα μετάβασης καταστάσεων του μπλοκ κρυφής μνήμης, το οποίο είναι μια μηχανή πεπερασμένων καταστάσεων που διευκρινίζει πως αλλάζει η κατάσταση ενός μπλοκ ανάλογα με την ενέργεια που εκτελείται. Οι μεταβάσεις για ένα μπλοκ συμβαίνουν και με την πρόσβαση στο συγκεκριμένο μπλοκ αλλά και σε μια διεύθυνση που αντικατοπτρίζεται στο μπλοκ αυτό.

Ουσιαστικά μόνο τα μπλοκ τα οποία είναι στην κρυφή μνήμη έχουν πληροφορίες κατάστασης σε επίπεδο υλικού. Παρόλα αυτά λογικά και τα μπλοκ τα οποία δεν είναι στην κρυφή μνήμη μπορεί αν θεωρηθούν ότι είναι σε μια ειδική κατάσταση invalid.

Ας δούμε πως μπορεί να υλοποιηθεί μια μορφή συνοχής σε ένα μονοεπεξεργαστή όπου έχουμε write through πολιτική για την κρυφή μνήμη και write no-allocate, το οποίο σημαίνει ότι η κρυφή μνήμη όταν δέχεται από τον επεξεργαστή μια αίτηση για εγγραφή δεν χρειάζεται να φορτώσει την διεύθυνση αλλά μπορεί να γράψει απευθείας στην κύρια μνήμη σε περίπτωση που δεν είναι το μπλοκ στην κρυφή μνήμη. Για ένα τέτοιο επεξεργαστή θα χρειαζόμασταν ένα FSM δύο καταστάσεων, valid και invalid και λειτουργεί ως εξής: Όταν ο επεξεργαστής θέλει να κάνει μια ανάγνωση και δεν έχει το μπλοκ στην κρυφή μνήμη, παράγεται μία συναλλαγή στο bus για να φορτωθεί το κατάλληλο μπλοκ από την κύρια μνήμη και το μπλοκ χαρακτηρίζεται valid. Για μια εγγραφή παράγεται στο bus συναλλαγή για την ανανέωση της μνήμης με την καινούρια τιμή και εγγράφεται και το μπλοκ στην κρυφή μνήμη αν είναι σε valid κατάσταση. Οι εγγραφές δεν αλλάζουν την κατάσταση του μπλοκ.

Αντίστοιχα τώρα σε ένα πολυεπεξεργαστή κάθε μπλοκ έχει μια κατάσταση σε κάθε κρυφή μνήμη και οι καταστάσεις αυτές αλλάζουν σύμφωνα με το FSM. Σε αυτή την περίπτωση μπορούμε να σκεφτούμε ότι έχουμε ένα διάνυσμα από ν καταστάσεις για κάθε μπλοκ, όπου το ν είναι ο αριθμός των επεξεργαστών και άρα των κρυφών μνημών που υπάρχουν. Για την διαχείριση αυτών των καταστάσεων έχουμε και ω FSM που υλοποιούνται στους ν ελεγκτές των κρυφών μνημών. Προφανώς, το FSM αυτό είναι το ίδιο για όλα τα μπλοκ, όμως το η κατάσταση ενός μπλοκ για διαφορετικές κρυφές μνήμες είναι διαφορετική.

Ας δούμε τώρα κάποια πράγματα για το *snooping* πρωτόκολλο. Για την διατήρηση συνοχής με την βοήθεια *snooping* πρωτοκόλλου, κάθε ελεγκτής κρυφής μνήμης δέχεται ως είσοδο αιτήσεις από τον επεξεργαστή αλλά και πληροφορίες από τις συναλλαγές που διαδραματίζονται στο bus από τις υπόλοιπες κρυφές μνήμες. Ανάλογα με την είσοδο που δέχεται, εκτελεί τις ανάλογες ενέργειες. Για τις αιτήσεις του επεξεργαστή μπορεί να ξεκινήσει μια συναλλαγή στο bus για να πάρει δεδομένα από την μνήμη ή να δώσει κατευθείαν δεδομένα στον επεξεργαστή αν τα έχει ήδη στην κρυφή του μνήμη. Για τις συναλλαγές που βλέπει στο bus και έχουν ξεκινήσει από άλλους ελεγκτές μπορεί

να χρειαστεί να ανανεώσει την κατάσταση ενός μπλοκ η να παρέμβει στην συναλλαγή παρέχοντας δεδομένα για να ολοκληρωθεί ή και τα δύο.

Ουσιαστικά το snooping πρωτόκολλο είναι ένας αλγόριθμος κατανεμημένος που αποτελείται από: α) ένα σύνολο καταστάσεων που σχετίζονται με το κάθε μπλοκ κρυφής μνήμης, β) το FSM, το οποίο ανάλογα με την τρέχουσα κατάσταση την ενέργεια που βλέπει ο ελεγκτής στο bus ή την αίτηση του επεξεργαστή κάνει τις ανάλογες μεταβάσεις και δίνει μια νέα κατάσταση για το μπλοκ, γ)τις ενέργειες που τελικά γίνονται και σχετίζονται με την κατάσταση του κάθε μπλοκ.

2.5.1. Υποκλέπτον Πρωτόκολλο για Write Through Caches

Τώρα, έχοντας ανατρέξει στις βασικές έννοιες θα περάσουμε στην ανάλυση ενός απλού πρωτοκόλλου για συνεκτικές write through και write no-allocate κρυφές μνήμες. Κάθε μπλοκ έχει δύο καταστάσεις την valid και την invalid (σχήμα 8).

Σημασιολογία καταστάσεων:

Invalid: σημαίνει ότι είτε το μπλοκ δεν βρίσκεται στην κρυφή μνήμη είτε βρίσκεται στην κρυφή μνήμη και δεν είναι έγκυρο.

Valid: σημαίνει ότι το μπλοκ βρίσκεται σε μία ή περισσότερες από μία κρυφές μνήμες σε ενημερωμένη κατάσταση και φυσικά και η κύρια μνήμη έχει την τελευταία τιμή αφού πρόκειται για write-through πολιτική.

Οι επεξεργαστές εκδίδουν δύο είδη αιτήσεων: αναγνώσεις(**PrRd**) και εγγραφές(**PrWr**). Η ανάγνωση ή η εγγραφή μπορεί να γίνει σε ένα μπλοκ μνήμης, το οποίο υπάρχει στην κρυφή μνήμη ή που δεν υπάρχει και άρα να πρέπει να φορτωθεί από την κύρια μνήμη

Από την μεριά του bus παράγονται δύο είδη συναλλαγών.

Ανάγνωση bus(**BusRd**): παράγεται από μια αίτηση *PrRd* του επεξεργαστή που αστοχεί στην κρυφή μνήμη. Ο ελεγκτής της κρυφής μνήμης τοποθετεί την διεύθυνση στο bus και περιμένει μια απάντηση από την κύρια μνήμη. Το μπλοκ που φορτώνεται μπορεί να τροποποιηθεί ή απλώς να διαβαστεί.

Εγγραφή στο bus(**BusWr**): παράγεται από μια αίτηση PrWr του επεξεργαστή που θέλει να γράψει ένα μπλοκ είτε αυτό υπάρχει στην κρυφή μνήμη και άρα έχω ευστοχία εγγραφής είτε όχι και άρα έχω αστοχία εγγραφής και η εγγραφή γίνεται κατ' ευθείαν στην κύρια μνήμη.

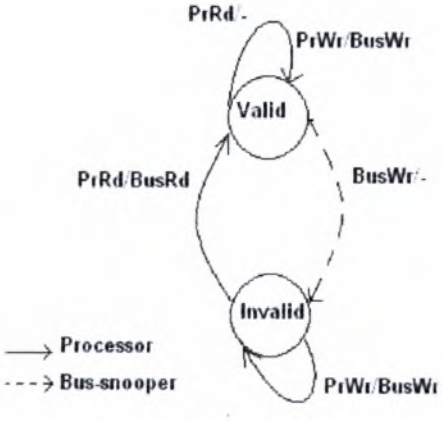
Η κατάσταση του κάθε μπλοκ αλλάζει κάθε φορά που ο ελεγκτής βλέπει κάποια ενέργεια στο bus ή κάποια αίτηση από τον επεξεργαστή. Δηλαδή στο συγκεκριμένο σχήμα: έστω ότι ένα μπλοκ δεν υπάρχει στην κρυφή μνήμη του επεξεργαστή (άρα κατά σύμβαση μπορεί να ξεκινήσει από την invalid κατάσταση αν και ουσιαστικά υπάρχει μόνο στην κύρια μνήμη) και ο επεξεργαστής αιτείται να το διαβάσει(**PrRd**) και παράγει γι αυτό το λόγο μια συναλλαγή ανάγνωσης στο bus(**BusRd**). Το μπλοκ τότε φορτώνεται από την κύρια μνήμη στην κρυφή και η κατάσταση του από invalid αλλάζει σε valid. Αν τώρα από αυτή την κατάσταση ο επεξεργαστής θέλει να διαβάσει ξανά (**PrRd**) το ίδιο μπλοκ, μπορεί να το κάνει χωρίς να δημιουργήσει καμιά συναλλαγή στο bus για να ενημερώσει τους υπόλοιπους επεξεργαστές, αφού έχει το τελευταίο αντίγραφο, άρα και το πιο ενημερωμένο. Από την ίδια κατάσταση αν θέλει να γράψει(**PrWr**) το μπλοκ αυτό και άρα θα έχουμε ευστοχία εγγραφής, τότε πρέπει να παράγει και την κατάλληλη συναλλαγή εγγραφής στο bus(**BusWr**) για να γίνει ορατό στους υπόλοιπους επεξεργαστές ότι κάποιος προτίθεται να γράψει το συγκεκριμένο μπλοκ στη μνήμη και άρα η τιμή του δεν θα είναι πλέον η ίδια. Ο ελεγκτής του επεξεργαστή που κάνει την εγγραφή δεν χρειάζεται όμως να μεταβεί σε άλλη κατάσταση αφού θα έχει και πάλι το μπλοκ στην κρυφή του μνήμη σε valid κατάσταση και άρα μετά την εγγραφή θα είναι και πάλι το τελευταίο και πιο ενημερωμένο αντίγραφο. Θα πρέπει μόνο σύμφωνα με την write through πολιτική να ενημερώσει την κύρια μνήμη για την νέα τιμή. Αν τώρα όμως κατά την διάρκεια που ένας επεξεργαστής έχει το μπλοκ στην κρυφή μνήμη, βρεθεί κάποιος άλλος επεξεργαστής που θέλει να γράψει σε αυτό το συγκεκριμένο μπλοκ τότε θα δει από το bus μία συναλλαγή *BusWr* και θα αλλάξει την κατάσταση από valid σε invalid ακυρώνοντας έτσι το τοπικό αντίγραφο, καταλαβαίνοντας ότι το μπλοκ έχει γραφτεί από άλλον επεξεργαστή και δεν θα είναι πια αυτή η κρυφή μνήμη που έχει την πιο ενημερωμένη τιμή.

Επίσης αν το μπλοκ είναι σε invalid κατάσταση και κάποιος επεξεργαστής θέλει να το γράψει, το μπλοκ παραμένει στην invalid κατάσταση εξαιτίας του write no-allocate, δηλαδή μπορεί να γραφτεί ένα μπλοκ χωρίς να φορτωθεί στην κρυφή μνήμη αλλά γράφοντας κατ' ευθείαν στην κύρια μνήμη εξαιτίας του no-allocate. Σε αυτήν την περίπτωση προφανώς και πάλι θα τοποθετηθεί το BusWr σήμα στο bus για να ακυρώσουν όσοι επεξεργαστές έχουν το αντίγραφο του

μπλοκ σε valid κατάσταση. Με λίγα λόγια μπορούν να υπάρχουν πολλοί ταυτόχρονοι αναγνώστες του μπλοκ χωρίς να προκαλούν συναλλαγές στο bus ή ακυρώσεις, όμως μια εγγραφή θα ακυρώσει όλα τα υπόλοιπα αντίγραφα.

Για να δείξουμε πως αυτό το απλό πρωτόκολλο διασφαλίζει τη συνοχή, πρέπει να δείξουμε πως για κάθε εκτέλεση μπορεί να κατασκευαστεί μια καθολική σειρά των λειτουργιών μνήμης για μια διεύθυνση που σέβεται την σειρά προγράμματος(program order) και την σειριοποίηση των εγγραφών. Ας υποθέσουμε γι αυτό ότι το bus είναι ατομικό όπως επίσης και οι λειτουργίες μνήμης, ο επεξεργαστής περιμένει μέχρι η προηγούμενη λειτουργία μνήμης να έχει ολοκληρωθεί πριν την έκδοση της επόμενης λειτουργίας μνήμης, οι κρυφές μνήμες είναι ενός επιπέδου μόνο και τέλος υποθέτουμε ότι οι ακυρώσεις μπλοκ γίνονται και άρα οι εγγραφές ολοκληρώνονται κατά τη διάρκεια των συναλλαγών του bus, και ότι η εξυπηρέτηση των αναγνώσεων και των εγγραφών από την μνήμη γίνεται με την σειρά με την οποία εμφανίζονται στο bus.

σχήμα7 ***Write-through, no-allocate cache***



Στο write through πρωτόκολλο λοιπόν όλες οι εγγραφές εμφανίζονται στο bus και αφού στο bus εξυπηρετείται μία συναλλαγή κάθε φορά, όλες οι εγγραφές σε μία διεύθυνση κατά τη διάρκεια μιας εκτέλεσης σειριοποιούνται σύμφωνα με την σειρά με την οποία εμφανίζονται στο bus, η οποία για αυτό ονομάζεται bus order. Επιπλέον αφού κάθε ελεγκτής κρυφή μνήμης που κάνει snooping ακυρώνει τα μπλοκ κατά τη διάρκεια των συναλλαγών στο bus, συμπεραίνουμε ότι και οι ακυρώσεις εκτελούνται σύμφωνα με το bus order.

Για να εξασφαλίσουμε την σειριοποίηση εγγραφών, πρέπει οι αναγνώσεις που γίνονται από κάθε επεξεργαστή να βλέπουν τις εγγραφές σε bus order. Όμως, οι

αναγνώσεις σε μια διεύθυνση δεν είναι εντελώς σειριοποιημένες, αφού κάθε ανάγνωση μπορεί να γίνεται ανεξάρτητα σε κάθε κρυφή μνήμη και άρα και πολλές αναγνώσεις παράλληλα χωρίς να φαίνεται καμιά ενέργεια στο bus. Μια ανάγνωση παράγει ενέργεια στο bus μόνο όταν πρόκειται για αστοχία ανάγνωσης οπότε και πρέπει να φορτώσει από την κύρια μνήμη δεδομένα. Η ανάγνωση αυτή που θα φανεί στο bus σειριοποιείται όπως και οι εγγραφές και άρα θα διαβάσει την τελευταία τιμή από την πιο πρόσφατη εγγραφή που εμφανίστηκε στο bus, δηλαδή σε bus order. Στην περίπτωση που έχουμε ευστοχία ανάγνωσης και καμιά συναλλαγή δεν δημιουργείται στο bus, μπορούμε να θεωρήσουμε ότι η τιμή που θα διαβαστεί έχει τοποθετηθεί στην κρυφή μνήμη από την πιο πρόσφατη εγγραφή στην συγκεκριμένη διεύθυνση από τον τοπικό επεξεργαστή ή από κάποια αστοχία μνήμης. Και στις δύο περιπτώσεις όμως εμφανίζονται συναλλαγές στο bus, άρα και οι ευστοχίες ανάγνωσης βλέπουν τιμές που παράγονται σύμφωνα με το bus order. Δηλαδή το πρωτόκολλο αυτό ικανοποιεί τις απαιτήσεις για συνοχή με τους περιορισμούς που θέτονται από το bus order άρα και το program order.

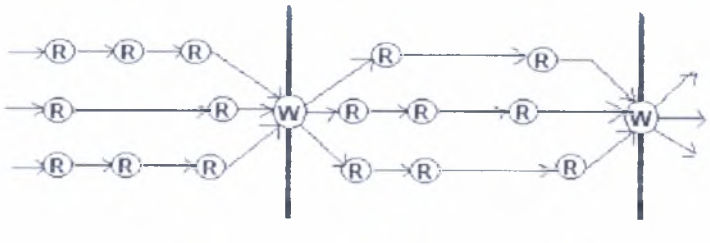
Γενικότερα μπορούμε να κατασκευάσουμε μια καθολική σειρά ώστε να ικανοποιείται η συνοχή αφού το πρωτόκολλο από την φύση του θέτει κάποιες μερικές σειριοποιήσεις όπως :

-Μια λειτουργία μνήμης Β είναι ακόλουθη μιας άλλης λειτουργίας μνήμης Α, αν και οι δύο εκδίδονται από τον ίδιο επεξεργαστή και η Β έπεται της Α σε σειρά προγράμματος.

-Μια ανάγνωση είναι ακόλουθη μιας εγγραφής αν η ανάγνωση δημιουργεί συναλλαγή στο bus που έπεται της συναλλαγής που δημιουργείται από την εγγραφή.

-Μια εγγραφή είναι ακόλουθη μιας εγγραφής ή μιας ανάγνωσης αν η συναλλαγή που δημιουργείται από την εγγραφή έπεται των συναλλαγών που δημιουργούνται από την εγγραφή ή την ανάγνωση.

σχήμα



Οπως φαίνεται και στο σχήμα παραπάνω, όταν διατηρούνται όλες οι μερικές διατάξεις τότε διατηρείται και η καθολική διάταξη. Η εγγραφή δημιουργεί μια ολική σειρά των γεγονότων μεταξύ των οποίων κάθε επεξεργαστής διαβάζει διευθύνσεις σε σειρά προγράμματος. Η εκτέλεση είναι συνεπής με οποιαδήποτε ολική διάταξη πάρουμε.

Παρόλο που διατηρείται όπως είδαμε παραπάνω η συνοχή με ένα πρωτόκολλο write through, υπάρχει ένα πρόβλημα που πηγάζει από την φύση του πρωτοκόλλου και αυτό είναι ότι σε κάθε εγγραφή που γίνεται τοπικά σε κάθε επεξεργαστή πρέπει να ενημερώνεται και η κύρια μνήμη, γεγονός που προκαλεί πολλή κίνηση στο bus, λόγω των συναλλαγών που πρέπει να γίνουν. Το πρόβλημα αυτό επιδεινώνεται σε ένα σύστημα πολυεπεξεργαστών όπου κάθε επεξεργαστής καταναλώνει εύρος ζώνης του κοινού bus για την εγγραφή στην κύρια μνήμη, και το οποίο κάνει προφανώς δύσκολη την κλιμάκωση ενός τέτοιου συστήματος. Γι αυτό οι περισσότεροι επεξεργαστές χρησιμοποιούν write back κρυφές μνήμες.

2.5.2. Υποκλέπτον Πρωτόκολλο για Write Back Caches

Ας δούμε λοιπόν τώρα πως λειτουργεί ένα snooping πρωτόκολλο για write back κρυφές μνήμες που συνδυάζονται με το write-allocate σχήμα, δηλαδή όταν ένας επεξεργαστής θέλει να γράψει ένα μπλοκ πρέπει πρώτα να το φορτώσει στην κρυφή μνήμη αν δεν το έχει και μετά να γίνει η εγγραφή. Καταρχήν, γενικά για τα snooping πρωτόκολλα δεν χρειάζεται να επέμβουμε και να αλλάξουμε τον επεξεργαστή παρά μόνο να εκμεταλλευτούμε και να επεκτείνουμε ίσως κάποιες λειτουργίες που συμβαίνουν φυσιολογικά στο σύστημα, όπως το να επεκτείνουμε τον ελεγκτή κρυφής μνήμης και να χρησιμοποιήσουμε τις ιδιότητες του bus. Οι λειτουργίες ανάγνωσης και εγγραφής χρησιμοποιούνται για να διατηρηθεί η συνοχή, ενώ οι ιδιότητες του bus, που παρέχουν σειριοποίηση για την διατήρηση της συνέπειας.

Αφού όπως είπαμε η write through πολιτική δεν κάνει καλή χρήση του bus, θα μελετήσουμε τις write back κρυφές μνήμης που καταφέρνουν να κάνουν πολύ πιο αποδοτική χρήση του περιορισμένου εύρους ζώνης που παρέχει το διαμοιραζόμενο bus. Με την πολιτική των write back δίνεται στις κρυφές μνήμες η δυνατότητα να γράφουν παράλληλα κάποια διεύθυνση, γεγονός το οποίο

αυξάνει την πολυπλοκότητα του πρωτοκόλλου και απαιτεί την μετάδοση πληροφοριών μεταξύ των κρυφών μνημών για την διατήρηση της συνοχής.

Σε ένα μονοεπεξεργαστή που έχουμε write back πολιτική, όταν συμβεί μια αστοχία εγγραφής στην κρυφή μνήμη, τότε φορτώνεται το ζητούμενο μπλοκ από την κύρια μνήμη, γράφεται η λέξη και το μπλοκ εισέρχεται σε κατάσταση modified ή αλλιώς dirty, ώστε να γραφτεί στην κύρια μνήμη όταν γίνει κάποια αστοχία χωρητικότητας(capacity miss) και θα προκαλέσει την αντικατάσταση του μπλοκ.

Σε ένα πολυεπεξεργαστή αυτή η modified κατάσταση χρησιμοποιείται από το πρωτόκολλο για να υποδηλώσει αποκλειστική ιδιοκτησία του μπλοκ από μια συγκεκριμένη κρυφή μνήμη. Μια κρυφή μνήμη λέμε ότι κατέχει ένα αποκλειστικό αντίγραφο μιας θέσης μνήμης αν είναι η μοναδική κρυφή μνήμη που έχει αντίγραφο και είναι έγκυρο. Φυσικά εξαρτάται από την υλοποίηση του πρωτοκόλλου αν και η κυρία μνήμη θα έχει έγκυρο αντίγραφο ή θεωρείται αποκλειστική ιδιοκτησία μόνο αν η κυρία μνήμη δεν έχει το μπλοκ αυτό έγκυρο. Αν μια κρυφή μνήμη δεν έχει αποκλειστικότητα ενός μπλοκ, τότε θα πρέπει κάθε φορά που ο συγκεκριμένος επεξεργαστή θέλει να γράψει το μπλοκ να στέλνει μηνύματα στους υπόλοιπους επεξεργαστές. Αν μια κρυφή μνήμη όμως έχει το μπλοκ σε modified κατάσταση, τότε μπορεί να το γράψει οποιαδήποτε χρονική στιγμή χωρίς να χρειάζεται να ειδοποιήσει κανένα άλλον.

Σε περίπτωση που έχω αστοχία εγγραφής σε μια κρυφή μνήμη σε ένα πρωτόκολλο ακύρωσης, παράγεται μια ειδική συναλλαγή που ονομάζεται αποκλειστική ανάγνωση(**read exclusive**), και χρησιμοποιείται για να ενημερώσει τις υπόλοιπες κρυφές μνήμες για την εγγραφή που πρόκειται να γίνει και να αποκτήσει ο επεξεργαστής το αντίγραφο του μπλοκ σε αποκλειστική ιδιοκτησία. Δεν επιτρέπεται να γράφουν πολλοί επεξεργαστές το ίδιο μπλοκ μνήμης ταυτόχρονα αφού αυτό θα οδηγήσει σε ασυνεπείς τιμές. Το read exclusive που παράγεται, σειριοποιείται στο bus και άρα μόνο μια από τις κρυφές μνήμες μπορεί να έχει το αντίγραφο του μπλοκ. Όταν ένα modified μπλοκ αντικαθίσταται από την κρυφή μνήμη, τότε τα δεδομένα του πρέπει να σταλούν πίσω στην κύρια μνήμη, ενώ όταν δεν είναι σε modified κατάσταση δεν χρειάζεται να ενημερώσει την κύρια μνήμη αφού ουσιαστικά δεν έχει το τελευταίο αντίγραφο ή αν το έχει τότε υπάρχει το ίδιο και στην κύρια μνήμη, οπότε και απλώς απορρίπτεται.

Εκτός από τα πρωτόκολλα ακύρωσης υπάρχουν και τα πρωτόκολλα ενημέρωσης (**updated-based**), τα οποία και δεν θα αναλύσουμε εκτενώς, παρά μόνο για την πληρότητα του θέματος θα γίνει μια αναφορά, αφού στην εργασία αυτή επιλέξαμε τα πρωτοκόλλα ακύρωσης. Σε ένα πρωτόκολλο ενημέρωσης όταν γράφεται ένα κοινό μπλοκ η τιμή του στέλνεται σε όλες τις κρυφές μνήμες που έχουν αντίγραφο. Για να γίνουν αυτές οι ενημερώσεις φυσικά γίνεται χρήση του κοινού bus για κοινοποίηση όλων των εγγραφών στις κρυφές μνήμες. Δηλαδή καταναλώνεται αρκετά περισσότερο εύρος ζώνης σε σχέση με ένα πρωτόκολλο ακύρωσης. Για αυτό το λόγο οι περισσότεροι πολυεπεξεργαστές χρησιμοποιούν πρωτόκολλα ακύρωσης και έτσι επιλέξαμε και εμείς να αποφύγουμε το πρωτόκολλο ενημέρωσης και να χρησιμοποιήσουμε ακυρωσης.

2.5.3. MESI πρωτόκολλο τεσσάρων καταστάσεων

Το πιο δημοφιλές πρωτόκολλο για write back κρυφές μνήμες το οποίο και χρησιμοποιείται ευρέως είναι το πρωτόκολλο ακύρωσης τεσσάρων καταστάσεων MESI, και προέρχεται από τα αρχικά των καταστάσεων που χρησιμοποιούμε στο συγκεκριμένο FSM: **M**odified **E**xclusive **S**hared **I**nvalid.

Το MESI δημοσιεύτηκε για πρώτη φορά από το πανεπιστήμιο του Illinois το 1984 και αναφέρεται συχνά ως το Illinois protocol και είναι επέκταση του πρωτοκόλλου τριών καταστάσεων MSI. Η βασική βελτίωση που εισάγει το MESI είναι ότι εξαιτίας της χρήσης μιας επιπλέον κατάστασης καταφέρνουμε να μειώσουμε τις συναλλαγές που πρέπει να γίνουν στο bus. Ας δούμε πρώτα κάποιους ορισμούς που θα μας βοηθήσουν να καταλάβουμε και αναλύσουμε καλύτερα πως λειτουργεί το πρωτόκολλο. Οι καταστάσεις έχουν την εξής σημασιολογία:

Invalid: σημαίνει ότι είτε το μπλοκ δεν βρίσκεται στην κρυφή μνήμη είτε βρίσκεται στην κρυφή μνήμη και δεν είναι έγκυρο.

Shared: σημαίνει ότι το μπλοκ βρίσκεται σε περισσότερες από μία κρυφή μνήμη σε μη τροποποιημένη κατάσταση, και η κύρια μνήμη έχει την τελευταία τιμή.

Exclusive: είναι μια ενδιάμεση στην *shared* και την *modified* κατάσταση. Σημαίνει ότι το μπλοκ είναι το μοναδικό αποκλειστικό αντίγραφο, υπάρχει μόνο σε μία κρυφή μνήμη και δεν είναι τροποποιημένο.

Modified: ή αλλιώς ***dirty,*** σημαίνει ότι μόνο αυτή η κρυφή μνήμη έχει αντίγραφο του μπλοκ και το αντίγραφο που υπάρχει στην κύρια μνήμη δεν είναι ενημερωμένο με την τελευταία τιμή.

Οι επεξεργαστές εκδίδουν δύο είδη αιτήσεων: αναγνώσεις(**PrRd**) και εγγραφές(**PrWr**). Η ανάγνωση ή η εγγραφή μπορεί να γίνει σε ένα μπλοκ μνήμης, το οποίο υπάρχει στην κρυφή μνήμη ή που δεν υπάρχει και άρα πρέπει να φορτωθεί από την κύρια μνήμη.

Από την μεριά του bus παράγονται τρία είδη συναλλαγών.

Ανάγνωση bus(**BusRd**): παράγεται από μια αίτηση *PrRd* του επεξεργαστή που αστοχεί στην κρυφή μνήμη. Ο ελεγκτής της κρυφής μνήμης τοποθετεί την διεύθυνση στο bus και περιμένει μια απάντηση από την κύρια μνήμη ή από μια άλλη κρυφή μνήμη ενδεχομένως που περιέχει τα δεδομένα. Το μπλοκ που φορτώνεται δεν πρόκειται να τροποποιηθεί.

Αποκλειστική ανάγνωση bus(**BusRdX**): παράγεται από μια αίτηση *PrWr* του επεξεργαστή σε ένα μπλοκ που δεν βρίσκεται στην κρυφή μνήμη ή βρίσκεται αλλά όχι στην *modified* κατάσταση. Ο ελεγκτής της κρυφής μνήμης τοποθετεί την διεύθυνση στο bus και ζητάει ένα αποκλειστικό αντίγραφο το οποίο και θα τροποποιήσει. Η απάντηση έρχεται και πάλι από την κύρια μνήμη ή από μια άλλη κρυφή μνήμη και περιέχει δεδομένα. Τα αντίγραφα του μπλοκ τα οποία υπάρχουν σε άλλες κρυφές μνήμες ακυρώνονται. Όταν πάρει η κρυφή μνήμη το αποκλειστικό αντίγραφο, μπορεί να γράψει ο επεξεργαστής χωρίς να χρειάζεται να ενημερώσει τους υπόλοιπους αφού έχει το αποκλειστικό αντίγραφο. Η ενέργεια αυτή είναι η μόνη η οποία δεν υπήρχε στο σύστημα και εισάγεται για την υποστήριξη του πρωτοκόλλου συνοχής.

Write Back (**BusWB**): παράγεται από τον ελεγκτή μιας κρυφής μνήμης όταν συμβεί κάποια αστοχία χωρητικότητας. Ο επεξεργαστής δεν ενημερώνεται για αυτή την ενέργεια και δεν αναμένει κάποια απάντηση. Η διεύθυνση τοποθετείται στο bus μαζί με τα δεδομένα με τα οποία θα ενημερωθεί η κύρια μνήμη με τις τελευταίες τιμές.

Για την υποστήριξη του πρωτοκόλλου θα πρέπει επιπλέον ο ελεγκτής να μπορεί να παρεμβαίνει και να παρέχει αυτός δεδομένα για το ζητούμενο μπλοκ από την κρυφή του μνήμη αντί να αφήνει την κύρια μνήμη να δώσει τα δεδομένα.

Αφού λοιπόν εξετάσαμε κάποιες βασικές έννοιες, θα αναλύσουμε τις μεταβάσεις από τις καταστάσεις για το FSM του MESI. Έστω ένα μπλοκ το οποίο

θέλει ο επεξεργαστής να διαβάσει και το οποίο δεν βρίσκεται στην κρυφή του μνήμη αρχικά, δηλαδή είναι σε κατάσταση invalid. Η ανάγνωση αυτή του επεξεργαστή μεταφράζεται συνεπώς σε *BusRd* ενέργεια στο bus για την εξυπηρέτηση της αστοχίας. Αν το μπλοκ έχει ήδη φορτωθεί και σε κάποιες άλλες κρυφές μνήμες και είναι σε έγκυρη κατάσταση, τότε το μπλοκ φορτώνεται στην τοπική κρυφή μνήμη σε κατάσταση *shared*. Αν όμως δεν υπάρχει πουθενά αλλού τότε φορτώνεται σε κατάσταση *exclusive*. Έτσι τώρα όταν το μπλοκ θα γραφεί από τον ίδιο επεξεργαστή, μπορεί να μεταβεί απευθείας από την *exclusive* στην *modified* κατάσταση χωρίς να παράγει κάποια νέα συναλλαγή στο *bus*, αφού δεν υπάρχει άλλος επεξεργαστής που να έχει αντίγραφο του μπλοκ. Ομοίως, στην κατάσταση αυτή ο τοπικός επεξεργαστής μπορεί να διαβάζει το μπλοκ χωρίς να φαίνεται στο bus. Αν όμως ενώ το μπλοκ είναι σε κατάσταση *exclusive* κάποιος άλλος επεξεργαστής έχει εκδηλώσει επιθυμία να το διαβάσει, δηλαδή παρατηρήσει ο ελεγκτής κάποιο *BusRd* στο bus τότε η κατάσταση του μπλοκ ΄΄πέφτει΄΄ στην *shared* κατάσταση και ο επεξεργαστής που έκανε την αίτηση παίρνει το μπλοκ από την κύρια μνήμη ή από την κρυφή μνήμη που έχει αποκλειστική ιδιοκτησία του μπλοκ ανάλογα με την σχεδιαστική επιλογή. Υπάρχει πάντα και η περίπτωση ενώ το μπλοκ βρίσκεται σε *exclusive* κατάσταση να θέλει κάποιος επεξεργαστής να γράψει το συγκεκριμένο μπλοκ, δηλαδή να παρατηρήσει ο ελεγκτής στο bus μια συναλλαγή BusRdX, τότε το μπλοκ ΄΄πέφτει΄΄ στην invalid κατάσταση αφού ακυρώνεται για να μπορέσει να φορτωθεί από τον άλλο επεξεργαστή και να το γράψει.

Ας υποθέσουμε τώρα ότι το μπλοκ βρίσκεται σε περισσότερες από μια κρυφή μνήμη, δηλαδή είναι σε *shared* κατάσταση. Από αυτή την κατάσταση, ο τοπικός επεξεργαστής μπορεί να διαβάσει το μπλοκ χωρίς να φαίνεται κάποια ενέργεια στο bus και παραμένοντας το μπλοκ στην ίδια κατάσταση. Αν παρατηρήσει ο ελεγκτής μια συναλλαγή BusRd στο bus από έναν άλλον επεξεργαστή που θέλει να διαβάσει, παραμένει στην *shared* κατάσταση. Αν παρατηρήσει ο ελεγκτής μια συναλλαγή BusRdX στο bus από έναν άλλον επεξεργαστή που θέλει να γράψει το μπλοκ τότε η κατάσταση του ΄΄πέφτει΄΄ σε *invalid* κατάσταση. Στην περίπτωση όμως που θέλει να γράψει ο τοπικός επεξεργαστή που έχει ήδη σε *shared* το μπλοκ, τότε τοποθετεί στο bus μια αίτηση *BusRdX* για να ακυρώσουν όλοι οι υπόλοιποι επεξεργαστές τα αντίγραφα τους, και μεταβαίνει σε *modified* κατάσταση αφού έχει γράψει το μπλοκ.

Έστω ότι το μπλοκ είναι σε *modified* κατάσταση, δηλαδή έχει γραφεί από τον επεξεργαστή, ο οποίος και έχει την τελευταία τιμή ενώ η κύρια μνήμη δεν έχει το πιο ενημερωμένο αντίγραφο του μπλοκ. Από αυτήν την κατάσταση ο τοπικός επεξεργαστής μπορεί να διαβάζει το μπλοκ ή ακόμα και να το γράφει χωρίς να παράγει καμιά συναλλαγή στο bus και χωρίς να αλλάζει κατάσταση. Αν όμως όσο είναι σε *modified* κατάσταση, ένας άλλος επεξεργαστής θέλει να διαβάσει το μπλοκ αυτό τότε έχουμε μια μετάβαση στην *shared* κατάσταση, αφού περισσότερες από μια κρυφές μνήμες έχουν το αντίγραφο. Το αντίγραφο αυτό το παρέχει φυσικά η κρυφή μνήμη η οποία και είχε το πιο πρόσφατο αντίγραφο, που το είχε γράψει ο τοπικός της επεξεργαστής με μια μέθοδο που ονομάζεται *cache-to-cache sharing*. Αν όμως κατά τη διάρκεια που βρίσκεται το μπλοκ σε *modified* κατάσταση κάποιος άλλος επεξεργαστής θελήσει να το γράψει, έχει τοποθετήσει μια *BusRdX* συναλλαγή, την οποία και παρατηρεί ο ελεγκτής της κρυφής μνήμης με το τροποποιημένο μπλοκ και το ακυρώνει, τότε το μπλοκ μεταβαίνει στην *invalid* κατάσταση και γίνεται μια ενέργεια *cache-to-cache sharing*.

Τέλος από την *invalid* κατάσταση όταν ο επεξεργαστής θέλει να γράψει το μπλοκ *PrWr*, τοποθετεί ένα σήμα *BusRdX* στο bus για να ακυρώσει όλα τα αντίγραφα που υπάρχουν στις υπόλοιπες κρυφές μνήμες και να το έχει αποκλειστικά εκείνος για εγγραφή.

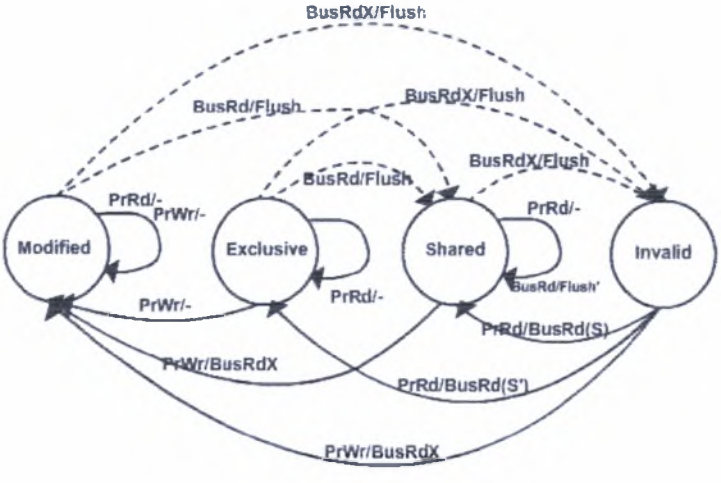
Γενικά, το πρωτόκολλο αυτό προσθέτει κάποιο επιπρόσθετες απαιτήσεις, όπως την ύπαρξη κάποιου σήματος S που ονομάζεται *shared signal* και το οποίο χρησιμοποιείται από τον ελεγκτή της κρυφής μνήμης για να δείξει αν υπάρχει άλλη κρυφή μνήμη που έχει αντίγραφο του μπλοκ που πρόκειται να διαβαστεί. Το σήμα S μπορεί να υλοποιηθεί ως ένα wired-OR σήμα, το οποίο γίνεται 1, εάν κάποιος από τους ελεγκτές έχει στην κρυφή του μνήμη το ζητούμενο μπλοκ, οπότε και να αποφασιστεί εάν το μπλοκ μνήμης φορτωθεί σε *shared* ή *exclusive* κατάσταση.

Στο σχήμα 9 παρακάτω, θα κάνουμε μερικές διευκρινήσεις. Υπάρχουν ουσιαστικά τρία είδη BusRd. Το BusRd, το οποίο έχει την γνωστή λειτουργία χωρίς να χρειάζεται να προσέξουμε τίποτα άλλο. Το BusRd(S), το οποίο σημαίνει ότι το μπλοκ αυτό υπάρχει και σε άλλη κρυφή μνήμη και το BusRd(S΄),που σημαίνει ότι το σήμα S δεν έχει τεθεί από άλλη κρυφή μνήμη και άρα είναι αποκλειστικό αντίγραφο. Το Flush΄ σημαίνει ότι το μπλοκ παρέχεται μόνο εάν

είναι ενεργοποιημένο το *cache-to-cache sharing* και τότε μόνο η κρυφή μνήμη που έχει το τελευταίο αντίγραφο μπορεί να δώσει τα δεδομένα.

Γενικά, αφού έχουμε write back κρυφές μνήμες, το μπλοκ μπορεί να γραφεί από τον επεξεργαστή πολλές φορές χωρίς να ενημερωθεί η κύρια μνήμη. Η ενημέρωση της κύριας μνήμης θα γίνει μόνο όταν έχουμε κάποια αστοχία στην κρυφή μνήμη λόγω μεγέθους και το μπλοκ πρέπει να αντικατασταθεί με ένα νέο. Η αντικατάσταση αυτή προφανώς αλλάζει την κατάσταση του μπλοκ σε invalid και την απομακρύνει από την κρυφή μνήμη. Αν το μπλοκ δεν ήταν σε modified κατάσταση, τότε δεν χρειάζεται κάποια ειδική μεταχείριση. Αν ήταν όμως σε modified κατάσταση τότε πρέπει να γίνει μια write back συναλλαγή.

σχήμα9 **MESI PROTOCOL**



Μπορούμε να δούμε πώς σε σχέση με το MSI, με το MESI πρωτόκολλο μειώνουμε τις προσβάσεις στο bus. Το MSI δεν διαθέτει κατάσταση E και έχοντας άλλη μια κατάσταση στο FSM δεν χρειάζεται να παράγουμε συναλλαγές στο bus αφού έχοντας περάσει στην *exclusive* κατάσταση μπορεί να γράψει ο επεξεργαστής χωρίς να πρέπει να ενημερώσει και άρα να περάσει στην *modified* κατάσταση.

Πριν προχωρήσουμε για να δούμε γιατί το πρωτόκολλο αυτό ικανοποιεί τις ιδιότητες της συνοχής και της ακολουθιακής συνέπειας, θα αναλύσουμε το ζήτημα που προκύπτει με το *cache-to-cache sharing*. Το θέμα που προκύπτει είναι το ποιος θα δώσει τα δεδομένα όταν παρατηρηθεί μια BusRd συναλλαγή

στο bus και το μπλοκ υπάρχει ενημερωμένο και στην κύρια μνήμη αλλά και σε κάποια από τις κρυφές μνήμες. Στην αρχική υλοποίηση από το πανεπιστήμιο του Illinois η επιλογή ήταν να παρέχει η κρυφή μνήμη που έχει το μπλοκ τα δεδομένα, *cache-to-cache sharing* και αυτό γινόταν επειδή η κρυφές μνήμες ήταν πιο γρήγορες από την κύρια μνήμη. Στα σημερινά συστήματα όμως η πρόσβαση στην κύρια μνήμη είναι γρηγορότερη από το να πάρουμε τα δεδομένα από την κρυφή μνήμη κάποιου άλλου επεξεργαστή. Επιπλέον το *cache-to-cache sharing* αυξάνει την πολυπλοκότητα του συστήματος που στηρίζεται σε πρωτόκολλο κοινού bus, αφού η κύρια μνήμη θα πρέπει να περιμένει μέχρι να αποφασιστεί αν θα παρέχει τα δεδομένα μια κρυφή μνήμη, πριν δεσμεύσει το bus για να δώσει αυτή τα δεδομένα. Ακόμα μεγαλύτερο πρόβλημα και καθυστέρηση υπάρχει αν το μπλοκ υπάρχει ως αντίγραφο σε πολλές κρυφές μνήμες και πρέπει να εφαρμοστεί κάποιος αλγόριθμος επιλογής για να αποφασιστεί ποια από όλες θα παρέχει τα δεδομένα.

Τώρα μπορούμε να δούμε για ποιο λόγο ικανοποιείται η συνοχή με αυτό το πρωτόκολλο. Η γνωστοποίηση μιας εγγραφής από ένα επεξεργαστή στους υπόλοιπους ικανοποιείται εξ ορισμού από το πρωτόκολλο αφού όταν γίνει μια εγγραφή οι επεξεργαστές επικοινωνούν μεταξύ τους για να ακυρώσουν τα υπόλοιπα αντίγραφα. Για σειριοποίηση της εγγραφής: Η συναλλαγή BusRdX αφού ακυρώνει τα αντίγραφα εγγυάται ότι η κρυφή μνήμη που θα γραφτεί έχει το μοναδικό έγκυρο αντίγραφο. Η συναλλαγή αυτή ακολουθείται άμεσα από την εγγραφή πριν αναλάβει ο ελεγκτής της κρυφής μνήμης να χειριστεί άλλες συναλλαγές από το bus, άρα σειριοποιείται σε σχέση με τις υπόλοιπες συναλλαγές. Ακόμη και μεταξύ δύο συναλλαγών που εμφανίζονται στο bus, μόνο ένας επεξεργαστής μπορεί να κάνει ευστοχίες εγγραφής. Η αλληλουχία των ευστοχιών εγγραφής εμφανίζεται μεταξύ μιας BusRdX και τις επόμενης συναλλαγής του bus για το μπλοκ. Οι αναγνώσεις από τον επεξεργαστή που έκανε τις ευστοχίες εγγραφής θα δουν τις συναλλαγές με αυτή τη σειρά. Σε ανάγνωση από άλλους επεξεργαστές, υπάρχει τουλάχιστον μια συναλλαγή για το μπλοκ αυτό που ξεχωρίζει την ολοκλήρωση της ανάγνωσης από τις ευστοχίες εγγραφής. Άρα οι συναλλαγές αυτές στο bus διασφαλίζουν ότι και οι αναγνώσεις βλέπουν τις εγγραφές σε συνεπή σειρά και συνεπώς οι αναγνώσεις από όλους τους επεξεργαστές βλέπουν τις εγγραφές με την ίδια σειρά.

Όσο για το αν εγγυάται το πρωτόκολλο συνέπεια, ισχύει και εδώ ότι εξαιτίας του σειριακού τρόπου με τον οποίο αποφασίζεται ποιος θα δεσμεύσει το bus, ορίζεται μια καθολική σειρά στις συναλλαγές του bus για όλα τα μπλοκ. Έτσι όλοι οι ελεγκτές κρυφής μνήμης παρατηρούν τις αναγνώσεις και τα BusRdX με την ίδια σειρά, άρα και οι ακυρώσεις γίνονται με την ίδια σειρά. Μεταξύ δύο συνεχόμενων συναλλαγών στο bus, οι επεξεργαστές μπορούν να εκτελούν μια ακολουθία λειτουργιών σε σειρά προγράμματος. Άρα κάθε εκτέλεση του προγράμματος ορίζει μια μερική διάταξη, όπου μια λειτουργία μνήμης Α ακολουθεί μια λειτουργία Β αν και οι δύο λειτουργίες εκδίδονται από τον ίδιο επεξεργαστή και Α είναι μεταγενέστερη της Β σε σειρά προγράμματος ή αν η λειτουργία Α παράγει μια συναλλαγή στο bus που ακολουθεί την λειτουργία για την Β.

Επιπλέον οι επαρκείς συνθήκες που εξασφαλίζουν συνέπεια είναι: Η ολοκλήρωση των εγγραφών ανιχνεύεται όταν εμφανίζονται BusRdX στο bus και η κρυφή μνήμη κάνει την εγγραφή. Η ολοκλήρωση μιας ανάγνωσης παράγει μια συναλλαγή στο bus η οποία ακολουθεί την συναλλαγή της εγγραφής της οποίας την τιμή θα επιστρέψει και άρα η εγγραφή θα πρέπει να έχει ολοκληρωθεί και να φαίνεται αυτό καθολικά. Μπορεί βέβαια να ακολουθεί μιας άλλης ανάγνωσης από τον ίδιο επεξεργαστή σε σειρά προγράμματος. Τέλος μπορεί να ακολουθεί σε σειρά προγράμματος την εγγραφή που έχει εκτελεστεί από τον ίδιο επεξεργαστή και άρα και πάλι έχει ολοκληρωθεί και το γεγονός έχει γίνει ορατό καθολικά. Άρα μπορούμε να δούμε ότι και οι συνθήκες ικανοποιούνται, και η συνέπεια διατηρείται.

*2.6. Συγχρονισμός (**Synchronization**)*

Ο σκοπός όλων αυτών των βελτιώσεων που κάνουμε σε επίπεδο υλικού είναι να μπορέσουν να χρησιμοποιηθούν από το λογισμικό και μέσω αυτού να εκμεταλλευτούμε την ύπαρξη δυο επεξεργαστών και άρα της μεγαλύτερης υπολογιστικής ισχύος. Αυτό μπορεί να επιτευχθεί με μηχανισμούς συγχρονισμού[1][2]. Τα βασικά δομικά στοιχεία των μηχανισμών αυτών, κάποιου είδους ατομικές λειτουργίες ανάγνωση-τροποποίηση-εγγραφή(read-modify-write), είναι συνήθως υλοποιημένα σε υλικό και πάνω σε αυτή την υποδομή

δίνεται ακολούθως η δυνατότητα να υλοποιηθούν πιο πολύπλοκοι αλγόριθμοι συγχρονισμού.

Η βασική δυνατότητα που πρέπει να παρέχεται από το υλικό είναι μια ατομική εντολή ή μια ακολουθία εντολών με δυνατότητα να διαβάζουν και να αλλάζουν ατομικά μια τιμή. Πάνω σε αυτή την δυνατότητα κατασκευάζονται software μηχανισμοί συγχρονισμού σε λογισμικό. Σε επίπεδο χρήστη όπως οι κλειδαριές(locks) και τα φράγματα (barriers). Γενικά, δεν αναμένουμε από τους χρήστες να χρησιμοποιούν τα βασικά δομικά αυτά στοιχεία, αλλά αντί γι αυτό τα δομικά αυτά στοιχεία μπορούν να χρησιμοποιηθούν από τους προγραμματιστές συστήματος για την δημιουργία βιβλιοθηκών συγχρονισμού και αυτές τις βιβλιοθήκες να χρησιμοποιούν οι προγραμματιστές.

Υπάρχει μια ποικιλία από δομικά στοιχεία συγχρονισμού[7], κάποια από αυτά υλοποιούν ατομικά μια απλή ανταλλαγή τιμών μεταξύ ενός καταχωρητή και μιας θέσης, όπως τα exch και swap. Μερικά πάλι υλοποιούν ατομικά την ίδια ανταλλαγή τιμών, αλλά υπό συνθήκη, ενώ κάποια άλλα υλοποιούν ατομικά και τροποποίηση της αρχικά αποθηκευμένης στη μνήμη τιμής. Ένα από αυτά που χρησιμοποιείται σε πολυεπεξεργαστές είναι το *test&set*, το οποίο αντιγράφει την τιμή μιας διεύθυνσης μνήμης σε ένα καταχωρητή και θέτει τη τιμή στην διεύθυνση=1 ατομικά, αν η τιμή του καταχωρητή είναι 0, οπότε το *test&set* ήταν επιτυχές. Αν είναι 1 τότε αυτό σημαίνει ότι έχει ήδη προσπελαστεί από άλλο επεξεργαστή, που για υλοποίηση μηχανισμού κλειδώματος θα αντικατοπτρίζει την κατοχή της κλειδαριάς από τον άλλο επεξεργαστή. Παρόμοια λειτουργεί και η ατομική εντολή compare&swap. Ένα ακόμη δομικό στοιχείο είναι εντολές τύπου *fetch&oper*(*increment-decrement-add*). Οι εντολές αυτές διαβάζουν ατομικά την τιμή μιας διεύθυνσης, την τοποθετούν σε καταχωρητή και γράφουν μια τροποποιημένη τιμή-ανάλογα με το oper- στην διεύθυνση.

Μια εναλλακτική επιλογή είναι να έχουμε ένα ζευγάρι από εντολές, όπου η δεύτερη εντολή επιστρέφει μια τιμή από την οποία μπορούμε να καταλάβουμε εάν το ζεύγος τον εντολών εκτελέστηκε ατομικά. Σε αυτό το είδος ανήκει το *load-linked* και *store-conditional(LL-SC),* όπου εάν τα περιεχόμενα της διεύθυνσης που καθορίζεται από το *load-linked* αλλαχθούν πριν το *store-conditional* στην ίδια διεύθυνση, τότε το SC αποτυγχάνει. Επίσης εάν ο επεξεργαστής κάνει context switch μεταξύ των εντολών πάλι αποτυγχάνει. Ο μηχανισμός αυτός μπορεί να μειώσει αισθητά την κίνηση στο bus σε σχέση με τους προηγούμενους.

Αφού έχουμε καταφέρει να δημιουργήσουμε μια ατομική λειτουργία, μπορούμε να χρησιμοποιήσουμε τους μηχανισμούς συνοχής για πολυεπεξεργαστές για να υλοποιήσουμε *spin locks*, δηλαδή locks όπου ο επεξεργαστής προσπαθεί να περάσει μπαίνοντας σε βρόχο μέχρι να πετύχει, σε επίπεδο λογισμικού[4]. Η πιο απλή υλοποίηση αν δεν υποστηρίζεται συνοχή κρυφής μνήμης έχει τις μεταβλητές κλειδώματος στην κύρια μνήμη. Ο επεξεργαστής προσπαθεί συνεχώς να πάρει την μεταβλητή, να την ανταλλάξει και να τεστάρει αν έχει πάρει το lock ως ελεύθερο και άρα μπορεί να το χρησιμοποιήσει.

Αν τώρα υποθέσουμε ότι έχουμε μηχανισμό συνοχής κρυφής μνήμης, μπορούμε να βάλουμε τα locks στην κρυφή μνήμη και μέσω του μηχανισμού να τα διατηρήσουμε συνεκτικά. Πλεονέκτημα της μεθόδου αυτής είναι ότι η διαδικασία του ΄΄spinning΄΄ μπορεί να γίνει στην τοπική μνήμη και επιπλέον έτσι εκμεταλλευόμαστε την τοπικότητα στις προσβάσεις των locks. Κάθε προσπάθεια να γίνει η ανταλλαγή κατά τη διάρκεια του spinning απαιτεί μια λειτουργία εγγραφής. Εάν έχουμε πολυεπεξεργαστή, κάθε επεξεργαστής που θέλει να πάρει το lock θα παράγει μια λειτουργία εγγραφής, οι περισσότερες εκ των οποίων θα προκαλέσουν αστοχίες εγγραφής αφού κάθε επεξεργαστής προσπαθεί αν πάρει την μεταβλητή lock σε exclusive κατάσταση. Στο σχήμα 7 μπορούμε να δούμε πως τα spin locks χρησιμοποιούν τον μηχανισμό συνοχής. Μόλις ο επεξεργαστής με το lock αποθηκεύει 0 στο lock, όλες οι υπόλοιπες κρυφές μνήμες ακυρώνονται και ξαναφορτώνουν την μεταβλητή. Όταν εξυπηρετηθεί η αστοχία, βρίσκουν την μεταβλητή κλειδωμένη και επιστρέφουν σε κατάσταση ΄΄spinning΄΄. Σε αντίθεση με τα παραπάνω, αν βασιστούμε σε LL-SC για τα spin locks δεν έχουμε συναλλαγές στο bus όσο περιμένουμε. Επίσης δεν γίνονται ακυρώσεις σε μια αποτυχημένη προσπάθεια να πάρουμε το lock. Ένας ακόμη δημοφιλής μηχανισμός συγχρονισμού, στον οποίο δεν θα επεκταθούμε είναι τα *barriers*. Αλγόριθμοι λογισμικού για *barriers* υλοποιούνται συνήθως με locks. Φυσικά υπάρχουν πολλές διαφορετικές υλοποιήσεις μηχανισμών συγχρονισμού με λογισμικό, στις οποίες δεν θα επεκταθούμε .

σχημα4

Step	Processor P0	Processor P1	Processor P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back from P0
4		(Waits while bus/ directory busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets Lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1, and sets Lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; generates write back
8		Spins, testing if lock = 0			None

3. Ο μικροεπεξεργαστής LEON2

3.1. Γενικά

Στο κεφάλαιο αυτό θα αναλύσουμε τα χαρακτηριστικά και την αρχιτεκτονική του μικροεπεξεργαστή LEON και τις επιλογές που έγιναν για την διαμόρφωση του ώστε να χρησιμοποιηθεί κατά την διάρκεια αυτής της εργασίας.

Ο LEON είναι ένας open-source 32-bit μικροεπεξεργαστής, υλοποιημένος με την γλώσσα περιγραφής υλικού VHDL, που ξεκίνησε αρχικά από την European Space Agency και αναπτύχθηκε περαιτέρω από την Gaisler Research, από την οποία και διατίθεται κάτω από το Lesser GNU Public License (LGPL) και αρχικά χρησιμοποιήθηκε για εφαρμογές διαστήματος[8].

3.2. Αρχιτεκτονική Συνόλου Εντολών

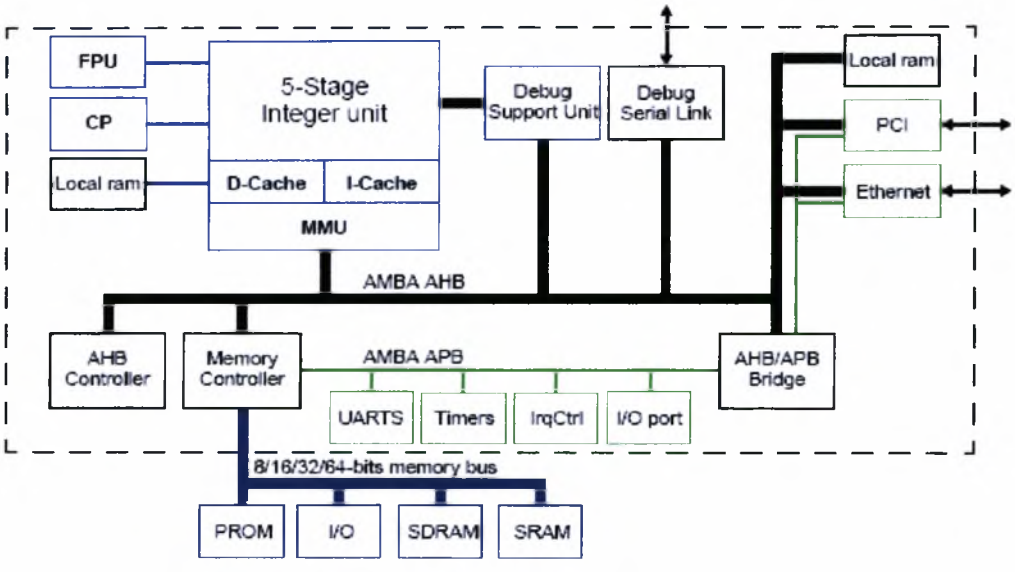
Είναι βασισμένος και υλοποιεί το σύνολο εντολών της αρχιτεκτονικής SPARC V8, η οποία είναι εξέλιξη της αρχιτεκτονικής RISC και δημιουργήθηκε από την Sun Microsystems με σκοπό την εύκολη υλοποίηση συστημάτων υψηλής απόδοσης με σχετικά χαμηλό κόστος και στοχεύει σε ενσωματωμένα συστήματα και SoC(System-On-Chip) εφαρμογές.

Το VHDL μοντέλο του LEON υλοποιεί έναν 32-bit επεξεργαστή πλήρως συμβατό με το πρότυπο της SPARC V8 αρχιτεκτονικής, χρησιμοποιεί big-endian κωδικοποίηση, διαθέτει 32-bit εσωτερικούς καταχωρητές και 72 εντολές που χωρίζονται σε τρία είδη με τρεις διαφορετικούς τρόπους διευθυνσιοδότησης, δηλαδή άμεσο τελούμενο, μετατόπιση και δεικτοδότηση. Επίσης υλοποιούνται προσημασμένες και μη-προσημασμένες πράξεις πολλαπλασιασμού, διαίρεσης και λειτουργίες MAC. Διαθέτει επικάλυψη εντολών 5-σταδίων (Προσκόμιση εντολής, Αποκωδικοποίηση, Εκτέλεση, Προσπέλαση Μνήμης και Εγγραφή) και διεπαφές για ξεχωριστή κρυφή μνήμη εντολών από κρυφή μνήμη δεδομένων σύμφωνα με την αρχιτεκτονική Harvard.

Ο κώδικας VHDL του LEON είναι πλήρως συνθέσιμος με τα περισσότερα εργαλεία synthesis, όπως είναι το xilinx xst, το synplify pro, το altera quartus και μπορεί να μετατραπεί ή να προστεθεί κάποια λειτουργική μονάδα σ' αυτόν αρκετά εύκολα εξαιτίας του γεγονότος ότι διαθέτει αρτηρίες AMBA-2.0 AHB/APB.

Παρακάτω δίνεται ένα διάγραμμα του μικροεπεξεργαστή LEON2(σχήμα 10). Εξαιτίας των διαφορετικών επιλογών που μπορούν να υπάρξουν για την διαμόρφωση του επεξεργαστή κάποιες από τις μονάδες που διακρίνονται στην εικόνα μπορούν να αφαιρεθούν.

σχήμα 10:



Ο LEON2 υλοποιεί τα ακόλουθα χαρακτηριστικά, αναλυτικά:

- *Μικροεπεξεργαστή RISC 32-bit*
- *Συμβατότητα με SPARC V8*
- *Επικάλυψη εντολών 5-σταδίων*
- *Πολλαπλασιασμό, διαίρεση, λειτουργίες MAC στο υλικό*
- *Ξεχωριστές μνήμες εντολών και δεδομένων*
- *Μονάδα διαχείρησης μνήμης για υποστήριξη εικονικής μνήμης (MMU)*
- *Διεπαφές μνημών για FLASH, SRAM, SDRAM και PROM*
- *On-chip μνήμη RAM*
- *Διαχείρηση διακοπών*
- *Υπομονάδα Κινητής Υποδιαστολής, (FPU)*
- *Υπομονάδα Αποσφαλμάτωσης, (DSU)*
- *Δύο 24-bit χρονιστές*
- *Δύο ελεγκτές σειριακής πόρτας, UART*
- *Ουρά 16-bit I/O απεικονισμένη στην μνήμη*

- *Watchdog και αδρανοποίηση*
- *Ελεγκτή Ethernet 10/100 MAC*
- *Διεπαφή PCI*
- *Διεπαφή για συν-επεξεργαστή*

Ειδικότερα:

3.3. Μονάδα Ακεραίων

Η μονάδα ακεραίων του LEON υλοποιεί πλήρως το πρότυπο της αρχιτεκτονικής SPARC V8 με το γνωστό μοντέλο επικάλυψης εντολών 5-σταδίων. Χρησιμοποιείται για εκτέλεση πράξεων ακεραίων, υπολογισμό διευθύνσεων μνήμης. Επίσης διατηρεί τον μετρητή προγράμματος (PC) και ελέγχει την εκτέλεση εντολών στη μονάδα κινητής υποδιαστολής και τον συνεπεξεργαστή. Διαθέτει καταχωρητές γενικού σκοπού οργανωμένους με την τεχνική των ''παραθύρων'' (*register windows*).

Ο αριθμός των παραθύρων των καταχωρητών μπορεί να διαμορφωθεί από 2 έως 32 σύμφωνα με το πρότυπο SPARC με προκαθορισμένη τιμή τα 8 παράθυρα. Κάθε παράθυρο αποτελείται από οκτώ καταχωρητές in, οκτώ καταχωρητές out και οκτώ καταχωρητές local, όπου οι καταχωρητές εισόδου(in) και εξόδου(out) είναι κοινοί σε παρακείμενα παράθυρα. Για παράδειγμα οι καταχωρητές in του παραθύρου 1 είναι οι ίδιοι με τους καταχωρητές out του παραθύρου 0, έχουν ίδια διεύθυνση. Δηλαδή δύο παρακείμενα παράθυρα μοιράζονται οκτώ καταχωρητές. Οι καταχωρητές τύπου local είναι μοναδικοί για κάθε παράθυρο και υπάρχουν επιπλέον οκτώ καθολικοί(global) καταχωρητές κοινοί για όλα τα παράθυρα. Άρα υπάρχουν ανάλογα με τη διαμόρφωση (configuration) 8+16*Nwindows, όπου Nwindows ο αριθμός των παραθύρων.

Για μονάδες πολλαπλασιασμού υπάρχει δυνατότητα επιλογής διαμόρφωσης τους όπως φαίνεται στο σχήμα 11, όπου διακρίνεται η καθυστέρηση για την εξαγωγή αποτελέσματος αλλά και το μέγεθος που δεσμεύουν σε χιλιάδες πύλες για την υλοποίηση τους :

σχήμα 11:

Configuration	Latency (clock)	Approx. area (Kgates)
iterative	35	1000
M16x16 + pipeline reg	5	6500
m16x16	4	6000
m32x8	4	5000
m32x16	2	9000
m32x32	1	15000

Πράξεις MAC εκτελούνται με είσοδο 16x16 bit και 40-bit συσσωρευτή, ενώ εντολές διαίρεσης με διαιρέτες με βάση το δύο και καθυστέρηση 35cc (κύκλους ρολογιού).

Η μονάδα ακεραίων, οι κρυφές μνήμες εντολών και δεδομένων και οι ελεγκτές τους θεωρούνται μαζί με την μονάδα κινητής υποδιαστολής και το συνεπεξεργαστή ως ο ΄΄πυρήνας΄΄(core) του επεξεργαστή.

3.4. Μονάδα Κινητής Υποδιαστολής

Ο LEON παρέχει 3 διεπαφές για μονάδες κινητής υποδιαστολή. Μια για το GRFPU, που είναι υλοποιημένο σύμφωνα με το πρότυπο IEEE-754 και διατίθεται σε εμπορική μορφή από την Gaisler Research. Μια για το Meiko FPU της Sun Microsystems και τέλος μια για το LTH FPU το οποίο και υλοποιείται αλλά είναι ημιτελές και δεν εκτελεί όλες τις εντολές κινητής υποδιαστολής της αρχιτεκτονικής SPARC V8 και άρα δεν μπορεί να χρησιμοποιηθεί για προγράμματα γενικού σκοπού παρά μόνο για περιορισμένες εφαρμογές.

3.5. Συνεπεξεργαστής

Υπάρχει η επιλογή ενεργοποίησης συνεπεξεργαστή αλλά διατίθεται μόνο η διεπαφή. Η υλοποίηση πρέπει να γίνει από τον χρήστη, ανάλογα με τις ανάγκες του.

3.6. Υποσύστημα Κρυφών Μνημών

Το υποσύστημα κρυφών μνημών υλοποιείται σύμφωνα με το αρχιτεκτονικό μοντέλο Harvard, ξεχωριστή κρυφή μνήμη εντολών και κρυφή μνήμη δεδομένων.

Για κάθε είδος κρυφής μνήμης μπορούμε να επιλέξουμε μέγεθος από 1 – 64 Kbytes για κάθε σύνολο, όπου κάθε σύνολο διαθέτει 4 – 8 μπλοκ των 4 bytes.

Επιπλέον υπάρχει η επιλογή οι κρυφές μνήμες να είναι άμεσης διευθυνσιοδότησης ή και σύνολο-συσχετιστικές 2 ή 4 δρόμων. Για τις σύνολο-συσχετιστικές διατίθενται 3 αλγόριθμοι αντικατάστασης:

1)LRU: αντικατάσταση του μπλοκ μνήμης που χρησιμοποιήθηκε λιγότερο πρόσφατα.

2)LRR: αντικατάσταση του μπλοκ μνήμης που αντικαταστάθηκε λιγότερο πρόσφατα.

3)Ψευδοτυχαία (*Random*) αντικατάσταση.

Για μείωση της ποινής αστοχίας της κρυφής μνήμης χρησιμοποιείται μια τεχνική κατά την οποία δεδομένα στέλνονται στον επεξεργαστή παράλληλα με την εγγραφή τους στην κρυφή μνήμη. Έτσι σε περίπτωση αστοχίας στην μνήμη δεδομένων , μόνο το ζητούμενο μπλοκ φορτώνεται.

Η κρυφή μνήμη δεδομένων χρησιμοποιεί ΄΄write through, no-allocate ΄΄ πολιτική εγγραφής, το οποίο σημαίνει ότι για κάθε εγγραφή στην κρυφή μνήμη ενημερώνεται άμεσα και η κύρια μνήμη και σε περίπτωση που θέλει ο επεξεργαστής να γράψει ένα μπλοκ μνήμης που δεν βρίσκεται στην κρυφή μνήμη, μπορεί να γράψει απευθείας στην κύρια μνήμη χωρίς να έχει φορτωθεί προηγουμένως το μπλοκ στην κρυφή μνήμη. Για μείωση του παγώματος του μηχανισμού επικάλυψης που προκαλείται από εντολές αποθήκευσης, υπάρχει χώρος εγγραφής (store buffer) εγγραφής διπλής λέξης.

Ο store buffer αυτός είναι προσωρινός χώρος αποθήκευσης που αποτελείται από 3x32bit καταχωρητές όπου τοποθετούνται τα προς αποθήκευση δεδομένα μέχρι να σταλούν στην συσκευή προορισμού. Ο buffer αδειάζει πριν από την φόρτωση δεδομένων μετά από αστοχία ανάγνωσης για να αποφευχθεί η πιθανότητα φόρτωσης παλιών αντιγράφων.

Τέλος για την διατήρηση της συνοχής της κρυφής μνήμης δεδομένων, όταν πολλές μονάδες έχουν την δυνατότητα να γράφουν στην μνήμη χρησιμοποιείται πρωτόκολλο υποκλοπής (snooping protocol) στις αρτηρίες AHB για έλεγχο μπλοκ που έχουν υποστεί αλλαγές, το οποίο όμως είναι διαθέσιμο μόνο όταν δεν είναι ενεργοποιημένη η MMU μονάδα, δηλαδή μόνο για φυσικές διευθύνσεις.

3.7. Μονάδα Διαχείρησης Μνήμης(MMU)

Η μονάδα διαχείρησης μνήμης μπορεί να ενεργοποιηθεί για να παρέχει μηχανισμούς προστασίας της μνήμης, σύμφωνα με το πρότυπο του SPARC V8, για

εξελιγμένα λειτουργικά συστήματα, όπως το linux ή το solaris και μετάφραση εικονικών διευθύνσεων σε φυσικές διευθύνσεις κύριας μνήμης, δηλαδή υλοποίηση εικονικής μνήμης. Όταν το MMU είναι ενεργοποιημένο οι κρυφές μνήμες λειτουργούν με φυσικές διευθύνσεις.

Η MMU μπορεί να διαμορφωθεί ώστε να υποστηρίζει ξεχωριστό ή κοινό TLB για τις μνήμες δεδομένων και εντολών. Το TLB είναι πλήρως συσχετιστικό και δύναται να έχει μέγεθος από 2 – 32 εγγραφές. Η οργάνωση του TLB και ο αριθμός των εγγραφών δεν είναι ορατά στο λογισμικό (λειτουργεί σαν μαύρο κουτί) και άρα δεν χρειάζονται μετατροπές στο λειτουργικό σύστημα για την υποστήριξη του.

3.8. Διεπαφές Συστήματος

Οι συνδέσεις στον επεξεργαστή LEON για επικοινωνία των μονάδων με τα περιφερειακά γίνονται μέσω των αρτηριών AMBA[11].Υπάρχουν δύο είδη αρτηριών AMBA:1)AHB , 2)APB.

Το Advanced High performance Bus (AMBA-2.0AHB) που χρησιμοποιείται για να συνδέσει περιφερειακά υψηλών ταχυτήτων, όπως τους ελεγκτές DMA, και τη μνήμη που είναι ενσωματωμένη στο τσιπ με τον επεξεργαστή. Συγκρούσεις μεταξύ των κρυφών μνημών δεδομένων και εντολών επιλύονται αφού μόνο μια διεπαφή ελεγκτή συνδέεται κάθε φορά στην αρτηρία.

Το Advanced Peripheral Bus (AMBA-2.0 APB) είναι χαμηλότερης πολυπλοκότητας και έχει βελτιωθεί για μικρότερη κατανάλωση ενέργειας για την επικοινωνία με τα περιφερειακά, κυρίως βοηθητικά ή γενικού σκοπού.

3.9. Διεπαφές Μνήμης

Οι ελεγκτές μνήμης υποστηρίζουν :

- PROM
- Στατική RAM (SRAM)
- Σύγχρονη δυναμική RAM (SDRAM)
- I/O με απεικόνιση στη μνήμη

3.10. Ελεγκτής διακοπών

Ο ελεγκτής των διακοπών μπορεί να χειριστεί συνολικά 15 διακοπές, που προέρχονται από εσωτερικές ή εξωτερικές πηγές.

3.11. Επιπλέον Μονάδες

- Μονάδα υποστήριξης αποσφαλμάτωσης(DSU)
- Διεπαφή PCI
- Ethernet MAC 10/100 Mbit

Επιπλέον υπάρχει και επιλογή για λειτουργία power-save όπου μετά την εμφάνιση κατάλληλου interrupt η μονάδα επεξεργασίας (IU , FPU) ξυπνάει για να συνεχίσει την λειτουργία της.

3.12. Διαμόρφωση(configuration)

Υπάρχουν διαφορετικοί τρόποι για να επιλέξουμε την διαμόρφωση του επεξεργαστή LEON2. Μπορούμε να ξεκινήσουμε το γραφικό περιβάλλον που παρέχεται από την Gaisler, για τον σκοπό αυτό, με την εντολή make xconfig στο cygwin και μέσω αυτού να επιλέξουμε την διαμόρφωση, ακολούθως εκτελούμε make dep για την αποθήκευση των αλλαγών. Μια άλλη πιο αυτοματοποιημένη επιλογή είναι να χρησιμοποιήσουμε τα scripts που υπάρχουν για να διαμορφωθεί ο επεξεργαστής κατάλληλα ανάλογα με κάποιο από τα board που υποστηρίζονται. Αυτό μπορεί να γίνει με την εντολή make config BOARD=gr-xc3s1500 στο cygwin για το board της Pender με την ενσωματωμένη FPGA Spartan 3 με το οποίο και θα δουλέψουμε. Τέλος υπάρχει και η δυνατότητα να γίνει η διαμόρφωση κατευθείαν στον κώδικα vhdl του επεξεργαστή κάνοντας μετατροπές στο αρχείο device.vhd, που περιέχει την διαμόρφωση του επεξεργαστή. Η διαμόρφωση που τελικά επιλέχθηκε για να δουλέψουμε παρουσιάζεται στο παράρτημα Α.

4. Θέματα υλοποίησης

4.1. Γενικά

Στην υποπαράγραφο αυτή θα ασχοληθούμε με μια εμβάθυνση στο θέμα της υλοποίησης των πρωτοκόλλων υποκλοπής τόσο για τις write-through όσο και για τις write-back κρυφές μνήμες και θα εξηγήσουμε κάποιες αλλαγές που πρέπει να γίνουν στα δύο πρωτόκολλα που μελετήσαμε για να είναι πιο ολοκληρωμένα[2].

Είναι γενικά αποδεδειγμένο ότι οι διαφορές σε απόδοση και κόστος που παρατηρούνται στις διάφορες υλοποιήσεις συστημάτων πολυεπεξεργαστών, ακόμα και αν αυτοί ανήκουν στην ίδια κατηγορία π.χ συμμετρικοί, δεν οφείλονται στα διαφορετικά πρωτόκολλα που χρησιμοποιούνται αλλά κυρίως στην διαφορετική οργάνωση και υλοποίηση των δομών που χρησιμοποιούνται για την υποστήριξη του πρωτοκόλλου. Χαρακτηριστικά, όπως το εύρος ζώνης και η καθυστέρηση που επιτυγχάνονται από ένα πρωτόκολλο εξαρτώνται από τον σχεδιασμό του bus, της κρυφής μνήμης και από άλλες σχεδιαστικές παραμέτρους.

Μέχρι τώρα έχουμε δει πως δουλεύει ένα snooping πρωτόκολλο ακύρωσης από ένα υψηλό επίπεδο, χωρίς να λάβουμε υποψιών κάποια θέματα που γεννιούνται κατά την προσπάθεια υλοποίησης, και όπως είναι γνωστό σύμφωνα με το ρητό ΄΄Devil is in the details΄΄. Γενικά μια υλοποίηση πρέπει να έχει τρεις απώτερους σκοπούς. Με σειρά σημαντικότητας:

1)ορθότητα, 2)υψηλή απόδοση, 3)ελαχιστοποίηση απαιτούμενου υλικού.

Τα ζητήματα που έχουν να κάνουν με την ορθότητα αφορούν κυρίως λειτουργίες που ενώ σε υψηλό επίπεδο θεωρούνται ατομικές τελικά δεν σημαίνει ότι θα είναι όντως ατομικές σε επίπεδο υλοποίησης- υλικού. Τα ζητήματα απόδοσης προκύπτουν γιατί ενώ θέλουμε να υπάρχει επικάλυψη των λειτουργιών μνήμης αντί να περιμένουμε να ολοκληρωθεί μια λειτουργία πριν προχωρήσουμε στην επόμενη. Σε αυτό το σημείο όμως που επιδιώκουμε απόδοση πολλές φορές βλάπτουμε την ορθότητα για την επίτευξη της αφού εισάγουμε πολύπλοκες λειτουργίες. Άρα η απόδοση περιορίζεται από τον παράγοντα της ορθότητας. Τέλος η απαίτηση για υψηλότερη απόδοση και ορθότητα στο σύστημα πολυεπεξεργαστή εισάγει ακόμα μεγαλύτερη πολυπλοκότητα σε σχέση με το

υλικό, γεγονός που καθιστά προφανώς το κόστος υλοποίησης του συστήματος μεγαλύτερο.

Άρα θα πρέπει να μελετήσουμε το σύστημα σε μεγαλύτερο βάθος για να μπορέσουμε να εντοπίσουμε τα προβλήματα και να προχωρήσουμε στην καταλληλότερη υλοποίηση.

4.2. Ατομικό bus-Κρυφή μνήμη ενός επιπέδου

Καθώς προχωράμε σε θέματα υλοποίησης θα εξετάσουμε μια πιο ρεαλιστική προσέγγιση του πρωτοκόλλου. Θεωρούμε ότι έχουμε κρυφή μνήμη ενός επιπέδου και ατομικό bus. Μερικές από τις σχεδιαστικές αποφάσεις που πρέπει να παρθούν είναι το πως θα υλοποιηθούν οι ετικέτες(tags) των μπλοκ κρυφής μνήμη και πως θα συμπεριφέρεται ή τι επεκτάσεις πρέπει να γίνουν στο ελεγκτή της κρυφής μνήμης για να υποστηρίξει την προσπέλαση και από τον επεξεργαστή αλλά και από το bus για να λειτουργήσει με επιτυχία το snooping πρωτόκολλο. Άλλη μια απόφαση που πρέπει να παρθεί είναι το πότε και σε τι μορφή θα εμφανίζονται τα αποτελέσματα από μια υποκλοπή.

Για να δούμε τι αλλαγές χρειαζόμαστε στα tags των μπλοκ κρυφής μνήμης αλλά και στον ελεγκτή, ας δούμε λίγο την λειτουργία ενός συμβατικού επεξεργαστή. Όταν λάβει μια λειτουργία μνήμης, ένα μέρος της διεύθυνσης της εντολής χρησιμοποιείται για την προσπέλαση του συνόλου της κρυφής μνήμης που ενδεχομένως περιέχει το ζητούμενου μπλοκ. Το tag συγκρίνεται με τα εναπομείναντα bits διεύθυνσης για να διευκρινιστεί αν το ζητούμενο μπλοκ είναι όντως παρόν. Αν είναι, εκτελείται η κατάλληλη ενέργεια και ανανεώνονται τα bits κατάστασης(state bits). Για παράδειγμα μία ευστοχία εγγραφής σε ένα μπλοκ, γράφει μια λέξη δεδομένων και ανανεώνει την κατάσταση του μπλοκ σε modified. Σε περίπτωση που μια λειτουργία απαιτεί την μεταφορά ενός μπλοκ από την κρυφή μνήμη στην κύρια ή το αντίθετο, τότε ο ελεγκτής θα πρέπει να ξεκινήσει μια συναλλαγή στο bus.

Αφού, λοιπόν θέλουμε να υποστηρίξουμε ένα snooping πρωτόκολλο θα πρέπει προφανώς να κάνουμε κάποιες μετατροπές στον ελεγκτή κρυφής μνήμης. Ο ελεγκτής θα πρέπει να έχει τη δυνατότητα να εξυπηρετεί τις αιτήσεις του επεξεργαστή αλλά και ταυτόχρονα να μπορεί να υποκλέπτει τις διευθύνσεις που εμφανίζονται στο bus και να τις συγκρίνει με τις διευθύνσεις που έχει στην

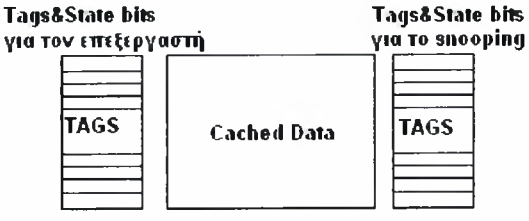
κρυφή του μνήμη. Και για τις δύο αυτές περιπτώσεις χρειαζόμαστε πρόσβαση στα tags της κρυφής μνήμης για να μπορούμε να ικανοποιήσουμε τις αιτήσεις. Είναι ουσιαστικά σαν να έχουμε δύο ελεγκτές ένα που να εξυπηρετεί τον επεξεργαστή και ένα που να παρακολουθεί το bus. Σε κάθε συναλλαγή που συμβαίνει στο bus ο ελεγκτής θα πρέπει να υποκλέπτει την διεύθυνση και αφού δει αν ταιριάζει με κάποια από τα tags που υπάρχουν στην κρυφή μνήμη να μπορεί να παρέμβει με κάποια ενέργεια ανάλογα με το πρωτόκολλο στην συναλλαγή σε περίπτωση που έχει τελικά την ζητούμενη διεύθυνση(ευστοχία υποκλοπής - **snoop hit**), π.χ να κάνει flush τα δεδομένα που υπάρχουν εκείνη την στιγμή στο bus για να δώσει αυτός τα πιο ενημερωμένα που πιθανώς έχει στην κρυφή του μνήμη. Σε περίπτωση που έχουμε αστοχία υποκλοπής (snoop miss) δεν χρειάζεται να κάνει κάποια ενέργεια.

Προφανώς από την απαίτηση που είδαμε πιο πάνω για την λειτουργία του snooping πρωτοκόλλου, θα πρέπει να γίνουν και κάποιες βελτιώσεις στην μορφή της q πληροφορίας που αποθηκεύεται στη κρυφή μνήμη. Δηλαδή, μέχρι τώρα σε ένα μονοεπεξεργαστή είχαμε ένα tag για κάθε λέξη δεδομένων και έτσι δεν μπορεί να προσπελαύνουν και οι δύο ελεγκτές (θεωρητικά του επεξεργαστή και του bus). Με αυτόν τον τρόπο θα πρέπει κάθε φορά που θέλει για παράδειγμα ο επεξεργαστής να προσπελάσει την κρυφή μνήμη να κλειδώνεται το snooping από έξω και να καθυστερούν έτσι οι συναλλαγές της μνήμης και άρα να καταναλώνεται άσκοπα το εύρος ζώνης του bus. Από την άλλη αν δεν επιτρέψουμε την είσοδο στον επεξεργαστή προς εύνοια του snooping, τότε ρίχνουμε αισθητά την απόδοση του επεξεργαστή. Γι αυτό θα πρέπει να δοθεί μια λύση για να μπορούν να γίνονται οι δύο ενέργειες ταυτόχρονα.

Η λύση σε αυτό το πρόβλημα μπορεί να επιτευχθεί είτε χρησιμοποιώντας μια dual port RAM είτε διπλασιάζοντας τα αντίγραφα των tags του κάθε μπλοκ αλλά όχι και τα bit που αντιστοιχούν στα δεδομένα καθώς οι προσβάσεις που μας περιορίζουν είναι οι προσβάσεις για ταυτόχρονο ψάξιμο για ίδια tags και όχι προσπέλαση των δεδομένων, που δεν είναι τόσο συχνή. Έτσι με την χρήση dual ported RAM μπορούν δύο ταυτόχρονα ελεγκτές να προσπελαύνουν τα ίδια tags, ενώ με τη λύση του διπλασιασμού των tags και πάλι δύο ελεγκτές μπορούν να κάνουν ταυτόχρονους ελέγχους, προσπελαύνοντας τώρα όμως τα δύο πανομοιότυπα αντίγραφα αντί το ίδιο. Προφανώς στην περίπτωση των διπλασιασμένων tags σε περίπτωση κάποια ενημέρωσης ή αλλαγής στα state

bits, θα πρέπει να ενημερώνονται και τα δύο αντίγραφα οπότε και οι ελεγκτές θα πρέπει κατά τη διάρκεια της ενημέρωσης να κλειδώνονται εκτός, γεγονός που καταναλώνει όμως πόρους. Φυσικά υπάρχουν τρόποι μείωσης του χρόνου κατά το οποίο οι ελεγκτές μένουν κλειδωμένοι εκτός, τους οποίους δεν θα αναπτύξουμε.

σχήμα 12



Μια ιδέα της οργάνωσης των κρυφών μνημών που χρησιμοποιούνται για πρωτόκολλα υποκλοπής μπορούμε να πάρουμε στο σχήμα 12, όπου έχουμε μια κρυφή μνήμη ενός επιπέδου με διπλά αντίγραφα των tags και state bits. Το ένα σετ χρησιμοποιείται για προσπέλαση από τον επεξεργαστή ενώ το άλλο για προσπέλαση από τον μηχανισμό που εκτελεί το snooping. Για οποιαδήποτε αλλαγή ενημερώνονται και τα δύο αντίγραφα. Αντίστοιχα, όταν έχουμε dual-ported RAM έχουμε βέβαια ένα αντίγραφο της πληροφορίας του tag και των data, υπάρχουν όμως και δυο πόρτες ανάγνωσης, δηλαδή μπορούν να γίνουν δύο ταυτόχρονες αναγνώσεις στο ίδιο πεδίο και να επιστραφούν τα δεδομένα σε διαφορετικές μονάδες που τα αιτήθηκαν.

Ας προχωρήσουμε τώρα σε ένα ακόμα θέμα που θα πρέπει να μας απασχολήσει. Το MESI πρωτόκολλο όπως έχουμε πει είναι ένα snooping protocol για write back κρυφές μνήμες. Αυτό που πρέπει να εξεταστεί είναι τι βελτιώσεις χρειαζόμαστε για να την υλοποίηση. Το write back περιλαμβάνει εκτός από τα εισερχόμενα μπλοκ και μπλοκ τα οποία πρέπει μετά από αστοχία να ενημερωθούν στην κύρια μνήμη, δηλαδή όταν γίνει κάποια αστοχία κρυφής μνήμης θα πρέπει να φορτωθεί πρώτα το μπλοκ που προκάλεσε την αστοχία στην κρυφή μνήμη και μετά να γίνει η ενημέρωση στην κύρια μνήμη με το μπλοκ που αντικαθίσταται. Για να μπορέσει ο επεξεργαστής να συνεχίσει όσο γρηγορότερα γίνεται από μια αστοχία, θα θέλαμε να καθυστερήσουμε τη συναλλαγή write back.

Για να γίνει αυτό χρειαζόμαστε πρώτα από όλα προσωρινό μέσο αποθήκευσης, τον *write-back buffer*, όπου το μπλοκ που θα αντικατασταθεί να

αποθηκεύεται προσωρινά μέχρι να φορτωθεί στην κρυφή μνήμη το καινούριο μπλοκ και πριν προχωρήσουμε σε δεύτερη συναλλαγή στο bus να ολοκληρώσουμε το write back. Όμως υπάρχει πάντα η περίπτωση όσο το μπλοκ που θα γίνει write back βρίσκεται ακόμη στο *buffer* να δει ο ελεγκτής κάποια συναλλαγή στο bus που αφορά την συγκεκριμένη διεύθυνση, οπότε και θα πρέπει ο *buffer* να είναι σε θέση να παρέχει τα δεδομένα αυτός στο bus και να ακυρώσει την διαδικασία για το write back που εκκρεμούσε, αφού τώρα θα υπάρχει αντίγραφο και σε άλλη κρυφή μνήμη και άρα είναι περιττό να γίνει και το write back. Για την βελτίωση αυτή χρειαζόμαστε και μια μονάδα σύγκρισης διευθύνσεων στο *buffer* για την λειτουργία του snooping.

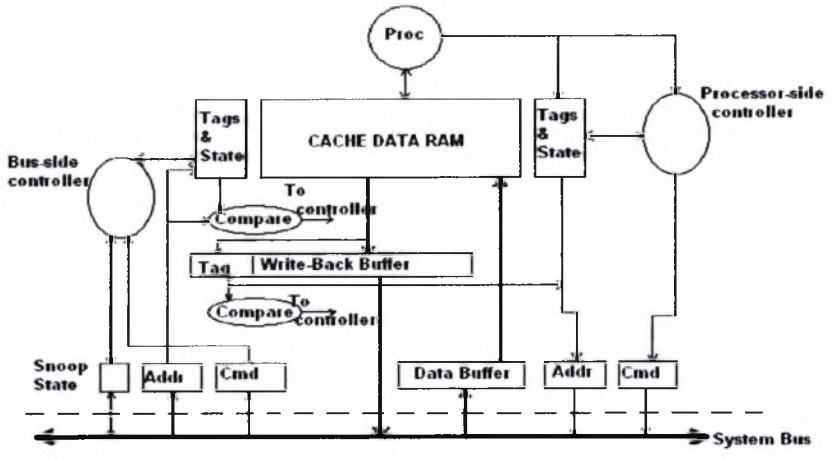
4.3. Βελτιωμένη Αρχιτεκτονική

Στο παρακάτω σχήμα μπορούμε να δούμε την τελική αρχιτεκτονική που χρησιμοποιούμε για να ικανοποιήσουμε την ορθότητα στο πρωτόκολλο MESI. Σε αυτό το σχήμα οι επεξεργαστές έχουν κρυφή μνήμη ενός επιπέδου, διπλά αντίγραφα των tags, για την προσπέλαση των μισών από τον ελεγκτή από την μεριά του bus για την διεξαγωγή του snooping και την προσπέλαση των άλλων μισών από τον επεξεργαστή. Ο ελεγκτής του επεξεργαστή μπορεί να ξεκινήσει μια συναλλαγή τοποθετώντας στο bus μια διεύθυνση. Σε περίπτωση write back τα δεδομένα τοποθετούνται στο write-back buffer. Ο ελεγκτής από την μεριά του bus που κάνει snooping συγκρίνει την διευθύνσεις που υποκλέπτει από το bus και με την διεύθυνση που βρίσκεται στο write-back buffer, εκτός από την σύγκριση με τις διευθύνσεις στην κρυφή μνήμη, για να μπορεί να αντεπεξέλθει στην περίπτωση που κάποιος επεξεργαστής ζητήσει το μπλοκ που είναι προς αντικατάσταση. Όπως μπορούμε να δούμε υπάρχει και ο data buffer που χρησιμοποιείται όταν έχουμε μια συναλλαγή ανάγνωσης στο bus.

Οι αιτήσεις φτάνουν στο bus σε καθολική σειρά. Όπως μπορούμε να δούμε και στο σχήμα οι υποθέσεις που κάναμε για το πρωτόκολλο MESI και φαίνονται στο σχήμα *11* ότι κάποιες ενέργειες, όπως οι προσβάσεις στο bus, συμβαίνουν αμέσως και ατομικά δεν ισχύουν. Βλέποντας το θέμα σε μεγαλύτερο βάθος όπως στο σχήμα *13* παρατηρούμε ότι μια αίτηση από τον επεξεργαστή δεν ολοκληρώνεται αμέσως όπως υποθέταμε μέχρι τώρα, αλλά της παίρνει κάποιο χρόνο και μπορεί συχνά να περιλαμβάνει και κάποια συναλλαγή στο bus. Ακόμα και με ατομικό bus βλέπουμε

ότι συνολικά η πράξη δεν είναι ατομική αφού αποτελείται από επιμέρους ενέργειες, στις οποίες μπορούν να παρεμβληθούν ενέργειες από άλλους επεξεργαστές.

σχήμα 13



Για παράδειγμα, έστω ότι έχουμε δυο επεξεργαστές P1 και P2, οι οποίοι έχουν το μπλοκ Α στην κρυφή του μνήμη ο καθένας. Αν τώρα ο P1 θέλει να γράψει το Α θα πρέπει να τοποθετήσει μια BusUpgr(ακύρωσε τα αντίγραφα χωρίς να φορτώσεις δεδομένα από την κύρια μνήμη- μια βελτιστοποίηση σε σχέση με την αίτηση BusRdX) στο bus και να αλλάξει την κατάσταση του μπλοκ από shared σε modified. Αν όμως ταυτόχρονα και P2 έχει εκδηλώσει και αυτός την πρόθεση να γράψει και έχει κερδίσει στο arbitration το bus, θα προλάβει να κάνει αυτός BusUpgr, το οποίο θα δει ο ελεγκτής του P1 και θα ακυρώσει το αντίγραφο του και θα το έχει έτσι σε invalid κατάσταση. Αλλά ακόμη εκκρεμεί η εγγραφή του P1. Τώρα όμως η αίτηση BusUpgr δεν ισχύει και πρέπει να αντικατασταθεί από μια αίτηση BusRdX. Έτσι ο ελεγκτής πρέπει να είναι σε θέση να υποκλέπτει τις διευθύνσεις και να τις συγκρίνει με την δική του αίτηση που εκκρεμεί και να μπορεί να κάνει τις αλλαγές που χρειάζεται.

Για την σωστή αντιμετώπιση αυτών των φαινομένων που στην αρχική μας προσέγγιση στο θέμα δεν είχαμε υπολογίσει θα πρέπει να κάνουμε βελτιώσεις και επεκτάσεις στο αρχικό FSM με κάποιες *μεταβατικές* καταστάσεις. Οι μεταβατικές καταστάσεις αυτές που δεν υπήρχαν στο προηγούμενο FSM είναι οι:

1) S->M

2) I->M

3) I->S, E

Οι ονομασίες προκύπτουν από τα ονόματα των καταστάσεων από τα οποία μεταβαίνουμε στην μεταβατική κατάσταση και των καταστάσεων στις οποίες καταλήγουμε από αυτές.

Με τις μεταβατικές καταστάσεις έχουμε κάποια επιπλέον σήματα τα οποία είναι

Αίτηση για Bus(**BusReq**): Ξεκινάει την διαδικασία για δέσμευση του bus μέσω arbitration.

Κατοχύρωση Bus(**BusGrant**): Κατοχυρώνει την δέσμευση του bus σε κάποιο επεξεργαστή για να εκτελέσει ατομικά την συναλλαγή του.

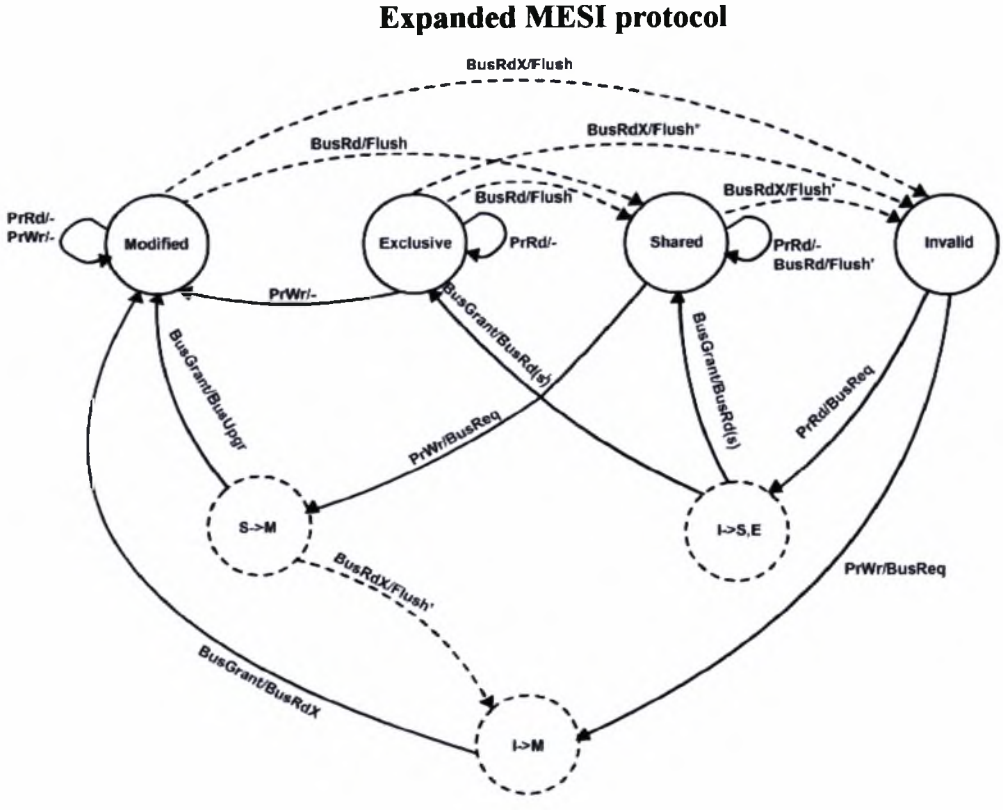
Τέλος, όπως αναφέρθηκε πιο πάνω, ορίζεται η συναλλαγή BusUpgr στο bus, η οποία ακυρώνει τα υπόλοιπα αντίγραφα στις κρυφές μνήμες, όπως και το BusRdX, αλλά δεν διαβάζει δεδομένα από την κύρια μνήμη αφού η συναλλαγή χρησιμοποιείται όταν υπάρχει ήδη ανανεωμένο αντίγραφο στην τοπική κρυφή μνήμη του επεξεργαστή.

Γενικά, οι μεταβατικές αυτές καταστάσεις δεν κωδικοποιούνται στα state bits του μπλοκ κρυφής μνήμης, τα οποία παραμένουν για να συμβολίζουν τις σταθερές καταστάσεις που υπήρχαν μέχρι τώρα στο MESI. Οι μεταβατικές καταστάσεις φαίνονται στο συνδυασμό των state bits και της κατάστασης του ελεγκτή κρυφής μνήμης.

Στο επεκτεταμένο MESI πρωτόκολλο που έχουμε εδώ, ισχύει ότι ίσχυε και στο απλό MESI πρωτόκολλο για τις μεταβάσεις με κάποιες αλλαγές που θα αναλυθούν παρακάτω. Αν ένα μπλοκ είναι στην κρυφή μνήμη σε κατάσταση invalid ή απουσιάζει εντελώς(άρα είναι πάλι σε invalid), τότε σε περίπτωση που θέλει ο επεξεργαστής αυτός να γράψει το μπλοκ θέτει το σήμα BusReq και έτσι ξεκινάει το arbitration για το bus και το μπλοκ μεταβαίνει στην *I->M* κατάσταση περιμένοντας να του δοθεί το bus για χρήση. Μόλις αποφασιστεί από το arbitration ότι θα πάρει το bus στέλνεται ένα σήμα BusGrant για αυτόν τον επεξεργαστή και έτσι μπορεί τώρα ο τοπικός ελεγκτής να τοποθετήσει ένα σήμα BusRdX στο bus για να ακυρώσουν όλοι οι υπόλοιποι επεξεργαστές αν έχουν το αντίγραφο και να το φορτώσει η τοπική κρυφή μνήμη από την κύρια ώστε να το γράψει. Έτσι το μπλοκ βγαίνει από την μεταβατική κατάσταση και μεταβαίνει στην κατάσταση modified. Παρομοίως σκεφτόμαστε και για τις υπόλοιπες

μεταβάσεις. Στο παρακάτω σχήμα (*σχήμα 14*) βλέπουμε αναλυτικά το πρωτόκολλο.

σχήμα 14



4.4. *Ζητήματα που προκύπτουν με το νέο πρωτόκολλο*

Εξαιτίας της μη-ατομικότητας των λειτουργιών μνήμης που εκδίδονται από διαφορετικούς επεξεργαστές, πρέπει να δοθεί ιδιαίτερη προσοχή για να μην αλλοιωθεί η σειρά που καθορίζεται από την σειριοποίηση των συναλλαγών του bus. Για μια ανάγνωση ο επεξεργαστής περιμένει την ολοκλήρωση και το αποτέλεσμα της λειτουργίας. Για μια εγγραφή μπορούμε να ανανεώσουμε το μπλοκ και να αφήσουμε τον επεξεργαστή να συνεχίσει να εκτελεί εντολές όσο ο ελεγκτής κρυφής μνήμης προσπαθεί να αποκτήσει την αποκλειστική ιδιοκτησία του μπλοκ. Το πρόβλημα είναι ότι υπάρχει ένα χρονικό διάστημα από την στιγμή που ο επεξεργαστής μεταβιβάζει την αίτηση για εγγραφή στην κρυφή μνήμη και την στιγμή που ο ελεγκτής δεσμεύει το bus για να τοποθετήσει σε αυτό την συναλλαγή BusRdX. Σε αυτό το χρονικό διάστημα μπορεί να εμφανιστούν άλλες συναλλαγές που να αλλάξουν την κατάσταση του μπλοκ. Αυτές οι ενέργειες κάνουν την διατήρηση της ακολουθιακής συνέπειας πιο δύσκολη. Μια συντηρητική λύση θα ήταν να μην επιτρέπει ο ελεγκτής κρυφής μνήμης που

εκδίδει την εγγραφή να ολοκληρώνει άλλες λειτουργίες μετά από αυτό σε σειρά προγράμματος μέχρι να γίνει ορατή η εγγραφή στους υπόλοιπους επεξεργαστές.

Στην πραγματικότητα δεν χρειάζεται η κρυφή μνήμη να περιμένει μέχρι την ολοκλήρωση της συναλλαγής BusRdX πριν αφήσει τον επεξεργαστή να προχωρήσει. Ο επεξεργαστής μπορεί να εξυπηρετεί λειτουργίες μνήμης μόλις η προηγούμενη συναλλαγή είναι στο bus. Η βάση για την διατήρηση της συνοχής και της συνέπειας είναι ότι όλοι οι ελεγκτές κρυφής μνήμης βλέπουν τις συναλλαγές BusRdX και BusUpgr που παράγονται κατά την εγγραφή κάποιου μπλοκ με την σειρά που έγινε η παραγωγή τους και τα δεδομένα γράφονται στην κρυφή μνήμη αμέσως μετά την συναλλαγή BusRdX. Στο σχήμα που χρησιμοποιούμε ο επεξεργαστής που κάνει την εγγραφή δεν γνωρίζει ακριβώς πότε εμφανίζεται η ακύρωση που πυροδότησε ο ίδιος στην σειρά προγράμματος των άλλων επεξεργαστών, γνωρίζει μόνο ότι γίνεται πριν από οποιαδήποτε άλλη λειτουργία που παράγει συναλλαγή στο bus και ότι όλοι οι επεξεργαστές τοποθετούν ακυρώσεις με την ίδια σειρά. Ομοίως οι ευστοχίες κρυφής μνήμης που έπονται γίνονται ορατές στην επόμενη συναλλαγή του bus.

Αυτή η βασική παρατήρηση είναι αυτή που είναι σημαντική για την διατήρηση της σωστής σειράς που προϋποθέτει η συνοχή και η ακολουθιακή συνέπεια και που ευθύνεται για την ορθότητα για σχήματα με πολυεπίπεδες ιεραρχίες κρυφής μνήμης, αρτηρίες διαιρεμένων συναλλαγών (split transaction buses).

Από αυτή την ανάλυση μπορούμε να δούμε ότι η σειριοποίηση εγγραφής και η ατομικότητα εγγραφής δεν έχει τόσο πολύ σχέση με το πότε κάνουν write back οι συναλλαγές στην κύρια μνήμη ή πότε ενημερώνεται μια συγκεκριμένη θέση μνήμης. Ένα write back μπορεί να προκληθεί είτε από μια εγγραφή είτε από μια ανάγνωση. Μια εγγραφή ακόμη και σε περίπτωση που γίνει αστοχία δεν χρειάζεται να εμφανίσει την τιμή στο bus παρά μόνο να προκαλέσει μια BusRdX συναλλαγή. Έτσι η εγγραφή ολοκληρώνεται ουσιαστικά αφού οποιαδήποτε μεταγενέστερη ανάγνωση θα επιστρέψει την ενημερωμένη τιμή εφόσον έχει προηγηθεί η ακύρωση από την BusRdX ή την BusUpgr.

4.5. Υλοποίηση ατομικών λειτουργιών

Μια επιπλέον παράμετρος που θα εξετάσουμε σε μεγαλύτερο βάθος είναι τις αποφάσεις που πρέπει να πάρουμε για να υλοποιήσουμε ατομικές λειτουργίες για τον συγχρονισμό ενός πολυεπεξεργαστικό συστήματος[2].

Έστω ότι έχουμε μια test&set εντολή, ένα πιθανό ερώτημα είναι εάν θα πρέπει τα locks να είναι στην κρυφή ή την κύρια μνήμη. Αν θεωρήσουμε τις μεταβλητές lock cacheable, μειώνεται η καθυστέρηση και η κίνηση αν το lock το παίρνει επαναλαμβανόμενα ο ίδιος επεξεργαστής, αφού η μεταβλητή παραμένει στην modified κατάσταση χωρίς να παράγονται αστοχίες ή ακυρώσεις. Επιπλέον το ΄΄spinning΄΄ από κάθε επεξεργαστή γίνεται στην κρυφή του μνήμη μειώνοντας την κίνηση στο bus.

Από την άλλη μεριά αν τα locks είναι στην κύρια μνήμη μειώνεται ο χρόνος μεταφοράς του lock από τον ένα επεξεργαστή στον άλλο. Με cacheable locks, ο επεξεργαστής που είναι σε ΄΄spinning΄΄ πρέπει να ακυρωθεί και σε εκείνο το σημείο να προσπαθήσει να πάρει το lock είτε από την κρυφή άλλου επεξεργαστή είτε από την κύρια μνήμη. Με τα locks στην κύρια μνήμη δεν χρειάζεται ακύρωση και εφόσον υπάρχει επεξεργαστής ΄΄spinning΄΄ στην κύρια μνήμη θα πάρει το lock με μικρή καθυστέρηση. Γενικά πάντως η τοπικότητα και η μειωμένη κίνηση δίνουν μεγαλύτερα κέρδη και γι αυτό προτιμάται να είναι cached τα locks.

Ένας τρόπος για να υλοποιήσουμε μία cacheable test&set που δεν ικανοποιείται στην κρυφή μνήμη είναι με μία συναλλαγή στο bus για ανάγνωση και άλλη μία για εγγραφή. Για να είναι αυτή η διαδικασία ατομική πρέπει να κρατήσουμε κλειδωμένο το bus μέχρι την ολοκλήρωση της εγγραφής, το οποίο αν και είναι εφικτό με ατομικό bus, δεν είναι τόσο απλό για split-transaction bus. Μια προσέγγιση που λύνει το πάνω πρόβλημα σε write-back caches είναι να εκτελεί ο επεξεργαστής την ανάγνωση και εγγραφή εάν έχει αποκτήσει αποκλειστική ιδιοκτησία του lock. Ακόμα και σε μη-ατομικό bus εισερχόμενες αναφορές από το bus για το lock αυτό κρατούνται στο buffer μέχρι την εγγραφή των δεδομένων στην κρυφή μνήμη.

Η υλοποίηση τώρα μιας LL-SC απαιτεί ειδική μεταχείρηση. Σε μια τυπική υλοποίηση η LL διαβάζει μια διεύθυνση και την τοποθετεί σε καταχωρητή και θέτει lock flag. Τυχόν ακυρώσεις από το bus μηδενίζουν το flag. Το SC αν το flag δεν είναι μηδενισμένο ολοκληρώνεται, διαφορετικά αποτυγχάνει. Ένα θέμα

που προκύπτει αν το lock flag είναι υλοποιημένο στο μπλοκ της κρυφής μνήμης, π.χ στα state bits του μπλοκ, είναι ότι πιθανή αντικατάσταση του μπλοκ μπορεί να προκαλέσει τον επεξεργαστή να προσπαθεί να ολοκληρώσει το SC ανεπιτυχώς εξαιτίας της αντικατάστασης. Ένα άλλο livelock που μπορεί να αντιμετωπίσουμε είναι εάν δύο διαδικασίες αποτυγχάνουν διαρκώς στο SC και κάθε τέτοια αποτυχία να προκαλεί μηδενισμό του flag στην άλλη διαδικασία. Γι αυτό το λόγο όταν χρησιμοποιείται το SC, δεν πρέπει να αντιμετωπίζεται ως μια απλή εγγραφή, αλλά ειδικά, με προσεκτικό προγραμματισμό, π.χ εκθετική υπαναχώρηση(exponential backoff) ώστε να μην προκαλεί διαδοχικές αποτυχίες και αδιέξοδο.

5. Υποδομή LEON2 διπλού πυρήνα

5.1. Γενικά

Στο κεφάλαιο αυτό θα ασχοληθούμε με τον επεξεργαστή LEON2 και τις μετατροπές που πρέπει να γίνουν σε αυτόν για να επεκταθεί η αρχιτεκτονική του και να μπορέσει να δημιουργηθεί η υποδομή ώστε να λειτουργήσει ως ένας επεξεργαστής διπλού πυρήνα αναλύοντας τα επιμέρους θεωρητικά κομμάτια και την εφαρμογή τους στον LEON2.

5.2. Πρωτόκολλο *Cache Coherence* για τον μικροεπεξεργαστή *LEON2*

Ας προχωρήσουμε να δούμε τώρα, από την θεωρητική προσέγγιση του θέματος στην πράξη τι μετατροπές πρέπει να γίνουν στον μικροεπεξεργαστή LEON2 της Gaisler για την υποστήριξη δύο επεξεργαστών το οποίο είναι και ζητούμενο για την διπλωματική εργασία αυτή.

Για να γίνει αυτό θα πρέπει να γίνει μια βαθύτερη ανάλυση του μικροεπεξεργαστή από αυτή που κάναμε στο κεφάλαιο 3. Θα πρέπει τώρα να μελετήσουμε σε επίπεδο κώδικα την λειτουργία των ελεγκτών κρυφής μνήμης που υπάρχει, του bus και του πυρήνα του επεξεργαστή και να αναλύσουμε τις υπάρχουσες σχεδιαστικές επιλογές για να γίνει κατανοητό πως μπορούμε από τον μονοεπεξεργαστή να περάσουμε σε ένα συμμετρικό πολυεπεξεργαστή δύο πυρήνων.

Ο LEON2 είναι ένας επεξεργαστής που έχει δημιουργηθεί με προσανατολισμό την εύκολη και αποδοτική υλοποίηση του σε κάποια διάταξη ASIC ή FPGA και είναι σε μεγάλο βαθμό παραμετροποιήσιμος ανάλογα με την τεχνολογία που θα επιλέξουμε να χρησιμοποιήσουμε και με την εφαρμογή στην οποία θέλουμε να χρησιμοποιηθεί. Σε αυτή την λογική ο κώδικας είναι δομημένος πάνω σε συγκεκριμένα δομικά στοιχεία ανάλογα με την τεχνολογία που πρόκειται να χρησιμοποιηθεί και παρέχονται wrappers πάνω σε αυτά ώστε να μπορεί ο προγραμματιστής του υλικού να χειρίζεται τις δομές χωρίς να χρειάζεται γνώση της επιλεγμένης τεχνολογίας.

Για την υλοποίηση ενός cache coherent protocol προφανώς θα πρέπει να μελετήσουμε για αρχή την υπάρχουσα κατάσταση στον επεξεργαστή σε σχέση με το πως υλοποιείται το υποσύστημα κρυφής μνήμης αφού σε αυτό θα πρέπει να υπάρχει η λειτουργικότητα για να υποστηριχτεί το πρωτόκολλο, όπως είδαμε και παραπάνω στην θεωρητική μας προσέγγιση[8].

Καταρχήν ας θυμηθούμε ότι ο LEON2 χρησιμοποιεί την πολιτική write-through no-allocate για τις εγγραφές στην κρυφή του μνήμη. Επίσης η αρχιτεκτονική της κρυφής μνήμης είναι τύπου Harvard, το οποίο σημαίνει ότι υπάρχει ένα υποσύστημα κρυφών μνημών, που αποτελείται από δυο ξεχωριστές κρυφές μνήμες, μια για τις εντολές και μια για τα δεδομένα. Αυτό φυσικά σημαίνει ότι υπάρχουν και δυο ελεγκτές, ένας για κάθε κρυφή μνήμη. Σε επίπεδο κώδικα τώρα τα αρχεία που απαρτίζουν το υποσύστημα κρυφής μνήμης είναι τα[8]:

icache.vhd:	υλοποιεί τον ελεγκτή της κρυφής μνήμης εντολών.
dcache.vhd:	υλοποιεί τον ελεγκτή της κρυφής μνήμης δεδομένων.
acache.vhd:	υλοποιεί το interface μεταξύ των ελεγκτών κρυφής μνήμης εντολών και δεδομένων και του AMBA bus.
cache.vhd:	αποτελεί το αρχείο που συνδυάζει τους ελεγκτές και το interface σε ένα ενιαίο υποσύστημα.

Έκτος αυτών των αρχείων υπάρχει και το αρχείο *cachemem.vhd* το οποίο δεν ανήκει στο υποσύστημα κρυφής μνήμης(σύμφωνα με την οργάνωση του κώδικα ανήκει στον processor) αλλά υλοποιεί τα RAM cells και για τα δύο είδη κρυφών μνημών και άρα είναι ένα από τα αρχεία που μας ενδιαφέρουν.

Από τα παραπάνω αρχεία δεν θα ασχοληθούμε καθόλου με τον κώδικα του αρχείου *icache.vhd* αφού το θέμα της διατήρησης του coherence δεν αφορά τις εντολές αλλά τα δεδομένα, τα οποία και είναι αυτά που μπορεί να είναι κοινά και άρα να γράφονται από περισσότερους τους ενός επεξεργαστές για την μεταξύ τους επικοινωνία. Επίσης το *cache.vhd* δε χρειάζεται να μελετηθεί αφού το μόνο το οποίο κάνει είναι να συνθέτει τα επιμέρους αρχεία των ελεγκτών και το interface τους σε μια δομική υπομονάδα και άρα δεν μπορεί να διαδραματίζει ρόλο για το πρωτόκολλο.

Εξαιτίας της πολιτικής write-through no-allocate για τις εγγραφές που χρησιμοποιεί ο LEON2, το πρωτόκολλο που θα πρέπει να υλοποιηθεί για την υποστήριξη του cache coherence θα είναι το απλό σχήμα ακύρωσης δύο καταστάσεων, οπου οι δυνατές καταστάσεις για ένα μπλοκ μνήμης θα είναι η valid και η invalid. Επιπλέον ο μικροεπεξεργαστής έχει ήδη υλοποιημένο ένα είδος snooping protocol για να μην δημιουργείται κάποιο πρόβλημα συνοχής της κρυφής μνήμης του μονοεπεξεργαστή κατά την προσπέλαση μπλοκ μνήμης από DMA συσκευές.

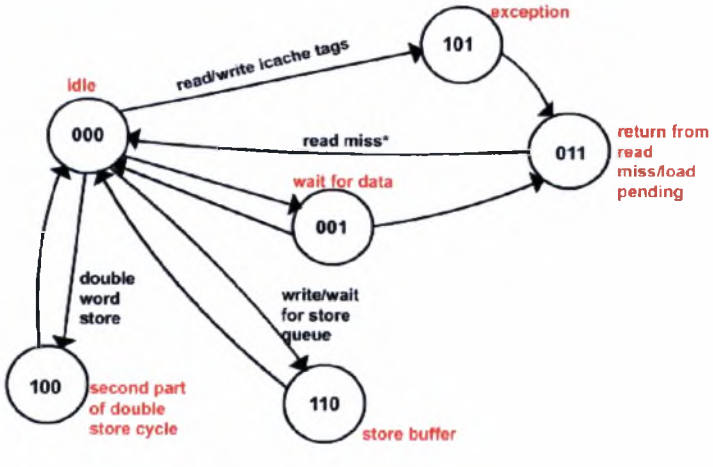
Όπως είδαμε και στην θεωρητική ανάλυση των πρωτοκόλλων για την σωστή υποστήριξη του coherence χρειαζόμαστε και σε αυτή την περίπτωση μια RAM με διπλά αντίγραφα των tags για να μπορεί να γίνεται ταυτόχρονη προσπέλαση εάν χρειαστεί και από τον επεξεργαστή αλλά και από τον μηχανισμό που υποκλέπτει το bus. Ένας άλλος τρόπος για την επίτευξη αυτής της λειτουργικότητας είναι να χρησιμοποιηθούν dual-ported RAMs, δηλαδή μνήμη που θα έχει δυο πόρτες για ανάγνωση και εγγραφή και θα συμπεριφέρεται σαν να υπήρχαν δύο αντίγραφα των tags. Στην περίπτωση του LEON2 μπορούμε να δούμε στο αρχείο *cachemem.vhd* ότι υπάρχει η δυνατότητα να υλοποιηθεί, ανάλογα με την επιλογή αν είναι ενεργοποιημένο ή όχι το snooping, dual-ported RAM ή απλή RAM. Ειδικότερα αν είναι ενεργοποιημένο το σήμα DSNOOP τότε χρησιμοποιούμε το component dpsyncram για να δημιουργήσουμε μια dual-ported RAM, ενώ εάν δεν είναι ενεργοποιημένο τότε χρησιμοποιούμε το απλό component syncram.

Προχωρώντας στην ανάλυση του αρχείου *dcache.vhd* που αποτελεί τον ελεγκτή της κρυφής μνήμης δεδομένων μπορούμε να δούμε ότι δέχεται ως είσοδο δεδομένα από τον επεξεργαστή, που ομαδοποιούνται στο σήμα dci, όπως και δεδομένα που προέρχονται από την κρυφή μνήμη και τα οποία ομαδοποιούνται στο σήμα dcramo. Προφανώς υπάρχουν και οι αντίστοιχες έξοδοι από τον ελεγκτή προς τον επεξεργαστή που είναι το σήμα dco, και από τον ελεγκτή προς την κρυφή μνήμη που είναι το σήμα dcrami και χρησιμοποιούνται για μεταφορά δεδομένων προς τον επεξεργαστή ή εγγραφή δεδομένων στην κρυφή μνήμη αντίστοιχα. Επίσης δέχεται ως είσοδο δεδομένα από το bus, σήμα ahbsi, για να μπορεί ουσιαστικά να υποκλέπτει(snoop) την όποια συναλλαγή γίνεται στο bus και υπάρχει επικοινωνία με το interface ανάμεσα στον ελεγκτή και το bus με τα σήματα mcdo(είσοδος στον ελεγκτή) και mcdi(έξοδος από τον ελεγκτή=>interface).

Αναλογικά με την θεωρητική προσέγγιση που έχουμε δει παραπάνω τις αιτήσεις από τον επεξεργαστή αποτελούν τα σήματα dci και οι απαντήσεις προς αυτόν τα dco, δηλαδή μπορούν να θεωρηθούν ως τα *PrRd* ή *PrWr*. Οι αιτήσεις άλλων επεξεργαστών, παρόλο που στην αρχική υλοποίηση έχουμε μονοεπεξεργαστή, μπορεί να είναι τα σήματα ahbsi, αφού ουσιαστικά αυτά περιέχουν οποιαδήποτε αίτηση εξυπηρετείται στο bus, και προς το παρόν θα προέρχονται από περιφερειακές συσκευές ή αργότερα όταν γίνει η επέκταση με το δεύτερο πυρήνα θα μπορούν να προέρχονται και από αυτόν, δηλαδή μέσα από αυτό μπορούμε να δούμε τα ΄΄θεωρητικά΄΄ *BusRd* ή *BusWr*. Τέλος τα dcramo και dcrami χρησιμοποιούνται για την επικοινωνία στην κρυφή μνήμη (αναγνωσή/εγγραφή/αναζήτηση).

Στην υλοποίηση αυτή του ελεγκτή υπάρχει ένα FSM που καθορίζει την λειτουργία του και τις ενέργειες που εκτελούνται ανάλογα με τις αιτήσεις που έχει δεχτεί. Σύμφωνα με το FSM, το οποίο αποτελείται από έξι καταστάσεις, η λειτουργία του ελεγκτή ξεκινάει και τελειώνει πάντα στην κατάσταση ΄΄000΄΄ η οποία είναι η idle κατάσταση. Από την idle κατάσταση ξεκινάει ουσιαστικά η αποκωδικοποίηση των σημάτων, τα οποία ενεργοποιούν με την σειρά τους επιμέρους σήματα και ελέγχους και τελικά ανάλογα με ποια σήματα ενεργοποιήθηκαν, ο ελεγκτής καταλαβαίνει τι είδους αίτηση έχει δεχτεί και πραγματοποιεί την κατάλληλη μετάβαση εάν χρειάζεται για να την εξυπηρετήσει. Ένα συνοπτικό FSM για την λειτουργία του επεξεργαστή φαίνεται στο *σχήμα15*.

σχήμα 15 **FSM Ελεγκτή Κρυφής Μνήμη Δεδομένων**



*Δεν απεικονίζονται με λεπτομέρεια οι συνθήκες των μεταβάσεων αλλά ένα γενικό πλάνο.

Ακολουθεί μια σύντομη περιγραφή της λειτουργικότητας του FSM. Όπως έχουμε δει η κατάσταση ΄΄000΄΄ είναι η idle από όπου ξεκινάει η εξυπηρέτηση των αιτήσεων και στην οποία επιστρέφουμε μετά το πέρας της ολοκλήρωσης τους:

Για την κανονική λειτουργία της κρυφής μνήμης, αν έχουμε πρόσβαση για ανάγνωση και τελικά γίνει κάποια αστοχία ανάγνωσης στην κρυφή μνήμη τότε μεταβαίνουμε στην κατάσταση ΄΄001΄΄ όπου εξυπηρετείται η αστοχία φορτώνοντας δεδομένα από την κύρια μνήμη. Σε περίπτωση ευστοχίας ανάγνωσης θα δεν χρειάζεται περαιτέρω κίνηση στο FSM. Αν πάλι έχουμε πρόσβαση για εγγραφή και δεν υπάρχει άλλη αίτηση εγγραφής που εκκρεμεί δεν χρειάζεται κάποια μετάβαση εκτός αν τα δεδομένα της εγγραφής είναι διπλής λέξης οπότε έχουμε μετάβαση στην κατάσταση ΄΄100΄΄ για την εξυπηρέτηση της. Αν υπάρχει προηγούμενο αίτηση εγγραφής που εκκρεμεί τότε έχουμε μετάβαση στην ΄΄110΄΄ και περιμένουμε να αδειάσει το buffer.

Η κατάσταση ΄΄001΄΄ εξυπηρετεί τις αστοχίες ανάγνωσης:
--Τα δεδομένα επιστρέφονται από την κύρια μνήμη. Όταν ολοκληρωθεί η ανάγνωση της κύριας μνήμης, αν δεν υπάρχει κάποια αίτηση εγγραφής που εκκρεμεί και δεν μπορούν να επιστραφούν τα δεδομένα απευθείας πηγαίνουμε στην μεταβατική κατάσταση ΄΄011΄΄, διαφορετικά επιστρέφουμε στην idle κατάσταση.

Η κατάσταση ΄΄011΄΄ είναι μεταβατική για επιστροφή στην idle από αστοχία ανάγνωσης με ένα load που εκκρεμεί.

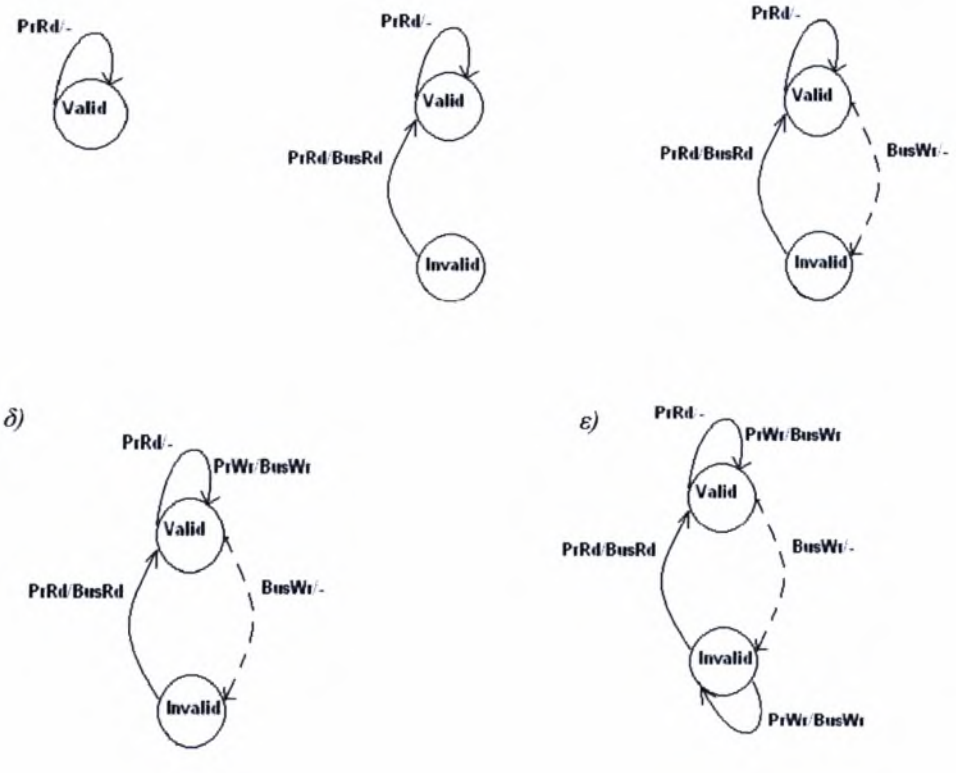
Η κατάσταση ΄΄100΄΄ εξυπηρετεί το δεύτερο μέρος μιας εγγραφής διπλής λέξης και επιστρέφει στην ΄΄000΄΄.

Η κατάσταση ΄΄110΄΄ εξυπηρετεί αιτήσεις εγγραφής όταν το store buffer είναι γεμάτο οπού και περιμένουν να αδειάσει και να μπουν στην ουρά. Μόλις μπουν στην ουρά επιστρέφουμε στην idle κατάσταση.

Εκτός της κανονικής λειτουργίας, ο ελεγκτής της κρυφής μνήμης δεδομένων εξυπηρετεί και διαγνωστική πρόσβαση στην κρυφή μνήμη εντολών από την μονάδα αποσφαλμάτωσης DSU, οπότε το FSM μεταβαίνει σε κατάσταση ΄΄101΄΄. Η κατάσταση ΄΄101΄΄ εξυπηρετεί αιτήσεις για διαγνωστική πρόσβαση στην κρυφή μνήμη εντολών και τα δεδομένα που παρέχονται προφανώς προέρχονται από την κρυφή μνήμη εντολών. Αν είναι έτοιμη η κρυφή μνήμη εντολών για διαγνωστική πρόσβαση τότε πηγαίνουμε στην ΄΄011΄΄.

Επιπλέον μέσα στο *dcache.vhd* υπάρχει κώδικας για τον χειρισμό δεδομένων που υποκλέπτονται(snooping) από το βασικό bus AHB. Στο AHB bus, το οποίο είναι αυτό που συνδέει όπως έχουμε πει περιφερειακά υψηλής ταχύτητας και το υποσύστημα κρυφών μνήμων αλλά και της κύριας μνήμης, παρουσιάζονται όλες οι συναλλαγές εγγραφής που οδηγούνται στην κύρια μνήμη. Αυτό σημαίνει ότι όταν ένας DMA ελεγκτής θελήσει να γράψει στην κύρια μνήμη τότε η συναλλαγή αυτή θα γίνει ορατή στο AHB bus και άρα οποιοσδήποτε άλλος ελεγκτής που ενδιαφέρεται μπορεί με τον κατάλληλο χειρισμό να υποκλέψει την συναλλαγή αυτή και να δράσει ανάλογα. Έτσι στον ελεγκτή της κρυφής μνήμης υπάρχει πρόβλεψη ώστε σε περίπτωση που βρεθεί κάποια διεύθυνση περιοχής μνήμης που είναι cacheable και αφορά στην συγκεκριμένη κρυφή μνήμη, δηλαδή υπάρχει σε αυτή αντίγραφο της συγκεκριμένης διεύθυνσης φορτωμένο από προηγούμενη συναλλαγή, τότε αφού εντοπιστεί από τον ελεγκτή και γίνει η ταυτοποίηση με το πεδίο tag το αντίγραφο αυτό στην κρυφή μνήμη ακυρώνεται, με την έννοια ότι μηδενίζονται τα valid bits. Με αυτό τον τρόπο οποιαδήποτε μετέπειτα προσπέλαση από τον επεξεργαστή στο συγκεκριμένο μπλοκ μνήμης θα προκαλεί αστοχία και θα φορτωθεί εκ νέου από την κύρια μνήμη. Έτσι διασφαλίζεται ότι ο επεξεργαστής δεν θα διαβάσει κάποιο παλιό αντίγραφο της μνήμης, το οποίο έχει γραφεί από DMA περιφερειακό.

σχήμα 16 *α)* *β)* *γ)*



Συνδυάζοντας λοιπόν την υπάρχουσα υποδομή του ελεγκτή(το FSM και τον χειρισμό του snooping) μπορούμε να προχωρήσουμε στις εξής διαπιστώσεις σε σχέση με την δυνατότητα επέκτασης του LEON2 σε πολυεπεξεργαστή δύο πυρήνων:

Οι αναγνώσεις από τον επεξεργαστή (**PrRd**) στην κρυφή μνήμη εξυπηρετούνται από το τρέχον πρωτόκολλο write-through no-allocate που χρησιμοποιείται ήδη. Έτσι οποιαδήποτε ευστοχία ανάγνωσης στην κρυφή μνήμη ολοκληρώνεται χωρίς καμιά επέκταση. Δηλαδή έστω ανάγνωση από τον επεξεργαστή σε μπλοκ το οποίο είναι σε **valid** κατάσταση στην κρυφή μνήμη, τότε απλά επιστρέφονται τα δεδομένα. Άρα αν το οπτικοποιήσουμε σε μορφή FSM παίρνουμε το σχήμα 16-α .

Ας σκεφτούμε τώρα την περίπτωση που έχουμε αστοχία ανάγνωσης στην κρυφή μνήμη, δηλαδή είτε το μπλοκ υπάρχει αλλά είναι **invalid** είτε δεν υπάρχει καθόλου(το οποίο θεωρούμε και πάλι invalid) και βρίσκεται στην κύρια μνήμη. Τότε από το FSM του ελεγκτή βλέπουμε ότι η αστοχία ανάγνωσης φορτώνει δεδομένα από την κύρια μνήμη στην κρυφή στέλνοντας μια αίτηση(**BusRd**) πάνω από το AHB bus στον ελεγκτή της κύριας μνήμης. Άρα συνδυασμένο με την προηγούμενη λειτουργία προκύπτει το FSM του σχήματος 16-β.

Προκύπτει επίσης ότι εφόσον ο ελεγκτής έχει την δυνατότητα να κάνει snooping στο bus και να βλέπει τις εγγραφές οι οποίες όπως είπαμε γίνονται από DMA περιφερειακά, να βλέπει και οποιαδήποτε άλλη εγγραφή(όλες περνούν από το bus). Αυτό σημαίνει ότι αν όπως σκοπεύουμε τοποθετήσουμε ένα δεύτερο επεξεργαστή ο οποίος να συνεργάζεται με την υπάρχουσα υποδομή θα στέλνει εγγραφές(**BusWr**) πάνω από το bus που θα μπορεί να υποκλέψει ο ελεγκτής κρυφής μνήμης και να προχωρήσει σε ακύρωση κάποιου μπλοκ, να γίνει **invalid** αν χρειάζεται. Με λίγα λόγια ο ελεγκτής μπορεί να χειριστεί την υποκλοπή εγγραφής από τον δεύτερο επεξεργαστή σαν να ήταν εγγραφή από DMA συσκευή. Έτσι ακολουθώντας την ίδια τεχνική και συνδυάζοντας με τις έως τώρα λειτουργίες, οπτικοποιείται στο FSM του σχήματος 16-γ.

Σε περίπτωση που έχουμε μια αίτηση εγγραφής από τον επεξεργαστή(**PrWr**) τότε αν το μπλοκ που θέλουμε να γράψουμε βρίσκεται σε **valid** κατάσταση στην κρυφή μνήμη τότε γράφεται το μπλοκ στην κρυφή μνήμη, παραμένει σε valid κατάσταση και επειδή η πολιτική εγγραφής είναι write-through εμφανίζεται η εγγραφή και στο bus για να ενημερωθεί η κύρια μνήμη. Δηλαδή παράγεται η μια

συναλλαγή **BusWr** που γίνεται ορατή σε όλους τους υπόλοιπους ελεγκτές για να ακυρώσουν το αντίγραφο αν υπάρχει (σχήμα 16-δ)

Τέλος επειδή έχουμε write no-allocate πολιτική εγγραφής δεν χρειάζεται όπως έχουμε ήδη δει να φορτωθεί το μπλοκ από την κύρια μνήμη στην κρυφή για να εγγραφεί εάν γίνει κάποια αστοχία εγγραφής στην κρυφή μνήμη. Άρα γι αυτό το λόγο μια αίτηση εγγραφής(**PrWr**) ενός μπλοκ που αστοχεί στην κρυφή μνήμη περνάει αναγκαστικά από το AHB bus για να φτάσει στον ελεγκτή της κύριας μνήμης και να γράψει απευθείας στην κύρια μνήμη, δηλαδή εμφανίζεται στο bus μια συναλλαγή **BusWr** και έτσι καταλήγουμε στο FSM του σχήματος16-ε, το οποίο είναι όμως το FSM το οποίο χρησιμοποιείται για την διατήρηση της συνοχής για write-through no-allocate μνήμες όπως το έχουμε δει στο σχήμα 7.

Το μοναδικό αρχείο το οποίο δεν έχουμε εξετάσει μέχρι τώρα είναι *acache.vhd* που είναι το interface μεταξύ του AHB bus και των ελεγκτών των κρυφών μνημών (εντολών και δεδομένων) δηλαδή ένας εύχρηστος τρόπος για να επικοινωνούν. Αφού δεν κάνουμε κάποια αλλαγή στις εξόδους ή τις εισόδους του κάθε ελεγκτή δεν θα χρειαστεί να αλλάξουμε και κάτι από τον μέχρι τώρα τρόπο επικοινωνίας με το bus.

Συμπερασματικά μπορούμε μετά από την παραπάνω ανάλυση να ισχυριστούμε ότι ο επεξεργαστής LEON2 διαθέτει την υποδομή για να υποστηρίξει έναν ή περισσότερους επεξεργαστές, αφού όπως είδαμε καλύπτει τελικά όλα τα σενάρια που μπορούν να δημιουργηθούν από τις ενέργειες του επεξεργαστή αντιμετωπίζοντας τα με τέτοιο τρόπο ώστε να μπορεί να διατηρήσει την ιδιότητα της συνοχής με δυο επεξεργαστές ακολουθώντας το διάγραμμα μετάβασης δύο καταστάσεων για write-through no-allocate κρυφές μνήμες.

Αυτό που πρέπει να μελετηθεί τώρα είναι το πώς μπορεί να εισαχθεί στο υπάρχον σύστημα ένας ακόμη επεξεργαστής διατηρώντας την λειτουργικότητα. Για να γίνει κάτι τέτοιο πρέπει να μελετηθεί ο τρόπος λειτουργίας του AMBA AHB bus και το τι αλλαγές θα πρέπει πιθανώς να γίνουν για την επικοινωνία του νέου πυρήνα με το υπόλοιπο σύστημα.

5.3. Προσθήκη πυρήνα στον LEON2

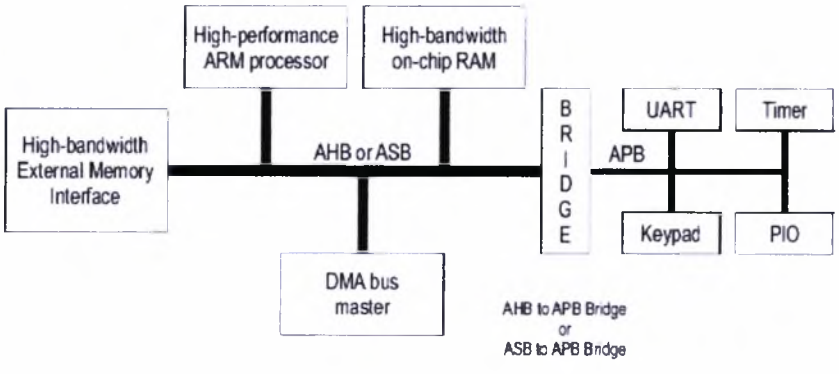
Ο μικροεπεξεργαστής LEON2 είναι δομημένος γύρω από ένα σύστημα από buses, το AMBA. Η ύπαρξη αυτού του συστήματος απλοποιεί την προσθήκη δομικών μονάδων είτε ως masters είτε ως slaves.

Η προδιαγραφή του *Advanced Microcontroller Bus Architecture* AMBA από την ARM ορίζει ένα πρότυπο επικοινωνίας για ενσωματωμένα συστήματα υψηλής απόδοσης [11]. Για τον LEON έχουν υλοποιηθεί τα AHB και APB buses, τα οποία είναι παραμετρόποιήσιμα μέσω του αρχείου *target.vhd*.

Το AHB αποτελεί το backbone bus. Είναι υψηλής απόδοσης και χρησιμοποιείται για να συνδέει μονάδες υψηλής ταχύτητας. Ο επεξεργαστής είναι ο μοναδικός master του bus και ο ελεγκτής κύριας μνήμης μαζί με το APB bridge αποτελούν τους slaves.

Το APB είναι για περιφερειακά χαμηλής κατανάλωσης ενέργειας και είναι βελτιστοποιημένο για ελάχιστη πολυπλοκότητα

σχήμα17 ***Τυπική διάταξη επεξεργαστών βασισμένων στο AMBA***



Έχοντας αναφέρει κάποια βασικά θέματα για την αρχιτεκτονική του bus, θα προχωρήσουμε στον τρόπο με τον οποίο θα προσθέσουμε τον δεύτερο επεξεργαστή. Η δεύτερη επεξεργαστική μονάδα την οποία θα προσθέσουμε αποτελείται από την υπομονάδα ακεραίων και το υποσύστημα κρυφών μνημών χωρίς την υπομονάδα κινητής υποδιαστολής. Ο μοναδικός τρόπος με τον οποίο μπορούμε να συνδέσουμε σωστά την επεξεργαστική μονάδα αυτή, είναι να την συνδέσουμε στο AHB bus προφανώς αφού πρόκειται για υψηλής ταχύτητας μονάδα και επιπλέον θα ανήκει στους masters του bus.

Στο AHB bus μπορούν να συνδεθούν μέχρι 16 masters και απεριόριστος αριθμός από slaves. Ο πρώτος επεξεργαστής(*proc0*) συνδέεται εξ ορισμού ως ο master 0, ενώ ο ελεγκτής κύριας μνήμης και το APB bridge είναι οι slaves 0 και 1 αντίστοιχα.

Σε επίπεδο κώδικα τώρα: Για να συνδέσουμε τον δεύτερο επεξεργαστή(*proc1*) σωστά θα πρέπει να γίνει μια σειρά βημάτων. Αρχικά στο αρχείο *mcore.vhd* θα πρέπει να βρούμε μια κενή θέση για να συνδεθεί ο επεξεργαστής. Γενικά οι masters συνδέονται στο bus με το σήμα ahbmi/ahbmo. Παρατηρούμε ότι εκτός από τον proc0 και η μονάδα dcom όπως και τα pci και eth μπορούν να είναι masters. Για τις ανάγκες της εργασίας μεταφέρουμε τις μονάδες αυτές μια θέση πιο πάνω και ελευθερώνουμε έτσι την θέση 1 στο ahbmi/ahbmo σήμα για τον proc1(στη θέση 0), δίνοντας του έτσι ανάλογη θέση στην προτεραιότητα για το bus. Το υποσύστημα κρυφής μνήμης και κατ' επέκταση ο επεξεργαστής περιλαμβάνει και δύο ειδικούς καταχωρητές, τους Cache Control Register(*CCR*) και Power-Down Register(*PWD*), οι οποίοι πρέπει να συνδεθούν σωστά για να λειτουργούν. Τα σήματα apbi/apbo χρησιμοποιούνται για την προσπέλαση τους. Τα apbi/apbo αποτελούν μέρος του APB bus και έχουν ελεύθερη τη θέση 14, όπου και θα συνδεθεί ο δεύτερος επεξεργαστής(proc1). Προφανώς στην διαδικασία πρέπει να εμπλακεί και ο master του APB για την σωστή επικοινωνία. Τις αλλαγές αυτές θα τις δούμε παρακάτω, Για την σωστή επικοινωνία με τον ελεγκτή κύριας μνήμης ο proc1 πρέπει να συνδεθεί και με το σήμα ahbsi(0), με το οποίο συνδέεται μόνο ο ελεγκτής κύριας μνήμης, όπως ακριβώς και ο proc0. Τέλος τα σήματα iui/iuo θα πρέπει να είναι διαφορετικά από τα iui/iuo του proc0 και γι αυτό ορίζονται ως iui2/iuo2.

Επιπλέον αλλαγές που πρέπει να γίνουν για την ορθή λειτουργία του δεύτερου επεξεργαστή είναι :

- Στο αρχείο *device.vhd* που περιέχει το τρέχον configuration του LEON2, αλλάζουμε σε 3 την σταθερά masters στο πεδίο ahb_config_type
- Στο αρχείο *ambacomp.vhd* που περιέχει όλες τις δηλώσεις των components που θα χρησιμοποιηθούν αλλάζουμε την σταθερά masters στο component ahbarb από 2 σε 3.

- Στο αρχείο *ahbarb.vhd* που αποτελεί τον arbiter/decoder του AHB να αλλαχθεί η σταθερά masters από 2 σε 3, ώστε να μετέχει και ο proc1 στην διαδικασία για δέσμευση του bus.
- Στο αρχείο *apbmst.vhd* που είναι το bridge μεταξύ AHB-APB να γίνει το mapping στην μνήμη των επιπλέον καταχωρητών(CCR και PWD) που προκύπτουν από τον proc1. Για τον σκοπό αυτό καταλαμβάνουμε τις θέσεις 0xD4 και 0xD8 που δεν χρησιμοποιούνται.

Ουσιαστικά με αυτόν τον τρόπο έχουμε επιτύχει να προσθέσουμε μια ακόμη επεξεργαστική μονάδα, η οποία να επικοινωνεί με την υπάρχουσα υποδομή του LEON2. Έχουμε μέχρι στιγμής εξασφαλίσει την ύπαρξη δυο επεξεργαστών που επικοινωνούν με το υπόλοιπο σύστημα και την διατήρηση της συνοχής των κρυφών τους μνημών. Αυτό όμως δεν σημαίνει ότι ο LEON2 έχει μετατραπεί από uniprocessor σε multiprocessor πλήρως λειτουργικό ακόμη. Κάτι τέτοιο απαιτεί περαιτέρω μελέτη σε θέματα τα οποία θα αναφερθούν παρακάτω.

Μια προσθήκη που έγινε ακόμη για το πέρασμα σε δύο πυρήνες είναι η προσθήκη του αναγνωριστικού του επεξεργαστή στο πεδίο του PSR 19-14 που εξ ορισμού είναι 0 για τον ένα επεξεργαστή. Αντί να χρησιμοποιούμε πάντα το ΄΄000000΄΄, τώρα με την εντολή WRPSR, η οποία εκτελείται μόνο σε supervisor mode-bit7=1 του PSR, μπορούμε να γράφουμε και αυτό το πεδίο. Με αυτόν τον τρόπο ο κάθε επεξεργαστής έχει το δικό του αναγνωριστικό. Έχοντας τώρα μοναδικό id για κάθε επεξεργαστή μπορούμε με το κατάλληλο interrupt να ενεργοποιούμε τον άλλο επεξεργαστή.

Ένα θέμα το οποίο πρέπει να μελετηθεί περαιτέρω για να μπορέσει να ολοκληρωθεί η υποδομή του διπλού πυρήνα είναι αυτό του interrupt controller και του debug support unit, όπως και της διαδικασίας booting του ενός επεξεργαστή ώστε να επεκταθεί για τους δύο.

Ένα από τα θέματα, το οποίο πρέπει να μελετήσουμε για την πλήρη υποστήριξη τους συστήματος δύο επεξεργαστών είναι αυτό του συγχρονισμού. Το σύνολο εντολών που υλοποιεί η αρχιτεκτονική του LEON2 είναι το sparc όπως έχουμε δει. Το sparc παρέχει δυο ατομικές εντολές, την swap και την ldstub[9]. Εμείς θα ασχοληθούμε με την swap που είναι πιο διαδεδομένη. Η swap (*swap* [*address*], *reg*$_{rd}$), ανταλλάσσει τον καταχωρητή *reg* με τα περιεχόμενα της διεύθυνσης μνήμης *[address]*. Η λειτουργία είναι ατομική, δεν μεσολαβούν

interrupts ή traps. Σε πολυεπεξεργαστές, δύο η περισσότεροι επεξεργαστές που εκτελούν swap για την ίδια λέξη είναι εγγυημένο ότι κρατούν το serial order. Επίσης υπάρχει και η εντολή *stbar*, ένα είδος barrier, που διασφαλίζει ότι προηγούμενα stores έχουν ολοκληρωθεί πριν εκτελεστούν τα επόμενα.

Με βάση την swap μπορούμε να χτίσουμε μηχανισμούς συγχρονισμού[4]. Μπορούμε για παράδειγμα να υλοποιήσουμε ένα μηχανισμό spin lock. Ο παρακάτω κώδικας υλοποιεί ένα spin lock

LOCK

```
Acquire:
        mov     1, %L0
        swap    [lock],%L0
        brz     %L0, Lock_success
Spin:
        ld      [lock],%L0
        tst     %L0
        be      Spin
        ba      Acquire
Lock_success:
        Critical_Section
```

UNLOCK

```
        stbar
        st      %G0,[lock]   #O G0 περιέχει πάντα την τιμή 0
```

Στον κώδικα παραπάνω στο μέρος acquire ο επεξεργαστής προσπαθεί να πάρει το lock από μια διεύθυνση με ατομική ανταλλαγή (swap) με την σταθερά 1. Αν επιστραφεί 0, τότε είναι ελεύθερο και παίρνει το lock για να μπει στο κρίσιμο τμήμα. Αν είναι διάφορο του 0 τότε το lock το έχει πάρει κάποιος άλλος επεξεργαστής και συνεχίζουμε στο τμήμα του spinning όπου εξετάζεται εάν έχει ελευθερωθεί το lock. Μόλις ελευθερωθεί ξαναμεταβαίνουμε στο acquire στάδιο και ξεκινάμε την διαδικασία να πάρουμε το Lock από την αρχή. Φυσικά ο επεξεργαστής που έχει πάρει το lock θα πρέπει και να το ελευθερώσει. Στο unlock τμήμα το stbar υπάρχει για να διασφαλίζει ότι έχουν ολοκληρωθεί όλα τα stores του χρήστη που εκκρεμούσαν πριν εκτελεστεί το store που ελευθερώνει το lock.

Για να παρέχουμε τον συγχρονισμό αυτού του είδους θα προσθέσουμε αυτή την λειτουργικότητα στο υλικό. Θα χρησιμοποιήσουμε κάποιους καταχωρητές που θα κρατάνε την τιμή του lock, δηλαδή 0 ή 1. Για την προσθήκη τους θα χρησιμοποιήσουμε τις κενές θέσεις στο *apbmst.vhd*. Έτσι οι θέσεις *0xDC* και *0xE0* χρησιμοποιούνται για το mapping στην μνήμη των δύο locks. Φυσικά χρήση των lock registers μπορεί να γίνει μόνο μέσω τη εντολής swap, διαφορετικά αφού δεν θα υπάρχει ατομικότητα δεν θα είναι σίγουρα τα αποτελέσματα του κλειδώματος. Επίσης προφανώς μπορεί κάθε bit από τους 32-bit καταχωρητές να χρησιμοποιηθεί ως ένα κλείδωμα.

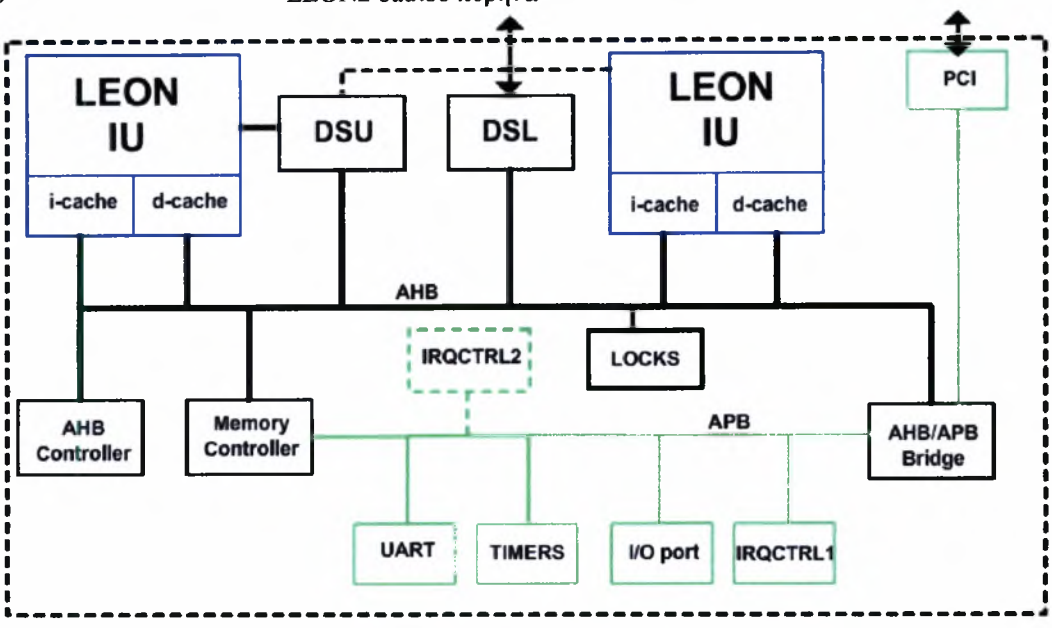
5.3. Συμπεράσματα - Μελλοντική δουλειά

Σε αυτό το σημείο ας κάνουμε μια ανακεφαλαίωση του τι έχει γίνει μέχρι τώρα και πώς μπορούμε να κινηθούμε μελλοντικά.

Αφού ολοκληρώθηκε η μελέτη πολυεπεξεργαστικών συστημάτων και των πρωτοκόλλων συνοχής που υπάρχουν για την υποστήριξη τους, ασχοληθήκαμε με τα βαθύτερα θέματα υλοποίησης και τις σχεδιαστικές αποφάσεις που προκύπτουν κατά την μετάβαση από την θεωρητική προσέγγιση στην υλοποίηση.

Στην προσπάθεια να υλοποιήσουμε τον μικροεπεξεργαστή LEON2(σχήμα18) σε σύστημα διπλού πυρήνα επαληθεύσαμε την δυνατότητα επέκτασης του πρωτοκόλλου συνοχής συσκευών DMA για τις write-through κρυφές μνήμες του μικροεπεξεργαστή. Προσθέσαμε επιτυχώς τον δεύτερο πυρήνα και υλοποιήσαμε ενδεικτικούς μηχανισμούς συγχρονισμού βασισμένους στα δομικά στοιχεία του επεξεργαστή για την διατήρηση της συνέπειας. Για την ολοκλήρωση της υλοποίησης και την λειτουργία του ως πολυεπεξεργαστικό σύστημα πρέπει να γίνει επιπλέον μελέτη για την επέκταση του ελεγκτή διακοπών για την υποστήριξη δύο πυρήνων. Επίσης η μελέτη και επέκταση της υπομονάδας αποσφαλμάτωσης για την υποστήριξη αποσφαλμάτωσης και στον δεύτερο πυρήνα. Ακόμη ένα σημαντικό σημείο είναι να γίνει η μελέτη ενεργοποίησης του δεύτερου πυρήνα από τον πρώτο.

σχήμα 18 LEON2 διπλού πυρήνα



Μετά την ολοκλήρωση των παραπάνω θεμάτων ουσιαστικά θα έχει προκύψει ένα λειτουργικό σύστημα διπλού πυρήνα, οπότε και θα μπορεί να γίνει η μελέτη συγγραφής παράλληλου κώδικα για την δοκιμή του, αφού θα έχει υλοποιηθεί σε FPGA. Φυσικά αργότερα μπορεί να χρησιμοποιηθεί για την μέτρηση απόδοσης προγραμμάτων παράλληλων και μη παράλληλων.

Περαιτέρω ενασχόληση μπορεί να είναι η μετατροπή της write-through κρυφής μνήμης σε write-back και φυσικά η υλοποίηση κάποιου πρωτοκόλλο συνοχής, όπως είναι το MESI. Αυτό που θα πρέπει να προσεχθεί ιδιαίτερα είναι ότι θα πρέπει να δημιουργηθεί ένα FSM που θα λειτουργεί ανεξάρτητα από το FSM της κρυφής μνήμης, που θα ελέγχει καθε συναλλαγή από το bus ή τον επεξεργαστή και ότι τα state bits που χρειαζόμαστε για την κωδικοποίηση των καταστάσεων του FSM θα πάρουν την θέση των valid bits στα μπλοκ κρυφής μνήμης.

Επιπλέον μπορεί να γίνει η ενεργοποίησης της μονάδας MMU για την χρήση εικονικών διευθύνσεων και η μελέτη προβλήματων συνοχής σε επίπεδο εικονικών διευθύνσεων(στο TLB).

Τέλος μπορεί να γίνει η συγγραφή διαφόρων συνεπεξεργαστών για την εξυπηρέτηση ειδικών λειτουργιών, π.χ υλοποίηση critical paths κομματιών αλγορίθμων και μέσα από αυτό μελέτη του παραλληλισμού σε επίπεδο thread και επικοινωνία όχι μέσω κοινής μνήμης με τον επεξεργαστή αλλά ειδικών καταχωρητών.

6. Διάταξη FPGA

6.1. Γενικά

Στο κεφάλαιο αυτό θα ασχοληθούμε με την ροή πληροφορίας για την σχεδίαση συστημάτων που υλοποιούνται σε επαναπρογραμματιζόμενες διατάξεις FPGA, δηλαδή τα στάδια από τα οποία πρέπει να περάσει η σχεδίαση ενός κυκλώματος μετά την λειτουργική του προσομοίωση για την σύνθεση του και την υλοποίηση σε FPGA. Επίσης περιγράφεται το board που χρησιμοποιήσαμε και τα χαρακτηριστικά του, αλλά και ειδικότερα η διαδικασία για την υλοποίηση συγκεκριμένα του επεξεργαστή LEON2 και τα εργαλεία που χρησιμοποιήθηκαν για να καταστεί κάτι τέτοιο δυνατό.

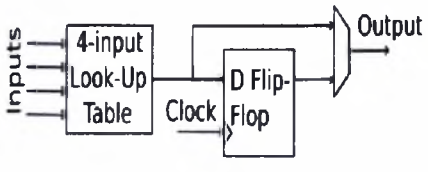
6.2. Τι είναι η FPGA

FPGA(Field Programmable Gate Array) είναι μια επαναπρογραμματιζόμενη διάταξη. Δηλαδή μια συσκευή ημιαγωγών που περιέχει προγραμματιζόμενα κομμάτια λογικής που ονομάζονται ''logic blocks'' και προγραμματιζόμενες διασυνδέσεις μεταξύ τους[19].

Τα logic blocks είναι ουσιαστικά ένα μικρό lookup table (**LUT**), ένα Dflipflop και ένα 2-to-1 mux (*σχήμα19*).

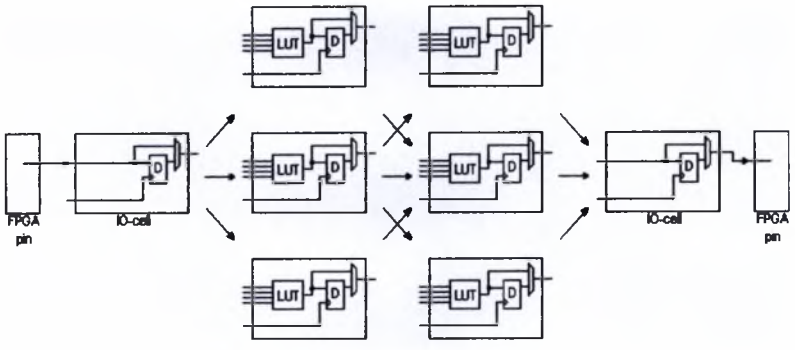
σχήμα 19

Logic Block



Οι FPGA αποτελούνται από συνήθως μερικές εκατοντάδες χιλιάδες LUT 's μέχρι μερικά εκατομμύρια. Το LUT είναι σαν μια μικρή RAM και συνήθως έχει 4 εισόδους και άρα μπορεί να υλοποιήσει οποιαδήποτε λογική πύλη μέχρι τέσσερις εισόδους. Για παράδειγμα μια πύλη and 3 εισόδων, τις οποίας η έξοδος περνάει από μια OR πύλη μαζί με μία ακόμη είσοδο μπορεί να ''χωρέσει'' σε ένα LUT. Τα LUT 's μπορούν να συνδεθούν μεταξύ τους με καλώδια και πολυπλέκτες. Τα καλώδια χρησιμοποιούνται επίσης και για να συνδέσουν το δίκτυο από το LUT 's με τα μπλοκ που χρησιμοποιούνται για I/O και τα pins του FPGA (*σχήμα 20*).

σχήμα 20 **Δίκτυο LUT 's**



Τα pins που αναφέραμε πιο πάνω είναι λογικά μπλοκ τα οποία έχουν σχεδιαστεί για να επιτελούν μια συγκεκριμένη δουλειά είτε αυτή είναι να μεταφέρουν ρεύμα στα LUT 'σε είτε να επικοινωνούν με τον έξω κόσμο μέσω I/O. Τα pins αυτά χωρίζονται σε δύο κατηγορίες τα αφοσιωμένα(*dedicated*) και αυτά που ορίζονται από τον χρήστη(*user*). Τα dedicated pins είναι hard-coded για κάποιες συγκεκριμένες συναρτήσεις όπως pins ρεύματος, pins γείωσης, configuration pins για το κατέβασμα του κυκλώματος. Τα user pins χρησιμοποιούνται για I/O και εξαρτάται από τον χρήστη αν θα ορίσει κάποιο pin ως είσοδο, έξοδο ή και διπλής κατεύθυνσης.

Τέλος μια FPGA είναι συνήθως ΄΄σύγχρονη΄΄. Αυτό σημαίνει ότι η σχεδίαση βασίζεται στο ρολόι και κάθε D-flipflop παίρνει μια καινούρια κατάσταση στην ακμή του ρολογιού. Ένα ρολόι μπορεί να οδηγεί πολλά D-flipflop ταυτόχρονα που απέχουν αρκετά μεταξύ τους. Αυτό μπορεί να δημιουργήσει προβλήματα χρονισμού καθώς το ρολόι όσο γρήγορα και να διαδίδεται από τα καλώδια στο επίπεδο του χρόνου που μιλάμε δημιουργούνται διαφορές και έτσι μπορεί αν φτάνει με καθυστέρηση σε κάποια D-flipflop το λεγόμενο clock skew. Γι αυτό και χρησιμοποιούνται ειδικά εσωτερικά καλώδια που επιτρέπουν την διάδοση του ρολογιού με μικρότερο skew. Επιπλέον συνήθως χρησιμοποιούνται πολλά ρολόγια που καλύπτουν μια περιοχή και γι αυτό χρειάζεται μεγάλη προσοχή στην περιγραφή ενός κυκλώματος για να μην δημιουργούνται λάθη χρονισμού, όπως και τεχνικές συγχρονισμού.

Μπορούμε να συνοψίσουμε λέγοντας ότι η FPGA είναι μια διάταξη που μπορεί να συμπεριφερθεί σαν μια συνάρτηση-κύκλωμα που έχει ορίσει ο χρήστης. Τα κυκλώματα αυτά μπορούν να υλοποιηθούν όσες φορές θέλουμε χωρίς περιορισμό, δηλαδή σε περίπτωση λανθασμένης συμπεριφοράς να

ξανασχεδιάζουμε και υλοποιούμε ξανά την λογική συνάρτηση και επίσης η FPGA μπορεί να λειτουργεί ως διαφορετικό κύκλωμα κάθε φορά. Τέλος σημειώνουμε ότι κάθε φορά που αποσυνδέουμε την τροφοδοσία η λειτουργικότητα της χάνεται και πρέπει να ξαναπρογραμματιστεί εκτός αν ο κώδικας είναι αποθηκευμένος σε ειδική μνήμη της FPGA και φορτώνεται από εκεί κάθε φορά που επανασυνδέεται στην τροφοδοσία.

6.3.Ροή πληροφορίας σχεδίασης σε FPGA

Η διαδικασία έχει συνοπτικά ως εξής: Αρχικά έχουμε την σχεδίαση του κυκλώματος σε μορφή VHDL κώδικα. Ακολούθως γίνεται η σύνθεση, το Place and Route και τέλος παράγεται ένα bitstream αρχείο, το οποίο και θα υλοποιήσει το κύκλωμα μας στην FPGA.

Αναλυτικότερα[16], αφού επιλέξουμε το κύκλωμα το οποίο θέλουμε να υλοποιήσουμε σε FPGA χρησιμοποιούμε την γλώσσα περιγραφής κυκλωμάτων VHDL(φυσικά μπορούμε να χρησιμοποιήσουμε και την Verilog) για την περιγραφή του κυκλώματος. Αφού περάσουμε τον συντακτικό έλεγχο, πρέπει να γίνει επιβεβαίωση της ορθής λειτουργίας του κυκλώματος με την χρήση κάποιων testbenches. Με αυτόν τον τρόπο προσομοίωσης, βεβαιωνόμαστε ότι το κύκλωμα λειτουργεί έτσι όπως θεωρητικά θα περιμέναμε ότι θα λειτουργούσε.

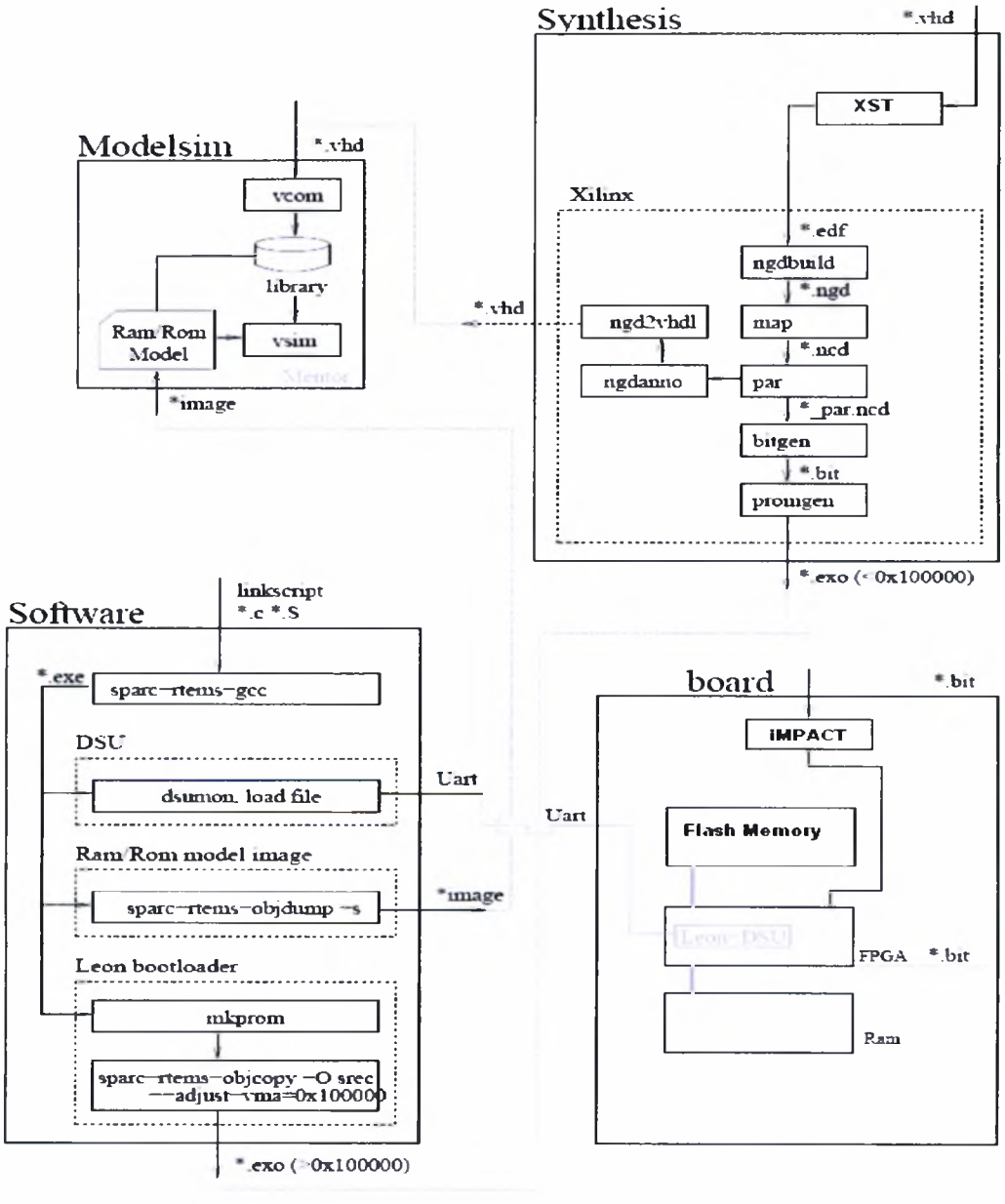
Στην επόμενη φάση θα πρέπει να γίνει η σύνθεση του κυκλώματος. Με την διαδικασία αυτή μετατρέπουμε τα μοντέλα περιγραφής της λειτουργίας του κυκλώματος σε λογικές πύλες. Θα πρέπει να δίνεται προσοχή ώστε ο κώδικας VHDL που γράφεται να μπορεί να είναι συνθέσιμος, για παράδειγμα αναδρομές που είναι επιτρεπτές σε κώδικα υψηλού επιπέδου δεν είναι συνθέσιμες αφού δεν έχουν νόημα σε επίπεδο υλικού. Σε αυτήν την φάση προκύπτουν ανάλογα με το εργαλείο δύο είδη αρχείων, τα NGD netlists και τα EDIF netlists αρχεία τα οποία περιέχουν την σχηματική απεικόνηση του κυκλώματος σε επίπεδο λογικών πυλών.

Ακολουθεί το translation, που είναι η διαδικασία που παίρνει ως είσοδο το αρχείο netlist από την φάση σύνθεσης και το μετατρέπει σε νέο netlist συμβατό με το database του εργαλείου, παίρνοντας υπόψη και ένα αρχείο περιορισμών το User Constraints File, το οποίο ορίζεται από τον χρήστη. Το ucf είναι ένα αρχείο

χωρικών και χρονικών περιορισμών που τίθονται ανάλογα με την FPGA και την σχεδίαση που χρησιμοποιούμε.

Το επόμενο βήμα είναι να γίνει το mapping, κατά το οποίο γίνεται δέσμευση πόρων στην FPGA ανάλογα με τα στοιχεία λογικής που χρησιμοποιήσαμε στην σχεδίαση. Χρησιμοποιείται επίσης και το ucf αρχείο και μπορεί να προστεθεί και επιπλέον λογική για την ικανοποίηση των χρονικών περιορισμών. Επιπλέον γίνονται κάποιες βελτιστοποιήσεις ανάλογα με την τεχνολογία της συσκευής που χρησιμοποιούμε και επίσης κάποιος έλεγχος σύμφωνα με τους κανόνες σχεδίασης για το νέο netlist.

σχήμα 21 **Ροή Σχεδίασης**



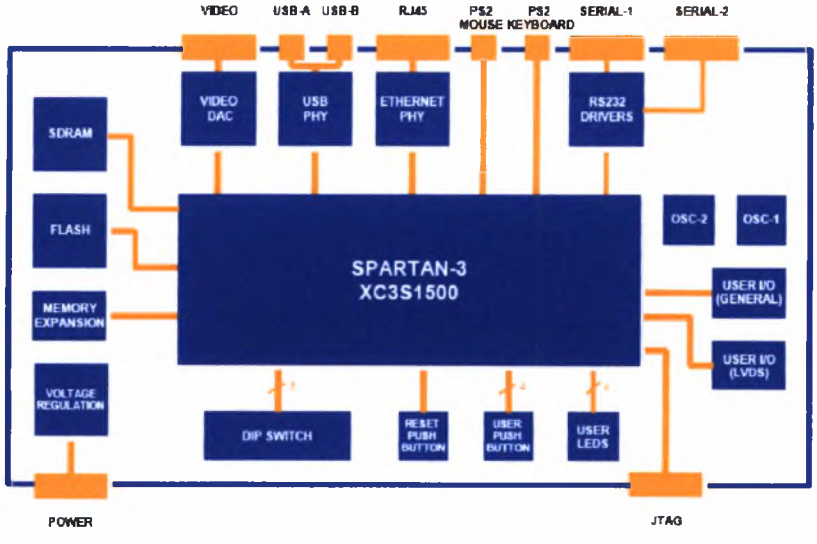
Στη συνέχεια περνάμε στο Place and Route. Σε αυτή την φάση τα στοιχεία του netlist απεικονίζονται στις φυσικές θέσεις της FPGA για να δημιουργηθεί ένα αρχείο το οποίο να μπορεί να υλοποιηθεί από τα LUT 's των FPGA's. Δηλαδή με αυτόν τον τρόπο επιλέγεται η καλύτερη θέση για το κάθε μπλοκ και η βέλτιστη δρομολόγηση με την ελάχιστη καθυστέρηση.

Από την φάση του Place and Route προκύπτει ένα bitstream αρχείο με την βοήθεια του εργαλείου bitgen. Το bitstream είναι ένα αρχείο που δεν περιέχει τίποτε περισσότερο από την αρχική πληροφορία μαζί με την επιμέρους επεξεργασία σε μορφή δυαδικού κώδικα ώστε να μπορέσει να υλοποιηθεί από τα λογικά κελιά της FPGA.

6.4.Πλατφόρμα Υλοποίησης

Αφού λοιπόν έχουμε δει το τι είναι μια FPGA και ποια είναι η ροή πληροφορίας για τον προγραμματισμό της θα περάσουμε να δούμε τα χαρακτηριστικά της FPGA που χρησιμοποιήσαμε για την εργασία αυτή.

σχήμα 22 **Τεχνική περιγραφή GR-XC3S1500**



Το board που επιλέξαμε για να δουλέψουμε είναι το GR-XC3S1500 από την Pender σε συνεργασία με την Gaisler Research που παρέχει τον μικρο-επεξεργαστή LEON2[12]. Είναι μια ευπροσάρμοστη πλατφόρμα για την υλοποίηση κυκλωμάτων FPGA. Ενσωματώνει την FPGA XC3S της οικογενείας των Spartan3 της Xilinx, που αποτελείται από 1,5 εκατομμύρια πύλες.

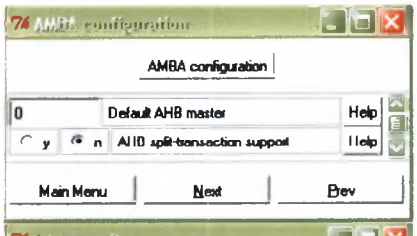
Παρέχονται επίσης ενσωματωμένη μνήμη Flash και μνήμη SDRAM, όπως επίσης και Ethernet, JTAG καλώδιο, σειριακή θύρα USB και PS2 διεπαφές για πληκτρολόγιο και ποντίκι.

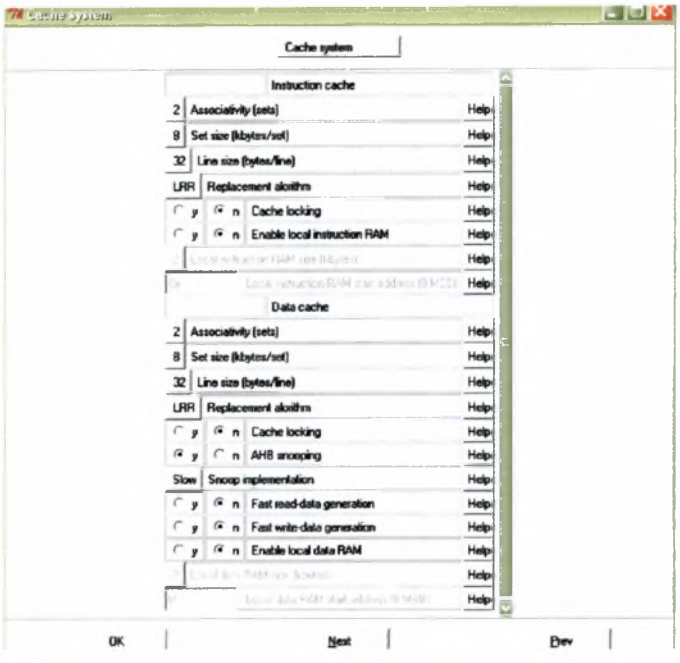
Η ύπαρξη μνήμης Flash Prom παρέχει την δυνατότητα αποθήκευσης FPGA configurations για να μην χρειάζεται η φόρτωση του bitfile κάθε φορά που επανασυνδέουμε στο ρεύμα το board.

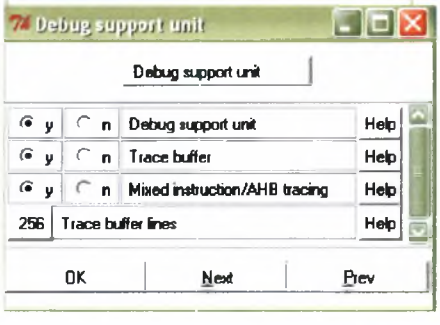
Οι ενσωματωμένες μνήμες μαζί με τις διεπαφές επικοινωνίας με τον χρήστη καθιστούν το board εύχρηστο για γρήγορη υλοποίηση και ανάπτυξη λογισμικού για τον LEON.

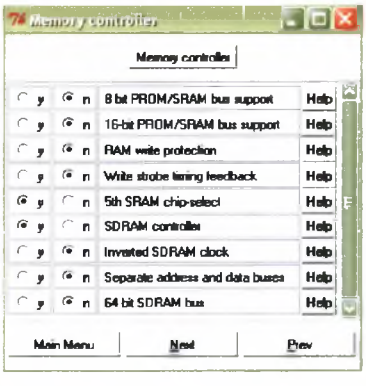
Παράρτημα Α

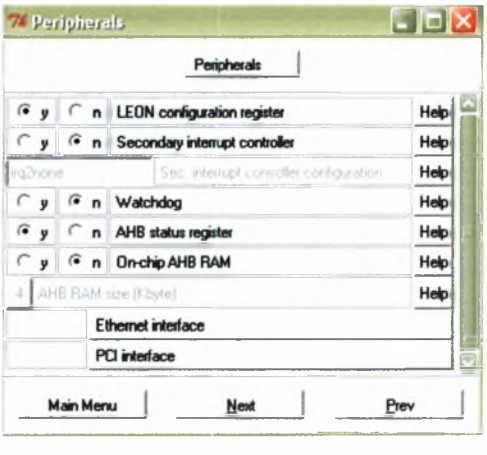
σχήμα23 **LEON2 configuration**

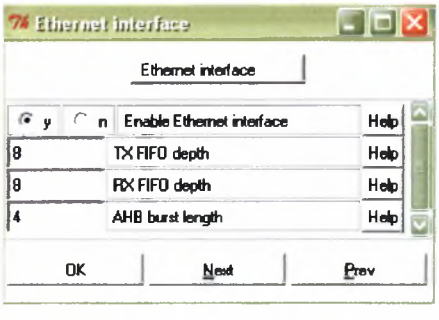












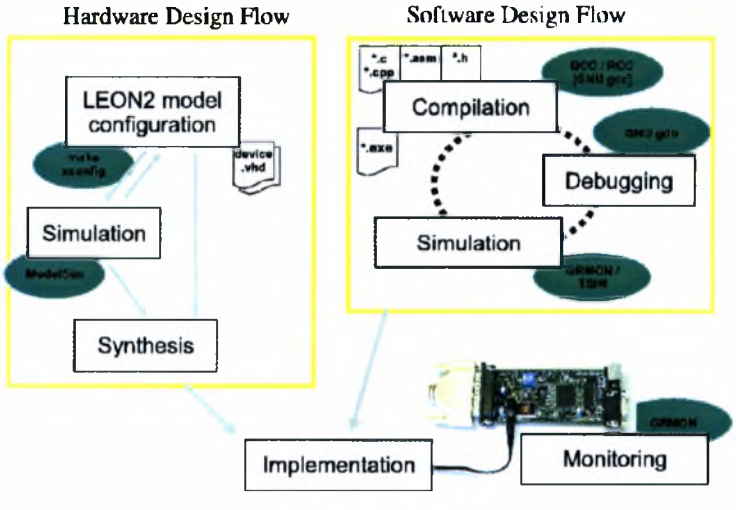
Παράρτημα Β. Υλοποίηση σε FPGA

Β.1. Υλοποίηση του LEON2 σε FPGA

Στην παράγραφο αυτή θα ασχοληθούμε με την ροή σχεδίασης που ακολουθήθηκε ειδικά για την εργασία αυτή, για την υλοποίηση του μικρο-επεξεργαστή LEON2 στην FPGA GR-XC3S1500. Σε κάθε βήμα που θα αναλύεται θα παρουσιάζονται και τα εργαλεία που χρησιμοποιήθηκαν αλλά και ο τρόπος για να γίνει πιο συγκεκριμένη η διαδικασία που περιγράφηκε στην υποπαράγραφο 4.2.

Η μεθοδολογία[6] που ακολουθείται για τον σχεδιασμό ενός System-On-Chip μπορεί να αναλυθεί σε δύο υπό-ροές σχεδίασης, αυτή της σχεδίασης του υλικού και αυτή της σχεδίασης του λογισμικού όπως φαίνεται στο σχήμα *21.* Οι δύο ροές αν και είναι στενά συνδεδεμένες μπορούν να καλυφθούν παράλληλα, αφού υπάρχουν ελάχιστες εξαρτήσεις ανάμεσα τους. Στο τέλος μπορούμε να πάρουμε το μοντέλο υλικού και την εφαρμογή λογισμικού και αφού τα κατεβάσουμε και τα δύο στην FPGA μας να κάνουμε το έλεγχο ορθής λειτουργίας μέσω κάποιων πιθανών μηνυμάτων.

σχήμα24 **Ροές σχεδίασης για τονLEON2**



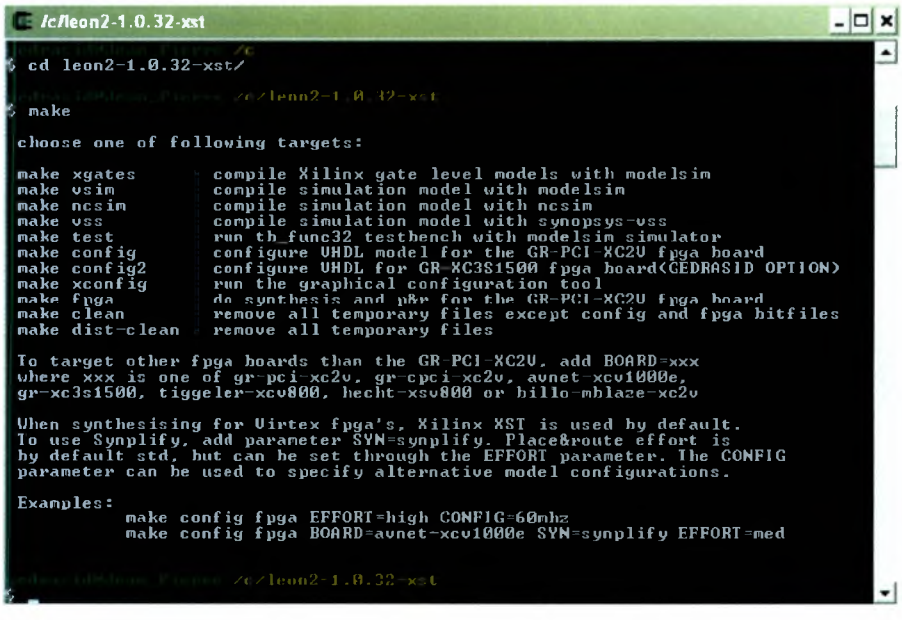
Στην περίπτωση μας η εφαρμογή θα είναι ένα απλό πρόγραμμα hello-world το οποίο θα επιβεβαιώνει την ύπαρξη του LEON2 στην FPGA και την σωστή

λειτουργία του. Στην διαδικασία αυτή φυσικά εμπλέκεται και ένας αριθμός από εργαλεία.

Β.1.1. Hardware ροή

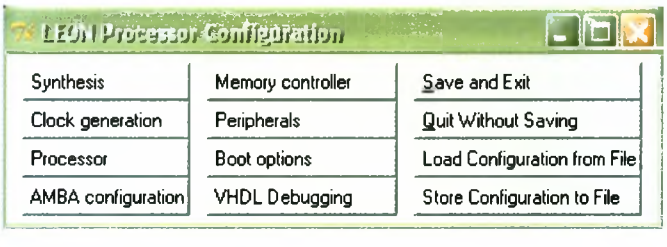
Για την hardware ροή θα χρειαστούμε καταρχήν την διαμόρφωση του LEON2 σύμφωνα με τις ανάγκες μας. Αυτό μπορεί να γίνει μέσω του γραφικού περιβάλλοντος που παρέχεται με τον LEON2 και υλοποιείται με TCL/TK scripts για Linux και Windows με το πρόγραμμα cygwin. Στην εργασία αυτή χρησιμοποιήθηκε το cygwin. Γενικά με τον επεξεργαστή παρέχονται αρκετά scripts για να διευκολύνουν την υλοποίηση του μοντέλου. Έτσι μπορούμε χρησιμοποιώντας την εντολή make να αποκτήσουμε μια σειρά από επιλογές που θα χρησιμεύσουν στα διαφορετικά βήματα της ροής.

σχήμα25 **Οθόνη επιλογών Make**



Μπορούμε να επιλέξουμε είτε configuration μέσω του γραφικού περιβάλλοντος, είτε μέσω της εντολής make config. Για το γραφικό πληκτρολογούμε make xconfig και παίρνουμε την επόμενη οθόνη (σχήμα26).

σχήμα26 **Γραφικό περιβάλλον configuration**

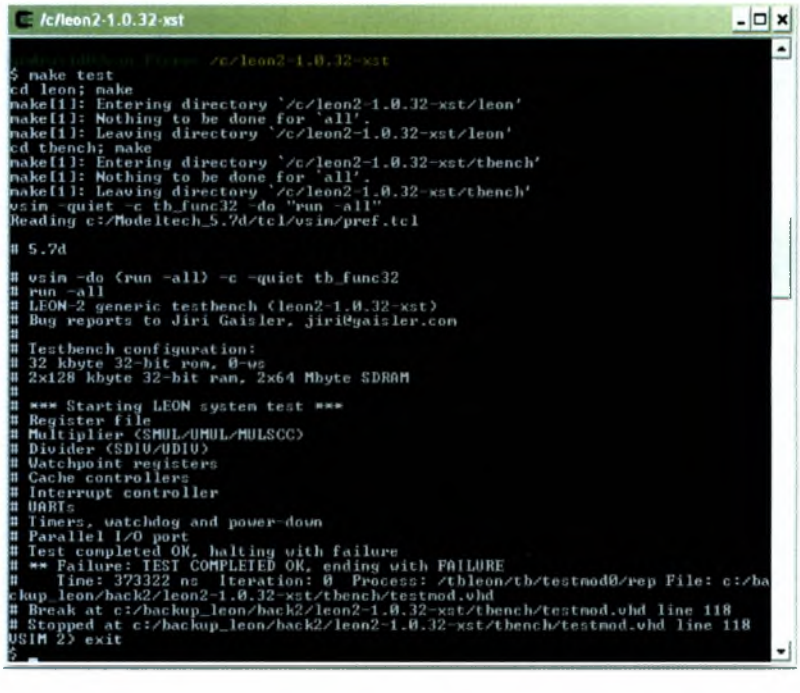


Από εδώ μπορούμε να επιλέξουμε το configuration σε κάθε μια από τις κατηγορίες που υπάρχουν. Οι επιλογές παρατίθενται στο παράρτημα Α. Φυσικά μετά την ολοκλήρωση των επιλογών μπορούμε μέσω της επιλογής ΄΄Save and Exit΄΄ να αποθηκεύσουμε τις επιλογές. Ακολούθως με την εντολή ΄΄make dep΄΄ αποθηκεύουμε τις επιλογές και στο αρχείο vhd (device.vhd) που χειρίζεται το μοντέλο του LEON2.

Ένας εναλλακτικός τρόπος για το configuration είναι να χρησιμοποιήσουμε τα scripts με την εντολή ΄΄make config BOARD=gr-xc3s1500΄΄. Με αυτόν τον τρόπο χρησιμοποιείται το αρχείο config.vhd που υπάρχει στον κατάλογο boards/gr-xc3s1500/ για την προτεινόμενη για το συγκεκριμένο board διαμόρφωση, και ακολούθως εκτελούμε την εντολή ΄΄make dep΄΄.

Αφού έχουμε διαμορφώσει το μοντέλο του επεξεργαστή μπορούμε να το προσομοιώσουμε χρησιμοποιώντας κάποια testbenches, τα οποία παρέχονται. Τα testbenches αυτά τεστάρουν τα βασικά δομικά κομμάτια του επεξεργαστή για την σωστή τους λειτουργία. Η εντολή ΄΄make test΄΄ είναι αυτή που ξεκινάει την προσομοίωση και τον λειτουργικό έλεγχο του συστήματος χρησιμοποιώντας τα testbenches από τον κατάλογο /tbench και κώδικες από το /tsource που τεστάρουν το σύστημα, σε συνεργασία με το εργαλείο Modelsim. Τα αποτελέσματα του testbench φαίνονται στο σχήμα 27.

σχήμα27

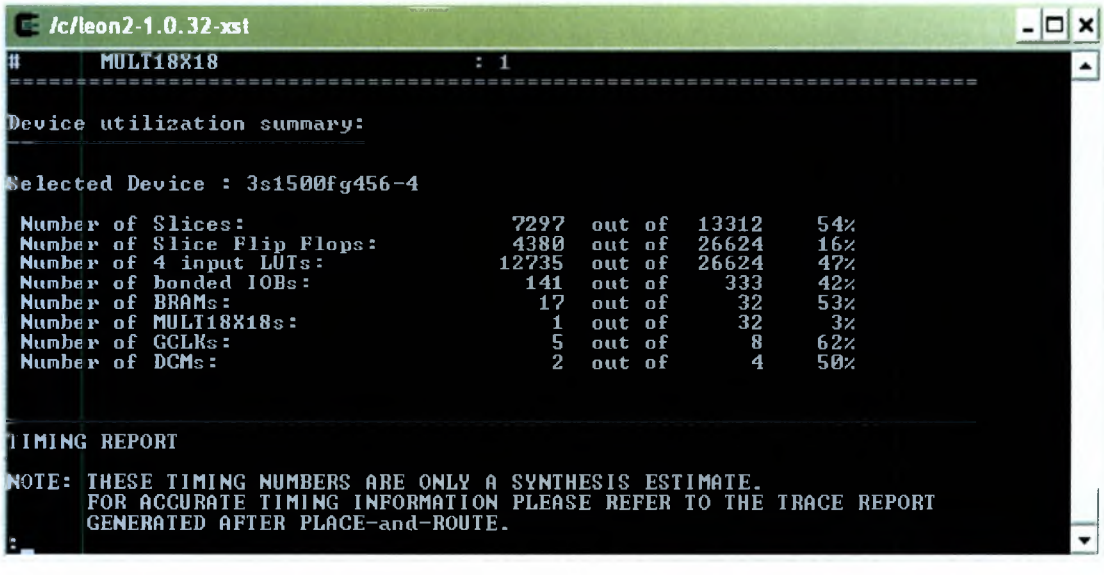


Το testbench ολοκληρώνεται χωρίς λάθη όταν εμφανίζεται το μήνυμα: **TEST completed OK, halting with FAILURE

(υπονοώντας ότι για να σταματήσει ο επεξεργαστής μετά την επιτυχή ολοκλήρωση του testbench πρέπει να προκληθεί σε αυτόν μια τεχνητή αποτυχία!)

Το επόμενο βήμα είναι να κάνουμε την σύνθεση του μοντέλου του επεξεργαστή και να φτιάξουμε το netlist για την συγκεκριμένη FPGA που δουλεύουμε. Επειδή ο LEON2 έχει σχεδιαστεί με επίκεντρο την δυνατότητα σύνθεσης του δουλεύει με τα περισσότερα εργαλεία για σύνθεση όπως το Xilinx ISE XST, Synplicity, Design Compiler. Εμείς θα ασχοληθούμε με το XST[15][17]. Για να γίνει αυτό έχουμε δύο επιλογές και πάλι, είτε να χρησιμοποιήσουμε το γραφικό περιβάλλον του Xilinx, είτε να χρησιμοποιήσουμε το script του LEON μέσω του cygwin. Θα ξεκινήσουμε από την δεύτερη επιλογή. Μέσω των scripts που παρέχονται υπάρχει η δυνατότητα χρήσης του εργαλείου XST με την εντολή ΄΄make fpga BOARD=gr-xc3s1500΄΄. Σε περίπτωση που δεν ενεργοποιηθεί η επιλογή BOARD η διαδικασία δεν θα εκτελεστεί για το σωστό board. Μόλις ολοκληρωθεί και το place&route παράγεται το αρχείο bitstream με τις απαραίτητες πληροφορίες για τον προγραμματισμό της FPGA με το hardware μοντέλο του LEON2. Τα αποτελέσματα από την ροή σχεδίασης φαίνονται παρακάτω(σχήμα28).

σχήμα28 **Αποτελέσματα ροής σχεδίασης**



Βέβαια, υπάρχει πάντα η επιλογή να μη χρησιμοποιούμε τα scripts αλλά το εργαλείο XST απευθείας. Έτσι μπορούμε είτε να τρέξουμε το έτοιμο project από τον υποκατάλογο /boards/gr-xc3s1500/ για το συγκεκριμένο board, που είναι το

αρχείο leon-ise6.npl, είτε να δημιουργήσουμε το δικό μας project file από την αρχή.

Β.1.2. Software ροή

Παράλληλα με την hardware ροή μπορούμε να αναπτύξουμε και το software. Το πρόγραμμα που θα προσπαθήσουμε να τρέξουμε στον επεξεργαστή είναι ένα απλό πρόγραμμα του τύπου ''hello world'', απλώς για να επιβεβαιώσουμε ότι ο επεξεργαστής που κατέβηκε στην FPGA λειτουργεί κανονικά. Για να γίνει κάτι τέτοιο θα πρέπει αφού γραφεί το πρόγραμμα να γίνει compile με το κατάλληλο εργαλείο, debug εάν χρειαστεί και υπάρχει η δυνατότητα προσομοίωσης σε ένα simulator του LEON2 πριν επιχειρήσουμε την εκτέλεση του από τον πραγματικό επεξεργαστή. Η διαδικασία σε βήματα είναι η ακόλουθη:

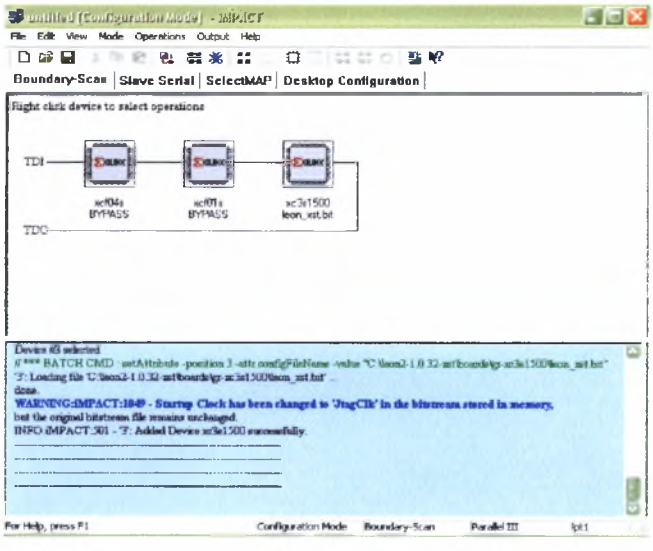
1. Compile & Link με το gcc
2. Debug με το gdb σε επίπεδο simulator
3. Debug με το gdb συνδεδεμένο στην FPGA μεσω του Grmon

Για το software υπάρχουν δύο C/C++ cross-compilers. Για την εργασία αυτή προτιμήθηκε ο RTEMS[13], ο οποίος είναι κατάλληλος για την αρχιτεκτονική του LEON. Περιέχει μεταξύ άλλων gcc compiler, gdb debbuger, linker, grmon για την επικοινωνία με τον LEON. Το RTEMS δεν είναι μόνο ένας cross-compiler, αλλά ένα λειτουργικό σύστημα με multi-tasking πυρήνα, που παρέχει και υποστήριξη για hardware υπομονάδα, το οποίο όμως δεν τρέχει πάνω από τον επεξεργαστή όπως για παράδειγμα το Linux πάνω στο οποίο μπορούν μετά να φορτωθούν και να εκτελεσθούν άλλα προγράμματα. Αντίθετα, με το RTEMS τα προγράμματα του χρήστη μεταφράζονται και ακολούθως συνδέονται με το λειτουργικό σύστημα (σαν να ήταν βιβλιοθήκη), καταλήγοντας έτσι σε νέο μεγάλο πρόγραμμα-εκτελέσιμο που περιέχει και το λειτουργικό σύστημα και το πρόγραμμα του χρήστη. Αφού έχουμε εγκαταστήσει το RTEMS θα περάσουμε στο compile&link του προγράμματος hello-world. Το compile γίνεται με την εντολή *sparc-rtems-gcc.exe –mv8 –msoft-float –O2 –qleon2 rtems-hello.c –o hello.* Όπου *–mv8* είναι flag για την παραγωγή SPARCV8 Mul/Div εντολών, *–msoft-float* σε περίπτωση που δεν έχουμε FP unit, *–O2* βελτιστοποιήσεις επιπέδου 2, *–qleon2* εκτελέσιμο για τον LEON2.

Β.1.3. Συνδυάζοντας τις ροές

Αφού γίνει και το debugging και πάρουμε ένα σωστό εκτελέσιμο και έχοντας το hardware μοντέλο του LEON μπορούμε να προχωρήσουμε με την υλοποίηση του συστήματος στην Spartan3 FPGA. Από το εργαλείο XST πήραμε ένα τελικό bitfile για τον προγραμματισμό της FPGA. Το .bit αρχείο αυτό θα κατέβει με την βοήθεια ενός άλλου εργαλείου της Xilinx, του iMPACT[15] και την χρήση ενός JTAG καλωδίου που συνδέεται στην παράλληλη θύρα του host H/Y που έχει το .bit και στην FPGA. Μόλις ξεκινήσει το iMPACT ανοίγει ένα πλαίσιο διαλόγου. Ξεκινάμε με την επιλογή *configure devices* για να γίνει η διαδικασία εντοπισμού του FPGA board που θα χρησιμοποιήσουμε. Ακολουθεί η επιλογή *Boundary Scan Mode* και *Automatically connect to cable and identify Boundary-Scan chain* για να εντοπίσει το εργαλείο το Board μέσω του καλωδίου JTAG. Εάν ο εντοπισμός είναι επιτυχής παίρνουμε την ακόλουθη οθόνη(σχήμα 29).

σχήμα29 **iMPACT**



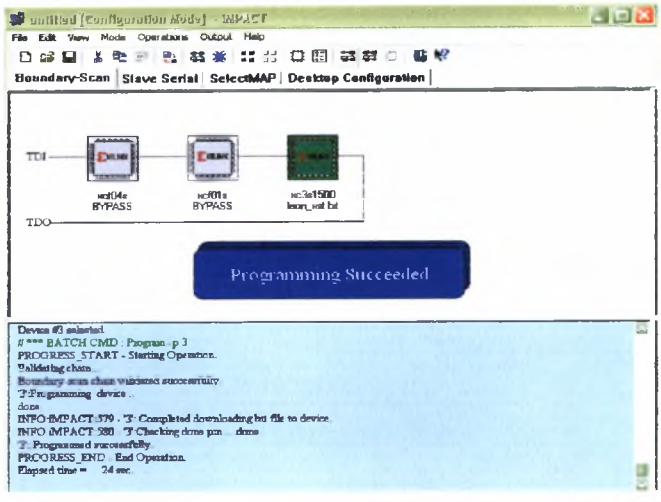
Οι συσκευές που εντοπίζονται είναι οι ακόλουθες, όπως φαίνεται και στην εικόνα 25:

1. XCF04S 4Mbit Platform Flash Prom
2. XCF01S 1Mbit Platform Flash Prom
3. XC3S1500 Spartan-3 XC3S1500 FPGA device

Οι πρώτες δύο συσκευές είναι Flash μνήμες στις οποίες μπορούμε να αποθηκεύσουμε .mcs αρχεία που έχουν αποθηκευμένο το configuration της FPGA ώστε να μην πρέπει να το φορτώνουμε από την αρχή κάθε φορά που αποσυνδέουμε το board από το ρεύμα. Η τρίτη συσκευή που εντοπίζεται είναι η FPGA στην οποία με διπλό-κλικ αναθέτουμε το .bit configuration αρχείο και

ακολούθως με δεξί-κλικ και επιλογή του ΄΄Program΄΄ προγραμματίζουμε την FPGA. Μόλις ολοκληρωθεί η διαδικασία ανάβει το ΄΄DONE΄΄ led στο board και εμφανίζεται η ακόλουθη οθόνη(σχήμα30).

Σχήμα30 **iMPACT**

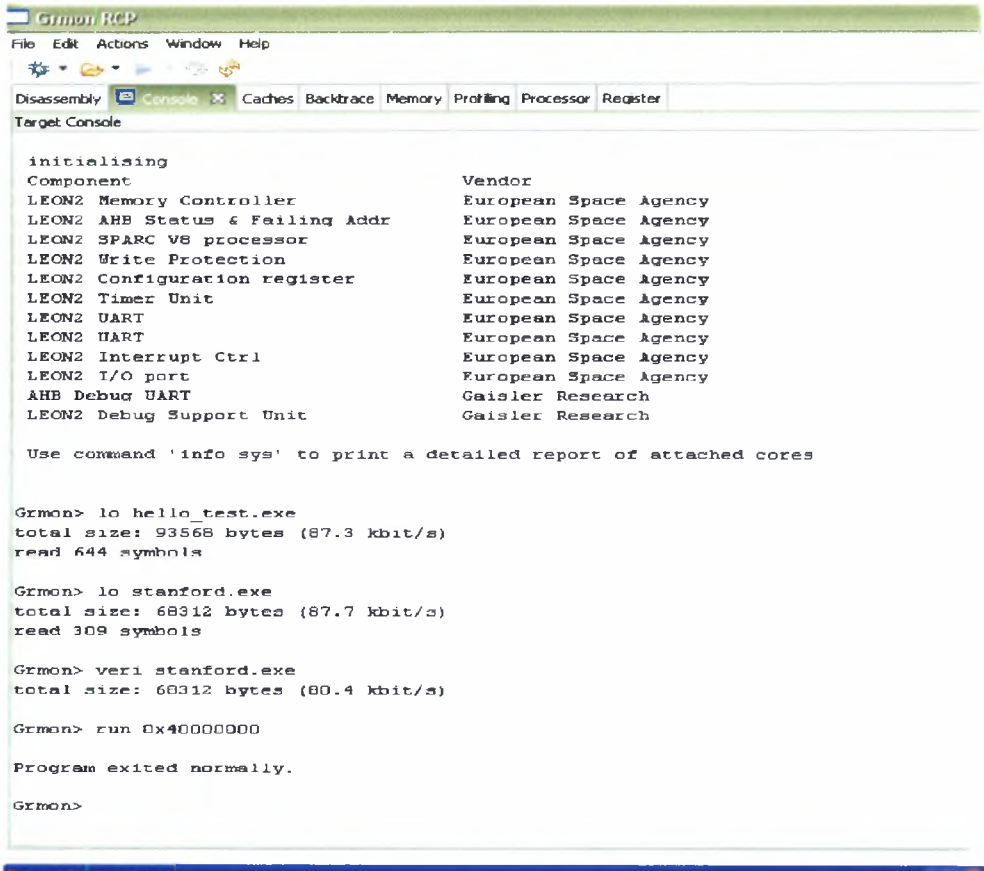


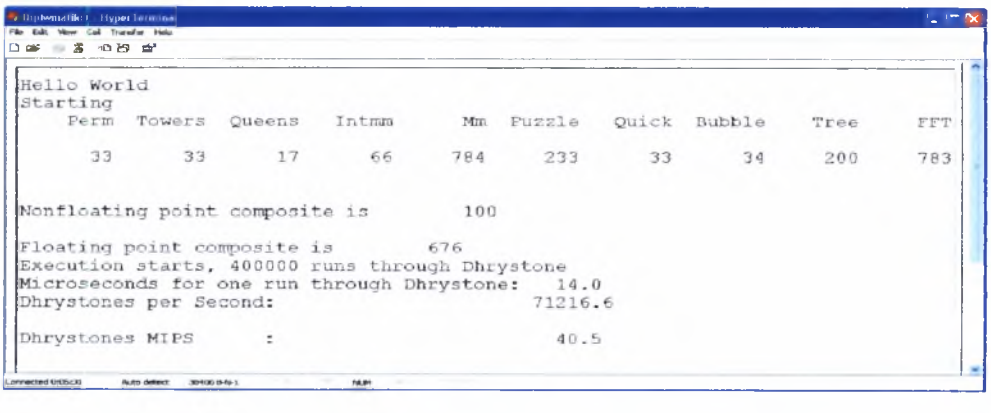
Έτσι τώρα έχουμε τον επεξεργαστή LEON2 κατεβασμένο στην FPGA και θα πρέπει να προχωρήσουμε κατεβάζοντας το hello-world στον LEON και να επικοινωνήσουμε με την βοήθεια του εργαλείου Grmon[14]. Το Grmon επιτρέπει παρακολούθηση του επεξεργαστή και debugging, παρέχοντας πρόσβαση στους εσωτερικούς καταχωρητές, τις μνήμες και τα βασικά περιφερειακά. Για να δουλέψει προϋποθέτει την ύπαρξη μονάδας DSU από το configuration του LEON, αφού μέσω αυτού και με την βοήθεια σειριακού καλωδίου γίνεται η σύνδεση με το host-pc.

Για τις ανάγκες της εργασίας χρησιμοποιήθηκε το GrmonRPC το οποίο είναι γραφικό interface του Grmon, αλλά παρέχει και γραμμή εντολών. Αφού κάνουμε ΄΄initialize target΄΄, με το οποίο εντοπίζεται ο επεξεργαστής που είναι προγραμματισμένος στην FPGA (σχήμα31), με την εντολή ΄΄info sys΄΄ παίρνουμε μια λεπτομερή περιγραφή του συστήματος που έχει φορτωθεί μαζί με τις διευθύνσεις μνήμης που καταλαμβάνει το κάθε δομικό στοιχείο. Για να φορτώσουμε το hello-world τοποθετούμε το εκτελέσιμο που προέκυψε από το compile στον φάκελο του GrmonRPC και εκτελούμε την εντολή ΄΄lo hello.exe΄΄(ή load [filename].exe) και έτσι φορτώνουμε το πρόγραμμα στην FPGA. Μπορούμε επίσης να επιβεβαιώσουμε με την εντολή ΄΄veri hello.exe΄΄ ότι το εκτελέσιμο έχει όντως φορτωθεί στην κύρια μνήμη του LEON2. Τέλος εκτελούμε την εντολή run και το πρόγραμμα εκτελείται και εμφανίζεται το μήνυμα ΄΄Program exited normally΄΄. Αυτό στο οποίο πρέπει να δοθεί ιδιαίτερη

προσοχή είναι το γεγονός ότι το GrmonRCP συνδέεται με την πρώτη σειριακή θύρα του board για να επικοινωνήσει με τον LEON2 και μέσω αυτού φορτώνει το πρόγραμμα στην κύρια μνήμη. Όμως τα δεδομένα που στέλνει για έξοδο δεν επιστρέφουν στην ίδια θύρα, αλλά προωθούνται στην δεύτερη σειριακή θύρα του board. Έτσι για να πάρουμε το output των προγραμμάτων που τρέξαμε το board συνδέθηκε μεσω σειριακής θύρα με ένα δεύτερο H/Y[10] και χρησιμοποιώντας το πρόγραμμα hyper-terminal των windows εμφανίσαμε τα outputs των προγραμμάτων στην οθόνη. Επίσης χρησιμοποιήθηκαν με επιτυχία και άλλα προγράμματα, όπως το stanford και το dhrystone benchmark.

σχήμα31 **GrmonRCP-Stanford benchmark output**





Βιβλιογραφία

[1] John L. Hennessy, David A. Patterson: "*Computer Architecture-A quantative approach-4th Edition*", Ed. Morgan Kaufmann 2007.

[2] David E. Culler, Jaswinder Pal Singh, Anoop Gupta: "*Parallel Computer Architecture-A hardware.software Approach*", Ed. Morgan Kaufmann 2003.

[3] Tartalja, I. And Milutinovi, V., "*The Cache Coherence Problem in Shared-Memory Multiprocessors : Software Solutions*", Computer Science Press., 1996.

[4] Graunke, G. and S. Thakkar, "*Synchronization Algorithms for Share-Memory Multiprocessors*", Computer, June 1990, pp 60-69.

[5] Fisher, M., and Lisitsa, A., "*The deductive verification of cache coherence protocols*", proceedings of AVOCS'03, 2003.

[6] E. Ostïa, J. Juan Chico, J. Viejo, M. J. Bellido, D. Guerrero, A. Millαn & P. Ruiz-de-Clavijo, "*A SOC design methodology for LEON2 on FPGA*", XII Taller Iberchip, IWS 2006

[7] Mohamad R. Neilforoshan, "*Syncronization and cache coherence in computer*"

[8] Jiri Gaisler: "*LEON2 Processor User's Manual*", Gaisler Research, 2004

[9] "*The SPARC Architecture Manual, Version 8*", SPARC International Inc., 1992.

[10] Jiri Gaisler. Mailing list leon sparc@yahoogroups.com.

[11] "*AMBA (tm) Specification, Rev. 2.0*", ARM Limited, 1999. http://www.arm.com/

[12] "*GR-XC3S1500 Development Board-User Manual*", Gaisler Research/ Pender Electronics, 2006.

[13] "Getting Started with RTEMS", Online Applications Research Corp., 2003. http://www.rtems.com/RTEMS/rtems.html

[14] "GRMON-User's Manual", Gaisler Research, 2005.

[15] "*ISE 6.3i Release Notes and Installation Guide*", Xilinx Inc. 2004

[16] "*ISE In-Depth Tutorial*", Xilinx Inc. 2004. http://ftp.xilinx.com

[17]"XST User Guide", Xilinx Inc. 2004. http://support.xilinx.com

[18] Daniel Mattson, Marcus Christensson, "*Evaluation of synthesizable CPU cores*", Master's Thesis Department of Computer Engineering Gothenburg 2004.

[19] "FPGA Tutorial", http://www.fpga4fun.com

[20] Vikas Agarwal, M.S. Hrishikesh, StephenW. Keckler, Doug Burger. *"The End of the Road for Conventional Microarchitectures"*. , 27th Annual International Symposium on Computer Architecture.

[21] Adve, S.V.Gharachorloo. "*Shared memory consistency models: a tutorial*", IEEE Computer, December 1996

[22]Dubois M., C Scheurich, "Synchronization, Coherence, and event ordering in Multiprocessor", Computer, Feb. 1988, 9-21.