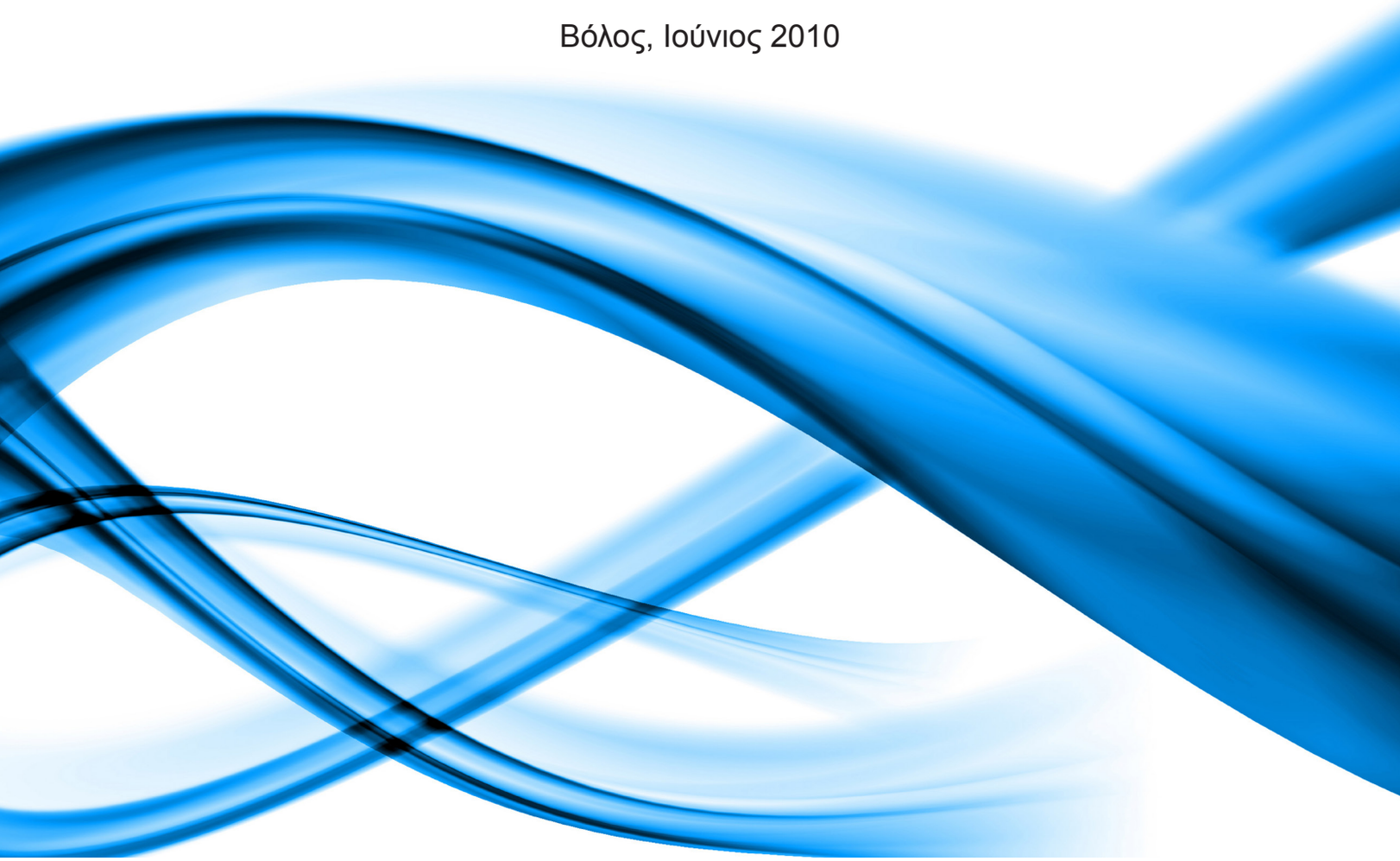
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Μελέτη μετατροπής του πυρήνα του επεξεργαστή Leon3 από αρχιτεκτονική SPARC σε MIPS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αρβανιτάκης Ιωάννης, Μάμαλης Δημήτριος

Βόλος, Ιούνιος 2010



Ευχαριστίες

Ξεκινώντας την συγγραφή αυτής της εργασίας, θα θέλαμε από κοινού να ευχαριστήσουμε όλους όσους συνέβαλαν στην δημιουργία της. Τον κ. Γ. Σταμούλη, επιβλέπων καθηγητή και Πρόεδρο του τμήματος, για την αποδοχή του. Τον κ. Γ. Δημητρίου, επιβλέπων καθηγητή, για την υπομονή του, την επιμονή του, τις συμβουλές και τις παρατηρήσεις του καθ' όλη την διάρκεια της εκπόνησης. Τον Δ. Συρίβελη η εμπειρία του οποίου μας έβγαλε πολλές φορές από την δύσκολη θέση.

Όλους τους φίλους, συγγενείς και συνεργάτες που μας υποστήριξαν όλα αυτά τα χρόνια και χωρίς αυτούς η πορεία μας αυτή θα ήταν ακόμα πιο δύσκολη και πολύ λιγότερο ευχάριστη.

Περιεχόμενα

1. Εισαγωγή

Σκοπός αυτής της εργασίας είναι η δημιουργία ενός μικροεπεξεργαστή αρχιτεκτονικής MIPS με βελτιωμένη δυνατότητα χειρισμού και ελέγχου των λειτουργιών του και η υλοποίησή του σε ένα Field-Programmable Gate Array (FPGA). Για να γίνει αυτό εφικτό, αποφασίσαμε να βασιστούμε στον ήδη υλοποιημένο μικροεπεξεργαστή αρχιτεκτονικής SPARC, τον Leon3, που δημιουργήθηκε από την Gaisler Research. Ο Leon3, παρέχει αρκετές λειτουργικές μονάδες οι οποίες παρέχουν πολλές δυνατότητες που λείπουν από τα υλοποιημένα μοντέλα MIPS ανοικτού κώδικα.

Αρχικά, παρουσιάζεται η λεπτομερής μελέτη των δύο διαφορετικών μικροεπεξεργαστών όπου γίνεται σαφές το τι χρειάζεται να τροποποιηθεί ώστε να φτάσουμε στο επιθυμητό αποτέλεσμα. Ακολουθεί η περιγραφή των βημάτων που έγιναν με σκοπό την μετατροπή της αρχιτεκτονικής του πυρήνα του επεξεργαστή Leon3, την προσομοίωση του τελικού κυκλώματος και τέλος την υλοποίησή του σε FPGA.

Καταλήγουμε στην σύνθεση ενός πιο ευέλικτου μικροεπεξεργαστή, στον οποίο συνδυάζονται η καλή λειτουργικότητα του Leon3 (σύστημα κρυφών μνημών, AMBA bus κ.α.) με τα πλεονεκτήματα της αρχιτεκτονικής συνόλου εντολών MIPS (χαμηλότερη κατανάλωση, μικρότερο μέγεθος κ.α.) και παρέχεται με απόλυτη συμβατότητα με τα μοντέλα υλοποίησης κυκλωμάτων FPGA. Ο μικροεπεξεργαστής αυτός, μπορεί να χρησιμοποιηθεί σε ποικίλες εφαρμογές και παρέχει πολλές δυνατότητες για περαιτέρω ανάπτυξη για εκπαιδευτικούς και ερευνητικούς σκοπούς.

2. Αρχιτεκτονική Υπολογιστών

2.1 Γενικά

Αρχιτεκτονική υπολογιστών είναι το εννοιολογικό σχέδιο και η θεμελιώδης λειτουργική δομή ενός υπολογιστικού συστήματος. Είναι ένα σχεδιάγραμμα και μια λειτουργική περιγραφή των απαιτήσεων και των εφαρμογών σχεδίου για τα διάφορα μέρη ενός υπολογιστή, που εστιάζουν κατά ένα μεγάλο μέρος στον τρόπο με τον οποίο η κεντρική μονάδα επεξεργασίας (ΚΜΕ) εκτελεί εσωτερικά και έχει πρόσβαση στις διευθύνσεις στη μνήμη.

Η αρχιτεκτονική υπολογιστών περιλαμβάνει τουλάχιστον 3 κύριες υποκατηγορίες:

1. Την αρχιτεκτονική του συνόλου εντολών (Instruction Set Architecture ή ISA). Είναι η αφηρημένη εικόνα ενός συστήματος υπολογισμού που βλέπει ένας προγραμματιστής γλώσσας μηχανής ή συμβολικής γλώσσας (assembly), συμπεριλαμβανομένων του συνόλου εντολών, του μεγέθους λέξης, των τρόπων διευθυνσιοδότησης μνήμης, των καταχωρητών του επεξεργαστή, και της απεικόνισης των διευθύνσεων και των δεδομένων.

2. Τη μικροαρχιτεκτονική, επίσης γνωστή ως οργάνωση υπολογιστών, που είναι μία περιγραφή του συστήματος σε χαμηλότερο επίπεδο, πιο συγκεκριμένο και λεπτομερές, που περιλαμβάνει τον τρόπο με τον οποίο τα κύρια μέρη του συστήματος διασυνδέονται και πώς επικοινωνούν προκειμένου να εφαρμοστεί το ISA. Το μέγεθος της κρυφής μνήμης ενός υπολογιστή για παράδειγμα, είναι ένα οργανωτικό ζήτημα που δεν έχει καμία σχέση γενικά με το ISA.

3. Το σχέδιο συστήματος που περιλαμβάνει όλα τα άλλα τμήματα υλικού μέσα σε ένα σύστημα υπολογισμού όπως:

- το σύστημα διασύνδεσης, όπως οι δίαυλοι επικοινωνίας και οι μεταγωγείς,
- τους ελεγκτές και την ιεραρχία μνήμης,

- μηχανισμούς χωρίς φόρτο της ΚΜΕ, όπως η άμεση πρόσβαση μνήμης (DMA),
- ζητήματα όπως η πολυεπεξεργασία.

Μόλις διευκρινιστούν το ISA και η μικροαρχιτεκτονική, η συσκευή πρέπει να σχεδιαστεί στο υλικό. Αυτή η διαδικασία σχεδίου καλείται *εφαρμογή*. Η εφαρμογή μπορεί να αναλυθεί περαιτέρω σε τρία (όχι πλήρως ευδιάκριτα) κομμάτια:

* Εφαρμογή λογικής: σχέδιο των blocks που καθορίζονται στη μικροαρχιτεκτονική (πρώτιστα) στα επίπεδα register-transfer και πυλών.

* Εφαρμογή κυκλωμάτων: σχέδιο σε επίπεδο τρανζίστορ των βασικών στοιχείων (πύλες, πολυπλέκτες, latches κ.λπ.) καθώς επίσης και μερικών μεγαλύτερων blocks (ALUs, caches κ.λπ.) που μπορούν να εφαρμοστούν σε αυτό το επίπεδο, ή ακόμα και (εν μέρει) στο φυσικό επίπεδο, για λόγους απόδοσης.

* Φυσική εφαρμογή: σχεδιάζονται τα φυσικά κυκλώματα, τα διαφορετικά τμήματα κυκλωμάτων τοποθετούνται σε ένα chip floor-plan ή σε έναν πίνακα και καθοδηγούνται τα καλώδια που τα συνδέουν.

2.2 Αρχιτεκτονική Συνόλου Εντολών (ISA)

Το ISA είναι ένας κατάλογος όλων των εντολών, και όλων των παραλλαγών τους, που ένας επεξεργαστής μπορεί να εκτελέσει (ή, στην περίπτωση μιας εικονικής μηχανής(virtual machine) ή ενός διερμηνέα, να προσομοιώσει).

Αυτό καθορίζει την προσωπικότητα ενός επεξεργαστή και επηρεάζει έμμεσα το γενικό σχέδιο του συστήματος. Το ISA διευκρινίζει πώς ένας επεξεργαστής λειτουργεί: ποιες εντολές εκτελεί και ποια ερμηνεία δίνεται σε αυτές τις εντολές. Περιγράφει ουσιαστικά έναν επεξεργαστή σε λογικό επίπεδο.

Είναι το μέρος της αρχιτεκτονικής υπολογιστών που σχετίζεται με τον προγραμματισμό, και συμπεριλαμβάνει τους εγγενείς τύπους δεδομένων, εντολές, καταχωρητές, τρόπους διευθυνσιοδότησης, αρχιτεκτονική μνήμης, διακοπές και

χειριστές εξαιρέσεων, και εξωτερική επικοινωνία (I/O). Το ISA περιλαμβάνει μια προδιαγραφή του συνόλου των opcodes (κωδικών λειτουργίας της γλώσσας μηχανής), των εγγενών εντολών που εφαρμόζονται από μία συγκεκριμένη σχεδίαση της ΚΜΕ. Στο ISA επίπεδο, μπορούμε να διαιρέσουμε τα σχέδια σε δύο κατηγορίες: CISC και RISC.

2.3 Είδη αρχιτεκτονικών συνόλου εντολών

2.3.1 CISC

Τα συστήματα αρχιτεκτονικής συνόλου εντολών CISC (Complex Instruction Set Computing) έχουν ένα αρκετά μεγάλο σύνολο σύνθετων εντολών, εντολών δηλαδή που ολοκληρώνονται σε περισσότερους του ενός κύκλους μηχανής. Η εξέλιξή τους βασίστηκε σε δύο άξονες. Αφενός το κόστος των μνημών RAM και των αποθηκευτικών μέσων ήταν ιδιαίτερα υψηλό κάνοντας έτσι επιτακτική την ανάγκη για προγράμματα μικρού μεγέθους. Αφ' ετέρου, με το σύνολο σύνθετων εντολών ο προγραμματισμός των επεξεργαστών γινόταν πολύ πιο εύκολος για τους προγραμματιστές. Όσο το κόστος αυτών των μονάδων μειώνεται η ανάγκη για CISC αρχιτεκτονικές δεν είναι τόσο επιτακτική. Σήμερα η τεχνική επικάλυψης έχει επιτρέψει τις πολύ υψηλές ταχύτητες ρολογιού που δεν μπορούν να εφαρμοστούν σε αρχιτεκτονικές CISC. Ακόμα και οι x86 δεν είναι πια CISC, από τη στιγμή που όλες οι σύνθετες εντολές του συνόλου εντολών IA32 μετατρέπονται εσωτερικά σε διαδοχικές απλούστερες εντολές RISC, ώστε να γίνει εφικτή η τεχνική επικάλυψης. Παράλληλα οι σύγχρονοι compilers αποφεύγουν τις σύνθετες – και συνεπώς πιο αργές – εντολές, οι οποίες παρέχονται κυρίως για λόγους συμβατότητας.

2.3.2 RISC

Εν αντιθέσει με την CISC αρχιτεκτονική, η RISC (Reduced Instruction Set Computing) χρησιμοποιεί ένα σύνολο απλών εντολών που ολοκληρώνονται σε έναν κύκλο μηχανής. Αναπτύχθηκε στα εργαστήρια της IBM στα μέσα της δεκαετίας του 1970 βασισμένη στην ιδέα ότι ένας υπολογιστής χρησιμοποιεί κατά κόρον μόνο το

20% του συνόλου των εντολών του. Έτσι συνθέτοντας ένα σύνολο απλών εντολών κατασκευάστηκε ένας επεξεργαστής με πολύ λιγότερα τρανζίστορ και επομένως με μικρότερο κόστος κατασκευής ο οποίος εκτελούσε περισσότερες εντολές στην μονάδα του χρόνου από τα αντίστοιχα CISC συστήματα. Ωστόσο η μείωση αυτή στον αριθμό των εντολών, είχε ως αποτέλεσμα ο κώδικας προγραμματισμού των επεξεργαστών να μεγαλώσει πολύ σε μέγεθος, πράγμα που, όπως αναφέραμε, σήμερα δεν αποτελεί τόσο μεγάλο πρόβλημα καθώς το κόστος των μνημών RAM και των αποθηκευτικών μέσων έχει μειωθεί δραματικά σε σχέση με την εποχή που αναπτύχθηκαν τα συστήματα αυτά. Η IBM επέμεινε στον σχεδιασμό RISC συστημάτων με διασημότερο όλων αυτό που χρησιμοποιείται μέχρι σήμερα στα PowerPC. Παράλληλα εταιρίες όπως η SUN και η MIPS Technologies ανέπτυξαν επιτυχώς τις δικές τους SPARC και MIPS αρχιτεκτονικές που χρησιμοποιούνται ευρέως σε embedded και low-end εφαρμογές και με τις οποίες θα ασχοληθούμε σε αυτήν την μελέτη. Οι αρχιτεκτονικές αυτές παλιότερα χρησιμοποιούνταν και σε high-end εφαρμογές. Από τότε η MIPS δε βγάζει πια high-end επεξεργαστές, ενώ η SUN έχει πουλήσει τα δικαιώματα των high-end SPARC – που ακόμα σχεδιάζονται – σε διάφορες εταιρίες.

2.3.3 EPIC

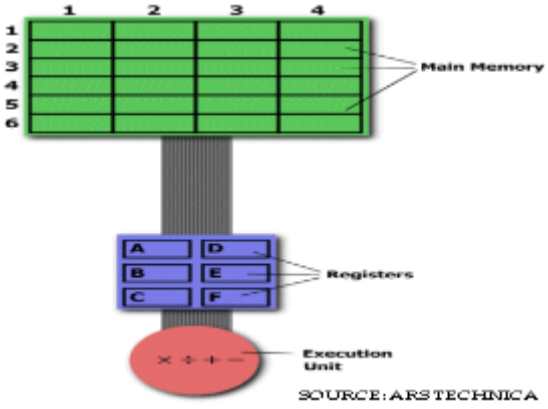
Παράλληλα, στον απόηχο της σύγκρουσης ανάμεσα στις δύο προαναφερθείσες αρχιτεκτονικές, αναπτύχθηκε από τις Hewlett-Packard και Intel ένα διαφορετικό μοντέλο αρχιτεκτονικής, η EPIC (Explicitly Parallel Instruction Computing) που προτείνει την παράλληλη εκτέλεση εντολών, όμως ξεφεύγει από τα πλαίσια της μελέτης μας και δεν θα ασχοληθούμε περαιτέρω με αυτό.

2.3.4 CISC vs RISC

Ο απλούστερος τρόπος να εξεταστούν τα πλεονεκτήματα και τα μειονεκτήματα των δύο αρχιτεκτονικών είναι με την αντιπαράθεση τους σε ένα πραγματικό παράδειγμα.

Παρακάτω είναι ένα διάγραμμα που αντιπροσωπεύει το σχέδιο αποθήκευσης για έναν γενικό υπολογιστή. Η κύρια μνήμη διαιρείται σε θέσεις που αριθμούνται από 1:1 έως

6:4. Η μονάδα εκτέλεσης είναι αρμόδια για την πραγματοποίηση όλων των υπολογισμών. Εντούτοις, η μονάδα εκτέλεσης μπορεί μόνο να λειτουργήσει στο στοιχείο που έχει φορτωθεί σε έναν από τους έξι καταχωρητές (A,B,C,D,E,F). Ας πούμε ότι θέλουμε να βρούμε το γινόμενο δύο αριθμών – ενός που αποθηκεύεται στη θέση 2:3 και ενός άλλου που αποθηκεύεται στη θέση 5:2 – και να αποθηκεύσουμε έπειτα το γινόμενο πίσω στη θέση 2:3.



Εικόνα 1: Ροή αποθήκευσης.

Προσέγγιση CISC

Ο αρχικός στόχος της αρχιτεκτονικής CISC είναι να ολοκληρωθεί ένα έργο μέσα σε λίγες γραμμές assembly. Αυτό επιτυγχάνεται δημιουργώντας επεξεργαστές που είναι σε θέση να εκτελέσουν μια σειρά πράξεων σε μία εντολή. Για αυτό το συγκεκριμένο έργο, ένας επεξεργαστής CISC θα ερχόταν προετοιμασμένος με μία συγκεκριμένη εντολή (θα την ονομάσουμε «MULT»). Η MULT είναι γνωστή ως «σύνθετη εντολή». Λειτουργεί άμεσα στις μονάδες μνήμης του υπολογιστή και δεν απαιτεί από τον προγραμματιστή να καλέσει ρητά οποιεσδήποτε λειτουργίες φόρτωσης ή αποθήκευσης. Μοιάζει πολύ με μια εντολή σε μια γλώσσα υψηλότερου επιπέδου.

Ένα από τα αρχικά πλεονεκτήματα αυτού του συστήματος είναι ότι ο μεταγλωττιστής πρέπει να κάνει πολύ λίγη εργασία για να μεταφράσει μια εντολή γλώσσας υψηλού επιπέδου σε assembly. Επειδή το μήκος του κώδικα είναι σχετικά σύντομο, πολύ λίγη RAM απαιτείται για να αποθηκεύσει τις εντολές. Η έμφαση δίνεται στη δημιουργία των σύνθετων εντολών άμεσα στο υλικό.

Η εργασία αυτή λοιπόν μπορεί να ολοκληρωθεί σε μία μόνο γραμμή κώδικα:

MULT 2:3, 5:2

Προσέγγιση RISC

Οι επεξεργαστές RISC χρησιμοποιούν μόνο τις απλές οδηγίες που μπορούν να εκτελεσθούν μέσα σε έναν κύκλο ρολογιού. Κατά συνέπεια, η εντολή «MULT» που περιγράφηκε ανωτέρω θα πρέπει να διαιρεθεί σε χωριστές εντολές: LOAD που κινεί τα στοιχεία από την μνήμη προς έναν καταχωρητή, PROD που βρίσκει το γινόμενο δύο τελεστέων που βρίσκονται μέσα στους καταχωρητές, και STORE που κινεί τα στοιχεία από έναν καταχωρητή προς τη μνήμη. Προκειμένου να εκτελεσθεί η ακριβής σειρά βημάτων που περιγράφονται στην προσέγγιση CISC, ένας προγραμματιστής θα πρέπει να γράψει τέσσερις γραμμές assembly.

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

Καταρχάς, αυτό μπορεί να φανεί ως ένας πολύ λιγότερο αποδοτικός τρόπος ολοκλήρωσης της πράξης. Επειδή υπάρχουν περισσότερες γραμμές κώδικα, απαιτείται περισσότερη RAM για να αποθηκεύσει τις εντολές σε επίπεδο assembly. Ο μεταγλωττιστής πρέπει επίσης να εκτελέσει περισσότερη εργασία για να μετατρέψει μια εντολή γλώσσας υψηλού επιπέδου στον κώδικα αυτής της μορφής.

Εντούτοις, η στρατηγική RISC έχει επίσης μερικά πολύ σημαντικά πλεονεκτήματα. Επειδή κάθε εντολή απαιτεί μόνο έναν κύκλο ρολογιού για να εκτελεστεί, ολόκληρο το πρόγραμμα θα εκτελεστεί περίπου στον ίδιο χρόνο όπως και η multi-cycle εντολή «MULT». Οι «μειωμένες εντολές» RISC απαιτούν λιγότερα τρανζίστορ από τις σύνθετες εντολές, αφήνοντας μεγαλύτερα περιθώρια για τους καταχωρητές γενικού σκοπού. Επειδή όλες οι εντολές εκτελούνται σε ένα ομοιόμορφο χρονικό διάστημα (δηλ. ένα ρολόι), η επικάλυψη (pipeline) είναι δυνατή.

Ο χωρισμός των εντολών LOAD και STORE μειώνει πραγματικά το φόρτο εργασίας που ο υπολογιστής πρέπει να εκτελέσει. Αφότου εκτελείται μια εντολή CISC «MULT», ο επεξεργαστής σβήνει αυτόματα τους καταχωρητές. Εάν ένας από τους τελεστέους πρέπει να χρησιμοποιηθεί για έναν άλλο υπολογισμό, ο επεξεργαστής πρέπει να ξαναφορτώσει τα δεδομένα από την μνήμη σε έναν καταχωρητή. Στην RISC αρχιτεκτονική, ο τελεστέος θα παραμείνει στον καταχωρητή έως ότου φορτωθεί μια άλλη τιμή στη θέση του.

Η εξίσωση απόδοσης

Η ακόλουθη εξίσωση χρησιμοποιείται συνήθως για την έκφραση της απόδοσης ενός υπολογιστή:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

Εξίσωση 1: Εξίσωση απόδοσης

Η προσέγγιση CISC προσπαθεί να ελαχιστοποιήσει τον αριθμό εντολών ανά πρόγραμμα, θυσιάζοντας τον αριθμό κύκλων ανά εντολή. Η RISC κάνει το αντίθετο, μειώνει τους κύκλους ανά εντολή με κόστος του αριθμού εντολών ανά πρόγραμμα.

2.4 Σύνοψη

Οι απόψεις σχετικά με το μέλλον της κάθε αρχιτεκτονικής διίστανται. Συνοψίζοντας τα παραπάνω αναφέρουμε τα βασικά χαρακτηριστικά και τις διαφορές των δύο αρχιτεκτονικών. Η έμφαση της CISC στο υλικό μειώνει την πολυπλοκότητα του λογισμικού αυξάνει όμως το μέγεθος των κυκλωμάτων και επομένως το κόστος κατασκευής τους. Αντιθέτως, η απλότητα του συνόλου εντολών της αρχιτεκτονικής RISC αφήνει χώρο για περισσότερους καταχωρητές στο κύκλωμα χρησιμοποιώντας τους πιο αποδοτικά για λειτουργίες μεταφοράς δεδομένων από και προς την μνήμη. Οι ξεχωριστές εντολές LOAD και STORE της RISC αρχιτεκτονικής, όπως

προαναφέρθηκε, μειώνουν τον φόρτο εργασίας του επεξεργαστή, αυξάνοντας όμως το μέγεθος του κώδικα (μαζί με τις υπόλοιπες απλές εντολές). Εντέλει, η δυνατότητα επικάλυψης που παρέχεται μέσω της αρχιτεκτονικής RISC έχει στην ουσία οδηγήσει στην εξάλειψη της αρχιτεκτονικής CISC.

CISC	RISC
Έμφαση στο Υλικό	Έμφαση στο Λογισμικό
Περιλαμβάνει multi-clock σύνθετες εντολές	Ενός κύκλου «μειωμένες» εντολές μόνο
Μνήμη-σε-μνήμη: LOAD και STORE που ενσωματώνονται στις εντολές	Καταχωρητής -σε- καταχωρητή: LOAD και STORE είναι ανεξάρτητες εντολές
Μικρός κώδικας, πολλοί κύκλοι ανά δευτερόλεπτο	Λίγοι κύκλοι ανά δευτερόλεπτο, πολλές γραμμές κώδικα
Τρανζίστορ χρησιμοποιούνται για αποθήκευση «σύνθετων» εντολών	Περισσότερα τρανζίστορ για καταχωρητές

Πίνακας 1: Βασικά χαρακτηριστικά και διαφορές CISC και RISC αρχιτεκτονικών

3. Αρχιτεκτονική Risc: Sparc vs Mips

3.1 Γενικά

Στην παράγραφο αυτή θα κάνουμε μια αναδρομή στην ιστορία των δύο αρχιτεκτονικών με τις οποίες θα ασχοληθούμε. Αρχικά περιγράφεται η διαδρομή της SPARC αρχιτεκτονικής και στην συνέχεια η αντίστοιχη της MIPS.

- **SPARC (από το ακρωνύμιο Scalable Processor Architecture)** είναι μια αρχιτεκτονική RISC που αναπτύχθηκε από τη Sun Microsystems και παρουσιάστηκε το 1986.Οι εφαρμογές της αρχιτεκτονικής SPARC σχεδιάστηκαν αρχικά και χρησιμοποιήθηκαν για συστήματα σταθμών εργασίας και κεντρικών υπολογιστών της SUN SUN-4, που αντικαθιστούν προηγούμενα SUN-3 συστήματά τους βασισμένα στην οικογένεια επεξεργαστών Motorola 68000. Αργότερα, οι επεξεργαστές SPARC χρησιμοποιήθηκαν στους κεντρικούς υπολογιστές SMP που παρήχθησαν, μεταξύ των άλλων, από τις Sun Microsystems, Solbourne και Fujitsu. Η SPARC International είχε σαν πρόθεση να ανοίξει την αρχιτεκτονική SPARC, για την οποία έχουν χορηγήσει άδεια σε διάφορους κατασκευαστές, συμπεριλαμβανομένης της Texas Instruments, Atmel, Cypress Semiconductor, και Fujitsu. Το αποτέλεσμα της πολιτικής της SPARC International, είναι ότι η αρχιτεκτονική SPARC είναι πλήρως ανοικτή. Έχουν υπάρξει τρεις σημαντικές αναθεωρήσεις της αρχιτεκτονικής. Η πρώτη δημοσιευμένη αναθεώρηση ήταν η 32-BIT SPARC έκδοση 7 (V7) το 1986. Η SPARC έκδοση 8 (V8), μία ενισχυμένη αρχιτεκτονική SPARC, κυκλοφόρησε το 1990 και τυποποιήθηκε ως IEEE 1754-1994. Στα πλαίσια αυτής της έκδοσης, υλοποιήθηκε από το Ευρωπαϊκό Κέντρο Διαστημικών Ερευνών και Τεχνολογίας, ο μικροεπεξεργαστής LEON που εξελίχθηκε από την Gaisler Research στις εκδόσεις Leon2, Leon3 και τις αντίστοιχες fault tolerant αρχιτεκτονικές Leon2-FT και Leon3-FT. Το 1993, κυκλοφόρησε από τη SPARC International, η 64-bit αρχιτεκτονική SPARC έκδοση 9(V9). Το 2002

η Fujitsu μαζί με την SUN κυκλοφόρησαν το SPARC Joint Programming Specification 1 (JPS1) στο οποίο περιγράφονται λειτουργίες επεξεργαστών κοινές για τους επεξεργαστές και των δύο εταιριών. Ο πρώτος επεξεργαστής συμβατός με το JPS1 ήταν ο UltraSparcIII που κυκλοφόρησε από την SUN και ο SPARC64 V κατασκευασμένος από την Fujitsu. Τον Δεκέμβριο του 2005 ξεκίνησε από την SUN το πρόγραμμα OpenSPARC T1 που οδήγησε στην σχεδίαση του UltraSPARC T1. Λίγους μήνες μετά, τον Μάρτιο του 2006, η SUN εξέδωσε τον κώδικα του T1 υπό την ισχύ της GNU General Public License. Πιο πρόσφατα, το 2008 κυκλοφόρησε επίσης από την SUN ο UltraSPARC T2, διαθέσιμος μέσω του GPL OpenSPARC T2 Project.

- **MIPS (από το ακρωνύμιο Microprocessor without Interlocked Pipeline Stages)** είναι μια αρχιτεκτονική RISC από τη MIPS Computer Systems (τώρα MIPS Technologies), που εμφανίστηκε όταν το 1981 η ομάδα του John L. Hennessy στο Stanford University ξεκίνησε ένα πρόγραμμα, βασικός σκοπός του οποίου ήταν η αύξηση της απόδοσης των επεξεργαστών μέσω της χρήσης τεχνικών επικάλυψης. Το πρόγραμμα αυτό οδήγησε στον πρώτο επεξεργαστή MIPS. Οι πρόωρες αρχιτεκτονικές MIPS ήταν 32-BIT, ενώ οι πιο πρόσφατες εκδόσεις ήταν 64-BIT. Υπάρχουν πολλαπλές αναθεωρήσεις του MIPS instruction set, που είναι οι MIPS I έως MIPS IV, MIPS B, MIPS32, και MIPS64. Οι τρέχουσες αναθεωρήσεις είναι οι MIPS32 και MIPS64. Αυτές καθορίζουν τον έλεγχο του register set καθώς επίσης και του instruction set. Διάφορες προαιρετικές επεκτάσεις είναι επίσης διαθέσιμες, συμπεριλαμβανομένων των MIPS-3D που είναι ένα απλό σύνολο floating-point SIMD εντολών που αφιερώνονται στο κοινό 3D tasks, MIPS16e που συμπιέζει τη ροή της πληροφορίας για να κάνει τα προγράμματα να λάβουν το λιγότερο χώρο (αντίστοιχο του Thumb encoding της αρχιτεκτονικής ARM), και η πρόσφατη προσθήκη της MIPS MT, νέες multithreading προσθήκες, σύστημα παρόμοιο με το HyperThreading του Pentium 4 της Intel. Ως αποτέλεσμα των παραπάνω εργασιών, οι MIPS επεξεργαστές υπερτερούν έναντι των SPARC ως προς το μέγεθος και την κατανάλωση ισχύος.

3.2 SPARC ISA

Ένας επεξεργαστής SPARC περιλαμβάνει μια μονάδα ακέραιων αριθμών (IU), μια μονάδα floating-point (FPU), και έναν προαιρετικό συνεπεξεργαστή (CP), κάθε μία μονάδα με τους καταχωρητές της. Αυτή η οργάνωση επιτρέπει το μέγιστο συγχρονισμό μεταξύ της integer unit, floating-point unit, και του coprocessor στην εκτέλεση μιας εντολής. Όλοι οι καταχωρητές – με την πιθανή εξαίρεση των συνεπεξεργαστών – είναι 32-BIT. Οι τελεστέοι είναι γενικά ενιαίοι καταχωρητές, ζευγάρια καταχωρητών, ή τετράδες καταχωρητών, αφού τα δεδομένα μπορεί να είναι 32bit, 64bit ή 128bit.

Ο επεξεργαστής μπορεί να είναι σε δύο καταστάσεις: user ή supervisor. Στην κατάσταση supervisor, ο επεξεργαστής μπορεί να εκτελέσει οποιαδήποτε εντολή, συμπεριλαμβανομένων των προνομιούχων εντολών. Στην κατάσταση user, μια προσπάθεια να εκτελεσθεί μια προνομιούχος εντολή θα προκαλέσει ένα trap στο λογισμικό του supervisor.

Integer Unit

Η IU περιέχει τους καταχωρητές γενικού σκοπού και ελέγχει τη γενική λειτουργία του επεξεργαστή. Η IU εκτελεί τις αριθμητικές εντολές ακέραιων αριθμών και υπολογίζει τις διευθύνσεις μνήμης για τις εντολές LOAD και STORE. Διατηρεί επίσης τους μετρητές προγράμματος και ελέγχει την εκτέλεση των εντολών για την FPU και τον CP. Όταν η IU έχει πρόσβαση σε μια εντολή από τη μνήμη, επισυνάπτει στη διεύθυνση ένα αναγνωριστικό χώρου διευθύνσεων (address space identifier ή ASI), οι οποίες κωδικοποιούν εάν ο επεξεργαστής είναι στην κατάσταση user ή supervisor, και εάν η πρόσβαση είναι στη μνήμη εντολών(instruction memory) ή στη μνήμη δεδομένων(data memory).

Floating-Point Unit

Η FPU έχει 32 32-bit floating-point f καταχωρητές. Οι floating-point εντολές load/store χρησιμοποιούνται για να μετακινήσουν τα δεδομένα μεταξύ της FPU και

της μνήμης. Η διεύθυνση μνήμης υπολογίζεται από την IU. Η floating-point unit εκτελεί τις εντολές (FPop) της floating-point αριθμητικής. Η μορφή των floating-point δεδομένων και το σύνολο εντολών συμμορφώνονται στα IEEE πρότυπα για τη δυαδική floating-point αριθμητική, ANSI/IEEE 754-1985. Εντούτοις, η SPARC δεν απαιτεί όλες οι πτυχές των προτύπων, όπως η βαθμιαία ανεπάρκεια, να εφαρμόζονται στο υλικό.

Συνεπεξεργαστής

Το σύνολο εντολών περιλαμβάνει την υποστήριξη για συνεπεξεργαστή. Ο συνεπεξεργαστής έχει το σύνολο καταχωρητών του, η πραγματική διαμόρφωση του οποίου καθορίζεται από την εφαρμογή και είναι κάποιος αριθμός 32-bit καταχωρητών. Οι εντολές load/store του συνεπεξεργαστή χρησιμοποιούνται για να μετακινήσουν τα δεδομένα μεταξύ των καταχωρητών του συνεπεξεργαστή και της μνήμης. Για κάθε load/store στο σύνολο εντολών, υπάρχει μια ανάλογη εντολή load/store του συνεπεξεργαστή.

3.2.1 Τύποι Δεδομένων

Η αρχιτεκτονική SPARC αναγνωρίζει τρεις θεμελιώδεις τύπους δεδομένων:

1. Προσημασμένους ακεραίους 8,16,32,64 bit
2. Μη προσημασμένους ακεραίους 8,16,32,64 bit
3. Κινητής υποδιαστολής 32,64,128 bit

Το μέγεθος ορίζεται ως byte, halfword, word, tagged word(30 bit τιμή και 2 tag bits), doubleword και quadword.

Οι προσημασμένοι ακέραιοι αριθμοί κωδικοποιούν ακέραιους συμπληρώματος ως προς δύο. Οι μη προσημασμένοι ακέραιοι είναι γενικού σκοπού και δεν κωδικοποιούν συγκεκριμένο τύπο δεδομένων. Μπορούν να αναπαριστούν αριθμούς, γραμματοσειρές, τιμές κ.α. Η μορφή των δεδομένων κινητής υποδιαστολής συμμορφώνονται στα πρότυπα της IEEE για τη δυαδική αριθμητική κινητής

υποδιαστολής. Ο τύπος tagged word καθορίζει μία λέξη στην οποία τα δύο ελάχιστα σημαντικά bits αντιμετωπίζονται ως bits ετικέτας.

3.2.2 Καταχωρητές

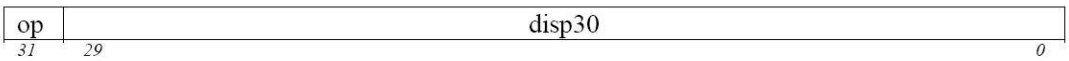
Ένας επεξεργαστής SPARC περιλαμβάνει δύο τύπους καταχωρητών: γενικού σκοπού ή καταχωρητές δεδομένων και καταχωρητές ελέγχου/θέσης. Οι καταχωρητές γενικού σκοπού της IU καλούνται καταχωρητές r, και οι καταχωρητές γενικού σκοπού της FPU καλούνται καταχωρητές f. Οι καταχωρητές του συνεπεξεργαστή εξαρτώνται από την υλοποίηση αυτού.

3.2.3 Εντολές

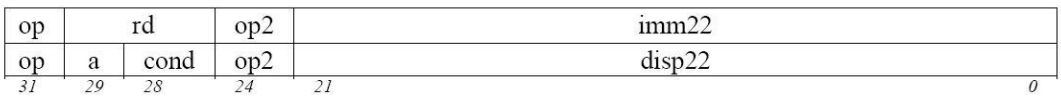
Ο επεξεργαστής έχει πρόσβαση στις εντολές από τη μνήμη και εκτελούνται, ακυρώνονται, ή παγιδεύονται. Οι εντολές κωδικοποιούνται με τρεις 32-bit μορφές και μπορούν να χωριστούν σε έξι γενικές κατηγορίες. Υπάρχουν 72 βασικές λειτουργίες εντολών.

Για κάθε πρόσβαση εντολής και κάθε κανονική πρόσβαση δεδομένων, η IU επισυνάπτει στην 32-bit διεύθυνση μνήμης ένα 8-bit προσδιοριστικό χώρου διευθύνσεων, ή ASI. Το ASI κωδικοποιεί εάν ο επεξεργαστής είναι στην κατάσταση supervisor ή user, και εάν είναι πρόσβαση εντολών ή δεδομένων.

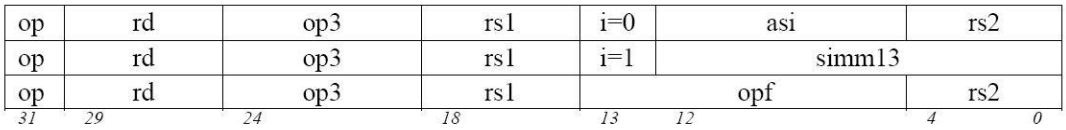
Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



Format 3 (*op* = 2 or 3): Remaining instructions



Εικόνα 2: Μορφοποίηση εντολών

Οι εντολές χωρίζονται σε τρεις βασικές κατηγορίες (Εικόνα 2). Η διαφοροποίησή τους καθορίζεται από το πεδίο op όπως φαίνεται στην εικόνα 3. Στη συνέχεια, οι εντολές που ανήκουν στα format 2 και format 3 διαφοροποιούνται με βάση τα op2 (εικόνα 4) και op3 (εικόνα 5)αντίστοιχα.

op Encoding (All Formats)

Format	*op*	Instructions
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	memory instructions
3	2	arithmetic, logical, shift, and remaining

Εικόνα 3: Κωδικοποίηση πεδίου op

op2 Encoding (Format 2)

op2	Instructions
0	UNIMP
1	unimplemented
2	Bicc
3	unimplemented
4	SETHI
5	unimplemented
6	FBfcc
7	CBccc

Εικόνα 4: Κωδικοποίηση πεδίου op2

op3[5:0] (op=2)

op3 [3:0]	op3 [5:4] = 0	op3 [5:4] = 1	op3 [5:4] = 2	op3 [5:4] = 3
0	ADD	ADDcc	TADDcc	WRASR† WRY‡
1	AND	ANDcc	TSUBcc	WRPSR
2	OR	ORcc	TADDccTV	WRWIM
3	XOR	XORcc	TSUBccTV	WRTBR
4	SUB	SUBcc	MULScc	FPop1 **See Table F-5**
5	ANDN	ANDNcc	SLL	FPop2 **See Table F-6**
6	ORN	ORNcc	SRL	CPop1
7	XNOR	XNORcc	SRA	CPop2
8	ADDX	ADDXcc	RDASR* RDY** STBAR***	JMPL
9			RDPSR	RETT
A	UMUL	UMULcc	RDWIM	Ticc **See Table F-7**
B	SMUL	SMULcc	RDTBR	FLUSH
C	SUBX	SUBXcc		SAVE
D				RESTORE
E	UDIV	UDIVcc		
F	SDIV	SDIVcc		

op3[5:0] (op=3)

op3 [3:0]	op3 [5:4] = 0	op3 [5:4] = 1	op3 [5:4] = 2	op3 [5:4] = 3
0	LD	LDA	LDF	LDC
1	LDUB	LDUBA	LDFSR	LDCSR
2	LDUH	LDUHA		
3	LDD	LDDA	LDDF	LDDC
4	ST	STA	STF	STC
5	STB	STBA	STFSR	STCSR
6	STH	STHA	STDFQ	STDCQ
7	STD	STDA	STDF	STDC
8				
9	LDSB	LDSBA		
A	LDSH	LDSHA		
B				
C				
D	LDSTUB	LDSTUBA		
E				
F	SWAP	SWAPA		

Εικόνα 5: Κωδικοποίηση πεδίου op3

Οι εντολές του SPARC Μπορούν να κατηγοριοποιηθούν σε 6 κατηγορίες: load/store,αριθμητική ακεραίων, μεταφορά ελέγχου (CTI), έλεγχος ανάγνωσης/εγγραφής καταχωρητών, λειτουργίες floating-point, λειτουργίες συνεπεξεργαστή. Οι εντολές load/store είναι οι μόνες με πρόσβαση στη μνήμη. Η αρχιτεκτονική SPARC είναι big-endian, δηλαδή η διεύθυνση μιας doubleword, word, ή halfword είναι η διεύθυνση του σημαντικότερου byte της. Η αύξηση της διεύθυνσης γενικά σημαίνει μείωση της σημαντικότητας της μονάδας που προσπελαύνεται .

Οι εντολές αριθμητικής ακέραιων αριθμών είναι γενικά εντολές σε τριαδικές register-address εντολές που υπολογίζουν ένα αποτέλεσμα που είναι μια πράξη μεταξύ δύο τελεστέων, και είτε το αποτέλεσμα γράφεται στον καταχωρητή προορισμού r είτε απορρίπτεται.

Μία εντολή μεταφοράς του ελέγχου μεταβάλλει την τιμή του pc (NPC).Αυτές οι εντολές μπορούν να κατηγοριοποιηθούν σύμφωνα με τον τρόπο υπολογισμού της διεύθυνσης και με το χρόνο που γίνεται (χωρίς καθυστέρηση, με καθυστέρηση, καθυστέρηση υπό προϋποθέσεις).

Οι εντολές floating-point είναι γενικά τριαδικές register-address εντολές. Υπολογίζουν το αποτέλεσμα 2 τελεστέων και το αποθηκεύουν στον καταχωρητή προορισμού f. Οι εντολές του συνεπεξεργαστή εκτελούνται από τον συνεπεξεργαστή που έχει ενσωματωθεί.

3.2.4 Μοντέλο Μνήμης

Το πρότυπο μνήμης SPARC καθορίζει τη σημασιολογία των λειτουργιών μνήμης όπως τα load και store, και καθορίζει πώς η σειρά με την οποία εκδίδονται αυτές οι λειτουργίες από έναν επεξεργαστή συσχετίζεται με τη σειρά με την οποία εκτελούνται από τη μνήμη. Διευκρινίζει επίσης πώς οι εντολές που έρχονται συγχρονίζονται με τις λειτουργίες της μνήμης. Το πρότυπο ισχύει και για uniprocessors και για τους shared-memory πολυεπεξεργαστές. Το μοντέλο μνήμης της αρχιτεκτονικής SPARC υποστηρίζει μηχανισμούς ισχυρής συνέπειας και ως εκ τούτου είναι συμβατό με τα πρότυπα Partial Store Ordering (PSO) και Total Store Ordering (TSO) η περιγραφή των οποίων ξεφεύγει από τους σκοπούς αυτής της εργασίας.

Η μνήμη είναι το σύνολο των θέσεων που προσεγγίζονται από τις εντολές load/store. Αυτές οι θέσεις περιλαμβάνουν την παραδοσιακή μνήμη, καθώς επίσης και τους I/O καταχωρητές, και τους καταχωρητές που είναι προσιτοί μέσω των ASI. Η πραγματική (ή κεντρική) μνήμη καθορίζεται ώστε να είναι εκείνες οι θέσεις μνήμης που προσεγγίζονται όταν :

· ο τομέας ASI είναι 8, 9, 0xA, ή 0xB, ή
· ο τομέας ASI, μαζί με έναν τομέα σε μια αντίστοιχη είσοδο MMU, υπονοεί μια αναφορά στην πραγματική μνήμη.

Η πραγματική μνήμη δεν πρέπει να προσεγγιστεί από οποιοδήποτε άλλο ASI, που είναι σε έναν καταχωρητή συνεπεξεργαστή, ή που είναι σε έναν βοηθητικό καταχωρητή κατάστασης.

Ένα καθοριστικό χαρακτηριστικό της πραγματικής μνήμης είναι ότι οι λειτουργίες που καθορίζονται σε αυτή είναι χωρίς παρενέργειες, δηλαδή μια load, μια store ή μια ατομική load-store σε μια θέση της πραγματικής μνήμης δεν έχουν καμία αισθητή επίδραση εκτός από εκείνη την θέση. Αντίθετα οι I/O καταχωρητές είναι θέσεις που δεν είναι πραγματική μνήμη. Επομένως μια load, μια store ή μια ατομική load-store σε αυτές τις θέσεις μπορούν να έχουν αισθητές παρενέργειες.

3.3 MIPS ISA

Ένας επεξεργαστής MIPS περιλαμβάνει τον πυρήνα (core), μια μονάδα κινητής υποδιαστολής (FPU) και προαιρετικά έναν συνεπεξεργαστή (CP), κάθε μία μονάδα με τους καταχωρητές της. Αυτή η οργάνωση επιτρέπει το μέγιστο συγχρονισμό μεταξύ του core, της FPU και του CP. Όλοι οι καταχωρητές είναι 32-bit και οι τελεστέοι είναι γενικά ενιαίοι καταχωρητές. Οι εντολές στον MIPS είναι όλες 32-bit. Αυτό σημαίνει ότι καμία εντολή δεν μπορεί να χωρέσει σε μόνο δύο ή τρία byte μνήμης καθώς και δεν μπορεί να είναι μεγαλύτερη από 32-bit. Δεδομένου ότι όλες οι ενέργειες πρέπει να γίνονται μέσα στη σωστή φάση της επικάλυψης και πρέπει να ολοκληρώνονται σε έναν κύκλο ρολογιού, οι εντολές πρέπει να χωράνε στον μηχανισμό επικάλυψης. Για αυτόν τον λόγο υπάρχει διαφοροποίηση στις εντολές πολλαπλασιασμού και διαίρεσης καθώς αυτές είναι αδύνατον να ολοκληρωθούν σε έναν κύκλο.

Ο επεξεργαστής μπορεί να είναι σε δύο καταστάσεις, user ή kernel. Επίσης υπάρχει η περίπτωση σε έναν επεξεργαστή MIPS να υποστηρίζεται και η κατάσταση supervisor (με ένα bit ακόμα) η οποία χρησιμοποιείται πολύ σπάνια.

Core

Ο core περιέχει τους καταχωρητές γενικού σκοπού και ελέγχει τη γενική λειτουργία του επεξεργαστή. Ο core εκτελεί τις αριθμητικές εντολές ακέραιων αριθμών και

υπολογίζει τις διευθύνσεις μνήμης για τις εντολές LOAD και STORE. Διατηρεί, επίσης, τους μετρητές προγράμματος.

Floating-Point Unit

Η FPU έχει 32, 32-bit floating-point f καταχωρητές. Οι floating-point εντολές load/store χρησιμοποιούνται για να μετακινήσουν τα δεδομένα μεταξύ της FPU και της μνήμης. Η διεύθυνση μνήμης υπολογίζεται από την IU. Η floating-point unit εκτελεί τις εντολές της floating-point αριθμητικής. Η μορφή των floating-point δεδομένων και το σύνολο εντολών συμμορφώνονται στα πρότυπα IEEE για τη δυαδική floating-point αριθμητική, IEEE 754, και είναι μονής ή διπλής ακρίβειας.

Συνεπεξεργαστής

Το σύνολο εντολών περιλαμβάνει την υποστήριξη για συνεπεξεργαστή. Ο συνεπεξεργαστής έχει το σύνολο καταχωρητών του, η πραγματική διαμόρφωση του οποίου καθορίζεται από την εφαρμογή και είναι κάποιος αριθμός 32-bit καταχωρητών. Οι εντολές load/store του συνεπεξεργαστή χρησιμοποιούνται για να μετακινήσουν τα δεδομένα μεταξύ των καταχωρητών του συνεπεξεργαστή και της μνήμης.

3.3.1 Τύποι δεδομένων

Η αρχιτεκτονική MIPS αναγνωρίζει τρεις θεμελιώδεις τύπους δεδομένων:

1. Προσημασμένους ακεραίους
2. Μη προσημασμένους ακεραίους
3. Κινητής υποδιαστολής

Το μέγεθος ορίζεται ως byte, halfword, word και doubleworld.

3.3.2 Καταχωρητές

Ένας επεξεργαστής MIPS περιλαμβάνει τους εξής τύπους καταχωρητών: zero(μόνιμα στο μηδέν), assembler temporary, arguments, temporaries, saved temporaries, kernel

και άλλους. Βέβαια εκτός από τον καταχωρητή zero και τον καταχωρητή ra που το υλικό αλληλεπιδρά με τον ρόλο τους, οι υπόλοιποι τύποι δεν σχετίζονται καθόλου με το υλικό και οι προγραμματιστές assembly μπορούν να τους χρησιμοποιήσουν κατά βούληση. Συνολικά είναι 32. Η επιλογή βέβαια του αριθμού καταχωρητών οφείλεται κατά ένα μεγάλο μέρος στις απαιτήσεις λογισμικού, και ένα σύνολο 32 καταχωρητών γενικής χρήσης είναι το δημοφιλέστερο στις σύγχρονες αρχιτεκτονικές. Οι καταχωρητές της FPU καλούνται καταχωρητές f και είναι και αυτοί 32.

3.3.3 Εντολές

Ο επεξεργαστής έχει πρόσβαση στις εντολές από τη μνήμη και εκτελούνται, ακυρώνονται, ή παγιδεύονται. Οι εντολές κωδικοποιούνται με τρεις 32-bit μορφές και μπορούν να χωριστούν σε τρεις γενικές κατηγορίες.

–R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)

–I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)

–J-type (6-bit opcode, 26-bit pseudo-direct address)

Οι εντολές του MIPS χωρίζονται σε 5 κατηγορίες:
αριθμητικές/λογικές/ολίσθησης/σύγκρισης, εντολές ελέγχου, load/store, εξαίρεσης, λοιπές εντολές (μεταφοράς από/σε καταχωρητή ειδικού σκοπού όπως οι hi, lo ή καταχωρητή συνεπεξεργαστή κλπ).

Οι περισσότερες εντολές είναι τριών τελεστέων. Οι αριθμητικές/λογικές εντολές δεν χρειάζεται να διευκρινίσουν θέσεις μνήμης αφού η λειτουργικότητά τους βασίζεται στην χρήση των δεδομένων κατ' ευθείαν από και προς τους καταχωρητές. Έτσι κάθε τέτοια εντολή καθορίζει δύο ανεξάρτητες πηγές (καταχωρητές ή καταχωρητής και σταθερά) και έναν καταχωρητή προορισμού.

Οι load/store εντολές είναι οι μόνες με πρόσβαση στη μνήμη. Μία εντολή μεταφοράς του ελέγχου μεταβάλλει την τιμή του pc. Οι εντολές floating-point είναι τριών τελεστέων. Υπολογίζουν το αποτέλεσμα δύο τελεστέων και το αποθηκεύουν στον

καταχωρητή προορισμού f. Οι εντολές του coprocessor εκτελούνται από τον coprocessor που έχει ενσωματωθεί.

4. LEON3

4.1 Γενικά

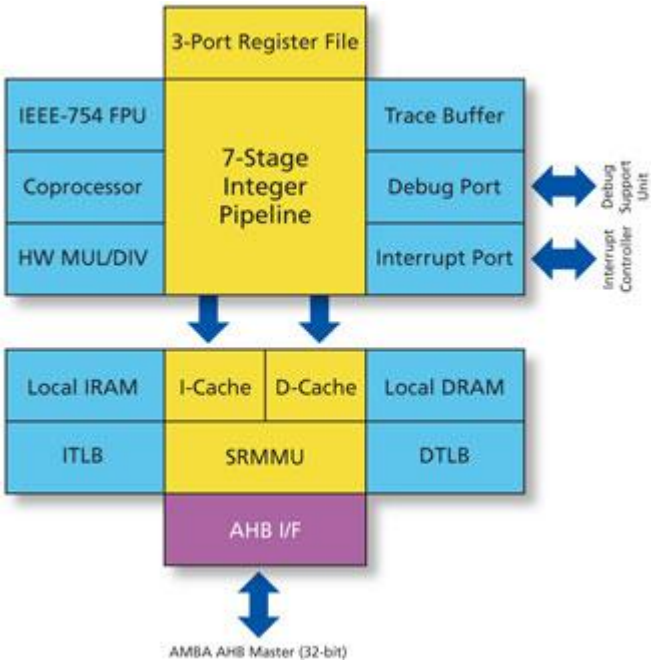
Στο κεφάλαιο αυτό θα αναλύσουμε την δομή και τα χαρακτηριστικά του μικροεπεξεργαστή LEON3 με τον οποίο θα ασχοληθούμε. Θα αναφερθούν τα βασικά χαρακτηριστικά της αρχιτεκτονικής του και οι μετατροπές που έγιναν για την προσαρμογή του στον σκοπό μας.

Πρόκειται για μία open-source υλοποίηση ενός 32-bit μικροεπεξεργαστή SPARC V8e αρχιτεκτονικής, σε γλώσσα περιγραφής υλικού VHDL που αρχικά αναπτύχθηκε από το Ευρωπαϊκό κέντρο διαστημικών ερευνών και τεχνολογίας και στην συνέχεια από την Aeroflex Gaisler AB. Σήμερα, παρέχεται από την Aeroflex Gaisler AB μία αρκετά παραμετροποιημένη μορφή του πυρήνα του κάτω από GNU Pulic Licence, πράγμα που τον καθιστά ιδιαίτερα εύχρηστο για εφαρμογές system-on-a-chip (SOC). Επίσης παρέχεται και με χαμηλού κόστους άδεια για εμπορική χρήση σε system-on-a-chip εφαρμογές. Για κρίσιμες εφαρμογές η Aeroflex Gaisler AB έχει αναπτύξει και μία Fault-Tolerant υλοποίηση του με την ονομασία LEON3-FT. Τέλος, υποστηρίζεται και από μερικά ολοκληρωμένα λειτουργικά συστήματα όπως το RTLinux, eCos, RTEMS, VxWorks και LynxOS.

4.2 Δομικές Μονάδες

Το μοντέλο του LEON3 είναι σχεδιασμένο για embedded εφαρμογές. Βασικά χαρακτηριστικά του είναι ο μηχανισμός επικάλυψης επτά σταδίων όπως και οι ξεχωριστές μονάδες πολλαπλασιασμού και διαίρεσης (HW MUL/DIV), κινητής υποδιαστολής, Instruction και Data Caches Harvard αρχιτεκτονικής (I- και D-caches), μονάδα διαχείρισης μνήμης (MMU) και μονάδα λειτουργιών MAC. Ακόμα, περιλαμβάνονται υλοποιημένες διεπαφές για AMBA-2.0 AHB bus, συνεπεξεργαστή και on-chip αποσφαλματωτή. Η μονάδα ακεραίων, ο φάκελος καταχωρητών γενικού σκοπού, οι κρυφές μνήμες εντολών και δεδομένων και οι ελεγκτές τους θεωρούνται

μαζί με την μονάδα κινητής υποδιαστολής και τον συνεπεξεργαστή ως ο πυρήνας του επεξεργαστή.



Εικόνα 6: Διάγραμμα πυρήνα Leon3

4.2.1 Μονάδα ακεραίων (Integer Unit)

Η μονάδα ακεραίων του LEON3 είναι αυτή που θα μας απασχολήσει περισσότερο. Εδώ υλοποιείται τόσο ο μηχανισμός επικάλυψης του LEON3 όσο και η επικοινωνία με τις κρυφές μνήμες εντολών και δεδομένων που θα αποτελέσουν την μεγαλύτερη πρόκληση. Χρησιμοποιείται για την εκτέλεση πράξεων ακεραίων και τον υπολογισμό διευθύνσεων μνήμης. Διατηρεί τον μετρητή προγράμματος (PC) και ελέγχει την εκτέλεση εντολών στην μονάδα κινητής υποδιαστολής και τον συνεπεξεργαστή. Διαθέτει καταχωρητές γενικού σκοπού οργανωμένους σε παράθυρα μεγέθους οκτώ και μπορεί να διαμορφωθεί από 2 έως 32 σύμφωνα με το πρότυπο της SPARC. Κάθε παράθυρο αποτελείται από 8 καταχωρητές εισόδου και 8 εξόδου, καθώς και 8 τοπικούς καταχωρητές. Επιπλέον υπάρχουν 8 καταχωρητές κοινοί για όλα τα παράθυρα. Οι καταχωρητές εισόδου ενός παραθύρου είναι ίδιοι με τους καταχωρητές εξόδου του προηγούμενου παραθύρου. Ως εκ τούτου, ανάλογα με την διαμόρφωση, υπάρχουν 8+16*N καταχωρητές όπου N ο αριθμός των παραθύρων.

Η διαδικασία της επικάλυψης υλοποιείται στα εξής επτά στάδια:
1. **FE (Instruction Fetch) :** Αν έχει επιλεγεί η χρήση κρυφής μνήμης εντολών, η εντολή φέρεται από εκεί. Ειδάλλως, η διαδικασία αυτή προωθείται στον διαχειριστή μνήμης. Η εντολή έρχεται στην IU στο τέλος αυτού του σταδίου.

2. **DE (Decode) :** Η εντολή αποκωδικοποιείται και υπολογίζονται οι διευθύνσεις των στόχων των εντολών CALL και Branch.

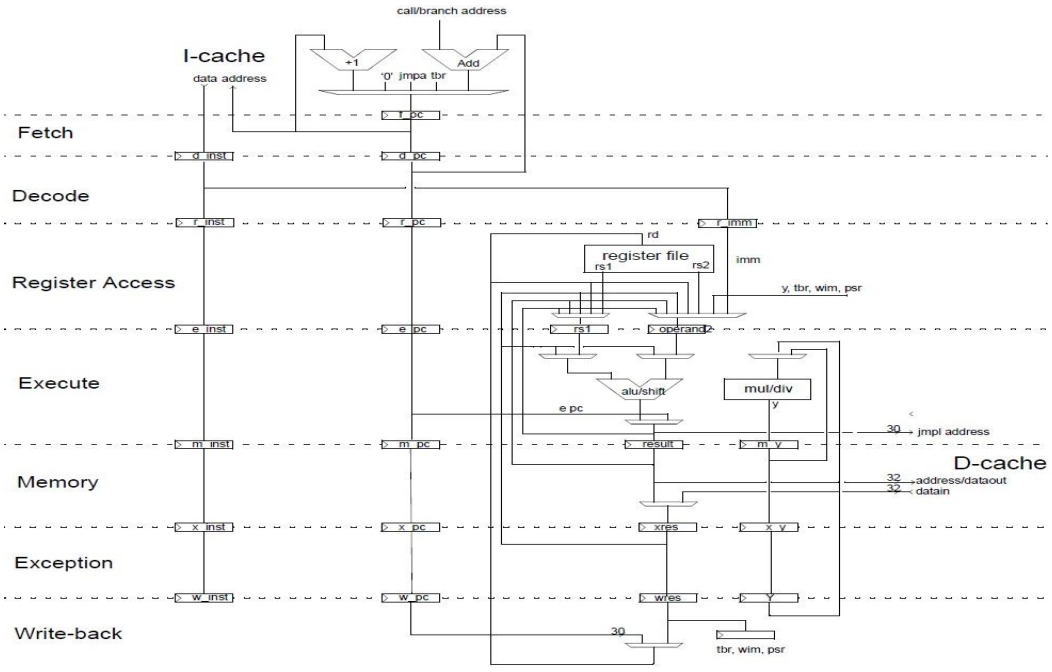
3. **RA (Register Access) :** Γίνεται η πρόσβαση στον φάκελο καταχωρητών για το διάβασμα των τελεστέων.

4. **EX (Execute) :** Λαμβάνουν χώρα οι πράξεις στην αριθμητική λογική μονάδα (ALU), οι λογικές πράξεις και οι πράξεις μετατόπισης. Σε περίπτωση εντολών μνήμης ή σε εντολές JMPL/RETT υπολογίζεται η διεύθυνση.

5. **ME (Memory) :** Προσπελαύνεται η κρυφή μνήμη δεδομένων (D-cache).

6. **XC (Exception) :** Γίνεται η επίλυση των traps των interrupts και πιθανή ευθυγράμμιση των δεδομένων από την D-cache.

7. **WR (Write) :** Τα αποτελέσματα γράφονται πίσω στον φάκελο καταχωρητών.



Εικόνα 7: Ροή δεδομένων της Integer Unit

4.2.2 Υποσύστημα Κρυφών Μνημών

Το υποσύστημα κρυφών μνημών είναι υλοποιημένο, βάσει της αρχιτεκτονικής Harvard, σε δύο ξεχωριστές μονάδες κρυφής μνήμης εντολών και δεδομένων. Κάθε είδος κρυφής μνήμης μπορεί να έχει μέγεθος 1-64 Kbytes για κάθε σύνολο και χωρίζεται σε γραμμές των 16 ή 32 bytes. Επιπλέον, υπάρχει η επιλογή οι κρυφές μνήμες να είναι άμεσης απεικόνισης ή και σύνολο-συσχετιστικές 2 ή 4 δρόμων. Για τις συνολο-συσχετιστικές διατίθενται 3 αλγόριθμοι αντικατάστασης: least-recently-used (LRU), least-recently-replaced (LRR) ή (pseudo-) random. Να σημειωθεί πως ο αλγόριθμος LRR μπορεί να χρησιμοποιηθεί μόνο για συνολο-συσχετιστική κρυφή μνήμη δύο δρόμων.

Σε περίπτωση αστοχίας της κρυφής μνήμης για να μειωθεί το κόστος αστοχίας, χρησιμοποιείται μία τεχνική κατά την οποία τα δεδομένα στέλνονται στον επεξεργαστή παράλληλα με την εγγραφή τους στην κρυφή μνήμη. Ως αποτέλεσμα φορτώνεται μόνο το ζητούμενο μπλοκ.

Η κρυφή μνήμη δεδομένων είναι υλοποιημένη με write-through, no-allocate πολιτική εγγραφής. Κάθε εγγραφή στην κρυφή μνήμη ενημερώνει άμεσα την κύρια μνήμη. Έτσι, σε περίπτωση που ο επεξεργαστής θέλει να γράψει ένα μπλοκ μνήμης που δεν βρίσκεται στην κρυφή μνήμη, μπορεί να γράψει απευθείας στην κύρια μνήμη χωρίς να πρέπει να φορτωθεί προηγουμένως το μπλοκ στην κρυφή μνήμη. Διαθέτει χώρο εγγραφής (store buffer) διπλής λέξης, ώστε να αποφευχθούν κατά το δυνατόν παγώματα του μηχανισμού επικάλυψης που προκαλείται από εντολές αποθήκευσης. Ο buffer αυτός υλοποιείται σε 3 καταχωρητές των 32 bit όπου τοποθετούνται τα προς αποθήκευση δεδομένα μέχρι να σταλούν στον προορισμό τους. Για να αποφευχθεί η πιθανότητα φόρτωσης παλαιών αντιγράφων ο store buffer αδειάζει πριν από την φόρτωση δεδομένων μετά από αστοχία ανάγνωσης.

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) στο AHB bus για έλεγχο μπλοκ που

έχουν υποστεί αλλαγές, το οποίο όμως είναι διαθέσιμο μόνο όταν δεν είναι ενεργοποιημένη η μονάδα διαχείρισης μνήμης, δηλαδή μόνο για φυσικές διευθύνσεις.

4.2.3 Μονάδα Διαχείρισης Μνήμης

Στην υλοποίηση αυτή του LEON3 είναι διαθέσιμη και μία μονάδα διαχείρισης μνήμης (Memory Management Unit - MMU) συμβατή με το πρότυπο της SPARC V8. Όταν δεν είναι ενεργοποιημένη, οι κρυφές μνήμες λειτουργούν κανονικά με φυσικές διευθύνσεις μνήμης. Σε αντίθετη περίπτωση, η MMU δίνει στις κρυφές μνήμες τις φυσικές διευθύνσεις από τις αντίστοιχες εικονικές διευθύνσεις μνήμης. Παράλληλα, σε αυτήν την περίπτωση απενεργοποιείται και το πρωτόκολλο υποκλοπής της AHB αρτηρίας. Παρέχεται η δυνατότητα χρησιμοποίησης διαμοιραζόμενου ή μη διαμοιραζόμενου TLB για τις μνήμες δεδομένων και εντολών. Το TLB είναι πλήρως συσχετιστικό και δύναται να έχει μέγεθος από 2 έως 32 εγγραφές. Η οργάνωσή του και ο αριθμός των εγγραφών δεν είναι ορατά από το λογισμικό και άρα δεν χρειάζονται μετατροπές στο λειτουργικό σύστημα για την υποστήριξή του. Με την ενεργοποίηση της μονάδας διαχείρισης μνήμης, καθίσταται δυνατή η υποστήριξη εξελιγμένων λειτουργικών συστημάτων, όπως διανομών linux και solaris.

4.2.4 Μονάδα Κινητής Υποδιαστολής και διεπαφή συνεπεξεργαστή.

Η SPARC V8 καθορίζει δύο προαιρετικούς συνεπεξεργαστές: μία μονάδα κινητής υποδιαστολής (Floating Point Unit - FPU) και έναν συνεπεξεργαστή υλοποιημένο από τον χρήστη ανάλογα με τις ανάγκες του.

Για την μονάδα κινητής υποδιαστολής, ο LEON3 παρέχει δύο διεπαφές. Μία για το GRFPU που είναι υλοποιημένο σύμφωνα με το πρότυπο IEEE-754 και διατίθεται σε εμπορική μορφή από την Aeroflex Gaisler AB και μία για το Meiko FPU της Sun Microsystems.

Όσον αφορά τον συνεπεξεργαστή, ο LEON μπορεί να παραμετροποιηθεί ούτως ώστε να παρέχει μία γενική διεπαφή πάνω στην οποία θα προσαρμοστεί ο εκάστοτε συνεπεξεργαστής. Η διεπαφή αυτή επιτρέπει την παράλληλη λειτουργία του

συνεπεξεργαστή για να αυξήσει την αποδοτικότητα του συστήματος. Όσο τα απαραίτητα δεδομένα είναι διαθέσιμα, ο συνεπεξεργαστής μπορεί να εκτελεί μία εντολή ανά κύκλο μηχανής. Στο τέλος αυτής τα δεδομένα γράφονται στον φάκελο καταχωρητών αυτού.

4.2.5 Διεπαφές Συστήματος

Η επικοινωνία του επεξεργαστή με τα περιφερειακά είναι δυνατή μέσω αρτηριών Advanced Microcontroller Bus Architecture (AMBA). Υπάρχουν δύο είδη αρτηριών AMBA, η Advanced High-performance Bus (AHB) και η Advanced Peripheral Bus (APB).

Το πρώτο είδος αρχιτεκτονικής αρτηρίας χρησιμοποιείται για να συνδέσει περιφερειακά υψηλών ταχυτήτων, όπως τους ελεγκτές DMA και την 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 δεύτερο είδος, είναι χαμηλότερης πολυπλοκότητας και έχει βελτιωθεί για μικρότερη κατανάλωση ενέργειας για την επικοινωνία με τα βοηθητικά περιφερειακά και τα περιφερειακά γενικού σκοπού.

4.2.6 Διεπαφές Μνήμης, ελεγκτής διακοπών και επιπλέον μονάδες

Οι ελεγκτές υποστηρίζουν

- PROM
- Static Ram
- Synch Dynamic Ram
- I/O με απεικόνιση στην μνήμη.

Ο ελεγκτής διακοπών μπορεί να χειριστεί συνολικά 15 διακοπές που προέρχονται από εσωτερικές ή εξωτερικές πηγές. Ακόμα, παρέχονται υλοποιημένες τόσο μία μονάδα υποστήριξης αποσφαλμάτωσης (Debug Support Unit - DSU), όσο και διεπαφές PCI,

Ethernet MAC 10/100 Mbit. Τέλος, υπάρχει επιλογή για λειτουργία power-save όπου μετά την εμφάνιση του κατάλληλου interrupt η μονάδα επεξεργασίας (IU, FPU) ξυπνάει για να συνεχίσει την λειτουργία της.

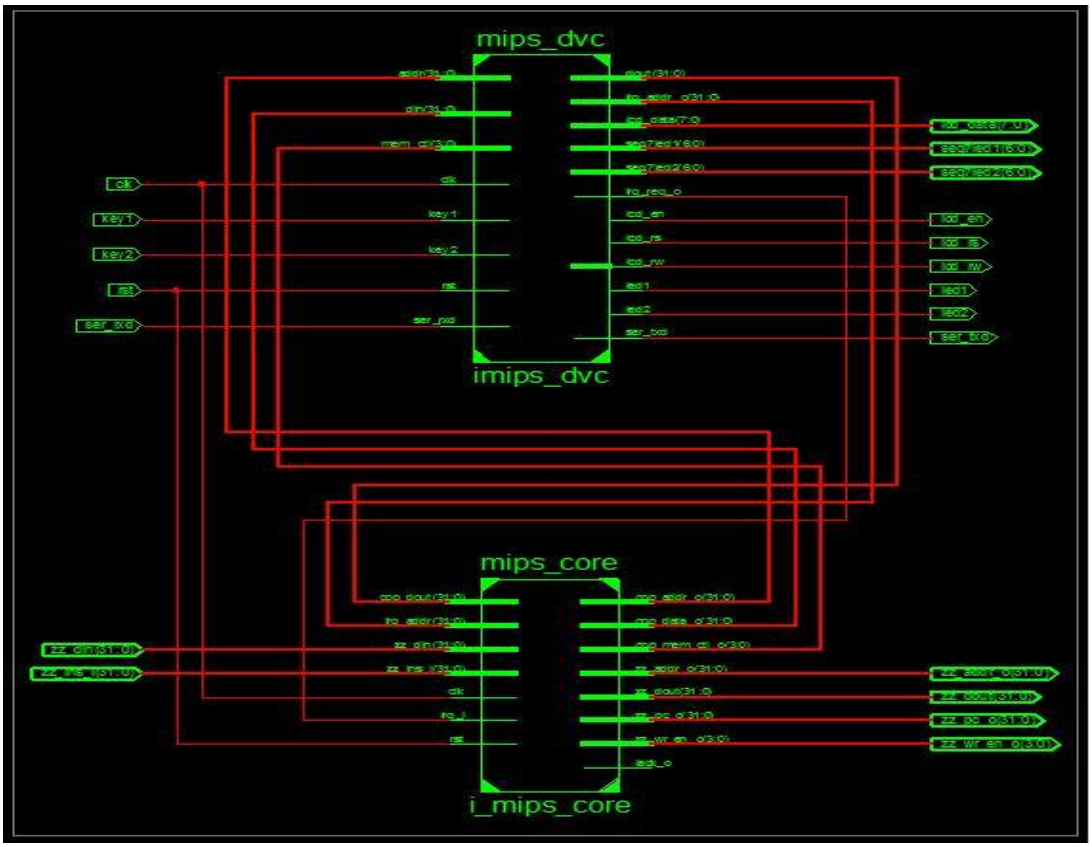
4.2.7 Διαμόρφωση

Τα περισσότερα από τα δομικά στοιχεία του Leon που αναφέρθηκαν παραπάνω δύναται να διαμορφωθούν ή και να απαλειφθούν από την σχεδίαση. Υπάρχουν τρεις τρόποι για να γίνει αυτή η επιλογή. Κατ' αρχάς μπορούμε να χρησιμοποιήσουμε το γραφικό περιβάλλον που παρέχει η Aeroflex Gaisler AB για τον σκοπό αυτό με την εντολή make xconfig. Ακολούθως, με την εντολή make dep αποθηκεύονται οι αλλαγές. Επιπροσθέτως, έχουμε την πιο αυτοματοποιημένη με επιλογή των scripts που υπάρχουν για να διαμορφωθεί ο επεξεργαστής ανάλογα με τις ανάγκες μας. Αυτό μπορεί να γίνει με την εντολή make config BOARD=gr-xc-1500s για το board με την ενσωματωμένη FPGA Spartan3, με το οποίο θα δουλέψουμε. Τέλος, μπορούμε να διαμορφώσουμε τον επεξεργαστή επεμβαίνοντας κατ' ευθείαν στο αρχείο config.vhd. Περισσότερα για την διαμόρφωση την οποία επιλέξαμε στο παράρτημα Β8.

5.MIPS789

5.1 Γενικά

Στο κεφάλαιο αυτό θα ασχοληθούμε με τον επεξεργαστή MIPS789 που βρήκαμε στην κοινότητα opencores και χρησιμοποιούμε αυτόν ως έναν από τους πιο αξιόπιστους που κυκλοφορούν ελεύθερα στο διαδίκτυο. Είναι ένας πυρήνας γραμμένος σε γλώσσα περιγραφής υλικού verilog-2001 και η λειτουργία του αναφέρεται ότι έχει ελεγχθεί με προγράμματα γλώσσας προγραμματισμού C. Ο επεξεργαστής αυτός υποστηρίζει σχεδόν όλες τις εντολές του MIPSI με μηχανισμό επικάλυψης 5 σταδίων.



Εικόνα 8: Δομικές μονάδες του MIPS789

5.2 Βασικά χαρακτηριστικά του MIPS

5.2.1 MIPS Datapath

Σπάζοντας την εκτέλεση των εντολών σε κύκλους ρολογιού μεγιστοποιούμε την απόδοση του συστήματος. Μεταξύ των σταδίων αυτών καταχωρητές αποθηκεύουν τα δεδομένα που υπάρχουν. Κάθε εντολή χρειάζεται από τρία μέχρι πέντε βήματα για να εκτελεστεί. Είναι γνωστό ότι ο κύκλος εντολής της κεντρικής μονάδας επεξεργασίας αποτελείται από μία ακολουθία βημάτων που εκτελούνται το ένα μετά το άλλο σειριακά. Επομένως αν για κάθε βήμα υπάρχει και μία υπομονάδα για την εκτέλεσή του τότε θα μπορούσαμε να εφαρμόσουμε την τεχνική των μερικώς επικαλυπτόμενων λειτουργιών στον κύκλο εντολής ενός επεξεργαστή.

1.Instruction fetch

Σε αυτό το στάδιο προσκομίζεται στον καταχωρητή IF/ID η εντολή που βρίσκεται αποθηκευμένη στην θέση μνήμης με διεύθυνση το περιεχόμενο του PC. Επίσης αυξάνεται η τιμή του PC κατά 4 ώστε να πάει στην επόμενη θέση μνήμης. Δεδομένου ότι το στάδιο αυτό εκτελείται πριν την αποκωδικοποίηση της εντολής, είναι κοινό για όλες τις εντολές.

2.Instruction Decode and register fetch

Η εντολή που βρίσκεται πλέον στον IF/ID αποκωδικοποιείται, διαβάζονται οι δύο καταχωρητές που δίνονται από τα πεδία rs και rt της εντολής και αποθηκεύονται στον καταχωρητή ID/EX. Και αυτό το στάδιο είναι κοινό για όλες τις εντολές.

3. Execution, memory address computation or branch completion

Από το στάδιο αυτό και έπειτα διαφοροποιούνται οι λειτουργίες ανάλογα με το είδος της εντολής. Σε περίπτωση εντολής προσπέλασης μνήμης, υπολογίζεται η διεύθυνση αυτής. Όταν η εντολή είναι αριθμητικής ή λογικής πράξης, εκτελείται η πράξη στην ALU. Τέλος, όταν πρόκειται για εντολή διακλάδωσης γίνεται ο υπολογισμός της

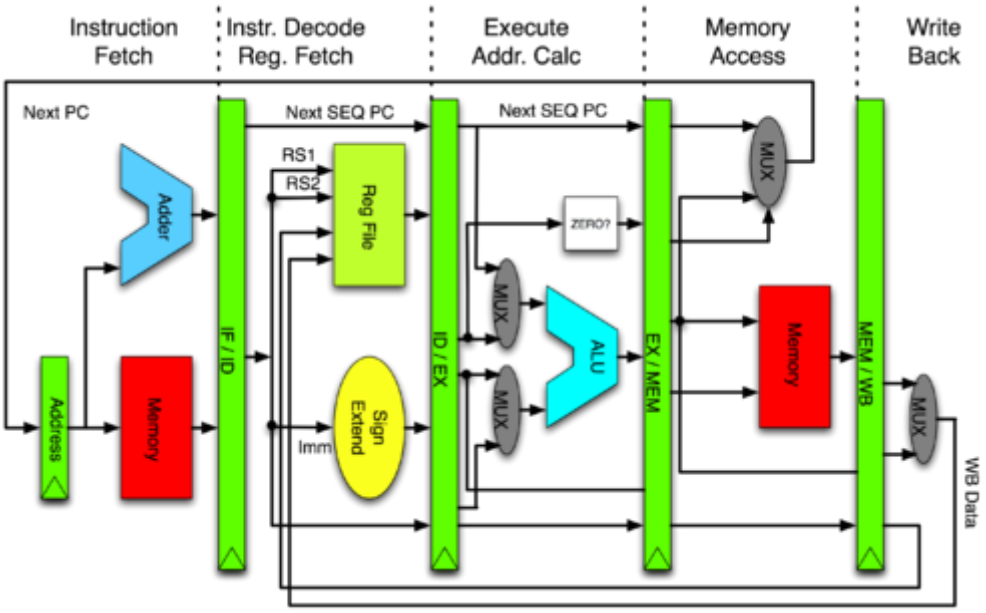
διεύθυνσης διακλάδωσης αν ισχύει η συνθήκη ελέγχου. Τα αποτελέσματα της ALU αποθηκεύονται στον καταχωρητή EX/MEM.

4.Data Memory Access

Σε περίπτωση εντολής προσπέλασης μνήμης, σε αυτό το στάδιο διαβάζονται τα δεδομένα από την μνήμη ή αποθηκεύονται σε αυτήν. Οι εντολές διακλάδωσης αποθηκεύουν στον PC την διεύθυνση προορισμού. Οι υπόλοιπες εντολές περνούν το στάδιο αυτό χωρίς να εκτελείται καμία λειτουργία. Τα αποτελέσματα γράφονται στον καταχωρητή MEM/WB.

5. Write Back

Τέλος, το αποτέλεσμα της όποιας εντολής γράφεται πίσω σε κάποιον από τους καταχωρητές γενικού σκοπού του φακέλου καταχωρητών.



Εικόνα 9: Στάδια επικάλυψης MIPS αρχιτεκτονικής

5.2.2 Pipeline Hazards

Επειδή στον MIPS χρησιμοποιείται η τεχνική του pipeline, θα πρέπει να αναφερθούμε και σε κινδύνους (hazards) που υπάρχουν από αυτή, διότι υπάρχουν καταστάσεις όπου η επόμενη εντολή δεν μπορεί να εκτελεστεί στον επερχόμενο

κύκλο ρολογιού. Υπάρχουν τρία είδη κινδύνων, structural hazards, control hazards και data hazards.

Structural hazards

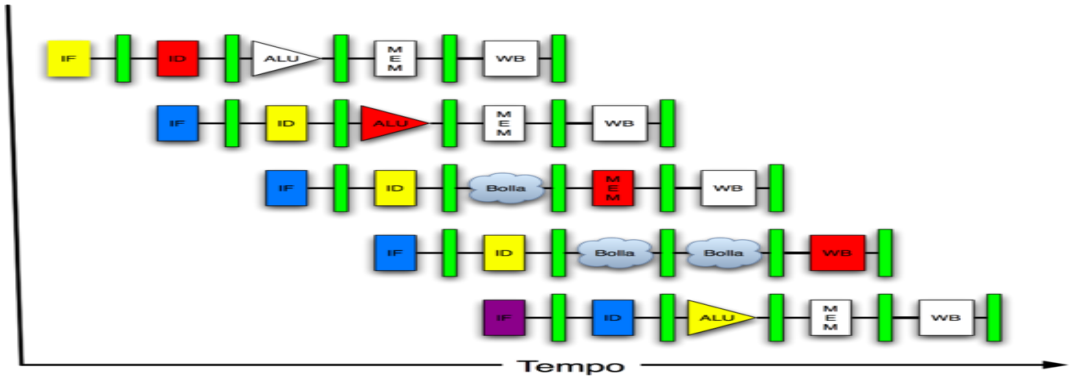
Το συγκεκριμένο είδος κινδύνου εμφανίζεται όταν το υλικό δεν μπορεί να υποστηρίξει τον συνδυασμό εντολών που θέλουν να εκτελεστούν στον ίδιο κύκλο ρολογιού, δηλαδή όταν δύο εντολές θέλουν να χρησιμοποιήσουν το ίδιο κομμάτι hardware. Αυτό μπορεί να ξεπεραστεί προσθέτοντας επιπλέον υλικό.

Control hazards

Το control hazard παρουσιάζεται όταν υπάρχει ανάγκη να παρθεί μία απόφαση βασισμένη στα αποτελέσματα μιας εντολής όταν άλλες εκτελούνται. Η σωστή εντολή δεν μπορεί να εκτελεστεί στο σωστό κύκλο ρολογιού γιατί η εντολή που ήρθε δεν είναι αυτή που χρειάζεται. Παρουσιάζεται όταν έχουμε εντολές branch και οδηγεί σε πάγωμα του επεξεργαστή. Μια λύση είναι η χρήση δυναμικής πρόβλεψης άλματος.

Data hazards

Τα data hazards συμβαίνουν όταν η διαδικασία επικάλυψης πρέπει να παγώσει, επειδή μία φάση μιας εντολής πρέπει να περιμένει να ολοκληρωθεί μια άλλη εντολή λόγω εξάρτησης ανάγνωσης μετά από εγγραφή. Η λύση για αυτό είναι να προστεθεί μία μονάδα υλικού ακόμα που λέγεται forward (προώθηση ή παροχέτευση). Η μονάδα forward δεν μπορεί να εμποδίσει το πάγωμα όταν μία εντολή, που ακολουθεί μια εντολή load,προσπαθεί να διαβάσει τον καταχωρητή στον οποίο γράφει η προηγούμενη εντολή load. Τότε η εντολή θα πρέπει να παγώσει για έναν κύκλο ρολογιού, δηλαδή μέχρι η load να γράψει τον καταχωρητή.

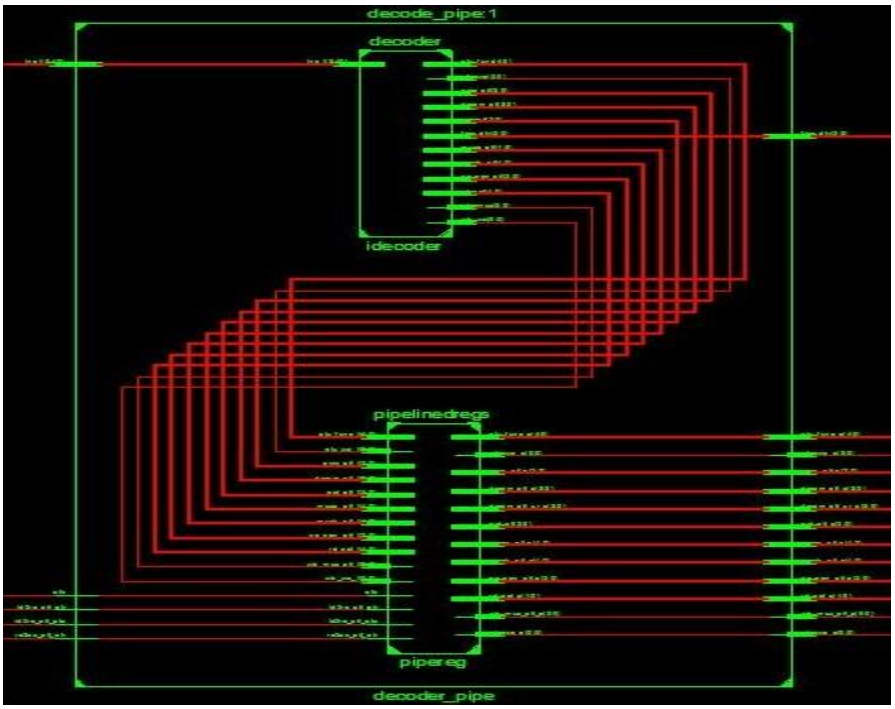


Εικόνα 10: Ροή δεδομένων με πάγωμα του επεξεργαστή.

5.3 Δομή του Mips789

Ο Mips789 αποτελείται από τις μονάδες mem_array και mips_sys. Εντός του mips_sys βρίσκεται ο πυρήνας (core) και ο συνεπεξεργαστής (CP). Το mem-array είναι μια ενοποιημένη εξωτερική μνήμη εντολών και δεδομένων, δύο θυρών και μεγέθους 8K. Ο core εκτελεί, όπως αναφέρθηκε παραπάνω, εντολές MIPSI. Στην mem_array, στέλνει τον pc, την διεύθυνση των δεδομένων και δεδομένα. Από εκεί δέχεται και την επόμενη εντολή και δεδομένα. Μέσω του CP εμφανίζονται τα αποτελέσματα στα leds του board και γίνεται ο έλεγχος για interrupts μιας και εκεί βρίσκεται υλοποιημένος και ο interrupt controller. Εσωτερικά περιέχει μία μνήμη δεδομένων, ως μια υποτυπώδη data cache, για άμεση πρόσβαση στα δεδομένα των εντολών. Από αυτές τις μονάδες θα κρατηθεί μόνο ο core, αφού θα χρησιμοποιήσουμε τις υπόλοιπες λειτουργικές μονάδες του LEON3. Ο MIPS789 έχει κάποιες διαφοροποιήσεις σε σχέση με τον MIPS που αναφέραμε πιο πάνω. Το pipeline του έχει ως εξής:

- **IF&ID (instruction fetch /decode) :** Εδώ έρχεται η επομένη εντολή και αποκωδικοποιείται παράγοντας όλα τα απαραίτητα σήματα ελέγχου για τα επόμενα στάδια. Για το χρονισμό διαθέτει μία μονάδα pipelinedregs όπου τα σήματα ελέγχου περνάνε από τον απαραίτητο αριθμό καταχωρητών ώστε να φτάνουν στα διάφορα στάδια στο σωστό χρόνο.



Εικόνα 11: Decode Stage

- RF (register fetch /generate next pc) : Προσκομίζονται οι τιμές των καταχωρητών και παράγει τον επόμενο pc (στον έλεγχο συμπεριλαμβάνει και την περίπτωση της διακλάδωσης). Στο RF_stage ο core έχει μια μονάδα σύγκρισης όπου ελέγχει τη συνθήκη για εκτέλεση ή όχι των branch εντολών. Σε περίπτωση branch στέλνει στον PC την διεύθυνση της εντολής όπου πρέπει μεταφερθεί η εκτέλεση. Με αυτόν τον τρόπο ελέγχει τα control hazards που ίσως υπάρξουν, χωρίς να χρησιμοποιεί πρόβλεψη άλματος. Επιπλέον στο hazard unit δεν πραγματοποιεί τον έλεγχο του data hazard στην περίπτωση που δύο διαδοχικές εντολές, εκ των οποίων η πρώτη είναι load χρησιμοποιούν τον καταχωρητή $rt η πρώτη για εγγραφή και η δεύτερη για ανάγνωση. Για παράδειγμα η αλληλουχία εντολών

lw $t0, 0($t1)

add $t2, $t3, $t0.

δημιουργεί πρόβλημα του οποίου η επίλυση θα αναλυθεί σε επόμενο κεφάλαιο.

- EXEC (execute instruction) : Η βασική διαφοροποίηση του EXEC έγκειται στο γεγονός ότι εδώ απουσιάζει ο έλεγχος για διακλάδωση αφού έχει ήδη υλοποιηθεί στο στάδιο του RF.

- DMEM (read/write data) : διαβάζει/γράφει δεδομένα από/προς τη μνήμη δεδομένων ή τη συσκευή με τον ίδιο τρόπο που το κάνει και η κλασσική αρχιτεκτονική MIPS.

- WB (result write back) : αποθηκεύει το αποτέλεσμα στον καταχωρητή επίσης χωρίς διαφοροποιήσεις από τις προσταγές της αρχιτεκτονικής MIPS.

6. Μετατροπές μονάδων αρχιτεκτονικής MIPS και σύνδεση με LEON3

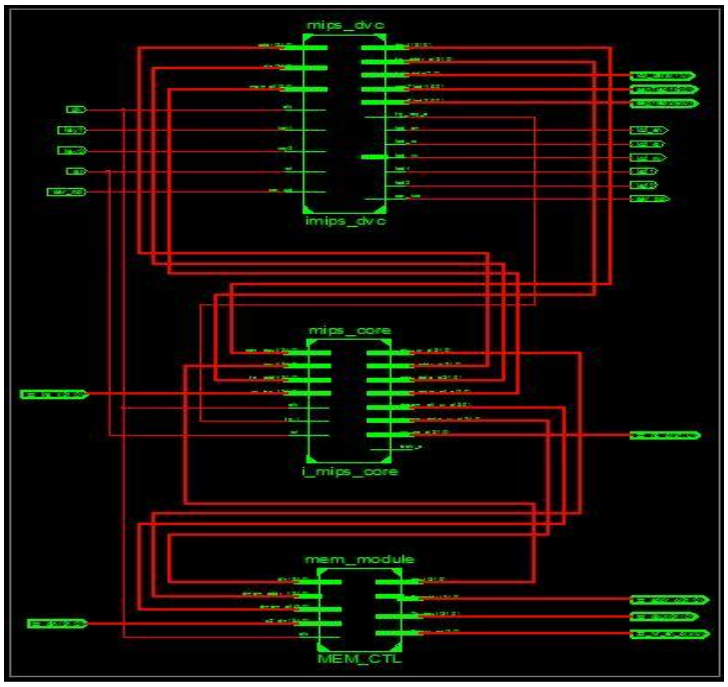
6.1 Γενικά

Στο κεφάλαιο αυτό θα γίνει λεπτομερής περιγραφή των στόχων μας και των επεμβάσεων στην δομή του Mips core. Βασικός στόχος είναι η χρησιμοποίηση μόνο των απολύτως απαραίτητων μονάδων αρχιτεκτονικής Mips. Ως αποτέλεσμα αφαιρέθηκαν τα συστήματα υποτυπώδους κρυφής μνήμης (mem_module), συνεπεξεργαστή (mips_dvc) καθώς και ο φάκελος καταχωρητών (reg_array). Ο Mips core που προέκυψε χρησιμοποιεί αυτούσια την μονάδα κρυφής μνήμης του Leon καθώς και τον φάκελο καταχωρητών αυτού χωρίς να γίνουν αλλαγές στην αρχιτεκτονική τους. Για να γίνει αυτό δυνατό, χρειάστηκε μελέτη των εσωτερικών δομών του Mips core, των σημάτων που αυτά παράγουν, του συγχρονισμού των σταδίων επικάλυψης και πληθώρας άλλων προβλημάτων που προκύπτουν με την τροποποίηση αυτών.Επίσης, θα αναλυθούν και κάποιες διορθώσεις στον αρχικό κώδικα mips όπως η πρόσθεση μονάδας ελέγχου των data hazards και έλεγχος εγγραφής προς τον καταχωρητή zero. Τέλος θα περιγραφεί η διαδικασία σύνδεσης των δύο projects.

6.2 Μονάδα κρυφής μνήμης δεδομένων (mem module)

Η μονάδα αυτή όπως περιγράφηκε παραπάνω, λειτουργεί ως μία υποτυπώδης data cache. Δέχεται από τον core τα σήματα dmem_data_ur_o, alu_ur_o, dmem_ctl_ur_o και παράγει τα σήματα dout (επιστροφή στον core) και zz_addr, zz_dout, και zz_wr_en που πηγαίνουν στην μνήμη. Αυτή θα αντικατασταθεί με την d-cache αρχιτεκτονικής Sparc του Leon. Επομένως τα σήματα εισόδου της mem module συνδέονται στην d-cache και από αυτήν λαμβάνουμε το σήμα dout ως είσοδο στον

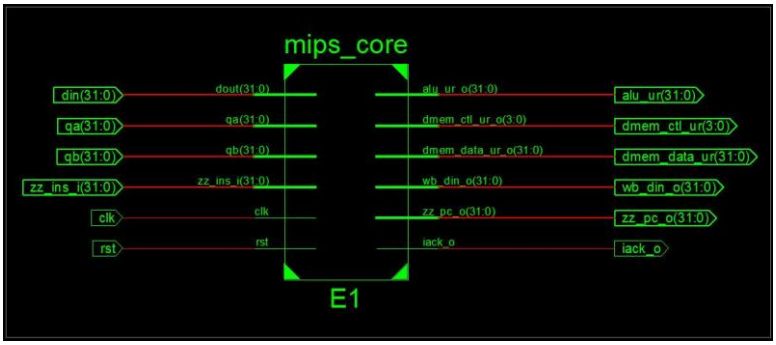
core. Τα υπόλοιπα σήματα εξόδου της mem module αποκόπτονται. Αρχικά η μονάδα αυτή μεταφέρθηκε εκτός του mips core και εν συνεχεία αφαιρέθηκε εντελώς από το όλο project.



Εικόνα 12: Mips Core με εξωτερική μονάδα mem_module.

6.3 Μονάδα συνεπεξεργαστή (mips_dvc)

Στην περίπτωση της μονάδας συνεπεξεργαστή, τα πράγματα είναι αρκετά πιο απλά μιας και τα σήματα εισόδου/εξόδου αυτού δεν επηρέαζαν την λειτουργικότητα του core.



Εικόνα 13: Mips Core χωρίς CP και mem_module.

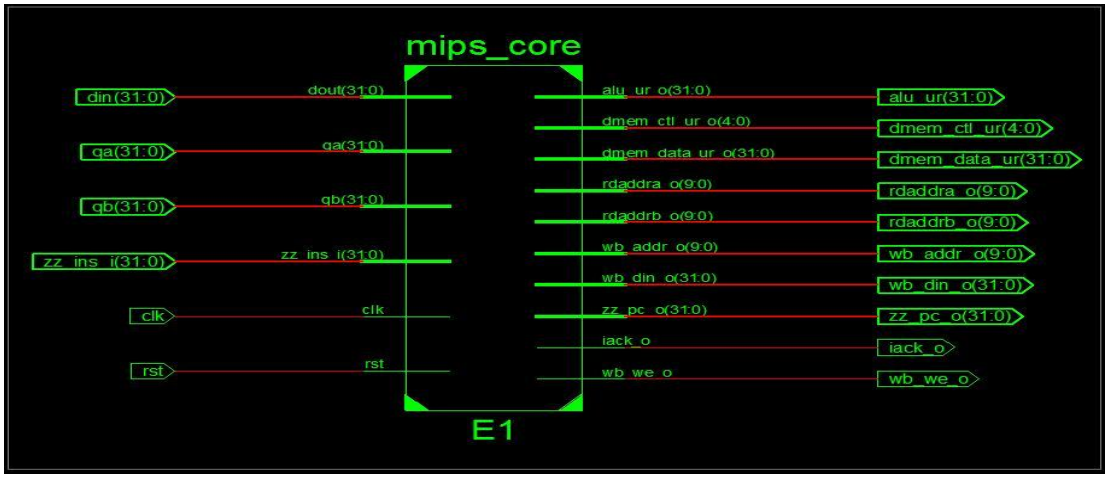
6.4 Φάκελος καταχωρητών (reg_array)

Ακόμα, αφαιρέθηκε ο φάκελος καταχωρητών αρχιτεκτονικής mips για να αντικατασταθεί από αυτόν του Leon. Ο reg_array υλοποιείται στο στάδιο RF_stage. Στον mips core που εξετάζουμε ο φάκελος καταχωρητών δέχεται ως είσοδο τα σήματα data, wraddress, wren, rd_address_a, rd_address_b, clock και παράγει τα σήματα qa, qb, rd_clk, cls. Αντίστοιχα στον Leon, ο φάκελος καταχωρητών , όπου και αποτελεί ξεχωριστή λειτουργική μονάδα, έχει ως σήματα εισόδου τα wdata, waddr, we, raddr1, re1, raddr2, re2, rclk, wclk, testin και ως σήματα εξόδου τα rdata1 και rdata2. Η αντιστοίχηση των σημάτων φαίνεται στον Πίνακα 2. Η ονοματολογία και η σύντομη περιγραφή των σημάτων αρκεί για να γίνει σαφής η λειτουργικότητά τους. Αξίζει να αναφερθούμε μόνο στις περιπτώσεις των δύο διαφορετικών clocks (rclk, wclk) του Leon και στα σήματα re1, re2 τα οποία κρατούνται σταθερά στο 1. Στον mips δεν υπάρχουν ξεχωριστά ρολόγια για τις λειτουργίες read και write επομένως συνδέονται και τα δύο στο κοινό ρολόι. Αντίστοιχα ο mips πάντα μπορεί να διαβάζει από τον φάκελο καταχωρητών πράγμα που οδηγεί στο να συνδέσουμε τα re1 και re2 σταθερά στο '1'. Τέλος, το σήμα testin δεν χρησιμοποιείται και αποκόπτεται.

Μία ακόμα βασική προσαρμογή για την επιτυχή σύνδεση του φακέλου καταχωρητών είναι η αλλαγή των εισόδων του regfile ώστε αυτό να δέχεται 5-bit διευθύνσεις και όχι 8-bit που δεχόταν. Αυτό έγινε επειδή ο MIPS δε χρησιμοποιεί παράθυρα καταχωρητών, αλλά σταθερά 32 ($=2^5$) καταχωρητές. Για να γίνει αυτό στο αρχείο leon3s.vhd δώσαμε τιμές στα IRFBITS και IREGNUM σταθερές ενώ πριν υπήρχε εξάρτηση από το μέγεθος του παραθύρου του regfile. Ο κώδικας παρατίθεται στο παράρτημα Α.1

	LEON3	MIPS789	Σύντομη Περιγραφή
Input	Wdata	Data	Write Data
	Waddr	Wraddress	Write Address
	we	wren	Write Enable
	raddr1	rd_address_a	Read Address
	re1	1	Read Enable
	raddr2	rd_address_b	Read Address
	re2	1	Read Enable
	rclk	clk	Read Clock
	wclk	clk	Write Clock
	testin	--	Diagnostics
Output	rdata1	qa	Read Data
	rdata2	qb	Read Data

Πίνακας 2: Αντιστοίχηση σημάτων φακέλου καταχωρητών.



Εικόνα 14: Mips Core με τα σήματα του φακέλου καταχωρητών.

6.5 Σύνδεση SPARC caches σε MIPS core.

Η σύνδεση της μονάδας κρυφής μνήμης αποτελεί ένα από τα πιο απαιτητικά κομμάτια του όλου project. Αυτό γιατί τα σήματα επικοινωνίας είναι πολλά, χρήζουν λεπτομερούς ανάλυσης και ο χρονισμός τους είναι απολύτως κρίσιμος για την ομαλή λειτουργία του core. Αρχικά μελετήθηκαν όλες οι συνδέσεις με την Integer Unit του Leon. Τα σήματα των cache ομαδοποιήθηκαν ανάλογα με το εκάστοτε στάδιο επικάλυψης στο οποίο συμμετείχαν και ανάλογα προσαρμόστηκαν στο σύστημα επικάλυψης του mips core. Εξ αρχής αποκόπηκαν τα σήματα τα οποία αφορούσαν αποκλειστικά εσωτερικές λειτουργίες της Integer Unit. Τέλος προστέθηκαν δύο μονάδες πολυπλεκτών με τις οποίες γίνεται επιλογή ενός από τα τέσσερα σήματα με τα οποία τροφοδοτούσαν οι cache του Leon την Integer Unit αυτού.

6.5.1 Κρυφή μνήμη εντολών, I-Cache

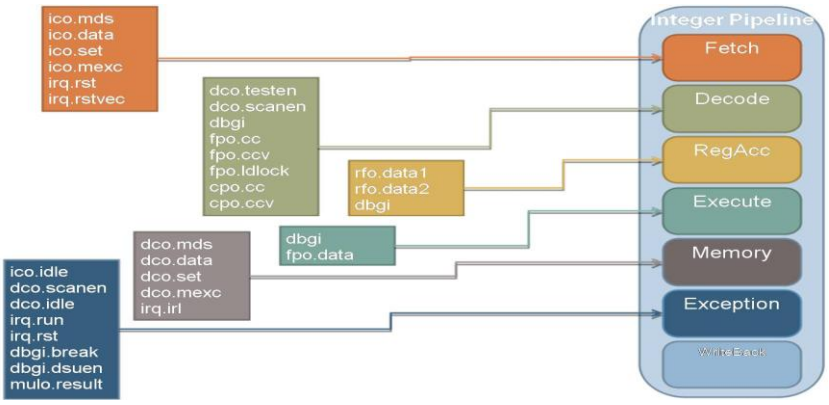
Μία από τις πιο σημαντικές ομάδες σημάτων εισόδου της κρυφής μνήμης εντολών είναι αυτή στην οποία περιέχονται τα σήματα που αφορούν τον pc. Στην I-cache που διαθέτει ο Leon στέλνονται τρία σήματα σχετικά με τον pc, ένα για κάθε διαφορετικό στάδιο του pipeline που επηρεάζουν τον pc. Ως εκ τούτου έχουμε τα σήματα dpc, rpc και fpc. Τα σήματα αυτά συνδέθηκαν το μεν rpc στο pc_next, το δε fpc στο zz_pc_o. Το σήμα dpc αποκόπηκε εντελώς καθώς δεν χρησιμοποιούταν. Παρομοίως τα σήματα fbranch και rbranch ήταν απαραίτητα στα στάδια fetch και reg_access του Leon. Στον

mips χειριζόμαστε μόνο ένα σήμα για τις αντίστοιχες λειτουργίες και έτσι συνδέθηκαν απ' ευθείας στο σήμα branch. Στην τελευταία ομάδα βρίσκονται σήματα που αν και δεν έχουν νόημα για τον mips είναι απαραίτητα για την I-cache και έτσι δόθηκαν σε αυτά οι standard τιμές που θα έκαναν την κρυφή μνήμη λειτουργική. Από τα σήματα εξόδου, αυτά που απαιτούνται για την λειτουργία του mips είναι τα data, set, hold και mds. Το πρώτο μεταφέρει τα δεδομένα προς τον επεξεργαστή σε μορφή τεσσάρων διαφορετικών λέξεων. Από εκεί επιλέγεται η σωστή με την χρήση του σήματος set. Τα hold και mds συνδυάζονται κατάλληλα ώστε να ειδοποιούν πότε είναι διαθέσιμα τα δεδομένα. Συγκεκριμένα, η λειτουργία αυτών των δύο έχει ως εξής: το σήμα mds αφορά την προετοιμασία των δεδομένων και το σήμα hold την αποστολή αυτών στον πυρήνα. Το mds είναι ‘0’ όσο τα δεδομένα προετοιμάζονται. Όταν είναι έτοιμα γίνεται ‘1΄. Αντίστοιχα το σήμα hold είναι ‘0’ καθ’ όλη τη διάρκεια προετοιμασίας των δεδομένων και γίνεται ‘1’ μόλις αυτά είναι έτοιμα για αποστολή στον core. Επίσης, πρέπει να σημειωθεί πως επειδή το σήμα hold όσο είναι ‘0’ παγώνει την I-cache, κάτι που αφορά άμεσα το πρώτο στάδιο επικάλυψης του επεξεργαστή, το σήμα hold είναι αυτό που τελικά ελέγχει και το πάγωμα του επεξεργαστή καθώς χρησιμοποιείται από όλους τους καταχωρητές του. Η αντιστοίχιση όλων των ανωτέρω παρατίθεται στον πίνακα 3.

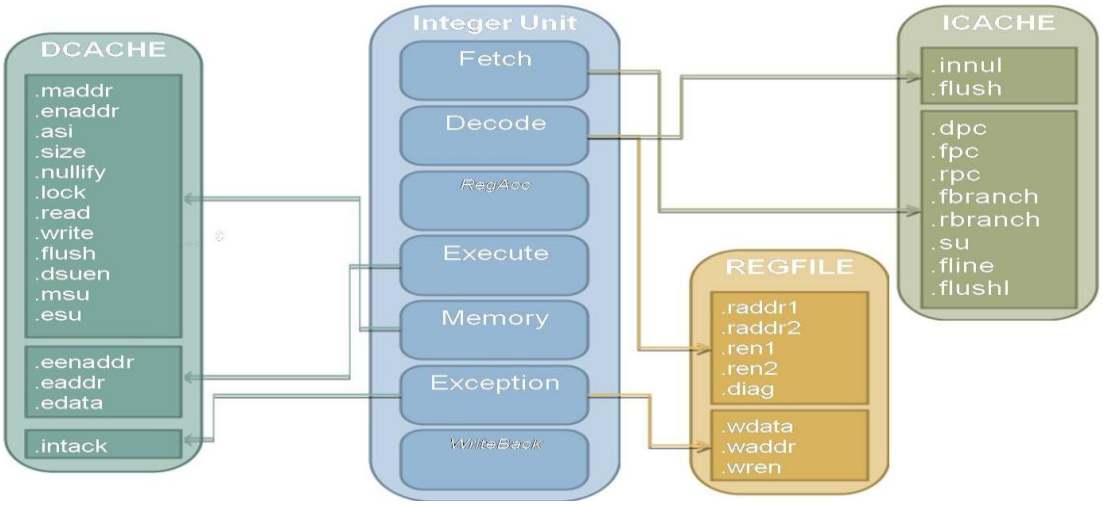
	LEON	MIPS	Σύντομη Περιγραφή
I-cache Input	inull	0	Instruction Nullify
	dpc	--	Decode pc
	rpc	pc_next	Raw pc
	fpc	zz_pc_o	Fetch pc
	fbranch	branch	Instruction branch
	rbranch	branch	Instruction branch
	su	--	Superuser mode

	fline	ifline = 29'b0	Flush line offset
	flush	0	Flush icache
	flushl	0	Flush line
I-cache Output	data	zz_ins_i	Data type
	set	iset	Set data type
	mexc	--	Memory exception
	hold	hold	Hold when ready
	flush	--	Flush in progress
	diagrdy	--	Diagnostic Access
	diagdata	--	Diagnotic data
	mds	imds	Memory data strobe
	cfg	--	Configure
	idle	--	Idle mode

Πίνακας 3: Αντιστοίχηση σημάτων I-cache.



Εικόνα 15: Είσοδοι της IU ανά στάδιο.



Εικόνα 16: Έξοδοι της IU προς D-, I- cache και Regfile.

6.5.2 Κρυφή μνήμη δεδομένων, D-Cache

Παρόμοια με την I-cache, στην D-cache υπάρχουν σήματα σημαντικά για την λειτουργία του mips και άλλα που είτε αποκόπτονται, είτε αρχικοποιούνται σε standard τιμές.

Μία από τις πλέον σημαντικές ομάδες σημάτων είναι αυτή που αφορά την αποκωδικοποίηση των εντολών. Πρόκειται για τα σήματα lock, write, enaddr, eenaddr, read και dsuen. Σε κάθε ένα από αυτά έχουν αντιστοιχηθεί ξεχωριστής σημαντικότητας bits του σήματος dmem_ctl_ur που παράγεται στο στάδιο decode του mips. Το πιο σημαντικό ψηφίο δίνεται στο σήμα lock και παραμένει 0 ανεξαρτήτως εντολής. Όμοια και το dsuen συνδέεται με το λιγότερο σημαντικό ψηφίο. Τα σήματα write και read αντιστοιχίζονται στο δεύτερο και τέταρτο πιο σημαντικό ψηφίο και ενεργοποιούνται κατά τις λειτουργίες store και load αντίστοιχα. Το τρίτο πιο σημαντικό ψηφίο συνδέεται στα σήματα enaddr και eenaddr και ενεργοποιείται τόσο στις εντολές store όσο και στις εντολές load.

Το σήμα asi που αφορά την κατάσταση (mode) του επεξεργαστή και παράγεται κατά το decode. Εδώ παράγεται και το σήμα size που αφορά το μέγεθος της πληροφορίας (word, half word κ.ο.κ.) Συνεχίζοντας, το σήμα maddress μεταφέρει το αποτέλεσμα της alu στην μνήμη. Τα σήματα nulliffy, flush, flushl, msu, esu και intack δεν έχουν

νόημα για τον mips όμως είναι απαραίτητα για την d-cache και για αυτόν τον λόγο δίνονται στην σταθερή τιμή ‘0’.

Από τα σήματα εξόδου, κρατούνται μόνο τα σήματα data, set, hold και mds που έχουν ακριβώς την ίδια λειτουργικότητα με τα αντίστοιχα σήματα εξόδου της i-cache. Η αντιστοίχηση παρατίθεται στον πίνακα 4.

	LEON	MIPS	Σύντομη Περιγραφή
D-cache Input	asi	asi_code	Address space identifier
	maddress	alu_ur	Memory address
	eaddress	eaddr = 32'b0	Execute Address
	edata	dmem_data_ur	Execute Data
	size	size	Data Size
	enaddr	dmem_ctl_ur(2)	Enable Address
	eenaddr	dmem_ctl_ur(2)	Execute Enable Address
	nullify	0	Nullify
	lock	dmem_ctl_ur(4)	Lock cache
	read	dmem_ctl_ur(1)	Read cache
	write	dmem_ctl_ur(3)	Write cache
	flush	0	Flush Cache
	flushl	0	Flush line
	dsuen	dmem_ctl_ur(0)	DSU enable

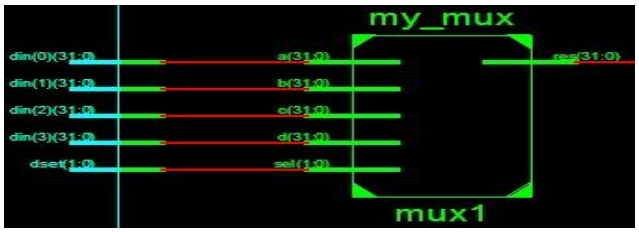
	msu	0	Memory Stage Supervisor
	esu	0	Execution Stage Supervisor
	intack	0	Interrupt Acknowledgment
D-cache Output	data	dout	Data type
	set	dset	Set data type
	mexc	--	Memory exception
	hold	hold	Hold cache
	mds	dmds	Memory data strobe
	werr	--	Write Error
	icdiag	--	I-cache Diagnostic
	cache	--	
	idle	--	Idle mode
	scanen	--	Scan enable
	testen	--	Test enable

Πίνακας 4: Αντιστοίχιση σημάτων D-cache.

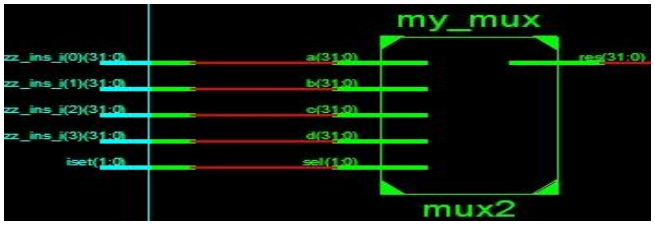
6.5.3 Επιλογή εντολής και δεδομένων

Επεξηγηματικά στην διαδικασία σύνδεσης των caches και εκτός του mips προστέθηκαν δύο 4-σε-1 πολυπλέκτες όπου με βάση τα iset και dset γίνεται η επιλογή μίας από τις 4 λέξεις που προωθούν οι caches μέσω των σημάτων ico.data και

dco.data καθώς η μνήμη στον LEON είναι μέχρι τεσσάρων τρόπων συνολο-συσχετιστική. Στον LEON η επιλογή της λέξης γίνεται εσωτερικά στην IU.



Εικόνα 17: Πολυπλέκτης δεδομένων.



Εικόνα 18: Πολυπλέκτης εντολών.

6.6 Instruction Register/core registers

Ο Mips789 δεν είχε κάποιον register ώστε να κρατάει την εντολή που έρχεται στον πυρήνα. Για αυτό τον λόγο προσθέσαμε τον instruction register που ελέγχεται από το σήμα hold και το imds για το πότε θα περνάει στην έξοδό του την τιμή που έχει στην είσοδο. Να σημειωθεί ότι με αυτόν τον τρόπο ουσιαστικά διασπάται το αρχικώς ενιαίο στάδιο IF/ID σύμφωνα με το οποίο είχε σχεδιαστεί ο MIPS789. Πρακτικά είναι αδύνατον να έχουμε συγχώνευση αυτών των δύο σταδίων και εκτέλεσή τους σε ένα κύκλο κάτι που αποτελούσε ένα από τα σοβαρά λάθη του αρχικού MIPS789. Επιπλέον, επειδή η εντολή λαμβάνεται με βάση τα σήματα hold και mds, το σήμα hold εισήχθηκε σε όλους τους registers του πυρήνα, ώστε να μην επιτρέπεται εγγραφή τους όσο το σήμα hold έχει τιμή ‘0’. Το σήμα mds, που ειδοποιεί τον επεξεργαστή πότε κάποια τιμή από τη μνήμη είναι έτοιμη για ανάγνωση, χρησιμοποιείται από τους registers εντολών και δεδομένων, επειδή είναι οι μόνοι που δέχονται τιμή απ’ ευθείας από τη μνήμη. Επιλεγμένος κώδικας από τους registers παρατίθεται στο παράρτημα Α.2.

Εικόνα 19: Καταχωρητής εντολών.

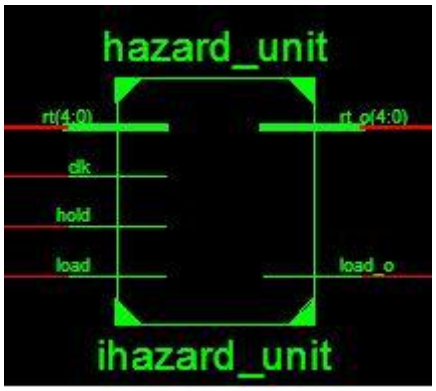
6.7 Διόρθωση διαδικασίας ελέγχου Data Hazards

Κατά τη διάρκεια της μελέτης της επικάλυψης του mips παρατηρήθηκε σοβαρό λογικό λάθος στην πρόβλεψη παγώματος του επεξεργαστή σε περίπτωση εμφάνισης ενός data hazard. Συγκεκριμένα, δεν αναγνώριζε την εξάρτηση ανάγνωσης μετά από εγγραφή σε περίπτωση που δύο διαδοχικές εντολές (με πρώτη μία εντολή load) χρησιμοποιούσαν τον ίδιο καταχωρητή στις κατάλληλες θέσεις.

lw $t0, 0($t1)

add $t2, $t3, $t0.

Για το πρόβλημα αυτό κατασκευάστηκε μονάδα ελέγχου πρόβλεψης τέτοιων κινδύνων της οποίας παραθέτουμε αυτούσιο τον κώδικα verilog. Η hazard unit λαμβάνει από τον mips το σήμα load από το decode stage, καθώς και τον καταχωρητή rt μαζί με τα clk και hold. Κρατάει σε καταχωρητές και επιστρέφει τα σήματα load_o και rt_o στο στάδιο decode με τα οποία γίνεται ο έλεγχος για το πάγωμα ή όχι του επεξεργαστή. Ο κώδικας βρίσκεται στο παράρτημα Α.3



Εικόνα 20: Βοηθητική μονάδα ελέγχου data hazards.

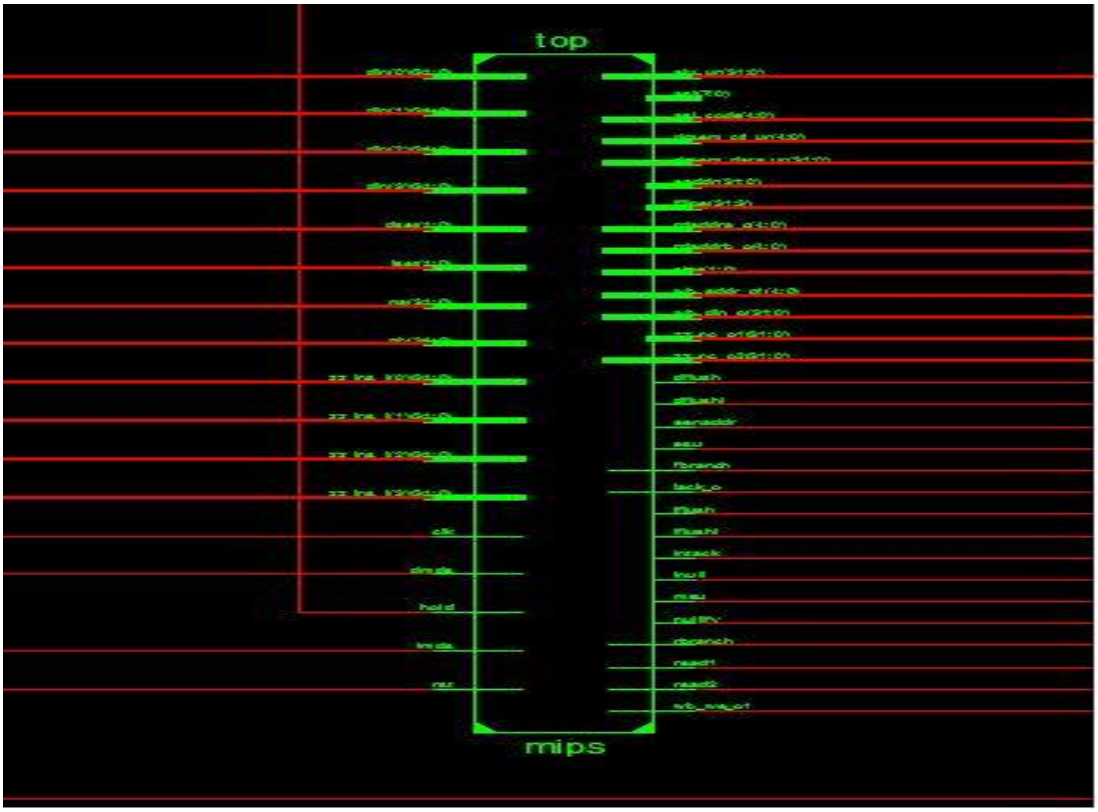
6.8 Σύνδεση VHDL και Verilog project

Όπως έχει ήδη αναφερθεί οι δύο επεξεργαστές είναι γραμμένοι σε διαφορετικές γλώσσες περιγραφής υλικού. Ο Leon είναι σε VHDL ενώ ο Mips σε verilog-2001. Για να συνδεθούν τα δύο project απαιτείται ένα vhdl top-file στο οποίο γίνεται το mapping όλων των ενδιάμεσων σημάτων, αφού δηλωθεί ως component το top-file της verilog και παρατίθεται στο παράρτημα Α.4.

6.9 Ενσωμάτωση Mips789 στο project

Το πρώτο βήμα για να γίνει η σύνδεση απαιτεί να τοποθετηθεί το top file ως component στο libiu.vhd που είναι package του Leon, ώστε να φαίνεται στα αρχεία που χρησιμοποιούν την iu3. Έπειτα στο proc3.vhd αφού η iu3 αφαιρέθηκε, χρησιμοποιείται ο mips και έγινε το port map των σημάτων όπως αναφέρθηκε παραπάνω. Οι κώδικες των δύο αρχείων παρατίθενται στο παράρτημα Α.5. Για να συνδεθούν τα 2 project και να συνεργάζονται με τα εργαλεία που θα χρησιμοποιηθούν αργότερα για τις διαδικασίες synthesis και simulation θα πρέπει τα αρχεία μας να τοποθετηθούν σε φάκελο με όνομα vlog εντός της grlib. Εκεί τοποθετούνται όλα τα αρχεία verilog που περιγράφουν τον mips και την όποια λογική έχει προστεθεί. Ο φάκελος αυτός, με τη σειρά του, τοποθετείται κάτω από το ./lib/gaisler/. Παράλληλα, στο dirs.txt που βρίσκεται στον ίδιο φάκελο πρέπει να καταγραφεί το όνομα του φακέλου vlog ώστε να συμπεριληφθεί στα αρχεία βιβλιοθηκών. Επίσης μέσα στο φάκελο vlog χρειάζεται ένα αρχείο κειμένου

vlogsyn.txt στο οποίο αναφέρονται τα ονόματα των αρχείων verilog που απαιτούνται για την εκτέλεση του project. Τέλος τα αρχεία vhdl που δημιουργήθηκαν, δηλαδή τα top.vhd, my_mux.vhd, reg_zero.vhd τοποθετούνται στον φάκελο ./lib/gaisler/leon3 και τα ονόματά τους συμπληρώνονται με τον ίδιο τρόπο στο vhdlsyn.txt. Από εκεί αφαιρείται το αρχείο iu3.vhd το οποίο πλέον δεν χρειάζεται στο project.



Εικόνα 21: Τελική μορφή Mips Core.

7.Προσομοίωση του Project

7.1 Γενικά

Στο κεφάλαιο που ακολουθεί περιγράφεται η διαδικασία προσομοίωσης του project καθώς και η διόρθωση όλων των σφαλμάτων που εντοπίστηκαν κατά τη διάρκειά της. Μελετήθηκαν διάφοροι τρόποι και εργαλεία με τα οποία θα γινόταν η προσπάθεια ελέγχου λειτουργικότητας του project. Τελικά αποφασίστηκε ότι ο καταλληλότερος τρόπος προσομοίωσης με σκοπό τον έλεγχο της λειτουργικότητας του όλου project είναι μέσω του εργαλείου Modelsim και συγκεκριμένα με την έκδοση 6.3f.

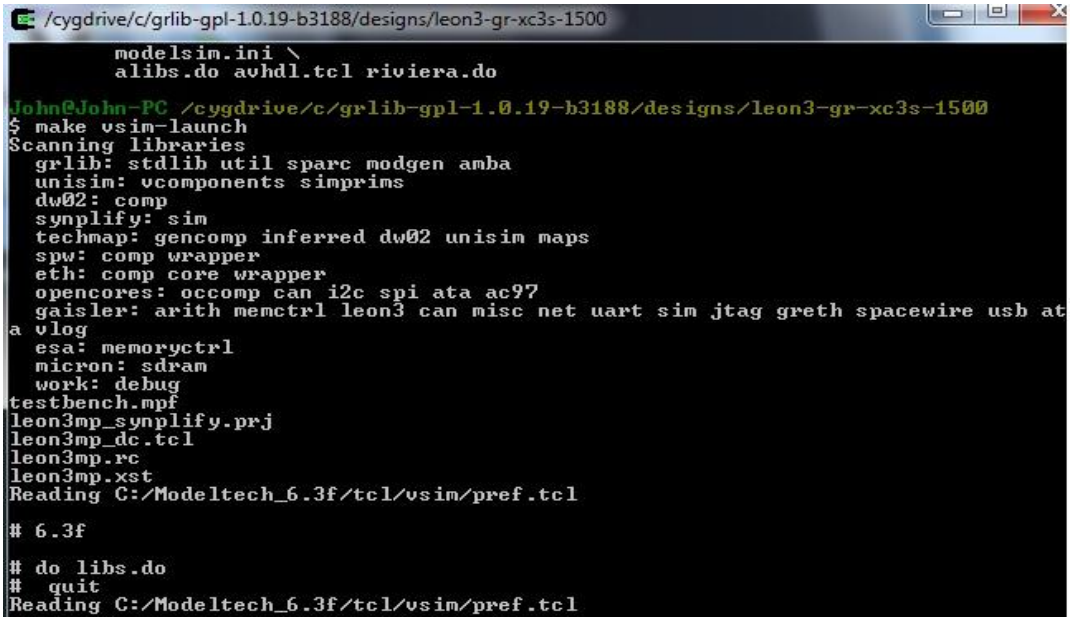
7.2 Προσομοίωση με Modelsim

Η grlib περιέχει scripts για την ευκολότερη χρήση του project Leon. Με αυτά επιταχύνονται πολλές διαδικασίες όπως αυτές της προσομοίωσης και του synthesis με διάφορα εργαλεία. Με τη χρήση Cygwin για χρήστες Windows ή κονσόλας για χρήστες Unix υπάρχουν πολλές επιλογές για τις λειτουργίες που θέλει να κάνει ο χρήστης.

```
John@John-LAPTOP /cygdrive/c/grlib-gpl-1.0.19-b3188/designs/leon3-gr-xc3s-1500
$ MAKE

 interactive targets:

 make avhdl-launch           : start active-hdl gui mode
 make riviera-launch         : start riviera
 make vsim-launch            : start modelsim
 make ncsim-launch           : compile design using ncsim
 make sonata-launch          : compile design using sonata
 make actel-launch-synp      : start Actel Designer for current project
 make ise-launch             : start ISE project navigator for XST project
 make ise-launch-synp        : start ISE project navigator for synplify project
 make quartus-launch         : start Quartus for current project
 make quartus-launch-synp    : start Quartus for synplify project
 make synplify-launch        : start synplify
 make xgrlib                 : start grlib GUI

 batch targets:

 make avhdl             : compile design using active-hdl gui mode
 make vsimsa            : compile design using active-hdl batch mode
 make riviera           : compile design using riviera
 make sonata            : compile design using sonata
 make vsim              : compile design using modelsim
 make ncsim             : compile design using ncsim
 make ghdl              : compile design using GHDL
 make actel             : synthesize with synplify, place&route Actel Designer
 make ise               : synthesize and place&route with Xilinx ISE
 make ise-map           : synthesize design using Xilinx XST
 make ise-prec          : synthesize with precision, place&route with Xilinx ISE
 make ise-synp          : synthesize with synplify, place&route with Xilinx ISE
 make isp-synp          : synthesize with synplify, place&route with ISPLever
 make quartus           : synthesize and place&route using Quartus
 make quartus-map       : synthesize design using Quartus
 make quartus-synp      : synthesize with synplify, place&route with Quartus
 make precision         : synthesize design using precision
 make synplify          : synthesize design using synplify
 make import-actel-cc   : import CoreMP7 files from CoreConsole library
 make scripts           : generate compile scripts only
 make clean             : remove all temporary files except scripts
 make distclean         : remove all temporary files
```

Εικόνα 22: grlib make scripts.

Με την εντολή make vsim-launch, ανοίγει ένα project στο modelsim φορτώνοντας όλα τα απαιτούμενα αρχεία και έχοντας για αρχείο προσομοίωσης το testbench.vhd το οποίο μπαίνει ως top-file. Μέσω αυτού μπορεί να αρχίσει η προσομοίωση.



Εικόνα 23: Εκκίνηση διαδικασίας προσομοίωσης.

Μελετώντας το testbench.vhd μπορεί να δει κανείς ότι φορτώνει στις μνήμες prom και sdram τα αρχεία prom.srec και sdram.srec. Αυτά περιέχουν αντίστοιχα:

-κώδικα assembly sparc όπου γίνεται η αρχικοποίηση του Leon και

-κώδικα assembly sparc όπου περιέχεται το προς εκτέλεση πρόγραμμα.

Έχοντας μελετήσει το testbench, το επόμενο βήμα ήταν η μετατροπή των prom.srec και sdram.srec ώστε πλέον να περιέχουν κώδικα mips και να μπορούν έτσι να διαβαστούν και να αποκωδικοποιηθούν σωστά οι εντολές από τον πυρήνα.

7.3 Srec files

7.3.1 Γενικά

Τα αρχεία srec είναι αρχεία ASCII κωδικοποίησης για δεδομένα στο δυαδικό σύστημα. Βασικό τους πλεονέκτημα έναντι των κλασσικών δυαδικών αρχείων είναι η

ευκολία με την οποία τροποποιούνται μέσω ενός απλού κειμενογράφου. Για την ασφαλή μετατροπή τους και την αναγνώριση πιθανών λαθών στο τέλος κάθε γραμμής υπάρχουν δύο χαρακτήρες που λειτουργούν ως checksum.

7.3.2 Μορφοποίηση srec αρχείων.

Τα αρχεία srec αποτελούνται από ASCII εγγραφές δεκαεξαδικών αριθμών. Όλοι οι δεκαεξαδικοί αριθμοί είναι Big Endian. Οι εγγραφές ακολουθούν την παρακάτω μορφοποίηση:

1. Start code, ένας χαρακτήρας S.
2. Record type, ένα ψηφίο, 0 έως 9, διευκρινίζοντας τον τύπο του τομέα δεδομένων.
3. Byte count, δύο δεκαεξαδικά ψηφία στα οποία παρουσιάζεται ο αριθμός των bytes (ζευγάρια δεκαεξαδικών ψηφίων) που ακολουθούν στο υπόλοιπο record (διεύθυνση, δεδομένα και checksum).
4. Address, τέσσερα, έξι ή οχτώ δεκαεξαδικά ψηφία (ανάλογα με την επιλογή του record type) για την τοποθεσία στη μνήμη του πρώτου byte δεδομένων.
5. Data, μια ακολουθία από 2n δεκαεξαδικά ψηφία, για n bytes δεδομένων.
6. Checksum, δύο δεκαεξαδικά ψηφία που υπολογίζονται ως εξής: αρχικά προσθέτουμε ανά ζεύγη τα δεκαεξαδικά ψηφία των πεδίων 3,4 και 5 του s-record. Από το αποτέλεσμα κρατάμε το συμπλήρωμα ως προς ένα του λιγότερο σημαντικού byte.

Record	Description	Address Bytes	Data Sequence
S0	Block header	2	Yes
S1	Data sequence	2	Yes
S2	Data sequence	3	Yes
S3	Data sequence	4	Yes
S5	Record count	2	No
S7	End of block	4	No
S8	End of block	3	No
S9	End of block	2	No

Πίνακας 5: Record types σε εγγραφές s-rec.

Παρακάτω ακολουθεί ένα παράδειγμα δομής μιας εγγραφής s-rec:

S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026

Start code Record type Byte count Address Data Checksum

7.3.3Διαδικασία δημιουργίας elf και srec αρχείων.

Η διαδικασία δημιουργίας αυτών των αρχείων γίνεται να πραγματοποιηθεί κατευθείαν τόσο από αρχεία γλώσσας υψηλού επιπέδου C και C++ όσο και από αρχεία κώδικα assembly ή εναλλακτικά με άμεση συγγραφή σε γλώσσα μηχανής σε

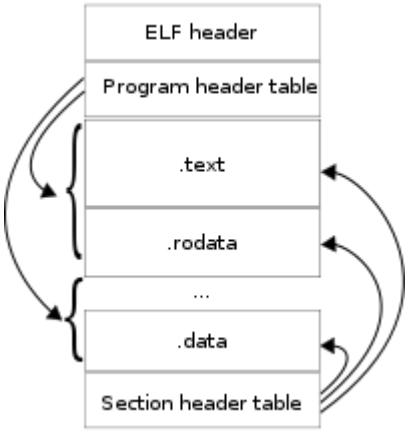
δεκαεξαδική μορφή. Φυσικά, υπάρχουν εργαλεία για την παραγωγή τέτοιων αρχείων, που κυκλοφορούν ελεύθερα. Ένα από αυτά, το οποίο και θα χρησιμοποιήσουμε, είναι ο μεταγλωττιστής mips-elf-gcc.Για καλύτερο έλεγχο της πληροφορίας επιλέξαμε να γίνει η παραγωγή τους σε δύο στάδια ώστε να έχουμε όσο το δυνατόν περισσότερα στοιχεία οπτικοποιημένα για τον παράλληλο έλεγχό τους κατά την προσομοίωση. Από τον κώδικα υψηλού επιπέδου, με την χρήση της εντολής

mips-elf-gcc test.s -o test.elf

παράγεται ένα αρχείο .elf (executable and linkable format). Κάθε ELF αρχείο αποτελείται από ένα ELF header που ακολουθείται από τον τομέα "file data". Στο file data περιέχονται

- Program header table που περιγράφει τα segments που ακολουθούν.
- Section header table που περιγράφει τα sections που ακολουθούν.
- Data όπου φιλοξενείται ο κώδικάς μας μαζί με τα απαιτούμενα headers του.

Τα segments περιέχουν πληροφορία απαραίτητη για την εκτέλεση του αρχείου, ενώ τα sections περιέχουν σημαντική πληροφορία για το linking.



Εικόνα 24: Δομή αρχείων elf.

Αυτό το αρχείο elf μπορεί να διαβαστεί με την εντολή,

mips-elf-objdump -d test.elf

και να αποκομίσουμε πληροφορίες για την κωδικοποίηση των εντολών και την τοποθέτησή τους στις θέσεις μνήμης.

Τέλος με την εντολή

mips-elf-objcopy -O test.elf test.srec

παράγεται το τελικό αρχείο srec με το format που περιγράφηκε παραπάνω και το οποίο περιέχει όλη την απαιτούμενη πληροφορία στην τελική μορφή.

Το αρχείο elf με τον κώδικα assembly που χρησιμοποιούμε παρατίθεται στο παράρτημα Β.1.

7.4 Srec αρχεία για τον MIPS

Στα αρχεία prom.s και sdram.s που περιέχονται στο project του Leon υπάρχει κώδικας assembly γραμμένος για sparc αρχιτεκτονική που αρχικοποιεί τον επεξεργαστή και τις επιμέρους λειτουργικές μονάδες. Ο κώδικας αυτός αρχικά μελετήθηκε και τροποποιήθηκε κατάλληλα ώστε να είναι συμβατός με τον mips core. Ύστερα, με την διαδικασία που περιγράφηκε παραπάνω παρήγαγε τα αρχεία srec που χρησιμοποιήθηκαν τελικά στο mips project.

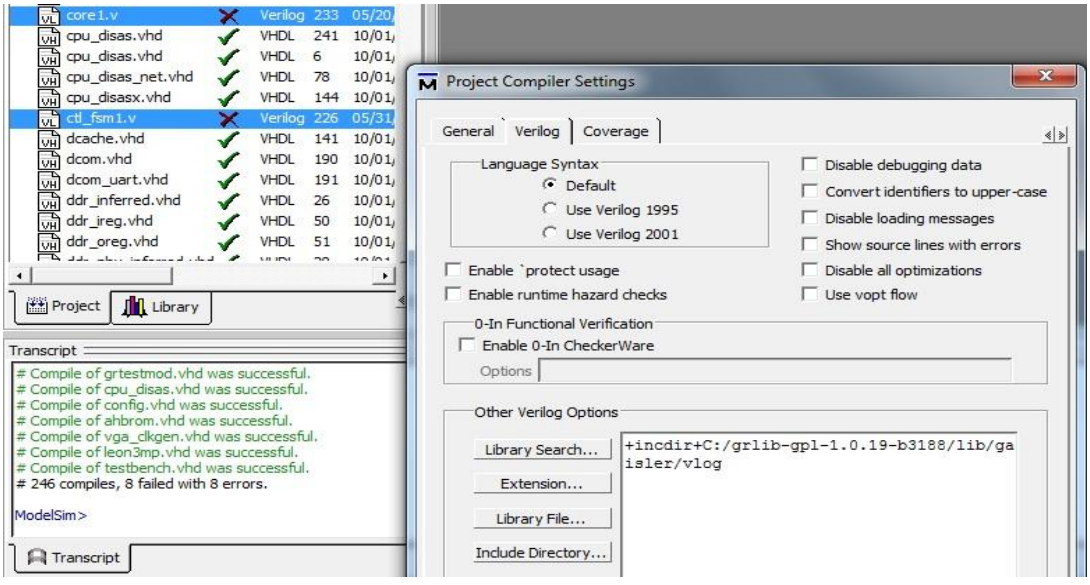
Στο αρχείο prom.s βρίσκεται ο κώδικας αρχικοποίησης του επεξεργαστή και στο sdram.s ένας απλός κώδικας assembly όπου πραγματοποιείται ένας απλός βρόχος. Στον κώδικα του prom.s, που είναι και ο πιο σημαντικός, γίνεται αρχικοποίηση καταχωρητών ελέγχου του συστήματος που βρίσκονται εκτός του πυρήνα. Σε αυτές τις θέσεις γράφονται τιμές που είναι δηλωμένες στο αρχείο prom.h. Στη συνέχεια διαβάζεται μία θέση μνήμης (0xFFFFF860) στην οποία βρίσκονται πληροφορίες σχετικά με το configuration του συστήματος και τέλος εκτελείται άλμα στην αρχή της RAM. Οι λειτουργίες αυτές είναι απαραίτητες για την εκκίνηση του επεξεργαστή και επομένως ο κώδικας prom.s εκτελείται υποχρεωτικά. Οι κώδικες prom.s και sdram.s που χρησιμοποιήθηκαν βρίσκονται στο παράρτημα Β.1 υπό την μορφή .elf files.

Δυστυχώς ο mips-gcc παρήγαγε αρχεία srec που δεν ήταν απολύτως συμβατά με τον συγκεκριμένο πυρήνα. Αφ' ενός η τοποθέτηση πλήθους πληροφοριών για header files και αφ' ετέρου η διαφορετική default θέση μνήμης στην οποία τοποθετούσε τον κώδικα οδηγούσαν στην λανθασμένη ανάγνωση της πληροφορίας. Αφού λοιπόν μελετήθηκε προσεκτικά η χρήση των αρχείων srec από τον Leon3, αφαιρέθηκαν όλες οι εγγραφές που δεν αφορούσαν τον κώδικά μας και κατόπιν μετακινήθηκαν στην κατάλληλη διεύθυνση μνήμης, δηλαδή στην 0x0 για την prom και στην 0x40000000 για την sdram. Τα srec αρχεία βρίσκονται στο παράρτημα Β.2.

7.5 Αποσφαλμάτωση και τελική προσομοίωση

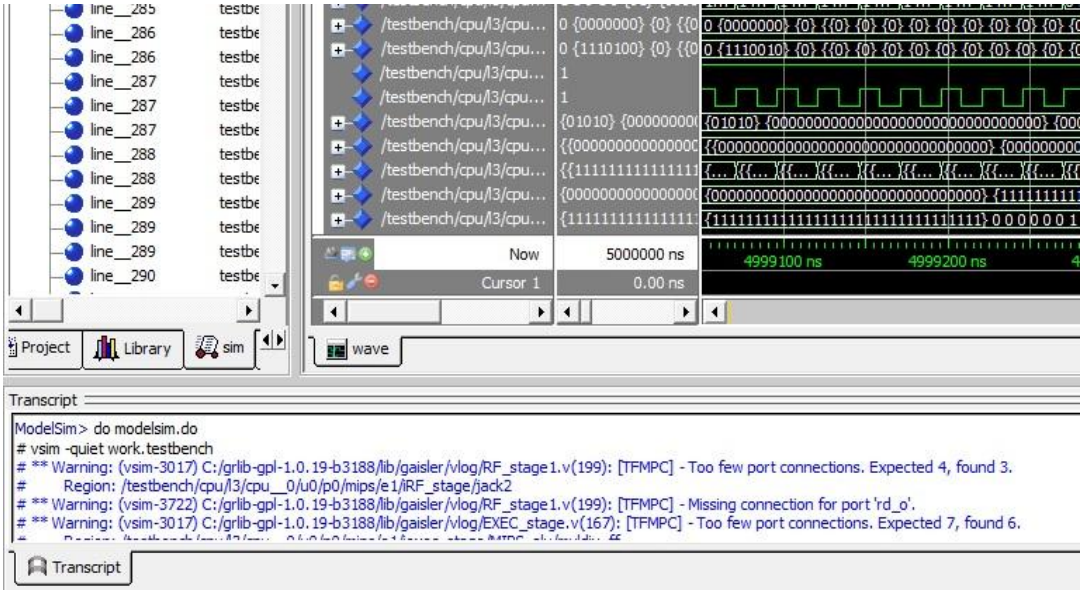
7.5.1 Διαδικασία έναρξης simulation

Μέσω της διαδικασίας που περιγράφθηκε παραπάνω έχουμε πλέον έτοιμο το testbench για τον MIPS. Κάνοντας compile όλα τα αρχεία του project ο κώδικας της verilog δεν περνάει τον έλεγχο γιατί όλα τα αρχεία κάνουν 'include mips_defs.v . Για να γίνει σωστά το compile πρέπει στο compile options sto tab της verilog, για κάθε αρχείο verilog, να κάνουμε include το directory που βρίσκεται το mips_defs.v



Εικόνα 25: Διόρθωση επιλογών compile για αρχεία verilog σε mixed project (VHDL-verilog).

Για τη διευκόλυνση του simulation δημιουργήσαμε ένα .do file (modelsim.do) με το οποίο ξεκινάει η προσομοίωση, καλείται η προβολή wave του modelsim, εισάγονται εκεί τα σήματα που πρέπει να μελετήσουμε για την ορθότητα της λειτουργίας του επεξεργαστή και αρχίζει η εκτέλεση της προσομοίωσης. Το modelsim.do βρίσκεται στο παράρτημα Β.3.



Εικόνα 26: Εκτέλεση testbench με βοηθητικό αρχείο .do .

Τα warnings που παράγονται δεν μας απασχολούν καθώς αφορούν συγκεκριμένα ports τα οποία δεν χρησιμοποιούνται στη σχεδίασή μας. Δεδομένου ότι το simulation γίνεται πλέον κανονικά προχωράμε στην διαδικασία της αποσφαλμάτωσης.

7.5.2 Διαδικασία Αποσφαλμάτωσης.

Σε αυτή την ενότητα θα αναφερθούμε σε σφάλματα που παρατηρήθηκαν κατά τη διάρκεια της προσομοίωσης καθώς και στις διορθωτικές κινήσεις που έγιναν ώστε αυτά να εξαλειφθούν. Πρόκειται για σφάλματα λογικών λαθών που προϋπήρχαν στον πυρήνα mips ή εμφανίστηκαν στις δομικές μονάδες που εμείς προσθέσαμε, σφάλματα χρονισμού κρίσιμων σημάτων και σφάλματα ασυμβατότητας των διαφορετικών γλωσσών περιγραφής υλικού και των εκδόσεών τους. Είναι πρακτικώς αδύνατον να παρουσιαστούν όλα τα βήματα καθώς πολλές φορές κάθε λύση οδηγούσε σε νέα προβλήματα, νέες σκέψεις, νέες αμφιβολίες για την ροή που ακολουθούσε η

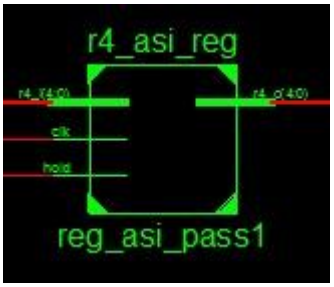
σχεδίαση. Πολλές φορές αναγκαστήκαμε είτε να επιστρέψουμε πίσω στον αρχικό κώδικα είτε να παραβλέψουμε κάποιες δυσλειτουργίες βασισμένοι σε σκέψεις ότι αυτές θα διορθωθούν σε επόμενα στάδια. Αποφασίστηκε λοιπόν να παρουσιάσουμε τις πιο χαρακτηριστικές περιπτώσεις και τις λύσεις αυτών.

7.5.2.1 Σφάλμα στις Branch

Το πρώτο σφάλμα που εντοπίστηκε ήταν η λανθασμένη εκτέλεση της εντολής branch, καθώς η εκτέλεση πήγαινε στην αμέσως επόμενη εντολή και όχι στην εντολή προορισμό. Η παρατήρηση της μεταβολής του pc_next, που παράγεται από το pc_gen module που βρίσκεται στο RF_components.v, έδωσε τη λύση. Το σήμα pc_next δεχόταν την επόμενη τιμή pc από την επιθυμητή. Η διεύθυνση του άλματος που υπολογιζόταν, υπολογιζόταν με βάση την νέα τιμή του pc – το οποίο είχε αλλάξει καθώς το decode είχε προχωρήσει ήδη στην επόμενη εντολή – και όχι σύμφωνα με την τιμή που είχε το pc την στιγμή που στο στάδιο decode γινόταν το branch. Αυτό διορθώθηκε βάζοντας στην τιμή του branch, pc-4 αντί για pc. Ο κώδικας παρατίθεται στο παράρτημα Β.4

7.5.2.2 Σφάλμα στο asi_code

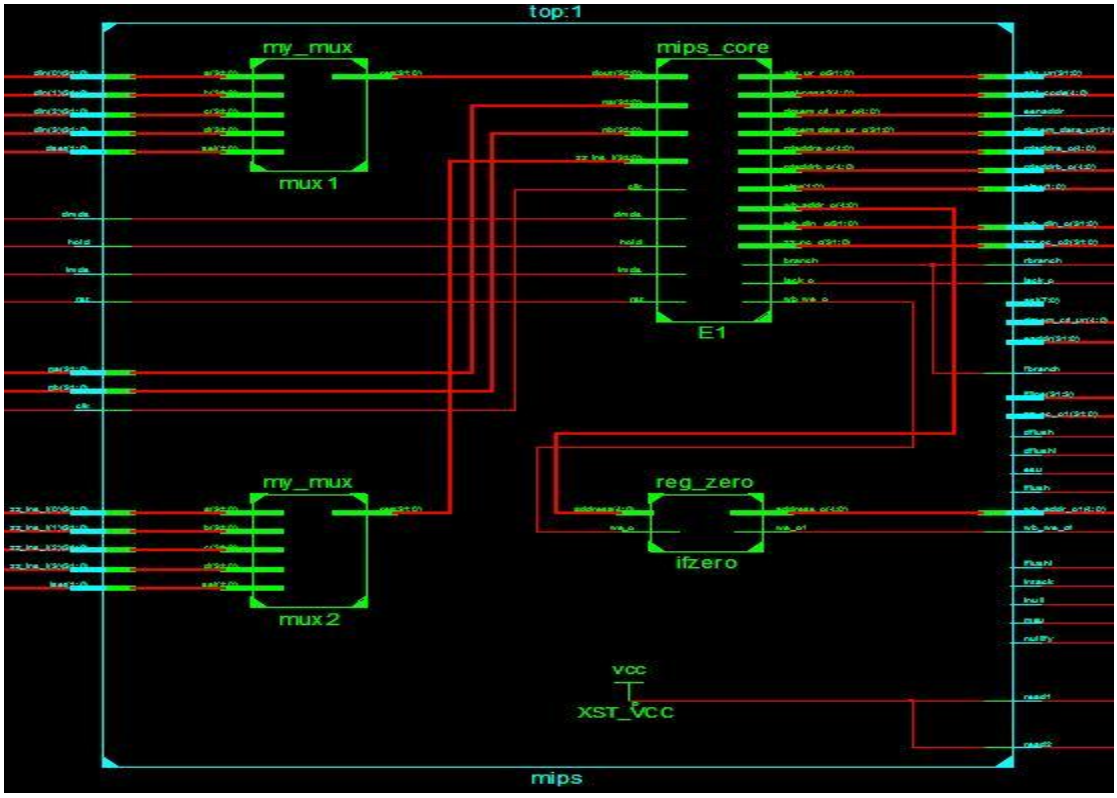
Το επόμενο σφάλμα που παρατηρήθηκε ήταν ότι το asi_code δεν είχε τις τιμές που του είχαμε αναθέσει. Αυτό οφειλόταν κυρίως σε ένα πρόβλημα χρονισμού. Τοποθετήθηκαν δύο registers, επίσης ελεγχόμενοι από το hold, με τους οποίους συγχρονίζεται η άφιξη του asi στις μονάδες κρυφής μνήμης εντολών και δεδομένων.



Εικόνα 27: Καταχωρητής ASI.

7.5.2.3 Μονάδα ελέγχου εγγραφής στον καταχωρητή zero.

Ένα ακόμα πρόβλημα που παρατηρήθηκε ήταν η έλλειψη ελέγχου εγγραφής στον καταχωρητή μηδέν. Σε μία τέτοια περίπτωση ο καταχωρητής μηδέν λάμβανε λανθασμένα τιμή διαφορετική από το μηδέν. Για να διορθωθεί αυτό κατασκευάστηκε η μονάδα reg_zero στην οποία στέλνονται από τον core τα σήματα address και we_o και αποκόπτει την εγγραφή σε περίπτωση που το address αναφέρεται στην καταχωρητή $0. Η πρώτη εντολή που θα φτάσει στην μονάδα reg_zero με σκοπό να γράψει στον $0 θα επιτύχει. Γι αυτό το λόγο πρέπει η πρώτη εντολή που θα επιχειρήσει να γράψει στον $0 να γράψει εκεί την τιμή μηδέν, κάτι που πρέπει να γίνει μέσα από τον κώδικα του prom.srec. Ο κώδικας παρατίθεται στο παράρτημα Β.5



Εικόνα 28: RTL schematic για το top.vhd.

7.5.2.4 Σφάλμα των read/write της data cache

Επίσης παρατηρήθηκε σφάλμα στο dci.read και dci.write της d-cache. Πιο Συγκεκριμένα διαπιστώθηκε ότι οι load ενεργοποιούσαν το write και οι store το read, ενώ προφανώς έπρεπε να συμβαίνει το αντίστροφο. Με αυτόν τον τρόπο οι load

λειτουργούσαν ως store και το ανάποδο. Το πρόβλημα παρουσιαζόταν στον τρόπο δήλωσης και παραγωγής του σήματος dmem_ctl_ur(4:0). Τελικά αντιστρέφοντας τη σειρά των επιμέρους σημάτων, και ειδικότερα αλλάζοντας το dmem_ctl_ur(3) να ενεργοποιεί το read και το dmem_ctl_ur(1) να ενεργοποιεί το write διορθώθηκε και αυτό.

7.5.2.5 Λανθασμένο πάγωμα του πυρήνα

Το προηγούμενο σφάλμα μας βοήθησε να δούμε και ότι το πάγωμα του πυρήνα λειτουργεί μεν αλλά δεν λειτουργεί σωστά. Συγκεκριμένα οι εντολές

sw $t1,...(εδώ από το προηγούμενο σφάλμα λειτουργεί ως lw)

lui $t1,...

δημιουργούσαν λανθασμένα πάγωμα. Αυτό συνέβαινε γιατί στο decode, όπου γίνεται ο έλεγχος παγώματος της ΜΕΔ, αρχικά είχαν χρησιμοποιηθεί bitwise και όχι logical operators της verilog, αλλά κυρίως διότι δεν ελέγχαμε εάν ο καταχωρητής rt διαβάζεται η όχι. Για να λυθεί αυτό το πρόβλημα δημιουργήθηκαν δύο νέα σήματα για κάθε εντολή, τα read_rs και read_rt. Είναι σήματα του decode που ενεργοποιούνται κάθε φορά που μία εντολή διαβάζει τον $rs η τον $rt. Έπειτα άλλαξε ο έλεγχος του παγώματος ώστε να συμπεριλάβει και αυτά τα σήματα. Ο τελικός κώδικας του decode παρατίθεται στο παράρτημα Β.6

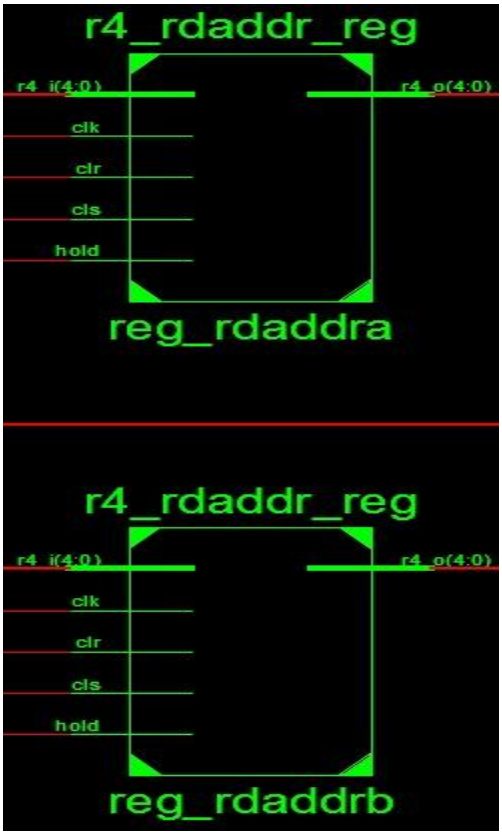
7.5.2.6 Πρόβλημα με την εντολή jump.

Στην εντολή jump παρουσιάστηκε πρόβλημα κατά την διάρκεια του άλματος όπου και χανόταν ο pc προορισμού. Παρατηρώντας ξανά το pc_gen module διαπιστώθηκε πως η κατάσταση του FMS άλλαζε συνεχώς δημιουργώντας πρόβλημα στην παραγωγή του pc. Αρχικά η τοποθέτηση ενός επιπλέον καταχωρητή ο οποίος θα κρατάει το σήμα ελέγχου της FSM φάνηκε να δίνει λύση στο πρόβλημα. Όχι όμως και την βέλτιστη. Μελετώντας τον κώδικα της FSM που βρίσκεται στο ctl_fsm.v διαπιστώθηκε η ύπαρξη ενός καταχωρητή που δεν ελέγχεται, ως όφειλε, από το σήμα hold. Προσθέτοντας το hold στον καταχωρητή αυτόν και αφαιρώντας αυτόν που

είχαμε ήδη προσθέσει, η εντολή jump λειτούργησε ομαλά.

7.5.2.7 Πρόβλημα με το zz_ins_o.

Το σήμα zz_ins_o στο οποίο βρίσκεται η εκάστοτε εντολή παρέχει κάποια πεδία χωρίς ενδιάμεση αποθήκευση. Αυτό δημιουργούσε πρόβλημα γιατί κάποιες φορές άλλαζε πρόωρα και έτσι γίνονται λάθος λειτουργίες, δημιουργώντας πρόβλημα για τους αριθμούς καταχωρητών ανάγνωσης. Διορθώθηκε τοποθετώντας καταχωρητές στα rd_addr_a και rd_addr_b που γράφονται στην πτώση του ρολογιού, ώστε οι αριθμοί καταχωρητών ανάγνωσης να φτάνουν στο φάκελο καταχωρητών στο σωστό χρόνο. Ο σχετικός κώδικας βρίσκεται στο παράρτημα Β.7



Εικόνα 29: Καταχωρητές address_a και address_b.

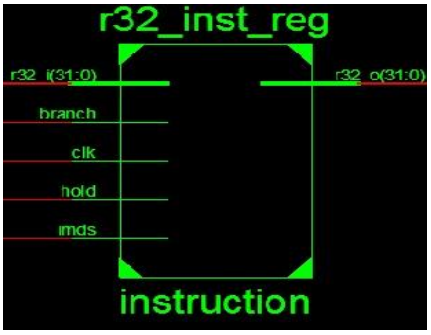
7.5.2.8 Διαθεσιμότητα τιμών στις μνήμες.

Η λογική που χρησιμοποιείται είναι ότι οι μνήμες πρέπει να έχουν όλες τις τιμές διαθέσιμες πριν την αρχή του κύκλου στον οποίο διαβάζονται ή γράφονται. Αυτό

ισχύει τόσο για τις cache όσο και για το reg_file για αυτό έγινε και η αλλαγή που περιγράφθηκε παραπάνω. Για να διορθωθεί το πρόβλημα αυτό στις cache προσθέσαμε ως έξοδο το pc_next το οποίο συνδέθηκε με το ici.rpc. Το zz_ins_o παρέμεινε συνδεδεμένο στο ici.fpc.

7.5.2.9 Ακύρωση εντολών μετά από branch.

Ένα ακόμα σφάλμα που παρατηρήθηκε είναι ότι η δεύτερη εντολή που ερχόταν μετά το branch εκτελούταν κανονικά ενώ θα έπρεπε να ακυρώνεται. Για να πραγματοποιήσουμε την ακύρωση αυτής της εντολής, συνδέσαμε το σήμα branch στον register που καταχωρούμε την εντολή(ins_reg) και όταν ενεργοποιείται το σήμα branch ο καταχωρητής δίνει στην έξοδο την τιμή 32'b0.



Εικόνα 30: Τελική μορφή instruction register (προσθήκη σήματος branch).

7.5.2.10 Πρόβλημα ανάγνωσης της sdram

Τέλος παρατηρήθηκε ένα πρόβλημα στην ανάγνωση της μνήμης sdram. Συγκεκριμένα, αν και όλα λειτουργούν σωστά κατά τη διάρκεια εκτέλεσης του κώδικα που βρίσκεται στην prom, μετά την τελευταία εντολή jump δεν φαίνεται να εκτελείται ο κώδικας της sdram παρ' όλο που ο pc τοποθετείται σωστά. Ενδεχομένως το πρόβλημα αυτό να οφείλεται σε λανθασμένο configuration του συστήματος μνήμης σε επίπεδο AMBA bus, κάτι που όμως ξεφεύγει από τα όρια μελέτης αυτής της διπλωματικής εργασίας. Για το λόγο αυτό και για να εκτελέσουμε τελικά τον κώδικα που επιθυμούσαμε, τα αρχεία των prom και sdram μετατράπηκαν και συγχωνεύτηκαν σε ένα αφού πρώτα αφαιρέθηκε η τελική jump. Με τον τρόπο αυτό μπορούμε τελικά να δούμε μέσα από την prom την εκτέλεση του loop που ήταν γραμμένο στην sdram.

7.5.3 Τελική προσομοίωση

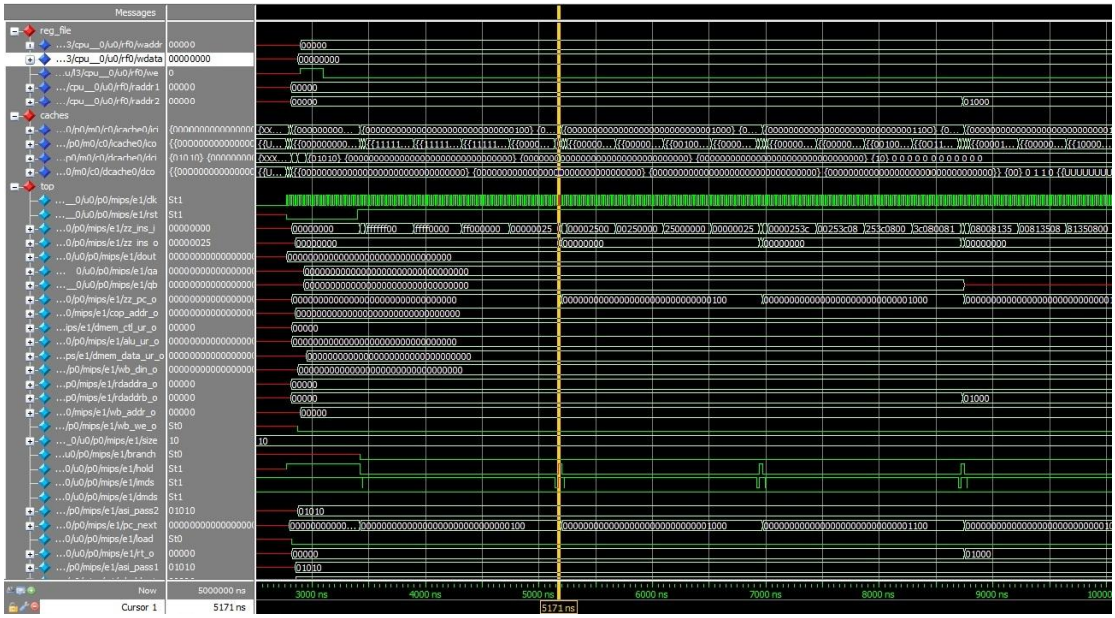
Στην παράγραφο αυτή θα δώσουμε μερικά χαρακτηριστικά παραδείγματα προσομοίωσης εντολών που ενεργοποιούν συγκεκριμένες λειτουργίες του επεξεργαστή. Αρχικά θα παρουσιαστεί ένα γενικό στιγμιότυπο τρεξίματος του πυρήνα και οι τιμές των βασικών σημάτων. Εν συνεχεία, θα εμβαθύνουμε στην εκτέλεση μίας λογικής (ori) εντολής και της πρόσβασής της στον φάκελο καταχωρητών, μίας εντολής φόρτωσης (lw) και μίας εντολής διακλάδωσης με έλεγχο συνθήκης (bne). Μέσα από αυτές τις τρεις εντολές θα περιγραφεί η λειτουργικότητα τόσο του πυρήνα, όσο και των κρυφών μνημών και του φακέλου καταχωρητών.

7.5.3.1 Γενικό στιγμιότυπο εκτέλεσης.

Στην εικόνα 30 παρουσιάζονται μερικά από τα βασικότερα σήματα, κατά τη διάρκεια εκκίνησης του επεξεργαστή και εκτέλεσης των τεσσάρων πρώτων εντολών του κώδικα prom.srec που βρίσκεται στο παράρτημα Β.2. Στο συγκεκριμένο στιγμιότυπο φαίνονται οι αρχικές τιμές των σημάτων και οι εναλλαγές τους. Πιο συγκεκριμένα, το clk έχει περίοδο 25ns(δηλαδή συχνότητα 40MHz) και φαίνεται η ορθή αλλαγή των σημάτων που αφορούν τον pc (pc_next, zz_pc_o), οι διαδοχικές αλλαγές της τιμής των σημάτων που αφορούν την εκάστοτε εντολή (zz_ins_i, zz_ins_o), τα σήματα πρόσβασης στον φάκελο καταχωρητών. Τα περισσότερα από αυτά τα σήματα θα αναλυθούν λεπτομερώς στις επόμενες παραγράφους όπου φαίνεται η λειτουργικότητάς τους ανά εντολή.

Αξίζει να αναφερθούμε στην περίπτωση του σήματος hold. Εδώ φαίνεται ξεκάθαρα ότι το πέρασμα των εντολών και η αλλαγή του pc γίνονται κατά τη διάρκεια στην οποία το hold έχει την τιμή ‘1’. Όταν αυτό έχει τιμή ‘0’, δε συμβαίνει τίποτα μέσα στον πυρήνα του επεξεργαστή!

Επίσης, παρατηρούμε την αρχική εγγραφή της τιμής ‘0’ στον καταχωρητή zero όπως έχει περιγραφεί στην παράγραφο 7.5.2.3



Εικόνα 31: Γενικό στιγμιότυπο εκτέλεσης

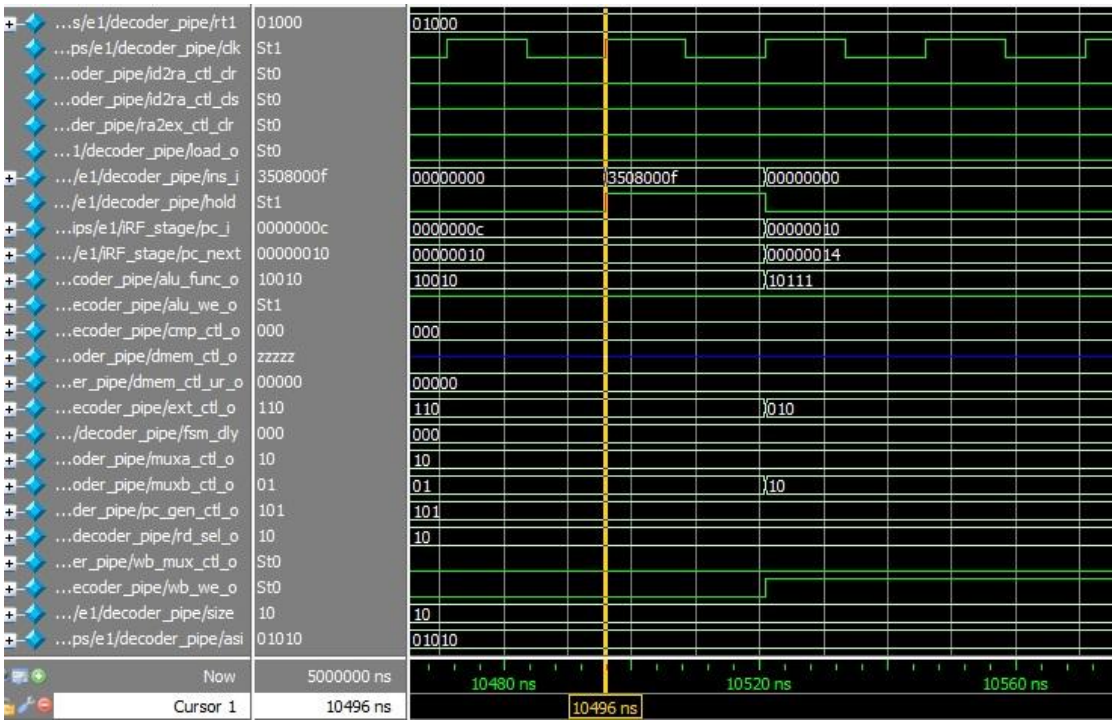
7.5.3.2 Στιγμιότυπο εκτέλεσης λογικής εντολής.

Στην παράγραφο αυτή θα ακολουθήσουμε την διαδικασία εκτέλεσης της εντολής

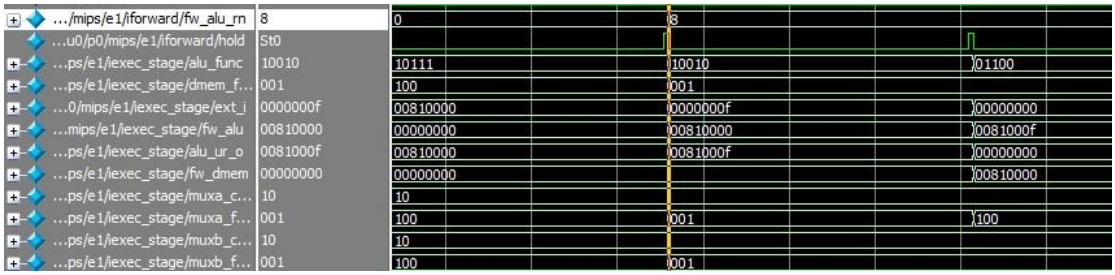
ori $t0, $t0, 0xf

που είναι κωδικοποιημένη στο δεκαεξαδικό 0x3508000f.

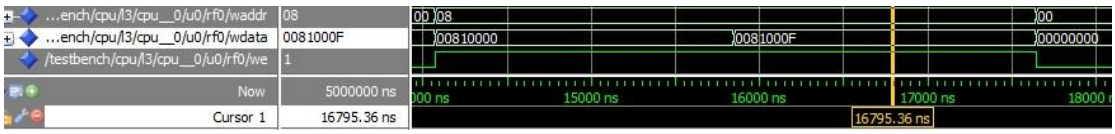
Η εντολή φτάνει στην άνοδο του hold μέσω του σήματος ins_i στο στάδιο αποκωδικοποίησης. Στην πτώση του hold παράγονται τα σχετικά σήματα (εικόνα 31). Πιο συγκεκριμένα, όπως φαίνεται στην εικόνα 33, η εντολή μετά την προσπέλαση του φακέλου καταχωρητών έχει εγγράψει επιτυχώς στον καταχωρητή 8 ($t0) – με τη χρήση προώθησης – την τιμή 0x0081000F μετά από τρεις φάσεις επικάλυψης, δεδομένου ότι η προηγούμενη εντολή lui παράγει την τιμή 0x81,. Η προώθηση φαίνεται στην εικόνα 32 όπου, μέσω του fw_alu και του ext_i, που είναι το άμεσο τελούμενο, το αποτέλεσμα της alu (alu_ur_o) είναι το σωστό αποτέλεσμα και είναι αυτό που γράφεται στον καταχωρητή 8.



Εικόνα 32: Άφιξη της εντολής ori $t0, $t0, 0xf στο στάδιο αποκωδικοποίησης και εκτέλεση αυτού.



Εικόνα 33:Προώθηση τιμής του καταχωρητή 8 για την πράξη ori $8,$8,0xf



Εικόνα 34: Προσπέλαση φακέλου καταχωρητών για εγγραφή από την εντολή ori $t0, $t0, 0xf.

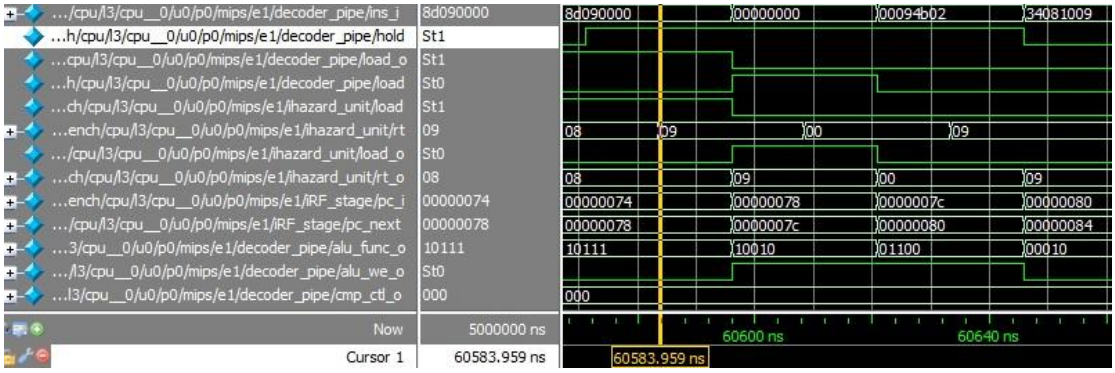
7.5.3.3 Στιγμιότυπο εκτέλεσης εντολής φόρτωσης.

Αφού παρουσιάστηκε η αποκωδικοποίηση μίας εντολής και η πρόσβαση αυτής στον φάκελο καταχωρητών, περνάμε στην επόμενη περίπτωση που είναι αυτή της εντολής φόρτωσης

lw $t1, 0($t0)

που φτάνει στον πυρήνα κωδικοποιημένη στο δεκαεξαδικό 0x8d090000.

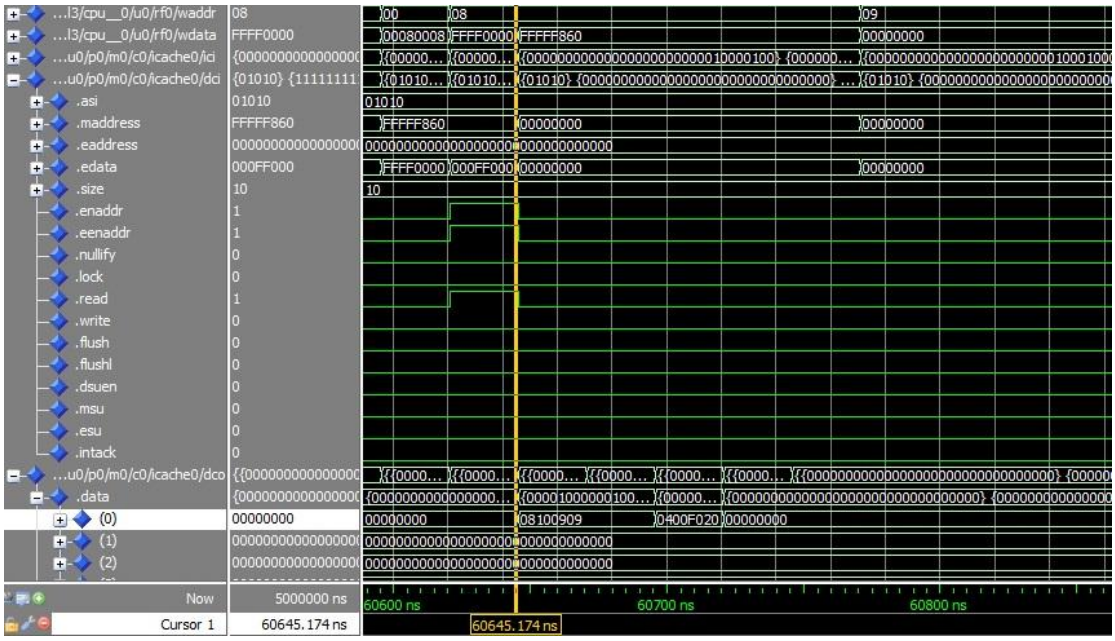
Εδώ θα μας απασχολήσουν διαφορετικά σήματα όπως το σήμα load_o που από το στάδιο αποκωδικοποίησης περνάει μέσα από την βοηθητική μονάδα ελέγχου hazard unit και το σήμα rt_o που μεταφέρει την διεύθυνση του καταχωρητή στην οποία θα γίνει η φόρτωση. Στην εικόνα 34 βλέπουμε το σήμα load_o να έχει ήδη πάρει την τιμή ‘1’ από το στάδιο της αποκωδικοποίησης και μέσω του hazard unit να μεταφέρεται στην επόμενη φάση της επικάλυψης καθώς επίσης και η τιμή του καταχωρητή rt (εδώ $t1, δηλαδή 9) . Κατά την άφιξη της επόμενης εντολής (0x00000000) γίνεται ο έλεγχος για ανάγνωση μετά από εγγραφή και εφ’ όσον δεν βρισκόμαστε σε τέτοια περίπτωση (η εντολή που ακολουθεί είναι nop), δεν γίνεται πάγωμα του επεξεργαστή.



Εικόνα 35: Άφιξη της εντολής lw $t1, 0($t0) στο στάδιο αποκωδικοποίησης και στο Hazard Unit.

Συνεχίζοντας, στην εικόνα 35, παρατηρούμε την ενεργοποίηση των σημάτων enaddr, eenaddr και read που επιτρέπουν την ανάγνωση του περιεχομένου της διεύθυνσης μνήμης που υπολογίζεται από την πρόσθεση του offset με το περιεχόμενο του καταχωρητή rs. Η τιμή που γράφεται στον καταχωρητή 9 όπως φαίνεται είναι 0. Να

σημειωθεί πως σε αντίστοιχη περίπτωση εντολής αποθήκευσης, όπως η sw, στην κρυφή μνήμη δεδομένων αντί του σήματος read ενεργοποιείται το σήμα write.



Εικόνα 36: Ανάγνωση τιμής από την κρυφή μνήμη δεδομένων.

7.5.3.4 Στιγμιότυπο εκτέλεσης εντολής διακλάδωσης.

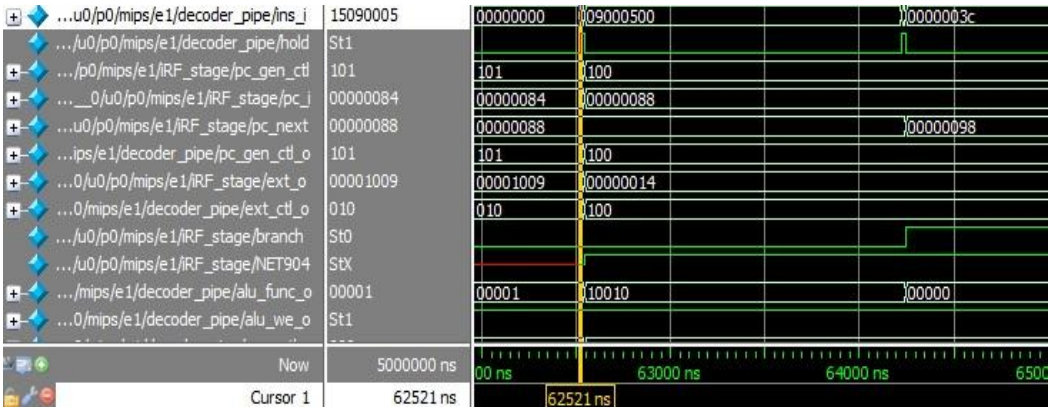
Παρακάτω θα εξετάσουμε την εκτέλεση μίας εντολής διακλάδωσης και συγκεκριμένα της

bne $t0,$t1, <L>

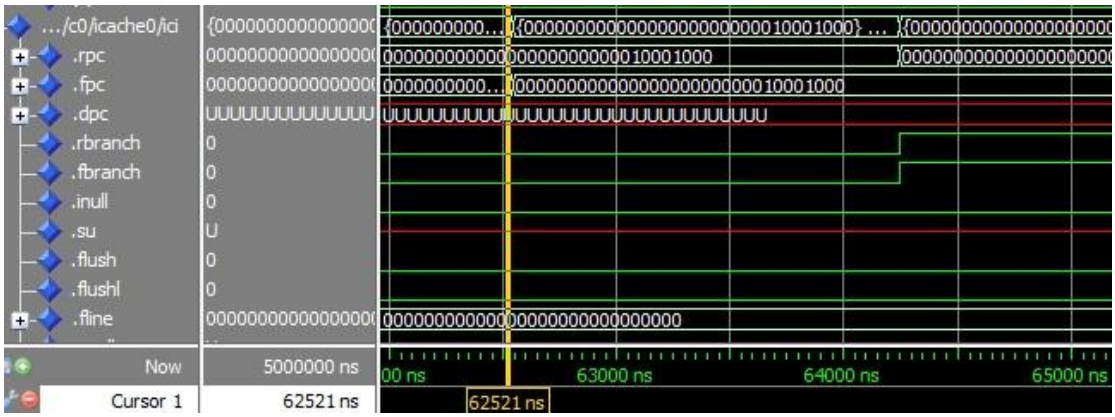
που φτάνει κωδικοποιημένη στο δεκαεξαδικό 0x15090005.

Η εντολή φτάνει μέσω του zz_ins_i, αποκωδικοποιείται και μετά γίνεται ο έλεγχος της συνθήκης άλματος. Το αποτέλεσμα αυτού, μεταφέρεται μέσω του εσωτερικού σήματος NET904 που αποκτά την τιμή ‘1’, αφού η συνθήκη είναι αληθής. Παράλληλα, μέσω του σήματος ext_o μεταφέρεται η τιμή που θα πρέπει να προστεθεί στον pc ώστε να υπολογιστεί η διεύθυνση στην οποία θα μεταφερθεί ο έλεγχος. Όπως φαίνεται στην εικόνα 36, στο στάδιο ανάγνωσης του φακέλου καταχωρητών έχει ενεργοποιηθεί το σήμα branch το οποίο ενεργοποιεί με τη σειρά του τα σήματα fbranh/rbranch της κρυφής μνήμης εντολών ώστε να έρθει η κατάλληλη εντολή, όπως φαίνεται στην εικόνα 37. Επίσης παρατηρούμε την

μετατροπή του σήματος pc_next. Αντί να αυξηθεί κατά 0x4, μεταβαίνει στην διεύθυνση μνήμης που βρίσκεται η ετικέτα <L> στον κώδικα prom.s.



Εικόνα 37: Εκτέλεση της εντολής bne $t0, $t1, <L>



Εικόνα 38: Ενεργοποίηση των σημάτων fbranch και rbranch στην κρυφή μνήμη εντολών.

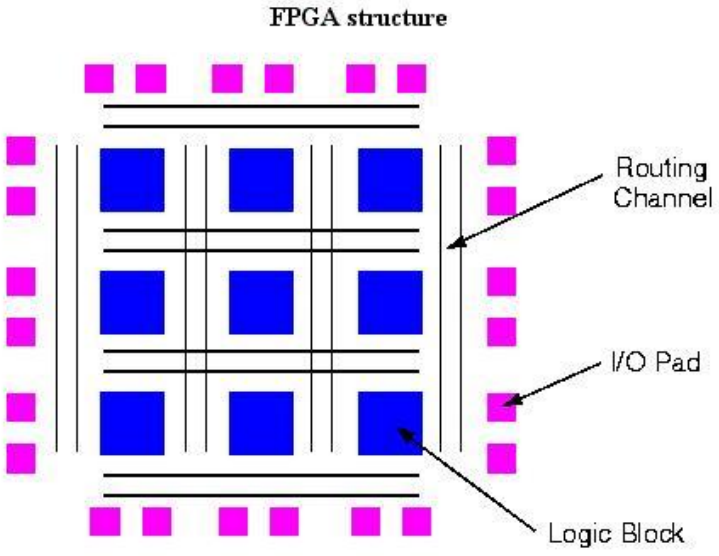
8. Διάταξη Field-programmable Gate Array

8.1 Γενικά

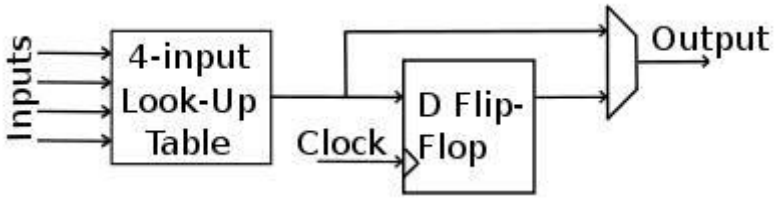
Στο παρόν κεφάλαιο θα ασχοληθούμε με την ροή πληροφορίας για την σχεδίαση συστημάτων που υλοποιούνται σε επαναπρογραμματιζόμενες διατάξεις FPGA. Θα περιγράψουμε την δομή των FPGA, το board που θα χρησιμοποιηθεί, καθώς και τα εργαλεία που χρησιμοποιήθηκαν κατά την διαδικασία σύνθεσης.

8.2 Τι είναι η FPGA

Η διάταξη FPGA είναι ένα ολοκληρωμένο κύκλωμα σχεδιασμένο με τέτοιο τρόπο ώστε να μπορεί να διαμορφωθεί κατά περίσταση ανάλογα με την επιθυμητή χρήση. Αποτελείται από logic blocks καθένα από τα οποία περιέχουν ένα μικρό Look Up Table (LUT), ένα D-FlipFlop και έναν πολυπλέκτη 2-σε-1.



Εικόνα 39: Δομή και οργάνωση μιας FPGA.



Εικόνα 40: LUT block.

Μία FPGA αποτελείται από μερικές εκατοντάδες έως μερικά εκατομμύρια LUTs. Αυτά λειτουργούν σαν μικρές ROM τεσσάρων εισόδων που μπορούν να υλοποιήσουν οποιαδήποτε λογική συνάρτηση τεσσάρων αγνώστων. Για παράδειγμα μία πύλη AND τριών εισόδων που οδηγεί μία πύλη AND δύο εισόδων μπορεί να υλοποιηθεί εντός ενός LUT. Μεταξύ τους, τα LUTs συνδέονται με καλώδια ή πολυπλέκτες. Η σύνδεση με τα I/O blocks και τα pins της FPGA γίνεται επίσης με καλώδια. Τα pins αυτά είναι σχεδιασμένα για να επιτελούν μία συγκεκριμένη δουλειά είτε αυτή είναι να ρευματοδοτούν τα LUTs είτε να συνδέουν τα LUTs με τον εξωτερικό κόσμο μέσω I/O blocks. Χωρίζονται σε dedicated και user Pins. Τα πρώτα αφορούν hard-coded συναρτήσεις, όπως pins ρεύματος, γείωσης, configuration. Τα user pins χρησιμοποιούνται από τον χρήστη ως σήματα input, output ή bidirectional. Επειδή τα flip-flops συνοδεύουν τα LUTs, δεν είναι αποδοτικό να χρησιμοποιούνται για υλοποίηση διατάξεων μνήμης. Για το λόγο αυτό, οι περισσότερες σημερινές FPGAs εμπεριέχουν ανεξάρτητες διατάξεις μνήμης που μπορούν να χρησιμοποιηθούν ως κρυφές μνήμες, φάκελοι καταχωρητών, κλπ

Οι FPGAs είναι συνήθως σύγχρονες. Η σχεδίαση βασίζεται στο ρολόι και κάθε flip-flop παίρνει μια καινούρια κατάσταση στην ακμή του ρολογιού. Το ίδιο ρολόι μπορεί να οδηγεί αρκετά flip-flops που ενδέχεται να απέχουν πολύ μεταξύ τους. Αυτό δημιουργεί προβλήματα συγχρονισμού λόγω καθυστέρησης του σήματος του ρολογιού μέσω των καλωδίων (clock skew). Για να αποφευχθεί το clock skew χρησιμοποιούνται ειδικά εσωτερικά καλώδια που επιτρέπουν τη διάδοση του σήματος του ρολογιού με μικρότερο skew. Επιπλέον χρησιμοποιούνται πολλαπλά ρολόγια, ένα για κάθε περιοχή, πράγμα που αυξάνει την δυσκολία συγχρονισμού των επιμέρους περιοχών.

Συμπερασματικά οι FPGAs μπορούμε να πούμε ότι είναι διατάξεις που μπορούν να συμπεριφερθούν ως συναρτήσεις – κυκλώματα ορισμένα από τον χρήστη κατά περίπτωση. Η σχεδίαση του κυκλώματος φορτώνεται στην FPGA κατά βούληση και αποβάλλεται κάθε φορά που αποσυνδέουμε την FPGA από την παροχή ρεύματος. Σε πολλά boards όπως σε αυτό που θα χρησιμοποιήσουμε, υπάρχουν ειδικές μονάδες prom στις οποίες είναι δυνατόν να αποθηκευτούν αρχεία σχεδίασης και να φορτώνονται κατ' επιλογή του χρήστη κάθε φορά που ανάβει η FPGA.

8.3 Παραγωγή bitstream για FPGA

Συνοπτικά η διαδικασία έχει ως εξής: αρχικά σχεδιάζεται το κύκλωμα σε κάποια γλώσσα περιγραφής υλικού. Εν συνεχεία, γίνεται η σύνθεση και το place and route. Τέλος παράγεται ένα bitstream αρχείο το οποίο φορτώνεται στην FPGA. Στο στάδιο της σχεδίασης του κυκλώματος, αφού φυσικά περάσουμε επιτυχώς τον συντακτικό έλεγχο, γίνεται επιβεβαίωση της σωστής λειτουργίας του κυκλώματος με την χρήση testbenches.

Κατά τη σύνθεση του κυκλώματος η περιγραφή του κυκλώματος μετατρέπεται σε λογικές πύλες. Εδώ πρέπει να δοθεί προσοχή για την αποφυγή δομικών λαθών. Τέτοια λάθη εμφανίζονται συνήθως λόγω της παρεχόμενης δυνατότητας από τις γλώσσες υψηλού επιπέδου να κατασκευάζονται κομμάτια κώδικα που στην πραγματικότητα δεν έχουν φυσική υπόσταση σε επίπεδο υλικού, όπως λογικά loops. Ακολουθεί το translation που είναι η διαδικασία μετατροπής του αρχείου netlist σε αντίστοιχο συμβατό με το board που θα χρησιμοποιήσουμε. Στην διαδικασία αυτή λαμβάνεται υπ' όψιν και το αρχείο User Constraints File (ucf) στο οποίο περιγράφονται οι χωρικοί και χρονικοί περιορισμοί που θέτει ο χρήστης ανάλογα με την FPGA και την σχεδίαση που έχει.

Στο επόμενο βήμα λαμβάνει χώρα το mapping κατά το οποίο γίνεται δέσμευση πόρων στην FPGA ανάλογα με τα στοιχεία λογικής που χρησιμοποιούνται. Παράλληλα χρησιμοποιείται και εδώ το αρχείο ucf. Εδώ γίνονται και οι όποιες βελτιστοποιήσεις που απαιτούνται για την συμβατότητα του project με την εκάστοτε συσκευή.

Στη συνέχεια περνάμε στο στάδιο του Place and Route. Σε αυτό τα στοιχεία του netlist απεικονίζονται στις φυσικές θέσεις της FPGA για να δημιουργηθεί η τελική σχεδίαση που θα μπορεί να υλοποιηθεί σε LUTs της FPGA. Αυτό αφορά κυρίως τις θέσεις των blocks που θα χρησιμοποιηθούν ώστε να έχουμε την βέλτιστη δρομολόγηση με την ελάχιστη καθυστέρηση.

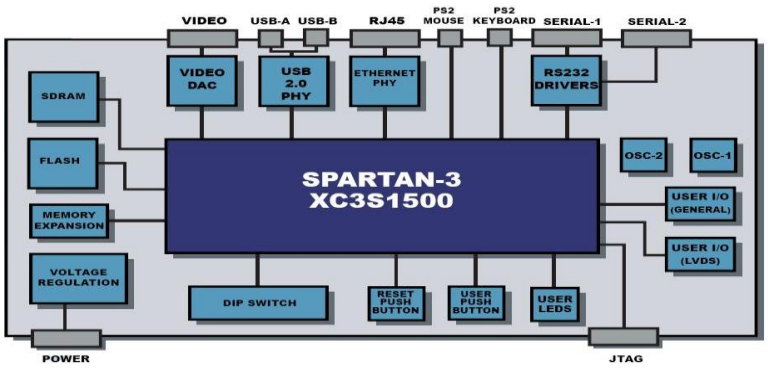
Τέλος, προκύπτει το αρχείο bitstream. Το αρχείο αυτό περιέχει την τελική σχεδίαση (κώδικας του χρήστη και βελτιστοποιήσεις) σε μορφή δυαδικού κώδικα και φορτώνεται στην FPGA.

8.4 Πλατφόρμα υλοποίησης GR-XC3s1500

Αφού περιγράψαμε την ροή της σχεδίασης για κυκλώματα FPGA θα δώσουμε και μία σύντομη περιγραφή της πλατφόρμας που θα χρησιμοποιήσουμε.

Το board που επιλέχτηκε είναι το gr-xc-3s1500 της Pender που υποστηρίζει πλήρως τον μικροεπεξεργαστή Leon 3 της Gaisler Research πάνω στον οποίο βασίζεται το project. Το board ενσωματώνει την FPGA xc3s της οικογένειας Spartan3 της Xilinx. Αποτελείται από 1.5 εκατομμύρια πύλες. Παρέχεται ακόμα ενσωματωμένη μνήμη Flash και μνήμη SDRAM, ethernet, JTAG, σειριακές θύρες USB και PS2 διεπαφές για ποντίκι και πληκτρολόγιο.

Η μνήμη Flash Prom παρέχει την δυνατότητα αποθήκευσης αρχείων bitstream ώστε να μην χρειάζεται να φορτώνεται εκ νέου το configuration σε κάθε άνοιγμα του board.



Εικόνα 41: Πλατφόρμα υλοποίησης GR-XC3s1500

9. Διαδικασία Synthesis.

9.1 Γενικά

Στο παρόν κεφάλαιο θα αναλύσουμε την διαδικασία Synthesis με το εργαλείο Xilinx από το στάδιο του configuration του αρχικού Leon μέχρι το τελικό πέρασμά του στην FPGA. Χρησιμοποιήθηκαν διάφορα εργαλεία που περιέχονται στο Xilinx WebPack όπως τα ISE, XST, IMPACT, fpga-editor καθώς και τα αντίστοιχα εργαλεία του Leon που εμπεριέχονται στην grlib.

9.2 Περιγραφή Leon configuration

Αρχικά, παρουσιάζονται οι αλλαγές που έχουν γίνει στο configuration του επεξεργαστή Leon. Η παραμετροποίηση του Leon μπορεί να γίνει είτε γραφικά με το εργαλείο make xconfig είτε αλλάζοντας απευθείας το αρχείο config.vhd που βρίσκεται ανάμεσα στα αρχεία του design που χρησιμοποιούμε. Το αρχείο config.vhd είναι διαθέσιμο στο παράρτημα B8.

9.3 Xilinx XST και Xilinx ISE

Όπως προαναφέρθηκε η grlib παρέχεται μαζί με αρκετά scripts που διευκολύνουν πολύ την όλη διαδικασία. Πρόκειται για TCL/TK scripts που εκτελούνται μέσω Unix κονσόλας. Για χρήστες Unix αρκεί μετά την εγκατάσταση των Xilinx tools να δηλωθεί το PATH στο οποίο περιέχονται τα εκτελέσιμα του Xilinx. Για χρήστες Windows χρειάζεται ο προσομοιωτής Cygwin και η τοποθέτηση της grlib απευθείας στον δίσκο c:\.

Έπειτα με την εντολή make xst ο χρήστης μπορεί να ξεκινήσει την διαδικασία Synthesis (Check syntax, generate post synthesis simulation model). Με τις εντολές make ise και make ise-launch η εκτέλεση μετά το στάδιο του Synthesis ακολουθείται από το Implement (Translate, Map, Place & route) και Generate programming file

στάδιο δίνοντας μας το τελικό bitstream που φορτώνεται στο board. Οι εντολές make xst και make ise τρέχουν σε περιβάλλον κονσόλας όπου και τυπώνουν τα αποτελέσματά τους. Αντίθετα, με την εντολή make ise-launch ξεκινάει το Xilinx σε γραφικό περιβάλλον, όπου εκτός από τα αποτελέσματα μπορούμε να δούμε πολλές ακόμα χρήσιμες πληροφορίες, όπως αναλυτικά τα RTL σχέδια του εκάστοτε project. Αξίζει να σημειωθεί πως η εκτέλεση του Xilinx σε γραφικό περιβάλλον προϋποθέτει κάποιες αλλαγές στον τρόπο εκτέλεσης του συγκεκριμένου project, όπως την χειροκίνητη πρόσθεση της βιβλιοθήκης στην οποία περιέχονται τα αρχεία verilog (./lib/gaisler/vlog) στην βιβλιοθήκη work.

Παρ' όλα αυτά χρησιμοποιώντας απευθείας το γραφικό περιβάλλον του Xilinx είναι αδύνατο να παραχθεί το bitstream αρχείο για λόγους που δεν έγιναν κατανοητοί. Σίγουρα όμως, αυτό δεν είναι πρόβλημα του project αλλά πιθανότατα κάποια διαφορά στην παραμετροποίηση των δύο τρόπων εκτέλεσης του Xilinx. Το bitstream αρχείο παράγεται κανονικά αν ακολουθήσουμε την εκτέλεση σε περιβάλλον κονσόλας με την εντολή make ise.

Τέλος χρειάζεται προσοχή κατά την επανεκτέλεση της διαδικασίας, αφού πριν από αυτή πρέπει να διαγράφεται το παλιό project file (leon3.xise) και να καθαρίζεται όλος ο φάκελος του project με την εντολή make distclean.

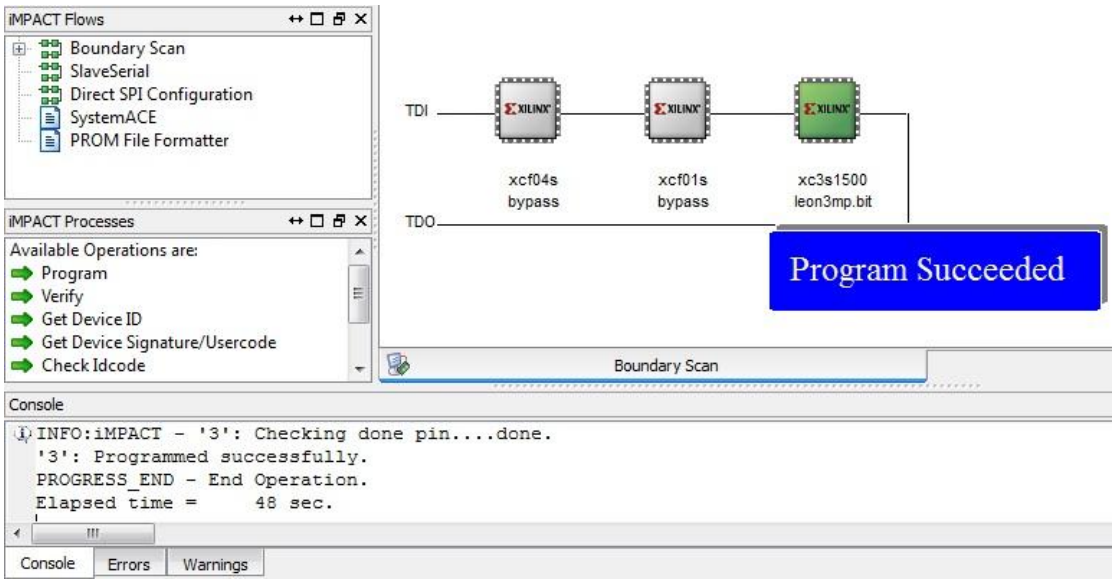
9.4 Impact

Ο τελικός προγραμματισμός του board θα γίνει με το εργαλείο IMPACT. Όπως και με τα εργαλεία xst και ise η εργασία αυτή μπορεί να γίνει τόσο σε γραφικό περιβάλλον όσο και σε περιβάλλον κονσόλας. Για τον προγραμματισμό από την κονσόλα απαιτούνται οι δύο εντολές make ise-prog-prom και make-ise-prog-fpga. Η πρώτη μεταφέρει το αρχείο bitstream στις proms του board και η δεύτερη αναλαμβάνει τον προγραμματισμό της FPGA. Οι δύο αυτές εργασίες γίνονται μέσω του καλωδίου JTAG που παρέχεται μαζί με το board.

Σε γραφικό περιβάλλον, ανοίγοντας το IMPACT και έχοντας συνδεδεμένο το board βλέπουμε τρία blocks υλικού. Τα δύο πρώτα αφορούν τις συσκευές prom και το τρίτο

την FPGA. Κάνοντας δεξί κλικ στην εικόνα του τελευταίου και επιλέγοντας Program device μπορούμε να φορτώσουμε το .bit file. Με παρόμοιο τρόπο περνάμε και από το στάδιο του verify.

Αν όλα πάνε καλά μπορούμε πλέον να δούμε αναμμένη και την ενδεικτική λυχνία του board που αποδεικνύει το επιτυχές φόρτωμα του project.



Εικόνα 42: Προγραμματισμός FPGA μέσω Impact.

10.Συμπεράσματα και μελλοντικές εργασίες.

Κλείνοντας αυτήν την εργασία, οφείλουμε να αναφερθούμε τόσο στα συμπεράσματα στα οποία καταλήξαμε κατά την ενασχόλησή μας, όσο και στις δυνατότητες περαιτέρω ανάπτυξης του συγκεκριμένου project.

Ήταν μία άριστη ευκαιρία να έρθουμε σε επαφή με την αρχιτεκτονική SPARC με την οποία δεν είχαμε ασχοληθεί κατά την διάρκεια των σπουδών μας και να εμβαθύνουμε στην ήδη γνωστή μας αρχιτεκτονική MIPS. Επίσης, η συνεχής χρήση εξειδικευμένων εργαλείων προσομοίωσης και σχεδίασης ήταν μία πολύ χρήσιμη εμπειρία για την συνέχεια της ακαδημαϊκής ή και επαγγελματικής σταδιοδρομίας.

Το τελικό project, είναι πλήρως λειτουργικό και συμβατό με τις τεχνικές υλοποίησης για κυκλώματα FPGA. Ωστόσο, τα προβλήματα που συναντήσαμε γέννησαν πληθώρα ιδεών για μελλοντικές εργασίες, στην υλοποίηση των οποίων είμαστε διατεθειμένοι να παρέχουμε την όποια βοήθεια και ελάχιστη εμπειρία μας.

Πιο συγκεκριμένα, υπάρχει η δυνατότητα σύνδεσης και με άλλες λειτουργικές μονάδες του επεξεργαστή Leon3 όπως η μονάδα MUL/DIV, η μονάδα FPU και η μονάδα CP. Ακόμα, με κατάλληλες παρεμβάσεις, μπορεί να υποστηριχτεί η μονάδα ελέγχου διακοπών. Ένα από τα πλέον χρήσιμα κομμάτια του Leon3, που θα μπορούσε να συνδεθεί, είναι εκείνο της μονάδας αποσφαλμάτωσης (DSU), πράγμα που θα έκανε το project ευκολότερα διαχωρίσιμο με τα εργαλεία ελέγχου του Leon3, που παρέχονται ελεύθερα από την κατασκευάστρια εταιρία, αρκεί αυτά να καταστούν συμβατά με την αρχιτεκτονική MIPS. Η ενασχόλησή μας με το θέμα αυτό, οδήγησε στο συμπέρασμα πως αυτή θα ήταν μία από τις σημαντικότερες τροποποιήσεις και ελπίζουμε πως σύντομα θα ξεκινήσει μία τέτοια προσπάθεια. Για να γίνει όμως αυτό, θα πρέπει πρώτα να γίνει απόλυτα κατανοητή η λειτουργία του AMBA bus, ο οποίος θα μπορούσε ίσως να αντικατασταθεί με κάποιον δίαυλο ανοικτού κώδικα για την περίπτωση που χρειαστεί κάποια τροποποίηση για χρήση σε περιβάλλον MIPS. Έτσι

θα λύνονταν τα προβλήματα που είχαμε με την προσπέλαση της SDRAM. Τέλος, σημαντικό αντικείμενο πιθανής μελλοντικής εργασίας είναι η προσαρμογή κάποιου από τα λειτουργικά συστήματα που παρέχονται με τον Leon και το board gr-xc-3s1500 της Pender σε κώδικα MIPS, ώστε να μπορεί να τρέχει στο σύστημα που σχεδιάσαμε. Κάτι τέτοιο θα επιτρέψει τη χρήση των περιφερειακών που υποστηρίζει το περιβάλλον του Leon3, αλλά και την εκτέλεση σύνθετων μετροπρογραμμάτων για την αξιολόγηση της όλης προσπάθειας και σύγκριση με την αρχιτεκτονική SPARC του αρχικού συστήματος.

Παράρτημα Α

Α.1 Τροποποιημένος κώδικας leon3s

```
-- Entity:     leon3s
-- File:    leon3s.vhd
-- Author:    Jiri Gaisler, Edvin Catovic, Gaisler Research
-- Description:    Top-level LEON3 component
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
library gaisler;
library techmap;
use techmap.gencomp.all;
use gaisler.leon3.all;
use gaisler.libiu.all;
use gaisler.libcache.all;
use gaisler.libproc3.all;
use gaisler.arith.all;
--library fpu;
--use fpu.libfpu.all;

entity leon3s is
  generic (
    hindex    : integer             := 0;
    fabtech   : integer range 0 to NTECH  := DEFFABTECH;
    memtech   : integer range 0 to NTECH  := DEFMEMTECH;
```

```
nwindows  : integer range 2 to 32 := 8;
dsu       : integer range 0 to 1  := 0;
fpu       : integer range 0 to 31 := 0;
v8        : integer range 0 to 63 := 0;
cp        : integer range 0 to 1  := 0;
mac       : integer range 0 to 1  := 0;
pclow     : integer range 0 to 2  := 2;
notag     : integer range 0 to 1  := 0;
nwp       : integer range 0 to 4  := 0;
icen      : integer range 0 to 1  := 0;
irepl     : integer range 0 to 2  := 2;
isets     : integer range 1 to 4  := 1;
ilinesize : integer range 4 to 8  := 4;
isetsize  : integer range 1 to 256 := 1;
isetlock  : integer range 0 to 1  := 0;
dcen      : integer range 0 to 1  := 0;
drepl     : integer range 0 to 2  := 2;
dsets     : integer range 1 to 4  := 1;
dlinesize : integer range 4 to 8  := 4;
dsetsize  : integer range 1 to 256 := 1;
dsetlock  : integer range 0 to 1  := 0;
dsnoop    : integer range 0 to 6  := 0;
ilram     : integer range 0 to 1 := 0;
ilramsize  : integer range 1 to 512 := 1;
ilramstart : integer range 0 to 255 := 16#8e#;
dlram     : integer range 0 to 1 := 0;
dlramsize  : integer range 1 to 512 := 1;
dlramstart : integer range 0 to 255 := 16#8f#;
mmuen     : integer range 0 to 1  := 0;
itlbnum   : integer range 2 to 64 := 8;
dtlbnum   : integer range 2 to 64 := 8;
tlb_type  : integer range 0 to 3  := 1;
```

```
    tlb_rep   : integer range 0 to 1  := 0;
    lddel     : integer range 1 to 2  := 2;
    disas     : integer range 0 to 2  := 0;
    tbuf      : integer range 0 to 64 := 0;
    pwd       : integer range 0 to 2  := 2;     -- power-down
    svt       : integer range 0 to 1  := 1;     -- single vector trapping
    rstaddr   : integer               := 0;
    smp       : integer range 0 to 15 := 0;     -- support SMP systems
    cached    : integer               := 0;    -- cacheability table
    scantest  : integer               := 0
  );
  port (
    clk    : in  std_ulogic;
    rstn   : in  std_ulogic;
    ahbi   : in  ahb_mst_in_type;
    ahbo   : out ahb_mst_out_type;
    ahbsi  : in  ahb_slv_in_type;
    ahbso  : in  ahb_slv_out_vector;
    irqi   : in  l3_irq_in_type;
    irqo   : out l3_irq_out_type;
    dbgi   : in  l3_debug_in_type;
    dbgo   : out l3_debug_out_type
  );
end;

architecture rtl of leon3s is

--constant IRFBITS  : integer range 6 to 10 := log2(NWINDOWS+1) + 4;
--constant IREGNUM  : integer := NWINDOWS * 16 + 8;


constant IREGNUM  : integer := 32;
```

```
constant IRFBITS  : integer := log2(IREGNUM);

signal holdn : std_logic;
signal rfi   : iregfile_in_type;
signal rfo   : iregfile_out_type;
signal crami : cram_in_type;
signal cramo : cram_out_type;
signal tbi   : tracebuf_in_type;
signal tbo   : tracebuf_out_type;
signal rst   : std_ulogic;
signal fpi   : fpc_in_type;
signal fpo   : fpc_out_type;
signal cpi   : fpc_in_type;
signal cpo   : fpc_out_type;
signal cpodb : fpc_debug_out_type;

signal rd1, rd2, wd : std_logic_vector(35 downto 0);
signal gnd, vcc : std_logic;

constant FPURFHARD : integer := 1; --1-is_fpga(memtech);
constant fpuarch   : integer := fpu mod 16;
constant fpunet    : integer := fpu / 16;

attribute sync_set_reset : string;
attribute sync_set_reset of rst : signal is "true";

begin

  gnd <= '0'; vcc <= '1';

-- leon3 processor core (iu, caches & mul/div)
```

```
  p0 : proc3
  generic map (hindex, fabtech, memtech, nwindows, dsu, fpuarch, v8, cp, mac,
    pclow, notag, nwp, icen, irepl, isets, ilinesize, isetsize, isetlock,
    dcen, drepl, dsets, dlinesize, dsetsize, dsetlock, dsnoop, ilram,
    ilramsize, ilramstart, dlram, dlramsize, dlramstart, mmuen, itlbnum, dtlbnum,
    tlb_type, tlb_rep, lddel, disas, tbuf, pwd, svt, rstaddr, smp, cached, 0, scantest)
  port map (clk, rst, holdn, ahbi, ahbo, ahbsi, ahbso, rfi, rfo, crami, cramo,
    tbi, tbo, fpi, fpo, cpi, cpo, irqi, irqo, dbgi, dbgo, gnd, clk, vcc);

-- IU register file

    rf0 : regfile_3p generic map (memtech, IRFBITS, 32, 1, IREGNUM)
        port map (clk, rfi.waddr( (IRFBITS-1) downto 0), rfi.wdata, rfi.wren,
          clk, rfi.raddr1((IRFBITS-1) downto 0),rfi.ren1 , rfo.data1,
          rfi.raddr2((IRFBITS-1) downto 0), rfi.ren2, rfo.data2, rfi.diag);

-- cache memory

    cmem0 : cachemem
    generic map (memtech, icen, irepl, isets, ilinesize, isetsize, isetlock, dcen,
                 drepl, dsets,  dlinesize, dsetsize, dsetlock, dsnoop, ilram,
                 ilramsize, dlram, dlramsize, mmuen)
    port map (clk, crami, cramo, clk);

-- instruction trace buffer memory

  tbmem_gen : if (tbuf /= 0) generate
    tbmem0 : tbufmem
      generic map (tech => memtech, tbuf => tbuf)
      port map (clk, tbi, tbo);
  end generate;
```

```
-- FPU

  fpu0 : if (fpu = 0) generate fpo.ldlock <= '0'; fpo.ccv <= '1'; fpo.holdn <= '1'; end
generate;

  grfpw0gen : if (fpuarch > 0) and (fpuarch < 8) generate
    fpu0: grfpwx
      generic map (fabtech, FPURFHARD*memtech, (fpuarch-1), pclow, dsu, disas,
fpunet, 0)
      port map (rst, clk, holdn, fpi, fpo);
  end generate;

  mfpw0gen : if (fpuarch = 15) generate
    fpu0 : mfpwx
      generic map (FPURFHARD*memtech, pclow, dsu, disas)
      port map (rst, clk, holdn, fpi, fpo);
  end generate;

   grlfpc0gen : if (fpuarch >= 8) and (fpuarch < 15) generate
     fpu0 : grlfpwx
       generic map (FPURFHARD*memtech, pclow, dsu, disas, (fpuarch-8), fpunet)
       port map (rst, clk, holdn, fpi, fpo);
   end generate;
-- Default Co-Proc drivers

  cpodb.data <= zero32;
  cpo <= (zero32, '0', "00", '0', '0', '0', cpodb);

-- 1-clock reset delay
  rstreg : process(clk)
  begin if rising_edge(clk) then rst <= rstn; end if; end process;
```

```
-- pragma translate_off
  bootmsg : report_version
  generic map (
    "leon3_" & tost(hindex) & ": LEON3 SPARC V8 processor rev " &
tost(LEON3_VERSION),
    "leon3_" & tost(hindex) & ": icache " & tost(isets*icen) & "*" & tost(isetsize*icen)
&
    " kbyte, dcache "  & tost(dsets*dcen) & "*" & tost(dsetsize*dcen) & " kbyte"
  );
-- pragma translate_on

end;
```

A.2 Ενδεικτικός κώδικας περιγραφής registers από το αρχείο ulit.v

```
module ext_ctl_reg_clr_cls(input[`EXT_CTL_LEN-1:0] ext_ctl_i,output
reg[`EXT_CTL_LEN-1:0] ext_ctl_o,input clk,input clr,input cls,input
hold);always@(posedge clk)if(hold==1)begin if(clr) ext_ctl_o<=0;else
if(cls)ext_ctl_o<=ext_ctl_o;else ext_ctl_o<=ext_ctl_i; end else ; endmodule

module ext_ctl_reg_clr(input[`EXT_CTL_LEN-1:0] ext_ctl_i,output
reg[`EXT_CTL_LEN-1:0] ext_ctl_o,input clk,input clr,input hold);always@(posedge
clk)if(hold==1)begin if(clr)ext_ctl_o<=0;else ext_ctl_o<=ext_ctl_i;end else
;endmodule

module ext_ctl_reg(input[`EXT_CTL_LEN-1:0] ext_ctl_i,output
reg[`EXT_CTL_LEN-1:0] ext_ctl_o,input clk,input hold);always@(posedge
clk)if(hold==1) ext_ctl_o<=ext_ctl_i; else ;endmodule
```

```
module ins_reg(input[`INS_LEN-1:0] ins_i,output reg[`INS_LEN-1:0] ins_o,input clk,input hold);always@(posedge clk)if(hold==1) ins_o<=ins_i; else ;endmodule

module r32_inst_reg(input[`R32_LEN-1:0] r32_i,output reg[`R32_LEN-1:0] r32_o,input clk,input hold,input imds,input branch);always@(posedge clk) if (hold == 1 || imds ==0) r32_o<=r32_i; else;endmodule

module r32_data_reg(input[`R32_LEN-1:0] r32_i,output reg[`R32_LEN-1:0] r32_o,input clk,input hold,input dmds);always@(posedge clk) if (hold == 1 || dmds ==0) r32_o<=r32_i; else ;endmodule

module r4_asi_reg(input[4:0] r4_i,output reg[4:0] r4_o,input clk,input hold);always@(posedge clk)if(hold ==0) ; else r4_o<=r4_i;endmodule

module branch_reg_whold(input[`PC_LEN-1:0] pc_i,output reg[`PC_LEN-1:0] pc_o,input clk,input branch);always@(posedge clk) if (branch ==0) pc_o<=pc_i;else ; endmodule

module r32_pc_reg(input[`R32_LEN-1:0] r32_i,output reg[`R32_LEN-1:0] r32_o,input clk,input hold,input branch);always@(posedge clk) if (hold == 1) r32_o<=r32_i; else if (branch == 1) ; else ; endmodule

module r4_rdaddr_reg(input[4:0] r4_i,output reg[4:0] r4_o,input clk,input hold,input clr,input cls);always@(negedge clk)if(hold==1)begin if(clr) r4_o<=0;else if(cls)r4_o<=r4_o;else r4_o<=r4_i;end else ;endmodule

module ext_ctl_reg_cls(input[`EXT_CTL_LEN-1:0] ext_ctl_i,output reg[`EXT_CTL_LEN-1:0] ext_ctl_o,input clk,input cls,input hold);always@(posedge clk)if(hold==1)begin if(cls) ext_ctl_o<=ext_ctl_o;else ext_ctl_o<=ext_ctl_i;end else ;endmodule
```

A.3 Data hazard unit και έλεγχος στο decode stage

```
module hazard_unit
      (
        clk,load,rt,hold,
        load_o,rt_o
      );

input clk;
wire clk;
input load;
wire load;
input [4:0] rt;
wire [4:0] rt;
output load_o;
wire load_o;
output [4:0] rt_o;
wire [4:0] rt_o;
input hold;
wire hold;

r1_reg load1 (.hold(hold),
      .clk(clk),
      .r1_i(load),
      .r1_o(load_o)
      );

r5_reg rt1
          (.hold(hold),
             .clk(clk),
             .r5_i(rt),
             .r5_o(rt_o)
```

```
        );
endmodule
```

Αρχικός έλεγχος για data hazards στο decode_pipe.v

```
always @(*)
   begin
    if (load==1'b1 &&(rt1==rs  || rt1==rt  )
        begin
                    load_o=0;
                    size=`SZWORD;
                ext_ctl = `EXT_SA;
                rd_sel = `RD_RD;
                cmp_ctl = `CMP_NOP;
                pc_gen_ctl = `PC_KEP;
                fsm_dly = `FSM_NOP;
                muxa_ctl = `MUXA_EXT;
                muxb_ctl = `MUXB_RT;
                alu_func = `ALU_SLL;
                alu_we = `DIS;
                dmem_ctl = 5'b0;
                wb_we =  `DIS;
                wb_mux = `WB_ALU;
                    asi = 5'b01010;
        end
  end if
      case (inst_op)//synthesis parallel_case
```

A.4 Τροποποιημένος κώδικας top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library gaisler;
use gaisler.libiu.all;



---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity top is
    Port ( din :in cdatatype;
              zz_ins_i :in cdatatype;
           clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
              qa: in  STD_LOGIC_VECTOR (31 downto 0);
              qb: in  STD_LOGIC_VECTOR (31 downto 0);
              alu_ur: out std_logic_vector(31 downto 0);
              iset:in std_logic_vector(1 downto 0);
              dset:in std_logic_vector(1 downto 0);
              dmem_data_ur : out std_logic_vector(31 downto 0);
              dmem_ctl_ur:out std_logic_vector (4 downto 0);
           zz_pc_o1 : out  STD_LOGIC_VECTOR (31 downto 0);
            zz_pc_o2 : out  STD_LOGIC_VECTOR (31 downto 0);
              iack_o:out STD_LOGIC;
              size:out std_logic_vector (1 downto 0);
              rdaddra_o:out STD_LOGIC_VECTOR (4 downto 0);
```

```
        rdaddrb_o:out STD_LOGIC_VECTOR (4 downto 0);
        wb_we_o1:out STD_LOGIC;
        wb_addr_o1:out STD_LOGIC_VECTOR (4 downto 0);
        wb_din_o:out STD_LOGIC_VECTOR (31 downto 0);
        iflush: out std_ulogic;
        iflushl: out std_ulogic;
      ifline: out std_logic_vector(31 downto 3);
        dflush: out std_ulogic;
        dflushl: out std_ulogic;
        read1:out STD_LOGIC;
        read2:out STD_LOGIC;
        hold:in std_ulogic;
        inull:out std_ulogic;
        asi:out STD_LOGIC_VECTOR (7 downto 0);
        nullify:out std_ulogic;
        esu:out std_ulogic;
        msu:out std_ulogic;
        intack:out std_ulogic;
        fbranch:out std_logic;
        rbranch:out std_logic;
        eenaddr:out std_logic;
        dmds : in  STD_LOGIC;
        imds : in  STD_LOGIC;
        eaddr:out std_logic_vector(31 downto 0);
        pc_next:out std_logic_vector(31 downto 0);
        asi_code:out std_logic_vector(4 downto 0)
        );
end top;

architecture Behavioral of top is

signal idata:std_logic_vector (31 downto 0);
```

```
signal ddata:std_logic_vector (31 downto 0);
signal fbranch1:std_logic;
signal zz_pc:std_logic_vector (31 downto 0);
signal dmem_ctl_ur1:std_logic_vector (4 downto 0);
signal address1:std_logic_vector (4 downto 0);
signal data2:std_logic;

component my_mux
Port ( a : in  STD_LOGIC_VECTOR (31 downto 0);
       b : in  STD_LOGIC_VECTOR (31 downto 0);
       c : in  STD_LOGIC_VECTOR (31 downto 0);
       d : in  STD_LOGIC_VECTOR (31 downto 0);
       sel : in  STD_LOGIC_VECTOR (1 downto 0);
       res : out  STD_LOGIC_VECTOR (31 downto 0));
end component;



component mips_core
port(
 clk,rst,hold,imds,dmds:in std_logic;
 size:out std_logic_vector(1 downto 0);
 zz_ins_i,dout: in std_logic_vector (31 downto 0);
 iack_o : out std_logic;
 zz_pc_o,alu_ur_o,dmem_data_ur_o,wb_din_o :out std_logic_vector (31 downto 0) ;
 dmem_ctl_ur_o:out std_logic_vector (4 downto 0);
 rdaddra_o:out STD_LOGIC_VECTOR (4 downto 0);
 rdaddrb_o:out STD_LOGIC_VECTOR (4 downto 0);
 wb_we_o:out STD_LOGIC;
 wb_addr_o:out STD_LOGIC_VECTOR (4 downto 0);
 branch :out STD_LOGIC;
 qa: in  STD_LOGIC_VECTOR (31 downto 0);
 qb: in  STD_LOGIC_VECTOR (31 downto 0);
```

```
 asi_pass2:out std_logic_vector(4 downto 0)
 );
 end component;

 component reg_zero is
Port(
     address:in std_logic_vector(4 downto 0);
     we_o:   in std_logic;
     address_o: out std_logic_vector(4 downto 0);
     we_o1:  out std_logic
) ;
end component ;

begin
eenaddr<=dmem_ctl_ur1(2);
dmem_ctl_ur<=dmem_ctl_ur1;
fbranch<=fbranch1;
rbranch<=fbranch1;
zz_pc_o1<=zz_pc;
zz_pc_o2<=zz_pc;
read1<='1';
read2<='1';
iflush<='0';
iflushl<= '0';
ifline<="00000000000000000000000000000000";
dflush<='0';
dflushl<= '0';
eaddr<="00000000000000000000000000000000";
inull<='0';
nullify<='0';
esu<='0';
msu<='0';
```

```
intack<='0';

ifzero:reg_zero port map(address => address1,we_o => data2 ,address_o =>
wb_addr_o1,we_o1 => wb_we_o1);


mux1: my_mux port map (din(0),din(1),din(2),din(3),dset,ddata);

mux2: my_mux port map (zz_ins_i(0),zz_ins_i(1),zz_ins_i(2),zz_ins_i(3),iset,idata);

E1 : mips_core port map (clk =>clk,rst => rst,dout=>ddata,zz_ins_i=>idata,iack_o
=>iack_o,zz_pc_o =>zz_pc,alu_ur_o=> alu_ur,hold=>hold,dmem_data_ur_o =>
dmem_data_ur,dmem_ctl_ur_o =>
dmem_ctl_ur1,qa=>qa,qb=>qb,rdaddra_o=>rdaddra_o,rdaddrb_o=>rdaddrb_o,wb_ad
dr_o=>address1,
wb_we_o=>data2,wb_din_o=>wb_din_o,size=>size,branch=>fbranch1,imds=>imds,
dmds=>dmds,
asi_pass2=>asi_code);

end Behavioral;
```

A.5 Τροποποιημένος κώδικας libiu.vhd και proc3.vhd

```
------------------------------------------------------------------------------
-- Package:     libiu
-- File:    libiu.vhd
-- Author:    Jiri Gaisler Gaisler Research
-- Description:    LEON3 IU types and components
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
library techmap;
```

```
use techmap.gencomp.all;
library gaisler;
use gaisler.leon3.all;
use gaisler.arith.all;
use gaisler.mmuconfig.all;
--library fpu;
--use fpu.libfpu.all;

package libiu is

constant RDBITS : integer := 32;
constant IDBITS : integer := 32;

subtype cword is std_logic_vector(IDBITS-1 downto 0);
type cdatatype is array (0 to 3) of cword;



--type ctagpartype is array (0 to 3) of std_logic_vector(1 downto 0);
--type cdatapartype is array (0 to 3) of std_logic_vector(3 downto 0);
--type cvalidtype is array (0 to 3) of std_logic_vector(7 downto 0);
type cpartype is array (0 to 3) of std_logic_vector(3 downto 0); -- byte parity

type iregfile_in_type is record
  raddr1    : std_logic_vector(4 downto 0); -- read address 1
  raddr2    : std_logic_vector(4 downto 0); -- read address 2
  waddr       : std_logic_vector(4 downto 0); -- write address
  wdata     : std_logic_vector(31 downto 0); -- write data
  ren1        : std_ulogic;             -- read 1 enable
  ren2        : std_ulogic;             -- read 2 enable
  wren        : std_ulogic;              -- write enable
  diag      : std_logic_vector(3 downto 0); -- write data
end record;
```

```
type iregfile_out_type is record
  data1       : std_logic_vector(RDBITS-1 downto 0); -- read data 1
  data2       : std_logic_vector(RDBITS-1 downto 0); -- read data 2
end record;

type cctrltype is record
  burst  : std_ulogic;              -- icache burst enable
  dfrz   : std_ulogic;              -- dcache freeze enable
  ifrz   : std_ulogic;              -- icache freeze enable
  dsnoop : std_ulogic;              -- data cache snooping
  dcs    : std_logic_vector(1 downto 0);    -- dcache state
  ics    : std_logic_vector(1 downto 0);    -- icache state
end record;

type icache_in_type is record
  rpc             : std_logic_vector(31 downto 0); -- raw address (npc)
  fpc             : std_logic_vector(31 downto 0); -- latched address (fpc)
  dpc             : std_logic_vector(31 downto 0); -- latched address (dpc)
  rbranch         : std_ulogic;           -- Instruction branch
  fbranch         : std_ulogic;           -- Instruction branch
  inull           : std_ulogic;           -- instruction nullify
  su              : std_ulogic;           -- super-user
  flush           : std_ulogic;           -- flush icache
  flushl          : std_ulogic;                   -- flush line
  fline           : std_logic_vector(31 downto 3);     -- flush line offset
  pnull           : std_ulogic;
end record;

type icache_out_type is record
  data            : cdatatype;
  set             : std_logic_vector(1 downto 0);
```

```
  mexc          : std_ulogic;
  hold          : std_ulogic;
  flush         : std_ulogic;          -- flush in progress
  diagrdy       : std_ulogic;          -- diagnostic access ready
  diagdata      : std_logic_vector(IDBITS-1 downto 0);-- diagnostic data
  mds           : std_ulogic;          -- memory data strobe
  cfg           : std_logic_vector(31 downto 0);
  idle          : std_ulogic;          -- idle mode
end record;

type icdiag_in_type is record
  addr          : std_logic_vector(31 downto 0); -- memory stage address
  enable        : std_ulogic;
  read          : std_ulogic;
  tag           : std_ulogic;
  ctx           : std_ulogic;
  flush         : std_ulogic;
  ilramen       : std_ulogic;
  cctrl         : cctrltype;
  pflush        : std_ulogic;
  pflushaddr    : std_logic_vector(VA_I_U downto VA_I_D);
  pflushtyp     : std_ulogic;
  ilock         : std_logic_vector(0 to 3);
  scanen        : std_ulogic;
end record;

type dcache_in_type is record
  asi           : std_logic_vector(4 downto 0);
  maddress      : std_logic_vector(31 downto 0);
  eaddress      : std_logic_vector(31 downto 0);
  edata         : std_logic_vector(31 downto 0);
  size          : std_logic_vector(1 downto 0);
```

```
  enaddr        : std_ulogic;
  eenaddr       : std_ulogic;
  nullify       : std_ulogic;
  lock          : std_ulogic;
  read          : std_ulogic;
  write         : std_ulogic;
  flush         : std_ulogic;
  flushl        : std_ulogic;                  -- flush line
  dsuen         : std_ulogic;
  msu           : std_ulogic;                  -- memory stage supervisor
  esu           : std_ulogic;                  -- execution stage supervisor
  intack        : std_ulogic;
end record;

type dcache_out_type is record
  data          : cdatatype;
  set           : std_logic_vector(1 downto 0);
  mexc          : std_ulogic;
  hold          : std_ulogic;
  mds           : std_ulogic;
  werr          : std_ulogic;
  icdiag        : icdiag_in_type;
  cache         : std_ulogic;
  idle          : std_ulogic;          -- idle mode
  scanen        : std_ulogic;
  testen        : std_ulogic;
end record;

type tracebuf_in_type is record
  addr          : std_logic_vector(11 downto 0);
  data          : std_logic_vector(127 downto 0);
  enable        : std_logic;
```

```
  write          : std_logic_vector(3 downto 0);
  diag           : std_logic_vector(3 downto 0);
end record;

type tracebuf_out_type is record
  data           : std_logic_vector(127 downto 0);
end record;



component iu3
  generic (
    nwin     : integer range 2 to 32 := 8;
    isets    : integer range 1 to 4 := 1;
    dsets    : integer range 1 to 4 := 1;
    fpu      : integer range 0 to 15 := 0;
    v8       : integer range 0 to 63 := 0;
    cp, mac  : integer range 0 to 1 := 0;
    dsu      : integer range 0 to 1 := 0;
    nwp      : integer range 0 to 4 := 0;
    pclow    : integer range 0 to 2 := 2;
    notag    : integer range 0 to 1 := 0;
    index    : integer range 0 to 15:= 0;
    lddel    : integer range 1 to 2 := 2;
    irfwt    : integer range 0 to 1 := 0;
    disas    : integer range 0 to 2 := 0;
    tbuf     : integer range 0 to 64 := 0;  -- trace buf size in kB (0 - no trace buffer)
    pwd      : integer range 0 to 2 := 0;   -- power-down
    svt      : integer range 0 to 1 := 0;   -- single-vector trapping
    rstaddr  : integer              := 0;
    smp      : integer range 0 to 15 := 0;   -- support SMP systems
    fabtech  : integer range 0 to NTECH := 0;
    clk2x    : integer := 0
```

```
  );
 port (
  clk   : in  std_ulogic;
  rstn  : in  std_ulogic;
  holdn : in  std_ulogic;
  ici   : out icache_in_type;
  ico   : in  icache_out_type;
  dci   : out dcache_in_type;
  dco   : in  dcache_out_type;
  rfi   : out iregfile_in_type;
  rfo   : in  iregfile_out_type;
  irqi  : in  l3_irq_in_type;
  irqo  : out l3_irq_out_type;
  dbgi  : in  l3_debug_in_type;
  dbgo  : out l3_debug_out_type;
  muli  : out mul32_in_type;
  mulo  : in  mul32_out_type;
  divi  : out div32_in_type;
  divo  : in  div32_out_type;
  fpo   : in  fpc_out_type;
  fpi   : out fpc_in_type;
  cpo   : in  fpc_out_type;
  cpi   : out fpc_in_type;
  tbo   : in  tracebuf_out_type;
  tbi   : out tracebuf_in_type;
  sclk  : in  std_ulogic
 );
end component;

 component tbufmem
 generic (
  tech   : integer := 0;
```

```
  tbuf  : integer := 0
  );
 port (
  clk : in std_ulogic;
  di  : in tracebuf_in_type;
  do  : out tracebuf_out_type);
 end component;

-- disassembly dummy module

 component cpu_disasx is
 port (
  clk   : in std_ulogic;
  rstn  : in std_ulogic;
  dummy : out std_ulogic;
  inst  : in std_logic_vector(31 downto 0);
  pc    : in std_logic_vector(31 downto 2);
  result: in std_logic_vector(31 downto 0);
  index : in std_logic_vector(3 downto 0);
  wreg  : in std_ulogic;
  annul : in std_ulogic;
  holdn : in std_ulogic;
  pv    : in std_ulogic;
  trap  : in std_ulogic;
  disas : in std_ulogic);
 end component;

    component top
      port(

           din :cdatatype;
           zz_ins_i : cdatatype;
```

```
clk : in  STD_LOGIC;
rst :in STD_LOGIC;
   qa: in  STD_LOGIC_VECTOR (31 downto 0);
   qb: in  STD_LOGIC_VECTOR (31 downto 0);
   iset:in std_logic_vector(1 downto 0);
   dset:in std_logic_vector(1 downto 0);
   alu_ur: out std_logic_vector(31 downto 0);
   dmem_data_ur : out std_logic_vector(31 downto 0);
   dmem_ctl_ur:out std_logic_vector (4 downto 0);
zz_pc_o1 : out  STD_LOGIC_VECTOR (31 downto 0);
 zz_pc_o2 : out  STD_LOGIC_VECTOR (31 downto 0);
   iack_o:out STD_LOGIC;
   size:out std_logic_vector (1 downto 0);
   rdaddra_o:out STD_LOGIC_VECTOR (4 downto 0);
   rdaddrb_o:out STD_LOGIC_VECTOR (4 downto 0);
   wb_we_o1:out STD_LOGIC;
   wb_addr_o1:out STD_LOGIC_VECTOR (4 downto 0);
   wb_din_o:out STD_LOGIC_VECTOR (31 downto 0);
   iflush: out std_ulogic;
   iflushl: out std_ulogic;
ifline: out std_logic_vector(31 downto 3);
   dflush:out std_ulogic;
   dflushl:out std_ulogic;
   read1:out STD_LOGIC;
   read2:out STD_LOGIC;
   inull:out std_ulogic;
   asi:out STD_LOGIC_VECTOR (7 downto 0);
   nullify:out std_ulogic;
   esu:out std_ulogic;
   msu:out std_ulogic;
   intack:out std_ulogic;
   fbranch:out std_logic;
```

```
        rbranch:out std_logic;
        eenaddr:out std_logic;
        hold:in std_logic;
        dmds : in  STD_LOGIC;
        imds : in  STD_LOGIC;
        eaddr:out std_logic_vector(31 downto 0);
        asi_code:out std_logic_vector(4 downto 0)
        );
end component ;

end;



------------------------------------------------------------------------------
-- Entity:     proc3
-- File:    proc3.vhd
-- Author:    Jiri Gaisler Gaisler Research
-- Description:    LEON3 processor core with pipeline, mul/div & cache control
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
library techmap;
use techmap.gencomp.all;

library gaisler;
use gaisler.leon3.all;
use gaisler.libiu.all;
```

```
use gaisler.libcache.all;
use gaisler.arith.all;
--library fpu;
--use fpu.libfpu.all;

entity proc3 is
  generic (
    hindex    : integer             := 0;
    fabtech   : integer range 0 to NTECH  := 0;
    memtech   : integer range 0 to NTECH  := 0;
    nwindows  : integer range 2 to 32 := 8;
    dsu       : integer range 0 to 1  := 0;
    fpu       : integer range 0 to 15 := 0;
    v8        : integer range 0 to 63 := 0;
    cp        : integer range 0 to 1  := 0;
    mac       : integer range 0 to 1  := 0;
    pclow     : integer range 0 to 2  := 2;
    notag     : integer range 0 to 1  := 0;
    nwp       : integer range 0 to 4  := 0;
    icen      : integer range 0 to 1  := 0;
    irepl     : integer range 0 to 2  := 2;
    isets     : integer range 1 to 4  := 1;
    ilinesize : integer range 4 to 8  := 4;
    isetsize  : integer range 1 to 256 := 1;
    isetlock  : integer range 0 to 1  := 0;
    dcen      : integer range 0 to 1  := 0;
    drepl     : integer range 0 to 2  := 2;
    dsets     : integer range 1 to 4  := 1;
    dlinesize : integer range 4 to 8  := 4;
    dsetsize  : integer range 1 to 256 := 1;
    dsetlock  : integer range 0 to 1  := 0;
    dsnoop    : integer range 0 to 6  := 0;
```

```
  ilram      : integer range 0 to 1 := 0;
  ilramsize  : integer range 1 to 512 := 1;
  ilramstart : integer range 0 to 255 := 16#8e#;
  dlram      : integer range 0 to 1 := 0;
  dlramsize  : integer range 1 to 512 := 1;
  dlramstart : integer range 0 to 255 := 16#8f#;
  mmuen      : integer range 0 to 1  := 0;
  itlbnum    : integer range 2 to 64 := 8;
  dtlbnum    : integer range 2 to 64 := 8;
  tlb_type   : integer range 0 to 3 := 1;
  tlb_rep    : integer range 0 to 1 := 0;
  lddel      : integer range 1 to 2 := 2;
  disas      : integer range 0 to 2  := 0;
  tbuf       : integer range 0 to 64 := 0;
  pwd        : integer range 0 to 2 := 0;
  svt        : integer range 0 to 1 := 0;   -- single-vector trapping
  rstaddr  : integer              := 0;
  smp        : integer range 0 to 15 := 0;  -- support SMP systems
  cached     : integer := 0;
  clk2x      : integer := 0;
  scantest : integer := 0
);
port (
  clk    : in  std_ulogic;
  rstn   : in  std_ulogic;
  holdn  : out std_ulogic;
  ahbi   : in  ahb_mst_in_type;
  ahbo   : out ahb_mst_out_type;
  ahbsi  : in  ahb_slv_in_type;
  ahbso  : in  ahb_slv_out_vector;
  rfi    : out iregfile_in_type;
  rfo    : in  iregfile_out_type;
```

```
    crami  : out cram_in_type;
    cramo  : in  cram_out_type;
    tbi    : out tracebuf_in_type;
    tbo    : in  tracebuf_out_type;
    fpi    : out fpc_in_type;
    fpo    : in  fpc_out_type;
    cpi    : out fpc_in_type;
    cpo    : in  fpc_out_type;
    irqi   : in  l3_irq_in_type;
    irqo   : out l3_irq_out_type;
    dbgi   : in  l3_debug_in_type;
    dbgo   : out l3_debug_out_type;
     iack_o : out std_logic;
    hclk, sclk   : in std_ulogic;
    hclken   : in std_ulogic
  );
end;

architecture rtl of proc3 is


constant IRFWT    : integer := regfile_3p_write_through(memtech);

signal ici : icache_in_type;
signal ico : icache_out_type;
signal dci : dcache_in_type;
signal dco : dcache_out_type;

signal holdnx, pholdn : std_logic;
signal muli  : mul32_in_type;
signal mulo  : mul32_out_type;
signal divi  : div32_in_type;
```

```
signal divo  : div32_out_type;


begin

  holdnx <= ico.hold and dco.hold; holdn <= holdnx;
  pholdn <= fpo.holdn;


-- integer unit

--     iu0 : iu3
--       generic map (nwindows, isets, dsets, fpu, v8, cp, mac, dsu, nwp, pclow,
--     0, hindex, lddel, IRFWT, disas, tbuf, pwd, svt, rstaddr, smp, fabtech, clk2x)
--       port map (clk, rstn, holdnx, ici, ico, dci, dco, rfi, rfo, irqi, irqo,
--                 dbgi, dbgo, muli, mulo, divi, divo, fpo, fpi, cpo, cpi, tbo, tbi, sclk);

       mips : top
          port map (clk=>clk ,rst=>rstn, din=>dco.data
,zz_ins_i=>ico.data,qa=>rfo.data1,hold=>holdnx,imds=>ico.mds,dmds=>dco.mds,qb
=>rfo.data2,
rdaddra_o=>rfi.raddr1,rdaddrb_o=>rfi.raddr2,wb_we_o1=>rfi.wren,wb_din_o=>rfi.w
data,wb_addr_o1=>
rfi.waddr,pc_next=>ici.rpc,zz_pc_o1=>ici.fpc,iack_o=>iack_o,iflush=>ici.flush,iflus
hl=>ici.flushl,
ifline=>ici.fline,dflush=>dci.flush,dflushl=>dci.flushl,read1=>rfi.ren1,read2=>rfi.ren
2,alu_ur=>dci.maddress,
dmem_data_ur=>dci.edata,size=>dci.size,dmem_ctl_ur(0)=>dci.dsuen,dmem_ctl_ur(
1)=>dci.read,
dmem_ctl_ur(2)=>dci.enaddr,dmem_ctl_ur(3)=>dci.write,dmem_ctl_ur(4)=>dci.lock,
 inull=>ici.inull,asi_code=>dci.asi,nullify=>dci.nullify,esu=>dci.esu,msu=>dci.msu,
intack=>dci.intack,eenaddr=>dci.eenaddr,eaddr=>dci.eaddress,iset=>ico.set,dset=>dc
```

```
o.set,rbranch=>ici.rbranch,
           fbranch=>ici.fbranch);
   -- multiply and divide units
-- Actel FPGAs cannot use inferred mul due to bug in synplify 8.9 and 9.0

  mgen : if v8 /= 0 generate
   mgen2 : if (fabtech = axcel) or (fabtech = apa3) generate
    mul0 : mul32 generic map (0, v8/16, (v8 mod 4)/2, mac)
       port map (rstn, clk, holdnx, muli, mulo);
   end generate;
   mgen3 : if not ((fabtech = axcel) or (fabtech = apa3)) generate
    mul0 : mul32 generic map (is_fpga(fabtech), v8/16, (v8 mod 4)/2, mac)
       port map (rstn, clk, holdnx, muli, mulo);
   end generate;
    div0 : div32 port map (rstn, clk, holdnx, divi, divo);
  end generate;
  nomgen : if v8 = 0 generate
    divo <= ('0', '0', "0000", zero32);
    mulo <= ('0', '0', "0000", zero32&zero32);
  end generate;

-- cache controller

  m0 : if mmuen = 0 generate
    c0 : cache
      generic map (hindex, dsu, icen, irepl, isets, ilinesize, isetsize,
    isetlock, dcen, drepl, dsets, dlinesize, dsetsize,  dsetlock, dsnoop,
    ilram, ilramsize, ilramstart, dlram, dlramsize, dlramstart, cached,
    clk2x, memtech, scantest)
      port map ( rstn, clk, ici, ico, dci, dco, ahbi, ahbo, ahbsi, ahbso, crami, cramo,
pholdn, hclk, sclk, hclken);
  end generate;
```

```
  m1 : if mmuen = 1 generate
    c0mmu : mmu_cache
      generic map (hindex=>hindex, memtech=>memtech, dsu=>dsu, icen=>icen,
irepl=>irepl,
     isets=>isets, ilinesize=>ilinesize, isetsize=>isetsize, isetlock=>isetlock,
        dcen=>dcen, drepl=>drepl, dsets=>dsets, dlinesize=>dlinesize,
dsetsize=>dsetsize,
     dsetlock=>dsetlock, dsnoop=>dsnoop, itlbnum=>itlbnum, dtlbnum=>dtlbnum,
     tlb_type=>tlb_type, tlb_rep=>tlb_rep, cached => cached, clk2x => clk2x,
     scantest => scantest)
      port map ( rstn, clk, ici, ico, dci, dco,
     ahbi, ahbo, ahbsi, ahbso, crami, cramo, pholdn, hclk, sclk, hclken);
  end generate;

end;
```

Παράρτημα Β

Β.1 Κώδικες .elf files

prom.elf

```
00000000 <A>:
  000000:   00000025   move   zero,zero
  000004:   3c080081   lui    t0,0x81
  000008:   3508000f   ori    t0,t0,0xf
  00000c:   e0080000   swc0   $8,0(zero)
  000010:   00000025   move   zero,zero
  000014:   00000025   move   zero,zero
  000018:   00000025   move   zero,zero
  00001c:   00000025   move   zero,zero
  000020:   00000025   move   zero,zero
  000024:   00000025   move   zero,zero
  000028:   00000025   move   zero,zero
  00002c:   00000025   move   zero,zero
  000030:   00000025   move   zero,zero
  000034:   00000025   move   zero,zero
  000038:   00000025   move   zero,zero
  00003c:   00000025   move   zero,zero

00000040 <B>:
  000040:   3c080008   lui    t0,0x8
  000044:   3c091038   lui    t1,0x1038
  000048:   35290033   ori    t1,t1,0x33
  00004c:   ad090000   sw     t1,0(t0)
  000050:   3c09e6b8   lui    t1,0xe6b8
  000054:   35296e60   ori    t1,t1,0x6e60
```

```
000058:    ad090004     sw    t1,4(t0)
00005c:    3c09000f     lui    t1,0xf
000060:    3529f000     ori    t1,t1,0xf000
000064:    ad090008     sw    t1,8(t0)
000068:    3c08ffff      lui    t0,0xffff
00006c:    3508f860     ori    t0,t0,0xf860
000070:    8d090000     lw    t1,0(t0)
000074:    00000000     nop
000078:    00094b02     srl    t1,t1,0xc
00007c:    34081009     li    t0,t0,0x1009
000080:    15090005     bne    t0,t1,4001f8 <L>
000084:    00000000     nop
000088:    3c088000     lui    t0,0x8000
00008c:    3c09e6a0     lui    t1,0xe6a0
000090:    35296e60     ori    t1,t1,0x6e60
000094:    ad090000     sw    t1,0(t0)

00000098 <L>:
000098:    34090002     li    t1,0x2
00009c:    3c1d040f     lui    sp,0x40f
0000a0:    37bdffe0     ori    sp,sp,0xffe0
0000a4:    013de822     sub    sp,t1,sp
0000a8:    3c084000     lui    t0,0x0040
0000ac:    01000008     jr    t0
0000b0:    00000000     nop
0000b4:    00000025     move    zero,zero
0000b8:    00000025     move    zero,zero
0000bc:    00000025     move    zero,zero
```

sdram.elf

```
000000D0 <G>:
0000D0:    00008020     add    s0,zero,zero
```

0000D4: 00008820 add s1,zero,zero

0000D8: 1211fffd beq s0,s1,400164 <G>

0000DC: 00000000 nop

B.2 Srec files

prom.srec

S01100006D6970735F70726F6D2E737265633D

S214000000000000253C0800813508000FE0080000CD

S2140000100000002500000025000000250000002547

S2140000200000002500000025000000250000002537

S2140000300000002500000025000000250000002527

S2140000403C0800083C09103835290033AD0900008B

S2140000503C09E6B835296E60AD0900043C09000F7E

S2140000603529F000AD0900083C08FFFF3508F860A8

S2140000708D0900000000000000094B02340810093A

S214000080150900050000000003C0880003C09E6A0B9

S21400009035296E60AD090000340900023C1D040FCE

S2140000A037BDFFE0013DE8223C0800400000002587

S2140000B000000025000000250000002501000008C3

S2140000C0000000000000002500000025000000025CC

S8044000407B

sdram.srec

S017000073696D706C655F617373656D626C792E73726563C4

S2140000D000008020000088201211fffd00000000B4

S8044000407B

B.3 modelsim.do file

vsim -quiet work.testbench

view wave

```
add wave sim:/testbench/cpu/l3/cpu__0/u0/rf0/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/m0/c0/icache0/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/m0/c0/dcache0/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/e1/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/e1/decoder_pipe/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/e1/iRF_stage/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/e1/iforward/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/e1/iexec_stage/*
add wave sim:/testbench/cpu/l3/cpu__0/u0/p0/mips/e1/ihazard_unit/*
run -all
```

B.4 Module pc_gen

```
module pc_gen(
        input [2:0]ctl,
        input hold,
        input clk,
        output reg   [31:0]pc_next,
          output reg   branch,
        input [3:0] pc_prectl,
        input check,
        input [31:0]s,
        input [31:0]pc,
        input [31:0]zz_spc,
        input [31:0]imm
    );

    wire [32:0] br_addr = pc-4 + imm ;
    always @ (*)
        if(pc_prectl == `PC_IGN )
        begin
```

```
        case (ctl)
          `PC_RET     :begin    pc_next = zz_spc ; branch=1; end
          `PC_J       :begin    pc_next ={pc[31:28],imm[27:0]}; branch=1; end
          `PC_JR      :begin     pc_next = s; branch=1; end
          `PC_BC      :begin    pc_next = (check)?({br_addr[31:0]}):(pc+4);
                          if (check == 1'b1)
                             branch=1;
                          else
                             branch=0;
                       end
         default
         /* `PC_NEXT    :*/begin    pc_next = pc + 4 ;  branch=0; end
        endcase
      end
      else
      begin
        case (pc_prectl)
          `PC_KEP     : pc_next=pc;
//          `PC_IRQ     : pc_next=irq;
         default
         /* `PC_RST     : pc_next='d0;*/
              pc_next =0;
        endcase
      end

endmodule
```

B.5 Entity reg_zero

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity reg_zero is
Port(
     address:in std_logic_vector(4 downto 0);
     we_o:   in std_logic;
     address_o:  out std_logic_vector(4 downto 0);
     we_o1:  out std_logic
) ;
end reg_zero;

architecture behavioural of reg_zero is
signal addr1:std_logic_vector(4 downto 0);

begin
addr1<=address;
process (addr1)
   variable i:integer:=0;

   begin
    if (((addr1(0) = '0') and (addr1(1)='0') and (addr1(2) = '0') and(addr1(3) = '0') and
(addr1(4) = '0'))and(i=0)) then
       we_o1<='1' after 0ns,'0' after 200ns;
        i:=1;

     elsif  ((addr1(0) = '0') and (addr1(1)='0') and (addr1(2) = '0') and(addr1(3) = '0')
and (addr1(4) = '0'))  then
```

```
        we_o1<='0';
      else
        we_o1 <= we_o;
end if;
end process;
address_o<=address;
end behavioural;
```

B.6 Τελικός έλεγχος για data hazards στο decode_pipe.v

```
if (load==1'b1 &&((rt1==rs && read_rs==1'b1) || (rt1==rt && read_rs==1'b1) ))
          begin
                        load_o=0;
                        read_rt=1'b0;
                      read_rs=1'b0;
                        size=`SZWORD;
                ext_ctl = `EXT_SA;
                rd_sel = /*`RD_RD*/`RD_NOP;
                cmp_ctl = `CMP_NOP;
                pc_gen_ctl = /*`PC_KEP*/`PC_NEXT;
                fsm_dly = /*`FSM_NOP*/`FSM_CUR;
                muxa_ctl = `MUXA_EXT;
                muxb_ctl = `MUXB_RT;
                alu_func = `ALU_SLL;
                alu_we = `DIS;
                dmem_ctl = 5'b0;
                wb_we =  `DIS;
                wb_mux = `WB_ALU;
                      asi = 5'b01010;
          end
```

B.7 Συμπληρωματικοί καταχωρητές rdaddr_reg

module r4_rdaddr_reg(input[4:0] r4_i,output reg[4:0] r4_o,input clk,input hold,input clr,input cls);always@(negedge clk)if(hold==1)begin if(clr) r4_o<=0;else if(cls)r4_o<=r4_o;else r4_o<=r4_i;end else ;endmodule

B.8 Αρχείο διαμόρφωσης config.vhd

```
------------------------------------------------------------------------------
-- LEON3 Demonstration design test bench configuration
-- Copyright (C) 2004 Jiri Gaisler, Gaisler Research
------------------------------------------------------------------------------

library techmap;
use techmap.gencomp.all;

package config is
-- Technology and synthesis options
  constant CFG_FABTECH : integer := spartan3;
  constant CFG_MEMTECH : integer := spartan3;
  constant CFG_PADTECH : integer := spartan3;
  constant CFG_NOASYNC : integer := 0;
  constant CFG_SCAN : integer := 0;

-- Clock generator
```

```
  constant CFG_CLKTECH : integer := spartan3;
  constant CFG_CLKMUL : integer := (4);
  constant CFG_CLKDIV : integer := (5);
  constant CFG_OCLKDIV : integer := 2;
  constant CFG_PCIDLL : integer := 0;
  constant CFG_PCISYSCLK: integer := 0;
  constant CFG_CLK_NOFB : integer := 0;

-- LEON3 processor core
  constant CFG_LEON3 : integer := 1;
  constant CFG_NCPU : integer := (1);
  constant CFG_NWIN : integer := (8);
  constant CFG_V8 : integer := 16#32#;
  constant CFG_MAC : integer := 0;
  constant CFG_SVT : integer := 1;
  constant CFG_RSTADDR : integer := 16#00000#;
  constant CFG_LDDEL : integer := (1);
  constant CFG_NWP : integer := (2);
  constant CFG_PWD : integer := 0*2;
  constant CFG_FPU : integer := 0 + 16*0;
  constant CFG_GRFPUSH : integer := 0;
```

```
constant CFG_ICEN : integer := 1;
constant CFG_ISETS : integer := 1;
constant CFG_ISETSZ : integer := 4;
constant CFG_ILINE : integer := 8;
constant CFG_IREPL : integer := 0;
constant CFG_ILOCK : integer := 0;
constant CFG_ILRAMEN : integer := 0;
constant CFG_ILRAMADDR: integer := 16#8E#;
constant CFG_ILRAMSZ : integer := 1;
constant CFG_DCEN : integer := 1;
constant CFG_DSETS : integer := 4;
constant CFG_DSETSZ : integer := 4;
constant CFG_DLINE : integer := 4;
constant CFG_DREPL : integer := 0;
constant CFG_DLOCK : integer := 0;
constant CFG_DSNOOP : integer := 1 + 0 + 4*0;
constant CFG_DFIXED : integer := 16#0#;
constant CFG_DLRAMEN : integer := 0;
constant CFG_DLRAMADDR: integer := 16#8F#;
constant CFG_DLRAMSZ : integer := 1;
constant CFG_MMUEN : integer := 0;
```

```
  constant CFG_ITLBNUM : integer := 2;
  constant CFG_DTLBNUM : integer := 2;
  constant CFG_TLB_TYPE : integer := 1 + 0*2;
  constant CFG_TLB_REP : integer := 1;
  constant CFG_DSU : integer := 0;
  constant CFG_ITBSZ : integer := 0;
  constant CFG_ATBSZ : integer := 0;
  constant CFG_LEON3FT_EN : integer := 0;
  constant CFG_IUFT_EN : integer := 0;
  constant CFG_FPUFT_EN : integer := 0;
  constant CFG_RF_ERRINJ : integer := 0;
  constant CFG_CACHE_FT_EN : integer := 0;
  constant CFG_CACHE_ERRINJ : integer := 0;
  constant CFG_LEON3_NETLIST: integer := 0;
  constant CFG_DISAS : integer := 0 + 0;
  constant CFG_PCLOW : integer := 2;

-- AMBA settings
  constant CFG_DEFMST : integer := (0);
  constant CFG_RROBIN : integer := 1;
  constant CFG_SPLIT : integer := 0;
```

```
  constant CFG_AHBIO : integer := 16#FFF#;
  constant CFG_APBADDR : integer := 16#800#;
  constant CFG_AHB_MON : integer := 0;
  constant CFG_AHB_MONERR : integer := 0;
  constant CFG_AHB_MONWAR : integer := 0;

-- DSU UART
  constant CFG_AHB_UART : integer := 1;

-- JTAG based DSU interface
  constant CFG_AHB_JTAG : integer := 1;

-- USB DSU
  constant CFG_GRUSB_DCL : integer := 0;
  constant CFG_GRUSB_DCL_UIFACE : integer := 1;
  constant CFG_GRUSB_DCL_DW : integer := 8;

-- Ethernet DSU
  constant CFG_DSU_ETH : integer := 0 + 0;
  constant CFG_ETH_BUF : integer := 1;
  constant CFG_ETH_IPM : integer := 16#C0A8#;
```

```
  constant CFG_ETH_IPL : integer := 16#0033#;
  constant CFG_ETH_ENM : integer := 16#00007A#;
  constant CFG_ETH_ENL : integer := 16#CC0001#;

-- LEON2 memory controller
  constant CFG_MCTRL_LEON2 : integer := 1;
  constant CFG_MCTRL_RAM8BIT : integer := 1;
  constant CFG_MCTRL_RAM16BIT : integer := 1;
  constant CFG_MCTRL_5CS : integer := 0;
  constant CFG_MCTRL_SDEN : integer := 1;
  constant CFG_MCTRL_SEPBUS : integer := 0;
  constant CFG_MCTRL_INVCLK : integer := 0;
  constant CFG_MCTRL_SD64 : integer := 0;
  constant CFG_MCTRL_PAGE : integer := 0 + 0;

-- AHB status register
  constant CFG_AHBSTAT : integer := 0;
  constant CFG_AHBSTATN : integer := 1;

-- AHB ROM
  constant CFG_AHBROMEN : integer := 0;
```

```
  constant CFG_AHBROPIP : integer := 0;
  constant CFG_AHBRODDR : integer := 16#000#;
  constant CFG_ROMADDR : integer := 16#000#;
  constant CFG_ROMMASK : integer := 16#E00# + 16#000#;

-- AHB RAM
  constant CFG_AHBRAMEN : integer := 0;
  constant CFG_AHBRSZ : integer := 1;
  constant CFG_AHBRADDR : integer := 16#A00#;

-- Gaisler Ethernet core
  constant CFG_GRETH : integer := 0;
  constant CFG_GRETH1G : integer := 0;
  constant CFG_ETH_FIFO : integer := 8;

-- ATA interface
  constant CFG_ATA : integer := 0;
  constant CFG_ATAIO : integer := 16#0#;
  constant CFG_ATAIRQ : integer := 0;
  constant CFG_ATADMA : integer := 0;
  constant CFG_ATAFIFO : integer := 8;
```

```
-- CAN 2.0 interface
  constant CFG_CAN : integer := 0;
  constant CFG_CAN_NUM : integer := 1;
  constant CFG_CANIO : integer := 16#0#;
  constant CFG_CANIRQ : integer := 0;
  constant CFG_CANSEPIRQ: integer := 0;
  constant CFG_CAN_SYNCRST : integer := 0;
  constant CFG_CANFT : integer := 0;

-- GR USB 2.0 Device Controller
  constant CFG_GRUSBDC : integer := 0;
  constant CFG_GRUSBDC_AIFACE : integer := 0;
  constant CFG_GRUSBDC_UIFACE : integer := 1;
  constant CFG_GRUSBDC_DW : integer := 8;
  constant CFG_GRUSBDC_NEPI : integer := 1;
  constant CFG_GRUSBDC_NEPO : integer := 1;
  constant CFG_GRUSBDC_I0 : integer := 1024;
  constant CFG_GRUSBDC_I1 : integer := 1024;
  constant CFG_GRUSBDC_I2 : integer := 1024;
  constant CFG_GRUSBDC_I3 : integer := 1024;
```

```
constant CFG_GRUSBDC_I4 : integer := 1024;
constant CFG_GRUSBDC_I5 : integer := 1024;
constant CFG_GRUSBDC_I6 : integer := 1024;
constant CFG_GRUSBDC_I7 : integer := 1024;
constant CFG_GRUSBDC_I8 : integer := 1024;
constant CFG_GRUSBDC_I9 : integer := 1024;
constant CFG_GRUSBDC_I10 : integer := 1024;
constant CFG_GRUSBDC_I11 : integer := 1024;
constant CFG_GRUSBDC_I12 : integer := 1024;
constant CFG_GRUSBDC_I13 : integer := 1024;
constant CFG_GRUSBDC_I14 : integer := 1024;
constant CFG_GRUSBDC_I15 : integer := 1024;
constant CFG_GRUSBDC_O0 : integer := 1024;
constant CFG_GRUSBDC_O1 : integer := 1024;
constant CFG_GRUSBDC_O2 : integer := 1024;
constant CFG_GRUSBDC_O3 : integer := 1024;
constant CFG_GRUSBDC_O4 : integer := 1024;
constant CFG_GRUSBDC_O5 : integer := 1024;
constant CFG_GRUSBDC_O6 : integer := 1024;
constant CFG_GRUSBDC_O7 : integer := 1024;
constant CFG_GRUSBDC_O8 : integer := 1024;
```

```
  constant CFG_GRUSBDC_O9 : integer := 1024;
  constant CFG_GRUSBDC_O10 : integer := 1024;
  constant CFG_GRUSBDC_O11 : integer := 1024;
  constant CFG_GRUSBDC_O12 : integer := 1024;
  constant CFG_GRUSBDC_O13 : integer := 1024;
  constant CFG_GRUSBDC_O14 : integer := 1024;
  constant CFG_GRUSBDC_O15 : integer := 1024;
-- UART 1
  constant CFG_UART1_ENABLE : integer := 1;
  constant CFG_UART1_FIFO : integer := 4;

-- UART 2
  constant CFG_UART2_ENABLE : integer := 0;
  constant CFG_UART2_FIFO : integer := 1;

-- LEON3 interrupt controller
  constant CFG_IRQ3_ENABLE : integer := 1;
  constant CFG_IRQ3_NSEC : integer := 0;

-- Modular timer
  constant CFG_GPT_ENABLE : integer := 0;
```

```
  constant CFG_GPT_NTIM : integer := 1;
  constant CFG_GPT_SW : integer := 8;
  constant CFG_GPT_TW : integer := 8;
  constant CFG_GPT_IRQ : integer := 8;
  constant CFG_GPT_SEPIRQ : integer := 0;
  constant CFG_GPT_WDOGEN : integer := 0;
  constant CFG_GPT_WDOG : integer := 16#0#;

-- GPIO port
  constant CFG_GRGPIO_ENABLE : integer := 0;
  constant CFG_GRGPIO_IMASK : integer := 16#0000#;
  constant CFG_GRGPIO_WIDTH : integer := 1;

-- Spacewire interface
  constant CFG_SPW_EN : integer := 0;
  constant CFG_SPW_NUM : integer := 1;
  constant CFG_SPW_AHBFIFO : integer := 4;
  constant CFG_SPW_RXFIFO : integer := 16;
  constant CFG_SPW_RMAP : integer := 0;
  constant CFG_SPW_RMAPBUF : integer := 4;
  constant CFG_SPW_RMAPCRC : integer := 0;
```

```
  constant CFG_SPW_NETLIST : integer := 0;
  constant CFG_SPW_FT : integer := 0;
  constant CFG_SPW_GRSPW : integer := 2;

-- VGA and PS2/ interface
  constant CFG_KBD_ENABLE : integer := 0;
  constant CFG_VGA_ENABLE : integer := 0;
  constant CFG_SVGA_ENABLE : integer := 0;

-- GRLIB debugging
  constant CFG_DUART : integer := 0;
end;
```

Βιβλιογραφία

[1] David A. Patterson, John L. Hennessy: "Computer organization and design – The hardware/Software Interface – 3rd Edition", Morgan Kaufmann, 2005

[2] John L. Hennessy, David A. Patterson: "Computer Architecture – A quantative approach – 4th Edition", Morgan Kaufmann, 2007

[3] Sandi Habinc: "Lessons Learned from FPGA Developments", Gaisler Research, 2002

[4]Jiri Gaisler, Edvin Catovic, Marko Isomaki, Kristoffer Glembo, Sandi Habinc: "Grlib IP Core User's Manual", Gaisler Research, 2008

[5] Jiri Gaisler, Edvin Catovic, Sandi Habinc: "Grlib IP Library User's Manual", Gaisler Research, 2008

[6]Jiri Gaisler, Marko Isomaki: "Leon 3 GR-XC3S-1500 Template Design – Based on GRLIB", Gaisler Research, 2006

[7] "GR-XC-3S-1500 Deve3lopment Board – User Manual", Gaisler Research/Pender Electronics, 2006

[8] Jiri Gaisler Mailing List, sparc@yahoogroups.com

[9] Lutz Buttelmann: "How to setup LEON3 VHDL simulation with Modelsim", 2007

[10] "The SPARC architecture manual – version 8" SPARC International Inc.

[11] "Amba Specification, Rev, 2.0", Arm Limited, 1999.

[12] "ISE Release Notes and Installation Guide", Xilinx Inc. 2004

[13] "ISE In-Depth Tutorial" Xilinx Inc. 2004

[14] "XST User Guide", Xilinx Inc. 2004

[15] “Fpga Editor Guide” Xilinx Inc. 2004

[16] “Fpga Tutorial” http://www.fpga4fun.com

[17] Miles J. Murdocca, Vincent P. Heuring: “Principles of Computer Architecture”, 1999

[18] Sivarama P. Dandamudi: “Guide to RISC Processors for Programmers and Engineers”, Springer, 2005

[19] “MIPS IV Instruction Set”, Revision 3.2, 1995

[20] “Sparc 7 Instruction Set- Assembly Language Syntax”, Atmel Inc., 2002

[21] Dominic Sweetman: “See MIPS run Linux – Second Edition”, Morgan Kaufmann, 2007

[22] Δ. Νικολός: «Αρχιτεκτονική Υπολογιστών», Πανεπιστημιακές εκδόσεις Θεσσαλίας, 2000