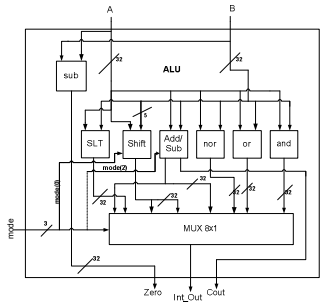
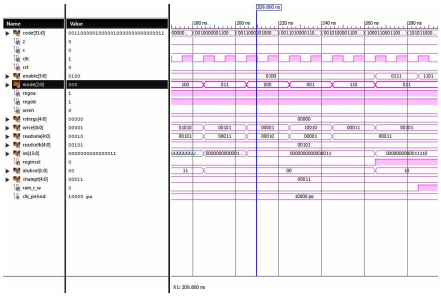
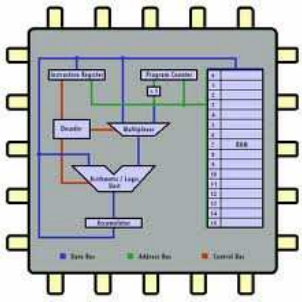
Τ.Ε.Ι. ΚΡΗΤΗΣ / ΠΑΡΑΡΤΗΜΑ ΧΑΝΙΩΝ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΗΣ

Μελέτη και σχεδίαση μιας υποτυπώδους κεντρικής μονάδας επεξεργασίας στα 32 μπιτ.







ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εμμανουήλ Καπαρού

Επιβλέπων : Δρ. Μηχ. Νικόλαος Στ. Πετράκης
Καθηγητής Εφαρμογών

Χανιά 2012

Πίνακας περιεχομένων:

Περίληψη

Η παρούσα πτυχιακή εργασία, αναφέρεται στην υλοποίηση ενός μικροεπεξεργαστή στα 32 bit, ο οποίος βασίζεται στη λειτουργία του MIPS, με χρήση ψηφιακής σχεδίασης και κώδικα της γλώσσας περιγραφής υλικού VHDL. Ο μικροεπεξεργαστής αυτός, αποτελείται από μια Αριθμητική και Λογική Μονάδα (ALU), ένα αρχείο καταχωρητών (Register File), μια Μονάδα Ελέγχου (Control Unit), καθώς και μερικούς ακόμα καταχωρητές και πολυπλέκτες και τις απαραίτητες αρτηρίες. Η ALU εκτελεί αριθμητικές και λογικές πράξεις μεταξύ ακεραίων. Στο αρχείο καταχωρητών υπάρχουν 32 καταχωρητές από και προς τους οποίους φορτώνονται και αποθηκεύονται δεδομένα. Η Μονάδα Ελέγχου παράγει σήματα ελέγχου με τα οποία ελέγχει τις υπόλοιπες μονάδες του επεξεργαστή. Οι επιπλέον καταχωρητές χρησιμοποιούνται τόσο κατά την εισαγωγή δεδομένων στον επεξεργαστή όσο και κατά την εξαγωγή από αυτόν. Επαληθεύτηκε η ορθή λειτουργία των κυκλωμάτων με πληθώρα κατάλληλων προσομοιώσεων στο περιβάλλον ISE της Xilinx.

Abstract

The present work is referred on the implementation of a 32 bits microprocessor, which is based on MIPS-ISA, using VHDL code for digital design. This microprocessor is consisted of an Arithmetic and Logic Unit (ALU), a Register File and a Control Unit with the supporting circuits. The ALU performs arithmetic and logic operations between integers. In the Register File there are 32 Registers, for loading and storing data. The Control Unit produces control signals in order to control the other units of the microprocessor. The extra registers are used either for importing data to the processor or for exporting data from it. The proper functioning was verified by corresponding simulation in the integrated environment of Xilinx ISE.

1. Εισαγωγή

Η παρούσα πτυχιακή εργασία αναφέρεται στη σχεδίαση ενός μικροεπεξεργαστή με τη χρήση κώδικα σε γλώσσα VHDL. Για την καταγραφή του κώδικα και των προσομοιώσεών του, χρησιμοποιήθηκε το πρόγραμμα ISE Project Navigator της Xilinx. Για την συγγραφή του κειμένου της παρούσας πτυχιακής εργασίας και για το σχεδιασμό των διαγραμμάτων που εμπεριέχονται, χρησιμοποιήθηκε το Microsoft Word 2007 και το Microsoft Visio 2007.

Ο μικροεπεξεργαστής που υλοποιήθηκε είναι εμπνευσμένος από την αρχιτεκτονική του συνόλου εντολών του MIPS-32. Έτσι, λειτουργεί στα 32 bits και υποστηρίζει τις παρακάτω εντολές, οι οποίες είναι ένα μικρό υποσύνολο των εντολών του MIPS-32:

1) and	8) slt
2) or	9) andi
3) nor	10) ori
4) add	11) addi
5) sub	12) slti
6) srl	13) sw
7) sll	14) lw

Στο εσωτερικό του κύκλωμα, περιέχει τις εξής μονάδες:

- 1 Αριθμητική και Λογική μονάδα (ALU), όπου γίνονται όλες οι αριθμητικές και λογικές πράξεις,
- 1 αρχείο καταχωρητών (Register File), το οποίο περιέχει 32 καταχωρητές των 32 bits, οι οποίοι χρησιμοποιούνται για την αποθήκευση δεδομένων κατά την εκτέλεση των εντολών του επεξεργαστή,
- 1 Μονάδα Ελέγχου, η οποία ελέγχει τις υπόλοιπες μονάδες του επεξεργαστή με διάφορα σήματα ελέγχου, έτσι ώστε να εκτελείται κάθε φορά η σωστή εντολή,
- 2 πολυπλέκτες, για την επιλογή του σήματος που θα περάσει από δυο συγκεκριμένα σημεία του κυκλώματος και τέλος,
- 4 καταχωρητές, 3 για την επιλογή των σημάτων που θα εισέλθουν από την είσοδο/έξοδο κωδικοποίησης και δεδομένων του επεξεργαστή και 1 για την εξαγωγή της διεύθυνσης μνήμης όταν εκτελείται εντολή μεταφοράς δεδομένων.

Η ALU, περιέχει 8 κυκλώματα των οποίων οι έξοδοι καταλήγουν στις εισόδους ενός πολυπλέκτη 8x1 για την επιλογή της εξόδου που πρέπει να βγει στην έξοδο της ALU,

ανάλογα με την πράξη που πρέπει να εκτελεστεί. Περιέχει. λοιπόν, έναν αθροιστή αφαιρέτη, για την εκτέλεση πρόσθεσης και αφαίρεσης, ένα κύκλωμα αμφίδρομης ολίσθησης για τις ολισθήσεις, 3 κυκλώματα με πύλες για την εκτέλεση των λογικών πράξεων and, or και nor και ένα συγκριτή, η οποίος χρησιμοποιείται όταν εκτελείται μια από τις εντολές slt και slti. Τέλος, η ALU περιέχει και έναν αφαιρέτη, ο οποίος χρησιμοποιείται για την έξοδο Zero.

Το Register File, περιέχει έναν αποκωδικοποιητή 5x32 για την επιλογή του καταχωρητή εγγραφής, δυο κυκλώματα πολυπλεκτών για την επιλογή των 2 καταχωρητών ανάγνωσης και ένα κύκλωμα με 32 καταχωρητές των 32 bits.

Όσον αφορά τη μονάδα ελέγχου, έχει μια είσοδο Reset για το μηδενισμό της, μια είσοδο ρολογιού για το χρονισμό της και μια είσοδο από την οποία παίρνει την κωδικοποίηση εντολής. Ανάλογα με την κωδικοποίηση εντολής που έχει λάβει, βγάζει τις κατάλληλες εξόδους για την εκτέλεση της. Οι έξοδοι αυτές είναι: μια έξοδος για να δείξει στην ALU τι πράξη θα εκτελέσει, μια έξοδο Enable για την ενεργοποίηση των κατάλληλων καταχωρητών που εισάγουν και εξάγουν δεδομένα στον επεξεργαστή, 3 εξόδους για τις ενεργοποιήσεις της εισόδου και των 2 εξόδων του Register File, μια έξοδο Reset για το μηδενισμό των καταχωρητών του επεξεργαστή αλλά και αυτών του Register File, μια έξοδο για την εξαγωγή της άμεσης τιμής όταν πρόκειται για άμεση εντολή ή της τιμής διεύθυνσης όταν πρόκειται για εντολή μεταφοράς δεδομένων, μια έξοδο για την εξαγωγή της ποσότητας ολίσθησης (shamt) όταν πρόκειται για εντολή ολίσθησης, 2 εξόδους για τον έλεγχο των σημάτων επιλογής των πολυπλεκτών του επεξεργαστή και τέλος, μια έξοδο για την επιλογή εγγραφής ή ανάγνωσης της μνήμης RAM, ανάλογα με την περίπτωση.

Μετά το πρώτο κεφάλαιο της παρούσας εργασίας που αποτελεί μια σύντομη εισαγωγή, το δεύτερο αναφέρεται στην ψηφιακή σχεδίαση των κυκλωμάτων του επεξεργαστή και περιέχει την περιγραφή αυτών, καθώς και τα ψηφιακά σχέδια και τα block διαγράμματά τους. Αρχικά, το κεφάλαιο ξεκινάει με την περιγραφή της λειτουργίας της ALU και με το κύκλωμά της. Στη συνέχεια, περιγράφονται ο αθροιστής αφαιρέτης και το κύκλωμα ολίσθησης της Αριθμητικής και Λογικής Μονάδας και περιέχονται τα Block διαγράμματα του καθενός. Τέλος, γίνεται μια περιγραφή των υπόλοιπων μονάδων της ALU. Στη συνέχεια του κεφαλαίου, περιγράφεται η λειτουργία του Register File και περιέχεται το κύκλωμά του. Μετά, έχουμε την περιγραφή της λειτουργίας των τριών μονάδων από τις οποίες αποτελείται το Αρχείο Καταχωρητών και τα block διαγράμματά τους. Το δεύτερο κεφάλαιο συνεχίζεται με την περιγραφή της λειτουργίας της Μονάδας Ελέγχου. Μετά την περιγραφή της, περιέχεται το διάγραμμα καταστάσεων της. Τελειώνοντας το δεύτερο κεφάλαιο της παρούσας εργασίας, έχουμε την περιγραφή της λειτουργίας του κυκλώματος του επεξεργαστή, καθώς και ένα σχήμα με το κύκλωμα αυτό.

Το τρίτο κεφάλαιο της παρούσας εργασίας, περιέχει τον κώδικα του επεξεργαστή και τις επεξηγήσεις αυτού. Αρχικά, έχουμε τον κώδικα της ALU και των μονάδων της. Στην συνέχεια, υπάρχει ο κώδικας του Register File και των μονάδων από τις οποίες αποτελείται. Μετά, το κεφάλαιο συνεχίζεται με τον κώδικα της Μονάδας Ελέγχου και τελειώνει με τον κώδικα του κυκλώματος του επεξεργαστή.

Το τέταρτο κεφάλαιο της εργασίας, περιέχει τις προσομοιώσεις του κώδικα και αιτιολογείται πως βλέπουμε μέσω αυτών ότι ο κώδικας δουλεύει σωστά. Αρχικά, έχουμε τις

προσομοιώσεις της ALU και των μονάδων της. Στη συνέχεια, έχουμε τις προσομοιώσεις του Register File και των μονάδων του και τέλος, ακολουθεί η προσομοίωση της Μονάδας Ελέγχου.

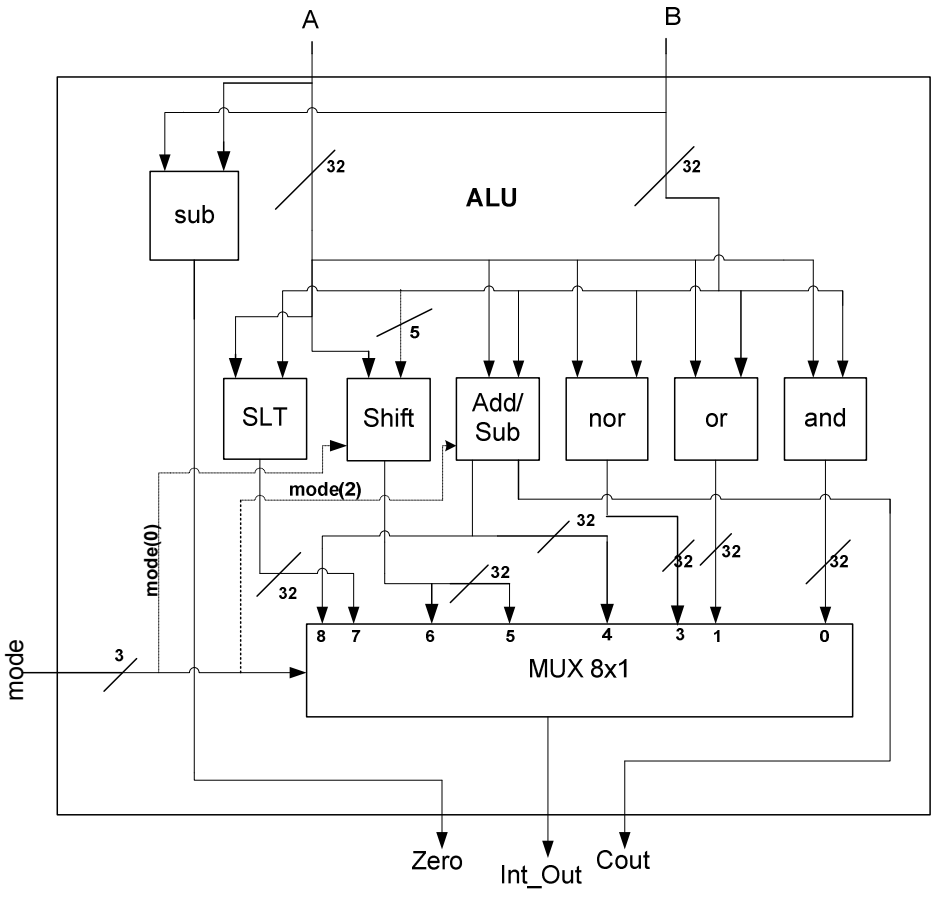
Τελειώνοντας την εργασία, υπάρχει ένα τελευταίο κεφάλαιο, με τα συμπεράσματα και το πώς μπορεί η σχεδίαση να βελτιστοποιηθεί.

Ασφαλώς, στο τέλος, υπάρχει βιβλιογραφία με τα βιβλία και τους ιστοτόπους που χρησιμοποιήθηκαν κατά την εκπόνηση της παρούσας πτυχιακής εργασίας.

2. Μελέτη

2.1 Αριθμητική και Λογική Μονάδα

Η αριθμητική και λογική μονάδα (ALU), είναι μια μονάδα του επεξεργαστή στην οποία εκτελούνται όλες οι αριθμητικές και λογικές πράξεις, όπως η πρόσθεση, η αφαίρεση, οι λογικές πράξεις and, or, nor, οι ολισθήσεις, οι συγκρίσεις κ.α.. Γενικά μια ALU μπορεί να εκτελέσει πράξεις με ακέραιους και με πραγματικούς αριθμούς. Στην παρούσα εργασία, όμως, θα ασχοληθούμε μόνο με ακέραιους αριθμούς. Η αριθμητική και λογική μονάδα που υλοποιήθηκε έχει 3 εισόδους και 3 εξόδους. Από τις 2 εισόδους των 32 bit παίρνει τους δυο αριθμούς που θα χρησιμοποιήσει στις πράξεις, για παράδειγμα αν πρόκειται για την πράξη A+B στη μία είσοδο παίρνει το A και στην άλλη το B. Από την άλλη είσοδο παίρνει μια κωδικοποίηση η οποία δείχνει την πράξη που θα εκτελέσει η ALU. Τέλος, στη μια έξοδο των 32 bit βγάζει το αποτέλεσμα της πράξης, ενώ στις άλλες δύο του ενός bit το τυχόν κρατούμενο της πράξης και το Zero. Η έξοδος Zero μας δείχνει αν οι δυο αριθμοί A και B είναι ίσοι ή όχι, πράγμα το οποίο επιτυγχάνεται αφαιρώντας από την είσοδο A την είσοδο B (δηλαδή A-B). Όταν το αποτέλεσμα της αφαίρεσης είναι 0 τότε οι 2 αριθμοί είναι ίσοι και η έξοδος Zero γίνεται ‘1’, ενώ για κάθε αποτέλεσμα της αφαίρεσης διάφορο του μηδενός, η έξοδος Zero παραμένει ‘0’. Στο παρακάτω σχήμα φαίνεται το Block διάγραμμα της ALU, καθώς και το πώς διασυνδέονται εσωτερικά τα κυκλώματα που την αποτελούν:



Σχήμα 2.1: ALU 32 bit.

Μέσα στην ALU, οι είσοδοι A και B συνδέονται με όλα τα επιμέρους κυκλώματά της και υπάρχει ένας πολυπλέκτης, ο οποίος ενεργοποιεί το κατάλληλο κύκλωμα, ανάλογα με το σήμα επιλογής του, και το βγάζει στην έξοδο Int_out. Στον παρακάτω πίνακα φαίνονται οι πράξεις που εκτελεί η Αριθμητική και Λογική Μονάδα και η κωδικοποίηση της κάθε μίας:

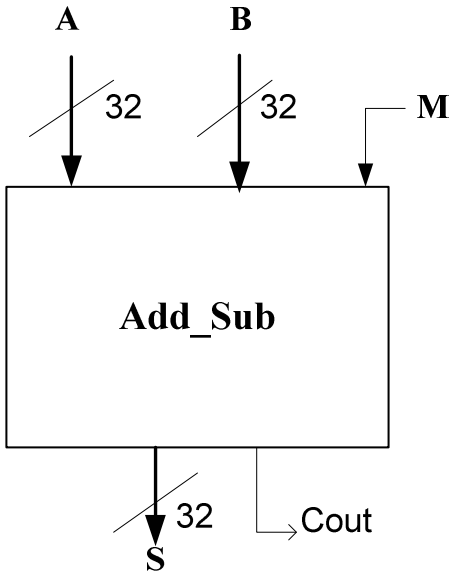
mode	Int_out
000	and A , B
001	or A , B
010	nor A , B
011	add A , B
100	srl A , B(4 down to 0)
101	sll A , B(4 down to 0)
110	slt A , B
111	sub A , B

Πίνακας 2.1: Κωδικοποίηση των λειτουργιών της ALU.

2.1.1 Ο αθροιστής – αφαιρέτης της ALU

Ένας αθροιστής, είναι ένα ψηφιακό κύκλωμα το οποίο εκτελεί την πρόσθεση μεταξύ δύο αριθμών. Ο αθροιστής - αφαιρέτης, έχει ένα σήμα επιλογής, ανάλογα με το οποίο εκτελεί την πρόσθεση ή την αφαίρεση μεταξύ δύο αριθμών. Αποτελείται από τόσους πλήρεις αθροιστές, όσο είναι το μέγεθος σε bit των εισόδων του A και B, οι οποίοι συνδέονται μεταξύ τους ως εξής: Από τις εισόδους A και B των αθροιστών δημιουργούνται οι είσοδοι A και B του αθροιστή - αφαιρέτη (Οι είσοδοι A και B του πρώτου αθροιστή λαμβάνουν το A(0) και B(0) του αθροιστή - αφαιρέτη, η είσοδοι A και B του δεύτερου λαμβάνουν το A(1) και B(1) του αθροιστή – αφαιρέτη κ.ο.κ.). Από τις εξόδους των αθροιστών δημιουργείται η έξοδος του αθροιστή – αφαιρέτη με τον ίδιο τρόπο όπως και με τις εισόδους. Η έξοδος κρατουμένου του κάθε αθροιστή συνδέεται με την είσοδο κρατουμένου του επόμενου αθροιστή (Η έξοδος κρατουμένου του πρώτου συνδέεται με την είσοδο κρατουμένου του δεύτερου κ.ο.κ.). Εξαιρούνται η είσοδος κρατουμένου του πρώτου αθροιστή, που αποτελεί την είσοδο κρατουμένου του αθροιστή – αφαιρέτη, και η έξοδος κρατουμένου του τελευταίου αθροιστή που αποτελεί την έξοδο κρατουμένου του αθροιστή αφαιρέτη. Η είσοδος B του αθροιστή αφαιρέτη συνδέεται με ένα δικτύωμα xor. Το δικτύωμα αυτό στη μία είσοδό του έχει τον αριθμό εισόδου και στην άλλη παίρνει το σήμα επιλογής του αθροιστή – αφαιρέτη. Όταν το σήμα επιλογής είναι ‘0’, τότε το δικτύωμα xor περνάει τον αριθμό ως έχει (A XOR ‘0’ = A). Στην αντίθετη περίπτωση, όταν δηλαδή το σήμα επιλογής είναι ‘1’, το δικτύωμα xor δημιουργεί σε συνδυασμό με το σήμα επιλογής το «συμπλήρωμα ως προς 2» του αριθμού (A XOR ‘1’ = A΄ συν την μονάδα στο αρχικό κρατούμενο) και έτσι ο αθροιστής γίνεται αφαιρέτης, καθώς εκτελεί αφαίρεση [(A + (-B)) = (A – B)]. Στην παρούσα εργασία,

υλοποιήθηκε ένας αθροιστής – αφαιρέτης στα 32 bit, ο οποίος εκτελεί πρόσθεση ή αφαίρεση μεταξύ 2 αριθμών των 32 bit. Έχει 2 εισόδους των 32 bit από τις οποίες παίρνει τους αριθμούς για την εκτέλεση της πράξης, και άλλη μια είσοδο του ενός bit από την οποία επιλέγεται αν θα γίνει πρόσθεση ή αφαίρεση. Όσον αφορά τις εξόδους του, έχει μια έξοδο των 32 bit για το αποτέλεσμα της πράξης και άλλη μία του ενός bit από την οποία βγαίνει το κρατούμενο της πράξης, εφόσον υπάρχει. Ο αθροιστής αφαιρέτης ενεργοποιείται από την ALU όταν τα bit 1 και 0 του mode είναι "11" και από το bit 2 του mode επιλέγεται αν θα εκτελεστεί αφαίρεση ή πρόσθεση. Για πρόσθεση το mode(2) πρέπει να είναι '0', ενώ για αφαίρεση πρέπει να είναι '1'. Στο παρακάτω σχήμα φαίνεται το block διάγραμμα του αθροιστή – αφαιρέτη:



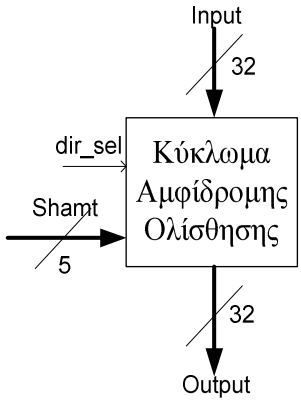
Σχήμα 2.2 : Αθροιστής αφαιρέτης των 32 bit.

2.1.2 Το κύκλωμα ολίσθησης της ALU

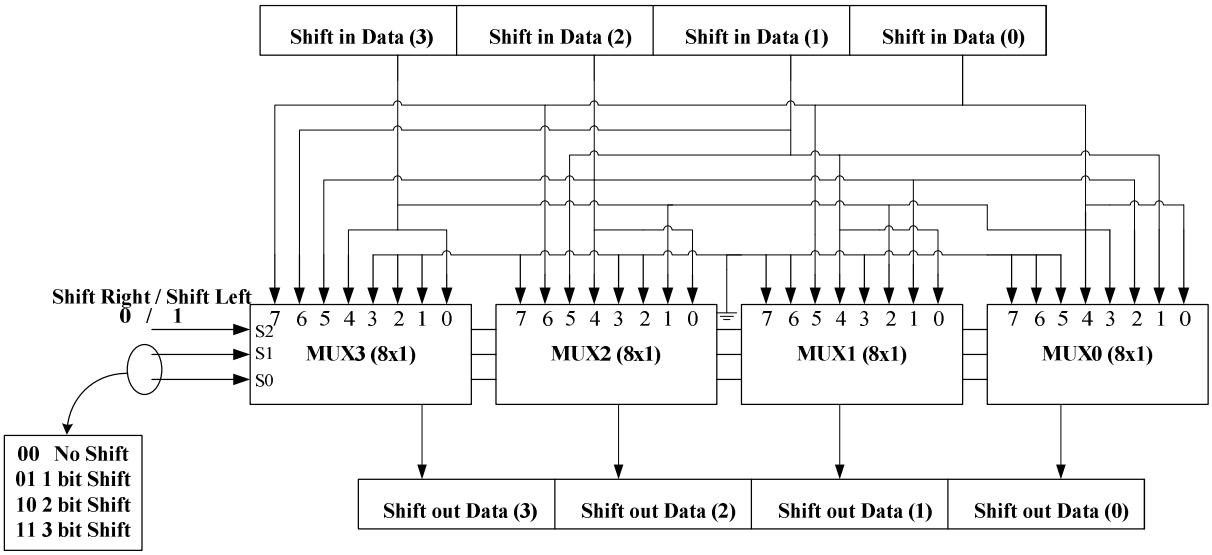
Υπάρχουν πολλοί τρόποι για να υλοποιήσει κανείς την ολίσθηση σε μια αριθμητική και λογική μονάδα. Μπορεί να χρησιμοποιηθεί καταχωρητής ολίσθησης σε συνδυασμό με ένα μετρητή. Ο καταχωρητής θα φορτώνει παράλληλα τον αριθμό που πρέπει να ολισθήσει και θα εκτελεί την ολίσθηση μέχρι ο μετρητής, στον οποίο έχουμε φορτώσει παράλληλα τον αριθμό που μας λέει πόσες ολισθήσεις θα γίνουν (Shift Amount ή shamt), να φτάσει στην τιμή '0'. Στην παρούσα εργασία, χρησιμοποιήθηκε μια άλλη λύση για την ολίσθηση. Χρησιμοποιήθηκαν πολυπλέκτες, στις εισόδους των οποίων συνδέθηκαν κατάλληλα τα ψηφία του αριθμού που θα ολισθαίνει, έτσι ώστε να βγάζουν στην έξοδό τους τον αριθμό ολισθημένο όσες φορές λέει το κοινό σήμα επιλογής τους.

Το κύκλωμα ολίσθησης που υλοποιήθηκε έχει 32 πολυπλέκτες 64x1 και οι συνδέσεις είναι ως εξής: Ο πρώτος πολυπλέκτης έχει στις 32 πρώτες εισόδους τον αριθμό χωρίς ολίσθηση και στις 32 υπόλοιπες τον αριθμό ολισθημένο κατά 31 θέσεις αριστερά. Ο δεύτερος πολυπλέκτης έχει στις 32 πρώτες εισόδους του τον αριθμό ολισθημένο κατά 1 θέση αριστερά και στις υπόλοιπες τον αριθμό ολισθημένο κατά 30 θέσεις δεξιά. Έτσι συνεχίζουμε μέχρι τον τελευταίο πολυπλέκτη, τον 32ο, ο οποίος έχει στις 32 πρώτες εισόδους του τον αριθμό ολισθημένο κατά 31 θέσεις αριστερά και στις υπόλοιπες τον αριθμό αμετάβλητο. Με αυτό

τον τρόπο έχουμε μια γρήγορη αμφίδρομη ολίσθηση, όπου η επιλογή δεξιάς – αριστερής ολίσθησης γίνεται με το σημαντικότερο ψηφίο του κοινού σήματος επιλογής των πολυπλεκτών. Στα παρακάτω σχήματα φαίνονται το block διάγραμμα του κυκλώματος ολίσθησης των 32 bit και το εσωτερικό του κυκλώματος ολίσθησης, το οποίο περιέχεται στην παρούσα εργασία στα 4 bit χάριν συντομίας. Βάση αυτού, σχεδιάστηκε το διάγραμμα του κυκλώματος ολίσθησης του μικροεπεξεργαστή, το οποίο είναι στα 32 bit.



Σχήμα 2.3(i) : Κύκλωμα Ολίσθησης.



Σχήμα 2.3(ii) : Το εσωτερικό του Κυκλώματος Ολίσθησης στα 4 bit.

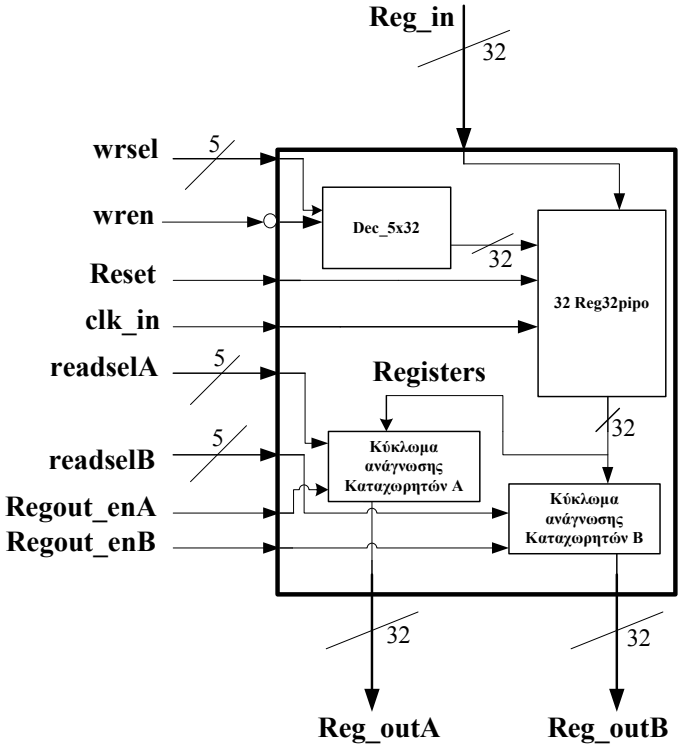
2.1.3 Τα υπόλοιπα στοιχεία της ALU

Εκτός από πρόσθεση, αφαίρεση και ολίσθηση, η ALU που υλοποιήθηκε εκτελεί και άλλες πράξεις. Η πρόσθεση, η αφαίρεση και η ολίσθηση, υλοποιήθηκαν με Structural σχεδίαση, ενώ οι υπόλοιπες πράξεις, οι λογικές πράξεις and, or και nor και η σύγκριση slt, υλοποιήθηκαν με Behavioral σχεδίαση. Όταν η ALU εκτελεί μια λογική πράξη, έχει στο εσωτερικό της ένα δικτύωμα πυλών, and, or και nor, το οποίο παίρνει στις εισόδους του τους 2 αριθμούς και βγάζει στην έξοδο το αποτέλεσμα της αντίστοιχης πράξης. Στην περίπτωση της σύγκρισης, υπάρχει ένας συγκριτής ο οποίος όταν η είσοδος A είναι μικρότερη από την είσοδο B βγάζει

‘1’, ενώ στην αντίθετη περίπτωση, όταν δηλαδή η είσοδος A είναι μεγαλύτερη ή ίση από την είσοδο B, βγάζει ‘0’. Η έξοδος του συγκριτή συνδέεται με το λιγότερο σημαντικό bit της εξόδου της ALU, ενώ τα υπόλοιπα 31 bit της παίρνουν την τιμή ‘0’ όταν εκτελείται σύγκριση.

2.2 Το κομμάτι των καταχωρητών.

Για να μπορεί ο επεξεργαστής να εκτελεί πράξεις, πρέπει από κάποιο σημείο να αντλεί δεδομένα και σε κάποιο άλλο σημείο να τα αποθηκεύει. Αυτά τα σημεία είναι οι καταχωρητές. Στην παρούσα εργασία υλοποιήθηκε ένας στοιχειώδης επεξεργαστής βασισμένος στον MIPS, οπότε το κομμάτι των καταχωρητών του (Register File) έχει 32 καταχωρητές παράλληλης εισόδου και παράλληλης εξόδου δεδομένων των 32 bit.



Σχήμα 2.4 : Το Register File και οι εσωτερικές συνδέσεις του.

Για να μπορεί να διαχειριστεί τους καταχωρητές ο επεξεργαστής, χρειάζεται ακόμη έναν αποκωδικοποιητή για να επιλέγει κάθε φορά σε ποιο καταχωρητή θα γράψει και ένα δικτύωμα με πολυπλέκτες που από το σήμα επιλογής τους θα επιλέγεται ποιανού καταχωρητή την έξοδο θα διαβάσει ο επεξεργαστής. Για να μπορεί ταυτόχρονα να διαβάζει 2 καταχωρητές και να γράφει έναν, έχει έναν αποκωδικοποιητή 5x32 για την επιλογή καταχωρητή εγγραφής και 2 δικτυώματα με 32 πολυπλέκτες 32x1 για την επιλογή των δύο καταχωρητών ανάγνωσης. Το Register File ως Component, έχει :

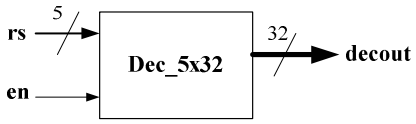
- 1 είσοδο των 32 bit για την εισαγωγή δεδομένων στους καταχωρητές
- 3 εισόδους των 5 bit για την επιλογή των 2 καταχωρητών από τους οποίους θα γίνει ανάγνωση δεδομένων και του καταχωρητή που θα γίνει εγγραφή δεδομένων

- 3 εισόδους του 1 bit οι οποίες θα ενεργοποιούν την επιλογή εγγραφής (τον αποκωδικοποιητή) και τις δυο επιλογές ανάγνωσης (τις εξόδους των πολυπλεκτών)
- 2 εισόδους του ενός bit, μια για το ρολόι και άλλη μία για το μηδενισμό (Reset) των καταχωρητών και
- 2 εξόδους των 32 bit για την έξοδο των δεδομένων των 2 καταχωρητών από τους οποίους γίνεται ανάγνωση.

Στο σχήμα 2.4 φαίνεται το Block διάγραμμα του Register File καθώς και οι διασυνδέσεις των κυκλωμάτων από τα οποία αποτελείται.

2.2.1 Εγγραφή καταχωρητών

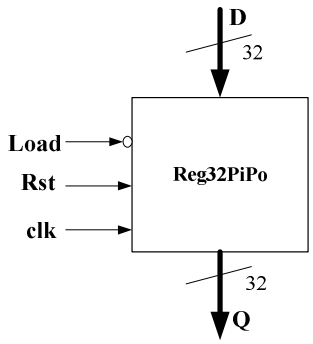
Όπως αναφέρθηκε και πιο πάνω, για να γίνει η επιλογή σε ποιόν από τους 32 καταχωρητές θα αποθηκευτούν τα δεδομένα εισόδου του Register File, χρησιμοποιείται ένας αποκωδικοποιητής 5x32. Έχει 1 είσοδο των 5 bit για την επιλογή του καταχωρητή, 1 είσοδο ενεργοποίησης του ενός bit και μια έξοδο των 32 bit. Αποκωδικοποιεί την είσοδο των 5 bits και βγάζει το κατάλληλο σήμα στην έξοδο των 32 bit, ώστε να ενεργοποιηθεί ο κατάλληλος καταχωρητής, μόνο όταν το σήμα ενεργοποίησης είναι ‘0’ (είναι ενεργό στο ‘0’). Στην αντίθετη περίπτωση δεν ενεργοποιεί κανέναν καταχωρητή. Οι καταχωρητές έχουν αριθμηθεί από το 1 μέχρι το 32, κι έτσι μετατρέποντας την είσοδο των 5 bits του αποκωδικοποιητή από το δυαδικό σύστημα αρίθμησης στο δεκαδικό, ενεργοποιείται ο καταχωρητής με την κατάλληλη αρίθμηση. Για παράδειγμα αν το σήμα των 5 bits είναι “00100” και το σήμα ενεργοποίησης του αποκωδικοποιητή είναι ‘0’, τότε το σήμα εξόδου του παίρνει την τιμή “11111111111111111111111111110111” (τα σήματα ενεργοποίησης των καταχωρητών είναι όλα ενεργά στο ‘0’) και έτσι ενεργοποιείται ο $4^{ος}$ καταχωρητής. Στην περίπτωση, όμως, που το σήμα ενεργοποίησης του αποκωδικοποιητή είναι ‘1’, όποια τιμή κι αν έχει το σήμα εισόδου των 5 bits, το σήμα εξόδου παίρνει την τιμή “11111111111111111111111111111111” έτσι ώστε όλοι οι καταχωρητές να είναι ανενεργοί. Στο παρακάτω σχήμα φαίνεται το block διάγραμμα του αποκωδικοποιητή:



Σχήμα 2.5 : Block διάγραμμα του Αποκωδικοποιητή

2.2.2 Καταχωρητής των 32 bit

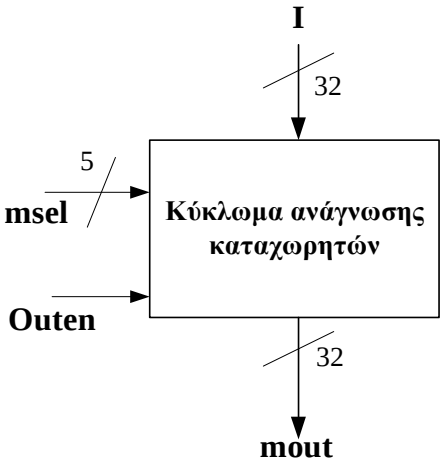
Ο μικροεπεξεργαστής που υλοποιήθηκε περιέχει μέσα στο Register File 32 καταχωρητές παράλληλης εισόδου και παράλληλης εξόδου των 32 bits. Κάθε καταχωρητής έχει μια είσοδο και μια έξοδο δεδομένων των 32 bits και άλλες 3 εισόδους του ενός bit : μια είσοδο για το ρολόι, μια για την ενεργοποίηση αποθήκευσης και μια τελευταία είσοδο για το μηδενισμό του καταχωρητή. Στο εσωτερικό του καταχωρητή, υπάρχουν πολυπλέκτες και flip – flop τύπου D, συνδεδεμένα έτσι ώστε όταν το σήμα ενεργοποίησης είναι ‘0’ και το Reset είναι επίσης ‘0’, να βγάζει στην έξοδο ότι είχε η είσοδος στον προηγούμενο παλμό ρολογιού. Στο παρακάτω σχήμα φαίνεται το block διάγραμμα ενός από τους καταχωρητές:



Σχήμα 2.6 : Καταχωρητής των 32 bit.

2.2.3 Ανάγνωση καταχωρητών

Για να μπορεί ο μικροεπεξεργαστής που υλοποιήθηκε στην παρούσα εργασία να κάνει ταυτόχρονη ανάγνωση 2 καταχωρητών, έχουμε συνδέσει τις εξόδους όλων των καταχωρητών σε 2 ίδια κυκλώματα 32 πολυπλεκτών 32x1 με τον παρακάτω τρόπο: Ο πρώτος πολυπλέκτης παίρνει στις εισόδους του το πρώτο λιγότερο σημαντικό bit (το bit '0') των εξόδων όλων των καταχωρητών. Ο δεύτερος πολυπλέκτης παίρνει στις εισόδους του το δεύτερο λιγότερο σημαντικό bit (το bit '1') των εξόδων όλων των καταχωρητών.



Σχήμα 2.7 : Κύκλωμα ανάγνωσης καταχωρητών.

Ο τρίτος καταχωρητής παίρνει το τρίτο κ.ο.κ., με τον 32^{ο} πολυπλέκτη να παίρνει στις εισόδους του το περισσότερο σημαντικό bit (το bit 31) της εξόδου όλων των καταχωρητών. Έτσι με το κοινό σήμα επιλογής των πολυπλεκτών επιλέγεται ο κατάλληλος καταχωρητής για ανάγνωση από 32 εξόδους των πολυπλεκτών. Επίσης, κάθε κύκλωμα πολυπλεκτών έχει από ένα σήμα ενεργοποίησης, που είναι ενεργό στο '1'. Τέλος, το κάθε ένα κύκλωμα ανάγνωσης καταχωρητών, έχει:

- 1 είσοδο όπου συνδέονται οι έξοδοι όλων των καταχωρητών
- 1 είσοδο των 5 bits για την επιλογή των καταχωρητών που θα διαβαστούν
- 1 είσοδο του ενός bit για την ενεργοποίηση της ανάγνωσης και τέλος

- 1 έξοδο των 32 bit για την έξοδο των δεδομένων ανάγνωσης

Στο σχήμα 2.7 φαίνεται το Block διάγραμμα του κυκλώματος ανάγνωσης καταχωρητών.

2.3 Η Μονάδα Ελέγχου

Για να λειτουργήσει ένας μικροεπεξεργαστής, χρειάζεται μια μονάδα, στην οποία θα γίνεται η διαχείριση όλων των σημάτων του. Αυτή η μονάδα ονομάζεται Μονάδα Ελέγχου (Control Unit ή για συντομία CU) και το μόνο που κάνει είναι να ελέγχει κάποιες εισόδους του μικροεπεξεργαστή και να βγάζει τα ανάλογα σήματα ελέγχου από τα οποία διαχειρίζεται τις υπόλοιπες μονάδες του επεξεργαστή. Η μονάδα ελέγχου που υλοποιήθηκε παίρνει ένα σήμα με την κωδικοποίηση της εντολής που πρέπει να εκτελέσει κάθε φορά ο επεξεργαστής και μετά τον έλεγχο και την αποκωδικοποίηση βγάζει ανάλογα με την εντολή τα κατάλληλα σήματα για να κάνουν την ανάλογη εργασία οι υπόλοιπες μονάδες του επεξεργαστή. Επειδή ο κώδικας μηχανής του επεξεργαστή που υλοποιήθηκε είναι βασισμένος στον κώδικα μηχανής του MIPS, η κωδικοποίηση που παίρνει ο επεξεργαστής για να δει την εντολή που θα εκτελέσει έχει χωριστεί σε κομμάτια ως εξής:

Οι εντολές που εκτελεί ο μικροεπεξεργαστής χωρίζονται σε 2 ομάδες: Στις εντολές τύπου R (από το Register που σημαίνει Καταχωρητής) και στις εντολές τύπου I (από το Immediate που σημαίνει Άμεσος). Αν πρόκειται για εντολές τύπου-R, η κωδικοποίηση χωρίζεται ως εξής:

- Τα πρώτα 6 bits της κωδικοποίησης, αποτελούν το κομμάτι op-code (operation code που σημαίνει κωδικός λειτουργίας) το οποίο μας προϊδεάζει για το ποια εντολή θα εκτελέσει ο επεξεργαστής.

- Οι 2 επόμενες ομάδες των 5 bits της κωδικοποίησης, αποτελούν τα κομμάτια rs και rt, τα οποία μας δείχνουν τους 2 καταχωρητές από τους οποίους θα αντλήσει δεδομένα η ALU για να εκτελέσει την πράξη της εντολής.

- Τα επόμενα 5 bits της κωδικοποίησης, αποτελούν το κομμάτι rd, το οποίο μας δείχνει τον καταχωρητή στον οποίο θα γίνει η αποθήκευση του αποτελέσματος της πράξης που εκτέλεσε η ALU.

- Τα επόμενα 5 bits της κωδικοποίησης, αποτελούν το κομμάτι shamt (Shift Amount που σημαίνει Ποσότητα ολίσθησης), το οποίο χρησιμοποιείται μόνο όταν εκτελείται εντολή ολίσθησης και μας δείχνει πόσες θέσεις ολίσθησης θα εκτελέσει η ALU.

- Τα τελευταία 6 bits της κωδικοποίησης, αποτελούν το κομμάτι funct (function code που σημαίνει κωδικός συνάρτησης) και σε συνδυασμό με το op-code μας δείχνουν επακριβώς την εντολή που θα εκτελέσει ο επεξεργαστής.

Στην άλλη περίπτωση, όταν δηλαδή πρόκειται για εντολές τύπου-I, η κωδικοποίηση χωρίζεται ως εξής:

- Τα πρώτα 6 bits της κωδικοποίησης, αποτελούν το κομμάτι op-code (operation code που σημαίνει κωδικός λειτουργίας) το οποίο μας δείχνει ποια εντολή θα εκτελέσει ο επεξεργαστής.

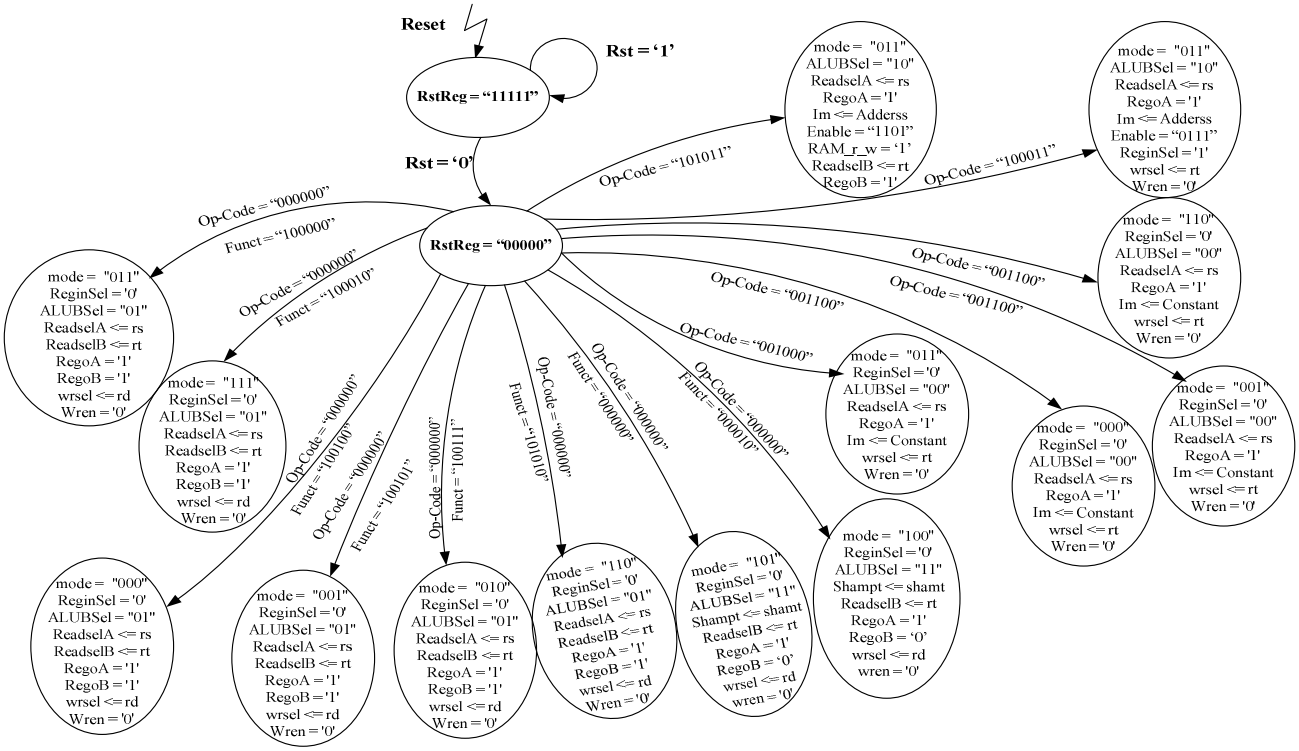
- Τα επόμενα 5 bits της κωδικοποίησης, αποτελούν το κομμάτι rs, το οποίο μας δείχνει τον καταχωρητή από τον οποίο θα φορτωθούν τα δεδομένα για την εκτέλεση της εντολής, όπως θα γινόταν αν είχαμε εντολή τύπου-R.

- Τα επόμενα 5 bits της κωδικοποίησης, αποτελούν το κομμάτι rt, το οποίο αυτή τη φορά μας δείχνει τον καταχωρητή στον οποίο θα γίνει η αποθήκευση δεδομένων μετά την εκτέλεση της εντολής.

- Τα τελευταία 16 bits της κωδικοποίησης, αποτελούν το κομμάτι address/constant (διεύθυνση/σταθερά) στο οποίο υπάρχει η σταθερά, όταν πρόκειται για άμεση εντολή, ενώ όταν πρόκειται για εντολή φόρτωσης δεδομένων εδώ υπάρχει η διεύθυνση της μνήμης που θα αποθηκευτούν ή θα φορτωθούν τα δεδομένα.

Η μονάδα ελέγχου, ελέγχει αρχικά τα 6 πρώτα bit της κωδικοποίησης εντολής. Επειδή οι εντολές τύπου-I που εκτελεί ο επεξεργαστής που υλοποιήθηκε έχουν όλες στο κομμάτι αυτό τιμή διαφορετική από το “000000” και μόνο όλες οι εντολές τύπου-R έχουν σε αυτό το κομμάτι την τιμή “000000”, όταν βλέπει την τιμή αυτή η μονάδα ελέγχου καταλαβαίνει πως η εντολή είναι τύπου-R. Έτσι, όταν η εντολή είναι τύπου-R, ελέγχει και τα 6 τελευταία bits της κωδικοποίησης και βλέπει ακριβώς ποια εντολή θα εκτελεστεί. Δίνει στην ALU το κατάλληλο mode, επιλέγει τους κατάλληλους καταχωρητές για ανάγνωση και στη συνέχεια επιλέγει τον κατάλληλο καταχωρητή για να αποθηκευτεί το αποτέλεσμα της πράξης. Στην περίπτωση που πρόκειται για εντολή τύπου-I, και συγκεκριμένα για τις άμεσες εντολές, μεταφέρονται στην είσοδο A της ALU τα δεδομένα του κατάλληλου καταχωρητή, ενώ στην άλλη είσοδο της ALU, μεταφέρεται η άμεση τιμή. Στη συνέχεια, επιλέγει η μονάδα ελέγχου τον κατάλληλο καταχωρητή για την αποθήκευση του αποτελέσματος της πράξης. Στην άλλη περίπτωση, όταν δηλαδή πρόκειται για εντολή μεταφοράς δεδομένων, η μονάδα ελέγχου κάνει τα εξής:

Αρχικά, δείχνει στην ALU ότι πρέπει να γίνει πρόσθεση (για τον υπολογισμό της διεύθυνσης της μνήμης RAM). Στη συνέχεια, μεταφέρει στην είσοδο B την τιμή της διεύθυνσης ολισθημένη κατά 2 θέσεις αριστερά, που σημαίνει ότι έχει τετραπλασιασθεί. Στην είσοδο A της ALU φορτώνονται τα δεδομένα του καταχωρητή βάσης και γίνεται η πρόσθεση για να υπολογισθεί η διεύθυνση. Στη συνέχεια, ενεργοποιεί τον καταχωρητή που θα περάσει τη διεύθυνση που υπολόγισε η ALU στη μνήμη. Όταν πρόκειται για φόρτωση δεδομένων από τη RAM (εντολή lw), ενεργοποιείται ο καταχωρητής εισόδου δεδομένων από τη μνήμη, καθώς και ο καταχωρητής στον οποίο θα αποθηκευτούν τα δεδομένα. Τέλος, όταν πρόκειται για αποθήκευση δεδομένων στη RAM (εντολή sw), ενεργοποιείται ο καταχωρητής εξόδου δεδομένων προς τη μνήμη, καθώς και ο καταχωρητής από τον οποίο θα φορτωθούν τα δεδομένα που θα καταλήξουν στη μνήμη. Στο παρακάτω σχήμα, φαίνεται το διάγραμμα καταστάσεων της μονάδας ελέγχου:



Σχήμα 2.8: Το διάγραμμα καταστάσεων της Μονάδας Ελέγχου.

Η Μονάδα ελέγχου που υλοποιήθηκε έχει τις εξής εισόδους και εξόδους:

- 1 είσοδο των 32 bit απ' όπου παίρνει την κωδικοποίηση εντολής.
- 2 εισόδους του ενός bit, απ' όπου παίρνει τις εξόδους Cout (έξοδος Κρατουμένου) και Zero της ALU.
- Άλλες 2 εισόδους του ενός bit, που από τη μια παίρνει το Reset και από την άλλη τους παλμούς ρολογιού.
- 1 έξοδο των $4^{ων}$ bit, απ' όπου βγάζει τα 4 σήματα ενεργοποίησης των καταχωρητών που υπάρχουν στο συνολικό κύκλωμα του επεξεργαστή.
- 1 έξοδο των 5 bit, από την οποία βγαίνουν τα σήματα μηδενισμού (Reset) των καταχωρητών.
- 1 έξοδο των $3^{ων}$ bit, απ' όπου στέλνει το σήμα mode στην ALU για να της δείξει τι πράξη θα εκτελέσει.
- 3 εξόδους του ενός bit, που από τις 2 πρώτες στέλνει τα σήματα ενεργοποίησης των εξόδων A και B του Register File, ενώ από την τρίτη στέλνει το σήμα ενεργοποίησης εγγραφής του Register File.
- 3 εξόδους των 5 bits, για την επιλογή του καταχωρητή εγγραφής και των 2 καταχωρητών ανάγνωσης.

- 1 έξοδο των 16 bits, απ' όπου βγάζει την άμεση τιμή όταν πρόκειται για άμεση εντολή ή την τιμή για τον υπολογισμό της διεύθυνσης μνήμης όταν πρόκειται για εντολή μεταφοράς δεδομένων.

- 1 έξοδο των 5 bits, απ' όπου βγαίνει το shamt (Ποσότητα ολίσθησης) για να δείξει στην ALU πόσες θέσεις ολίσθηση θα εκτελέσει όταν πρόκειται για εντολή ολίσθησης.

- 1 εξόδους του ενός bit, από την οποία ενεργοποιεί την ανάγνωση ή την εγγραφή της μνήμης RAM.

- Τέλος, η μονάδα ελέγχου έχει άλλες 2 εξόδους, μια του ενός κι άλλη μια των 2 bits, οι οποίες ελέγχουν το σήμα επιλογής των 2 πολυπλεκτών που υπάρχουν στο συνολικό κύκλωμα του επεξεργαστή.

2.4 Το κύκλωμα του μικροεπεξεργαστή

Ένας μικροεπεξεργαστής για να λειτουργήσει, χρειάζεται εκτός από τις παραπάνω μονάδες και κάποια ακόμη στοιχεία. Για τη σωστή διαχείριση των δεδομένων που εισέρχονται και εξέρχονται από τους διάφορους κόμβους χρειάζονται καταχωρητές. Επίσης, σε ορισμένες περιπτώσεις, η είσοδος μιας μονάδας του επεξεργαστή πρέπει να συνδεθεί με πολλά σημεία. Για να γίνει η επιλογή ποιο σημείο θα ενεργοποιείται κάθε φορά, χρειάζεται και πολυπλέκτες. Έτσι, ο μικροεπεξεργαστής που υλοποιήθηκε στην παρούσα εργασία, εκτός από την Αριθμητική και Λογική Μονάδα (ALU), το Αρχείο Καταχωρητών (Register File) και τη Μονάδα Ελέγχου (Control Unit), περιέχει ακόμα 4 καταχωρητές και 2 πολυπλέκτες.

Οι τρεις πρώτοι καταχωρητές είναι τοποθετημένοι στην είσοδο κωδικοποίησης εντολής, η οποία είναι ταυτόχρονα και έξοδος δεδομένων προ τη μνήμη. Ο πρώτος καταχωρητής ενεργοποιείται για να εισάγει στη μονάδα ελέγχου την κωδικοποίηση εντολής, ενώ ο δεύτερος και ο τρίτος, ο ένας για να εισάγει και ο άλλος για να εξάγει δεδομένα από και προς το αρχείο καταχωρητών όταν πρόκειται για εντολή μεταφοράς δεδομένων. Ο τέταρτος καταχωρητής χρησιμοποιείται για την εξαγωγή της διεύθυνσης που υπολογίζει η ALU όταν πρόκειται για εντολή μεταφοράς δεδομένων.

Όσον αφορά τους πολυπλέκτες, ο ένας πολυπλέκτης είναι 2x1 (η κάθε μια από τις 2 εισόδους του και η έξοδός του είναι των 32 bits) και βρίσκεται στην είσοδο του Register File. Αυτό συμβαίνει επειδή ο καταχωρητής αποθήκευσης δεδομένων αποθηκεύει δεδομένα από την έξοδο της ALU όταν πρόκειται για άμεση εντολή ή για εντολή τύπου-R, ενώ όταν πρόκειται για την εντολή sw, αποθηκεύει τα δεδομένα που εισέρχονται από τη μνήμη RAM. Ο άλλος πολυπλέκτης είναι 4x1 (και εδώ όλες οι είσοδοι και η έξοδός του είναι στα 32 bits) και βρίσκεται στην είσοδο Β της ALU. Η είσοδος Β της ALU μπορεί να πάρει 4 διαφορετικές τιμές, ανάλογα με την εντολή που εκτελείται. Στην πρώτη περίπτωση παίρνει την τιμή της εξόδου Β του Register File, εφόσον πρόκειται για εντολή τύπου-R. Στη δεύτερη περίπτωση παίρνει την άμεση τιμή όταν πρόκειται για άμεση εντολή. Στην τρίτη περίπτωση παίρνει την τιμή της διεύθυνσης τετραπλασιασμένη (ολισθημένη κατά 2 θέσεις αριστερά, πράγμα που επιτεύχθηκε γειώνοντας τα 2 πρώτα bit της τρίτης εισόδου του πολυπλέκτη, συνδέοντας την τιμή της διεύθυνσης όπως έρχεται από τη μονάδα ελέγχου στα επόμενα 16 bit και γειώνοντας τα υπόλοιπα bit της εισόδου του πολυπλέκτη), όταν πρόκειται για εντολή μεταφοράς

δεδομένων. Τέλος, στην τέταρτη περίπτωση παίρνει το shamt (ποσότητα ολίσθησης), όταν πρόκειται για εντολή ολίσθησης. Στις επόμενες παραγράφους θα εξηγήσω πως εκτελεί ο επεξεργαστής 3 παραδείγματα εντολών.

2.4.1 Επεξήγηση λειτουργίας επεξεργαστή

Στην παράγραφο αυτή θα επεξηγηθούν 3 από τις εντολές που εκτελεί ο επεξεργαστής που υλοποιήθηκε.

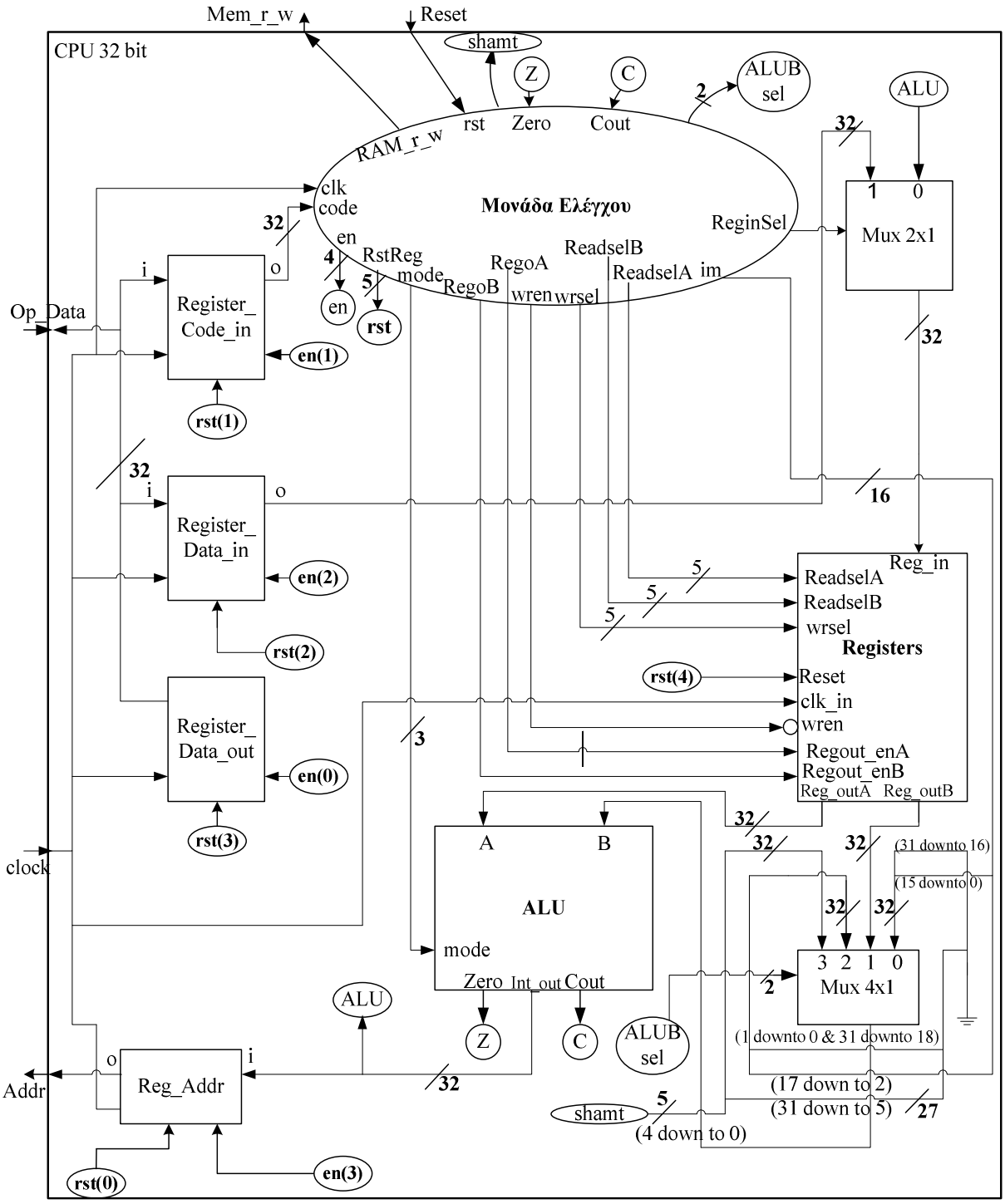
Έστω ότι στην είσοδο κωδικοποίησης ο επεξεργαστής δέχεται την τιμή "00000000001000100001100000100111". Αρχικά, η μονάδα ελέγχου βγάζει από το σήμα Enable την τιμή "0100" για να ενεργοποιηθεί ο καταχωρητής από τον οποίο θα περάσει το σήμα κωδικοποίησης εντολής. Αυτό γίνεται σε όλες τις εντολές. Στη συνέχεια η μονάδα ελέγχου αφού πάρει την κωδικοποίηση εντολής την αποκωδικοποιεί για να δει τι εντολή θα εκτελεστεί. Από τα πρώτα 6 bit (πεδίο op-code) καταλαβαίνει ότι πρόκειται για εντολή τύπου-R και από τα τελευταία 6 bit (πεδίο function) καταλαβαίνει ότι πρόκειται για την nor. Έτσι, δίνει στο σήμα mode της ALU την τιμή "010" για να εκτελέσει τη λογική πράξη nor. Στη συνέχεια, δίνει στο σήμα επιλογής του πολυπλέκτη 2x1 που βρίσκεται στην είσοδο δεδομένων του Register File την τιμή '0' και στο σήμα επιλογής του άλλου πολυπλέκτη 4x1 που βρίσκεται στην είσοδο B της ALU την τιμή '01'. Έτσι, ο καταχωρητής αποθήκευσης θα πάρει τιμή από την έξοδο της ALU και η είσοδος B της ALU θα πάρει τιμή από την έξοδο B του Register File. Στη συνέχεια, μεταφέρει το rs (τα bit 21 ως 25 της κωδικοποίησης εντολής) στην είσοδο ReadselA (η είσοδος επιλογής του πρώτου καταχωρητή ανάγνωσης) του Register File, το rt (τα bit 16 ως 20 της κωδικοποίησης εντολής) στην είσοδο ReadselB (η είσοδος επιλογής του δεύτερου καταχωρητή ανάγνωσης) του Register File και ενεργοποιεί τις δυο εξόδους A και B του Register File, οι οποίες συνδέονται με τις δυο εισόδους A και B της ALU, δίνοντάς τους την τιμή '1'. Τέλος, μεταφέρει το rd (τα bit 11 ως 15 της κωδικοποίησης εντολής) στην είσοδο wrsel (η είσοδος επιλογής του καταχωρητή εγγραφής) του Register File και ενεργοποιεί το σήμα wren (σήμα ενεργοποίησης εγγραφής) του Register File, δίνοντάς του την τιμή '0'. Έτσι, το αποτέλεσμα της πράξης που εκτέλεσε η ALU θα μεταφερθεί στον καταχωρητή αποθήκευσης για να το αποθηκεύσει. Σε αυτό το σημείο τελειώνει η εντολή, η οποία λέει στον επεξεργαστή να εκτελέσει την πράξη not μεταξύ του καταχωρητή 1 και του καταχωρητή 2 και το αποτέλεσμα να αποθηκευτεί στον καταχωρητή 3.

Έστω, τώρα, ότι στην είσοδο κωδικοποίησης ο επεξεργαστής δέχεται την τιμή "00100000010000110000000000001010". Αρχικά, όπως και στο προηγούμενο παράδειγμα, η μονάδα ελέγχου βγάζει από το σήμα Enable την τιμή "0100" για να ενεργοποιηθεί ο καταχωρητής από τον οποίο θα περάσει το σήμα κωδικοποίησης εντολής. Στη συνέχεια η μονάδα ελέγχου αφού πάρει την κωδικοποίηση εντολής την αποκωδικοποιεί για να δει τι εντολή θα εκτελεστεί. Από τα πρώτα 6 bit (πεδίο op-code) καταλαβαίνει ότι πρόκειται για εντολή τύπου-I και συγκεκριμένα για την addi. Έτσι, δίνει στο σήμα mode της ALU την τιμή "011" για να εκτελέσει πρόσθεση. Στη συνέχεια, δίνει στο σήμα επιλογής του πολυπλέκτη 2x1 που βρίσκεται στην είσοδο δεδομένων του Register File την τιμή '0' και στο σήμα επιλογής του άλλου πολυπλέκτη 4x1 που βρίσκεται στην είσοδο B της ALU την τιμή '00'. Έτσι, ο καταχωρητής αποθήκευσης θα πάρει τιμή από την έξοδο της ALU και η είσοδος B της ALU θα πάρει τιμή από την έξοδο im της μονάδας ελέγχου, η οποία θα βγάλει την άμεση

τιμή. Στη συνέχεια, μεταφέρει το rs (τα bit 21 ως 25 της κωδικοποίησης εντολής) στην είσοδο ReadselA (η είσοδος επιλογής του πρώτου καταχωρητή ανάγνωσης) του Register File, το immediate (τα bit 0 ως 15 της κωδικοποίησης εντολής) στην έξοδό της im και ενεργοποιεί την έξοδο A του Register File, οι οποία συνδέεται με την είσοδο A της ALU, δίνοντάς της την τιμή '1'. Τέλος, μεταφέρει το rt (τα bit 16 ως 20 της κωδικοποίησης εντολής) στην είσοδο wrsel (η είσοδος επιλογής του καταχωρητή εγγραφής) του Register File και ενεργοποιεί το σήμα wren (σήμα ενεργοποίησης εγγραφής) του Register File, δίνοντάς του την τιμή '0'. Έτσι, το αποτέλεσμα της πράξης που εκτέλεσε η ALU θα μεταφερθεί στον καταχωρητή αποθήκευσης για να το αποθηκεύσει. Σε αυτό το σημείο τελειώνει η εντολή, η οποία λέει στον επεξεργαστή να προσθέσει στον καταχωρητή 2 τον αριθμό 10 και το αποτέλεσμα να αποθηκευτεί στον καταχωρητή 3.

Τέλος, ας υποθέσουμε ότι στην είσοδο κωδικοποίησης ο επεξεργαστής δέχεται την τιμή "10101100010000110000000000000100". Αρχικά, όπως και στο προηγούμενο παράδειγμα, η μονάδα ελέγχου βγάζει από το σήμα Enable την τιμή "0100" για να ενεργοποιηθεί ο καταχωρητής από τον οποίο θα περάσει το σήμα κωδικοποίησης εντολής. Στη συνέχεια η μονάδα ελέγχου αφού πάρει την κωδικοποίηση εντολής την αποκωδικοποιεί για να δει τι εντολή θα εκτελεστεί. Από τα πρώτα 6 bit (πεδίο op-code) καταλαβαίνει ότι πρόκειται για εντολή τύπου-I και συγκεκριμένα για την sw. Έτσι, δίνει στο σήμα mode της ALU την τιμή "011" για να εκτελέσει πρόσθεση, από την οποία θα υπολογιστεί η διεύθυνση μνήμης στην οποία θα αποθηκευτούν τα δεδομένα. Στη συνέχεια, δίνει στο σήμα επιλογής του πολυπλέκτη 4x1 που βρίσκεται στην είσοδο B της ALU την τιμή '10'. Έτσι, η είσοδος B της ALU θα πάρει τιμή από την έξοδο im της μονάδας ελέγχου, η οποία θα βγάλει την τιμή της διεύθυνσης τετραπλασιασμένη με τον τρόπο που εξήγησα παραπάνω. Στη συνέχεια, μεταφέρει το rs (τα bit 21 ως 25 της κωδικοποίησης εντολής) στην είσοδο ReadselA (η είσοδος επιλογής του πρώτου καταχωρητή ανάγνωσης) του Register File, το immediate (τα bit 0 ως 15 της κωδικοποίησης εντολής) στην έξοδό της im και ενεργοποιεί την έξοδο A του Register File, οι οποία συνδέεται με την είσοδο A της ALU, δίνοντάς της την τιμή '1'. Στη συνέχεια, βγάζει από το σήμα Enable την τιμή "1101", ώστε να ενεργοποιήσει τους καταχωρητές μέσω των οποίων θα βγουν τα δεδομένα του καταχωρητή που θα φορτωθεί και η διεύθυνση μνήμης που θα υπολογίσει η ALU, ενεργοποιεί την εγγραφή της μνήμης RAM βγάζοντας από το σήμα RAM_r_w την τιμή '1', μεταφέρει το rt (τα bit 16 ως 20 της κωδικοποίησης εντολής) στην είσοδο ReadselB (η είσοδος επιλογής του δεύτερου καταχωρητή ανάγνωσης) του Register File και ενεργοποιεί την έξοδο B του Register File, δίνοντάς της την τιμή '1'. Έτσι, στη διεύθυνση που υπολόγισε η ALU θα μεταφερθούν τα δεδομένα του καταχωρητή που φορτώθηκε. Σε αυτό το σημείο τελειώνει η εντολή, η οποία λέει στον επεξεργαστή να φορτώσει τα δεδομένα του καταχωρητή 3 στη διεύθυνση μνήμης A(16) (υποθέτουμε ότι ο καταχωρητής 2 περιέχει τη βάση του πίνακα A).

Στο σχήμα 2.9 φαίνεται το κύκλωμα του μικροεπεξεργαστή που υλοποιήθηκε. Για να μη γίνουν πολύπλοκες οι συνδέσεις, έχουν μπει μερικοί κύκλοι με ονομασίες στο παραπάνω κύκλωμα. Όσοι από αυτούς τους κύκλους έχουν ίδιο όνομα, σημαίνει ότι είναι συνδεδεμένοι μεταξύ τους. Επίσης, Υπάρχουν και 2 σήματα το ένα στα 4 και το άλλο στα 5 bit, τα οποία έχουν κι αυτά κύκλους. Επειδή τα bit των 2 αυτών σημάτων συνδέονται σε διαφορετικό σημείο το καθένα, έχει μπει ο αριθμός του bit που συνδέεται μέσα σε παρένθεση.



Σχήμα 2.9: Το top module του επεξεργαστή

Για παράδειγμα, στον καταχωρητή Register_Data_in υπάρχει μια σύνδεση rst(2). Αυτό σημαίνει ότι το σήμα μηδενισμού (Reset) του συγκεκριμένου καταχωρητή είναι το bit 2 του σήματος RstReg που βγάζει η μονάδα ελέγχου.

3. Σχεδίαση – Υλοποίηση

Το κεφάλαιο αυτό της παρούσας εργασίας, περιλαμβάνει κομμάτια του κώδικα που γράφτηκαν για τη σχεδίαση του μικροεπεξεργαστή. Ο κώδικας αυτός είναι γραμμένος στη γλώσσα περιγραφής υλικού VHDL. Για την εγγραφή του κώδικα χρησιμοποιήθηκε το πρόγραμμα ISE Project Navigator της Xilinx.

3.1 Ο κώδικας της ALU

Η Αριθμητική και Λογική μονάδα που σχεδιάστηκε, περιλαμβάνει 2 components. Τον αθροιστή – αφαιρέτη και το κύκλωμα ολίσθησης. Για τα υπόλοιπα κυκλώματα που περιέχει η ALU δε δημιουργήθηκαν components, επειδή χρησιμοποιήθηκε behavioral σχεδίαση και περιλαμβάνονται όλα στον κώδικα του top module.

3.1.1 Ο κώδικας του αθροιστή – αφαιρέτη της ALU

Για τον κώδικα του αθροιστή – αφαιρέτη της ALU χρησιμοποιήθηκε συνδυασμός της structural και της Data – Flow σχεδίασης. Αρχικά ορίστηκαν οι βιβλιοθήκες που χρησιμοποιούνται σε όλα τα αρχεία του κώδικα:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Στη συνέχεια, ακολουθεί το entity, το οποίο περιέχει τις εισόδους και τις εξόδους του αθροιστή – αφαιρέτη:

```
entity AddSub_32bits is
   Port (X          : in  STD_LOGIC_VECTOR (31 downto 0);
         Y          : in  STD_LOGIC_VECTOR (31 downto 0);
         Cin        : in  STD_LOGIC;
         M          : in  STD_LOGIC;
         S          : out STD_LOGIC_VECTOR (31 downto 0);
         Cout       : out STD_LOGIC);
end AddSub_32bits;
```

όπως φαίνεται και από τον κώδικα, στο τέλος του ονόματος κάθε σήματος αναφέρεται αν πρόκειται για είσοδο ή για έξοδο, όπως επίσης από πόσα bits αποτελείται.

Συνεχίζοντας, ακολουθεί το όνομα της αρχιτεκτονικής που χρησιμοποιείται, καθώς επίσης και τα components που περιέχονται στο συγκεκριμένο κύκλωμα:

```
architecture MIX of AddSub_32bits is
component Full_adder is
Port (  X       : in  STD_LOGIC;
        Y       : in  STD_LOGIC;
        Cin     : in  STD_LOGIC;
        Cout    : out  STD_LOGIC;
        S       : out  STD_LOGIC
```

```
        );
    end component;
```

Όσον αφορά το component, δηλώνεται όπως και το entity, δηλώνοντας με τον ίδιο ακριβώς τρόπο τις εισόδους και τις εξόδους του. Ο αθροιστής – αφαιρέτης που υλοποιήθηκε αποτελείται από 32 πλήρεις αθροιστές, γι αυτό το component του είναι ένας πλήρης αθροιστής.

Στη συνέχεια, ακολουθούν τα σήματα που χρησιμοποιήθηκαν για τις εσωτερικές συνδέσεις του κυκλώματος και μετά ξεκινάει το κυρίως μέρος του κώδικα:

```
signal s_AddorSubb : STD_LOGIC;
signal Adder_Cout,x0r : STD_LOGIC_VECTOR (31 downto 0);
begin
```

Μετά το begin, ακολουθούν κάποιες εξισώσεις, με τις οποίες στο συγκεκριμένο component δημιουργείται το κύκλωμα xor για να μπορεί ο αθροιστής να λειτουργήσει ταυτόχρονα και ως αφαιρέτης. Παρακάτω βλέπουμε μερικές από αυτές τις εξισώσεις. Οι υπόλοιπες είναι πανομοιότυπες απλά αλλάζουν οι αριθμοί και πάνε μέχρι το 31:

```
x0r(0)  <= Y(0)  xor s_AddorSubb;
x0r(1)  <= Y(1)  xor s_AddorSubb;
x0r(2)  <= Y(2)  xor s_AddorSubb;
        .
        .
        .
x0r(29) <= Y(29) xor s_AddorSubb;
x0r(30) <= Y(30) xor s_AddorSubb;
x0r(31) <= Y(31) xor s_AddorSubb;
```

Μετά από αυτές τις εξισώσεις, υπάρχει η σύνδεση του signal s_AddorSubb με την είσοδο M:

```
s_AddorSubb <= M;
```

Στο τελευταίο σημείο του κώδικα, υπάρχουν τα Port Maps, τα οποία από το component του πλήρους αθροιστή, δημιουργούν τους 32 πλήρεις αθροιστές από τους οποίους αποτελείται ο αθροιστής – αφαιρέτης. Κάθε port map περιλαμβάνει τις εισόδους και τις εξόδους του component καθώς και τις συνδέσεις τους. Παρακάτω φαίνονται τα 2 πρώτα και το τελευταίο port map του κώδικα του αθροιστή – αφαιρέτη της ALU:

```
Adder0 : Full_Adder
Port map
(
        X       =>      X(0),
        Y       =>      x0r(0),
        Cin     =>      s_AddorSubb,
        S       =>      S(0),
```

```
        Cout    =>      Adder_Cout(0)
);


Adder1 : Full_Adder
Port map
(
        X       =>      X(1),
        Y       =>      x0r(1),
        Cin     =>      Adder_Cout(0) ,
        S       =>      S(1),
        Cout    =>      Adder_Cout(1)
);

Adder31 : Full_Adder
Port map
(
        X       =>      X(31),
        Y       =>      x0r(31),
        Cin     =>      Adder_Cout(30) ,
        S       =>      S(31),
        Cout    =>      Cout
);
end MIX;
```

Τα υπόλοιπα port maps, είναι ακριβώς ίδια με το port map του Adder1 , απλά αλλάζουν οι αριθμοί στις παρενθέσεις.

3.1.2 Ο κώδικας του κυκλώματος ολίσθησης της ALU

Το κύκλωμα ολίσθησης της ALU που υλοποιήθηκε στην παρούσα εργασία, όπως αναφέρεται και στο προηγούμενο κεφάλαιο, αποτελείται από 32 πολυπλέκτες 64x1 με τις εισόδους και τις εξόδους τους να είναι στα 32 bits. Για την υλοποίηση του κυκλώματος αυτού, χρησιμοποιήθηκε structural σχεδίαση. Αρχικά, ο κώδικας μετά από τις βιβλιοθήκες που είναι οι ίδιες με αυτές του αθροιστή - αφαιρέτη, περιλαμβάνει το entity με τις δηλώσει των εισόδων και των εξόδων του κυκλώματος:

```
entity Combinational_bidirectional_shift is
   Port (Input      :       in      STD_LOGIC_VECTOR (31 downto 0);
         dir_sel    :       in      STD_LOGIC;
         shamt      :       in      STD_LOGIC_VECTOR (4 downto 0);
         Output     :       out     STD_LOGIC_VECTOR (31 downto 0)
        );
end Combinational_bidirectional_shift;
```

Ο κώδικας συνεχίζεται με την αρχιτεκτονική και τα components, που στην περίπτωση αυτή είναι ένας πολυπλέκτης 64x1:

```
architecture Structural of Combinational_bidirectional_shift is
component Mux_64x1 is
Port (  I       : in    STD_LOGIC_VECTOR (63 downto 0);
        sel     : in    STD_LOGIC_VECTOR (5 downto 0);
        ShOut   : out   STD_LOGIC
     );
end component;
```

Τέλος, έχουμε το κυρίως μέρος του κώδικα με τα port maps. Εδώ βλέπουμε τα πρώτα 2 port maps και το τελευταίο. Με τον ίδιο τρόπο γράφτηκαν και τα υπόλοιπα, απλά επειδή είναι πάρα πολλά για συντομία δε συμπεριλαμβάνονται όλα στο κείμενο της παρούσας εργασίας. Έχουν, όμως, επεξηγηθεί οι συνδέσεις τους στο προηγούμενο κεφάλαιο.

```
begin
Mux0 : Mux_64x1
Port map
        ( I(0)                  => Input(0),
          I(1)                  => Input(1),
          I(2)                  => Input(2),
          I(3)                  => Input(3),

                          .
                          .
                          .
          I(30)                 => Input(30),
          I(31)                 => Input(31),
          I(32)                 => Input(0),
          I(33)                 => '0',
          I(34)                 => '0',
          I(35)                 => '0',
                          .
                          .
                          .
          I(63)                 => '0',
          sel(5)                => dir_sel,
          sel(4 downto 0)       => shamt,
          ShOut                 => Output(0)
        );
Mux1 : Mux_64x1
Port map
        ( I(0)                  => Input(1),
```

```
        I(1)                    => Input(2),
        I(2)                    => Input(3),
        I(3)                    => Input(4),

                                .
                                .
                                .
        I(30)                   => Input(31),
        I(31)                   => '0',
        I(32)                   => Input(1),
        I(33)                   => Input(0),
        I(34)                   => '0',
        I(35)                   => '0',
                                .
                                .
                                .
        I(63)                   => '0',
        sel(5)                  => dir_sel,
        sel(4 downto 0)         => shamt,
        ShOut                   => Output(1)
      );
                                .
                                .
                                .
Mux31 : Mux_64x1
Port map
      ( I(0)                    => Input(31),
        I(1)                    => '0',
        I(2)                    => '0',
        I(3)                    => '0',
        I(4)                    => '0',
        I(5)                    => '0',
        I(6)                    => '0',
        I(7)                    => '0',
        I(8)                    => '0',
        I(9)                    => '0',
        I(10)                   => '0',
                                .
                                .
                                .
        I(31)                   => '0',
```

```
            I(32)                   => Input(31),
            I(33)                   => Input(30),
            I(34)                   => Input(29),

                                .
                                .
                                .
            I(61)                   => Input(2),
            I(62)                   => Input(1),
            I(63)                   => Input(0),
            sel(5)                  => dir_sel,
            sel(4 downto 0)  => shamt,
            ShOut                           => Output(31)
           );
end Structural;
```

3.1.3 Ο κώδικας του top module της ALU

Για τον κώδικα του top module της ALU που υλοποιήθηκε, χρησιμοποιήθηκε ο συνδυασμός της behavioral και της structural σχεδίασης. Αρχικά, όπως και στα προηγούμενα κομμάτια του κώδικα, υπάρχουν οι βιβλιοθήκες και το entity. Οι βιβλιοθήκες είναι ακριβώς οι ίδιες με πριν, ενώ το entity είναι το εξής:

```
entity ALU32 is
Port (  A           : in    STD_LOGIC_VECTOR (31 downto 0);
        B           : in    STD_LOGIC_VECTOR (31 downto 0);
        mode        : in    STD_LOGIC_VECTOR (2 downto 0);
        Int_Out     : out   STD_LOGIC_VECTOR (31 downto 0);
        Zero        : out   STD_LOGIC;
        Cout        : out   STD_LOGIC
     );
end ALU32;
```

Στη συνέχεια του κώδικα, βρίσκεται η αρχιτεκτονική και ακολουθούν τα components, τα οποία στην περίπτωση αυτή είναι ο αθροιστής – αφαιρέτης και το κύκλωμα ολίσθησης:

```
architecture Mix of ALU32 is
component AddSub_32bits is
Port (  X       : in    STD_LOGIC_VECTOR (31 downto 0);
        Y       : in    STD_LOGIC_VECTOR (31 downto 0);
        Cin     : in    STD_LOGIC;              --Carry in
        M       : in    STD_LOGIC;              --Select add or sub
        S       : out   STD_LOGIC_VECTOR (31 downto 0);
        Cout    : out   STD_LOGIC
     );
end component;
```

```
Component Combinational_bidirectional_shift is
Port (  Input       : in    STD_LOGIC_VECTOR (31 downto 0);
        dir_sel     : in    STD_LOGIC;
        shamt       : in    STD_LOGIC_VECTOR (4 downto 0);
        Output      : out   STD_LOGIC_VECTOR (31 downto 0)
      );
end component;
```

Μετά από τα components, ο κώδικας συνεχίζει με τα signals μετά από τα οποία ξεκινάει το κυρίως μέρος του με τα port maps:

```
begin
AddSubb : AddSub_32bits
Port map
     ( X      =>     A,
       Y      =>     B,
       Cin    =>     mode(2),
       M      =>     mode(2),
       S      =>     S_out,
       Cout   =>     Cout
     );
Shift : Combinational_bidirectional_shift
Port map
     (
       Input   =>    A,
       dir_sel =>    mode(0),          --Shift direction select
       shamt   =>    B(4 downto 0),    --Shift Amount
       Output  =>    Sh_out            --Shift Output
     );
```

Στη συνέχεια, περνάμε στο behavioral κομμάτι του κώδικα της ALU, το οποίο αποτελείται από μια διαδικασία (process) η οποία εμπεριέχει τον πολυπλέκτη της ALU, σχεδιασμένο με χρήση της εντολής case, καθώς και το κύκλωμα που βγάζει την έξοδο Zero :

```
process ( mode,A,B,Sh_out,S_out)
begin
  case mode is
    when "000"      => Int_out <= A and B;      --and
    when "001"      => Int_out <= A or B;       --or
    when "010"      => Int_out <= A nor B;      --nor
    when "011"      => Int_out <= S_out;        --add
    when "100"      => Int_out <= Sh_out;       --srl
    when "101"      => Int_out <= Sh_out;       --sll
    when "110"      =>                          --slt
                 if  A < B  then
       Int_out      <= "00000000000000000000000000000001";
```

```
                else
            Int_out         <= "00000000000000000000000000000000";
                end if;
        when "111"          => Int_out <= S_out;            --sub
        when others => Int_out <= "00000000000000000000000000000000";
    end case;
            if  A = B  then
              Zero <= '1';
            else
              Zero <= '0';
            end if;
end process;
end Mix;
```

3.2 Ο κώδικας του Register File

Το Register File που υλοποιήθηκε, περιέχει 4 components: Έναν αποκωδικοποιητή 5x32 για την επιλογή καταχωρητή εγγραφής, 2 κυκλώματα πολυπλεκτών για την επιλογή των 2 καταχωρητών ανάγνωσης και άλλο ένα component το οποίο αποτελείται από 32 καταχωρητές παράλληλης εισόδου και εξόδου οι οποίοι είναι στα 32 bits.

3.2.1 Ο κώδικας του αποκωδικοποιητή 5x32

Για τον κώδικα του αποκωδικοποιητή, χρησιμοποιήθηκε behavioral σχεδίαση. Ο αποκωδικοποιητής έχει 2 εισόδους, μια των 5 και μια του ενός bit, και μια έξοδο των 32 bits. Άρα το entity, το οποίο ακολουθεί μετά τις βιβλιοθήκες οι οποίες είναι σε όλα τα κομμάτια του κώδικα ίδιες, θα είναι:

```
entity Dec_5x32 is
Port ( rs       : in  STD_LOGIC_VECTOR (4 downto 0);    --Register Select
       en       : in  STD_LOGIC;                        --Enable
       decout   : out  STD_LOGIC_VECTOR (31 downto 0));--Decoder output
end Dec_5x32;
```

Ο κώδικας συνεχίζεται με την αρχιτεκτονική και το begin που δηλώνει ότι ξεκινάει το κυρίως μέρος του:

```
architecture Behavioral of Dec_5x32 is

begin
```

Στο κυρίως μέρος του κώδικα του αποκωδικοποιητή, υπάρχει μια διαδικασία που περιγράφει τη λειτουργία του. Για την περιγραφή αυτή, χρησιμοποιήθηκε μια if και μέσα σε αυτή μια case. Επειδή μέσα στην case ορίζεται τι έξοδο θα βγάλει ανά είσοδο που λαμβάνει ο αποκωδικοποιητής κι επειδή ακόμα, όπως έχει επεξηγηθεί και στο προηγούμενο κεφάλαιο, οι έξοδοι συνδέονται με τα σήματα enable των καταχωρητών και πρέπει να ενεργοποιούν έναν καταχωρητή κάθε φορά ο κώδικας έχει ως εξής:

```
if (en = '0') then          -- Enable is active low
        case    rs      is
when "00000" => decout <= "11111111111111111111111111111110";
--Reg_0 enabled
when "00001" => decout <= "11111111111111111111111111111101";
--Reg_1 enabled
when "00010" => decout <= "11111111111111111111111111111011";
--Reg_2 enabled
                                .
                                .
                                .
when "11111" => decout <= "01111111111111111111111111111111";
--Reg_31 enabled
when others    => decout <= "11111111111111111111111111111111";
--All Registers disabled
        end case;
else
                    decout <= "11111111111111111111111111111111";
--If signal en is disabled all Registers are disabled too
 end if;
 end process;
 end Behavioral;
```

Σε αυτό το σημείο τελειώνει ο κώδικας του αποκωδικοποιητή. Η έξοδος του αποκωδικοποιητή παίρνει σήματα με πολλά '1' και το πολύ ένα '0' επειδή τα σήματα Load των καταχωρητών, από τα οποία ενεργοποιείται η φόρτωση τιμής, είναι Active Low (Ενεργά στο '0').

3.2.2 Ο κώδικας του καταχωρητή των 32 bits

Για τον κώδικα του καταχωρητή των 32 bits, χρησιμοποιήθηκε η behavioral σχεδίαση. Ο καταχωρητής, όπως αναφέρεται και στο προηγούμενο κεφάλαιο της παρούσας εργασίας, έχει 4 εισόδους, 3 του ενός και μια των 32 bits, και 1 έξοδο των 32 bits. Έτσι, το entity είναι ως εξής:

```
entity Reg32PiPo is     --Register 32 bit Parallel input Parallel output
Port ( D        : in  STD_LOGIC_VECTOR (31 downto 0);
       Load     : in  STD_LOGIC;
       Rst      : in  STD_LOGIC;     --Reset

        clk     : in  STD_LOGIC;     --clock
        Q       : out  STD_LOGIC_VECTOR (31 downto 0)
      );
end Reg32PiPo;
```

Στη συνέχεια, έχουμε την αρχιτεκτονική και μετά ξεκινάει το κυρίως μέρος του κώδικα, όπου υπάρχει μια διαδικασία που περιγράφει τη λειτουργία του καταχωρητή. Αυτή η διαδικασία περιέχει μια εντολή if:

```
architecture Behavioral of Reg32PiPo is
begin
process (clk,Rst)                --Το σήμα Reset είναι ασύγχρονο
  begin
    if Rst = '1' then            -- Reset is Active High
        Q <= "00000000000000000000000000000000"; --Μηδενισμός
    else
       if clk' event and clk = '1' and Load = '0' then    -- Load is Active Low
         Q <= D;                 --Φόρτωση τιμής
       end if;
    end if;
end process;
end Behavioral;
```

Σε αυτό το σημείο τελειώνει ο κώδικας του καταχωρητή των 32 bits.

3.2.3 Ο κώδικας του πολυπλέκτη 32x1

Τα υπόλοιπα 2 components του Register File, αποτελούνται από πολυπλέκτες 32x1. Ο κώδικας λοιπόν που ακολουθεί είναι ο κώδικας ενός πολυπλέκτη 32x1. Για αυτό τον κώδικα χρησιμοποιήθηκε behavioral σχεδίαση. Οι βιβλιοθήκες, όπως έχει αναφερθεί πολλές φορές στο κεφάλαιο αυτό της εργασίας, είναι οι ίδιες. Το entity έχει ως εξής:

```
entity Mux32x1 is
Port (  I       : in  STD_LOGIC_VECTOR (31 downto 0); --Input
        msel    : in  STD_LOGIC_VECTOR (4 downto 0);  --mux select
        Outen   : in  STD_LOGIC;                      --Output Enable
        mout    : out  STD_LOGIC);                    --mux output
end Mux32x1;
```

Στη συνέχεια ακολουθεί η αρχιτεκτονική και το κυρίως μέρος. Στο κυρίως μέρος, όπως και πριν, περιγράφεται η λειτουργία του πολυπλέκτη.

```
architecture Behavioral of Mux32x1 is
begin
process (I,msel,Outen)
        begin
if Outen = '1' then      --Output Enable is Active High


   case msel is
when "00000" => mout <=  I(0);
when "00001" => mout <=  I(1);
when "00010" => mout <=  I(2);
```

```
                    .
                    .
                    .
when "11111" => mout <= I(31);
when others => mout <= 'Z';
    end case;
else
    mout <= 'Z';
end if;
end process;
end Behavioral;
```

Σε αυτό το σημείο τελειώνει ο κώδικας του πολυπλέκτη 32x1.

3.2.4 Ο κώδικας του top module του Register File

Για τον κώδικα του top module του Register File, χρησιμοποιήθηκε structural μορφή σχεδίασης, όπως έγινε και με το top module της ALU. Έτσι, μετά τις βιβλιοθήκες, ακολουθεί το entity, όπου αναφέρονται οι είσοδοι και οι έξοδοι όπως έχουν περιγραφτεί στο προηγούμενο κεφάλαιο της εργασίας:

```
entity Registers is
 Port ( Reg_in        : in  STD_LOGIC_VECTOR (31 downto 0);
        wrsel         : in  STD_LOGIC_VECTOR (4 downto 0);
        Reset         : in     STD_LOGIC;
        wren          : in  STD_LOGIC;
        clk_in        : in  STD_LOGIC;
        readselA      : in  STD_LOGIC_VECTOR (4 downto 0);
        readselB      : in  STD_LOGIC_VECTOR (4 downto 0);
        Regout_enA    : in  STD_LOGIC;
        Regout_enB    : in  STD_LOGIC;
        Reg_outA      : out STD_LOGIC_VECTOR (31 downto 0);
        Reg_outB      : out STD_LOGIC_VECTOR (31 downto 0));
end Registers;
```

Στη συνέχεια, ο κώδικας συνεχίζεται με την αρχιτεκτονική, τα components και τα signals:

```
architecture Structural of Registers is
component Dec_5x32 is   --Decoder
Port (  rs            : in     STD_LOGIC_VECTOR (4 downto 0);
        en            : in     STD_LOGIC;
        decout        : out    STD_LOGIC_VECTOR (31 downto 0)
      );
end component;

component Reg32PiPo is  --Register
Port (  D             : in     STD_LOGIC_VECTOR (31 downto 0);
```

```
        Load        : in    STD_LOGIC;
        Rst         : in    STD_LOGIC;
        clk         : in    STD_LOGIC;
        Q           : out   STD_LOGIC_VECTOR (31 downto 0)
    );
end component;

component Mux32x1 is    --multiplexor
Port (  I           : in    STD_LOGIC_VECTOR (31 downto 0);
        msel        : in    STD_LOGIC_VECTOR (4 downto 0);
        Outen       : in    STD_LOGIC;
        mout        : out   STD_LOGIC);
end component;

signal DEC_out,Rin : STD_LOGIC_VECTOR (31 downto 0);
signal R31mux, ···,R0mux: STD_LOGIC_VECTOR (31 downto 0);
-- Εδώ έχουμε τα signals R31mux,R30mux,···,R0mux. Η αρίθμηση ξεκινάει
--από την τιμή 31 και φτάνει μέχρι την τιμή 0.Αυτά τα signal, όπως θα φανεί
--παρακάτω στα port maps, συνδέουν τις εξόδους των καταχωρητών με τους
--πολυπλέκτες
signal mselP,mselS : STD_LOGIC_VECTOR (4 downto 0);
signal Res : STD_LOGIC;
```

Τέλος, ξεκινάει το κυρίως μέρος του κώδικα με τις συνδέσεις κάποιων από τα signals και τα port maps:

```
begin
Rin         <= Reg_in;
Res         <= Reset;
mselP       <= readselA;
mselS       <= readselB;

----------------------Decoder-----------------------------------------------------

Decoder : Dec_5x32
Port map
  (     rs          =>      wrsel,
        en          =>      wren,
        decout      =>      DEC_out
  );



----------------------Registers---------------------------------------------------

Register0 : Reg32PiPo
Port map
  (     D     =>      Rin ,
```

```
        Load   =>   DEC_out (0),
        Rst    =>   Res,
        clk    =>   clk_in,
        Q      =>   R0mux
    );
Register1 : Reg32PiPo
Port map
    (   D      =>   Rin ,
        Load   =>   DEC_out (1),
        Rst    =>   Res,
        clk    =>   clk_in,
        Q      =>   R1mux
    );
                .
                .
                .
Register31 : Reg32PiPo
Port map
    (   D      =>   Rin,
        Load   =>   DEC_out (31),
        Rst    =>   Res,
        clk    =>   clk_in,
        Q      =>   R31mux
    );

----------Multiplexors Primary (Κύκλωμα ανάγνωσης A)----------
Mux0Pr : Mux32x1
Port map
    (   I(0)   =>   R0mux(0),
        I(1)   =>   R1mux(0),
        I(2)   =>   R2mux(0),
        I(3)   =>   R3mux(0),
                .
                .
                .
        I(30)  =>   R30mux(0),
        I(31)  =>   R31mux(0),
        msel   =>   mselP,
        Outen  =>   Regout_enA,
        mout   =>   Reg_outA(0)
    );
Mux1Pr : Mux32x1
Port map
```

```
    (   I(0)    =>    R0mux(1),
        I(1)    =>    R1mux(1),
        I(2)    =>    R2mux(1),
                .
                .
                .
        I(31)   =>    R31mux(1),
        msel    =>    mselP,
        Outen   =>    Regout_enA,
        mout    =>    Reg_outA (1)
    );
                .
                .
                .
Mux31Pr : Mux32x1
Port map
    (   I(0)    =>    R0mux(31),
        I(1)    =>    R1mux(31),
                .
                .
                .
        I(31)   =>    R31mux(31),
        msel    =>    mselP,
        Outen   =>    Regout_enA,
        mout    =>    Reg_outA (31)
    );

----------Multiplexors Secondary (Κύκλωμα ανάγνωσης B)----------
Mux0Sec : Mux32x1
Port map
    (   I(0)    =>    R0mux(0),
        I(1)    =>    R1mux(0),
        I(2)    =>    R2mux(0),
                .
                .
                .
        I(31)   =>    R31mux(0),
        msel    =>    mselS,
        Outen   =>    Regout_enB,
        mout    =>    Reg_outB(0)
    );
```

```
                .
                .
                .
Mux31Sec : Mux32x1
Port map
  (     I(0)    =>      R0mux(31),
        I(1)    =>      R1mux(31),
                .
                .
                .
        I(31)   =>      R31mux(31),
        msel    =>      mselS,
        Outen   =>      Regout_enB,
        mout    =>      Reg_outB(31)
   );
end Structural;
```

Σε αυτό το σημείο τελειώνει ο κώδικας του Register File.

3.3 Ο κώδικας της Μονάδας Ελέγχου

Για τον κώδικα της μονάδας ελέγχου που υλοποιήθηκε, χρησιμοποιήθηκε behavioral μορφή σχεδίασης. Επειδή ο κώδικας της μονάδας ελέγχου είναι πολύ μεγάλος σε μέγεθος, το κείμενο της παρούσας εργασίας περιλαμβάνει ένα μέρος αυτού.

Αρχικά, μετά από τις βιβλιοθήκες, ο κώδικας έχει το entity:

```
entity Control_Unit is
Port (  Code      :     in  STD_LOGIC_VECTOR (31 downto 0);
        Z         :     in  STD_LOGIC;
        C         :      in  STD_LOGIC;
        clk       :     in  STD_LOGIC;
        Rst       :     in  STD_LOGIC;
        Enable    :     out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
        mode      :     out  STD_LOGIC_VECTOR (2 downto 0);
        RegoA     :     out  STD_LOGIC := '0';
        RegoB     :     out  STD_LOGIC := '0';
        wren      :     out  STD_LOGIC := '1';
        RstRegs   :     out  STD_LOGIC_VECTOR (4 downto 0) := "00000";
        wrsel     :     out  STD_LOGIC_VECTOR (4 downto 0);
        ReadselA  :     out  STD_LOGIC_VECTOR (4 downto 0);
        ReadselB  :     out  STD_LOGIC_VECTOR (4 downto 0);
        im        :     out  STD_LOGIC_VECTOR (15 downto 0);
        ReginSel  :     out  STD_LOGIC;
        ALUBSel   :     out  STD_LOGIC_VECTOR (1 downto 0);
```

```
        Shamt    :      out  STD_LOGIC_VECTOR (4 downto 0);
        RAM_r_w:    out  STD_LOGIC := '0');
end Control_Unit;
```

Όπως φαίνεται από το entity, οι είσοδοι Enable, RegoB, wren και RstRegs αρχικοποιήθηκαν όλες στο '0' εκτός από το wren που είναι ενεργό στο '0' και αρχικοποιήθηκε στο '1'.

Ο κώδικας συνεχίζεται με την αρχιτεκτονική και ακολουθεί το κυρίως μέρος του, το οποίο περιέχει μια διαδικασία που περιγράφει τη λειτουργία της μονάδας ελέγχου ανά εντολή. Για την περιγραφή αυτή χρησιμοποιήθηκαν οι εντολές case και if. Επειδή οι εντολές είναι πολλές και ο κώδικας πολύ μεγάλος, όπως αναφέρθηκε και πιο πάνω, στο κείμενο της παρούσας εργασίας εμπεριέχονται μόνο 4 εντολές από αυτές που εκτελεί ο επεξεργαστής: Οι εντολές add,sll,slti και sw.

```
architecture Behavioral of Control_Unit is
begin
process (clk,Rst)
        begin
        if Rst = '1' then
                RstRegs <= "11111";
        else
          if clk' event and clk = '1' then
        Enable                  <= "0100";--Ενεργοποιείται ο καταχωρητής
--που εισάγει την κωδικοποίηση εντολής στη μονάδα ελέγχου
case (Code(31 downto 26)) is
        when "000000" =>
                if Code (5 downto 0) = "100000" then – Εντολή add
                mode            <= "011"; --HALU εκτελεί πρόσθεση
                ReginSel        <= '0';  -- Η έξοδος της ALU μεταφέρεται στην
--είσοδο των καταχωρητών του Register File
                ALUBSel         <= "01"; --Η έξοδος ανάγνωσης Β του Register
--File μεταφέρεται στην είσοδο Β της ALU
                ReadselA        <= Code(25 downto 21); -- Επιλέγεται ο
--πρώτος καταχωρητής ανάγνωσης από το κομμάτι rs της κωδικοποίησης
--εντολής
                ReadselB        <= Code(20 downto 16); -- Επιλέγεται ο
--δεύτερος καταχωρητής ανάγνωσης από το κομμάτι rt της κωδικοποίησης
--εντολής
                RegoA           <= '1';  -- Ενεργοποιείται η έξοδος ανάγνωσης Α
--του Register File
                RegoB           <= '1';  -- Ενεργοποιείται η έξοδος ανάγνωσης Β
--του Register File
                wrsel           <= Code(15 downto 11); -- Επιλέγεται ο
--καταχωρητής εγγραφής από το κομμάτι rd της κωδικοποίησης εντολής
                wren            <= '0';  -- Ενεργοποιείται ο αποκωδικοποιητής
--του Register File
```

```
                .
                .
                .
if Code (5 downto 0) = "000000" then – Εντολή sll
                mode        <= "101"; --Η ALU εκτελεί αριστερή ολίσθηση
                ReginSel    <= '0';  -- Η έξοδος της ALU μεταφέρεται στην
--είσοδο των καταχωρητών του Register File
                ALUBSel     <= "11"; --Η είσοδος Β της ALU παίρνει το
--shamt
                ReadselA    <= Code(20 downto 16); -- Επιλέγεται ο
--καταχωρητής ανάγνωσης από το κομμάτι rt της κωδικοποίησης
--εντολής
                RegoA       <= '1';  -- Ενεργοποιείται η έξοδος ανάγνωσης Α
--του Register File
                wrsel       <= Code(15 downto 11); -- Επιλέγεται ο
--καταχωρητής εγγραφής από το κομμάτι rd της κωδικοποίησης εντολής
                wren        <= '0';  -- Ενεργοποιείται ο αποκωδικοποιητής
--του Register File
                .
                .
                .
end if;
                .
                .
                .
when "001010" => -- Εντολή slti
        mode        <= "110"; --Η ALU εκτελεί την εντολή slt
        ReginSel    <= '0';  -- Η έξοδος της ALU μεταφέρεται στην
--είσοδο των καταχωρητών του Register File
        ALUBSel     <= "00"; --Η είσοδος Β της ALU παίρνει την άμεση
--τιμή
        ReadselA    <= Code(25 downto 21); -- Επιλέγεται ο
--καταχωρητής ανάγνωσης από το κομμάτι rs της κωδικοποίησης
--εντολής
        RegoA       <= '1';  -- Ενεργοποιείται η έξοδος ανάγνωσης Α
--του Register File
        im          <= Code(15 downto 0); -- Η έξοδος im της μονάδας
--ελέγχου βγάζει την άμεση τιμή
        wrsel       <= Code(20 downto 16); -- Επιλέγεται ο
--καταχωρητής εγγραφής από το κομμάτι rt της κωδικοποίησης εντολής
        wren        <= '0';  -- Ενεργοποιείται ο αποκωδικοποιητής
--του Register File
```

```
                    .
                    .
                    .
when "101011" => -- Εντολή sw
        mode        <= "011"; -- Η ALU εκτελεί πρόσθεση για τον
--υπολογισμό της διεύθυνσης
        ALUBSel     <= "10"; --Η είσοδος Β της ALU παίρνει την τιμή
--της διεύθυνσης (από το κομμάτι address της κωδικοποίησης εντολής)
--τετραπλασιασμένη
        ReadselA    <= Code(25 downto 21); -- Επιλέγεται ο
--καταχωρητής ανάγνωσης από το κομμάτι rs της κωδικοποίησης
--εντολής
        RegoA       <= '1';  -- Ενεργοποιείται η έξοδος ανάγνωσης Α
--του Register File
        im          <= Code(15 downto 0); -- Η έξοδος im της μονάδας
--ελέγχου βγάζει το κομμάτι address της κωδικοποίησης εντολής
        Enable      <= "1101";  -- Ενεργοποιούνται οι καταχωρητές
--Register Data_out και Reg_Addr
        RAM_r_w     <= '1';  -- Ενεργοποιείται η εγγραφή της RAM
        ReadselB    <= Code(20 downto 16); -- Επιλέγεται ο
--καταχωρητής του οποίου τα δεδομένα θα μεταφερθούν στη RAM από το
--κομμάτι rt της κωδικοποίησης εντολής
        RegoB       <= '1';  -- Ενεργοποιείται η έξοδος ανάγνωσης Β
--του Register File
                    .
                    .
                    .
end case;
         end if;
        end if;
end process;
end Behavioral;
```

Σε αυτό το σημείο τελειώνει ο κώδικας της μονάδας ελέγχου.

3.4 Ο κώδικας του top module του επεξεργαστή

Για τον κώδικα του top module του επεξεργαστή, χρησιμοποιήθηκε structural σχεδίαση.

Ο κώδικας, μετά τις βιβλιοθήκες έχει το entity και ακολουθεί η αρχιτεκτονική:

```
entity CPU is
Port (  Op_Data     :     inout   STD_LOGIC_VECTOR (31 downto 0);
        clock_input :     in      STD_LOGIC;
        Reset       :     in      STD_LOGIC;
```

```
        Address     :     out   STD_LOGIC_VECTOR (31 downto 0);
        Mem_r_w     :     out   STD_LOGIC);
end CPU;
architecture Structural of CPU is
```

Στη συνέχεια, ο κώδικας περιλαμβάνει τα components τα οποία είναι η μονάδα ελέγχου, το Register File, η ALU, ένας καταχωρητής των 32 bit και 2 πολυπλέκτες:

```
component ALU32 is
Port ( A            :     in   STD_LOGIC_VECTOR (31 downto 0);
       B            :     in   STD_LOGIC_VECTOR (31 downto 0);
       mode         :     in   STD_LOGIC_VECTOR (2 downto 0);
       Int_Out      :     out  STD_LOGIC_VECTOR (31 downto 0);
       Zero         :     out  STD_LOGIC;
       Cout         :     out  STD_LOGIC);
end component;
component Registers is
 Port ( Reg_in      :     in   STD_LOGIC_VECTOR (31 downto 0);
       wrsel        :     in   STD_LOGIC_VECTOR (4 downto 0);
       Reset        :     in   STD_LOGIC;
       wren         :     in   STD_LOGIC;
       clk_in       :     in   STD_LOGIC;
       readselA     :     in   STD_LOGIC_VECTOR (4 downto 0);
       readselB     :     in   STD_LOGIC_VECTOR (4 downto 0);
       Regout_enA   :     in   STD_LOGIC;
       Regout_enB   :     in   STD_LOGIC;
       Reg_outA     :     out STD_LOGIC_VECTOR (31 downto 0);
       Reg_outB     :     out STD_LOGIC_VECTOR (31 downto 0)
     );
end component;
component Register32top is
Port ( i            :     in   STD_LOGIC_VECTOR (31 downto 0);
       clk          :     in   STD_LOGIC;
       Reset        :     in   STD_LOGIC;
       Enable       :     in   STD_LOGIC;
       o            :     out  STD_LOGIC_VECTOR (31 downto 0)
     );
end component;
component MUX4x1 is
Port ( I0           :     in   STD_LOGIC_VECTOR (31 downto 0);
       I1           :     in   STD_LOGIC_VECTOR (31 downto 0);
       I2           :     in   STD_LOGIC_VECTOR (31 downto 0);
       I3           :     in   STD_LOGIC_VECTOR (31 downto 0);
       ALUBsel      :     in   STD_LOGIC_VECTOR (1 downto 0);
       O            :     out STD_LOGIC_VECTOR (31 downto 0)
     );
```

```
end component;
component MUX2x1 is
Port (  I0          :       in  STD_LOGIC_VECTOR (31 downto 0);
        I1          :       in  STD_LOGIC_VECTOR (31 downto 0);
        ReginSel    :       in  STD_LOGIC;
        O           :       out STD_LOGIC_VECTOR (31 downto 0)
    );
end component;
```

Όσον αφορά το component Register32top, ο κώδικάς του είναι σε behavioral σχεδίαση και είναι ίδιος με τον κώδικα του component του Register File Reg32PiPo. Επίσης, τα components MUX4x1 και MUX2x1 υλοποιήθηκαν κι αυτά με behavioral σχεδίαση και ο κώδικάς τους είναι όπως ο κώδικας του component του Register File Mux32x1. Έτσι, ο κώδικάς τους παραλείπεται από το κείμενο της παρούσας εργασίας.

Ο κώδικας του top module του επεξεργαστή που υλοποιήθηκε, συνεχίζεται με τα signals:

```
signal ALUsB                    : STD_LOGIC_VECTOR (1 downto 0);
signal ALU_mode                 : STD_LOGIC_VECTOR (2 downto 0);
signal Ren                      : STD_LOGIC_VECTOR (3 downto 0);
signal Rrst,Sha,Rwrsel,RsA,RsB  : STD_LOGIC_VECTOR (4 downto 0);
signal Immediate                : STD_LOGIC_VECTOR (15 downto 0);
signal ALU_out,Code_in,Data_in  : STD_LOGIC_VECTOR (31 downto 0);
signal Rinp,RegoB,ALU_A,ALU_B: STD_LOGIC_VECTOR (31 downto 0);
signal Carry,Zero,Rins,Rwren,RoAen,RoBen: STD_LOGIC;
```

Τέλος, έχουμε το κυρίως μέρος του κώδικα με τα port maps:

```
begin
CU    :     Control_Unit
Port map (  Code        =>      Code_in,
            Z           =>      Zero,
            C           =>      Carry,
            clk         =>      clock_input,
            Rst         =>      Reset,
            Enable      =>      Ren,
            mode        =>      ALU_mode,
            RegoA       =>      RoAen,
            RegoB       =>      RoBen,
            wren        =>      Rwren,
            RstRegs     =>      Rrst,
            wrsel       =>      Rwrsel,
            ReadselA    =>      RsA,
            ReadselB    =>      RsB,
            im          =>      Immediate,
            ReginSel    =>      Rins,
            ALUBSel     =>      ALUsB,
```

```
                Shamt  =>      Sha,
                RAM_r_w     =>      Mem_r_w
        );
ALU     :       ALU32
Port map (      A           =>      ALU_A,
                B           =>      ALU_B,
                mode        =>      ALU_mode,
                Int_Out     =>      ALU_out,
                Zero        =>      Zero,
                Cout        =>      Carry
        );
Register_File   :       Registers
Port map (      Reg_in      =>      Rinp,
                wrsel       =>      Rwrsel,
                Reset       =>      Rrst(4),
                wren        =>      Rwren,
                clk_in      =>      clock_input,
                readselA    =>      RsA,
                readselB    =>      RsB,
                Regout_enA  =>       RoAen,
                Regout_enB  =>      RoBen,
                Reg_outA    =>      ALU_A,
                Reg_outB    =>      RegoB
        );
Multiplexor_4X1     :       MUX4x1
Port map (      I0 (15 downto 0)        =>      Immediate,
                I0 (31 downto 16)       =>      "0000000000000000",
                I1                      =>      RegoB,
                I2 (1 downto 0)         =>      "00",
                I2 (17 downto 2)        =>      Immediate,
                I2 (31 downto 18)       =>      "00000000000000",
                I3 (4 downto 0)         =>      Sha,
                I3 (31 downto 5)        => "00000000000000000000000000000",
                ALUBsel                 =>      ALUsB,
                O                       =>      ALU_B
        );
Multiplexor_2X1     :       MUX2x1
Port map (      I0                      =>      ALU_out,
                I1                      =>      Data_in,
                ReginSel                =>      Rins,
                O                       =>      Rinp
        );
Register_Code_in    :       Register32top
Port map (      i                       =>      Op_Data,
                clk                     =>      clock_input,
```

```
                Reset           =>  Rrst(1),
                Enable          =>  Ren(1),
                o               =>  Code_in
        );
Register_Data_in    :   Register32top
Port map (      i               =>  Op_Data,
                clk             =>  clock_input,
                Reset           =>  Rrst(2),
                Enable          =>  Ren(2),
                o               =>  Data_in
        );
Register_Data_out   :   Register32top
Port map (      i               =>  RegoB,
                clk             =>  clock_input,
                Reset           =>  Rrst(3),
                Enable          =>  Ren(0),
                o               =>  Op_Data
        );
Register_Address_out    :   Register32top
Port map (      i               =>  ALU_out,
                clk             =>  clock_input,
                Reset           =>  Rrst(0),
                Enable          =>  Ren(3),
                o               =>  Address
        );
end Structural;
```

Σε αυτό το σημείο τελειώνει ο κώδικας του top module του επεξεργαστή.

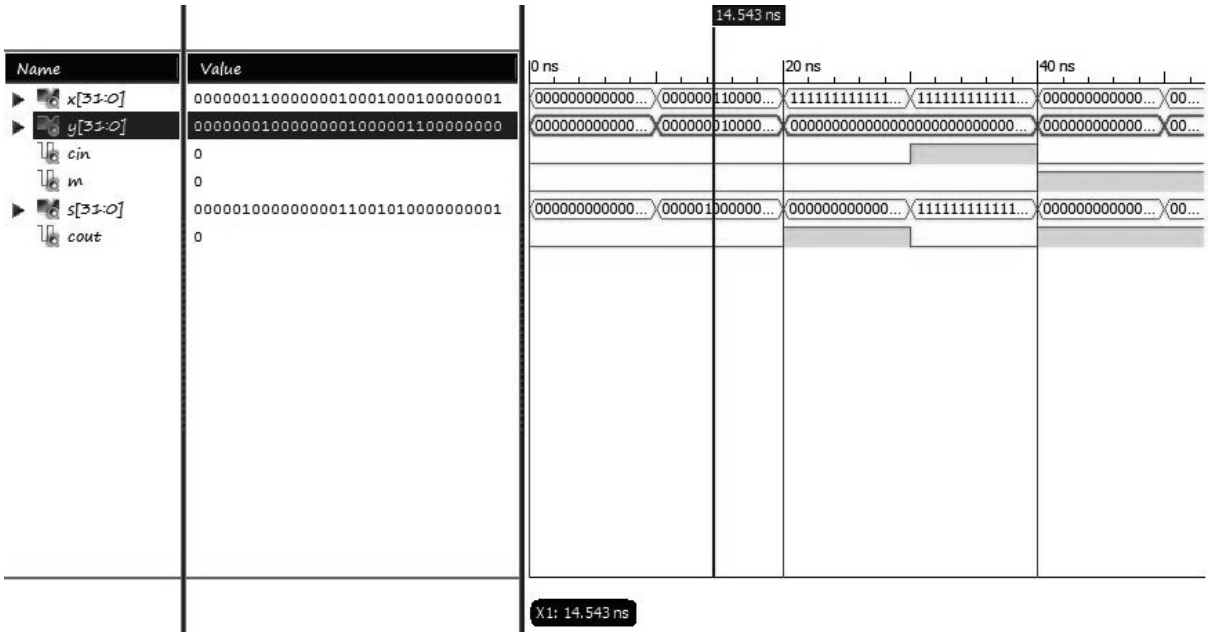
4. Προσομοιώσεις

Το κεφάλαιο αυτό της παρούσας εργασίας περιέχει τις προσομοιώσεις του κώδικα που περιέχεται στο προηγούμενο κεφάλαιο, μέσω των οποίων επιβεβαιώνεται η σωστή λειτουργία του επεξεργαστή.

4.1 Οι προσομοιώσεις της ALU

4.1.1 Η προσομοίωση του αθροιστή - αφαιρέτη της ALU

Η προσομοίωση του αθροιστή – αφαιρέτη της Αριθμητικής και Λογικής Μονάδας, φαίνεται στις παρακάτω εικόνες:



Εικόνα 4.1: Προσομοίωση του αθροιστή – αφαιρέτη της ALU (α)

Για την προσομοίωση που φαίνεται στην εικόνα 4.1(α), γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

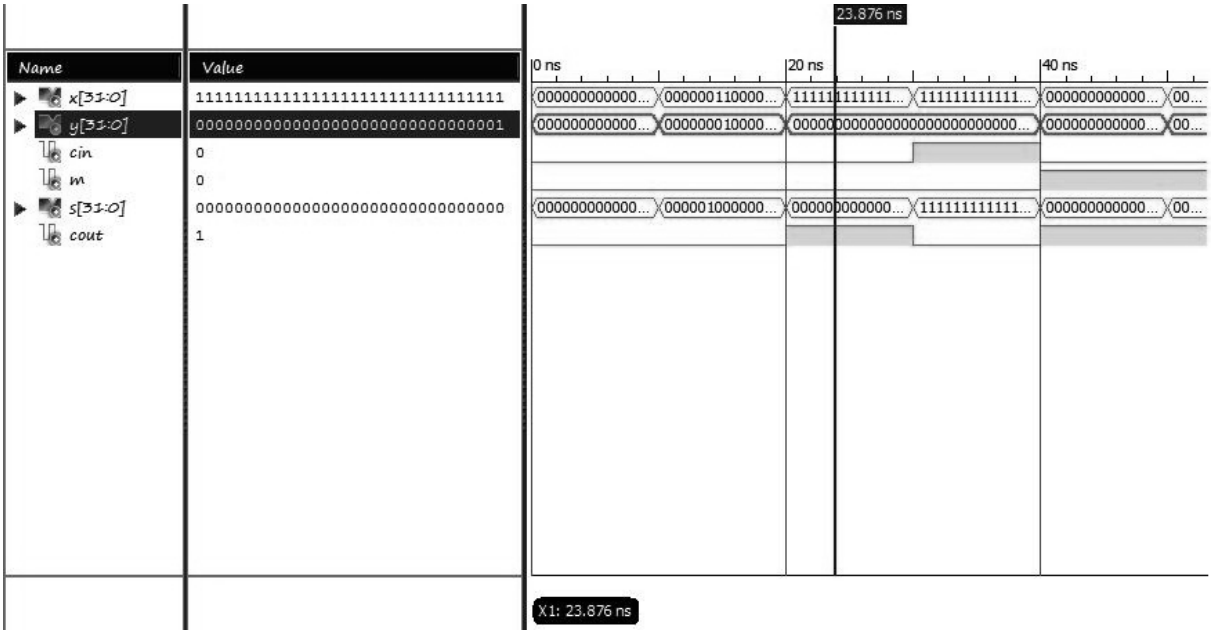
```
X     <=    "00000011000000010001000100000001";
Y     <=    "00000001000000001000001100000000";
M     <=    '0'; --Ο αθροιστής – αφαιρέτης εκτελεί πρόσθεση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο X η τιμή "00000011000000010001000100000001" και στην είσοδο Y η τιμή "00000001000000001000001100000000", ενώ στην είσοδο M δόθηκε μηδενική τιμή. Το αποτέλεσμα "00000100000000011001010000000001" που βγάζει η έξοδος S του αθροιστή – αφαιρέτη, όπως φαίνεται από την εικόνα 4.1, είναι σωστό, αφού:

00000011000000010001000100000001

\+ 00000001000000001000001100000000

00000100000000011001010000000001



Εικόνα 4.2: Προσομοίωση του αθροιστή – αφαιρέτη της ALU (β)

Για την προσομοίωση που φαίνεται στην εικόνα 4.2, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

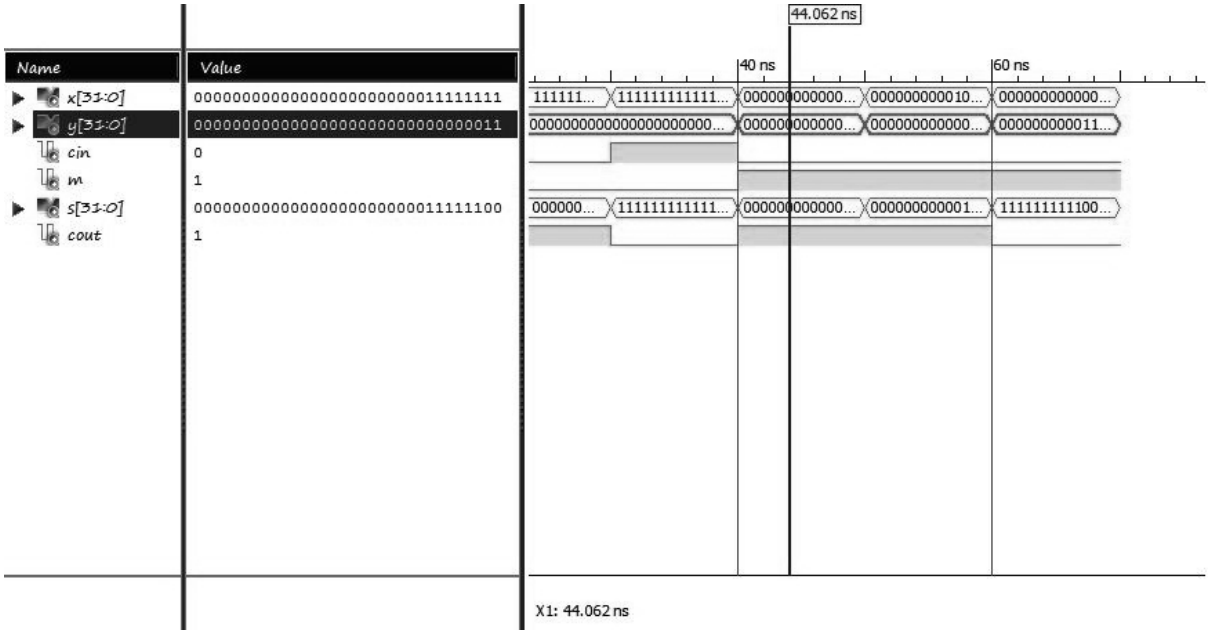
```
X   <=   "11111111111111111111111111111111";
Y   <=   "00000000000000000000000000000001";
M   <=   '0'; --Ο αθροιστής – αφαιρέτης εκτελεί πρόσθεση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο X η τιμή "11111111111111111111111111111111" και στην είσοδο Y η τιμή "00000000000000000000000000000001", ενώ στην είσοδο M δόθηκε μηδενική τιμή. Το αποτέλεσμα "00000000000000000000000000000000" που βγάζει η έξοδος S του αθροιστή – αφαιρέτη και η τιμή '1' της εξόδου Cout, όπως φαίνεται από την εικόνα 4.2, είναι σωστά, αφού:

11111111111111111111111111111111

\+ 00000000000000000000000000000001

10000000000000000000000000000000



Εικόνα 4.3: Προσομοίωση του αθροιστή – αφαιρέτη της ALU (γ)

Για την προσομοίωση που φαίνεται στην εικόνα 4.1(γ), γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

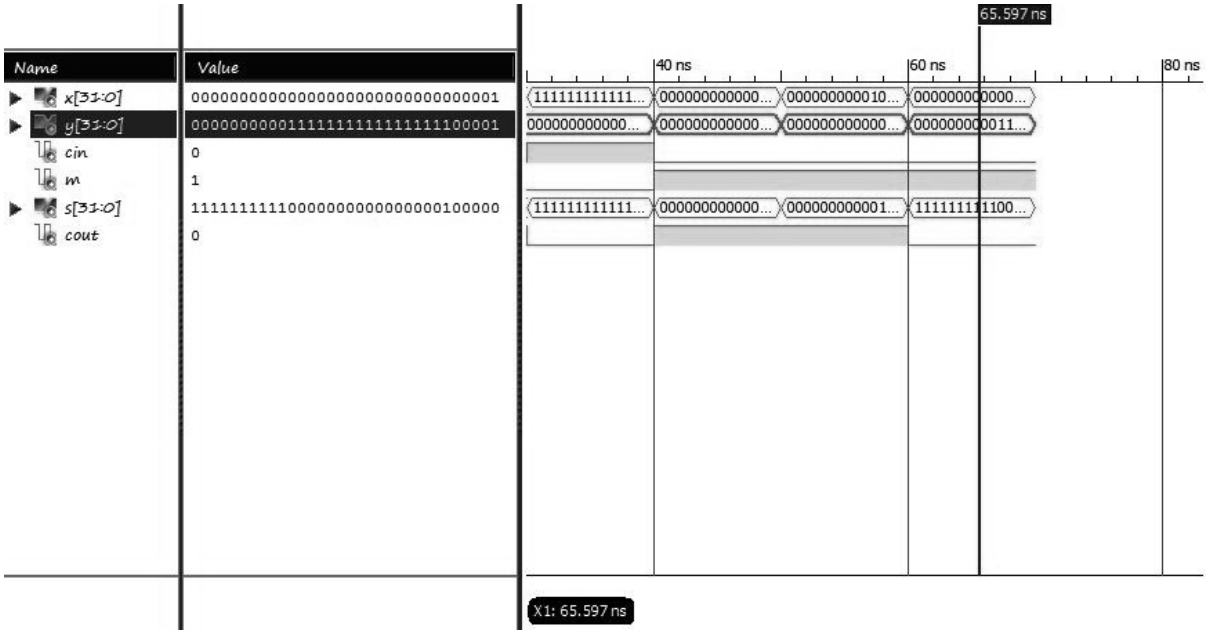
```
X   <=   "00000000000000000000000011111111";
Y   <=   "00000000000000000000000000000011";
M   <=   '1'; --Ο αθροιστής – αφαιρέτης εκτελεί αφαίρεση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο X η τιμή "00000000000000000000000011111111" και στην είσοδο Y η τιμή "00000000000000000000000000000011", ενώ στην είσοδο M δόθηκε η τιμή '1'. Το αποτέλεσμα "00000000000000000000000011111100" που βγάζει η έξοδος S του αθροιστή – αφαιρέτη, όπως φαίνεται από την εικόνα 4.3, είναι σωστό, αφού:

00000000000000000000000011111111

\- 00000000000000000000000000000011

00000000000000000000000011111100



Εικόνα 4.4: Προσομοίωση του αθροιστή – αφαιρέτη της ALU (δ)

Για την προσομοίωση που φαίνεται στην εικόνα 4.4 γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

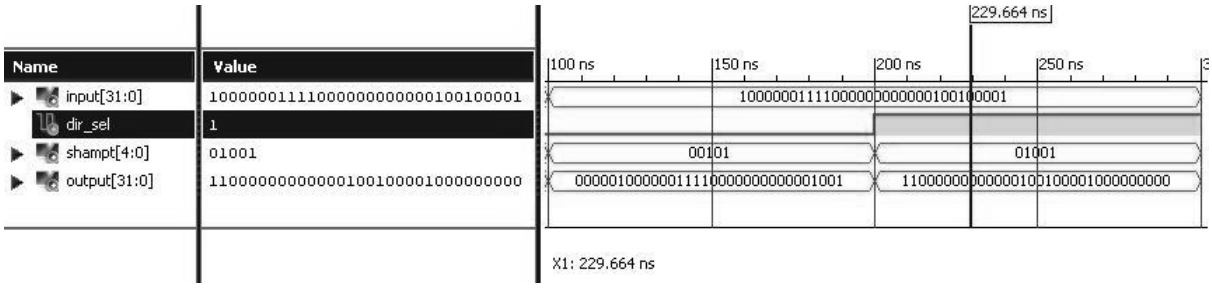
```
X    <=    "00000000000000000000000000000001";
Y    <=    "00000000011111111111111111100001";
M    <=    '1'; --Ο αθροιστής – αφαιρέτης εκτελεί αφαίρεση
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο X η τιμή "00000000000000000000000000000001" και στην είσοδο Y η τιμή "00000000011111111111111111100001", ενώ στην είσοδο M δόθηκε η τιμή '1'. Το αποτέλεσμα "11111111110000000000000000100000" που βγάζει η έξοδος S του αθροιστή – αφαιρέτη, όπως φαίνεται από την εικόνα 4.4, είναι σωστό, αφού:

$$\begin{array}{rl} & 00000000000000000000000000000001 \\ - & \underline{00000000011111111111111111100001} \\ & 11111111110000000000000000100000 \end{array}$$

4.1.2 Η προσομοίωση του κυκλώματος ολίσθησης της ALU

Η προσομοίωση του κυκλώματος ολίσθησης της ALU, φαίνεται στην παρακάτω εικόνα:



Εικόνα 4.5: Προσομοίωση του κυκλώματος ολίσθησης της ALU

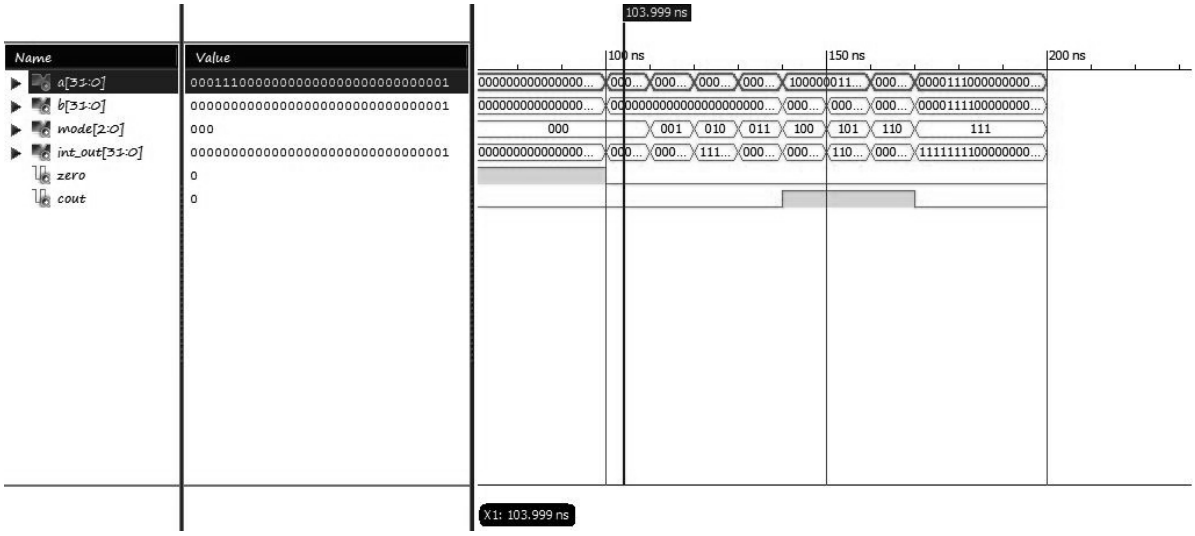
Για την προσομοίωση που φαίνεται στην εικόνα 4.5, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
 Input      <=   "10000001111000000000000100100001";
dir_sel     <=   '0'; --Δεξιά ολίσθηση
shamt  <=   "00101"; --Ολίσθηση κατά 5 θέσεις
wait for 100 ns;
Input       <=   "10000001111000000000000100100001";
dir_sel     <=   '1';--Αριστερή ολίσθηση
shamt  <=   "01001";--Ολίσθηση κατά 9 θέσεις
wait for 100 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, αρχικά δόθηκε στην είσοδο Input του κυκλώματος ολίσθησης η τιμή "10000001111000000000000100100001", στην είσοδο dir_sel η τιμή '0' και στην είσοδο shamt η τιμή "00101". Η τιμή "00000100000011110000000000001001" που βγάζει η έξοδος output του κυκλώματος ολίσθησης, όπως φαίνεται από την 4.5, είναι σωστή, καθώς είναι η τιμή της εισόδου Input ολισθημένη κατά 5 θέσεις δεξιά, που είναι και το ζητούμενο. Στη συνέχεια, μετά από χρόνο 100ns, δόθηκε στην είσοδο Input του κυκλώματος ολίσθησης και πάλι η τιμή "10000001111000000000000100100001", στην είσοδο dir_sel η τιμή '1' και στην είσοδο shamt την τιμή "01001". Η τιμή "11000000000000100100001000000000" που βγάζει η έξοδος output του κυκλώματος ολίσθησης, όπως φαίνεται από την 4.5, είναι σωστή, καθώς είναι η τιμή της εισόδου Input ολισθημένη κατά 9 θέσεις αριστερά, που είναι και το ζητούμενο.

4.1.3 Η προσομοίωση της Αριθμητικής και Λογικής Μονάδας

Η προσομοίωση της Αριθμητικής και Λογικής Μονάδας, φαίνεται στις παρακάτω εικόνες:



Εικόνα 4.6: Προσομοίωση της ALU (α).

Για την προσομοίωση που φαίνεται στην εικόνα 4.6, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

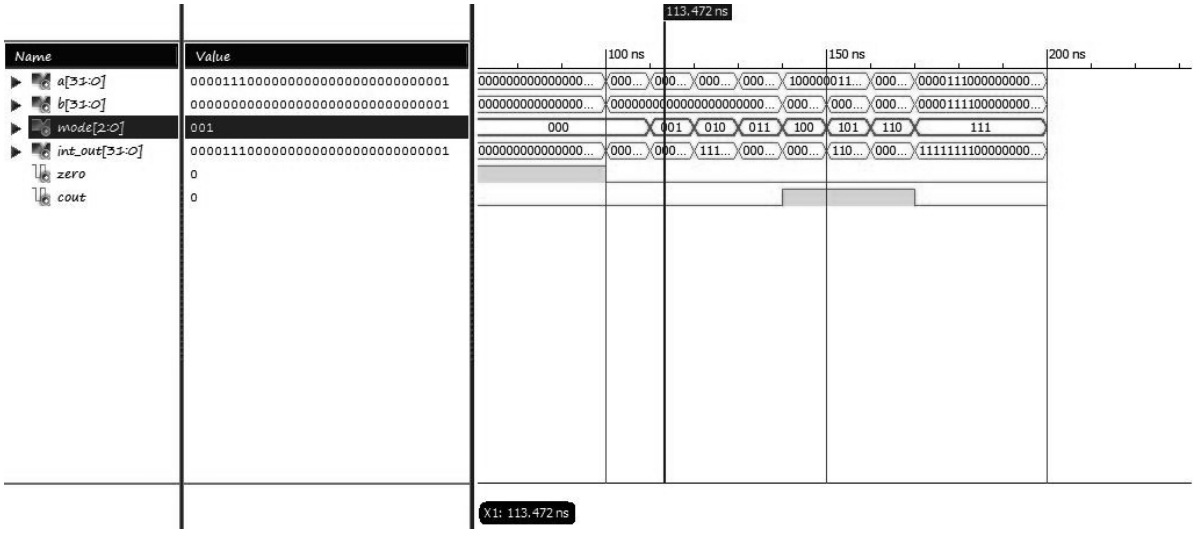
```
A     <=    "00011100000000000000000000000001";
B     <=    "00000000000000000000000000000001";
mode  <= "000"; --Λογική πράξη and
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "00011100000000000000000000000001" και στην είσοδο B η τιμή "00000000000000000000000000000001", ενώ στην είσοδο mode δόθηκε η τιμή "000". Το αποτέλεσμα "00000000000000000000000000000001" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.6, είναι σωστό, αφού:

00011100000000000000000000000001

and 00000000000000000000000000000001

00000000000000000000000000000001



Εικόνα 4.7: Προσομοίωση της ALU (β).

Για την προσομοίωση που φαίνεται στην εικόνα 4.7, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

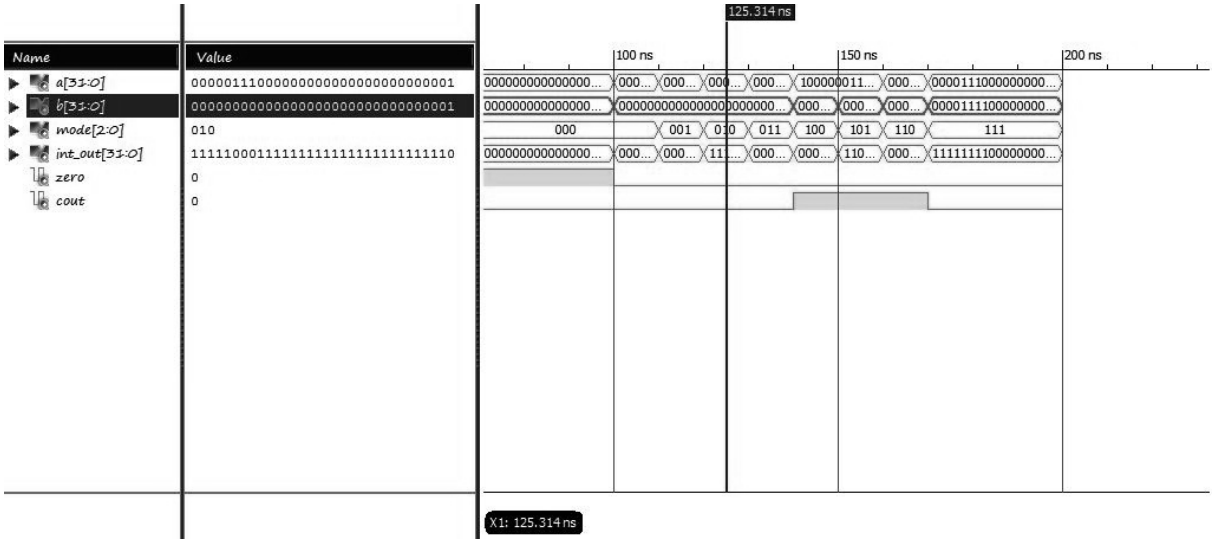
```
A     <=    "00001110000000000000000000000001";
B     <=    "00000000000000000000000000000001";
mode  <= "001"; --Λογική πράξη or
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "00001110000000000000000000000001" και στην είσοδο B η τιμή "00000000000000000000000000000001", ενώ στην είσοδο mode δόθηκε η τιμή "001". Το αποτέλεσμα "00001110000000000000000000000001" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.7, είναι σωστό, αφού:

00001110000000000000000000000001

or 00000000000000000000000000000001

00001110000000000000000000000001



Εικόνα 4.8: Προσομοίωση της ALU (γ).

Για την προσομοίωση που φαίνεται στην εικόνα 4.8, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

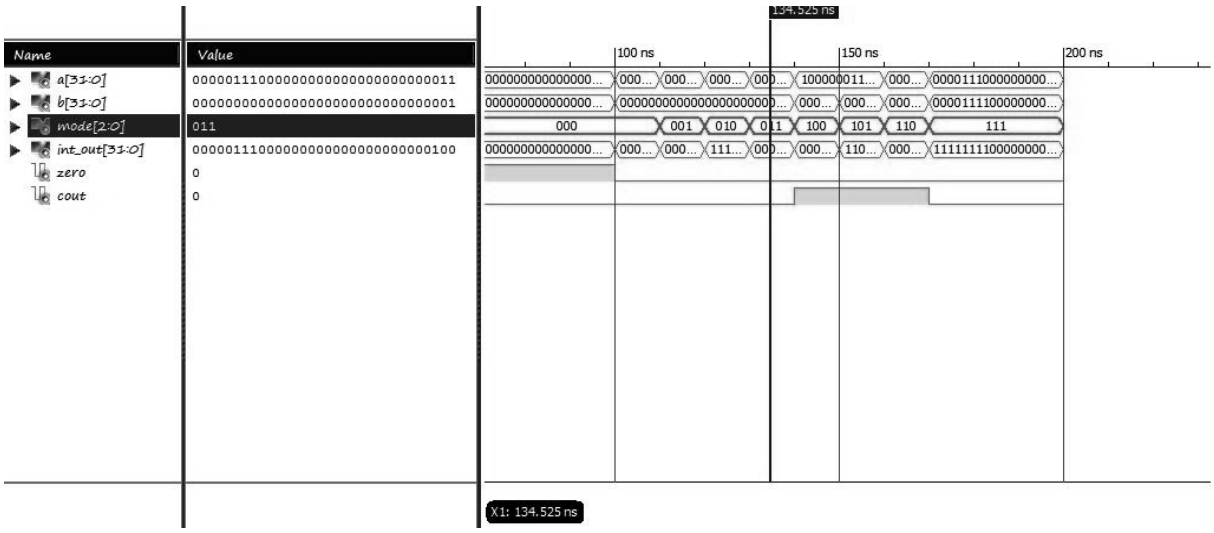
```
A     <=    "00000111000000000000000000000001";
B     <=    "00000000000000000000000000000001";
mode  <= "010"; --Λογική πράξη nor
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "00000111000000000000000000000001" και στην είσοδο B η τιμή "00000000000000000000000000000001", ενώ στην είσοδο mode δόθηκε η τιμή "010". Το αποτέλεσμα "11111000111111111111111111111110" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.8, είναι σωστό, αφού:

00000111000000000000000000000001

nor 00000000000000000000000000000001

11111000111111111111111111111110



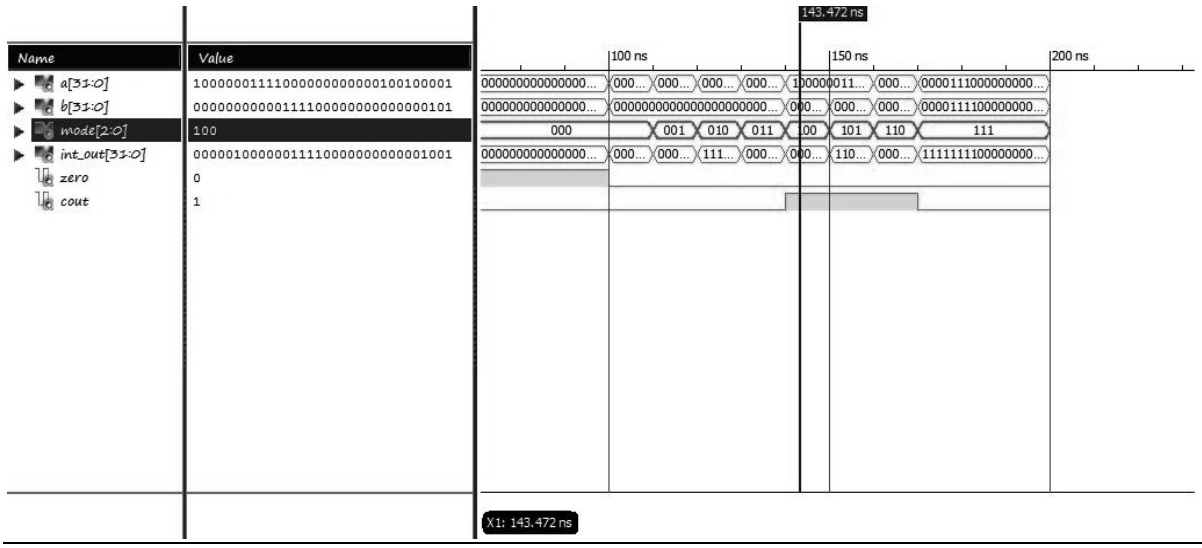
Εικόνα 4.9: Προσομοίωση της ALU (δ).

Για την προσομοίωση που φαίνεται στην εικόνα 4.9, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
A      <=     "00000111000000000000000000000001";
B      <=     "00000000000000000000000000000001";
mode   <= "011"; --Πρόσθεση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "00000111000000000000000000000011" και στην είσοδο B η τιμή "00000000000000000000000000000001", ενώ στην είσοδο mode δόθηκε η τιμή "011". Το αποτέλεσμα "00000111000000000000000000000100" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.9, είναι σωστό, αφού:

```
  00000111000000000000000000000011
+ 00000000000000000000000000000001
  00000111000000000000000000000100
```

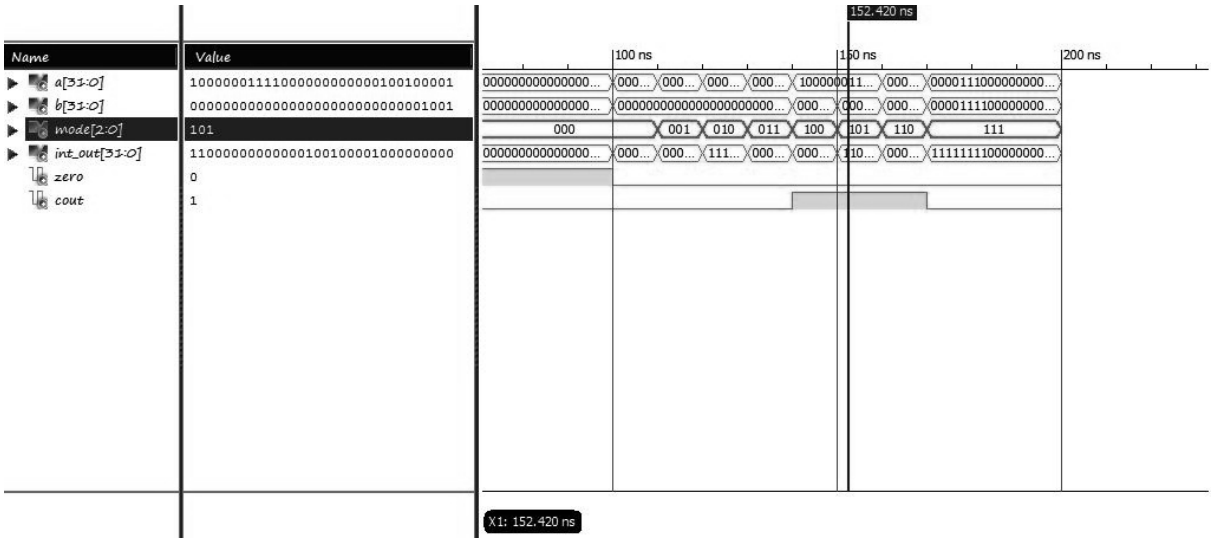


Εικόνα 4.10: Προσομοίωση της ALU (ε).

Για την προσομοίωση που φαίνεται στην εικόνα 4.10, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
A     <=    "10000001111000000000000100100001";
B     <=    "00000000000111100000000000000101";
mode  <= "100"; --Δεξιά ολίσθηση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "10000001111000000000000100100001" και στην είσοδο B η τιμή "00000000000111100000000000000101", ενώ στην είσοδο mode δόθηκε η τιμή "100". Το αποτέλεσμα "00000100000011111000000000001001" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.10, είναι σωστό, αφού είναι η τιμή της εισόδου A ολισθημένη κατά 5 θέσεις δεξιά, που είναι και το ζητούμενο.

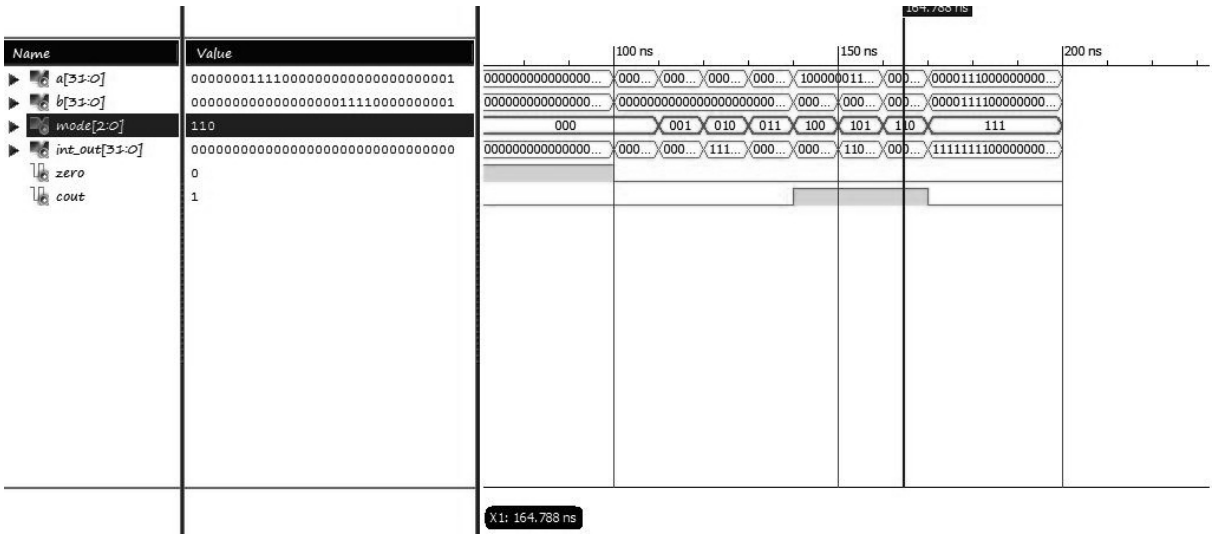


Εικόνα 4.11: Προσομοίωση της ALU (στ).

Για την προσομοίωση που φαίνεται στην εικόνα 4.11, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
A     <=    "10000001111000000000000100100001";
B     <=    " 00000000000000000000000000001001";
mode  <= "101"; --Αριστερή ολίσθηση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "10000001111000000000000100100001" και στην είσοδο B η τιμή "00000000000000000000000000001001", ενώ στην είσοδο mode δόθηκε η τιμή "101". Το αποτέλεσμα "11000000000000100100001000000000" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.11, είναι σωστό, αφού είναι η τιμή της εισόδου A ολισθημένη κατά 9 θέσεις αριστερά, που είναι και το ζητούμενο.

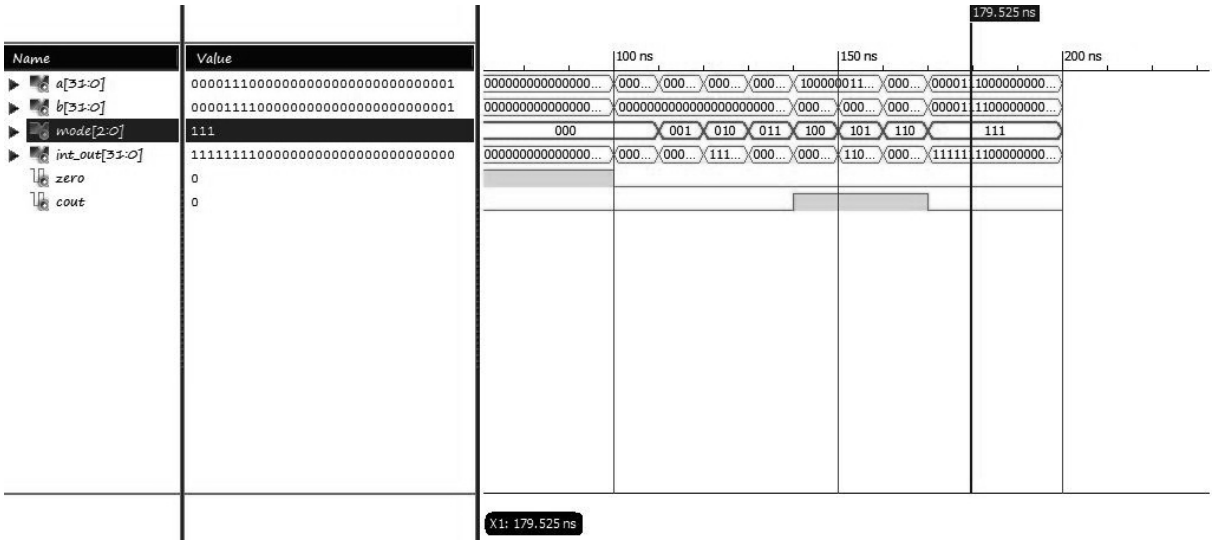


Εικόνα 4.12: Προσομοίωση της ALU (ζ).

Για την προσομοίωση που φαίνεται στην εικόνα 4.12, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
A      <=     "00000001111000000000000000000001";
B      <=     "00000000000000000011110000000001";
mode   <= "110"; --Αριστερή ολίσθηση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "00000001111000000000000000000001" και στην είσοδο B η τιμή "00000000000000000011110000000001", ενώ στην είσοδο mode δόθηκε η τιμή "110". Το αποτέλεσμα "00000000000000000000000000000000" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.12, είναι σωστό, αφού είναι η τιμή της εισόδου A είναι μεγαλύτερη από την τιμή της εισόδου B.



Εικόνα 4.13: Προσομοίωση της ALU (η).

Για την προσομοίωση που φαίνεται στην εικόνα 4.13, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
A     <=    "00001110000000000000000000000001";
B     <=    "00001111000000000000000000000001";
mode  <= "111"; --Αφαίρεση
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο A η τιμή "00001110000000000000000000000001" και στην είσοδο B η τιμή "00001111000000000000000000000001", ενώ στην είσοδο mode δόθηκε η τιμή "111". Το αποτέλεσμα "11111111000000000000000000000000" που βγάζει η έξοδος Int_out της ALU, όπως φαίνεται από την εικόνα 4.13, είναι σωστό, αφού:

00001110000000000000000000000001

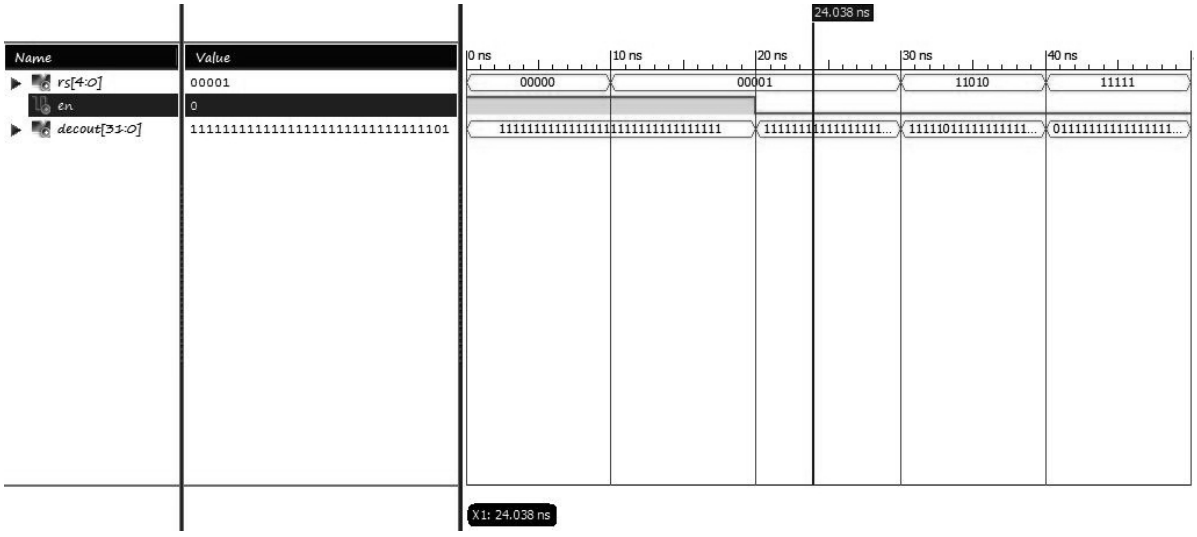
\- 00001111000000000000000000000001

11111111000000000000000000000000

4.2 Η προσομοίωση του Register File

4.2.1 Η προσομοίωση του αποκωδικοποιητή

Η προσομοίωση του αποκωδικοποιητή του Register File, φαίνεται στις παρακάτω εικόνες:

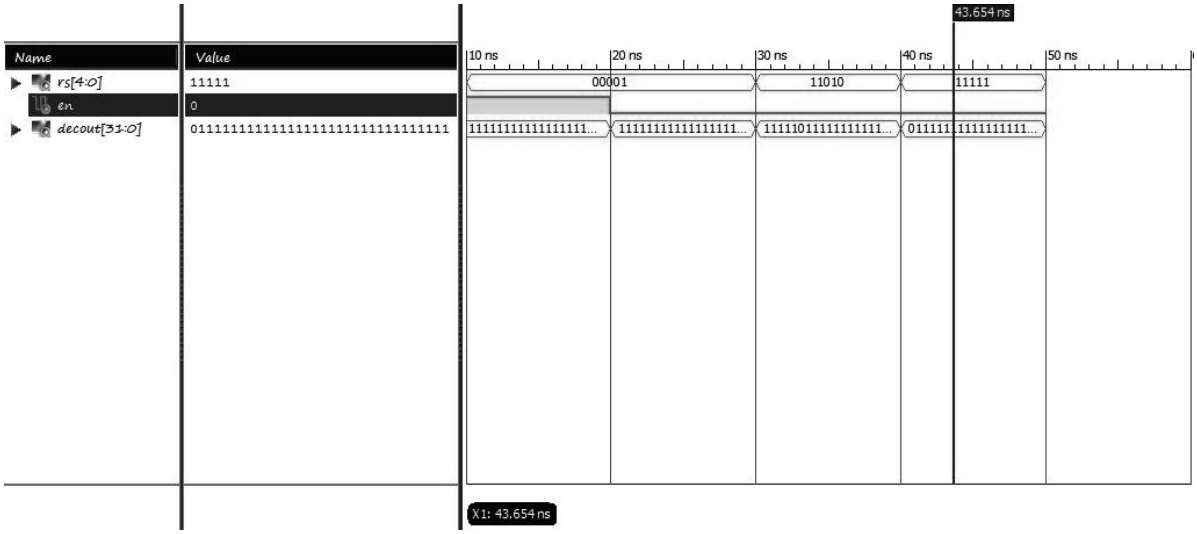


Εικόνα 4.14: Προσομοίωση του αποκωδικοποιητή του Register File(α)

Για την προσομοίωση που φαίνεται στην εικόνα 4.14, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
rs    <=    "00001"; --Επιλέγεται ο καταχωρητής 1
en    <=    '0';-- Ενεργοποιείται ο αποκωδικοποιητής
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο rs η τιμή "00001" για να γίνει επιλογή του καταχωρητή 1 και στην είσοδο en η τιμή '0' για ενεργοποιηθεί ο αποκωδικοποιητής (η είσοδος αυτή είναι ενεργή στο 0). Η τιμή "11111111111111111111111111111101" που βγάζει η έξοδος decout του αποκωδικοποιητή, όπως φαίνεται από την εικόνα 4.14, είναι σωστή, αφού έτσι θα ενεργοποιηθεί ο καταχωρητής 1.



Εικόνα 4.15: Προσομοίωση του αποκωδικοποιητή του Register File(β)

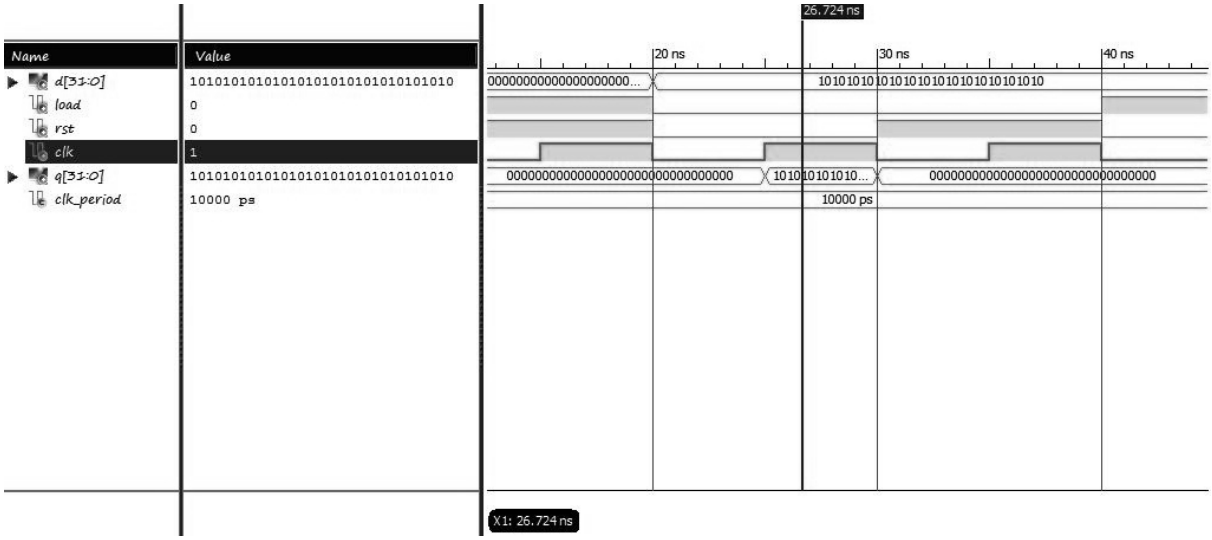
Για την προσομοίωση που φαίνεται στην εικόνα 4.15, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
rs      <=      "11111"; --Επιλέγεται ο καταχωρητής 31
en      <=      '0';-- Ενεργοποιείται ο αποκωδικοποιητής
wait for 10 ns;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο rs η τιμή "11111" για να γίνει επιλογή του καταχωρητή 31 και στην είσοδο en η τιμή '0' για να ενεργοποιηθεί ο αποκωδικοποιητής (η είσοδος αυτή είναι ενεργή στο 0). Η τιμή "01111111111111111111111111111111" που βγάζει η έξοδος decout του αποκωδικοποιητή, όπως φαίνεται από την εικόνα 4.15, είναι σωστή, αφού έτσι θα ενεργοποιηθεί ο καταχωρητής 31.

4.2.2 Η προσομοίωση του καταχωρητή των 32 bit του Register File

Η προσομοίωση του καταχωρητή των 32 bits του Register File που υλοποιήθηκε, φαίνεται στην παρακάτω εικόνα:



Εικόνα 4.16: Προσομοίωση του καταχωρητή των 32 bits του Register File

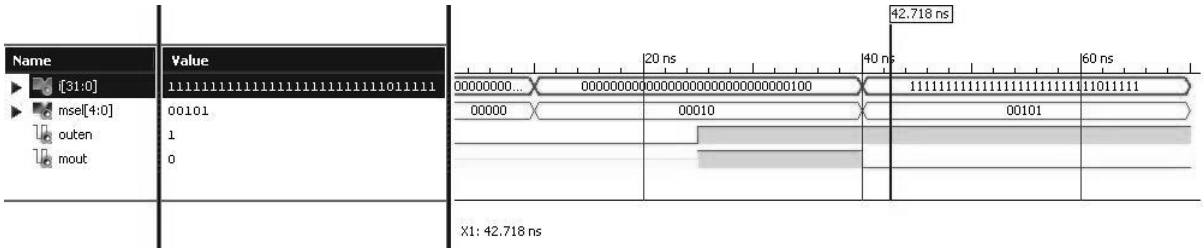
Για την προσομοίωση που φαίνεται στην εικόνα 4.16, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
wait for clk_period;
Rst     <=      '1';--Μηδενισμός καταχωρητή
wait for clk_period;
Load    <=      '0';--Ενεργοποίηση φόρτωσης τιμής
Rst     <=      '0';--Απενεργοποίηση σήματος Reset
D       <=      "10101010101010101010101010101010";
wait for clk_period;
Rst     <=      '1';--Μηδενισμός καταχωρητή
wait for clk_period;
Load    <=      '1';--Απενεργοποίηση φόρτωσης τιμής
Rst     <=      '0';--Απενεργοποίηση σήματος Reset
D       <=      "10101010101010101010101010101010";
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, αρχικά δόθηκε στην είσοδο Rst η τιμή '1' για να μηδενιστεί ο καταχωρητής. Στη συνέχεια, μετά το πέρας μιας περιόδου του ρολογιού, δόθηκε στην είσοδο Load η τιμή '0' και στην είσοδο Rst η τιμή '0' για ενεργοποιηθεί η φόρτωση τιμής του καταχωρητή και να απενεργοποιηθεί το σήμα Reset. Επίσης, στην είσοδο D δόθηκε η τιμή "10101010101010101010101010101010". Η τιμή "10101010101010101010101010101010" που βγάζει η έξοδος Q του καταχωρητή, όπως φαίνεται από την εικόνα 4.16, είναι σωστή, αφού είναι η τιμή που παίρνει στην είσοδό του. Στη συνέχεια, δόθηκε στην είσοδο Rst η τιμή '1' για να μηδενιστεί ξανά ο καταχωρητής. Μετά το πέρας μιας περιόδου του ρολογιού, δόθηκε στην είσοδο Load η τιμή '1' και στην είσοδο Rst η τιμή '0' για να απενεργοποιηθεί η φόρτωση τιμής και το σήμα Reset του καταχωρητή. Η τιμή "00000000000000000000000000000000" που βγάζει η έξοδος Q του καταχωρητή, όπως φαίνεται από την εικόνα 4.16, είναι σωστή, αφού έχει μηδενιστεί ο καταχωρητής και στη συνέχεια, ενώ είναι απενεργοποιημένο το σήμα Reset, είναι ανενεργή η φόρτωση τιμής του, οπότε και δε φορτώνεται η τιμή της εισόδους του Q.

4.2.3 Η προσομοίωση του πολυπλέκτη 32x1 του Register File

Η προσομοίωση του πολυπλέκτη 32x1 του Register File, φαίνεται στην παρακάτω εικόνα:



Εικόνα 4.17: Προσομοίωση του πολυπλέκτη 32x1 του Register File

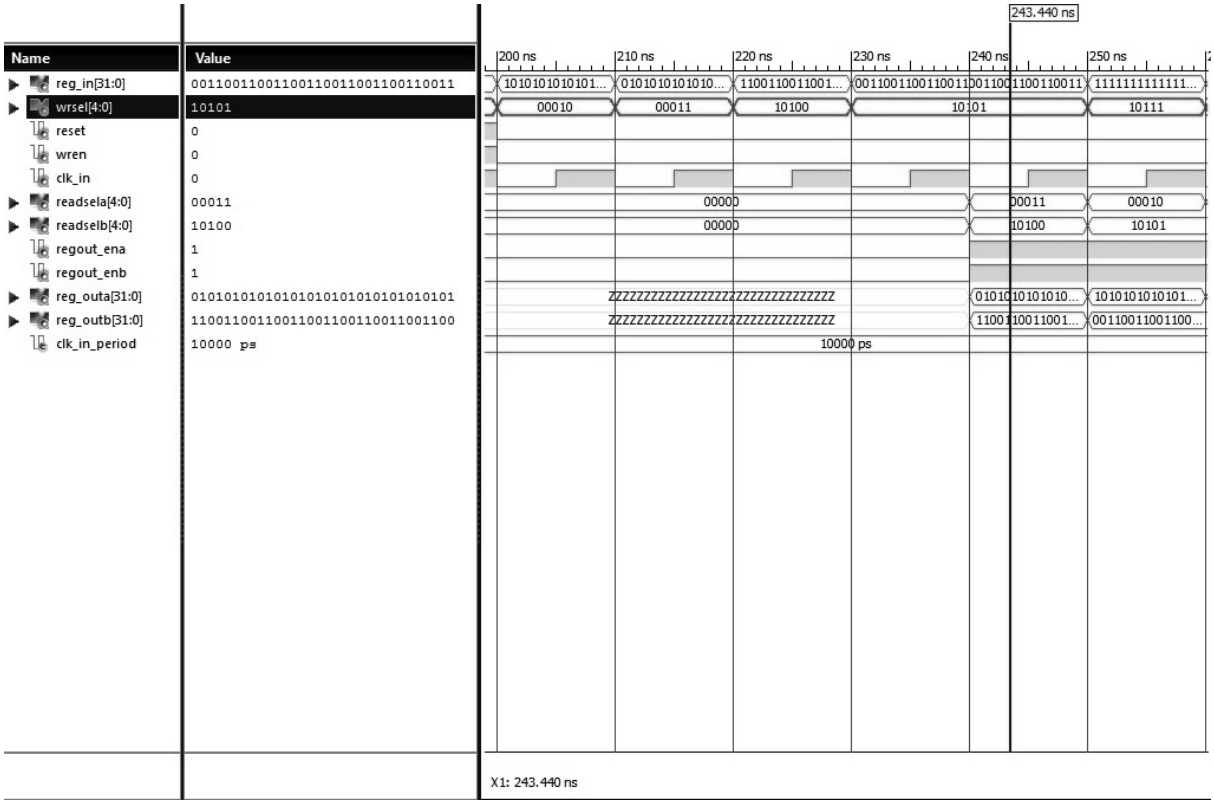
Για την προσομοίωση που φαίνεται στην εικόνα 4.17, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
I       <=      "00000000000000000000000000000100";
msel    <=      "00010"; --Επιλογή της εισόδου I(2)
Outen   <=      '0'; --Η έξοδος του πολυπλέκτη είναι απενεργοποιημένη
wait for 15 ns;
I       <=      "00000000000000000000000000000100";
msel    <=      "00010";--Επιλογή της εισόδου I(2)
Outen   <=      '1'; --Ενεργοποίηση της εξόδου του πολυπλέκτη
wait for 15 ns;
I       <=      "11111111111111111111111111011111";
msel    <=      "00101";--Επιλογή της εισόδου I(5)
wait for 15 ns; --Ενεργοποίηση της εξόδου του πολυπλέκτη
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, αρχικά δόθηκε στην είσοδο I του πολυπλέκτη η τιμή "00000000000000000000000000000100", στην είσοδο msel η τιμή "00010" για την επιλογή της εισόδου I(2) και το σήμα ενεργοποίησης της εξόδου του πολυπλέκτη παρέμεινε ανενεργό. Όπως φαίνεται από την εικόνα 4.17, εδώ ο πολυπλέκτης δούλεψε σωστά, καθώς δεν έβγαλε καμία είσοδο στην έξοδό του, αλλά η έξοδός του παρέμεινε στην τιμή 'Z' που είχε από πριν. Στη συνέχεια, μετά το πέρας χρόνου 25ns, δόθηκε στην είσοδο I του πολυπλέκτη ξανά η τιμή "00000000000000000000000000000100", στην είσοδο msel την τιμή "00010" για την επιλογή της εισόδου I(2) και ενεργοποιήθηκε το σήμα ενεργοποίησης της εξόδου του πολυπλέκτη, με την τιμή '1' που του δόθηκε. Όπως φαίνεται από την εικόνα 4.17, και εδώ ο πολυπλέκτης δούλεψε σωστά, καθώς η έξοδός του πήρε την τιμή '1', την οποία έχει μόνο η είσοδός του I(2). Στη συνέχεια, πάλι μετά το πέρας χρόνου 25ns, δόθηκε στην είσοδο I του πολυπλέκτη η τιμή "11111111111111111111111111011111" αυτή τη φορά, στην είσοδο msel η τιμή "00101" για την επιλογή της εισόδου I(5) και παρέμεινε ενεργό το σήμα ενεργοποίησης της εξόδου του. Όπως φαίνεται από την εικόνα 4.17, και εδώ ο πολυπλέκτης δούλεψε σωστά, καθώς η έξοδός του πήρε την τιμή '0', την οποία έχει μόνο η είσοδός του I(5).

4.2.4 Η προσομοίωση του top module του Register File

Η προσομοίωση του top module του Register, φαίνεται στις παρακάτω εικόνες:



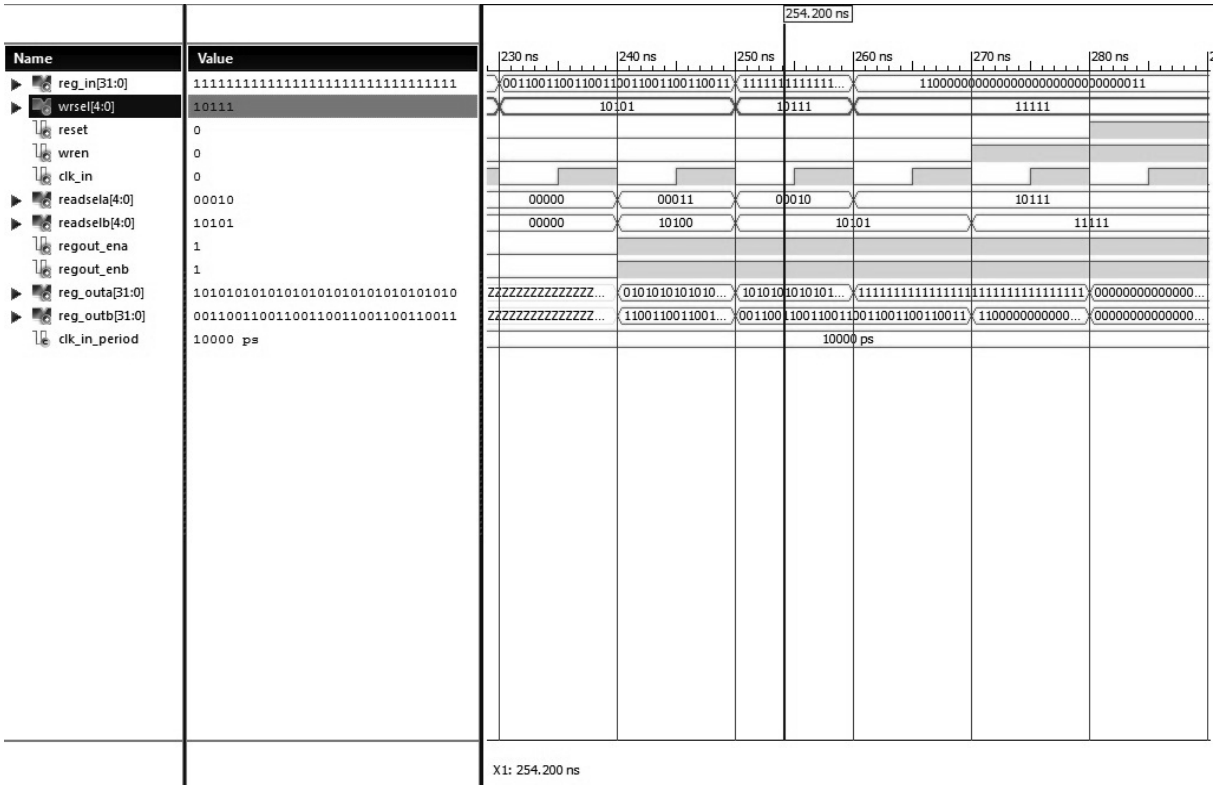
Εικόνα 4.18: Προσομοίωση του top module του Register File(α)

Για την προσομοίωση που φαίνεται στην εικόνα 4.18, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
Reset <= '1'; --Μηδενισμός καταχωρητών
wait for clk_in_period*10; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
--10 περιόδους ρολογιού
Reset <= '0'; --Απενεργοποίηση του σήματος Reset των
--καταχωρητών
wrsel <= "00010"; --Επιλογή εγγραφής του καταχωρητή 2
wren <= '0'; --Ενεργοποίηση εγγραφής καταχωρητή
Reg_in <= "10101010101010101010101010101010";
wait for clk_in_period; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --1 περίοδο ρολογιού
wrsel <= "00011";--Επιλογή εγγραφής του καταχωρητή 3
Reg_in <= "01010101010101010101010101010101";
wait for clk_in_period; ; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --1 περίοδο ρολογιού
wrsel <= "10100"; --Επιλογή εγγραφής του καταχωρητή 20
Reg_in <= "11001100110011001100110011001100";
wait for clk_in_period; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --1 περίοδο ρολογιού
```

```
wrsel       <=   "10101"; --Επιλογή εγγραφής του καταχωρητή 20
Reg_in      <=   "00110011001100110011001100110011";
wait for clk_in_period; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --1 περίοδο ρολογιού
readselA    <=   "00011"; --Επιλογή ανάγνωσης του καταχωρητή 3
readselB    <=   "10100"; --Επιλογή ανάγνωσης του καταχωρητή 20
Regout_enA  <=   '1'; --Ενεργοποίηση της εξόδου A του Register File
Regout_enB  <=   '1'; --Ενεργοποίηση της εξόδου B του Register File
wait for clk_in_period;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, για τις 10 πρώτες περιόδους του σήματος ρολογιού, δόθηκε στην είσοδο Reset η τιμή '1' για μηδενιστούν οι καταχωρητές. Στη συνέχεια, δόθηκε στην είσοδο Reset η τιμή '0', στην είσοδο wrsel η τιμή "00010", για να γίνει επιλογή του καταχωρητή 2, στην είσοδο wren η τιμή '0' για να ενεργοποιηθεί η εγγραφή του καταχωρητή, και στην είσοδο Reg_in η τιμή "10101010101010101010101010101010", που είναι η τιμή που θα αποθηκευτεί στον καταχωρητή 2. Στη συνέχεια, μετά το πέρας μιας περιόδου του σήματος ρολογιού, δόθηκε στην είσοδο wrsel η τιμή "00011", για να γίνει επιλογή του καταχωρητή 3 και στην είσοδο Reg_in η τιμή "01010101010101010101010101010101", που είναι η τιμή που θα αποθηκευτεί στον καταχωρητή. Στη συνέχεια, μετά το πέρας μιας περιόδου του σήματος ρολογιού, δόθηκε στην είσοδο wrsel η τιμή "10100", για να γίνει επιλογή του καταχωρητή 20 και στην είσοδο Reg_in την τιμή "11001100110011001100110011001100", που είναι η τιμή που θα αποθηκευτεί στον καταχωρητή. Στη συνέχεια, μετά το πέρας μιας περιόδου του σήματος ρολογιού, δόθηκε στην είσοδο wrsel η τιμή "10101", για να γίνει επιλογή του καταχωρητή 21 και στην είσοδο Reg_in η τιμή "00110011001100110011001100110011", που είναι η τιμή που θα αποθηκευτεί στον καταχωρητή. Τέλος, μετά το πέρας μιας περιόδου του σήματος ρολογιού, δόθηκε στην είσοδο readselA η τιμή "00011", για να γίνει επιλογή του καταχωρητή 3, στην είσοδο readselB η τιμή "10100", για να γίνει επιλογή του καταχωρητή 20 και στις εισόδους Regout_enA και Regout_enB η τιμή '1' για να ενεργοποιηθούν οι 2 έξοδοι του Register File. Όπως φαίνεται από την εικόνα 4.18, εδώ το Register File δούλεψε σωστά, καθώς στην έξοδο A έβγαλε την τιμή που αποθηκεύτηκε στον καταχωρητή 3 και στην έξοδο B την τιμή που αποθηκεύτηκε στον καταχωρητή 20, που είναι και το ζητούμενο.

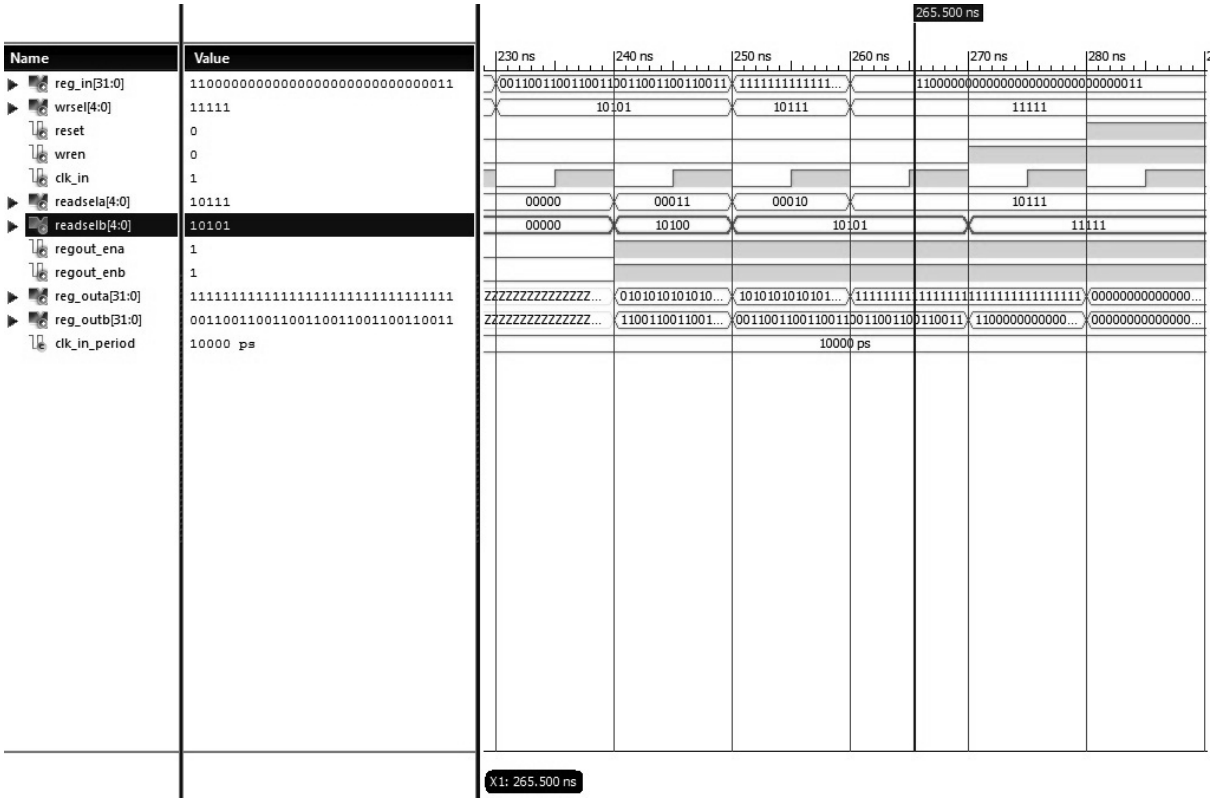


Εικόνα 4.19: Προσομοίωση του top module του Register File(β)

Για την προσομοίωση που φαίνεται στην εικόνα 4.19, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
readselA    <=    "00010"; --Επιλογή του καταχωρητή 2
readselB    <=    "10101"; --Επιλογή του καταχωρητή 21
wrsel       <=    "10111";--Επιλογή του καταχωρητή 23
Reg_in      <=    "11111111111111111111111111111111";
wait for clk_in_period;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο readselA η τιμή "00010", για να γίνει επιλογή του καταχωρητή 2, στην είσοδο readselB η τιμή "10101", για να γίνει επιλογή του καταχωρητή 21, στην είσοδο wrsel η τιμή "10111", για να γίνει επιλογή του καταχωρητή 23, και στην είσοδο Reg_in η τιμή "11111111111111111111111111111111", που είναι η τιμή που θα αποθηκευτεί στον καταχωρητή.. Όπως φαίνεται από την εικόνα 4.19, και εδώ το Register File δούλεψε σωστά, καθώς στην έξοδο A έβγαλε την τιμή που αποθηκεύτηκε προηγουμένως στον καταχωρητή 2 και στην έξοδο B την τιμή που αποθηκεύτηκε προηγουμένως στον καταχωρητή 21, που είναι και το ζητούμενο.

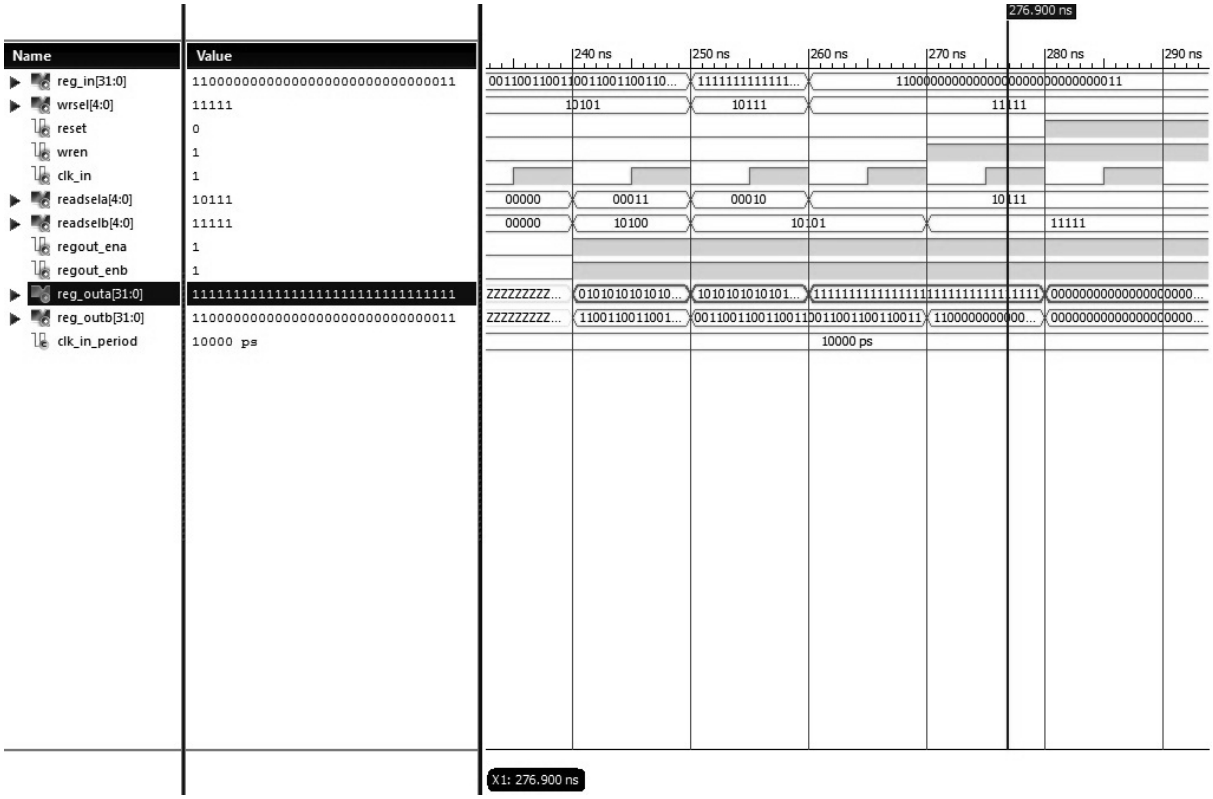


Εικόνα 4.20: Προσομοίωση του top module του Register File(γ)

Για την προσομοίωση που φαίνεται στην εικόνα 4.20, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
readselA    <=    "10111"; --Επιλογή του καταχωρητή 23
wrsel       <=    "11111";--Επιλογή του καταχωρητή 31
Reg_in      <=    "11000000000000000000000000000011";
wait for clk_in_period;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο readselA την τιμή "10111", για να γίνει επιλογή του καταχωρητή 23, στην είσοδο wrsel η τιμή "11111", για να γίνει επιλογή του καταχωρητή 31, και στην είσοδο Reg_in η τιμή "11000000000000000000000000000011", που είναι η τιμή που θα αποθηκευτεί στον καταχωρητή. Όπως φαίνεται από την εικόνα 4.20, και εδώ το Register File δούλεψε σωστά, καθώς στην έξοδο A έβγαλε την τιμή που αποθηκεύυηκε προηγουμένως στον καταχωρητή 23, που είναι και το ζητούμενο.



Εικόνα 4.21: Προσομοίωση του top module του Register File(δ)

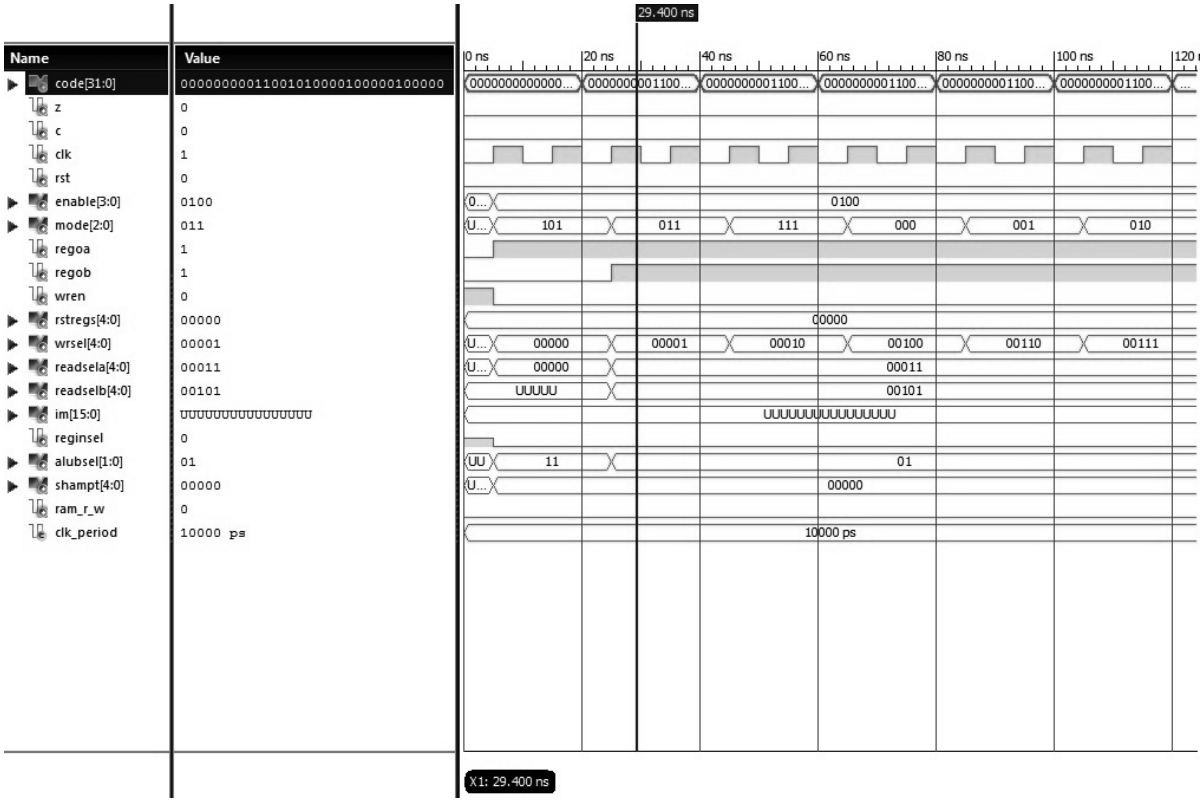
Για την προσομοίωση που φαίνεται στην εικόνα 4.21, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
readselB        <=      "11111"; --Επιλογή του καταχωρητή 31
wait for clk_in_period;
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο readselB την τιμή "11111", για να γίνει επιλογή του καταχωρητή 31. Όπως φαίνεται από την εικόνα 4.21, και εδώ το Register File δούλεψε σωστά, καθώς στην έξοδο B έβγαλε την τιμή που αποθηκεύτηκε προηγουμένως στον καταχωρητή 31, που είναι και το ζητούμενο.

4.3 Η προσομοίωση της Μονάδας Ελέγχου

Επειδή οι εντολές που εκτελεί ο επεξεργαστής που υλοποιήθηκε είναι 14 και συνεπώς η προσομοίωση της Μονάδας Ελέγχου είναι πολύ μεγάλη, στο κείμενο της παρούσας εργασίας περιέχεται ένα μέρος αυτής. Το μέρος αυτό της προσομοίωσης φαίνεται στις παρακάτω εικόνες:

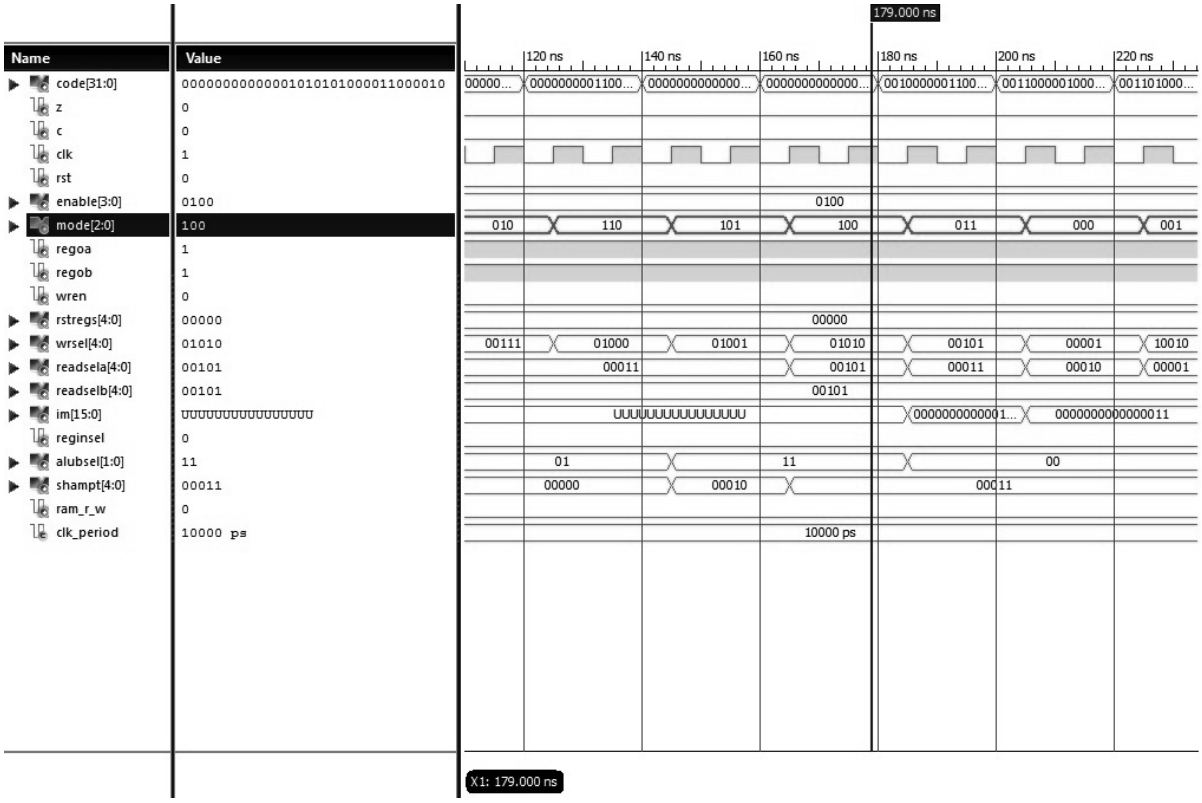


Εικόνα 4.22: Προσομοίωση του top module του Register File(α)

Για την προσομοίωση που φαίνεται στην εικόνα 4.22, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
Code <= "00000000011001010000100000100000"; --add $Reg1,$Reg3,$Reg5
wait for clk_period*2; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --2 περιόδους ρολογιού
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο Code η τιμή "00000000011001010000100000100000", ώστε να δώσει η μονάδα ελέγχου τα κατάλληλα σήματα για την εκτέλεση της πράξης Reg1 = Reg3 + Reg5. Όπως φαίνεται από την εικόνα 4.22, εδώ η μονάδα ελέγχου δούλεψε σωστά, καθώς έδωσε στη μνήμη RAM το σήμα για ανάγνωση, ενεργοποίησε τον καταχωρητή εισόδου κωδικοποίησης εντολής του επεξεργαστή, απενεργοποίησε τα σήματα Reset όλων των καταχωρητών, επέλεξε τους καταχωρητές 3 και 5 για ανάγνωση, τον καταχωρητή 1 για εγγραφή, ενεργοποίησε την είσοδο εγγραφής και τις εξόδους ανάγνωσης του Register File, έδωσε στην ALU το κατάλληλο mode για πρόσθεση και τέλος, έδωσε τα κατάλληλα σήματα επιλογής των πολυπλεκτών του επεξεργαστή ώστε οι είσοδος των καταχωρητών να παίρνει τιμή από την έξοδο της ALU και η είσοδος B της ALU να παίρνει τιμή από την έξοδο B του Register File.

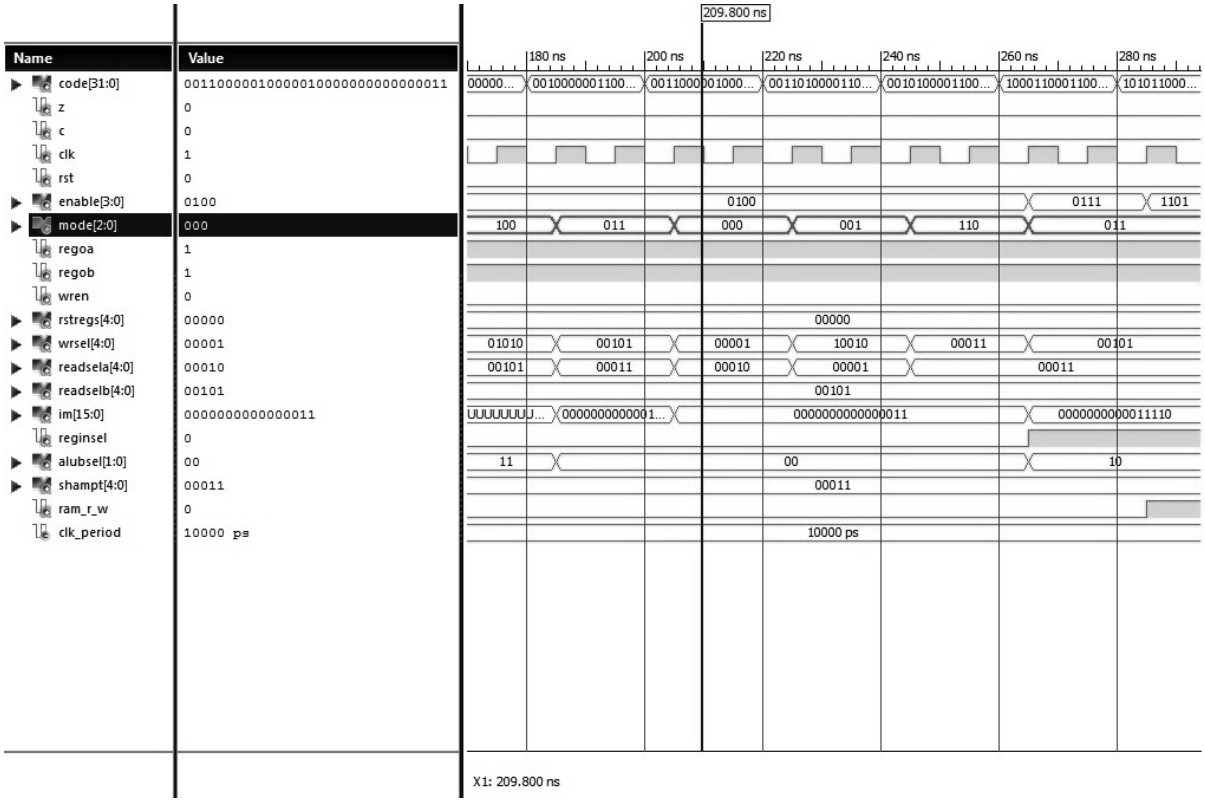


Εικόνα 4.23: Προσομοίωση του top module του Register File(β)

Για την προσομοίωση που φαίνεται στην εικόνα 4.23, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
Code <= "00000000000001010101000011000010"; --srl $Reg10,$Reg5,3
wait for clk_period*2; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --2 περιόδους ρολογιού
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο Code η τιμή "00000000000001010101000011000010", ώστε να δώσει η μονάδα ελέγχου τα κατάλληλα σήματα για την εκτέλεση δεξιάς ολίσθησης $3^{ων}$ θέσεων του καταχωρητή 5 και το αποτέλεσμα να αποθηκευτεί στον καταχωρητή 10. Όπως φαίνεται από την εικόνα 4.23, και εδώ η μονάδα ελέγχου δούλεψε σωστά, καθώς έδωσε στη μνήμη RAM το σήμα για ανάγνωση, ενεργοποίησε τον καταχωρητή εισόδου κωδικοποίησης εντολής του επεξεργαστή, απενεργοποίησε τα σήματα Reset όλων των καταχωρητών, επέλεξε τον καταχωρητή 5 για ανάγνωση και τον καταχωρητή 10 για εγγραφή, ενεργοποίησε την είσοδο εγγραφής και την έξοδο ανάγνωσης A του Register File, έδωσε στην ALU το κατάλληλο mode για δεξιά ολίσθηση, έβγαλε το κατάλληλο shamt και τέλος, έδωσε τα κατάλληλα σήματα επιλογής των πολυπλεκτών του επεξεργαστή ώστε οι είσοδος των καταχωρητών να παίρνει τιμή από την έξοδο της ALU και η είσοδος B της ALU να παίρνει το shamt.

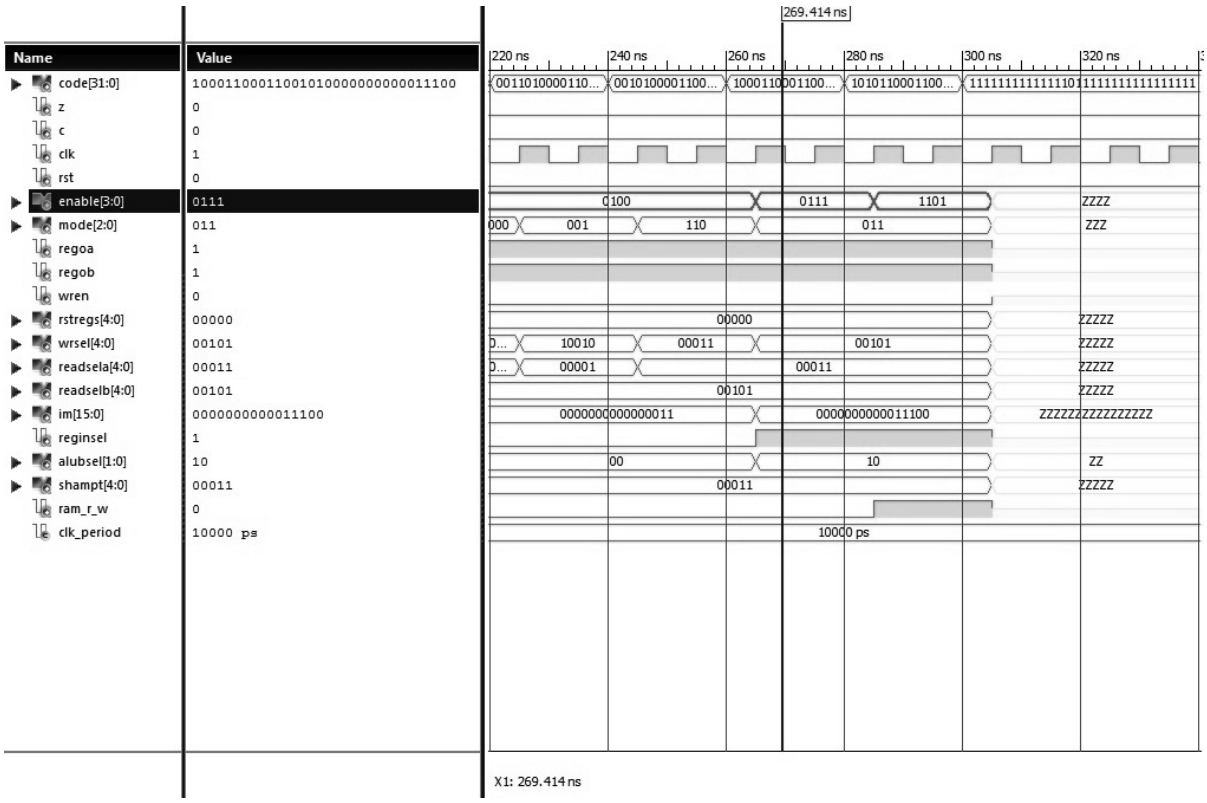


Εικόνα 4.24: Προσομοίωση του top module του Register File(γ)

Για την προσομοίωση που φαίνεται στην εικόνα 4.24, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
Code <= "00110000010000010000000000000011"; --andi $Reg1,$Reg2,3
wait for clk_period*2; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --2 περιόδους ρολογιού
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο Code η τιμή "00110000010000010000000000000011", ώστε να δώσει η μονάδα ελέγχου τα κατάλληλα σήματα για την εκτέλεση της άμεσης λογικής πράξης Reg1 = Reg2 and 3. Όπως φαίνεται από την εικόνα 4.24, και εδώ η μονάδα ελέγχου δούλεψε σωστά, καθώς έδωσε στη μνήμη RAM το σήμα για ανάγνωση, ενεργοποίησε τον καταχωρητή εισόδου κωδικοποίησης εντολής του επεξεργαστή, απενεργοποίησε τα σήματα Reset όλων των καταχωρητών, επέλεξε τον καταχωρητή 2 για ανάγνωση και τον καταχωρητή 1 για εγγραφή, ενεργοποίησε την είσοδο εγγραφής και την έξοδο ανάγνωσης A του Register File, έδωσε στην ALU το κατάλληλο mode για να εκτελέσει λογική πράξη and, έβγαλε την άμεση τιμή από την έξοδό της im και τέλος, έδωσε τα κατάλληλα σήματα επιλογής των πολυπλεκτών του επεξεργαστή ώστε οι είσοδος των καταχωρητών να παίρνει τιμή από την έξοδο της ALU και η είσοδος B της ALU να παίρνει την άμεση τιμή.



Εικόνα 4.25: Προσομοίωση του top module του Register File(δ)

Για την προσομοίωση που φαίνεται στην εικόνα 4.25, γράφτηκε ο παρακάτω κώδικας στο testbench αρχείο της προσομοίωσης:

```
Code <= "10001100011001010000000000011110"; --lw $Reg5,28($Reg3)
wait for clk_period*2; --Το κύκλωμα παραμένει στην ίδια κατάσταση για
 --2 περιόδους ρολογιού
```

Όπως φαίνεται από το κομμάτι αυτό του κώδικα του testbench αρχείου, δόθηκε στην είσοδο Code η τιμή "10001100011001010000000000011110", ώστε να δώσει η μονάδα ελέγχου τα κατάλληλα σήματα για τη φόρτωση στον καταχωρητή 5 τα περιεχόμενα της διεύθυνσης Reg3(7) της μνήμης RAM. Όπως φαίνεται από την εικόνα 4.25, και εδώ η μονάδα ελέγχου δούλεψε σωστά, καθώς έδωσε στη μνήμη RAM το σήμα για ανάγνωση, ενεργοποίησε τους καταχωρητές εισόδου κωδικοποίησης εντολής, εισόδου δεδομένων και εξόδου διεύθυνσης RAM του επεξεργαστή, απενεργοποίησε τα σήματα Reset όλων των καταχωρητών, επέλεξε τον καταχωρητή 3 για ανάγνωση και τον καταχωρητή 5 για εγγραφή, ενεργοποίησε την είσοδο εγγραφής και την έξοδο ανάγνωσης A του Register File, έδωσε στην ALU το κατάλληλο mode για να εκτελέσει πρόσθεση για τον υπολογισμό της διεύθυνσης μνήμης, έβγαλε την τιμή της διεύθυνσης από την έξοδό της im και τέλος, έδωσε τα κατάλληλα σήματα επιλογής των πολυπλεκτών του επεξεργαστή ώστε οι είσοδος των καταχωρητών να παίρνει τιμή από τον καταχωρητή εισόδου δεδομένων του επεξεργαστή και η είσοδος B της ALU να παίρνει την τιμή της διεύθυνσης που βγάζει η μονάδα ελέγχου τετραπλασιασμένη.

Επειδή ο επεξεργαστής που υλοποιήσαμε στην παρούσα εργασία δεν είναι πλήρης, δεν καταφέραμε να προσομοιώσουμε το top module του. Έτσι το κεφάλαιο αυτό της παρούσας εργασίας τελειώνει σε αυτό το σημείο.

5. Συμπεράσματα

Αυτό το κεφάλαιο, αναφέρεται στα συμπεράσματα που βγήκαν μετά την ολοκλήρωση της παρούσας εργασίας. Επίσης, αυτό το κεφάλαιο αναφέρεται στο πως μπορεί αυτή η σχεδίαση να βελτιστοποιηθεί.

Σκοπός της παρούσας εργασίας ήταν η εις βάθος κατανόηση της γλώσσας περιγραφής υλικού VHDL και της σχεδίασης ενός μικροεπεξεργαστή. Έτσι, σχεδιάστηκαν με λεπτομέρεια διάφορα τμήματα ενός μικροεπεξεργαστή με χρήση της VHDL. Αρχικά, έγιναν τα ψηφιακά σχέδια του κάθε κυκλώματος του μικροεπεξεργαστή, τα οποία περιέχονται στο δεύτερο κεφάλαιο της εργασίας. Στη συνέχεια έγινε η καταγραφή του κώδικα, ο οποίος περιέχεται στο τρίτο κεφάλαιο. Μετά την καταγραφή κάθε μέρους του κώδικα, για να βεβαιωθούμε ότι δουλεύει σωστά, γινόταν η προσομοίωσή του. Οι προσομοιώσεις περιέχονται στο τέταρτο κεφάλαιο της παρούσας εργασίας.

Ένα από τα συμπεράσματα που βγήκαν από την εργασία αυτή, είναι ότι με τη χρήση της γλώσσας VHDL, μπορούμε να σχεδιάσουμε πολύ εύκολα και γρήγορα ένα ψηφιακό κύκλωμα, γράφοντας μερικές γραμμές κώδικα. Εκεί που με χρήση ψηφιακού σχεδίου θα θέλαμε μέρες ολόκληρες για μια πολύπλοκη σχεδίαση, με τη χρήση της γλώσσας αυτής, μέσα σε μερικές ώρες έχουμε πολύ εύκολα το σχέδιο έτοιμο. Επίσης, μπορούμε να προσομοιώσουμε την λειτουργία του κυκλώματος που περιγράψαμε, για να δώσουμε στις εισόδους του κυκλώματος μερικές τιμές, έτσι ώστε να δούμε τις τιμές που δίνουν οι έξοδοί του και να επιβεβαιωθεί η σωστή λειτουργία του.

Ένα δεύτερο συμπέρασμα που βγήκε από την εκπόνηση της παρούσας εργασίας, είναι ότι με όσο περισσότερη λεπτομέρεια σχεδιάζει κανείς ένα ψηφιακό κύκλωμα, τόσο περισσότερο κατανοεί τη λειτουργία του. Επίσης, όσο περισσότερο χρόνο καταναλώνει κανείς για μια ψηφιακή σχεδίαση, τόσες περισσότερες λεπτομέρειες κατανοεί πάνω στη σχεδίαση αυτή. Για παράδειγμα, όταν ασχολείται κανείς με την υλοποίηση ενός μικροεπεξεργαστή, αρχικά σκέφτεται ότι πρέπει να φτιάξει μια Αριθμητική και Λογική Μονάδα. Στη συνέχεια, πρέπει να αποφασίσει τι πράξεις θα εκτελεί αυτή η μονάδα. Οπότε, όταν αποφασίζεται ότι θα εκτελεί εκτός των άλλων τις πράξεις της πρόσθεσης και της αφαίρεσης, σκέφτεται ότι πρέπει να υλοποιηθεί κι ένας αθροιστής - αφαιρέτης. Για να γίνει αυτό, πρέπει πρώτα να υλοποιηθεί ένας πλήρης αθροιστής. Έτσι κατανοείται σε όλο και μεγαλύτερο βάθος το κύκλωμα του επεξεργαστή και η λειτουργία του.

Ένα ακόμα συμπέρασμα που βγήκε από την παρούσα εργασία είναι ότι κατά τη σχεδίαση ενός μικροεπεξεργαστή ή γενικότερα οποιουδήποτε κυκλώματος που λειτουργεί στα 32 bits, είναι πολύ χρήσιμο πάνω στα σχέδια να αναγράφονται ονομασίες των διάφορων εισόδων, εξόδων και γενικότερα των καλωδίων που συνδέουν τα διάφορα σημεία του κυκλώματος. Επειδή οι είσοδοι, οι έξοδοι και οι συνδέσεις είναι αρκετές όταν η σχεδίαση είναι στα 32 bits, τα λάθη γίνονται πολύ εύκολα. Έτσι, όταν υπάρχουν οι ονομασίες τους πάνω στα διάφορα σχέδια των κυκλωμάτων, μπορούν να αποφευχθούν τα λάθη αυτά, ή τουλάχιστον θα είναι πολύ πιο εύκολο να διορθωθούν.

Όσον αφορά τη ολοκλήρωση της υλοποίησης, θα μπορούσε στη σχεδίαση του μικροεπεξεργαστή να προστεθεί ακόμα ένας καταχωρητής, ο οποίος θα λειτουργεί ως

Program Counter. Η δουλειά του καταχωρητή αυτού, θα είναι να καταμετρά τις εντολές που εκτελεί ο μικροεπεξεργαστής, έτσι ώστε όταν εκτελείται μια εντολή άλματος (η υλοποίηση της παρούσας εργασίας δε συμπεριλαμβάνει τέτοια εντολή), να δείχνει σε ποιο σημείο του κώδικα να μεταπηδήσει ο επεξεργαστής. Επίσης, για να μπορέσει να λαμβάνει εντολές ο επεξεργαστής, είναι απαραίτητο να σχεδιαστεί και μια μνήμη RAM από την οποία θα δέχεται τις εντολές ο μικροεπεξεργαστής και από ή προς την οποία θα δέχεται ή θα στέλνει τα δεδομένα όταν εκτελεί εντολές μεταφοράς δεδομένων.

Μια ακόμα βελτιστοποίηση που θα μπορούσε να γίνει στη σχεδίαση του μικροεπεξεργαστή, ήταν να υπάρχει υποστήριξη για περισσότερες εντολές από το σύνολο των εντολών του MIPS-32. Όσον αφορά, για παράδειγμα τις εντολές αριθμητικών πράξεων, θα μπορούσε να προστεθεί η υποστήριξη των πράξεων με πραγματικούς αριθμούς σε κινητή υποδιαστολή σύμφωνα με το πρότυπο IEEE 754. Για να γίνει αυτό, η υποστήριξη δηλαδή των πράξεων με πραγματικούς αριθμούς, θα πρέπει να υλοποιηθεί μια δεύτερη αριθμητική και λογική μονάδα (συνεπεξεργαστής). Οπότε σ’ αυτήν την περίπτωση ανάλογα με το αν η εντολή αναφέρεται σε πράξη με ακέραιους ή πραγματικούς, η Μονάδα Ελέγχου θα ενεργοποιήσει την κατάλληλη ALU για την εκτέλεση της πράξης. Θα μπορούσαν επίσης να προστεθούν και άλλα κυκλώματα, για την εκτέλεση π.χ. της πράξης του πολλαπλασιασμού ή ακόμα και της διαίρεσης, με ακέραιους ή ακόμα και με πραγματικούς αριθμούς. Τέλος, θα μπορούσαν εκτός από τις εντολές αριθμητικών πράξεων, να προστεθούν και επιπλέον εντολές μεταφοράς δεδομένων, όπως η αποθήκευση μισής λέξης (sh) και η αποθήκευση ενός byte (sb).

Για να κλείσουμε το κεφάλαιο αυτό, μια τελευταία βελτιστοποίηση της σχεδίασης του μικροεπεξεργαστή, θα μπορούσε να είναι η υποστήριξη πράξεων με πραγματικούς αριθμούς διπλής ακρίβειας (64 bits).

Βιβλιογραφία

Βιβλία:

- «ΟΡΓΑΝΩΣΗ ΚΑΙ ΣΧΕΔΙΑΣΗ ΥΠΟΛΟΓΙΣΤΩΝ» (ΤΟΜΟΣ Α’), 3η έκδοση, DAVID A. PATTERSON – JOHN L. HENNESSY
- «ΨΗΦΙΑΚΗ ΣΧΕΔΙΑΣΗ», 2η έκδοση, M. MORRIS MANO

Ιστότοποι:

- http://esd.cs.ucr.edu/labs/tutorial/
- http://el.wikipedia.org/wiki/Αριθμητική_και_Λογική_Μονάδα
- http://www.d.umn.edu/~gshute/spimsal/talref.html
- http://coe.uncc.edu/~amukherj/INTRO2VHDL/alu.eg1.pdf
- http://users.eecs.northwestern.edu/~debjit/files/361_Project.pdf
- http://en.wikipedia.org/wiki/MIPS_architecture