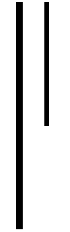


Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Προηγμένα Συστήματα Πληροφορικής»

**Μεταπτυχιακή Διατριβή**

| Τίτλος Διατριβής | **Υπολογισμός AVF για επεξεργαστή MIPS** |
|---|---|
| Ονοματεπώνυμο Φοιτητή | **Εμμανουήλ Ευάγγελος** |
| Πατρώνυμο | **Ιωάννης** |
| Αριθμός Μητρώου | **ΜΠΣΠ/08064** |
| Επιβλέπων | **Δ. Γκιζόπουλος, Αναπληρωτής Καθηγητής** |

Ημερομηνία Παράδοσης **Ιούνιος 2011**

**Τριμελής Εξεταστική Επιτροπή**

| (υπογραφή) | (υπογραφή) | (υπογραφή) |
|---|---|---|
| Δ. Γκιζόπουλος<br>Αναπλ. Καθ. | Μ. Ψαράκης<br>Επικ. Καθ. | Α. Πικράκης<br>Λέκτορας |

## Περίληψη

Όσο η τεχνολογία κατασκευής ολοκληρωμένων κυκλωμάτων και επεξεργαστών βελτιώνεται, τόσο πιο ευάλωτα γίνονται τα κυκλώματα αυτά σε κρούσεις σωματιδίων άλφα και νετρονίων. Το κόστος προστασίας ενός επεξεργαστή από αυτά τα φαινόμενα είναι σημαντικό. Ο συντελεστής αρχιτεκτονικής ευπάθειας, Architectural Vulnerability Factor (AVF) είναι η πιθανότητα ένα σφάλμα τέτοιου τύπου να προκαλέσει λάθος στην εκτέλεση κάποιου προγράμματος. Ο AVF λοιπόν μπορεί να χρησιμοποιηθεί σαν εργαλείο κατά τη σχεδίαση ώστε να παρθούν σωστές αποφάσεις για το ποιες δομές πρέπει να προστατευθούν και με ποιο τρόπο, ώστε να επιτευχθεί το βέλτιστο κόστος προστασίας.

Σε αυτή την εργασία, αναπτύχθηκε ένας προσομοιωτής και ένα μοντέλο ενός επεξεργαστή βασισμένο στην βασική αρχιτεκτονική MIPS32 προκειμένου να υπολογιστεί ο AVF του. Παρουσιάζονται πειράματα που εκτελέστηκαν και έδωσαν AVF από 33,38% μέχρι 36,45%, αριθμοί που επιβεβαιώνουν το γεγονός πως δεν είναι ανάγκη να προστατευθούν όλα τα bit κάθε δομής του επεξεργαστή. Επίσης παρατηρήθηκαν και σημαντικές αποκλείσεις στην κάθε δομή. Συγκεκριμένα η δομή IF/ID είχε τιμές από 47,64% μέχρι 57,80%, η ID/EX από 38,99% μέχρι 45,33%, η EX/MEM από 16,36% μέχρι 26,95% και η MEM/WB από 22,07% μέχρι 31,07%. Ένα ακραίο παράδειγμα είναι τα opcode bit του καταχωρητή PC τα οποία έχουν AVF 100% μιας και ένα οποιοδήποτε λάθος σε αυτά τα bit θα οδηγήσει στην εκτέλεση μίας άλλης εντολής από την επιθυμητή.

## Abstract

As IC and microprocessor manufacturing technologies improve, the more vulnerable these devices become to alpha particles and neutron strikes. The cost of protecting the internal structures of a CPU from such events is significant. AVF is the fraction of such faults resulting in user-visible errors. Thus, AVF can be used as a design tool enabling informed decisions to be made regarding which structures should be protected and in what way in order to minimize radiation hardening costs.

This dissertation shows the development of a simulator and a model of a core MIPS32 architecture based microprocessor with the goal of computing it's AVF. Various experiments where executed which gave AVF values ranging from 33.38% to 36.45%. These results suggest that it is indeed not necessary to protect every single bit inside a microprocessor. Significant variations where observed among the AVF values of individual hardware structures. Specifically, the IF/ID register had AVF values from 47.64% to 57.80%, ID/EX AVF ranged from 38.99% to 45.33%, EX/MEM was from 16.36% to 26.95% and MEM/WB had AVF values from 22.07% to 31.07%. Another example to this are the opcode bits of the PC register with an AVF of 100% since any error in those bits will lead to the execution of an incorrect operation.

# Περιεχόμενα


1 Εισαγωγή ..... 5
1.1 ΑΝΤΙΚΕΙΜΕΝΟ ..... 5
1.2 AVF ..... 5
1.3 ΚΙΝΔΥΝΟΙ ΕΝΟΣ ΣΦΑΛΜΑΤΟΣ ..... 5
1.4 ΔΙΑΡΘΡΩΣΗ ΚΕΙΜΕΝΟΥ ..... 6
2 Σχετική εργασία ..... 7
2.1 ΕΙΣΑΓΩΓΗ ..... 7
2.2 ΠΡΟΗΓΟΥΜΕΝΕΣ ΕΡΓΑΣΙΕΣ ..... 7
2.3 ΥΠΑΡΧΟΝΤΑ ΠΑΚΕΤΑ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ AVF ..... 10
3 Ορισμός του προβλήματος και μεθοδολογία ..... 11
3.1 ΕΙΣΑΓΩΓΗ ..... 11
3.2 ΠΕΔΙΟ ΕΦΑΡΜΟΓΗΣ ..... 11
3.2.1 Αρχιτεκτονική MIPS 32 ..... 12
3.2.2 Υποστηριζόμενο σύνολο εντολών ..... 12
3.2.3 Μοντέλο διοχέτευσης ..... 15
3.2.4 Μοντέλο Κρυφής μνήμης ..... 16
3.3 ΜΕΘΟΔΟΛΟΓΙΑ ..... 18
3.4 ΜΟΝΤΕΛΟ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ MIPS32 ..... 18
3.5 ΒΑΣΙΚΟΙ ΤΥΠΟΙ ΕΝΤΟΛΩΝ ..... 18
3.5.1 Τύπος R ..... 19
3.5.2 Τύπος I ..... 21
3.5.3 Τύπος J ..... 23
3.6 ΔΙΟΧΕΤΕΥΣΗ ..... 23
3.7 ΣΤΑΔΙΑ ΤΗΣ ΔΙΟΧΕΤΕΥΣΗΣ ..... 25
3.7.1 Στάδιο IF ..... 25
3.7.2 Στάδιο ID ..... 26
3.7.3 Στάδιο EX ..... 26
3.7.4 Στάδιο MEM ..... 27
3.7.5 Στάδιο WB ..... 27
3.8 ΚΑΤΑΧΩΡΗΤΕΣ ΤΗΣ ΔΙΟΧΕΤΕΥΣΗΣ ..... 28
3.8.1 IF/ID (64 bits) ..... 28
3.8.2 ID/EX (157 bits) ..... 28
3.8.3 EX/MEM (75 bits) ..... 28
3.8.4 MEM/WB (71 bits) ..... 29
3.9 ΚΡΥΦΗ ΜΝΗΜΗ ..... 29
3.10 ΜΝΗΜΗ ΣΥΣΤΗΜΑΤΟΣ ..... 29
3.11 ΚΩΔΙΚΟΙ ΕΝΤΟΛΩΝ ..... 29
3.12 BIT ΕΛΕΓΧΟΥ ΤΗΣ ΔΙΟΧΕΤΕΥΣΗΣ ..... 30
3.12.1 IF ..... 31
3.12.2 ID ..... 31
3.12.3 EX ..... 31
3.12.4 MEM ..... 33
3.12.5 WB ..... 35
3.13 ΣΥΝΟΨΗ BIT ΕΛΕΓΧΟΥ ..... 36
3.14 ACE BITS ..... 37
3.14.1 ACE bits Ελέγχου ..... 37
3.14.2 IF/ID ACE bits ..... 39
3.14.3 ID/EX ACE bits ..... 44
3.14.4 EX/MEM ACE bits ..... 51
3.14.5 MEM/WB ACE bits ..... 57

3.14.6 ACE bits της διοχέτευσης ........ 6263
3.15 ΕΠΙΒΕΒΑΙΩΣΗ ΤΟΥ ΜΟΝΤΕΛΟΥ ........ 6364

**4 Ανάπτυξη Λογισμικού ........ 65**
4.1 ΕΙΣΑΓΩΓΗ ........ 65
4.2 ΜΕΘΟΔΟΛΟΓΙΑ ΑΝΑΠΤΥΞΗΣ ........ 65
4.2.1 Μέθοδος Staged Delivery ........ 66
4.2.2 Προσέγγιση Top-Down ........ 66
4.2.3 Προσέγγιση Bottom-UP ........ 67
4.2.4 UML 2.0 ........ 67
4.2.5 Σχεδιαστικά μοτίβα (Design Patterns) ........ 67
4.3 UML ΔΙΑΓΡΑΜΜΑΤΑ ΛΟΓΙΣΜΙΚΟΥ ........ 70
4.3.1 UML μοντέλου της διοχέτευσης ........ 70
4.3.2 UML διάγραμμα μοντέλου επεξεργαστή ........ 71

**5 Πειράματα ........ 73**
5.1 ΕΙΣΑΓΩΓΗ ........ 73
5.2 ΜΟΡΦΗ ΑΡΧΕΙΟΥ ΠΕΙΡΑΜΑΤΟΣ ........ 73
5.3 ΠΕΙΡΑΜΑΤΑ ........ 74
5.3.1 Αριθμοί Fibonacci ........ 74
5.3.2 Αλγόριθμός Selection Sort [15] ........ 78
5.3.3 Πολλαπλασιασμός ακεραίων με συνεχείς προσθέσεις ........ 87
5.3.4 Υπολογισμός n! ........ 88
5.4 ΣΥΓΚΕΝΤΡΩΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ ........ 90

**6 Συμπεράσματα ........ 91**
6.1 ΔΥΣΚΟΛΙΕΣ ........ 91
6.2 ΣΧΟΛΙΑ ΕΠΙ ΤΩΝ ΑΠΟΤΕΛΕΣΜΑΤΩΝ ........ 91
6.3 ΜΕΛΛΟΝΤΙΚΗ ΕΡΓΑΣΙΑ ........ 92
6.4 ΕΠΙΛΟΓΟΣ ........ 92

**7 Βιβλιογραφία ........ 93**

**8 Ευρετήρια ........ 95**

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

# ~~1~~ 1 Εισαγωγή

## ~~1.1~~ 1.1 ΑΝΤΙΚΕΙΜΕΝΟ

Στην συγκεκριμένη εργασία, επικεντρωνόμαστε στον υπολογισμό του AVF ενός μοντέλου επεξεργαστή βασισμένου στην αρχιτεκτονική MIPS. Αναπτύξαμε έναν προσομοιωτή και ένα μοντέλο της αρχιτεκτονικής MIPS 32 βασιζόμενοι στην τρέχουσα βιβλιογραφία σχετικά με τον AVF, την αρχιτεκτονική MIPS καθώς και διάφορες προγραμματιστικές πρακτικές αντικειμενοστρεφούς προγραμματισμού.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## ~~1.2~~ 1.2 AVF

Ο συντελεστής αρχιτεκτονικής ευπάθειας (AVF) είναι ο λόγος των σφαλμάτων που θα εμφανιστούν στον χρήστη προς τα συνολικά σφάλματα που παρουσιάστηκαν εντός ενός ολοκληρωμένου κυκλώματος κατά τη λειτουργία του.

Για διάφορους λόγους που παρουσιάζονται στο Κεφάλαιο 2 όπως η κοσμική ακτινοβολία και ακτινοβολία από τη συσκευασία ενός ολοκληρωμένου κυκλώματος, ενδέχεται η τιμή κάποιων στοιχείων μνήμης να αλλοιωθεί κατά τη λειτουργία του. Ο ρυθμός που συμβαίνουν αυτές οι αλλοιώσεις συνδέεται με το μέγεθος του κάθε τρανζίστορ, το ηλεκτρικό δυναμικό με το οποίο λειτουργεί η συσκευή, την τεχνολογία κατασκευής της συσκευής καθώς και το περιβάλλον μέσα στο οποίο λειτουργεί το κύκλωμα. Όπως παρουσιάζουμε στα επόμενα κεφάλαια, υπάρχουν τρόποι ανίχνευσης και αντιμετώπισης αυτών των διαταραχών αλλά κάθε τρόπος έχει διαφορετική αποτελεσματικότητα και κόστος. Ο AVF λοιπόν είναι ένα χρήσιμο εργαλείο το οποίο μπορεί να μας δείξει πόσο ευάλωτη είναι μία δομή ώστε να μπορούν να ληφθούν αποφάσεις κατά την σχεδίαση ενός ολοκληρωμένου κυκλώματος σχετικά με ποιες περιοχές πρέπει να προστατευθούν και ποιες όχι. Πρέπει να σημειωθεί πως ο AVF εξαρτάται άμεσα από τις εφαρμογές που θα εκτελεί το ολοκληρωμένο επειδή κάθε εφαρμογή χρησιμοποιεί διαφορετικά μέρη του κυκλώματος. Για παράδειγμα στην περίπτωση ενός επεξεργαστή, μας ενδιαφέρει να ξέρουμε τι τύπου εφαρμογές θα εκτελούνται και υπολογίζουμε τον AVF τρέχοντας παρόμοιες δοκιμαστικές εφαρμογές στο μοντέλο μας.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## ~~1.3~~ 1.3 ΚΙΝΔΥΝΟΙ ΕΝΟΣ ΣΦΑΛΜΑΤΟΣ

Οι κίνδυνοι που ενέχει ένα σφάλμα σε κάποιο bit ενός επεξεργαστή εξαρτώνται από την εφαρμογή. Για παράδειγμα ένα σφάλμα σε έναν επεξεργαστή γραφικών ενός υπολογιστή γραφείου μπορεί να μην έχει καμία επίδραση, μπορεί να αλλοιώσει το χρώμα κάποιου pixel στην οθόνη η μπορεί να προκαλέσει τέτοιο σφάλμα ικανό να τερματίσει την λειτουργία ολόκληρου του υπολογιστικού συστήματος. Ανάλογα με την λειτουργία που εκτελεί αυτό το σύστημα, μπορεί είτε να μην παρατηρήσουμε καν το ότι συνέβη

κάποιο σφάλμα είτε να βρεθούν σε κίνδυνο ακόμα και ανθρώπινες ζωές αν για παράδειγμα αυτός ο υπολογιστής ελέγχει κάποια κρίσιμη λειτουργία μίας κατασκευής ή εγκατάστασης.

Ιδιαίτερο ενδιαφέρον έχουν μικροεπεξεργαστές και μικροελεγκτές που αποτελούν μέρη κρίσιμων συστημάτων όπως αεροπλάνων, πυρηνικών αντιδραστήρων, δορυφόρων, οπλικών συστημάτων κλπ. Μέχρι αυτή τη στιγμή, δεν έχει καταγραφεί κάποιο καταστροφικό σφάλμα για το οποίο είναι υπεύθυνα τέτοιου είδους σφάλματα αλλά έχουν ήδη υπάρξει σημαντικές οικονομικές ζημιές, με το πρώτο να αναφέρεται από την Intel το 1978, η IBM εντόπισε παρόμοια προβλήματα το 1986, η Normand το 1996, η Sun Microsystems το 2000, η Cypress semiconductor το 2004 και η Hewlett-Packard το 2005 [13].

## 1.4 1.4 ΔΙΑΡΘΡΩΣΗ ΚΕΙΜΕΝΟΥ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Το παρόν κείμενο οργανώνεται σε 8 κεφάλαια. Ξεκινάμε από την παρουσίαση της τρέχουσας κατάστασης γύρω από τον AVF, παρουσιάζουμε την μεθοδολογία που ακολουθήθηκε στο παρόν έργο και τέλος παραθέτουμε κάποια πειράματα και συμπεράσματα.

Στο 2° κεφάλαιο παρουσιάζουμε κάποιες από τις σημαντικότερες εργασίες σχετικά με τον υπολογισμό του AVF. Γίνεται αναφορά στις κυριότερες μεθόδους υπολογισμού του AVF και γίνεται αναλυτικότερη παρουσίασή του.

Στο 3° κεφάλαιο παρουσιάζουμε αναλυτικότερα το πρόβλημα το οποίο επιχειρούμε να λύσουμε. Περιγράφουμε την μέθοδο που ακολουθήσαμε με λεπτομέρεια και παρουσιάζουμε το μοντέλο του επεξεργαστή που μελετήθηκε. Επίσης παρουσιάζουμε την ανάλυση που είναι απαραίτητη για τον υπολογισμό του AVF.

Στο 4° κεφάλαιο παρουσιάζουμε τη μεθοδολογία που ακολουθήθηκε κατά τον σχεδιασμό και ανάπτυξη του λογισμικού. Επίσης, παραθέτονται κάποια διαγράμματα με σκοπό να βοηθηθεί ο αναγνώστης στην κατανόηση του κώδικα.

Στο 5° κεφάλαιο παρουσιάζεται ο κώδικας από τα πειράματα που εκτελέσαμε καθώς και τα αποτελέσματά τους.

Στο 6° κεφάλαιο παρατίθενται τα συμπεράσματα της εργασίας καθώς και σκέψεις για πιθανή μετέπειτα έρευνα πάνω στο θέμα.

Το 7° κεφάλαιο περιέχει την βιβλιογραφία και το 8° κεφάλαιο τα ευρετήρια πινάκων και εικόνων.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

# 2 Σχετική εργασία

## 2.1 ΕΙΣΑΓΩΓΗ

Σε αυτό το κεφάλαιο κάνουμε μία επισκόπηση της έρευνας που έχει γίνει στον τομέα της ανεκτικότητας σε σφάλματα.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## 2.2 ΠΡΟΗΓΟΥΜΕΝΕΣ ΕΡΓΑΣΙΕΣ

Τα σημαντικότερα βιβλία στον τομέα της ανεκτικότητας σε σφάλματα έχουν γίνει από τους Israel Koren, C. Mani Krishna και Shubu Mukherjee.

Οι εργασίες αυτές συνοψίζονται στα βιβλία τους:

- Architecture Design for Soft Errors [12]
- Fault Tolerant Systems [14]

Και στα δύο βιβλία παρουσιάζονται διάφορες τεχνικές ανίχνευσης και ανάκαμψης ενός συστήματος από διαφόρων ειδών σφάλματα. Επίσης παρουσιάζονται διάφοροι τρόποι αξιολόγησης της αξιοπιστίας (reliability) και της διαθεσιμότητας (availability) ενός συστήματος.

Σε αυτή την εργασία θα εστιάσουμε σε δύο συγκεκριμένες δημοσιευμένες εργασίες των Shubu Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt και Todd Austin. Μας ενδιαφέρουν δύο εργασίες σχετικές με τον υπολογισμό του AVF.

- Measuring Architectural Vulnerability Factors. [10]
- A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. [11]

***Measuring Architectural Vulnerability Factors [10]***

Η συγκεκριμένη εργασία ξεκινάει με το να εστιάσει την προσοχή του αναγνώστη στο γεγονός πως τα λεγόμενα μαλακά σφάλματα (soft errors) θα αποτελούν ένα όλο και σημαντικότερο πρόβλημα στις ερχόμενες γενιές ολοκληρωμένων κυκλωμάτων. Αναφέρει πως υπάρχουν τρόποι αντιμετώπισης αλλά έχουν κάποιο κόστος σε έναν ή και περισσότερους από τους παρακάτω τομείς: απόδοση, καταναλισκόμενη ισχύς, μέγεθος του τσιπ και σχεδιαστικός χρόνος. Συνεχίζει λέγοντας πως πρέπει να βρεθεί ένας τρόπος προκειμένου να μπορεί κάποιος να εκτιμήσει στα αρχικά στάδια του σχεδιασμού ενός νέου ολοκληρωμένου την πιθανότητα να εμφανιστούν μαλακά σφάλματα ώστε να μπορέσει να κρίνει αν και ποιες μεθόδους αντιμετώπισής τους θα χρησιμοποιήσει όπως διάφορους αλγόριθμους κωδικοποίησης δεδομένων ή ακόμα και μεθόδους σε επίπεδο τρανζίστορ, αναλόγως του κόστους κάθε μεθόδου και της

**Μορφοποιήθηκε:** Ελληνικά

**Μορφοποιήθηκε:** Ελληνικά

αναμενόμενης αποτελεσματικότητάς της στην εκάστοτε δομή. Στη συγκεκριμένη εργασία, παρουσιάζεται ακριβώς μία τέτοια μέθοδος.

Παράμετροι που επηρεάζουν την εμφάνιση σφαλμάτων σε ένα κύκλωμα περιλαμβάνουν το αποθηκευμένο φορτίο, το εμβαδόν του επιρρεπούς σε σφάλματα μέρους του κυκλώματος καθώς και η δυνατότητα συλλογής φορτίου αυτής της περιοχής από προσκρούσεις σωματιδίων και κοσμικής ακτινοβολίας.

Αναφέρεται πως καθώς μικραίνουν τα μεγέθη των συστατικών μειώνεται το φορτίο ανά συσκευή, οπότε και αυξάνεται η πιθανότητα να προκληθεί σφάλμα. Επειδή όμως μειώνεται και το μέγεθος των συστατικών μειώνεται και η πιθανότητα σύγκρουσης με κάποια σωματίδια. Η συνολική επίδραση απ' τα δύο παραπάνω γεγονότα είναι σχεδόν μηδενική όσον αφορά το κάθε bit ενός τσιπ ξεχωριστά. Με βάση το νόμο του Moore ο αριθμός των τρανζίστορ αναμένεται να αυξάνεται εκθετικά για αρκετά ακόμα χρόνια κατ' επέκταση εκθετικά θα αυξηθεί κι ο αριθμός των σφαλμάτων σ' ένα ολοκληρωμένο κύκλωμα. Στη συνέχεια αναφέρονται μερικά από τα παραδείγματα της επίδρασης των σφαλμάτων αυτών στη βιομηχανία ημιαγωγών.

Μορφοποιήθηκε: Ελληνικά

Τα σφάλματα που μπορεί να συμβούν, τα χωρίζουν σε δύο κατηγορίες. SDC (Silent Data Corruption) και DUE (Detected Unrecoverable Error). Η κατηγορία SDC αναφέρεται σε σφάλματα κατά τα οποία αλλοιώνεται κάποιο bit χωρίς να γίνεται αντιληπτό και αυτή η αλλοίωση οδηγεί σε εσφαλμένη εκτέλεση του προγράμματος. Αυτό είναι και το αντικείμενο της συγκεκριμένης εργασίας. Τα σφάλματα DUE συμβαίνουν όταν κάποιο σφάλμα ανιχνεύεται όπως για παράδειγμα με έλεγχο ισοτιμίας αλλά δεν μπορεί να διορθωθεί. Η περίπτωση όπου κάποιο σφάλμα ανιχνεύεται και διορθώνεται δεν μας απασχολεί.

Αναφέρεται πως ο συνολικός ρυθμός σφαλμάτων δεν αντιστοιχεί στο ρυθμό SDC σφαλμάτων. Αυτό συμβαίνει διότι σφάλματα σε κάποιες δομές, όπως στις δομές πρόβλεψης άλματος, δεν επηρεάζουν το τελικό αποτέλεσμα παρά μόνο την απόδοση του επεξεργαστή. Επίσης μπορεί να συμβαίνουν σφάλματα σε κάποια bit τα οποία σε εκείνη τη φάση της λειτουργίας του ολοκληρωμένου κυκλώματος δεν χρησιμοποιούνται. Εισάγεται λοιπόν η ιδέα του παράγοντα AVF (Architectural Vulnerability Factor) ο οποίος είναι ουσιαστικά η πιθανότητα ένα σφάλμα σε μία δομή σε ένα ολοκληρωμένο κύκλωμα να προκαλέσει κάποιο λάθος SDC.

Τέλος, αναφέρεται στη μέθοδο την οποία χρησιμοποίησαν προκειμένου να υπολογίσουνε τον AVF μίας αρχιτεκτονικής βασισμένη στον Itanium-2. Ονόμασαν τη μέθοδο ανάλυση ACE (Architectural Correct Execution). Όπως είπαμε, δεν οδηγεί κάθε σφάλμα σε λάθος στην εκτέλεση. Με την ανάλυση ACE, προσδιορίζονται τα bit στα οποία αν συμβεί κάποιο σφάλμα θα επηρεάσει τη σωστή εκτέλεση του προγράμματος. Ο AVF ενός συγκεκριμένου στοιχείου μνήμης είναι το κλάσμα του χρόνου κατά τον οποίο το συγκεκριμένο στοιχείο κρατάει κάποιο ACE bit. Υποθέτοντας πως όλα τα στοιχεία μνήμης εμφανίζουν ίδιο ρυθμό σφαλμάτων, ο τελικός παράγοντας AVF της δομής είναι ο μέσος όρος των AVF των στοιχείων μνήμης ή ο μέσος όρος των στοιχείων που κρατούν ACE bits κάθε στιγμή κατά τη διάρκεια της λειτουργίας του ολοκληρωμένου κυκλώματος.

Τον AVF μπορεί να τον υπολογίσει κάποιος είτε βασιζόμενος στον νόμο του Little [17] με βάση τον παρακάτω τύπο:

$$AVF = \frac{B_{ACE}\, x\, L_{ACE}}{\Sigma \acute{\upsilon} \nu o \lambda o\, \tau \omega \nu\, bit\, \sigma \tau \eta\, \delta o \mu \acute{\eta}}$$

Όπου: $B_{ACE}$ Ο μέσος ρυθμός εισόδου ACE bits στη διοχέτευση.

$L_{ACE}$ Ο μέσος χρόνος παραμονής των ACE bits στη διοχέτευση.

Εναλλακτικά, μπορεί κάποιος να παρακολουθήσει απευθείας την εκτέλεση ενός προγράμματος με χρήση ενός μοντέλου του ολοκληρωμένου κυκλώματος που μελετά και να καταγράψει ποια bit ήταν ACE bits κατά την εκτέλεση της προσομοίωσης. Στη συνέχεια, διαιρεί αυτό τον αριθμό με το σύνολο των bit στη δομή που μελετά και το αποτέλεσμα είναι ο AVF της δομής. Θεωρούμε πάντα πως κάθε bit είναι ACE εκτός και αν μπορούμε να αποδείξουμε το αντίθετο.

***A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor [11]***

Η συγκεκριμένη εργασία εστιάζει στην ανάλυση ACE με σκοπό τον υπολογισμό του AVF μίας δομής. Αναφέρεται στον μηχανισμό με τον οποίο δημιουργούνται τα λεγόμενα “soft errors”. Αυτά είναι

σφάλματα τα οποία δεν οφείλονται σε κάποιο μόνιμο πρόβλημα της συσκευής αλλά σε αλλαγές στην τιμή κάποιου bit λόγω εξωτερικών επιδράσεων όπως η κοσμική ακτινοβολία και τα σωματίδια άλφα. Αναφέρεται στην επίδραση που έχουν στην βιομηχανία ημιαγωγών αυτά τα φαινόμενα περιγράφοντας δύο περιστατικά που ενέπλεκαν την Sun Microsystems και την Normand. Συγκεκριμένα η Sun το 2000 αναγνώρισε πως κάποια σφάλματα σε μη προστατευμένες κρυφές μνήμες οφείλονταν σε κρούσεις σωματιδίων κοσμικής ακτινοβολίας και με αυτό τον τρόπο έχασε έναν μεγάλο πελάτη που πήγε στην ανταγωνίστρια IBM. Η Normand το 1996 ανέφερε πάρα πολλά παρόμοια περιστατικά τα οποία εντόπισε μελετώντας τα αρχεία καταγραφής σφαλμάτων πολλών μεγάλων υπολογιστικών συστημάτων της. Επίσης, ο φόβος και μόνο για αυτά τα σφάλματα έκανε την Fujitsu να προστατεύσει το 80% των bit σε ένα πρόσφατο μοντέλο του SPARC επεξεργαστή της.

Κάνει μία αναφορά στις διάφορες μεθόδους που υπάρχουν για την αντιμετώπιση τέτοιων λαθών αλλά σημειώνεται πως όλες αυτοί οι μέθοδοι έχουν σημαντικό κόστος στην απόδοση, στην απαιτούμενη ενέργεια, στο μέγεθος του τσιπ και στον χρόνο που απαιτείται για τη σχεδίαση. Ισχυρίζεται πως είναι καλύτερο ο υπολογισμός για το ποιες περιοχές πρέπει να προστατευθούν να γίνεται στα πρώτα στάδια της σχεδίασης.

Στην μέθοδο που περιγράφει η εργασία [11], χωρίζει τα bit κάθε δομής σε δύο κατηγορίες` στα ACE και στα un-ACE bit. Τα ACE bit είναι αυτά στα οποία αν συμβεί κάποιο σφάλμα, θα επηρεάσει την εκτέλεση του προγράμματος. Τα υπόλοιπα bit είναι τα un-ACE bit τα οποία δεν έχουν επίδραση στην ορθή εκτέλεση του προγράμματος.

Οι συγγραφείς δηλώνουν πως το αν κάποιο bit είναι ACE ή un-ACE εξαρτάται από το πρόγραμμα που εκτελείται καθώς και από τι θεωρούμε “έξοδο” της δομής που μελετάμε. Συνήθως έξοδος αποτελεί ο δίαυλος εισόδου/εξόδου ενός επεξεργαστή. Αν όμως την εκτέλεση την παρακολουθούμε με κάποιον debugger, έξοδος μπορεί να θεωρηθεί και μία μεταβλητή της οποίας την τιμή παρακολουθούμε. Επομένως είναι πολύ σημαντικό να ορίσουμε σωστά το τι καθιστά την έξοδο του συστήματος που μελετάμε. Θεωρούμε πως όλα τα bit είναι ACE ώστε η ανάλυσή μας να είναι όσο ασφαλέστερη γίνεται και προσπαθούμε να βρούμε ποια από αυτά τα bit είναι un-ACE.

Υπάρχουν δύο κύριες κατηγορίες των μη κρίσιμων bit. Στη μία κατηγορία ανήκουν τα μη κρίσιμα bit της μικροαρχιτεκτονικής. Τέτοια bit μπορούν να προκύψουν από τέσσερις καταστάσεις. Αχρησιμοποίητα bit κατάστασης, bit που ανήκουν σε εντολές που προσκομίστηκαν με διαδικασίες εικασίας οι οποίες αποδεικνύονται λανθασμένες, bit που ανήκουν σε δομές που επηρεάζουν την απόδοση του συστήματος και όχι την ορθή εκτέλεση όπως είναι οι δομές πρόβλεψης διακλάδωσης και bit τα οποία ήταν ACE αλλά παύουν να είναι μετά τη χρήση τους.

Στην δεύτερη κατηγορία ανήκουν τα μη κρίσιμα bit τα οποία θα μπορούσαν να επηρεάσουν την ορθή εκτέλεση του προγράμματος. Οι συγγραφείς εντοπίζουν πέντε κατηγορίες τέτοιων bit. Bit τα οποία ανήκουν σε εντολές NOP, bit που ανήκουν σε εντολές που βελτιώνουν την απόδοση του συστήματος, bit κώδικα του οποίου το αποτέλεσμα δεν χρησιμοποιείται, εντολές οι οποίες προσκομίστηκαν αλλά δεν εκτελέστηκαν λόγω της συνθήκης που είναι ενσωματωμένη σε αυτές καθώς και bit των οποίων το αποτέλεσμα επικαλύπτεται από κάποια επόμενη πράξη.

Για τον υπολογισμό του AVF με την μέθοδο που παρουσιάζεται στην εργασία [11] χρειάζονται τα εξής στοιχεία:

- Άθροισμα όλων των κύκλων παραμονής όλων των ACE bit σε κάθε δομή που εξετάζουμε κατά την εκτέλεση του προγράμματος.
- Σύνολο των κύκλων που χρειάστηκαν για την εκτέλεση του προγράμματος.
- Συνολικός αριθμός των bit που είναι παρόντα στην δομή που εξετάζουμε.

Προκειμένου να υπολογίσουν τον AVF, χρειάστηκαν 3 στάδια. Πρώτον, όπως κυλάει μία εντολή μέσα στον επεξεργαστή, καταγράφεται σε κάθε δομή από την οποία περνάει πληροφορία σχετική με τους κύκλους που παρέμεινε σε κάθε δομή, το αν η εντολή δεσμεύτηκε [1]στο τέλος και άλλες πληροφορίες. Στο 2ο στάδιο, όταν η εντολή δεσμεύεται, μπαίνει σε ένα παράθυρο δευτερεύουσας ανάλυσης όπου εξετάζουν αν το αποτέλεσμά της χρησιμοποιείται με κάποιο τρόπο ή αν τελικά δεν έχει καμία επίδραση

[1] Σε ορισμένες αρχιτεκτονικές, όταν η εκτέλεση μίας εντολής ολοκληρώνεται, τα αποτελέσματά της δεν αποθηκεύονται αμέσως αλλά μπαίνουν σε μία ουρά. Όταν εκπληρωθούν ορισμένες προϋποθέσεις, τα αποτελέσματα αυτά αποθηκεύονται στην μνήμη συστήματος και λέμε πως η εντολή είναι πλέον δεσμευμένη (committed).



Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά

στην έξοδο που παρατηρείται από τον χρήστη. Τέλος, στο 3ο στάδιο, χρησιμοποιώντας την πληροφορία που μαζεύτηκε στα στάδια 1 και 2, υπολογίζουν τον AVF όποιας δομής επιθυμούν.

Αναφορές γίνονται στην ορολογία των σφαλμάτων. Συγκεκριμένα αναφέρεται στους όρους MTBF, FIT, SDC, DUE και AVF. Αναφέρει πως οι σχεδιαστές χρησιμοποιούν πολύπλοκα και λεπτομερή μοντέλα για να υπολογίζουν τον ρυθμό FIT (Failure In Time) ο οποίος είναι πόσα σφάλματα συμβαίνουν σε μία περιοχή σε χρονικό διάστημα ενός δισεκατομμυρίου ωρών. Ο ρυθμός FIT ο οποίος εν τέλει επηρεάζει την λειτουργία της συσκευής είναι το γινόμενο του γενικού FIT και του AVF. Σημερινές τιμές ενός τυπικού ρυθμού FIT για κυκλώματα SRAM είναι μεταξύ 0,001-0,01 FIT/bit στο επίπεδο της θάλασσας. Αυτός ο ρυθμός αναμένεται να παραμείνει σταθερός για αρκετές γενιές ακόμα εκτός και εάν μειωθεί δραματικά η τάση του ρεύματος που χρειάζεται από ένα τσιπ για να λειτουργήσει. Ο ρυθμός FIT που σχετίζεται με λογικές πύλες είναι αμελητέος μπροστά σε αυτόν των στοιχείων μνήμης οπότε δεν μας απασχολεί.

Τέλος, παρουσιάζουν τα αποτελέσματα της ανάλυσής τους σε ένα μοντέλο βασισμένο σε έναν Itanium2 IA64 επεξεργαστή.

## ~~2.3~~ 2.3 ΥΠΑΡΧΟΝΤΑ ΠΑΚΕΤΑ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ AVF

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Στις εργασίες που αναφέραμε στην ενότητα 2.2, χρησιμοποιήθηκε το πλαίσιο (framework) “Asim” [16]. Βασικό πακέτο λογισμικού του συγκεκριμένου πλαισίου είναι το “Architects' Workbench”. Ενώ αρχικά έγινε μία προσπάθεια να χρησιμοποιηθεί το ίδιο λογισμικό και η ίδια εργασία, τελικά πάρθηκε η απόφαση ανάπτυξης ενός καινούριου πακέτου λογισμικού, ειδικά κατασκευασμένου για τον σκοπό του υπολογισμού του AVF ενός συστήματος. Κύριοι λόγοι για αυτήν την απόφαση ήταν η σχεδόν ανύπαρκτη υποστήριξη του Architect's Workbench καθώς και κάποιες ιδιαιτερότητες του που καθιστούσαν την εγκατάσταση και λειτουργία του προβληματική. Συγκεκριμένα για την σωστή λειτουργία του απαιτεί συγκεκριμένα πακέτα Linux να είναι εγκατεστημένα στο σύστημα. Μερικά από αυτά τα πακέτα είτε είναι δυσεύρετα, είτε έχουν αλλάξει σημαντικά στην τρέχουσα έκδοσή τους και δεν είναι συμβατά με την τρέχουσα έκδοση του Architect’s Workbench. Επίσης, η τεκμηρίωση του προγράμματος είναι διάσπαρτη και ασαφής.

Ένας άλλος προσομοιωτής που χρησιμοποιείται και είναι ο επίσημα υποστηριζόμενος από την MIPS είναι το “Open Virtual Platforms™ (OVP™)” project. Το πρόβλημα με αυτό το πακέτο είναι πως ναι μεν η MIPS παρέχει μοντέλα των επεξεργαστών της για προσομοίωση αλλά τα μοντέλα αυτά είναι κλειστά με μη διαθέσιμο τον πηγαίο κώδικά τους. Επίσης, η ανάπτυξη νέων μοντέλων δεν είχε νόημα μιας και η προσομοίωση σε αυτό το πακέτο λογισμικού γίνετε σε επίπεδο λειτουργικότητας και όχι μικροαρχιτεκτονικής.

Υπάρχουν και κάποια άλλα πακέτα λογισμικού τα οποία όμως δέχονται το μοντέλο υπό μορφή behavioral VHDL αλλά στη συγκεκριμένη εργασία εστιάζουμε στον υπολογισμό του AVF στα πρώτα στάδια σχεδίασης όπου ο κώδικας VHDL δεν είναι ακόμα διαθέσιμος.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

# 3 Ορισμός του προβλήματος και μεθοδολογία

## 3.1 ΕΙΣΑΓΩΓΗ

Βασικός στόχος αυτής της εργασίας είναι ο υπολογισμός του AVF ενός μικροεπεξεργαστή. Βασική έγνοια μας είναι ο υπολογισμός αυτός να μπορεί να γίνει στα πρώτα στάδια της σχεδίασης ενός επεξεργαστή. Κύριος λόγος για αυτή την απόφαση είναι το γεγονός πως σε μετέπειτα στάδια μερικές αποφάσεις που έχουν ληφθεί σχετικά με την σχεδίαση είναι δύσκολο να αλλάξουν. Επίσης η προσομοίωση ενός συστήματος απαιτεί όλο και περισσότερο χρόνο, όσο λεπτομερέστερο είναι το μοντέλο που θα χρησιμοποιηθεί [10].

Όπως είδαμε στα προηγούμενα κεφάλαια, υπάρχουν διάφοροι τρόποι υπολογισμού του AVF. Η συγκεκριμένη εργασία στρέφεται γύρω από τον υπολογισμό του AVF ενός μοντέλου μικροεπεξεργαστή αρχιτεκτονικής MIPS 32 με τη μέθοδο της ανάλυσης ACE. Η αρχιτεκτονική MIPS επιλέχτηκε κυρίως λόγω της απλότητάς της και η ανάλυση ACE λόγω του ότι είναι καλά καταγεγραμμένη τεχνική και η ποιότητα των αποτελεσμάτων της εξαρτάται από τη λεπτομέρεια και την ακρίβεια του μοντέλου και όχι από παράγοντες όπως στατιστικά σφάλματα.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

## 3.2 ΠΕΔΙΟ ΕΦΑΡΜΟΓΗΣ

Μολονότι η συγκεκριμένη εργασία επικεντρώνεται στην αρχιτεκτονική MIPS 32, πρέπει να σημειωθεί πως το λογισμικό που αναπτύχθηκε μπορεί εύκολα να επεκταθεί προκειμένου να μπορέσει να υπολογιστεί και ο AVF κάποιας άλλης αρχιτεκτονικής. Επιλέγουμε την αρχιτεκτονική MIPS μιας και είναι αρκετά απλή στην μοντελοποίηση της. Επίσης περιγράφεται αρκετά καλά στην βιβλιογραφία και το σύνολο εντολών που περιλαμβάνει είναι περιορισμένο.

Δεν επιλέγουμε κάποια συγκεκριμένη υλοποίηση της αρχιτεκτονικής MIPS μιας και αντικείμενο δεν είναι να μοντελοποιήσουμε και να αξιολογήσουμε την απόδοση κάποιου συγκεκριμένου επεξεργαστή αλλά να βγάλουμε κάποια γενικά συμπεράσματα για την επίδραση των διαφόρων παραμέτρων που πρέπει να λάβει υπόψη του ένας σχεδιαστής κατά την σχεδίαση ενός επεξεργαστή καθώς και να αναπτύξουμε κάποιες τεχνικές τις οποίες μπορεί να χρησιμοποιήσει κάποιος σχεδιαστής προκειμένου να βρει τον AVF ενός υπολογιστικού συστήματος..

Θα αρκεστούμε λοιπόν στο να μοντελοποιήσουμε κυρίως τη διοχέτευση (pipeline) και την κρυφή μνήμη (cache memory). Θα υπάρχει και ένα μοντέλο της εξωτερικής μνήμης αλλά δεν θα είναι λεπτομερές. Ο λόγος για αυτή την επιλογή είναι πως με το να μοντελοποιήσουμε με μικρή λεπτομέρεια την εξωτερική μνήμη, εκτός του ότι μικραίνουμε το χρόνο προσομοίωσης, γενικεύουμε τα αποτελέσματά της μελέτης και δεν τα περιορίζουμε σε κάποια συγκεκριμένη υλοποίηση.

Φυσικά το λογισμικό που αναπτύξαμε μπορεί να τροποποιηθεί προκειμένου να μοντελοποιηθεί το σύστημα με πολύ μεγαλύτερη λεπτομέρεια αλλά αυτό δεν αποτελεί αντικείμενο αυτής της εργασίας.

### 3.2.1 Αρχιτεκτονική MIPS 32

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Η αρχιτεκτονική MIPS είναι μία RISC αρχιτεκτονική η οποία είναι αρκετά καλά καταγεγραμμένη στην βιβλιογραφία. Είναι ιδανική για κάποιον ο οποίος ξεκινάει να ασχολείται με θέματα αρχιτεκτονικής επεξεργαστών μιας και στη βασική της μορφή είναι πολύ απλή αλλά υπάρχουν και υλοποιήσεις της οι οποίες είναι αρκετά πολύπλοκες για όποιον θέλει να εμβαθύνει. Αυτοί είναι οι λόγοι στους οποίους βασίστηκε η απόφαση της επιλογής της συγκεκριμένης αρχιτεκτονικής προς μελέτη.

Όντας μία RISC αρχιτεκτονική, ξέρουμε πως κάθε εντολή είναι σταθερού μήκους. Μαζί με το γεγονός πως επιλέγουμε να μην περιλάβουμε πολύπλοκες δομές όπως branch predictors ή άλλα περιφερειακά μπορούμε να κρατήσουμε το μοντέλο αρκετά απλό ώστε η επεξεργαστική ισχύς που θα απαιτείται για την προσομοίωση του να μην είναι μεγάλη.

Η διοχέτευση της αρχιτεκτονικής MIPS χωρίζεται σε 5 στάδια τα οποία αναλύουμε στην ενότητα 4.3. Η διοχέτευση είναι σχεδιασμένη έτσι ώστε στην περίπτωση που δεν έχουμε κάποια αστοχία στην κρυφή μνήμη ή κάποιον κίνδυνο ελέγχου ή  δεδομένων, σε κάθε κύκλο ρολογιού εκτελείται μία εντολή. Επίσης ξέρουμε πως στις εντολές άλματος εκτελείται πάντα και η επόμενη εντολή της εντολής άλματος. Λόγω των τελευταίων δύο γεγονότων, η αρχιτεκτονική MIPS εκτός από την ταχύτητά της, είναι ιδιαιτέρως προβλέψιμη οπότε είναι ευκολότερο να παρακολουθήσει κανείς την εκτέλεση ενός προγράμματος καθώς και να εντοπίσει λάθη στην εκτέλεσή του.

Η κρυφή μνήμη χωρίζεται σε μνήμη εντολών και δεδομένων. Παρουσίαση του μοντέλου της κρυφής μνήμης ακολουθεί στην ενότητα 3.2.4.

### 3.2.2 Υποστηριζόμενο σύνολο εντολών

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Υλοποιούμε το βασικό σύνολο εντολών της αρχιτεκτονικής MIPS 32. Δεν υλοποιούμε εντολές κινητής υποδιαστολής μιας και είναι περισσότερο εντολές συνεπεξεργαστή παρά του πυρήνα της βασικής αρχιτεκτονικής. Ακολουθεί λίστα με τα μνημονικά ονόματα κάθε εντολής και τη λειτουργία της. Το μοντέλο MIPS που υλοποιήσαμε υποστηρίζει όλες τις παρακάτω 29 εντολές.

***add Rd, Rs, Rt***

Προσθέτει τα περιεχόμενα των καταχωρητών Rs και Rt και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Τα δεδομένα θεωρείται πως απεικονίζουν προσημασμένους αριθμούς.

***addi Rt, Rs, immediate***

Προσθέτει τα περιεχόμενα του καταχωρητή Rs στο πεδίο immediate το οποίο έχει επεκταθεί στα 32 bit με λειτουργία επέκτασης προσήμου. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt.

***addiu Rt, Rs, immediate***

Προσθέτει τα περιεχόμενα του καταχωρητή Rs στο πεδίο immediate το οποίο έχει επεκταθεί στα 32 bit με την προσθήκη 16 μηδενικών στις υψηλότερες θέσεις, χωρίς επέκταση πρόσημου. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt.

***addu Rd, Rs, Rt***

Προσθέτει τα περιεχόμενα των καταχωρητών Rs και Rt και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Τα δεδομένα θεωρείται πως απεικονίζουν μη προσημασμένους αριθμούς.

***and Rd, Rs, Rt***

Εκτελεί την λογική πράξη AND μεταξύ των bit των περιεχομένων των καταχωρητών Rs και Rt και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd.

***andi Rt, Rs, immediate***

Εκτελεί την λογική πράξη AND μεταξύ των bit των περιεχομένων του καταχωρητή Rs και του πεδίου immediate το οποίο έχει επεκταθεί στα 32 bit με την προσθήκη 16 μηδενικών στις υψηλότερες θέσεις, χωρίς επέκταση πρόσημου. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt.

***beq Rs, Rt, immediate***

Εφόσον τα περιεχόμενα των καταχωρητών Rs και Rt είναι ίσα, εκτελεί άλμα σε διεύθυνση που προκύπτει από τον παρακάτω τύπο:

JumpAddress = PC + 4 + (SignExtendedImmediate << 2)

***bne Rs, Rt, immediate***

Εφόσον τα περιεχόμενα των καταχωρητών Rs και Rt δεν είναι ίσα, εκτελεί άλμα σε διεύθυνση που προκύπτει από τον παρακάτω τύπο:

JumpAddress = PC + 4 + ( SignExtendedImmediate << 2 )

***j address***

Εκτελεί άλμα χωρίς όρους προς διεύθυνση που υπολογίζεται ως εξής:

Τα 4 υψηλότερα bit της νέας διεύθυνσης προέρχονται από τα 4 υψηλότερα bit της τρέχουσας διεύθυνσης. Τα δύο τελευταία bit συμπληρώνονται από δύο μηδενικά και τα υπόλοιπα ενδιάμεσα bit συμπληρώνονται από τα bit του πεδίου address.

***jal address***

Εκτελείται άλμα χωρίς όρους και η διεύθυνση της μεθεπόμενης εντολής αποθηκεύεται στον καταχωρητή $31. Η διεύθυνση άλματος υπολογίζεται με τον ίδιο τρόπο με την εντολή J:

Τα 4 υψηλότερα bit της νέας διεύθυνσης προέρχονται από τα 4 υψηλότερα bit της τρέχουσας διεύθυνσης. Τα δύο τελευταία bit συμπληρώνονται από δύο μηδενικά και τα υπόλοιπα ενδιάμεσα bit συμπληρώνονται από τα bit του πεδίου address.

***jr Rs***

Εκτελείται άλμα υπό όρους προς την διεύθυνση που περιέχεται στον καταχωρητή Rs. Μπορεί να χρησιμοποιηθεί σε συνδυασμό με την εντολή Jal προκειμένου να επιστρέφει η εκτέλεση του προγράμματος στο σημείο που ήταν μετά από κλήση μίας ρουτίνας.

***lbu Rt, immediate(Rs)***

Διαβάζεται ένα byte από τη θέση μνήμης που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του με πρόσημο επεκταμένου πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt με τα 24 υψηλότερα bit να συμπληρώνονται με μηδενικά.

***lhu Rt, immediate(Rs)***

Διαβάζονται δύο bytes από τη θέση μνήμης που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του με πρόσημο επεκταμένου πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt με τα 16 υψηλότερα bit να συμπληρώνονται με μηδενικά.

***lui Rt, immediate***

Τα 16 bit του πεδίου immediate αποθηκεύονται στα 16 υψηλότερα bit του καταχωρητή Rt. Τα υπόλοιπα 16 bit του καταχωρητή Rt συμπληρώνονται με μηδενικά.

***lw Rt, immediate(Rs)***

Διαβάζεται μία λέξη των 32 bit από τη θέση μνήμης που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του με πρόσημο επεκταμένου πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt.

***nor Rd, Rs, Rt***

Εκτελείται η λογική πράξη NOR μεταξύ των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd.

***or Rd, Rs, Rt***

Εκτελείται η λογική πράξη OR μεταξύ των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd.

***ori Rt, Rs, immediate***

Εκτελείται η λογική πράξη OR μεταξύ των περιεχομένων του καταχωρητή Rs και του επεκταμένου με μηδενικά πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt.

***slt Rd, Rs, Rt***

Εφόσον έχουμε Rs < Rt, η τιμή του Rd τίθεται ίση με 0x00000001. Ειδάλλως, η τιμή του Rd γίνεται 0x00000000. Τα δεδομένα που περιέχονται στους καταχωρητές Rs και Rt θεωρούνται προσημασμένοι αριθμοί.

***slti Rt, Rs, immediate***

Εφόσον έχουμε Rs < immediate, η τιμή του Rt τίθεται ίση με 0x00000001. Ειδάλλως, η τιμή του Rt γίνεται 0x00000000. Το πεδίο immediate επεκτείνεται στα 32 bit με τη μέθοδο επέκτασης πρόσημου.

***sltiu Rt, Rs, immediate***

Εφόσον έχουμε Rs < immediate, η τιμή του Rt τίθεται ίση με 0x00000001. Ειδάλλως, η τιμή του Rt γίνεται 0x00000000. Το πεδίο immediate επεκτείνεται στα 32 bit χωρίς να γίνετε επέκταση πρόσημου.

***sltu Rd, Rs, Rt***

Εφόσον έχουμε Rs < Rt, η τιμή του Rd τίθεται ίση με 0x00000001. Ειδάλλως, η τιμή του Rd γίνεται 0x00000000. Τα δεδομένα που περιέχονται στους καταχωρητές Rs και Rt θεωρούνται μη προσημασμένοι αριθμοί.

***sll Rd, Rt, shamt***

**Μορφοποιήθηκε:** Ελληνικά

Εκτελείται αριστερή λογική ολίσθηση των περιεχομένων του καταχωρητή Rt κατά τόσες θέσεις όσες ορίζονται στο πεδίο shamt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd.

***srl Rd, Rt, shamt***

Εκτελείται δεξιά λογική ολίσθηση των περιεχομένων του καταχωρητή Rt κατά τόσες θέσεις όσες ορίζονται στο πεδίο shamt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd.

***sb Rt, immediate(Rs)***

Αποθηκεύεται το λιγότερο σημαντικό byte του καταχωρητή Rt στην μνήμη σε διεύθυνση που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του με πρόσημο επεκταμένου πεδίου immediate.

***sh Rt, immediate(Rs)***

Αποθηκεύονται τα δύο λιγότερο σημαντικά byte του καταχωρητή Rt στην μνήμη σε διεύθυνση που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του με πρόσημο επεκταμένου πεδίου immediate.

***sw Rt, immediate(Rs)***

Αποθηκεύεται το περιεχόμενο του καταχωρητή Rt στην μνήμη σε διεύθυνση που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του με πρόσημο επεκταμένου πεδίου immediate.

***sub Rd, Rs, Rt***

Αφαιρείται το περιεχόμενο του καταχωρητή Rt από το περιεχόμενο του καταχωρητή Rs. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. Οι αριθμοί θεωρούνται προσημασμένοι.

***subu Rd, Rs, Rt***

Αφαιρείται το περιεχόμενο του καταχωρητή Rt από το περιεχόμενο του καταχωρητή Rs. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. Τα δεδομένα των Rs και Rt θεωρούνται μη προσημασμένοι αριθμοί.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### ~~3.2.3~~ 3.2.3 Μοντέλο διοχέτευσης

Η διοχέτευση ενός επεξεργαστή MIPS αποτελείται από 5 στάδια. Ανάλογα με το ποιες τεχνολογίες χρησιμοποιούνται, τόσο πολυπλοκότερα μπορούν να είναι αυτά τα στάδια. Εδώ ασχολούμαστε με μία απλή υλοποίηση. Δεν περιλαμβάνουμε δομές για πρόβλεψη διακλάδωσης ή για εκτέλεση εκτός σειράς. Επίσης δεν υποστηρίζονται πράξεις κινητής υποδιαστολής. Υποστηρίζεται το βασικό σύνολο εντολών που αναφέρθηκε στην ενότητα 3.2.2 και μοντελοποιούνται οι άκρως απαραίτητες μονάδες για την υλοποίησή του. Επίσης, υλοποιούνται μονάδες ελέγχου κινδύνων δεδομένων και ελέγχου. Για την αντιμετώπιση αυτών των κινδύνων, μοντελοποιήσαμε λογική προώθησης δεδομένων καθώς και εισαγωγής καθυστερήσεων (stalls) στη διοχέτευση.

***Στάδιο προσκόμισης εντολής (Instruction Fetch, IF)***

Το πρώτο στάδιο είναι το στάδιο προσκόμισης εντολής (Instruction Fetch, IF για συντομία). Σε αυτό το στάδιο διαβάζεται η προς εκτέλεση εντολή από τη μνήμη. Είναι το 2ο απλούστερο στάδιο στη διοχέτευση της αρχιτεκτονικής MIPS μιας και δεν εκτελείται καμία άλλη εργασία πέρα από αναγνώσεις μνήμης και αύξηση του καταχωρητή προγράμματος (Program Counter) ώστε να δείχνει στην εντολή που πρέπει να αναγνωσθεί στον επόμενο κύκλο ρολογιού.

*Στάδιο αποκωδικοποίησης εντολής (Instruction Decode, ID)*

Το στάδιο αποκωδικοποίησης εντολής σκοπό έχει την ανάγνωση της εντολής που προσκομίστηκε από το στάδιο προσκόμισης και την αποκωδικοποίησή της. Σκοπός αυτού του σταδίου είναι να δημιουργηθούν τα κατάλληλα bit ελέγχου και να αναγνωστούν τα σωστά δεδομένα από το αρχείο καταχωρητών και να προωθηθούν στα επόμενα στάδια για επεξεργασία. Εδώ βρίσκεται μέρος της λογικής εντοπισμού κινδύνων δεδομένων καθώς και η λογική εντοπισμού κινδύνων ελέγχου. Επίσης εδώ πραγματοποιείται η ανάγνωση των δεδομένων εισόδου των εντολών από το αρχείο καταχωρητών, όπου αυτό απαιτείται.

*Στάδιο εκτέλεσης εντολής (Execute, EX)*

Στο στάδιο εκτέλεσης γίνονται όλες οι μαθηματικές και λογικές πράξεις κάθε εντολής. Εδώ φτάνουν τα δεδομένα των τελεστέων από το στάδιο αποκωδικοποίησης, η διεύθυνση άλματος στην περίπτωση εντολών άλματος καθώς και δεδομένα που αφορούν το στάδιο προσπέλασης μνήμης για τις εντολές ανάγνωσης και αποθήκευσης δεδομένων από και προς τη μνήμη. Εδώ βρίσκεται μέρος της λογικής εντοπισμού κινδύνων δεδομένων καθώς και μέρος της λογικής προώθησης δεδομένων.

*Στάδιο προσπέλασης μνήμης (Memory, MEM)*

Στο στάδιο προσπέλασης μνήμης, βρίσκεται η λογική που αλληλεπιδρά με την κρυφή μνήμη δεδομένων. Σαν εισόδους δέχεται μία διεύθυνση μνήμης καθώς και δεδομένα προς εγγραφή αν πρόκειται περί εντολή εγγραφής αντί για ανάγνωσης. Επίσης, μας ενδιαφέρει και το μήκος των δεδομένων που θα αναγνωστούν ή θα εγγραφούν. Αυτά μπορεί να είναι ένα byte, δύο bytes είτε μία πλήρη λέξη των τεσσάρων bytes. Τα αποτελέσματα από εντολές ανάγνωσης από τη μνήμη καθώς και ό,τι δεδομένα προέρχονται από το στάδιο εκτέλεσης, προωθούνται στο στάδιο επανεγγραφής.

*Στάδιο εγγραφής αποτελεσμάτων (Write Back, WB)*

Το στάδιο εγγραφής αποτελεσμάτων είναι υπεύθυνο για να αποθηκεύσει τα δεδομένα που προέρχονται από κάποιο από τα προηγούμενα στάδια σε κάποιον καταχωρητή στο αρχείο καταχωρητών. Είναι το απλούστερο στάδιο στη διοχέτευση ενός MIPS όσον αφορά την σχεδίαση και υλοποίησή του.

### ~~3.2.4~~ 3.2.4 Μοντέλο Κρυφής μνήμης

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Η κρυφή μνήμη στην αρχιτεκτονική MIPS βασίζεται στην αρχιτεκτονική Harvard. Σύμφωνα με την αρχιτεκτονική αυτή, η κρυφή μνήμη χωρίζεται στη μνήμη προγράμματος και στη μνήμη δεδομένων.

Αυτός ο διαχωρισμός προέκυψε από τους πρώτους υπολογιστές όπου το πρόγραμμα ήταν αποθηκευμένο σε διάτρητες κάρτες, επομένως η μνήμη προγράμματος δεν ήταν εγγράψιμη. Αργότερα παρουσιάστηκε η αρχιτεκτονική von Newmann σύμφωνα με την οποία η μνήμη δεδομένων και προγράμματος ήταν κοινή.

Τα πλεονεκτήματα της αρχιτεκτονικής Harvard είναι πως το σύστημα μπορεί να προσκομίζει μία εντολή με ταυτόχρονη ανάγνωση ή εγγραφή δεδομένων στη μνήμη του συστήματος. Το μειονέκτημα είναι πως η μνήμη δεδομένων και η μνήμη προγράμματος έχουν σταθερό μέγεθος, οριζόμενο από την αρχιτεκτονική. Σε αντίθεση, συστήματα που βασίζονται στην αρχιτεκτονική von Newmann, έχουν την δυνατότητα να μπορούν να χρησιμοποιήσουν όλη την διαθέσιμη μνήμη καθώς και να μπορούν να αλλάζουν τον κώδικα του προγράμματος κατά την εκτέλεση, μία πρακτική που δεν θεωρείται καλή αλλά έχει τις χρήσεις της, αλλά δεν μπορούν να προσκομίσουν μία εντολή και να προσπελαύνουν δεδομένα ταυτόχρονα.

Οι σχεδιαστές της αρχιτεκτονικής MIPS, ενδιαφερόμενοι περισσότερο για την ταχύτητα, αποφάσισαν να βασιστούν στην αρχιτεκτονική Harvard. Επομένως, ένας επεξεργαστής MIPS περιέχει δύο είδη κρυφής μνήμης, δεδομένων και προγράμματος.

***Κρυφή μνήμη προγράμματος (Instruction Cache)***

Η κρυφή μνήμη προγράμματος περιέχει τις εντολές που θα εκτελέσει ο επεξεργαστής. Το στάδιο προσκόμισης εντολής της διοχέτευσης ζητάει κάθε φορά τα δεδομένα που περιέχονται σε μία διεύθυνση. Αυτά τα δεδομένα αναπαριστούν μία εντολή η οποία αποκωδικοποιείται σε μετέπειτα στάδιο της διοχέτευσης και προωθείται προς εκτέλεση.

Η κρυφή μνήμη προγράμματος δεν μπορεί να εγγραφεί απευθείας από κανένα στάδιο της διοχέτευσης. Τα δεδομένα της προέρχονται από τη μνήμη συστήματος και η ανάγνωσή τους πραγματοποιείται όταν αυτό είναι απαραίτητο. Πιο συγκεκριμένα, μία ανάγνωση από τη μνήμη συστήματος προς την μνήμη δεδομένων συμβαίνει όταν ζητηθούν τα δεδομένα κάποιας διεύθυνση τα οποία δεν βρίσκονται στην κρυφή μνήμη προγράμματος.

Λόγω του γεγονότος αυτού, πως δηλαδή δεν μπορεί να γράψει η διοχέτευση απευθείας στην κρυφή μνήμη προγράμματος, η δομή της είναι απλούστερη σε σχέση με τη δομή της κρυφής μνήμης δεδομένων. Σε μία τυπική κρυφή μνήμη, ανάλογα το είδος της, πρέπει σε τακτά χρονικά διαστήματα να ελέγχεται το αν βρίσκεται σε συμφωνία με την μνήμη του συστήματος. Αν βρεθεί πως η κρυφή μνήμη έχει κάποια δεδομένα νεότερα από αυτά που είναι αποθηκευμένα στην μνήμη του συστήματος, τότε πρέπει να πραγματοποιηθούν οι σχετικές ενημερώσεις. Λόγω όμως του γεγονότος πως οι διευθύνσεις μνήμης που χρησιμοποιούνται για τον κώδικα του προγράμματος δεν γράφονται ποτέ, δεν χρειάζεται να γίνεται αυτός ο έλεγχος μιας και είμαστε σίγουροι πως τα δεδομένα είναι ίδια και στις δύο μνήμες.

***Κρυφή μνήμη δεδομένων (Data Cache)***

Μορφοποιήθηκε: Ελληνικά

Μορφοποιήθηκε: Ελληνικά

Η κρυφή μνήμη δεδομένων περιέχει ό,τι ακριβώς προτείνει το όνομά της. Είναι μία μνήμη κατάλληλη για την προσωρινή αποθήκευση δεδομένων τα οποία χρησιμοποιούνται από το τρέχον πρόγραμμα.

Πρόσβαση στην κρυφή μνήμη δεδομένων έχει η διοχέτευση μέσω του σταδίου προσπέλασης μνήμης. Δεδομένα μπορούν είτε να αποθηκευτούν, είτε να διαβαστούν από αυτή τη μνήμη. Αν πρόκειται περί εντολής ανάγνωσης, στη συνέχεια προωθούνται στο στάδιο Write Back ώστε να αποθηκευτούν σε κάποιον καταχωρητή του αρχείου καταχωρητών. Σε αυτή την περίπτωση, αν τα δεδομένα που ζητήθηκαν δεν βρίσκονται ήδη σε κάποιο block της κρυφής μνήμης, τότε η μνήμη επικοινωνεί με την μνήμη συστήματος. Τα δεδομένα μεταφέρονται μέσω του κατάλληλου διαύλου και καταλαμβάνουν την κατάλληλη θέση στη κρυφή μνήμη. Ταυτόχρονα, γίνονται διαθέσιμα στη διοχέτευση για περαιτέρω επεξεργασία.

Η διαδικασία της ανάγνωσης είναι πανομοιότυπη με την κρυφή μνήμη προγράμματος. Η διαφορά βρίσκεται στην περίπτωση της εγγραφής δεδομένων. Υπάρχουν δύο πολιτικές για τις περιπτώσεις εγγραφής:

- Ετερόχρονη εγγραφή (write back)
- Ταυτόχρονη εγγραφή (write through)

Στην πολιτική ετερόχρονης εγγραφής, η μεταφορά των τροποποιημένων δεδομένων προς τη μνήμη συστήματος συμβαίνει σε δύο περιπτώσεις. Σε τακτά χρονικά διαστήματα, όσα δεδομένα δεν έχουν ήδη αποθηκευτεί μετά από τροποποίησή τους στη μνήμη συστήματος, εγγράφονται σε αυτή. Εναλλακτικά, η εγγραφή συμβαίνει όταν πρέπει να αντικατασταθεί κάποιο block που περιέχει δεδομένα τα οποία δεν έχουν εγγραφεί στη μνήμη συστήματος. Αυτό μπορεί να συμβεί μετά από αίτημα ανάγνωσης όπου συμβαίνει τα δεδομένα από δύο διαφορετικές διευθύνσεις να πρέπει αν εγγραφούν στο ίδιο block. Στην πολιτική write through, η εγγραφή των δεδομένων γίνεται ταυτόχρονα και στην κρυφή μνήμη αλλά και στην μνήμη συστήματος.

Η μέθοδος ετερόχρονης εγγραφής μας επιτρέπει να γράψουμε τα δεδομένα στη μνήμη συστήματος όταν τα σχετικά μέρη του επεξεργαστή δεν χρησιμοποιούνται για κάποια άλλη λειτουργία. Με αυτό τον τρόπο κερδίζουμε κύκλους ρολογιού, άρα έχουμε και βελτιωμένη απόδοση για τον επεξεργαστή. Το μειονέκτημα είναι πως η δομή της κρυφής μνήμης γίνεται πολύπλοκη και ακριβότερη.

Από την άλλη, η μέθοδος ταυτόχρονης εγγραφής είναι η απλούστερη στην υλοποίηση άρα και η μέθοδος με το μικρότερο κόστος. Το πρόβλημα όμως είναι πως αν έχουμε συνεχόμενες προσπελάσεις η απόδοση είναι μικρότερη λόγω του ότι πρέπει κάθε φορά να περιμένουμε να ολοκληρωθεί η προηγούμενη προσπέλαση.

Στη συγκεκριμένη εργασία, η μέθοδος που θα μοντελοποιήσουμε είναι αυτή της ταυτόχρονης εγγραφής αλλά πρέπει να σημειωθεί πως μπορεί εύκολα να μοντελοποιηθεί οποιαδήποτε μέθοδος χωρίς να χρειαστεί να αλλάξει παρά μόνο τον κώδικα του μοντέλου της κρυφής μνήμης.

## ~~3.3~~ 3.3 ΜΕΘΟΔΟΛΟΓΙΑ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Για τον υπολογισμό του AVF στην αρχιτεκτονική που επιλέξαμε, χρησιμοποιήσαμε τη μέθοδο της ανάλυσης ACE. Κατά την ανάλυση ACE, το AVF δίνεται από τον τύπο:

$$AVF = \frac{\sum \text{Αριθμός των ACE bit σε μία δομή x Χρόνος παραμονής των bit στη δομή}}{\text{Σύνολο των bit στη δομή x Συνολικός χρόνος εκτέλεσης}}$$

Όπου η μονάδα του χρόνου είναι κύκλοι ρολογιού.

Όπως φαίνεται, θέλουμε έναν τρόπο προκειμένου να μπορούμε να καταγράφουμε σε κάθε κύκλο ρολογιού, πόσα ACE bits υπάρχουν μέσα σε κάθε δομή του επεξεργαστή. Επίσης, χρειάζεται να γνωρίζουμε τον συνολικό χρόνο εκτέλεσης του προγράμματος. Αφού αποκτήσουμε αυτά τα δεδομένα, ο υπολογισμός του AVF είναι πολύ εύκολος. Η μέθοδός μας είναι παρόμοια με αυτή που περιγράφεται στην εργασία [11].

Χρειαζόμαστε ένα μοντέλο του pipeline του επεξεργαστή το οποίο περιέχει τις κατάλληλες μονάδες απαραίτητες για το σύνολο εντολών που θέλουμε να υποστηρίξουμε. Αυτό το μοντέλο εκτός του ότι πρέπει να εκτελεί σωστά προγράμματα βασιζόμενα σε αυτό το σύνολο εντολών, πρέπει να έχει ενσωματωμένες λειτουργίες προκειμένου να μας επιστρέφει πληροφορίες για τα ACE bit κάθε δομής καθώς και το χρόνο σε κύκλους ρολογιού για τον οποίο τα bit αυτά παρέμειναν στη δομή. Επίσης χρειαζόμαστε τις ίδιες πληροφορίες για το σύνολο των bit στον επεξεργαστή που μοντελοποιούμε. Χρήσιμο είναι να μπορούμε να δούμε τα δεδομένα που περιέχονται στη μνήμη του μοντέλου μας καθώς και αυτά που περιέχονται στους διάφορους καταχωρητές.

Με βάση τα παραπάνω, καταλήξαμε πως η καλύτερη μέθοδος είναι η ανάπτυξη ενός προγράμματος βασισμένο στη λογική MVC (Model View Controller) [18]. Αναπτύξαμε ένα μοντέλο (Model) της αρχιτεκτονικής MIPS 32 όπως αυτή περιγράφεται σε αυτή την εργασία. Κάθε εντολή, όπως κινείται μέσα στη διοχέτευση, έχει συνημμένη πληροφορία για τα ACE bit της σε κάθε στάδιο καθώς και για το χρόνο παραμονής της σε κάθε στάδιο. Το ποια bit είναι ACE bit για κάθε εντολή σε κάθε στάδιο το υπολογίζουμε θεωρητικά και στη συνέχεια, εισάγουμε αυτή την πληροφορία στον κώδικα του προσομοιωτή προκειμένου να συνάπτει αυτή την πληροφορία σε κάθε εντολή που αποκωδικοποιείται στο στάδιο αποκωδικοποίησης εντολής της διοχέτευσης. Σε κάθε κύκλο ρολογιού, το στάδιο ελέγχου (Controller) του μοντέλου, στέλνει "παλμούς ρολογιού" στο μοντέλο και στο τέλος κάθε κύκλου προσομοίωσης, διαβάζει και καταγράφει την πληροφορία που είναι συνημμένη στην εντολή που βρίσκεται στο τελευταίο στάδιο της διοχέτευσης. Τέλος, μετά τη λήξη της προσομοίωσης, το στάδιο της παρουσίασης των δεδομένων (View) του προγράμματος που αναπτύξαμε, αποθηκεύει τα δεδομένα που συλλέγησαν κατά την προσομοίωση σε ένα αρχείο καταγραφής σε μορφή κατάλληλη για περαιτέρω επεξεργασία προκειμένου να μπορούμε να εξάγουμε συμπεράσματα για τον AVF. Το αρχείο αυτό, στη συνέχεια το εισάγουμε στο πρόγραμμα MATLAB προκειμένου να επεξεργαστούμε τα δεδομένα με σκοπό την καλύτερη κατανόηση της συμπεριφοράς του επεξεργαστή σχετικά με τον AVF.

## ~~3.4~~ 3.4 ΜΟΝΤΕΛΟ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ MIPS32

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Στις επόμενες ενότητες παρουσιάζουμε αναλυτικά την αρχιτεκτονική MIPS που μοντελοποιούμε. Παρουσιάζουμε το βασικό σύνολο εντολών, αναλύουμε κάθε στάδιο της διοχέτευσης, παρουσιάζουμε τη δομή της μνήμης συστήματος και πραγματοποιούμε ανάλυση για το ποια bit κάθε σταδίου της διοχέτευσης είναι ACE bit σε κάθε εντολή.

## ~~3.5~~ 3.5 ΒΑΣΙΚΟΙ ΤΥΠΟΙ ΕΝΤΟΛΩΝ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Υλοποιούμε το βασικό σύνολο εντολών του MIPS32, όπως αυτό αναφέρεται στο βιβλίο "Computer Organization and Design" [9].

Στην αρχιτεκτονική MIPS, κάθε εντολή έχει σταθερό μήκος, 32 bits. Αυτό διευκολύνει κατά πολύ τη σχεδίαση της μονάδας αποκωδικοποίησης.

Οι εντολές χωρίζονται σε τρεις τύπους:

- Τύπος R.
- Τύπος I.
- Τύπος J.

Ο τύπος της κάθε εντολής εξαρτάται από την προέλευση των δεδομένων των τελεστέων καθώς και τον προορισμό του αποτελέσματος της εντολής. Τα δεδομένα μπορούν να έχουν είτε πηγή, είτε προορισμό κάποιον καταχωρητή του επεξεργαστή, είτε την μνήμη του συστήματος.

### 3.5.1 Τύπος R

Σε αυτήν την κατηγορία ανήκουν οι εντολές οι οποίες χρησιμοποιούν καταχωρητές και για τα δεδομένα εισόδου αλλά και για τον προορισμό των αποτελεσμάτων. Κάθε εντολή μπορεί να χρησιμοποιήσει έως και τρεις καταχωρητές. Δύο καταχωρητές για τα δεδομένα εισόδου και έναν για το αποτέλεσμα.

Τα πεδία της εντολής είναι τα εξής:

- Opcode – πεδίο κοινό σε κάθε εντολή, περιέχει τον κωδικό που προσδιορίζει μοναδικά την εντολή.
- Rs – Κωδικός πρώτου τελεστή.
- Rt – Κωδικός δεύτερου τελεστή.
- Rd – Κωδικός καταχωρητή αποθήκευσης του αποτελέσματος.
- Shamt – Πεδίο θέσεων ολίσθησης.
- Func – Πεδίο κωδικού αριθμητικής λειτουργίας.

Το πεδίο Opcode είναι ίδιο για όλες τις εντολές τύπου R. Συγκεκριμένα, έχει την τιμή $000000_2$. Τα πεδία Rs, Rt και Rd έχουν μήκος 5 bits το κάθε ένα και δείχνουν τον αριθμό του εκάστοτε καταχωρητή. Το πεδίο shamt έχει μήκος 5 bits και δείχνει τον αριθμό των θέσεων ολίσθησης. Το πεδίο Func περιέχει τον κωδικό της αριθμητικής ή λογικής πράξης που πρέπει να εκτελέσει η λογική μονάδα αριθμητικής (ALU).

Τα πεδία των εντολών τύπου R καθώς και οι θέσεις τους φαίνονται παρακάτω:

| opcode | Rs | Rt | Rd | shamt | func |
|---|---|---|---|---|---|
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |

Οι εντολές τύπου R του βασικού συνόλου εντολών είναι οι παρακάτω:

| **Μνημονικό** | **Λειτουργία** |
|---|---|
| add | Πρόσθεση των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| addu | Πρόσθεση χωρίς πρόσημο των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| and | Λογική πράξη AND μεταξύ των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| jr | Άλμα σε διεύθυνση που υποδεικνύεται από τον καταχωρητή Rs. |
| nor | Λογική πράξη NOR μεταξύ των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| or | Λογική πράξη OR μεταξύ των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| slt | Σύγκριση των τιμών των καταχωρητών Rs και Rt. Εφόσον έχουμε Rs < Rt, θέση της |

| Μνημονικό | Λειτουργία |
|---|---|
| | τιμής του Rd = 1. Ειδάλλως, Rd = 0. |
| sltu | Σύγκριση των τιμών των καταχωρητών Rs και Rt. Θεωρούμε πως τα δεδομένα είναι μη προσημασμένοι αριθμοί. Εφόσον έχουμε Rs < Rt, θέση της τιμής του Rd = 1. Ειδάλλως, Rd = 0. |
| sll | Ολίσθηση προς τα αριστερά των περιεχομένων του καταχωρητή Rt κατά τόσες θέσεις, όσες ορίζονται στο πεδίο shamt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| srl | Ολίσθηση προς τα δεξιά των περιεχομένων του καταχωρητή Rt κατά τόσες θέσεις, όσες ορίζονται στο πεδίο shamt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| sub | Αφαίρεση των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |
| subu | Αφαίρεση χωρίς πρόσημο των περιεχομένων των καταχωρητών Rs και Rt. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. |

**Πίνακας 1: Εντολές τύπου R που υποστηρίζει το μοντέλο μας.**

### ~~3.5.2~~ 3.5.2 Τύπος Ι

Μορφοποιημένο: Κουκκίδες και αρίθμηση

Σε αυτή την κατηγορία ανήκουν οι εντολές των οποίων μέρος τους αποτελεί τα δεδομένα ενός εκ των τελεστέων της εντολής. Συγκεκριμένα, στις εντολές τύπου Ι, τα τελευταία 16 bit αποτελούν το πεδίο Immediate το οποίο αναλόγως της εντολής επεκτείνεται σε 32 bit και χρησιμοποιείται ως τελεστής στα επόμενα στάδια της διοχέτευσης. Συνήθως οι πράξεις εκτελούνται μεταξύ αυτού του πεδίου και του καταχωρητή Rs. Το αποτέλεσμα της κάθε εντολής τύπου Ι συνήθως αποθηκεύεται στον καταχωρητή Rt. Εξαίρεση αποτελούν οι εντολές άλματος τύπου Ι, όπου το αποτέλεσμα αποθηκεύεται στον καταχωρητή PC καθώς και οι εντολές προσπέλασης μνήμης όπου το αποτέλεσμα αποθηκεύεται σε μία θέση στη μνήμη συστήματος.

Τα πεδία αυτών των εντολών είναι τα εξής:

- Opcode – πεδίο κοινό σε κάθε εντολή, περιέχει τον κωδικό που προσδιορίζει μοναδικά την εντολή.
- Rs – Κωδικός του πρώτου τελεστή.
- Rt – Κωδικός καταχωρητή αποθήκευσης του αποτελέσματος.
- Immediate – Πεδίο που περιέχει τα δεδομένα του δεύτερου τελεστή.

Το πεδίο opcode, όπως και στις υπόλοιπες εντολές, χρησιμεύει στην αναγνώριση της εντολής. Κάθε εντολή τύπου Ι έχει διαφορετικό κωδικό μήκους 6 bit ο οποίος βρίσκεται σε αυτό το πεδίο. Το πεδίο Rs περιέχει τον αριθμό του πρώτου καταχωρητή εισόδου και έχει μήκος 5 bit. Το πεδίο Rt, σε αντίθεση με τις εντολές τύπου R, περιέχει τον κωδικό του καταχωρητή προορισμού. Τα υπόλοιπα 16 bit, καταλαμβάνονται από το πεδίο Immediate. Σε αυτό το πεδίο περιέχονται τα δεδομένα του δεύτερου τελεστή. Τα 16 αυτά bit επεκτείνονται είτε με επέκταση προσήμου, είτε με την προσθήκη 16 μηδενικών στα 16 σημαντικότερα bit προκειμένου το μήκος της λέξης που θα προωθηθεί στο επόμενο στάδιο της διοχέτευσης να είναι 32bit.

Μορφοποιήθηκε: Ελληνικά

Μορφοποιήθηκε: Ελληνικά

Η μορφή μίας εντολής τύπου Ι φαίνεται στην παρακάτω εικόνα:

| opcode | Rs | Rt | Immediate |
|---|---|---|---|
| 31 26 | 25 21 | 20 16 | 15 0 |

Πρέπει να είναι εμφανές στον αναγνώστη, πως τα 16 bit του πεδίου immediate είναι ανεπαρκή για την αποθήκευση μίας λέξης των 32 bit που είναι το μήκος λέξης της αρχιτεκτονικής MIPS. Το μειονέκτημα αυτών των εντολών βρίσκεται στη μέγιστη τιμή του δεύτερου τελεστή. Πολλές φορές όμως θέλουμε να συγκρίνουμε μία μεταβλητή με μία σταθερή και σχετικά μικρή τιμή ή να εργαστούμε με ένα ή δύο byte τη φορά. Μία από τις συχνότερες λειτουργίες σε ένα πρόγραμμα για παράδειγμα, είναι η αύξηση ή μείωση της τιμής μίας μεταβλητής κατά μία μονάδα. Επίσης, τα περισσότερα άλματα που μπορεί να χρειαστεί να γίνουν σε ένα πρόγραμμα δεν απέχουν πολλές θέσεις μνήμης μεταξύ τους. Για αυτές τις περιπτώσεις, οι εντολές τύπου Ι είναι ιδιαίτερα χρήσιμες μιας και δεσμεύουμε έναν λιγότερο καταχωρητή σε σχέση με μία εντολή R και αποφεύγουμε προσπελάσεις στην μνήμη συστήματος μιας και ένας εκ των τελεστέων προσκομίζεται κατά το στάδιο IF της διοχέτευσης, όντας κομμάτι της προς εκτέλεση εντολής.

Στον παρακάτω πίνακα φαίνονται οι εντολές τύπου Ι:

| Μνημονικό | Λειτουργία |
|---|---|
| addi | Πρόσθεση των περιεχομένων του καταχωρητή Rs με το πεδίο Immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt. |
| addiu | Πρόσθεση των περιεχομένων του καταχωρητή Rs με το πεδίο Immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt. Οι αριθμοί θεωρούνται μη προσημασμένοι. |
| andi | Λογική πράξη AND μεταξύ των περιεχομένων του καταχωρητή Rs με το πεδίο Immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt. |
| beq | Άλμα σε διεύθυνση μνήμης που απέχει τόσες θέσεις όσες υποδεικνύονται από το |

| Μνημονικό | Λειτουργία |
|---|---|
| | πεδίο immediate, εφόσον τα περιεχόμενα των καταχωρητών Rs και Rt είναι ίσα. |
| bne | Άλμα σε διεύθυνση μνήμης που απέχει τόσες θέσεις όσες υποδεικνύονται από το πεδίο immediate, εφόσον τα περιεχόμενα των καταχωρητών Rs και Rt δεν είναι ίσα. |
| lbu | Ανάγνωση ενός byte από τη μνήμη συστήματος. Η διεύθυνση προκύπτει από την άθροιση των περιεχομένων του καταχωρητή Rs και του πεδίου Immediate. Το byte αποθηκεύεται στον καταχωρητή Rt και τα 24 υψηλότερα bit του συμπληρώνονται με μηδενικά. |
| lhu | Ανάγνωση δύο bytes από τη μνήμη συστήματος. Η διεύθυνση προκύπτει από την άθροιση των περιεχομένων του καταχωρητή Rs και του πεδίου Immediate. Τα bytes αποθηκεύονται στον καταχωρητή Rt και τα 16 υψηλότερα bit του συμπληρώνονται με μηδενικά. |
| lui | Αποθήκευση των 16 bits του πεδίου immediate στα 16 υψηλότερα bit του καταχωρητή Rt. Τα 16 λιγότερο σημαντικά bit συμπληρώνονται με μηδενικά. |
| lw | Ανάγνωση μίας λέξης των 32 bit από τη μνήμη συστήματος. Η διεύθυνση προκύπτει από την άθροιση των περιεχομένων του καταχωρητή Rs και του πεδίου Immediate. |
| ori | Λογική πράξη OR μεταξύ των περιεχομένων του καταχωρητή Rs με το πεδίο Immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rt. |
| slti | Σύγκριση των τιμών του καταχωρητή Rs και του πεδίου immediate. Εφόσον έχουμε Rs < Immediate, θέση της τιμής του Rt = 1. Ειδάλλως, Rd = 0. |
| sltiu | Σύγκριση των τιμών του καταχωρητή Rs και του πεδίου immediate. Εφόσον έχουμε Rs < Immediate, θέση της τιμής του Rt = 1. Ειδάλλως, Rd = 0. Οι τελεστές θεωρούνται μη προσημασμένοι αριθμοί. |
| sb | Αποθήκευση ενός byte από τον καταχωρητή Rt στη μνήμη συστήματος στη διεύθυνση που προκύπτει από την πρόσθεση του καταχωρητή Rs στο πεδίο immediate. |
| sh | Αποθήκευση δύο bytes από τον καταχωρητή Rt στη μνήμη συστήματος στη διεύθυνση που προκύπτει από την πρόσθεση του καταχωρητή Rs στο πεδίο immediate. |
| sw | Αποθήκευση μίας λέξης των 32 bit από τον καταχωρητή Rt στη μνήμη συστήματος στη διεύθυνση που προκύπτει από την πρόσθεση του καταχωρητή Rs στο πεδίο immediate. |

**Πίνακας 2: Εντολές τύπου I που υποστηρίζει το μοντέλο μας.**

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### ~~3.5.3~~ 3.5.3 Τύπος J

Οι εντολές τύπου J είναι εντολές άλματος. Έχουν δύο πεδία. Το πεδίο opcode και το πεδίο address. Το πεδίο address έχει μήκος 26 bit. Μιας και δεν έχουμε κανένα άλλο δεδομένο εισόδου πέρα της διεύθυνσης άλματος, οι εντολές τύπου J είναι άλματα άνευ όρων. Λόγω του γεγονότος αυτού, το πεδίο Address είναι σημαντικά μεγαλύτερο από τις εντολές άλματος τύπου I και επομένως τα άλματα μπορούν να είναι σε πολύ μεγαλύτερη απόσταση. Πιο συγκεκριμένα, οι εντολές τύπου J έχουν την παρακάτω εικόνα:

- Opcode – Πεδίο κοινό σε κάθε εντολή, περιέχει τον κωδικό που προσδιορίζει μοναδικά την εντολή.
- Address – Πεδίο που περιέχει μέρος της διεύθυνσης προορισμού.

Το πεδίο opcode είναι ίδιο σε μέγεθος και λειτουργία με τις εντολές I και R. Η διαφοροποίηση έρχεται στο πεδίο Address το οποίο έχει μήκος 26 bit και χρησιμοποιείται προκειμένου να υπολογίσουμε την διεύθυνση προορισμού.

Η μορφή μίας εντολής τύπου I φαίνεται στην παρακάτω εικόνα:

| opcode | Address |
|---|---|
| 31 26 | 25 0 |

Η διεύθυνση προορισμού υπολογίζεται ως εξής:

Λόγω του γεγονότος πως η κάθε εντολή έχει μήκος 4 bytes, μπορούμε να εξάγουμε το συμπέρασμα πως οι εντολές βρίσκονται σε διευθύνσεις οι οποίες είναι πολλαπλάσια του 4. Ποιο συγκεκριμένα, τα δύο τελευταία bit των διευθύνσεων είναι πάντα ίσα με $00_2$. Βασιζόμενοι σε αυτό το συμπέρασμα, μπορούμε να παραλείπουμε να αναγράφουμε αυτά τα 2 bit στο πεδίο Address και όταν έρχεται η στιγμή να υπολογίσουμε τη διεύθυνση της επόμενης εντολής, να προσθέτουμε αυτά τα δύο μηδενικά με μία πράξη ολίσθησης προς τα αριστερά κατά δύο θέσεις. Προκειμένου να συμπληρώσουμε τα άλλα 4 bit, χρησιμοποιούμε τα πρώτα 4 bit της τρέχουσας διεύθυνσης. Αποτέλεσμα είναι να μπορούμε κάθε φορά να πραγματοποιούμε ένα άλμα σε οποιοδήποτε σημείο ενός block 268,435,455 διευθύνσεων ή 255 Mbyte με μία μόνο εντολή.

Στον παρακάτω πίνακα φαίνονται οι εντολές τύπου J:

| Μνημονικό | Λειτουργία |
|---|---|
| j | Άλμα άνευ όρων σε διεύθυνση που υπολογίζεται από την τρέχουσα διεύθυνση και την τιμή του πεδίου Address. |
| jal | Άλμα άνευ όρων σε διεύθυνση που υπολογίζεται από την τρέχουσα διεύθυνση και την τιμή του πεδίου Address. Η διεύθυνση της μεθεπόμενης εντολής αποθηκεύεται στον καταχωρητή $31 ($ra). |

**Πίνακας 3: Εντολές τύπου J που υποστηρίζει το μοντέλο μας.**

Μορφοποιημένο: Κουκκίδες και αρίθμηση

## ~~3.6~~ 3.6 ΔΙΟΧΕΤΕΥΣΗ

Σε κάθε επεξεργαστή, κάθε εντολή και τα δεδομένα της ακολουθούν μία συγκεκριμένη διαδρομή. Αυτή η πορεία των δεδομένων μέσα στον επεξεργαστή ονομάζεται διαδρομή δεδομένων (datapath). Αν οι εντολές εξυπηρετούνταν μία από μία μονολιθική μονάδα, η σχεδίαση του επεξεργαστή θα ήταν πολύπλοκη και η απόδοση χαμηλά μιας και θα έπρεπε να συγχρονίζονται πολλές λειτουργικές μονάδες οι οποίες διαφέρουν πολύ ως προς τον χρόνο που απαιτεί κάθε μονάδα προκειμένου να εκτελέσει τη λειτουργία της. Επίσης, δεν χρησιμοποιούν όλες οι εντολές τις ίδιες λειτουργικές μονάδες ταυτόχρονα. Για παράδειγμα, μία εντολή πρόσθεσης χρησιμοποιεί την ALU και διαβάζει και αποθηκεύει δεδομένα

από το αρχείο καταχωρητών σε διαφορετικούς χρόνους. Αποτέλεσμα αυτού είναι να υπάρχουν στιγμές όπου διάφορες λειτουργικές μονάδες του επεξεργαστή να μην πραγματοποιούν κάποια εργασία.

Για να αντιμετωπισθούν αυτά τα προβλήματα, οι σχεδιαστές επεξεργαστών χωρίζουν το datapath σε πολλά και διαφορετικά μέρη και η εκτέλεση κάθε εντολής περνάει από όλα τα μέρη με την ίδια σειρά. Αποτέλεσμα αυτής της τεχνικής είναι η καλύτερη αξιοποίηση των λειτουργικών μονάδων του επεξεργαστή. Η δομή που προκύπτει ονομάζεται διοχέτευση (pipeline). Το παραπάνω μπορεί να γίνει καλύτερα κατανοητό με ένα παράδειγμα.

Ας υποθέσουμε πως έχουμε στη σειρά 3 εντολές που εκτελούν κάποια αριθμητική πράξη. Επίσης ας υποθέσουμε πως η διοχέτευση αποτελείται από τρία στάδια: Στάδιο προσκόμισης εντολής, στάδιο εκτέλεσης εντολής, στάδιο επανεγγραφής στο αρχείο καταχωρητών.

- Στον πρώτο κύκλο του ρολογιού, προσκομίζεται η 1η εντολή.
- Στον 2ο κύκλο η 1η εντολή βρίσκεται στο στάδιο εκτέλεσης και ταυτόχρονα προσκομίζεται η 2η εντολή.
- Στον 3ο κύκλο η 1η εντολή βρίσκεται στο στάδιο επανεγγραφής και αποθηκεύεται το αποτέλεσμά της, η 2η εντολή βρίσκεται στο στάδιο εκτέλεσης και ταυτόχρονα προσκομίζεται η 3η εντολή.
- Στον 4ο κύκλο η 2η εντολή βρίσκεται στο στάδιο επανεγγραφής και η 3η στο στάδιο εκτέλεσης.
- Στον 5ο κύκλο φτάνει και η 3η εντολή στο στάδιο επανεγγραφής και τελειώνει το πρόγραμμά μας.

| | IF | EX | WB |
|---|---|---|---|
| Κύκλος 1 | Εντολή 1 | | |
| Κύκλος 2 | Εντολή 2 | Εντολή 1 | |
| Κύκλος 3 | Εντολή 3 | Εντολή 2 | Εντολή 1 |
| Κύκλος 4 | | Εντολή 3 | Εντολή 2 |
| Κύκλος 5 | | | Εντολή 3 |

**Εικόνα 1: Εκτέλεση τριών εντολών με διοχέτευση τριών σταδίων.**

Στην περίπτωση που δεν υπήρχε η διοχέτευση, κάθε εντολή θα χρειαζόταν 3 κύκλους ρολογιού για την εκτέλεσή της. Αποτέλεσμα θα ήταν να χρειαζόμασταν 3 * 3 = 9 κύκλους αντί για μόλις 5. Η σημασία της διοχέτευσης είναι εμφανής.

| | IF | EX | WB |
|---|---|---|---|
| Κύκλος 1 | Εντολή 1 | | |
| Κύκλος 2 | | Εντολή 1 | |
| Κύκλος 3 | | | Εντολή 1 |
| Κύκλος 4 | Εντολή 2 | | |
| Κύκλος 5 | | Εντολή 2 | |
| Κύκλος 6 | | | Εντολή 2 |
| Κύκλος 7 | Εντολή 3 | | |
| Κύκλος 8 | | Εντολή 3 | |
| Κύκλος 9 | | | Εντολή 3 |

**Εικόνα 2: Εκτέλεση τριών εντολών χωρίς διοχέτευση.**

## ~~3.7~~ 3.7 ΣΤΑΔΙΑ ΤΗΣ ΔΙΟΧΕΤΕΥΣΗΣ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Στην αρχιτεκτονική MIPS32 που εξετάζουμε, η διοχέτευση χωρίζεται σε 5 στάδια μεταξύ των οποίων παρεμβάλλονται 4 καταχωρητές. Τα στάδια αυτά είναι τα:

- Instruction Fetch – Στάδιο προσκόμισης εντολής. Εδώ προσκομίζεται η εντολή από τη μνήμη.
- Instruction Decode – Στάδιο αποκωδικοποίησης εντολής. Εδώ αναγνωρίζεται η εντολή και δημιουργούνται τα κατάλληλα σήματα ελέγχου των επομένων σταδίων. Επίσης, σε αυτό το στάδιο διαβάζονται όποτε χρειάζεται τα δεδομένα των τελεστέων από το αρχείο καταχωρητών.
- EXecute – Στάδιο εκτέλεσης. Εδώ βρίσκεται η αριθμητική και λογική μονάδα (ALU) και πραγματοποιούνται όλες οι αριθμητικές και λογικές πράξεις.
- MEMmory – Στάδιο προσπέλασης μνήμης. Εδώ εκτελούνται όλες οι προσπελάσεις μνήμης.
- Write Back – Στάδιο επανεγγραφής. Εδώ γράφονται τα αποτελέσματα όποτε χρειάζεται στο αρχείο καταχωρητών.

### ~~3.7.1~~ 3.7.1 Στάδιο IF

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Στο στάδιο IF πραγματοποιείται η προσκόμιση των εντολών από τη μνήμη. Το εν λόγω στάδιο αποτελείται από έναν καταχωρητή, έναν πολυπλέκτη, μία μονάδα πρόσθεσης με τη σταθερά 4 καθώς και κύκλωμα επικοινωνίας με τη μνήμη εντολών. Αναλυτικότερα:

- Καταχωρητής PC (Program Counter) – Ο συγκεκριμένος καταχωρητής περιέχει τη διεύθυνση της εντολής που πρέπει να προσκομισθεί από τη μνήμη.
- Μονάδα πρόσθεσης – Αυτή η μονάδα προσθέτει την τιμή 4 στην τιμή του καταχωρητή PC ώστε και προκύπτει η διεύθυνση της επόμενης εντολής. Αυτό είναι εφικτό μιας και όλες οι εντολές της αρχιτεκτονικής MIPS έχουν σταθερό μήκος 4 bytes.
- Πολυπλέκτης – Η διεύθυνση της προς προσκόμιση εντολής στην περίπτωση που έχουμε εντολή άλματος προκύπτει στο στάδιο αποκωδικοποίησης και είναι διαθέσιμη στο στάδιο εκτέλεσης. Για αυτό το λόγο, χρειαζόμαστε έναν πολυπλέκτη ο οποίος επιλέγει κάθε φορά το αν η τιμή του PC θα προέρχεται από τον αθροιστή του σταδίου IF ή από τη λογική άλματος των επόμενων σταδίων της διοχέτευσης.

- Διεπαφή με την κρυφή μνήμη εντολών – Αφού έχει υπολογιστεί η διεύθυνση της προσκόμισης εντολής, προσκομίζεται η επιθυμητή εντολή από την κρυφή μνήμη εντολών.

Τα αποτελέσματα της επεξεργασίας του σταδίου IF, προωθούνται στο στάδιο ID.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### 3.7.2 3.7.2 Στάδιο ID

Το στάδιο ID είναι το στάδιο στο οποίο αποκωδικοποιείται η εντολή που προσκομίστηκε στο στάδιο IF. Συγκεκριμένα, εδώ εκτελούνται οι ακόλουθες λειτουργίες:

- Δημιουργούνται σήματα ελέγχου των υπολοίπων σταδίων.
- Πραγματοποιείται ανάγνωση δεδομένων από το αρχείο καταχωρητών.
- Επεκτείνεται το πεδίο immediate από τα 16 στα 32 bit.
- Προωθούνται οι κωδικοί των καταχωρητών Rs, Rt και Rd στα επόμενα στάδια.
- Υπολογίζεται η διεύθυνση άλματος στις εντολές άλματος.
- Πραγματοποιείται έλεγχος για data και control hazards.

Για την εκτέλεση των παραπάνω λειτουργιών, το στάδιο ID περιέχει τα παρακάτω τμήματα:

- Λογική αποκωδικοποίησης εντολής. Δημιουργεί τα σήματα ελέγχου βασισμένη στα πεδία opcode και func.
- Λογική σύγκρισης των δεδομένων που αναγνώστηκαν από το αρχείο καταχωρητών. Χρησιμοποιείται στις εντολές άλματος υπό συνθήκες.
- Λογική επέκτασης πρόσημου του πεδίου immediate. Συμπληρώνει σωστά τα 16 υψηλότερα bit της τιμής του πεδίου immediate κατά την επέκταση της από 16 σε 32 bit.
- Γραμμές προώθησης των τιμών των πεδίων Rs, Rt και Rd στα επόμενα στάδια. Χρησιμοποιούνται στο να προσδιορίζεται ο καταχωρητής προορισμού καθώς και στο να αντιμετωπίζονται κίνδυνοι ελέγχου και δεδομένων.
- Αρχείο καταχωρητών. Εδώ βρίσκονται οι 32 καταχωρητές γενικής χρήσης του επεξεργαστή. Σε αυτούς βρίσκονται τα δεδομένα πάνω στα οποία δρουν στην πλειοψηφία τους οι εντολές που εκτελούνται. Μόνη εξαίρεση αποτελεί η εντολή άλματος χωρίς όρους, “j”.
- Λογική ανίχνευσης κινδύνων ελέγχου και δεδομένων. Η συγκεκριμένη λογική, ανιχνεύει το αν τα δεδομένα εισόδου μίας εντολής τροποποιούνται από κάποια προηγούμενη εντολή η οποία δεν έχει φτάσει ακόμα στο στάδιο επανεγγραφής. Σε αυτή την περίπτωση, γίνεται προώθηση του αποτελέσματος και προστίθενται οι απαραίτητες καθυστερήσεις στα κατάλληλα στάδια της διοχέτευσης.
- Λογική υπολογισμού της διεύθυνσης άλματος. Αυτή η λογική υπολογίζει τη διεύθυνση της επόμενης εντολής στις εντολές άλματος και προωθεί τη διεύθυνση αυτή στο επόμενο στάδιο προκειμένου να εκτελεστεί το άλμα.

Τα αποτελέσματα της επεξεργασίας του σταδίου ID, προωθούνται στο στάδιο EX.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### 3.7.3 3.7.3 Στάδιο EX

Στο στάδιο εκτέλεσης πραγματοποιούνται οι περισσότερες λειτουργίες της κάθε εντολής. Εδώ εκτελούνται όλες οι αριθμητικές και λογικές πράξεις και σε αυτό το στάδιο φτάνουν οι διευθύνσεις άλματος προτού προωθηθούν στο στάδιο προσκόμισης εντολής. Στο στάδιο εκτέλεσης βρίσκονται οι παρακάτω μονάδες:

- Αριθμητική και λογική μονάδα (ALU). Η αριθμητική και λογική μονάδα εκτελεί όλες τις μαθηματικές και λογικές πράξεις της κάθε εντολής.
- Λογική ελέγχου της ALU. Η λογική ελέγχου της ALU έχει σαν είσοδο τα σήματα ελέγχου από το στάδιο ID καθώς και το πεδίο func όπου αυτό υπάρχει και παράγει τα κατάλληλα σήματα ελέγχου για την ALU. Ο σκοπός αυτής της μονάδας είναι να απλοποιείται η σχεδίαση της μονάδας αποκωδικοποίησης εντολής καθώς και να μειώνεται ο αριθμός των bit ελέγχου που πρέπει να περνάνε από το στάδιο ID στο στάδιο EX.

- Λογική άλματος. Η λογική άλματος αποτελείται από μία πύλη AND μεταξύ του bit ελέγχου που παράγεται στις εντολές άλματος και του bit σύγκρισης των δύο καταχωρητών που αναγνώστηκαν στο στάδιο ID.
- Κύκλωμα προώθησης της διεύθυνσης άλματος στο στάδιο IF.
- Λογική ανίχνευσης κινδύνων ελέγχου και δεδομένων. Η συγκεκριμένη λογική, ανιχνεύει το αν τα δεδομένα εισόδου μίας εντολής τροποποιούνται από κάποια προηγούμενη εντολή η οποία δεν έχει φτάσει ακόμα στο στάδιο επανεγγραφής. Σε αυτή την περίπτωση, γίνεται προώθηση του αποτελέσματος και προστίθενται οι απαραίτητες καθυστερήσεις στα κατάλληλα στάδια της διοχέτευσης.
- Πολυπλέκτης επιλογής του κωδικού του καταχωρητή προορισμού. Ο καταχωρητής προορισμού μπορεί να είναι είτε ο Rt, είτε ο Rd. Αναλόγως τα σήματα ελέγχου που δημιουργούνται στο στάδιο ID, επιλέγεται ένας από τους δύο και ο κωδικός του προωθείται στα επόμενα στάδια της διοχέτευσης.
- Πολυπλέκτες επιλογής της πηγής των δεδομένων εισόδου στην ALU. Τα δεδομένα εισόδου στην ALU μπορεί να προέρχονται είτε από αυτά που αναγνώστηκαν από το αρχείο καταχωρητών, είτε από το πεδίο immediate, είτε από δεδομένα που προωθήθηκαν από κάποιο άλλο στάδιο της διοχέτευσης. Υπάρχουν λοιπόν δύο πολυπλέκτες ελεγχόμενοι από τα bit ελέγχου που δημιουργούνται στο στάδιο ID καθώς και από την λογική εντοπισμού κινδύνων δεδομένων και ελέγχου οι οποίοι επιλέγουν κάθε φορά την σωστή πηγή των δεδομένων εισόδου.
- Κύκλωμα προώθησης της τιμής του δεύτερου καταχωρητή που αναγνώστηκε στο στάδιο ID προς το στάδιο MEM. Σε αυτόν τον καταχωρητή περιέχονται τα προς εγγραφή δεδομένα στις εντολές εγγραφής δεδομένων στη μνήμη συστήματος.

Τα αποτελέσματα της επεξεργασίας του σταδίου EX, προωθούνται στο στάδιο MEM.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### 3.7.4 Στάδιο MEM

Στο στάδιο προσπέλασης μνήμης πραγματοποιούνται όλες οι προσπελάσεις στη μνήμη δεδομένων για όσες εντολές το απαιτούν. Στο συγκεκριμένο στάδιο απαντώνται τα παρακάτω τμήματα:

- Κύκλωμα διοχέτευση του αποτελέσματος της ALU προς τη μνήμη συστήματος ως διεύθυνση εγγραφής ή ανάγνωσης.
- Κύκλωμα προώθησης του αποτελέσματος της ALU προς το στάδιο WB για όσες εντολές αποθηκεύουν το αποτέλεσμά τους σε κάποιον καταχωρητή.
- Κύκλωμα προώθησης του αποτελέσματος της ALU στην λογική προώθησης για την περίπτωση που έχουμε κινδύνους ελέγχου ή δεδομένων.
- Κύκλωμα προώθησης των δεδομένων προς εγγραφή στη μνήμη για όσες εντολές είναι απαραίτητο.
- Κύκλωμα προώθησης των δεδομένων που αναγνώστηκαν από τη μνήμη στο στάδιο WB για όσες εντολές χρειάζεται.
- Κύκλωμα προώθησης των κατάλληλων σημάτων ελέγχου προς τη μνήμη για τις λειτουργίες εγγραφής και ανάγνωσης.
- Κύκλωμα προώθησης του κωδικού του καταχωρητή προορισμού και των ανάλογων bit ελέγχου στο στάδιο WB.

Όπως βλέπουμε, το στάδιο προσπέλασης μνήμης αποτελείται από ελάχιστα λογικά κυκλώματα και περισσότερο από δρόμους προώθησης σημάτων. Είναι από τα απλούστερα στην υλοποίηση τμήματα της διοχέτευσης.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### 3.7.5 Στάδιο WB

Το στάδιο επανεγγραφής είναι το τελευταίο στάδιο της διοχέτευσης της αρχιτεκτονικής MIPS που εξετάζουμε. Λειτουργία του σταδίου WB είναι να προωθεί προς το αρχείο καταχωρητών τα

αποτελέσματα όσων εντολών απαιτούν εγγραφή δεδομένων προς κάποιον καταχωρητή. Αποτελείται από τα παρακάτω τμήματα:

- Ένας πολυπλέκτης ο οποίος επιλέγει αν τα δεδομένα προς εγγραφή προέρχονται από τη μνήμη του συστήματος ή από την ALU.
- Κύκλωμα προώθησης των δεδομένων προς εγγραφή καθώς και του κωδικού του καταχωρητή προορισμού και του bit ελέγχου της λειτουργίας της εγγραφής προς το αρχείο καταχωρητών.
- Προώθηση των δεδομένων προς εγγραφή καθώς και του κωδικού του καταχωρητή προορισμού προς την λογική προώθησης δεδομένων για την περίπτωση που έχουμε κίνδυνο ελέγχου ή δεδομένων.

## ~~3.8~~ 3.8 ΚΑΤΑΧΩΡΗΤΕΣ ΤΗΣ ΔΙΟΧΕΤΕΥΣΗΣ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Μεταξύ των διαφορετικών σταδίων της διοχέτευσης, παρεμβάλλονται κάποιοι ενδιάμεσοι καταχωρητές. Κάθε καταχωρητής, περιέχει τα δεδομένα που θα αποτελέσουν την είσοδο του επόμενου σταδίου στον επόμενο κύκλο ρολογιού.

### ~~3.8.1~~ 3.8.1 IF/ID (64 bits)

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ο καταχωρητής IF/ID παρεμβάλλεται μεταξύ των σταδίων προσκόμισης εντολής και αποκωδικοποίησης εντολής. Λόγω του γεγονότος πως όλα τα bit ελέγχου δημιουργούνται στο στάδιο αποκωδικοποίησης, αυτός ο καταχωρητής δεν περιέχει κανένα τέτοιο bit. Περιέχει όμως τα 32 bit της εντολής που προσκομίστηκε στο στάδιο προσκόμισης καθώς και τη διεύθυνση της εντολής που προσκομίσθηκε προσαυξημένη κατά 4. Η διεύθυνση αυτή χρησιμοποιείται στα μετέπειτα στάδια της διοχέτευσης προκειμένου να υπολογίζονται οι διευθύνσεις άλματος εφόσον ζητηθεί από κάποια εντολή άλματος. Επομένως, ο καταχωρητής IF/ID έχει συνολικό μήκος 32 + 32 = 64 bits.

| Next Program Counter | Instruction |
|---|---|
| 32 bits | 32 bits |

### ~~3.8.2~~ 3.8.2 ID/EX (157 bits)

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ο καταχωρητής ID/EX παρεμβάλλεται των σταδίων αποκωδικοποίησης και εκτέλεσης. Είναι ο μεγαλύτερος ενδιάμεσος καταχωρητής μιας και περιέχει όλα τα bit ελέγχου που προωθούνται στα επόμενα στάδια της διοχέτευσης, δεδομένα που διαβάζονται από το αρχείο καταχωρητών, το πεδίο immediate της εντολής καθώς και πληροφορία για τους καταχωρητές πηγής και προορισμού των δεδομένων για χρήση στην λογική εντοπισμού κινδύνου και προώθησης.

Συνολικά περιέχονται 7 bit ελέγχου του σταδίου εκτέλεσης, 4 bit ελέγχου του σταδίου προσπέλασης μνήμης, 2 bit ελέγχου του σταδίου ετερόχρονης εγγραφής, 1 bit ισότητας για τις εντολές άλματος, η διεύθυνση άλματος για τις εντολές άλματος, τα περιεχόμενα δύο καταχωρητών του αρχείου καταχωρητών, το πεδίο immediate με την κατάλληλη επέκταση ώστε να έχει τελικό μήκος 32 bits και τέλος οι κωδικές λέξεις των καταχωρητών Rs, Rt και Rd από τα αντίστοιχα πεδία της εντολής που αποκωδικοποιήθηκε. Στο σύνολό του αποτελείται από 157 bits.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 4 bits | 2 bits | 1 bit | 32 bits | 32 bits | 32 bits | 32 bits | 5 bits | 5 bits | 5 bits |

### ~~3.8.3~~ 3.8.3 EX/MEM (75 bits)

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ο καταχωρητής EX/MEM παρεμβάλλεται των σταδίων εκτέλεσης και προσπέλασης μνήμης. Περιλαμβάνει τα 4 bit ελέγχου του σταδίου μνήμης, 2 bit ελέγχου του σταδίου επανεγγραφής, το αποτέλεσμα της ALU (32 bits), τα δεδομένα που αναγνώστηκαν από τον καταχωρητή Rt (32 bits) καθώς

και τον κωδικό του καταχωρητή προορισμού (5 bits). Το συνολικό μήκος αυτού του καταχωρητή είναι 75 bit.

| M ctrl | WB ctrl | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|---|
| 4 bits | 2 bits | 32 bits | 32 bits | 5 bits |

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

### ~~3.8.4~~ 3.8.4 MEM/WB (71 bits)

Ο καταχωρητής MEM/WB είναι ο τελευταίος ενδιάμεσος καταχωρητής της διοχέτευσης της αρχιτεκτονικής μας. Παρεμβάλλεται των σταδίων προσπέλασης μνήμης και επανεγγραφής. Περιέχει τα 2 bit ελέγχου του σταδίου επανεγγραφής, τα δεδομένα που αναγνώστηκαν από τη μνήμη (32 bit), το προωθούμενο αποτέλεσμα της ALU (32 bit) και τον κωδικό του καταχωρητή προορισμού (5 bit). Στο σύνολό του αποτελείται από 71 bits.

| WB ctrl | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | 32 bits | 32 bits | 5 bits |

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## ~~3.9~~ 3.9 ΚΡΥΦΗ ΜΝΗΜΗ

Η κρυφή μνήμη δεδομένων και εντολών του μοντέλου μας έχουν τα εξής χαρακτηριστικά:

- Direct Mapped
- 4 bytes ανά block
- Index: 8 bits (256 blocks)
- Tag: 22 bits
- Byte Offset: 2 bits

Κάθε διεύθυνση χωρίζεται σε 3 πεδία, Tag, Index, Offset. Το πεδίο Index δείχνει το block της κρυφής μνήμης το οποίο πρέπει να περιέχει τα δεδομένα, το πεδίο Tag τότε συγκρίνεται με την τιμή που είναι αποθηκευμένη στο block που επιλέχτηκε. Αν οι τιμές τους είναι ίδιες και το bit “valid” είναι 1, τότε έχουμε μία επιτυχία στην κρυφή μνήμη (cache hit) και διαβάζονται τα δεδομένα ξεκινώντας από το byte που δείχνει το πεδίο offset. Ελέγχεται πάντα πως τα δεδομένα που ζητάμε είναι σωστά ευθυγραμμισμένα με τον παρακάτω τύπο:

Addr MOD Size

Όπου:

- Addr η διεύθυνση των δεδομένων.
- Size το μέγεθος των δεδομένων σε bytes.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## ~~3.10~~ 3.10 ΜΝΗΜΗ ΣΥΣΤΗΜΑΤΟΣ

Η μνήμη συστήματος που υλοποιούμε έχει την παρακάτω δομή:

- 0x00000000 - 0x07FFFFFF : Text Segment
- 0x08000000 - 0x17FFFFFF : Data Segment

Στο Text Segment περιέχονται οι εντολές του προγράμματος.

Στο Data Segment περιλαμβάνονται τα δεδομένα.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## ~~3.11~~ 3.11 ΚΩΔΙΚΟΙ ΕΝΤΟΛΩΝ

Όπως είδαμε, κάθε εντολή, ασχέτως του τύπου της, αναγνωρίζεται από τα πρώτα 6 bit της τα οποία αποτελούν το πεδίο opcode. Επιπλέον, οι εντολές τύπου R έχουν την ίδια τιμή στο πεδίο opcode και η επιθυμητή λειτουργία ορίζεται στο πεδίο func. Ακολουθεί συγκεντρωτικός πίνακας του βασικού συνόλου εντολών. Ο πίνακας περιέχει το μνημονικό όνομα της κάθε εντολής, τον τύπο της, την αγγλική φράση η

**Μορφοποιήθηκε:** Ελληνικά

οποία περιγράφει την λειτουργία της εντολής και από την οποία πηγάζει το αγγλικό μνημονικό όνομα της εντολής, την τιμή του πεδίου opcode και την τιμή του πεδίου func, όπου αυτό είναι παρόν.

| Όνομα | Τύπος | Λειτουργία | Opcode | | Func | |
|---|---|---|---|---|---|---|
| | | | hex | bin | hex | bin |
| add | R | Add | 00 | 00 0000 | 20 | 10 0000 |
| addi | I | Add Immediate | 08 | 00 1000 | - | - |
| addiu | I | Add Immediate Unsigned | 09 | 00 1001 | - | - |
| addu | R | Add Unsigned | 00 | 00 0000 | 21 | 10 0001 |
| and | R | And | 00 | 00 0000 | 24 | 10 0100 |
| andi | I | And Immediate | 0C | 00 1100 | - | - |
| beq | I | Branch On Equal | 04 | 00 0100 | - | - |
| bne | I | Branch On Not Equal | 05 | 00 0101 | - | - |
| j | J | Jump | 02 | 00 0010 | - | - |
| jal | J | Jump And Link | 03 | 00 0011 | - | - |
| jr | R | Jump Register | 00 | 00 0000 | 8 | 00 1000 |
| lbu | I | Load Byte Unsigned | 24 | 10 0100 | - | - |
| lhu | I | Load Halfword Unsigned | 25 | 10 0101 | - | - |
| lui | I | Load Upper Immediate | 0F | 00 1111 | - | - |
| lw | I | Load Word | 23 | 10 0011 | - | - |
| nor | R | Nor | 00 | 00 0000 | 27 | 10 0111 |
| or | R | Or | 00 | 00 0000 | 25 | 10 0101 |
| ori | I | Or Immediate | 0D | 00 1101 | - | - |
| slt | R | Set Less Than | 00 | 00 0000 | 2A | 10 1010 |
| slti | I | Set Less Than Immediate | 0A | 00 1010 | - | - |
| sltiu | I | Set Less Than Immediate Unsigned | 0B | 00 1011 | - | - |
| sltu | R | Set Less Than Unsigned | 00 | 00 0000 | 2B | 10 1011 |
| sll | R | Shift Left Logical | 00 | 00 0000 | 0 | 00 0000 |
| srl | R | Shift Right Logical | 00 | 00 0000 | 2 | 00 0010 |
| sb | I | Store Byte | 28 | 10 1000 | - | - |
| sh | I | Store Halfword | 29 | 10 1001 | - | - |
| sw | I | Store Word | 2B | 10 1011 | - | - |
| sub | R | Subtract | 00 | 00 0000 | 22 | 10 0010 |
| subu | R | Subtract Unsigned | 00 | 00 0000 | 23 | 10 0011 |

**Πίνακας 4: Κωδικοποίηση εντολών (opcodes).**

## ~~3.12~~ 3.12 ΒΙΤ ΕΛΕΓΧΟΥ ΤΗΣ ΔΙΟΧΕΤΕΥΣΗΣ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Στο στάδιο ID, παράγονται διάφορα σήματα ελέγχου από την λογική αποκωδικοποίησης. Σκοπός αυτών των σημάτων είναι να ελέγξουν τις λειτουργίες των διαφόρων μονάδων στα επόμενα στάδια προκειμένου να εκτελεστούν οι επιθυμητές λειτουργίες.

Ο κύριος λόγος για τον οποίο σε κάθε στάδιο προωθούμε αυτά τα σήματα ελέγχου αντί για το opcode της εντολής είναι πως με αυτό τον τρόπο, δεν χρειαζόμαστε ξεχωριστή λογική αποκωδικοποίησης σε κάθε στάδιο. Επίσης, ο έλεγχος στην αρχιτεκτονική MIPS είναι ενσωματωμένος στη διοχέτευση. Αυτό σημαίνει πως τα σήματα ελέγχου είναι στην ουσία συγκεκριμένα bits εντός των ενδιάμεσων καταχωρητών. Τα bit ελέγχου κάθε σταδίου προωθούνται μέχρι και το στάδιο για το οποίο έχουν ενδιαφέρον. Με αυτό τον τρόπο, μειώνουμε τα απαιτούμενα τρανζίστορ εντός των σταδίων αλλά και τον αριθμό bits που χρειάζεται κάθε ενδιάμεσος καταχωρητής.

Ακολουθεί παρουσίαση των bit ελέγχου του κάθε καταχωρητή.

### 3.12.1 IF

Το μόνο bit ελέγχου που απαντάται στο στάδιο IF είναι το bit που ελέγχει τον πολυπλέκτη που επιλέγει την τιμή του καταχωρητή PC. Πρέπει όμως να σημειωθεί πως το συγκεκριμένο bit δεν αποθηκεύεται σε κάποιον καταχωρητή και προέρχεται απευθείας από μία πύλη AND του σταδίου EX επομένως η τιμή του δεν μπορεί να διαταραχθεί από συγκρούσεις σωματιδίων για αρκετό χρονικό διάστημα προκειμένου να δημιουργηθεί κάποιο πρόβλημα στην εκτέλεση του προγράμματος.

### 3.12.2 ID

Το στάδιο ID, όπως και το στάδιο IF δεν χρειάζεται κάποιο control bit για τη λειτουργία του.

### 3.12.3 EX

Μία λειτουργία που εκτελείται στο στάδιο εκτέλεσης είναι να επιλέγεται και να προωθείται ο κωδικούς του καταχωρητή προορισμού προς το στάδιο προσπέλασης μνήμης. Αυτή η ανάγκη πηγάζει από το γεγονός πως κάποιες εντολές έχουν ως καταχωρητή προορισμού το πεδίο Rt της εντολής ενώ άλλες το πεδίο Rd. Προκειμένου να πραγματοποιηθεί αυτή η επιλογή, χρειαζόμαστε έναν πολυπλέκτη και ένα bit ελέγχου, το *RegDst*.

Στις εντολές άλματος, χρησιμοποιούνται το πεδίο *NPC* το οποίο περιέχει την διεύθυνση άλματος, το bit *equal* καθώς και το bit ελέγχου *branch*. Τα bit equal και branch αποτελούν τις εισόδους μίας πύλης AND της οποία η έξοδος προωθείται στον ένα και μοναδικό πολυπλέκτη του σταδίου προσκόμισης εντολής μαζί με τη διεύθυνση άλματος.

Βασικό κομμάτι της λογικής εκτέλεσης είναι η αριθμητική λογική μονάδα (Arithmetic Logic Unit, ALU). Η ALU είναι η υπεύθυνη για την εκτέλεση όλων των λογικών και αριθμητικών πράξεων που απαιτούνται από την κάθε εντολή. Η πρώτη είσοδος της ALU είναι σταθερή και δέχεται δεδομένα από τον καταχωρητή Rs είτε δεδομένα προωθημένα από κάποιο άλλο στάδιο στην περίπτωση που έχουμε κίνδυνο δεδομένων. Η δεύτερη είσοδος μπορεί να δέχεται δεδομένα είτε από τον καταχωρητή Rt είτε δεδομένα προωθημένα από κάποιο άλλο στάδιο στην περίπτωση που έχουμε κίνδυνο δεδομένων είτε από το πεδίο Immediate της εντολής. Την επιλογή την κάνουμε μέσω ενός πολυπλέκτη ο οποίος ελέγχεται από το bit *ALUSrc*.

Η λειτουργία που θα εκτελέσει η ALU καθορίζεται από τα παρακάτω bit ελέγχου, τα οποία όμως βρίσκονται εσωτερικά του σταδίου EX και δεν προωθούνται μέσω της διοχέτευσης.

- unsigned bit (1)
- negate B bit (1)
- invert A bit (1)
- function bits (add, and, or, shift, set less than, shift)
- shift left / right

Αυτά τα bit παράγονται από την λογική ελέγχου της ALU η οποία λαμβάνει υπόψη τόσο την τιμή του πεδίου func μίας εντολής όσο και τα bit ελέγχου *ALUOp* τα οποία προωθούνται από το στάδιο ID προς το στάδιο EX μέσω του ενδιάμεσου καταχωρητή ID/EX.

Στον παρακάτω πίνακα φαίνονται οι συνδυασμοί των bit που παράγονται από την λογική ελέγχου της ALU προκειμένου να εκτελεστούν διάφορες πράξεις:

| **Λειτουργία ALU** | **ALU control bits** | | | | |
|---|---|---|---|---|---|
| | **Unsigned** | **Negate B** | **Invert A** | **Function** | **Shift L/R** |
| Πρόσθεση | 0 | 0 | 0 | 000 | 0 |
| Πρόσθεση, χωρίς πρόσημο | 1 | 0 | 0 | 000 | 0 |
| Αφαίρεση | 0 | 1 | 0 | 000 | 0 |
| Αφαίρεση, χωρίς πρόσημο | 1 | 1 | 0 | 000 | 0 |
| AND | 1 | 0 | 0 | 001 | 0 |
| OR | 1 | 0 | 0 | 010 | 0 |
| NOR | 1 | 1 | 1 | 001 | 0 |
| Set less than | 0 | 0 | 0 | 011 | 0 |
| Set less than, χωρίς πρόσημο | 1 | 0 | 0 | 011 | 0 |
| Αριστερή ολίσθηση | 1 | 0 | 0 | 100 | 0 |
| Δεξιά ολίσθηση | 1 | 0 | 0 | 100 | 1 |

**Πίνακας 5: Bit ελέγχου ALU.**

Οι παραπάνω λειτουργίες δεν είναι οι μόνες που υποστηρίζονται στην αρχιτεκτονική MIPS. Είναι οι λειτουργίες που θα υποστηρίξουμε εμείς στην υλοποίησή μας. Στην πραγματικότητα περιλαμβάνονται και λειτουργίες κινητής υποδιαστολής αλλά και άλλες σχετικές με εξαιρέσεις καθώς και λειτουργίες εισόδου-εξόδου.

Όπως αναφέραμε προηγουμένως, η πράξη που θα πραγματοποιηθεί στην ALU ορίζεται στο πεδίο func των εντολών τύπου R. Όμως, στις εντολές τύπου I και J αυτό το πεδίο δεν υπάρχει. Προκειμένου να ορίζουμε σε αυτές τις εντολές τι πράξη πρέπει να γίνει, χρειαζόμαστε ξεχωριστά control bits, τα οποία ονομάζουμε ALUOp. Μία τιμή αυτών των bits θα σημαίνει πως ο κωδικός της πράξης βρίσκεται στο πεδίο func ενώ οι υπόλοιπες τιμές αυτού του πεδίου θα αντιστοιχούν σε συγκεκριμένες λειτουργίες.

Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά

| **Λειτουργία ALU** | ALUOp |
|---|---|
| Πρόσθεση | 0000 |
| Πρόσθεση, χωρίς πρόσημο | 0001 |
| Αφαίρεση | 0010 |
| AND | 0100 |
| OR | 0101 |
| Set Less Than | 0110 |
| Set less than, χωρίς πρόσημο | 0111 |
| Λειτουργία οριζόμενη από “func” | 1000 |

**Πίνακας 6: ALUOp bits.**

Ο παρακάτω πίνακας δείχνει τις εντολές που υλοποιούμε καθώς και τις τιμές που πρέπει να έχουν τα bit ελέγχου για κάθε μία από αυτές τις εντολές.

| Εντολή | Λειτουργία ALU | ALU control bits | EX control bits | | | |
|---|---|---|---|---|---|---|
| | | | ALUOp | ALUSrc | Branch | RegDst |
| add | Πρόσθεση | 000000000 | 1000 | 0 | 0 | 1 |
| addi | Πρόσθεση | 000000000 | 0000 | 1 | 0 | 0 |
| addiu | Πρόσθεση, χωρίς πρόσημο | 100000000 | 0001 | 1 | 0 | 0 |
| addu | Πρόσθεση, χωρίς πρόσημο | 100000000 | 1000 | 0 | 0 | 1 |
| and | AND | 100000010 | 1000 | 0 | 0 | 1 |
| andi | AND | 100000010 | 0100 | 1 | 0 | 0 |
| beq | - | 000100000 | xxxx | 0 | 1 | x |
| bne | - | 110100000 | xxxx | 0 | 1 | x |
| j | - | 001000000 | xxxx | 0 | 1 | x |
| jal | Πρόσθεση | 001000000 | 0000 | 0 | 1 | 0 |
| jr | - | | xxxx | 0 | 1 | 1 |
| lbu | Πρόσθεση | 000000000 | 0000 | 1 | 0 | 0 |
| lhu | Πρόσθεση | 000000000 | 0000 | 1 | 0 | 0 |
| lui | OR | 100000100 | 0101 | 1 | 0 | 0 |
| lw | Πρόσθεση | 000000000 | 0000 | 1 | 0 | 0 |
| nor | NOR | 100110010 | 1000 | 0 | 0 | 1 |
| or | OR | 100000100 | 1000 | 0 | 0 | 1 |
| ori | OR | 100000100 | 0101 | 1 | 0 | 0 |
| slt | Set less than | 000000110 | 1000 | 0 | 0 | 1 |
| slti | Set less than | 000000110 | 0110 | 1 | 0 | 0 |
| sltiu | Set less than, χωρίς πρόσημο | 100000110 | 0111 | 1 | 0 | 0 |
| sltu | Set less than, χωρίς πρόσημο | 100000110 | 1000 | 0 | 0 | 1 |
| sll | Αριστερή μετατόπιση | 100001000 | 1000 | 0 | 0 | 1 |
| srl | Δεξιά μετατόπιση | 100001001 | 1000 | 0 | 0 | 1 |
| sb | Πρόσθεση | 000000000 | 0000 | 1 | 0 | x |
| sh | Πρόσθεση | 000000000 | 0000 | 1 | 0 | x |
| sw | Πρόσθεση | 000000000 | 0000 | 1 | 0 | x |
| sub | Αφαίρεση | 000100000 | 1000 | 0 | 0 | 1 |
| subu | Αφαίρεση, χωρίς πρόσημο | 100100000 | 1000 | 0 | 0 | 1 |

**Πίνακας 7: Bit ελέγχου σταδίου εκτέλεσης.**

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

### ~~3.12.4~~ 3.12.4 MEM

Στο στάδιο προσπέλασης μνήμης, εκτελούνται λειτουργίες εγγραφής και ανάγνωσης από και προς την κύρια μνήμη. Επίσης, τα δεδομένα από το στάδιο εκτέλεσης προωθούνται και στο στάδιο WB. Αυτό είναι απαραίτητο για τις εντολές που δεν γράφουν στην μνήμη αλλά σε κάποιον καταχωρητή.

Χρειάζονται δύο bit ελέγχου τα οποία ορίζουν την λειτουργία της μονάδας μνήμης. Το bit *Mem-Read* ορίζει αν θα έχουμε προσπέλαση στη μνήμη για ανάγνωση και το *mem-write* χρησιμοποιείται για λειτουργία εγγραφής. Χρειαζόμαστε δύο bit μιας και υπάρχει το ενδεχόμενο να μην πραγματοποιηθεί καμία προσπέλαση στη μνήμη. Τέλος, χρειαζόμαστε κάποια bit τα οποία ορίζουν το μέγεθος των δεδομένων που θα εγγραφούν ή θα αναγνωστούν από τη μνήμη. Τα πιθανά μεγέθη είναι:

- Byte (8 bits)
- Halfword (16 bits)
- Word (32 bits)

Υπάρχει και το μέγεθος "Double Word (64 bit)" αλλά δεν το χρησιμοποιούμε στην υλοποίησή μας. Για την κωδικοποίηση των παραπάνω περιπτώσεων, είναι αρκετά 2 bits ελέγχου (Data Size bits).

Συνοψίζοντας, τα bit ελέγχου του σταδίου προσπέλασης μνήμης είναι τα παρακάτω:

Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά
Μορφοποιήθηκε: Ελληνικά

| **Εντολή** | **MEM control bits** | | |
|---|---|---|---|
| | **Mem-Read** | **Mem-Write** | **Data Size** |
| add | 0 | 0 | xx |
| addi | 0 | 0 | xx |
| addiu | 0 | 0 | xx |
| addu | 0 | 0 | xx |
| and | 0 | 0 | xx |
| andi | 0 | 0 | xx |
| beq | 0 | 0 | xx |
| bne | 0 | 0 | xx |
| j | 0 | 0 | xx |
| jal | 0 | 0 | xx |
| jr | 0 | 0 | xx |
| lbu | 1 | 0 | 00 |
| lhu | 1 | 0 | 01 |
| lui | 0 | 0 | xx |
| lw | 1 | 0 | 10 |
| nor | 0 | 0 | xx |
| or | 0 | 0 | xx |
| ori | 0 | 0 | xx |
| slt | 0 | 0 | xx |
| slti | 0 | 0 | xx |
| sltiu | 0 | 0 | xx |
| sltu | 0 | 0 | xx |
| sll | 0 | 0 | xx |
| srl | 0 | 0 | xx |
| sb | 0 | 1 | 00 |
| sh | 0 | 1 | 01 |
| sw | 0 | 1 | 10 |
| sub | 0 | 0 | xx |

| Εντολή | MEM control bits | | |
|---|---|---|---|
| | Mem-Read | Mem-Write | Data Size |
| subu | 0 | 0 | xx |

**Πίνακας 8: Bit ελέγχου σταδίου προσπέλασης μνήμης.**

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### 3.12.5 3.12.5 WB

Στο στάδιο Write Back φτάνουν τα δεδομένα που αποτελούν το αποτέλεσμα μίας εντολής και προωθούνται προς τον κατάλληλο καταχωρητή. Τα δεδομένα αυτά είτε προέρχονται από μία αριθμητική ή λογική πράξη από την αριθμητική και λογική μονάδα, είτε είναι δεδομένα τα οποία αναγνώστηκαν από την μνήμη συστήματος.

Αυτό το στάδιο περιλαμβάνει έναν πολυπλέκτη ο οποίος επιλέγει την πηγή των δεδομένων. Αυτή μπορεί να είναι είτε η μνήμη συστήματος, είτε το αποτέλεσμα της αριθμητικής λογικής μονάδας. Ο συγκεκριμένος πολυπλέκτης ελέγχεται από το σήμα ελέγχου *MemToReg*.

Επίσης, από αυτό το στάδιο προωθείται στο αρχείο καταχωρητών ο κωδικός του καταχωρητή προορισμού καθώς και το σήμα *RegWrite* το οποίο σηματοδοτεί λειτουργία εγγραφής των δεδομένων στον καταχωρητή που έχει επιλεγεί ως καταχωρητής προορισμού.

Συνοψίζοντας, τα bit ελέγχου του σταδίου προσπέλασης μνήμης είναι τα παρακάτω:

| Εντολή | WB control bits | |
|---|---|---|
| | MemToReg | RegWrite |
| add | 0 | 1 |
| addi | 0 | 1 |
| addiu | 0 | 1 |
| addu | 0 | 1 |
| and | 0 | 1 |
| andi | 0 | 1 |
| beq | x | 0 |
| bne | x | 0 |
| j | x | 0 |
| jal | x | 0 |
| jr | x | 0 |
| lbu | 1 | 1 |
| lhu | 1 | 1 |
| lui | 0 | 1 |
| lw | 1 | 1 |
| nor | 0 | 1 |
| or | 0 | 1 |
| ori | 0 | 1 |
| slt | 0 | 1 |
| slti | 0 | 1 |
| sltiu | 0 | 1 |

| Εντολή | WB control bits | |
|---|---|---|
| | MemToReg | RegWrite |
| sltu | 0 | 1 |
| sll | 0 | 1 |
| srl | 0 | 1 |
| sb | x | 0 |
| sh | x | 0 |
| sw | x | 0 |
| sub | 0 | 1 |
| subu | 0 | 1 |

**Πίνακας 9: Bit ελέγχου σταδίου επανεγγραφής.**

## ~~3.13~~ 3.13 ΣΥΝΟΨΗ BIT ΕΛΕΓΧΟΥ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Τα bit ελέγχου για την κάθε εντολή συνοψίζονται στον παρακάτω πίνακα:

| Εντολή | NPCSrc | EX control bits | | | | M control bits | | | WB control bits | | Hex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ALU Op | ALU Src | Branch | RegDst | Mem-Read | Mem-Write | Data Size | Mem ToReg | Reg Write | |
| add | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| addi | xx | 0000 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0101 |
| addiu | xx | 0001 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0301 |
| addu | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| and | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| andi | 00 | 0100 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0901 |
| beq | 00 | 0010 | 0 | 1 | x | 0 | 0 | xx | x | 0 | 0480 |
| bne | 01 | 0011 | 0 | 1 | x | 0 | 0 | xx | x | 0 | 0680 |
| j | 01 | 1111 | 0 | 1 | x | 0 | 0 | xx | x | 0 | 1e80 |
| jal | 11 | 1111 | 0 | 1 | 0 | 0 | 0 | xx | x | 1 | 1e81 |
| jr | xx | 1000 | 0 | 1 | 1 | 0 | 0 | xx | x | 0 | 10c0 |
| lbu | xx | 0000 | 1 | 0 | 0 | 1 | 0 | 00 | 1 | 1 | 0123 |
| lhu | xx | 0000 | 1 | 0 | 0 | 1 | 0 | 01 | 1 | 1 | 0127 |
| lui | xx | 0101 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0b01 |
| lw | xx | 0000 | 1 | 0 | 0 | 1 | 0 | 10 | 1 | 1 | 012b |
| nor | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 4041 |
| or | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| ori | xx | 0101 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0b01 |
| slt | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| slti | xx | 0110 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0d01 |

| Εντολή | NPCSrc | EX control bits | | | | M control bits | | | WB control bits | | Hex |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ALU Op | ALU Src | Branch | RegDst | Mem-Read | Mem-Write | Data Size | Mem ToReg | Reg Write | |
| sltiu | xx | 0111 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 0f01 |
| sltu | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| sll | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| srl | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| sb | xx | 0000 | 1 | 0 | x | 0 | 1 | 00 | x | 0 | 0110 |
| sh | xx | 0000 | 1 | 0 | x | 0 | 1 | 01 | x | 0 | 0114 |
| sw | xx | 0000 | 1 | 0 | x | 0 | 1 | 10 | x | 0 | 0118 |
| sub | xx | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |
| subu | | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 1041 |

**Πίνακας 10: Bit ελέγχου της διοχέτευσης.**

## ~~3.14~~ 3.14 ACE BITS

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Επόμενο βήμα είναι να αναλύσουμε και να βρούμε ποια είναι τα κρίσιμα bits για κάθε εντολή σε κάθε στάδιο της διοχέτευσης. Οι ενδιάμεσοι καταχωρητές παρουσιάστηκαν στην παράγραφο 4.4. Παρακάτω παραθέτονται σε μορφή κατάλληλη για επεξεργασία κατά την ανάλυση ACE:

***IF/ID (64 bits)***

| Next Program Counter | Instruction |
|---|---|
| 32 bits | 32 bits |

***ID/EX (157 bits)***

| Control bits | NPC | Eq | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|
| 13 bits | 32 bits | 1 bit | 32 bits | 32 bits | 32 bits | 5 bist | 5 bits | 5 bits |

***EX/MEM (75 bits)***

| Control bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 6 bits | 32 bits | 32 bits | 5 bits |

***MEM/WB (71 bits)***

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | 32 bits | 32 bits | 5 bits |

### ~~3.14.1~~ 3.14.1 ACE bits Ελέγχου

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ξεκινάμε την ανάλυση με το να υπολογίσουμε τα ACE bits όσον αφορά τα bit ελέγχου. Στις προηγούμενες ενότητες παρουσιάσαμε κάθε εντολή καθώς και τα bit ελέγχου που αυτή χρειάζεται σε

κάθε στάδιο της διοχέτευσης. Στον παρακάτω πίνακα συνοψίζουμε αυτή την ανάλυση ώστε να είναι ευκολότερα προσβάσιμη από τον αναγνώστη αλλά και για να μας βοηθήσει στα επόμενα βήματα της ανάλυσής μας.

| Εντολή | EX control bits | | | | M control bits | | | WB control bits | | ACE bits |
|---|---|---|---|---|---|---|---|---|---|---|
| | ALUOp | ALU Src | Branch | Reg Dst | Mem-Read | Mem-Write | Data Size | MemToReg | Reg Write | |
| add | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| addi | 0000 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| addiu | 0001 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| addu | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| and | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| andi | 0100 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| beq | xxxx | 0 | 1 | x | 0 | 0 | xx | x | 0 | 5 |
| bne | xxxx | 0 | 1 | x | 0 | 0 | xx | x | 0 | 5 |
| j | xxxx | 0 | 1 | x | 0 | 0 | xx | x | 0 | 5 |
| jal | 1111 | 0 | 1 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| jr | xxxx | 0 | 1 | 1 | 0 | 0 | xx | x | 0 | 6 |
| lbu | 0000 | 1 | 0 | 0 | 1 | 0 | 00 | 1 | 1 | 13 |
| lhu | 0000 | 1 | 0 | 0 | 1 | 0 | 01 | 1 | 1 | 13 |
| lui | 0101 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| lw | 0000 | 1 | 0 | 0 | 1 | 0 | 10 | 1 | 1 | 13 |
| nor | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| or | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| ori | 0101 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| slt | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| slti | 0110 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| sltiu | 0111 | 1 | 0 | 0 | 0 | 0 | xx | 0 | 1 | 11 |
| sltu | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| sll | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| srl | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| sb | 0000 | 1 | 0 | x | 0 | 1 | 00 | x | 0 | 11 |
| sh | 0000 | 1 | 0 | x | 0 | 1 | 01 | x | 0 | 11 |
| sw | 0000 | 1 | 0 | x | 0 | 1 | 10 | x | 0 | 11 |
| sub | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |
| subu | 1000 | 0 | 0 | 1 | 0 | 0 | xx | 0 | 1 | 11 |

**Πίνακας 11: ACE bits ελέγχου.**

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### 3.14.2 3.14.2 IF/ID ACE bits

Ο καταχωρητής IF/ID περιέχει την εντολή που προσκομίστηκε από τη μνήμη στο στάδιο IF. Επίσης περιέχει και την διεύθυνση της επόμενης εντολής η οποία χρησιμοποιείται στις εντολές άλματος. Συνολικά αποτελείται από 64 bit.

| Next Program Counter | Instruction |
|---|---|
| 32 bits | 32 bits |

***add ( 28 ACE bits )***

Η εντολή add είναι τύπου R. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Το μόνο πεδίο που δεν μας ενδιαφέρει είναι το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης. Επίσης, δεν μας ενδιαφέρει η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***addi ( 32 ACE bits )***

Η εντολή addi είναι τύπου I. Μας ενδιαφέρουν τα πεδία opcode, Rs,Rt και immediate. Δεν μας ενδιαφέρει η διεύθυνση επόμενης εντολής.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***addiu ( 32 ACE bits )***

Η εντολή addiu είναι ίδια με την addi όσων αφορά τα πεδία που είναι κρίσιμα. Μας ενδιαφέρουν τα πεδία opcode, Rs,Rt και immediate. Δεν μας ενδιαφέρει η διεύθυνση επόμενης εντολής.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***addu ( 28 ACE bits )***

Η εντολή addu είναι παρόμοια με την add. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Δεν μας ενδιαφέρουν το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης καθώς και η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***and ( 28 ACE bits )***

Η εντολή and είναι τύπου R. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Δεν μας ενδιαφέρει το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης. Επίσης, δεν μας ενδιαφέρει η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***andi ( 32 ACE bits )***

Η εντολή andi εκτελεί την ίδια πράξη με την εντολή and με τη διαφορά πως είναι εντολή τύπου I και επομένως ο δεύτερος τελεστέος περιέχεται στα 16 τελευταία bit της εντολής. Μας ενδιαφέρουν τα πεδία opcode, Rs,Rt και immediate. Δεν μας ενδιαφέρει η διεύθυνση επόμενης εντολής.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***beq ( 64 ACE bits )***

Με την εντολή beq εκτελείται άλμα σε μία διεύθυνση εφόσον τα περιεχόμενα δύο καταχωρητών που συγκρίνονται είναι ίσα. Η διεύθυνση υπολογίζεται από το πεδίο immediate και την διεύθυνση της επόμενης εντολής. Επίσης οι δύο καταχωρητές που πρέπει να συγκριθούν ορίζονται στα πεδία Rt και Rd. Επομένως, και τα 64 bit του καταχωρητή IF/ID είναι κρίσιμα.

| Next Program Counter | Instruction |
|---|---|
| 32 bits | 32 bits |

***bne ( 64 ACE bits )***

Με την εντολή bne εκτελείτε άλμα σε μία διεύθυνση εφόσον τα περιεχόμενα δύο καταχωρητών που συγκρίνονται δεν είναι ίσα. Η διεύθυνση υπολογίζεται από το πεδίο immediate και την διεύθυνση της επόμενης εντολής. Επίσης οι δύο καταχωρητές που πρέπει να συγκριθούν ορίζονται στα πεδία Rt και Rd. Επομένως, και τα 64 bit του καταχωρητή IF/ID είναι κρίσιμα.

| Next Program Counter | Instruction |
|---|---|
| 32 bits | 32 bits |

***j ( 36 ACE bits )***

Η εντολή j εκτελεί άλμα χωρίς όρους σε διεύθυνση που προκύπτει από τα 4 bit της διεύθυνσης της επόμενης εντολής καθώς και τα 26bit του πεδίου address της εντολής. Επομένως τα κρίσιμα bit για την εντολή j είναι 36bits.

| Next Program Counter | Instruction |
|---|---|
| 4 bits | 32 bits |

***jal ( 36 ACE bits )***

Η εντολή jal εκτελεί άλμα με τον ίδιο τρόπο με την εντολή j με τη διαφορά πως αποθηκεύει την διεύθυνση της επόμενης εντολής στον καταχωρητή $31. Επομένως τα bit που είναι κρίσιμα είναι ίδια με αυτά της εντολής j.

| Next Program Counter | Instruction |
|---|---|
| 4 bits | 32 bits |

***jr ( 11 ACE bits )***

Η εντολή jr είναι μία εντολή τύπου j και εκτελεί άλμα στην διεύθυνση που περιέχεται στον καταχωρητή που ορίζετε στο πεδίο Rs. Τα κρίσιμα bit είναι το πεδίο opcode καθώς και το πεδίο Rs, στο σύνολο τους 11.

| Next Program Counter | Instruction |
|---|---|
| - | 11 bits |

***lbu ( 32 ACE bits )***

Η εντολή lbu διαβάζει από τη μνήμη ένα byte. Η διεύθυνση προκύπτει από τα περιεχόμενα του καταχωρητή Rs και του πεδίου immediate. Το byte αποθηκεύεται στον καταχωρητή Rt. Επομένως, όλα τα bits του πεδίου της εντολής είναι κρίσιμα.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***lhu ( 32 ACE bits )***

Η εντολή lhu φορτώνει από τη μνήμη 16bits. Η διεύθυνση προσπέλασης της μνήμης προκύπτει από τα περιεχόμενα του καταχωρητή Rs και του πεδίου immediate. Το byte αποθηκεύεται στον καταχωρητή Rt. Επομένως, όλα τα bits του πεδίου της εντολής είναι κρίσιμα.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***lui ( 28 ACE bits )***

Η εντολή lui μεταφέρει τα 16 bit του πεδίου immediate στα υψηλότερα 16 bit του καταχωρητή Rt. Ο καταχωρητής Rs δεν χρησιμοποιείται. Επομένως τα κρίσιμα bit είναι τα opcode bits (6), τα bit του καταχωρητή Rt (5) και τα 16 bit του πεδίου immediate. Σύνολο 28 bits.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***lw ( 32 ACE bits )***

Η εντολή lhu φορτώνει από τη μνήμη μία λέξη των 32 bit.. Η διεύθυνση προσπέλασης της μνήμης προκύπτει από τα περιεχόμενα του καταχωρητή Rs και του πεδίου immediate. Το byte αποθηκεύεται στον καταχωρητή Rt. Επομένως, όλα τα bits του πεδίου της εντολής είναι κρίσιμα.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***nor ( 28 ACE bits )***

Η εντολή nor είναι τύπου R. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Δεν μας ενδιαφέρει το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης. Επίσης, δεν μας ενδιαφέρει η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***or ( 28 ACE bits )***

Η εντολή or είναι τύπου R. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Δεν μας ενδιαφέρει το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης. Επίσης, δεν μας ενδιαφέρει η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***ori ( 32 ACE bits )***

Η εντολή ori εκτελεί την ίδια πράξη με την εντολή or με τη διαφορά πως είναι εντολή τύπου I και επομένως ο δεύτερος τελεστέος περιέχεται στα 16 τελευταία bit της εντολής. Μας ενδιαφέρουν τα πεδία opcode, Rs,Rt και immediate. Δεν μας ενδιαφέρει η διεύθυνση επόμενης εντολής.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***slt ( 28 ACE bits )***

Η εντολή slt θέτει την τιμή 1 στον καταχωρητή Rd, εφόσον τα περιεχόμενα των καταχωρητών Rs και Rt είναι ίσα. Ως εντολή τύπου R, μας ενδιαφέρει και το πεδίο func, αλλά όχι το πεδίο shamt.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***slti ( 32 ACE bits )***

Η εντολή slti θέτει την τιμή 1 στον καταχωρητή Rt, εφόσον τα περιεχόμενα του καταχωρητή Rs και του πεδίου immediate είναι ίσα. Επομένως μας ενδιαφέρουν και τα 32 bit της εντολής.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***sltiu ( 32 ACE bits )***

Η εντολή sltiu είναι πανομοιότυπη με την εντολή slti με τη διαφορά πως οι αριθμοί θεωρούνται πως είναι μη προσημασμένοι.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***sltu ( 28 ACE bits )***

Η εντολή sltu είναι πανομοιότυπη με την εντολή slt με τη διαφορά πως οι αριθμοί θεωρούνται πως είναι μη προσημασμένοι.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***sll ( 28 ACE bits )***

Η εντολή sll πραγματοποιεί αριστερή ολίσθηση των περιεχομένων του καταχωρητή Rt κατά τόσα bit όσα ορίζονται στο πεδίο shamt και το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. Το πεδίο Rs δεν χρησιμοποιείται.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***srl ( 28 ACE bits )***

Η εντολή sll πραγματοποιεί δεξιά ολίσθηση των περιεχομένων του καταχωρητή Rt κατά τόσα bit όσα ορίζονται στο πεδίο shamt και το αποτέλεσμα αποθηκεύεται στον καταχωρητή Rd. Το πεδίο Rs δεν χρησιμοποιείται.

| Next Program Counter | Instruction |
|---|---|

| | |
|---|---|
| - | 28 bits |

***sb ( 32 ACE bits )***

Η εντολή sb αποθηκεύει ένα byte στη μνήμη. Η διεύθυνση υπολογίζεται από τα περιεχόμενα του καταχωρητή Rs και το πεδίο immediate. Τα δεδομένα προς αποθήκευση περιέχονται στον καταχωρητή Rt.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***sh ( 32 ACE bits )***

Η εντολή sh αποθηκεύει δύο bytes στη μνήμη. Η διεύθυνση υπολογίζεται από τα περιεχόμενα του καταχωρητή Rs και το πεδίο immediate. Τα δεδομένα προς αποθήκευση περιέχονται στον καταχωρητή Rt.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***sw ( 32 ACE bits )***

Η εντολή sw αποθηκεύει μία λέξη των 32 bit στη μνήμη. Η διεύθυνση υπολογίζεται από τα περιεχόμενα του καταχωρητή Rs και το πεδίο immediate. Τα δεδομένα προς αποθήκευση περιέχονται στον καταχωρητή Rt.

| Next Program Counter | Instruction |
|---|---|
| - | 32 bits |

***sub ( 28 ACE bits )***

Η εντολή sub είναι τύπου R. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Το μόνο πεδίο που δεν μας ενδιαφέρει είναι το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης. Επίσης, δεν μας ενδιαφέρει η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

***subu ( 28 ACE bits )***

Η εντολή subu είναι παρόμοια με την εντολή sub, με τη διαφορά πως τα δεδομένα είναι μη προσημασμένοι αριθμοί. Μας ενδιαφέρουν τα πεδία opcode, Rs, Rt, Rd και func. Το μόνο πεδίο που δεν μας ενδιαφέρει είναι το shamt το οποίο χρησιμοποιείται μόνο στις εντολές ολίσθησης. Επίσης, δεν μας ενδιαφέρει η διεύθυνση της επόμενης εντολής μιας και χρησιμοποιείται μόνο στις εντολές άλματος.

| Next Program Counter | Instruction |
|---|---|
| - | 28 bits |

Οι παραπάνω παρατηρήσεις συνοψίζονται στον ακόλουθο πίνακα:

| **Εντολή** | **Πεδίο του καταχωρητή** | | **ACE bits** |
|---|---|---|---|
| | **NPC** | **Instruction** | |
| add | - | 28 bits | 28 bits |

| Εντολή | Πεδίο του καταχωρητή | | ACE bits |
|---|---|---|---|
| | NPC | Instruction | |
| addi | - | 32 bits | 32 bits |
| addiu | - | 32 bits | 32 bits |
| addu | - | 28 bits | 28 bits |
| and | - | 28 bits | 28 bits |
| andi | - | 32 bits | 32 bits |
| beq | 32 bits | 32 bits | 64 bits |
| bne | 32 bits | 32 bits | 64 bits |
| j | 4 bits | 32 bits | 36 bits |
| jal | 4 bits | 32 bits | 36 bits |
| jr | - | 11 bits | 11 bits |
| lbu | - | 32 bits | 32 bits |
| lhu | - | 32 bits | 32 bits |
| lui | - | 28 bits | 28 bits |
| lw | - | 32 bits | 32 bits |
| nor | - | 28 bits | 28 bits |
| or | - | 28 bits | 28 bits |
| ori | - | 32 bits | 32 bits |
| slt | - | 28 bits | 28 bits |
| slti | - | 32 bits | 32 bits |
| sltiu | - | 32 bits | 32 bits |
| sltu | - | 28 bits | 28 bits |
| sll | - | 28 bits | 28 bits |
| srl | - | 28 bits | 28 bits |
| sb | - | 32 bits | 32 bits |
| sh | - | 32 bits | 32 bits |
| sw | - | 32 bits | 32 bits |
| Sub | - | 28 bits | 28 bits |
| subu | - | 28 bits | 28 bits |
| nop[1] | - | 17 bits | 17 bits |

**Πίνακας 12: ACE bits καταχωρητή IF/ID**

Μορφοποιημένο: Κουκκίδες και αρίθμηση

### ~~3.14.3~~ 3.14.3 ID/EX ACE bits

Ο καταχωρητής ID/EX περιέχει τα bit ελέγχου για τα στάδια EX, MEM και WB. Επίσης περιέχει την τιμή της επόμενης εντολής το οποίο χρησιμοποιείται σε μερικές εντολές άλματος. Ακολουθούν οι δύο

[1] Η εντολή nop στον mips είναι η εντολή sll 0, 0, 0 αλλά την αναφαίρουμε ξεχωριστά μιας και δεν πραγματοποιείτε καμία εγγραφή στο αρχείο καταχωρητών οπότε τα ACE bits της είναι διαφορετικά από αυτά μίας τυπικής sll.

τιμές που έχουν αναγνωσθεί από το αρχείο καταχωρητών, το πεδίο immediate το οποίο εξυπηρετεί διάφορους σκοπούς καθώς και οι τιμές των καταχωρητών Rt και Rd. Στο σύνολό του, αποτελείται από 158 bits. Ο παρακάτω πίνακας, απεικονίζει τα πεδία του καταχωρητή καθώς και τον αριθμό των bits του κάθε πεδίου. Ακολουθούν οι υποστηριζόμενες εντολές και σε κάθε μία από αυτές, αναλύεται ποια bits είναι κρίσιμα για την ορθή εκτέλεση της εντολής ( ACE bits ).

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 4 bits | 2 bits | 1 bit | 32 bits | 32 bits | 32 bits | 32 bits | 5 bits | 5 bits | 5 bits |

***add ( 86 ACE bits )***

Το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Η συγκεκριμένη εντολή προσθέτει δύο αριθμούς οι οποίοι περιέχονται στα πεδία Data Read 1 και 2. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***addi ( 80 ACE bits )***

Η εντολή addi προσθέτει δύο αριθμούς. Ο πρώτος περιέχεται σε έναν καταχωρητή και ο δεύτερος βρίσκεται στο πεδίο immediate. Το πεδίο NPC, Rd καθώς και το Data Read 2 δεν μας ενδιαφέρουν.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***addiu ( 80 ACE bits )***

Η εντολή addiu είναι παρόμοια με την addi. Οι δύο τελεστές περιέχονται σε έναν καταχωρητή και στο πεδίο immediate. Το πεδίο NPC, Rd καθώς και το Data Read 2 δεν μας ενδιαφέρουν.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***addu ( 86 ACE bits )***

Η εντολή addu είναι παρόμοια με την add. Συγκεκριμένα, το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Η εντολή προσθέτει δύο αριθμούς οι οποίοι περιέχονται στα πεδία Data Read 1 και 2. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***and ( 86 ACE bits )***

Η εντολή and ανήκει στις εντολές τύπου R. Πραγματοποιεί την λογική πράξη AND μεταξύ των δύο τελεστών οι οποίοι περιέχονται στα πεδία Data Read 1 και 2. Το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***andi ( 80 ACE bits )***

Η εντολή andi εκτελεί την λογική πράξη AND μεταξύ δύο αριθμών. Ο πρώτος περιέχεται σε έναν καταχωρητή και ο δεύτερος βρίσκεται στο πεδίο immediate. Το πεδίο NPC, Rd καθώς και το Data Read 2 δεν μας ενδιαφέρουν.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***beq ( 139 ACE bits )***

Η εντολή beq εκτελεί ένα άλμα προς την υπολογιζόμενη διεύθυνση άλματος εφόσον οι τιμές των καταχωρητών Rs και Rt, οι οποίες αποθηκεύονται στα πεδία Data Read 1 και 2 είναι ίδιες. Η διεύθυνση άλματος προκύπτει από το πεδίο NPC και το πεδίο immediate.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 bits | 2 bits | 1 bits | 1 bit | 32 bits | - | - | 32 bits | 5 bits | 5 bits | - |

***bne ( 139 ACE bits )***

Η εντολή bne εκτελεί ένα άλμα προς την υπολογιζόμενη διεύθυνση άλματος εφόσον οι τιμές που αποθηκεύονται στα πεδία Data Read 1 και 2 δεν είναι ίδιες. Η διεύθυνση άλματος προκύπτει από το πεδίο NPC και το πεδίο immediate.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 bits | 2 bits | 1 bits | 1 bit | 32 bits | - | - | 32 bits | 5 bits | 5 bits | - |

***j ( 47 ACE bits )***

Η εντολή j εκτελεί ένα άλμα προς την υπολογιζόμενη διεύθυνση άλματος. Η διεύθυνση άλματος προκύπτει από τα 4 σημαντικότερα bits του πεδίου NPC και το πεδίο address, το οποίο αποθηκεύεται στο πεδίο immediate.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 bits | 2 bits | 1 bits | 1 bit | 32 bits | - | - | 32 bits | - | - | - |

***jal ( 117 ACE bits )***

Η εντολή j εκτελεί ένα άλμα προς την υπολογιζόμενη διεύθυνση άλματος και αποθηκεύει την διεύθυνση της μεθεπόμενης εντολής στον καταχωρητή $31. Η διεύθυνση άλματος προκύπτει από τα 4 σημαντικότερα bits του πεδίου NPC και το πεδίο address, το οποίο αποθηκεύεται στο πεδίο immediate. Η διεύθυνση άλματος προκύπτει από την λογική που χρησιμοποιείται αποκλειστικά για τις εντολές άλματος. Η διεύθυνση της μεθεπόμενης εντολής προκύπτει από την ALU με το να γράφεται το περιεχόμενο του NPC στο πεδίο Data Read 1 και ο αριθμός 4 στο πεδίο Data Read 2. Ο κωδικός του καταχωρητή προορισμού ( 31 ) αποθηκεύεται στο πεδίο Rt.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | 1 bit | 32 bits | 32 bits | 32 bits | 32 bits | 5 bits | 5 bits | - |

***jr ( 49 ACE bits )***

Η εντολή Jr εκτελεί ένα άλμα στην διεύθυνση που δείχνει ο καταχωρητής Rs. Το πεδίο NPC καθώς και τα Rt και Rd δεν μας ενδιαφέρουν. Η τιμή άλματος περιέχεται στην τιμή Read Data 1. Η τιμή του Read Data 2 δεν μας ενδιαφέρει. Μας ενδιαφέρουν τα 6 τελευταία bit του πεδίου immediate μιας και καθορίζουν τη λειτουργία της ALU.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 bits | 2 bits | 1 bits | 1 bit | 32 bits | 32 bits | - | 6 bits | 5 bits | 5 bits | - |

***lbu ( 82 ACE bits )***

Η εντολή lbu φορτώνει ένα byte από τη μνήμη χωρίς να κάνει επέκταση προσήμου. Η διεύθυνση προκύπτει από την πρόσθεση του πεδίου Data Read 1 και του πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή που δείχνει το πεδίο Rt. Τα πεδία Data Read 2, Rd και NPC δεν μας ενδιαφέρουν.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 4 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***lhu ( 82 ACE bits )***

Η εντολή lbu φορτώνει μισή λέξη, 16 bits, από τη μνήμη χωρίς να κάνει επέκταση προσήμου. Η διεύθυνση προκύπτει από την πρόσθεση του πεδίου Data Read 1 και του πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή που δείχνει το πεδίο Rt. Τα πεδία Data Read 2, Rd και NPC δεν μας ενδιαφέρουν.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 4 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***lui ( 48 ACE bits )***

Η εντολή lui αποθηκεύει στα 16 bit του πεδίου immediate στα 16 πρώτα bit του καταχωρητή προορισμού. Τα υπόλοιπα 16 bit τα μηδενίζει. Τα πεδία NPC, Data Read 1 και 2 καθώς και το πεδίο Rd, δεν χρησιμοποιούνται.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | - | - | 32 bits | 5 bits | 5 bits | - |

***lw ( 82 ACE bits )***

Η εντολή lw φορτώνει μία λέξη, 32 bits, από τη μνήμη. Η διεύθυνση προκύπτει από την πρόσθεση του πεδίου Data Read 1 και του πεδίου immediate. Το αποτέλεσμα αποθηκεύεται στον καταχωρητή που δείχνει το πεδίο Rt. Τα πεδία Data Read 2, Rd και NPC δεν μας ενδιαφέρουν.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 4 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***nor ( 86 ACE bits )***

Η εντολή and ανήκει στις εντολές τύπου R. Πραγματοποιεί την λογική πράξη NOR μεταξύ των δύο τελεστών οι οποίοι περιέχονται στα πεδία Data Read 1 και 2. Το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***or ( 86 ACE bits )***

Η εντολή είναι παρόμοια με την εντολή and. Πραγματοποιεί την λογική πράξη OR μεταξύ των δύο τελεστών οι οποίοι περιέχονται στα πεδία Data Read 1 και 2. Το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***ori ( 80 ACE bits )***

Η εντολή ori εκτελεί την λογική πράξη AOR μεταξύ δύο αριθμών. Ο πρώτος περιέχεται σε έναν καταχωρητή και ο δεύτερος βρίσκεται στο πεδίο immediate. Το πεδίο NPC, Rd καθώς και το Data Read 2 δεν μας ενδιαφέρουν, μας ενδιαφέρουν όμως τα πεδία Rt και Rs.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***slt ( 86 ACE bits )***

Η εντολή συγκρίνει τα περιεχόμενα του καταχωρητή Rs με τον Rt. Εφόσον η τιμή που κρατείται στον Rs είναι μικρότερη από αυτή του Rt, θέτει στον καταχωρητή Rd την τιμή 1. Στην αντίθετη περίπτωση, θέτει την τιμή 0. Χρησιμοποιούνται τα πεδία Data Read 1 και 2, το πεδίο Rd και 6 από τα bit του πεδίου immediate τα οποία αποτελούν το πεδίο func της εντολής. Το πεδίο NPC δεν χρησιμοποιείται.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***slti ( 80 ACE bits )***

Η εντολή συγκρίνει τα περιεχόμενα του καταχωρητή Rs με το πεδίο immediate. Εφόσον η τιμή που κρατείται στον Rs είναι μικρότερη από αυτή του Rt, θέτει στον καταχωρητή Rd την τιμή 1. Στην αντίθετη περίπτωση, θέτει την τιμή 0. Χρησιμοποιούνται τα πεδία Data Read 1, το πεδίο Rt και το πεδίο immediate. Τα πεδία NPC και Rd δεν χρησιμοποιούνται.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***sltiu ( 80 ACE bits )***

Η εντολή συγκρίνει τα περιεχόμενα του καταχωρητή Rs με το πεδίο immediate θεωρώντας πως οι αριθμοί δεν έχουν πρόσημο. Εφόσον η τιμή που κρατείται στον Rs είναι μικρότερη από αυτή του Rt, θέτει στον καταχωρητή Rd την τιμή 1. Στην αντίθετη περίπτωση, θέτει την τιμή 0. Χρησιμοποιούνται τα πεδία Data Read 1, το πεδίο Rt και το πεδίο immediate. Τα πεδία NPC και Rd δεν χρησιμοποιούνται.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - |

***sltu ( 86 ACE bits )***

Η εντολή συγκρίνει τα περιεχόμενα του καταχωρητή Rs με τον Rt και θεωρεί πως δεν έχουν πρόσημο. Εφόσον η τιμή που κρατείται στον Rs είναι μικρότερη από αυτή του Rt, θέτει στον καταχωρητή Rd την τιμή 1. Στην αντίθετη περίπτωση, θέτει την τιμή 0. Χρησιμοποιούνται τα πεδία Data Read 1 και 2, το πεδίο Rd και 6 από τα bit του πεδίου immediate τα οποία αποτελούν το πεδίο func της εντολής. Το πεδίο NPC δεν χρησιμοποιείται.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|

| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |
|---|---|---|---|---|---|---|---|---|---|---|

***sll ( 59 ACE bits )***

Η εντολή sll πραγματοποιεί αριστερή ολίσθηση του περιεχομένου του καταχωρητή Rt και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Τα πεδία NPC και Data Read 1 δεν μας ενδιαφέρουν. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 11 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU καθώς και η shamt, που ορίζει τις θέσεις ολίσθησης. Τέλος, μας ενδιαφέρει το πεδίο Rd μιας και δείχνει τον καταχωρητή προορισμού.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | - | 32 bits | 11 bits | - | 5 bits | 5 bits |

***nop ( 22 ACE bits )***

Η εντολή nop είναι στην ουσία η εντολή sll $0, $0, 0x0000. Τα ACE bits της είναι παρόμοια με της sll αλλά μιας και το αποτέλεσμα προωθείται στον καταχωρητή 0 ο οποίος έχει πάντα την τιμή 0, δεν έχει σημασία ούτε το πεδίο Data Read 2 αλλά ούτε και οι θέσεις ολίσθησης. Αποτέλεσμα είναι να μας ενδιαφέρουν μόνο τα 6 τελευταία bit του immediate και το πεδίο Rd.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | - | - | 6 bits | 5 bits | 5 bits | 5 bits |

***srl ( 59 ACE bits )***

Η εντολή sll πραγματοποιεί δεξιά ολίσθηση του περιεχομένου του καταχωρητή Rt και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Τα πεδία NPC και Data Read 1 δεν μας ενδιαφέρουν. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 11 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU καθώς και η shamt, που ορίζει τις θέσεις ολίσθησης. Τέλος, μας ενδιαφέρει το πεδίο Rd μιας και δείχνει τον καταχωρητή προορισμού.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | - | 32 bits | 11 bits | - | 5 bits | 5 bits |

***sb ( 83 ACE bits )***

Η εντολή sb αποθηκεύει ένα byte στη θέση μνήμης που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του πεδίου immediate. Τα δεδομένα προς εγγραφή βρίσκονται στον καταχωρητή Rt. Δεν μας ενδιαφέρουν τα πεδία NPC, Rt και Rd ενώ είναι απαραίτητα τα πεδία Data Read 1, Data read 2 και Immediate.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 6 bits | 4 bits | 1 bits | - | - | 32 bits | 8 bits | 32 bits | 5 bits | 5 bits | - |

***sh ( 91 ACE bits )***

Η εντολή sh αποθηκεύει δύο bytes στη θέση μνήμης που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του πεδίου immediate. Τα δεδομένα προς εγγραφή βρίσκονται στον καταχωρητή Rt. Δεν μας ενδιαφέρουν τα πεδία NPC, Rt και Rd ενώ είναι απαραίτητα τα πεδία Data Read 1, Data read 2 και Immediate.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 6 bits | 4 bits | 1 bits | - | - | 32 bits | 16 bits | 32 bits | 5 bits | 5 bits | - |

***sw ( 107 ACE bits )***

Η εντολή sw αποθηκεύει μία λέξη στη θέση μνήμης που προκύπτει από την πρόσθεση των περιεχομένων του καταχωρητή Rs και του πεδίου immediate. Τα δεδομένα προς εγγραφή βρίσκονται στον καταχωρητή Rt. Δεν μας ενδιαφέρουν τα πεδία NPS, Rt και Rd ενώ είναι απαραίτητα τα πεδία Data Read 1, Data read 2 και Immediate.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 6 bits | 4 bits | 1 bits | - | - | 32 bits | 32 bits | 32 bits | 5 bits | 5 bits | - |

***sub ( 86 ACE bits)***

Το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Η συγκεκριμένη εντολή αφαιρεί τον δεύτερο τελεστή από τον πρώτο. Οι δύο αριθμοί περιέχονται στα πεδία Data Read 1 και 2. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU. Τέλος, μας ενδιαφέρει το πεδίο Rd.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

***subu ( 86 ACE bits)***

Η εντολή εκτελεί παρόμοια λειτουργία με την sub και τα ACE bits είναι ίδια. Συγκεκριμένα, το πεδίο NPC δεν μας ενδιαφέρει μιας και χρησιμεύει μόνο στις εντολές άλματος. Η συγκεκριμένη εντολή αφαιρεί τον δεύτερο τελεστή από τον πρώτο. Οι δύο αριθμοί περιέχονται στα πεδία Data Read 1 και 2. Από το πεδίο immediate μας ενδιαφέρουν μόνο τα τελευταία 6 bit από τα οποία προκύπτει η τιμή func που ορίζει την λειτουργία της ALU. Τέλος, μας ενδιαφέρει το πεδίο Rd.

| EX | M | WB | Eq | NPC | Data Read 1 | Data Read 2 | Immediate | Rs | Rt | Rd |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 bits | 2 bits | 2 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits |

Συνοψίζοντας την ανάλυση για τον καταχωρητή ID/EX δημιουργούμε τον παρακάτω πίνακα.

| **Εντολή** | **Πεδία καταχωρητή ID/EX** | | | | | | | | | **ACE bits** |
|---|---|---|---|---|---|---|---|---|---|---|
| | **ctrl** | **Eq** | **NPC** | **DR 1** | **DR 2** | **Imm** | **Rs** | **Rt** | **Rd** | |
| add | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| addi | 11 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 85 bits |
| addiu | 11 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 85 bits |
| addu | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| and | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| andi | 11 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 85 bits |
| beq | 5 bits | 1 bit | 32 bits | - | - | 32 bits | 5 bits | 5 bits | - | 80 bits |
| bne | 5 bits | 1 bit | 32 bits | - | - | 32 bits | 5 bits | 5 bits | - | 80 bits |
| j | 5 bits | 1 bit | 32 bits | - | - | 32 bits | - | - | - | 70 bits |
| jal | 11 bits | 1 bit | 32 bits | 32 bits | 32 bits | 32 bits | 5 bits | 5 bits | - | 150 bits |
| jr | 6 bits | 1 bit | 32 bits | 32 bits | - | 6 bits | 5 bits | 5 bits | - | 87 bits |
| lbu | 13 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 87 bits |
| lhu | 13 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 87 bits |
| lui | 11 bits | - | - | - | - | 32 bits | 5 bits | 5 bits | - | 53 bits |

| Εντολή | Πεδία καταχωρητή ID/EX | | | | | | | | | ACE bits |
|---|---|---|---|---|---|---|---|---|---|---|
| | ctrl | Eq | NPC | DR 1 | DR 2 | Imm | Rs | Rt | Rd | |
| lw | 13 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 87 bits |
| nor | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| or | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| ori | 11 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 85 bits |
| slt | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| slti | 11 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 85 bits |
| sltiu | 11 bits | - | - | 32 bits | - | 32 bits | 5 bits | 5 bits | - | 85 bits |
| sltu | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| sll | 11 bits | - | - | - | 32 bits | 11 bits | - | 5 bits | 5 bits | 64 bits |
| srl | 11 bits | - | - | - | 32 bits | 11 bits | - | 5 bits | 5 bits | 64 bits |
| sb | 11 bits | - | - | 32 bits | 8 bits | 32 bits | 5 bits | 5 bits | - | 93 bits |
| sh | 11 bits | - | - | 32 bits | 16 bits | 32 bits | 5 bits | 5 bits | - | 101 bits |
| sw | 11 bits | - | - | 32 bits | 32 bits | 32 bits | 5 bits | 5 bits | - | 117 bits |
| sub | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| subu | 11 bits | - | - | 32 bits | 32 bits | 6 bits | 5 bits | 5 bits | 5 bits | 96 bits |
| nop[1] | 11 bits | - | - | - | - | 6 bits | - | 5 bits | 5 bits | 27 bits |

**Πίνακας 13: ACE bits καταχωρητή ID/EX.**

### ~~3.14.4~~ 3.14.4 EX/MEM ACE bits

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ο καταχωρητής EX/MEM περιέχει τα bit που προωθούνται από το στάδιο εκτέλεσης, στο στάδιο μνήμης. Αυτά είναι τα bit ελέγχου των σταδίων μνήμης και επανεγγραφής, η διεύθυνση άλματος, το bit "zero", το αποτέλεσμα της αριθμητικής λογικής μονάδας, τα δεδομένα που αναγνώστηκαν από τον δεύτερο καταχωρητή από το αρχείο καταχωρητών καθώς και ο κωδικός του καταχωρητή προορισμού. Στο σύνολό τους, είναι 108 bits.

| M ctrl | WB ctrl | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|---|
| 4 bits | 2 bits | 32 bits | 32 bits | 5 bits |

***add ( 41 ACE bits )***

Στην εντολή add, στο στάδιο EX/MEM δεν εκτελείται καμία λειτουργία. Τα αποτελέσματα που έρχονται από το στάδιο EX, απλά προωθούνται στο στάδιο WB. Επομένως δεν μας απασχολεί το πεδίο Read Data 2 και μας ενδιαφέρουν μόνο τα πεδία ALU result και R Dest καθώς και τα control bits.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***addi ( 41 ACE bits )***

[1] Η εντολή nop στον mips είναι η εντολή sll 0, 0, 0 αλλά την αναφέρουμε ξεχωριστά μιας και δεν πραγματοποιείται καμία εγγραφή στο αρχείο καταχωρητών οπότε τα ACE bits της είναι διαφορετικά από αυτά μίας τυπικής sll.

Στην εντολή addi δεν μας απασχολεί το πεδίο Read Data 2 και μας ενδιαφέρουν μόνο τα πεδία ALU result και R Dest καθώς και τα control bits.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***addiu ( 41 ACE bits )***

Στην εντολή addiu δεν μας απασχολεί το πεδίο Read Data 2 και μας ενδιαφέρουν μόνο τα πεδία ALU result και R Dest καθώς και τα control bits.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***addu ( 41 ACE bits )***

Στην εντολή addu δεν μας απασχολεί το πεδίο Read Data 2 και μας ενδιαφέρουν μόνο τα πεδία ALU result και R Dest καθώς και τα control bits.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***and ( 41 ACE bits )***

Στην εντολή and δεν μας απασχολεί το πεδίο Read Data 2 και μας ενδιαφέρουν μόνο τα πεδία ALU result και R Dest καθώς και τα control bits.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***andi ( 41 ACE bits )***

Στην εντολή andi δεν μας απασχολεί το πεδίο Read Data 2 και μας ενδιαφέρουν μόνο τα πεδία ALU result και R Dest καθώς και τα control bits.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***aeq ( 3 ACE bits )***

Η εντολή beq είναι εντολή άλματος. Η περισσότερη δουλειά των εντολών άλματος εκτελείται στα προηγούμενα στάδια. Μας ενδιαφέρουν μόνο τα bit ελέγχου.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 3 bits | - | - | - |

***ane ( 3 ACE bits )***

Η εντολή bne είναι παρόμοια με την εντολή beq. Όπως και στην beq, μας ενδιαφέρουν μόνο τα bit ελέγχου.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 3 bits | - | - | - |

***j ( 3 ACE bits )***

Η εντολή j είναι παρόμοια με τις εντολές beq και bne. Μας ενδιαφέρουν μόνο τα bit ελέγχου.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 3 bits | - | - | - |

***jal ( 41 ACE bits )***

Η εντολή jal διαφέρει αρκετά από τις υπόλοιπες εντολές άλματος μιας και εκτός από το άλμα, εκτελεί και εγγραφή δεδομένων στον καταχωρητή #31. Μας ενδιαφέρουν τα bit ελέγχου καθώς και το αποτέλεσμα της ALU μαζί με την κωδική λέξη του καταχωρητή προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***jr ( 3 ACE bits )***

Η εντολή jr εκτελεί άλμα στην διεύθυνση που δείχνει ένας καταχωρητής. Όσον αφορά τα στάδια MEM και WB, έχει τα ίδια κρίσιμα bit με τις εντολές bne, beq και j. Επομένως μας ενδιαφέρουν μόνο τα bit ελέγχου.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 3 bits | - | - | - |

***lbu ( 43ACE bits )***

Η εντολή lbu διαβάζει ένα byte από τη μνήμη. Η διεύθυνση ανάγνωσης περιέχεται στο αποτέλεσμα της ALU. Επίσης μας ενδιαφέρουν τα bit ελέγχου καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 6 bits | 32 bits | - | 5 bits |

***lhu ( 43 ACE bits )***

Η εντολή lhu διαβάζει δύο bytes από τη μνήμη. Η διεύθυνση ανάγνωσης περιέχεται στο αποτέλεσμα της ALU. Επίσης μας ενδιαφέρουν τα bit ελέγχου καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 6 bits | 32 bits | - | 5 bits |

***lui ( 41 ACE bits )***

Η εντολή lui αποθηκεύει τα 16 bit του πεδίου immediate στα 16 υψηλότερα bits ενός καταχωρητή. Μας ενδιαφέρουν τα bit ελέγχου καθώς και το αποτέλεσμα της ALU όπως και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***lw ( 43 ACE bits )***

Η εντολή lw διαβάζει μία λέξη (4 bytes) από τη μνήμη. Η διεύθυνση ανάγνωσης περιέχεται στο αποτέλεσμα της ALU. Επίσης μας ενδιαφέρουν τα bit ελέγχου καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|

| 6 bits | 32 bits | - | 5 bits |
|---|---|---|---|

***nor ( 41 ACE bits )***

Η εντολή nor, όπως και όλες οι υπόλοιπες εντολές τύπου R, δεν εκτελούν προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***or ( 41 ACE bits )***

Η εντολή or είναι εντολή τύπου R. Δεν εκτελείται καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***ori ( 41 ACE bits )***

Η εντολή ori είναι εντολή τύπου I και εκτελεί παρόμοια λειτουργία με την εντολή or. Δεν εκτελείται καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***slt ( 41 ACE bits )***

Η εντολή slt δεν εκτελεί καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***slti ( 41 ACE bits )***

Η εντολή slti δεν εκτελεί καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***sltiu ( 41 ACE bits )***

Η εντολή sltiu δεν εκτελεί καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|

| 4 bits | 32 bits | - | 5 bits |
|---|---|---|---|

***sltu ( 41 ACE bits )***

Η εντολή sltu δεν εκτελεί καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***sll ( 41 ACE bits )***

Η εντολή sll είναι εντολή τύπου R. Δεν εκτελείται καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***srl ( 41 ACE bits )***

Η εντολή srl είναι εντολή τύπου R. Δεν εκτελείται καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***sb ( 45 ACE bits )***

Η εντολή sb αποθηκεύει ένα byte στην μνήμη. Η διεύθυνση μνήμης προέρχεται από την ALU και το byte προς αποθήκευση περιέχεται στο πεδίο “read data 2”. Το πεδίο του καταχωρητή προορισμού δεν μας ενδιαφέρει.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 5 bits | 32 bits | 8 bits | - |

***sh ( 53 ACE bits )***

Η εντολή sh αποθηκεύει δύο bytes στην μνήμη. Η διεύθυνση μνήμης προέρχεται από την ALU και τα δύο bytes περιέχονται στο πεδίο “read data 2”. Το πεδίο του καταχωρητή προορισμού δεν μας ενδιαφέρει.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 5 bits | 32 bits | 16 bits | - |

***sw ( 69 ACE bits )***

Η εντολή sw αποθηκεύει μία λέξη (4 bytes) στην μνήμη. Η διεύθυνση μνήμης προέρχεται από την ALU και η προς αποθήκευση λέξη περιέχεται στο πεδίο “read data 2”. Το πεδίο του καταχωρητή προορισμού δεν μας ενδιαφέρει.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|

| 5 bits | 32 bits | 32 bits | - |
|---|---|---|---|

***sub ( 41 ACE bits )***

Η εντολή sub είναι εντολή τύπου R. Δεν εκτελείται καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

***subu ( 41 ACE bits )***

Η εντολή subu είναι εντολή τύπου R. Δεν εκτελείται καμία προσπέλαση μνήμης. Τα αποτελέσματα της ALU προωθούνται στο επόμενο στάδιο. Ενδιαφέρον έχουν τα bit ελέγχου, το αποτέλεσμα της ALU καθώς και ο καταχωρητής προορισμού.

| Control Bits | ALU result | Read Data 2 | R Dest. |
|---|---|---|---|
| 4 bits | 32 bits | - | 5 bits |

Τα παραπάνω συμπεράσματα συνοψίζονται στον παρακάτω πίνακα:

| **Εντολή** | **Πεδία καταχωρητή EX/MEM** | | | | **ACE bits** |
|---|---|---|---|---|---|
| | **Control Bits** | **ALU result** | **Read Data 2** | **R Dest.** | |
| Add | 4 bits | 32 bits | - | 5 bits | 41 bits |
| Addi | 4 bits | 32 bits | - | 5 bits | 41 bits |
| addiu | 4 bits | 32 bits | - | 5 bits | 41 bits |
| Addu | 4 bits | 32 bits | - | 5 bits | 41 bits |
| And | 4 bits | 32 bits | - | 5 bits | 41 bits |
| andi | 4 bits | 32 bits | - | 5 bits | 41 bits |
| beq | 3 bits | - | - | - | 3 bits |
| bne | 3 bits | - | - | - | 3 bits |
| j | 3 bits | - | - | - | 3 bits |
| jal | 4 bits | 32 bits | - | 5 bits | 41 bits |
| jr | 3 bits | - | - | - | 3 bits |
| lbu | 6 bits | 32 bits | - | 5 bits | 43 bits |
| lhu | 6 bits | 32 bits | - | 5 bits | 43 bits |
| lui | 4 bits | 32 bits | - | 5 bits | 41 bits |
| lw | 6 bits | 32 bits | - | 5 bits | 43 bits |
| Nor | 4 bits | 32 bits | - | 5 bits | 41 bits |
| Or | 4 bits | 32 bits | - | 5 bits | 41 bits |
| ori | 4 bits | 32 bits | - | 5 bits | 41 bits |
| slt | 4 bits | 32 bits | - | 5 bits | 41 bits |
| slti | 4 bits | 32 bits | - | 5 bits | 41 bits |
| sltiu | 4 bits | 32 bits | - | 5 bits | 41 bits |

| Εντολή | Πεδία καταχωρητή EX/MEM | | | | ACE bits |
|---|---|---|---|---|---|
| | **Control Bits** | **ALU result** | **Read Data 2** | **R Dest.** | |
| sltu | 4 bits | 32 bits | - | 5 bits | 41 bits |
| sll | 4 bits | 32 bits | - | 5 bits | 41 bits |
| srl | 4 bits | 32 bits | - | 5 bits | 41 bits |
| sb | 5 bits | 32 bits | 8 bits | - | 45 bits |
| sh | 5 bits | 32 bits | 16 bits | - | 53 bits |
| sw | 5 bits | 32 bits | 32 bits | - | 69 bits |
| Sub | 4 bits | 32 bits | - | 5 bits | 41 bits |
| subu | 4 bits | 32 bits | - | 5 bits | 41 bits |
| nop[1] | 4 bits | - | - | 5 bits | 9 bits |

**Πίνακας 14: ACE bits καταχωρητή EX/MEM.**

### ~~3.14.5~~ 3.14.5 MEM/WB ACE bits

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ο καταχωρητής MEM/WB παρεμβάλλεται των σταδίων μνήμης και επιστροφής του αποτελέσματος στο αρχείο καταχωρητών. Είναι ο δεύτερος μικρότερος σε μέγεθος καταχωρητής, μετά τον καταχωρητή IF/ID, και περιλαμβάνει 2 bit ελέγχου, τη λέξη που αναγνώστηκε από τη μνήμη δεδομένων, το αποτέλεσμα της αριθμητικής λογικής μονάδας καθώς και τον κωδικό του καταχωρητή προορισμού. Στο σύνολό τους, είναι 71 bits.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | 32 bits | 32 bits | 5 bits |

***add ( 39 ACE bits )***

Η εντολή add αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***addi ( 39 ACE bits )***

Η εντολή addi αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***addiu ( 39 ACE bits )***

Η εντολή addiu αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

[1] Ειδική περίπτωση της sll (Rs = 0, Rt = 0, shamt = 0).

***addu ( 39 ACE bits )***

Η εντολή addu αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***and ( 39 ACE bits )***

Η εντολή and αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***andi ( 39 ACE bits )***

Η εντολή andi αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***beq ( 1 ACE bit )***

Η εντολή beq δεν έχει δεδομένα προς αποθήκευση οπότε μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει πράξη εγγραφής ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***bne ( 1 ACE bit )***

Η εντολή bne δεν έχει δεδομένα προς αποθήκευση οπότε μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει πράξη εγγραφής ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***j ( 1 ACE bit )***

Η εντολή j δεν έχει δεδομένα προς αποθήκευση οπότε μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει πράξη εγγραφής ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***jal ( 39 ACE bits )***

Η εντολή jal εκτελεί άλμα αλλά και αποθηκεύει τη διεύθυνση της μεθεπόμενης εντολής στον καταχωρητή $31 επομένως έχει 39 κρίσιμα bits.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***jr ( 1 ACE bits )***

Η εντολή jr δεν έχει δεδομένα προς αποθήκευση οπότε μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει πράξη εγγραφής ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***lbu ( 39 ACE bits )***

Η εντολή lbu αποθηκεύει σε καταχωρητή τα δεδομένα που αναγνώστηκαν από τη μνήμη. Ενδιαφέρον έχουν 39 bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | 32 bits | - | 5 bits |

***lhu ( 39 ACE bits )***

Η εντολή lhu αποθηκεύει σε καταχωρητή τα δεδομένα που αναγνώστηκαν από τη μνήμη επομένως έχει 39 κρίσιμα bits.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | 32 bits | - | 5 bits |

***lui ( 39 ACE bits )***

Η εντολή lui αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***lw ( 39 ACE bits )***

Η εντολή lw αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | 32 bits | - | 5 bits |

***nor ( 39 ACE bits )***

Η εντολή nor αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***or ( 39 ACE bits )***

Η εντολή or αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***ori ( 39 ACE bits )***

Η εντολή ori αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***slt ( 39 ACE bits )***

Η εντολή slt αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***slti ( 39 ACE bits )***

Η εντολή slti αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***sltiu ( 39 ACE bits )***

Η εντολή sltiu αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rt επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***sltu ( 39 ACE bits )***

Η εντολή sltu αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***sll ( 39 ACE bits )***

Η εντολή sll αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***srl ( 39 ACE bits )***

Η εντολή srl αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***sb ( 1 ACE bit )***

Η εντολή sb αποθηκεύει δεδομένα στη μνήμη αλλά δεν αλλάζει την τιμή κανενός καταχωρητή. Επομένως μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει εγγραφή σε καταχωρητή ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***sh ( 1 ACE bit )***

Η εντολή sb αποθηκεύει δεδομένα στη μνήμη αλλά δεν αλλάζει την τιμή κανενός καταχωρητή. Επομένως μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει εγγραφή σε καταχωρητή ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***sw ( 1 ACE bit )***

Η εντολή sb αποθηκεύει δεδομένα στη μνήμη αλλά δεν αλλάζει την τιμή κανενός καταχωρητή. Επομένως μας ενδιαφέρει μόνο το bit που ορίζει το αν θα γίνει εγγραφή σε καταχωρητή ή όχι.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 1 bit | - | - | - |

***sub ( 39 ACE bits )***

Η εντολή sub αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

***subu ( 39 ACE bits )***

Η εντολή subu αποθηκεύει το αποτέλεσμα της πράξης της ALU στον καταχωρητή Rd επομένως έχει 39 κρίσιμα bit.

| Control bits | MEM Data Read | ALU result | R Dest. |
|---|---|---|---|
| 2 bits | - | 32 bits | 5 bits |

Οι παραπάνω παρατηρήσεις καταγράφονται στον ακόλουθο πίνακα:

| **Εντολή** | **Πεδία καταχωρητή MEM/WB** | | | | **ACE bits** |
|---|---|---|---|---|---|
| | **Control bits** | **MEM Data Read** | **ALU result** | **R Dest.** | |
| add | 2 bits | - | 32 bits | 5 bits | 39 bits |
| addi | 2 bits | - | 32 bits | 5 bits | 39 bits |
| addiu | 2 bits | - | 32 bits | 5 bits | 39 bits |
| addu | 2 bits | - | 32 bits | 5 bits | 39 bits |
| and | 2 bits | - | 32 bits | 5 bits | 39 bits |
| andi | 2 bits | - | 32 bits | 5 bits | 39 bits |
| beq | 1 bit | - | - | - | 1 bit |

| Εντολή | Πεδία καταχωρητή MEM/WB | | | | ACE bits |
|---|---|---|---|---|---|
| | Control bits | MEM Data Read | ALU result | R Dest. | |
| bne | 1 bit | - | - | - | 1 bit |
| j | 1 bit | - | - | - | 1 bit |
| jal | 2 bits | - | 32 bits | 5 bits | 39 bits |
| jr | 1 bit | - | - | - | 1 bit |
| lbu | 2 bits | 32 bits | - | 5 bits | 39 bits |
| lhu | 2 bits | 32 bits | - | 5 bits | 39 bits |
| lui | 2 bits | - | 32 bits | 5 bits | 39 bits |
| lw | 2 bits | 32 bits | - | 5 bits | 39 bits |
| nor | 2 bits | - | 32 bits | 5 bits | 39 bits |
| or | 2 bits | - | 32 bits | 5 bits | 39 bits |
| ori | 2 bits | - | 32 bits | 5 bits | 39 bits |
| slt | 2 bits | - | 32 bits | 5 bits | 39 bits |
| slti | 2 bits | - | 32 bits | 5 bits | 39 bits |
| sltiu | 2 bits | - | 32 bits | 5 bits | 39 bits |
| sltu | 2 bits | - | 32 bits | 5 bits | 39 bits |
| sll | 2 bits | - | 32 bits | 5 bits | 39 bits |
| srl | 2 bits | - | 32 bits | 5 bits | 39 bits |
| sb | 1 bit | - | - | - | 1 bit |
| sh | 1 bit | - | - | - | 1 bit |
| sw | 1 bit | - | - | - | 1 bit |
| sub | 2 bits | - | 32 bits | 5 bits | 39 bits |
| subu | 2 bits | - | 32 bits | 5 bits | 39 bits |
| nop[1] | 2 bits | - | - | 5 bits | 7 bits |

**Πίνακας 15: ACE bits καταχωρητή MEM/WB.**

### ~~3.14.6~~ 3.14.6 ACE bits της διοχέτευσης

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Συνοψίζοντας τα παραπάνω, σχηματίζεται ο παρακάτω πίνακας. Πρέπει να παρατηρήσουμε πως από εντολή σε εντολή οι διαφορές στα ACE bits είναι αισθητές.

| Εντολή | ACE bits καταχωρητών Pipeline | | | |
|---|---|---|---|---|
| | IF/ID (64 bits) | ID/EX (157 bits) | EX/MEM (75 bits) | MEM/WB (71 bits) |
| add | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| addi | 32 bits (50.00%) | 85 bits (54.14%) | 41 bits (54.67%) | 39 bits (54.93%) |
| addiu | 32 bits (50.00%) | 85 bits (54.14%) | 41 bits (54.67%) | 39 bits (54.93%) |
| addu | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |

[1] Ειδική περίπτωση της sll (Rs = 0, Rt = 0, shamt = 0).

| **Εντολή** | **ACE bits καταχωρητών Pipeline** | | | |
|---|---|---|---|---|
| | **IF/ID (64 bits)** | **ID/EX (157 bits)** | **EX/MEM (75 bits)** | **MEM/WB (71 bits)** |
| and | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| andi | 32 bits (50.00%) | 85 bits (54.14%) | 41 bits (54.67%) | 39 bits (54.93%) |
| beq | 64 bits (100.0%) | 80 bits (50.96%) | 3 bits ( 4.00%) | 1 bit ( 1.50%) |
| bne | 64 bits (100.0%) | 80 bits (50.96%) | 3 bits ( 4.00%) | 1 bit ( 1.50%) |
| j | 36 bits (56.25%) | 70 bits (44.59%) | 3 bits ( 4.00%) | 1 bit ( 1.50%) |
| jal | 36 bits (56.25%) | 155 bits (98.73%) | 41 bits (54.67%) | 39 bits (54.93%) |
| jr | 11 bits (17.19%) | 87 bits (55.42%) | 3 bits ( 4.00%) | 1 bit ( 1.50%) |
| lbu | 32 bits (50.00%) | 87 bits (55.42%) | 43 bits (57.34%) | 39 bits (54.93%) |
| lhu | 32 bits (50.00%) | 87 bits (55.42%) | 43 bits (57.34%) | 39 bits (54.93%) |
| lui | 28 bits (46.75%) | 53 bits (33.76%) | 41 bits (54.67%) | 39 bits (54.93%) |
| lw | 32 bits (50.00%) | 87 bits (55.42%) | 43 bits (57.34%) | 39 bits (54.93%) |
| nor | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| or | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| ori | 32 bits (50.00%) | 85 bits (54.14%) | 41 bits (54.67%) | 39 bits (54.93%) |
| slt | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| slti | 32 bits (50.00%) | 85 bits (54.14%) | 41 bits (54.67%) | 39 bits (54.93%) |
| sltiu | 32 bits (50.00%) | 85 bits (54.14%) | 41 bits (54.67%) | 39 bits (54.93%) |
| sltu | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| sll | 28 bits (46.75%) | 64 bits (40.77%) | 41 bits (54.67%) | 39 bits (54.93%) |
| srl | 28 bits (46.75%) | 64 bits (40.77%) | 41 bits (54.67%) | 39 bits (54.93%) |
| sb | 32 bits (50.00%) | 117 bits (74.53%) | 45 bits (60.00%) | 1 bit ( 1.50%) |
| sh | 32 bits (50.00%) | 117 bits (74.53%) | 53 bits (70.67%) | 1 bit ( 1.50%) |
| sw | 32 bits (50.00%) | 117 bits (74.53%) | 69 bits (92.00%) | 1 bit ( 1.50%) |
| sub | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| subu | 28 bits (46.75%) | 96 bits (61.15%) | 41 bits (54.67%) | 39 bits (54.93%) |
| nop[1] | 17 bits (26.56%) | 27 bits (17.20%) | 9 bits (12.00%) | 7 bits (9.86%) |

**Πίνακας 16: ACE bits καταχωρητών pipeline.**

## ~~3.15~~ 3.15 ΕΠΙΒΕΒΑΙΩΣΗ ΤΟΥ ΜΟΝΤΕΛΟΥ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Η τυπική επιβεβαίωση ενός οποιουδήποτε μοντέλου επεξεργαστή αποτελεί NP-complete πρόβλημα μιας και τα πιθανά δεδομένα εισόδου (τα προγράμματα που θα κληθεί να εκτελέσει ο επεξεργαστής) είναι ουσιαστικά άπειρα. Επομένως, την ορθή λειτουργία του μοντέλου την επιβεβαιώνουμε πειραματικά.

Το λογισμικό που αναπτύξαμε έχει τη δυνατότητα μετά το πέρας της εκτέλεσης να δημιουργήσει αρχεία με τα τελικά περιεχόμενα της μνήμης συστήματος καθώς και με πληροφορία σχετικά με την κατάσταση των καταχωρητών του επεξεργαστή σε κάθε βήμα της εκτέλεσης. Μετά από λεπτομερή

[1] Ειδική περίπτωση της sll (Rs = 0, Rt = 0, shamt = 0).

μελέτη των πειραμάτων που εκτελέσαμε, επιβεβαιώσαμε πως για αυτά τα πειράματα το μοντέλο λειτούργησε σωστά. Να σημειωθεί πως πέρα των πειραμάτων που παρουσιάζουμε στο 5$^{ο}$ Κεφάλαιο, εκτελέσαμε και άλλα μικρότερα που δεν έχουν ερευνητική αξία παρά μόνο επιβεβαίωναν την ορθή λειτουργία της κάθε εντολής.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

# 4 Ανάπτυξη Λογισμικού

## 4.1 ΕΙΣΑΓΩΓΗ

Σε αυτό το κεφάλαιο περιγράφεται η σχεδίαση του λογισμικού. Αναλύεται η μεθοδολογία που ακολουθήθηκε, περιγράφονται με λεπτομέρεια τα σχεδιαστικά μοτίβα (Design Patterns) που χρησιμοποιήθηκαν και αναφέρεται η λογική πίσω από τις αποφάσεις που ελήφθησαν.

Σκοπός του λογισμικού είναι η παραγωγή ενός αρχείου στο οποίο καταγράφεται κάθε εντολή πόσο χρόνο πέρασε σε κάθε στάδιο της διοχέτευσης και ποια bits ήταν κρίσιμα κάθε φορά κατά την εκτέλεση ενός προγράμματος. Κάθε επεξεργαστής μπορεί να διαιρεθεί σε αρκετά διακριτά μέρη όπως η κρυφή μνήμη, η διαδρομή των δεδομένων (datapath) και άλλα. Αυτά τα κομμάτια με τη σειρά τους μπορούν να χωριστούν σε άλλα μικρότερα ανάλογα την αρχιτεκτονική που μοντελοποιούμε και την ακρίβεια που θέλουμε να έχουν τα αποτελέσματα των πειραμάτων μας. Για παράδειγμα η κρυφή μνήμη μπορεί να χωριστεί σε κρυφή μνήμη δεδομένων και εντολών. Η διοχέτευση στο μοντέλο μας έχει 5 στάδια. Επίσης κάθε ένα από αυτά τα κομμάτια μπορεί να υλοποιηθεί με διαφορετικό τρόπο καθώς και να μοντελοποιηθεί με μεγαλύτερη ή και μικρότερη λεπτομέρεια.

Αυτά τα εξαρτήματα μπορούμε να τα μοντελοποιήσουμε είτε χρησιμοποιώντας τεχνικές δομημένου είτε αντικειμενοστρεφούς προγραμματισμού.

Αν επιλέξουμε την λύση του δομημένου προγραμματισμού, θα είναι δυσκολότερο να χρησιμοποιούμε ξανά κομμάτια κώδικα χωρίς να χρειαστεί να γίνουν μεγάλες αλλαγές. Επίσης, ένα πακέτο λογισμικού βασισμένο σε δομημένο προγραμματισμό καταλήγει να είναι δύσκολο στην διαχείριση του, δεν είναι εύκολα κατανοητό από κάποιον ο οποίος θα θέλει να το επεκτείνει και λόγω της πολυπλοκότητας του συγκεκριμένου συστήματος, είναι πιο επιρρεπές σε προγραμματιστικά σφάλματα.

Η λύση του αντικειμενοστρεφούς προγραμματισμού φαίνεται πιο δελεαστική. Κατ αρχάς, το σύστημα μας αποτελείται από πολλά τμήματα, αρκετά από τα οποία μπορούν να υλοποιηθούν με διαφορετικούς τρόπους. Με τον αντικειμενοστραφή προγραμματισμό μπορούμε να κάνουμε ακριβώς αυτό. Να φτιάξουμε μία δομή κλάσεων οι οποίες μπορούν να υλοποιούνται διαφορετικά, αναλόγως του τι θέλουμε να μοντελοποιήσουμε κάθε φορά. Επίσης μπορούμε να επαναχρησιμοποιούμε κομμάτια κώδικα με αποτέλεσμα να γλιτώνουμε χρόνο κατά την ανάπτυξη του λογισμικού αλλά και ο κώδικάς μας να είναι πιο συνεπής στις αρχικές σχεδιαστικές αποφάσεις. Επιπλέον, επειδή ο τρόπος σκέψης του ανθρώπου είναι πιο κοντά στον αντικειμενοστραφή παρά στον δομημένο προγραμματισμό, είναι ευκολότερο για κάποιο τρίτο άτομο να δει τον πηγαίο κώδικα και να κατανοήσει τη λειτουργία του.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## 4.2 ΜΕΘΟΔΟΛΟΓΙΑ ΑΝΑΠΤΥΞΗΣ

Η φύση του προβλήματος προς αντιμετώπιση, οδήγησε στην απόφαση να χρησιμοποιηθούν μεθοδολογίες αντικειμενοστρεφούς προγραμματισμού.

Η σχεδίαση έγινε με τη μέθοδο Staged Delivery και επιλέχτηκε η UML 2.0 για τη μοντελοποίηση του λογισμικού και δημιουργία των απαραίτητων διαγραμμάτων. Η σχεδίαση έγινε σε άλλα κομμάτια με βάση την top-down προσέγγιση και σε άλλα με βάση την bottom-up. Η πλειοψηφία του κώδικα βασίστηκε σε καταγεγραμμένα σχεδιαστικά μοτίβα.

**Μορφοποιήθηκε:** Ελληνικά

### ~~4.2.1~~ 4.2.1 Μέθοδος Staged Delivery

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Η μέθοδος staged delivery είναι μία υβριδική μέθοδος ανάπτυξης. Συνδυάζει στοιχεία της μεθόδου Waterfall και Iterative. Στην μέθοδο Waterfall η σχεδίαση και ανάπτυξη γίνεται σε τέσσερα στάδια. Πρώτα έχουμε την ανάλυση των απαιτήσεων, ύστερα γίνεται η σχεδίαση του λογισμικού με βάση της απαιτήσεις, έπειτα έρχεται η ανάπτυξη του κώδικα και τέλος ελέγχουμε αν ο κώδικας ανταποκρίνεται στις απαιτήσεις. Στην μέθοδο Iterative, το έργο χωρίζεται σε υποσύνολα βάση λειτουργικότητας. Στη συνέχεια υλοποιείται κάθε κομμάτι με διαδικασίες παρόμοιες της μεθοδολογίας waterfall και στο τέλος ενώνονται όλα τα τμήματα στο τελικό επιθυμητό σύστημα.

Με τη μέθοδο *staged delivery* η σχεδίαση του λογισμικού γίνεται όπως και στα δύο πρώτα στάδια της διαδικασίας waterfall. Η ανάλυση των απαιτήσεων καθώς και η σχεδίαση γίνεται για το σύνολο του λογισμικού στην αρχή. Ύστερα όμως η ανάπτυξη και ο έλεγχος του κώδικα χωρίζεται σε μικρότερα τμήματα, όπως και στη διαδικασία iterative.

Στην συγκεκριμένη εργασία, πρώτα σχεδιάσαμε πλήρως όλο το λογισμικό καθώς και το μοντέλο που θέλουμε να μελετήσουμε. Μόλις ολοκληρώθηκε η σχεδίαση, υλοποιήσαμε και δοκιμάσαμε κάθε τμήμα ξεχωριστά και προχωρήσαμε στα επόμενα. Όποτε χρειαζόμασταν κάποιο τμήμα που δεν είχαμε υλοποιήσει, φτιάχναμε ένα αντικείμενο το οποίο προσομοίαζε τη λειτουργία του τελικού τμήματος. Όταν για παράδειγμα υλοποιούσαμε το τμήμα προσκόμισης εντολής το οποίο αλληλεπιδρά με την κρυφή μνήμη εντολών, υλοποιήσαμε στην θέση της κρυφής μνήμης εντολών ένα αντικείμενο το οποίο επέστρεφε συγκεκριμένα δεδομένα ανάλογα με την θέση μνήμης που ζητάγαμε μέσω μίας εντολής “switch .. case” για συγκεκριμένες θέσεις μνήμης και την τιμή μηδέν (εντολή nop) για όλες τις υπόλοιπες. Με αυτόν τον τρόπο δεν είχαμε και περιορισμούς ως προς το ποιο τμήμα του λογισμικού έπρεπε να υλοποιηθεί πρώτο. Το μόνο που χρειάστηκε να υλοποιήσουμε πρώτο ήταν τα interfaces όλων των κλάσεων, πράγμα το οποίο έγινε στον πρώτο κύκλο ανάπτυξης.

### ~~4.2.2~~ 4.2.2 Προσέγγιση Top-Down

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Στη συγκεκριμένη προσέγγιση η σχεδίαση του συστήματος ξεκινά από τα ανώτερα επίπεδα του συστήματος και κατεβαίνουνε προς τα κάτω αναλύοντας κάθε φορά κάθε υποσύστημα.

Για παράδειγμα, ας υποθέσουμε πως το σύστημα που θέλουμε να μοντελοποιήσουμε είναι ένας επεξεργαστής. Αρχικά, μπορούμε να τον χωρίσουμε στη διοχέτευση και στην κρυφή μνήμη. Η διοχέτευση με τη σειρά του μπορεί να σπάσει σε άλλα 5 στάδια, τα Instruction Fetch, Instruction Decode, Execute, Memory, Writeback. Η κρυφή μνήμη μπορεί να χωριστεί είτε κατά επίπεδα, είτε κατά λειτουργία όπως σε κρυφή μνήμη εντολών και δεδομένων. Μπορούμε να προχωρήσουμε το μοντέλο μας όσο χαμηλά θέλουμε, ακόμα και σε επίπεδο τρανζίστορ εφόσον αυτό έχει νόημα.

Τα πλεονεκτήματα αυτής της προσέγγισης είναι τα παρακάτω:

- Η σχεδίαση μας απλοποιείται σε μικρότερα τμήματα.
- Λόγω αυτού του γεγονότος, η ανάπτυξη του κάθε κομματιού του λογισμικού μπορεί να γίνει ανεξάρτητα από τα άλλα μέρη.
- Έχουμε καθαρή εικόνα για το πως πρέπει να συνεργάζονται τα μέρη του λογισμικού.
- Γλιτώνουμε πολύτιμο χρόνο μιας και ο κάθε προγραμματιστής δεν χρειάζεται να ανησυχεί για το πως θα συνεργαστεί ο κώδικάς του με άλλα κομμάτια του συστήματος μιας και αυτό είναι ήδη προκαθορισμένο από την αρχική σχεδίαση.
- Ο παραγόμενος κώδικας είναι ευκολότερος στην αποσφαλμάτωση μιας και τα σφάλματα περιορίζονται στο κάθε τμήμα του.

Ένα από τα βασικά μειονεκτήματα αυτής της προσέγγισης είναι το γεγονός πως επειδή τα τελικά υποσυστήματα σχεδιάζονται με γνώμονα το top-level σύστημα, είναι δύσκολο να επαναχρησιμοποιηθούν σε άλλα έργα χωρίς σημαντικές αλλαγές.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

### ~~4.2.3~~ 4.2.3 Προσέγγιση Bottom-UP

Στην Bottom-Up προσέγγιση, ο σχεδιαστής ξεκινάει με τα χαμηλότερου επιπέδου τμήματα του συστήματος και χρησιμοποιώντας τα ,"χτίζει" το τελικό σύστημα. Ένα καλό παράδειγμα αυτής της τεχνικής είναι τα γνωστά σε όλους μας τουβλάκια LEGO. Πρώτα κατασκευάζονται τα τουβλάκια και μετά ο χρήστης τους, κατασκευάζει το τελικό προϊόν από αυτά τα μέρη.

Βασικό πλεονέκτημα αυτής της μεθόδου είναι πως μπορούμε να χρησιμοποιήσουμε τα κατώτερου επιπέδου μέρη πολλές φορές σε διαφορετικά συστήματα.

Το κύριο μειονέκτημα αυτής της προσέγγισης είναι πως πρέπει να έχουμε εξ αρχής μία πολύ καλή ιδέα για την τελική δομή του συστήματος που θέλουμε να κατασκευάσουμε και αυτή την καλή ιδέα πρέπει να την έχει κάθε μέλος της ομάδας. Αποτέλεσμα αυτού είναι, ειδικά σε μεγάλα και πολύπλοκα έργα, να πρέπει αρκετές φορές να γίνονται απρόβλεπτες αλλαγές οι οποίες επηρεάζουν σχεδόν όλο το έργο και να χάνεται πολύτιμος χρόνος. Επίσης, κάθε προγραμματιστής δεν μπορεί να εστιάσει στο κομμάτι κώδικα που γράφει αλλά πρέπει να σκέφτεται πώς αυτό το κομμάτι θα συνεργάζεται με τα υπόλοιπα στο μέλλον.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

### ~~4.2.4~~ 4.2.4 UML 2.0

Η UML είναι μία οικογένεια γραφικών αναπαραστάσεων υποστηριζόμενες από ένα μεταμοντέλο και βοηθάει στην περιγραφή και σχεδιασμό συστημάτων λογισμικού, ειδικά συστήματα βασισμένα στο μοντέλο του αντικειμενοστρεφούς προγραμματισμού [6].

Είναι ιδιαίτερα χρήσιμη στον σχεδιασμό πολύπλοκων συστημάτων και βοηθάει δραματικά σε θέματα επικοινωνίας όλων των εμπλεκόμενων μελών μιας και μπορεί να χρησιμοποιηθεί σαν μία κοινή γλώσσα. Κάποιος μπορεί να παραστήσει σε UML από πολύ μικρά τμήματα κώδικα μέχρι και ένα ολόκληρο σύστημα σε όσο μεγάλο βάθος θέλει. Υπάρχουν μάλιστα εργαλεία με τα οποία μπορεί κάποιος να περιγράψει ένα σύστημα και το εργαλείο μπορεί να του παράγει κώδικα βασισμένο σε αυτή την περιγραφή. Η UML μπορεί να χρησιμοποιηθεί ως ένα εργαλείο σε μια συζήτηση των 5 λεπτών έως ακόμα και ως ένα είδος γλώσσας προγραμματισμού.

Στην συγκεκριμένη εργασία την χρησιμοποιήσαμε προκειμένου να σχεδιάσουμε τη βασική δομή του λογισμικού που αναπτύξαμε και τα δύο κυριότερα και χρησιμότερα προς τον αναγνώστη διαγράμματα που δημιουργήσαμε παρατίθενται στο τέλος αυτού του κεφαλαίου.

Πρέπει να σημειωθεί πως αν και η UML ορίζει συγκεκριμένες γραφικές αναπαραστάσεις, μπορεί να κριθεί σκόπιμο εντός μίας ομάδας μηχανικών λογισμικού να τροποποιηθούν αυτές βάσει τις ανάγκες της ομάδας. Βασική προϋπόθεση είναι φυσικά είναι όλα τα μέλη της ομάδας να είναι ενήμερα για αυτές τις συμβάσεις.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

### ~~4.2.5~~ 4.2.5 Σχεδιαστικά μοτίβα (Design Patterns)

Όπως και σε άλλες πλευρές της μηχανικής, έτσι και στους μηχανικούς λογισμικού παρατηρήθηκαν κατά καιρούς κάποια συγκεκριμένα σχεδιαστικά μοτίβα τα οποία επαναλαμβάνονταν αρκετά συχνά. Τα τελευταία χρόνια γίνεται μία προσπάθεια να καταγραφούν αυτά τα μοτίβα και να καθιερωθεί μία γενικά αποδεκτή ονοματολογία.

Ο λόγος για αυτή την προσπάθεια είναι ώστε να μπορούν οι αρχιτέκτονες και οι μηχανικοί λογισμικού να έχουν μία κοινή γλώσσα όταν μιλάνε για τον σχεδιασμό ενός συστήματος καθώς και για να επιταχυνθεί αυτή η διαδικασία μιας και δεν χρειάζεται κάθε φορά "να ανακαλύπτουμε τον τροχό".

Τα κυριότερα βιβλία γύρω από τα σχεδιαστικά μοτίβα κατά τη συγγραφή αυτής της εργασίας είναι τα παρακάτω:

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, (1995).
  ISBN:0-201-63361-2.

- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, (1996).
  ISBN:0-471-95869-7. [3]

- Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, (2000).
  ISBN:0-471-60695-2. [4]

- Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, (2002).
  ISBN:978-0321127426. [5]

- Gregor Hohpe, Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, (2003).
  ISBN:0-321-20068-3. [7]

- Eric T. Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, (2004).
  ISBN:0-596-00712-4. [8]


***Abstract Factory [1]***

Θα ήταν χρήσιμο να μπορούμε να δούμε τι επίδραση έχουν στο AVF διαφορετικές υλοποιήσεις του επεξεργαστή που μοντελοποιούμε. Επίσης θα ήταν καλό να μπορέσουμε να χρησιμοποιήσουμε το ίδιο πρόγραμμα και στο μέλλον με μοντέλα άλλων επεξεργαστών χωρίς όμως να χρειάζεται να κάνουμε μεγάλες αλλαγές στον κώδικά μας. Κατάλληλο για αυτή του δουλειά είναι το σχεδιαστικό πρότυπο "Abstract Factory". Με βάση αυτό το πρότυπο έχουμε μία abstract class η οποία παρέχει ένα κοινό interface για τις διαφορετικές υλοποιήσεις. Η κάθε διαφορετική υλοποίηση είναι ένα subclass βασιζόμενο πάνω στο προαναφερθέν abstract class. Τέλος, έχουμε μία class η οποία παίρνει σαν παράμετρο ένα instance του subclass που μας ενδιαφέρει και περιέχει κλήσεις στις μεθόδους αυτής της factory class προκειμένου να δημιουργήσει τη δομή που θέλουμε.

Παράδειγμα:

```
CPU* AVFCalculator::CCreateCPU( CPUFactory& factory) {
CPU* cpuObject = new factory.newCPU();
 CPUComponent* InstructionFetchStage = new factory.newIF();
 CPUComponent* InstructionDecodeStage = new factory.newID();
 CPUComponent* ExecuteStage = new factory.newEX();
 CPUComponent* MemoryStage = new factory.newMEM();
 CPUComponent* WritebackStage = new factory.newWB();

cpuObject->setIF( InstructionFetchStage );
cpuObject->setID( InstructionDecodeStage );
cpuObject->setIEX( ExecuteStage );
cpuObject->setIMEM( MemoryStage );
cpuObject->setIWB( WritebackStage );
```

```
return cpuObject;
}
```

Με αυτό τον τρόπο μπορούμε να υλοποιούμε διαφορετικές εκδοχές του επεξεργαστή χωρίς να αλλάζουμε όλο το πρόγραμμά μας παρά μόνο τη factory class που περνάμε για παράμετρο παραπάνω.

**Μορφοποιήθηκε:** Ελληνικά

***Facade [2]***

Το συγκεκριμένο μοτίβο χρησιμοποιείται για να παρέχουμε μία ενοποιημένη διεπαφή έναντι ενός συνόλου διεπαφών για ένα σύστημα.

Αν για παράδειγμα προκειμένου να εκτελέσουμε μία λειτουργία σε ένα σύστημα κλάσεων πρέπει να καλέσουμε αρκετές από τις μεθόδους τους με συγκεκριμένο τρόπο κάθε φορά, μπορούμε να δημιουργήσουμε μία κλάση η οποία θα δέχεται τις αιτήσεις μας, θα εκτελεί αυτή τις απαιτούμενες κλήσεις στο σύστημα και θα μας επιστρέφει τα αποτελέσματα.

Με αυτό τον τρόπο, κρατάμε τον κώδικα των "πελατών" του συστήματος μικρό και ευανάγνωστο. Επίσης, ο κώδικας αυτός είναι ανεξάρτητος από τη δομή του συστήματος και αν αλλάξει κάτι στο σύστημα πρέπει να αλλάξουμε μόνο την κλάση Facade.

Παράδειγμα (υποθετικό):

```
Class CCPUFacade {
public:
        int GetTotalBits(CPU*);
}

Implementation:
int CCPUFacade::GetTotalNumberOfBits(CPU * qCpu) {
        int nTempBits = 0;
         nTempBits +=  qCpu->IF.getBits();
         nTempBits +=  qCpu->ID.getBits();
         nTempBits +=  qCpu->EX.getBits();
         nTempBits +=  qCpu->MEM.getBits();
         nTempBits +=  qCpu->WB.getBits();

         nTempBits +=  qCpu->InstructionCache.getBits();
        nTempBits +=  qCpu->DataCache.getBits();
        return nTempBits;
}
```

**Μορφοποιήθηκε:** Ελληνικά

**Μορφοποιήθηκε:** Ελληνικά

**Μορφοποιήθηκε:** Ελληνικά

**Μορφοποιήθηκε:** Ελληνικά

**Μορφοποιήθηκε:** Ελληνικά

Χρησιμοποιώντας την παραπάνω κλήση σε κάθε πελάτη του αντικειμένου που αντιπροσωπεύει τον επεξεργαστή αντί της συγκεκριμένης αλληλουχίας κλήσεων, καταφέρνουμε να απλοποιήσουμε τον κώδικα όλων των μεθόδων-πελατών του προγράμματος. Επίσης, αν αποφασίσουμε πως πρέπει να προσθέσουμε ή να αφαιρέσουμε κάποιο κομμάτι από το μοντέλο του επεξεργαστή, θα χρειάζεται να ενημερώσουμε μόνο τη συγκεκριμένη μέθοδο και όχι όλον τον κώδικα σε όλο το λογισμικό.

## ~~4.3~~ 4.3 UML ΔΙΑΓΡΑΜΜΑΤΑ ΛΟΓΙΣΜΙΚΟΥ

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Δεν θα μπούμε σε λεπτομέρειες σχετικά με το λογισμικό που αναπτύξαμε, ο αναγνώστης μπορεί να κοιτάξει τον κώδικα ο οποίος είναι αρκετά απλός και με αρκετά σχόλια τα οποία βοηθούν την κατανόησή του. Άλλωστε, ο κώδικας αυτός καθαυτός δεν έχει και τόσο μεγάλη σημασία όσο η μεθοδολογία που ακολουθήσαμε για την σχεδίαση του καθώς και η ανάλυση ACE η οποία φαίνεται στο κεφάλαιο 4.

Εδώ παραθέτουμε κάποια UML διαγράμματα τα οποία θα βοηθήσουν τον αναγνώστη στην καλύτερη κατανόηση του κώδικα που αναπτύξαμε καθώς και στο να βοηθηθεί να αναπτύξει το δικό του μοντέλο.

### ~~4.3.1~~ 4.3.1 UML μοντέλου της διοχέτευσης

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Το μοντέλο της διοχέτευσης αποτελείτε από τα παρακάτω αντικείμενα:

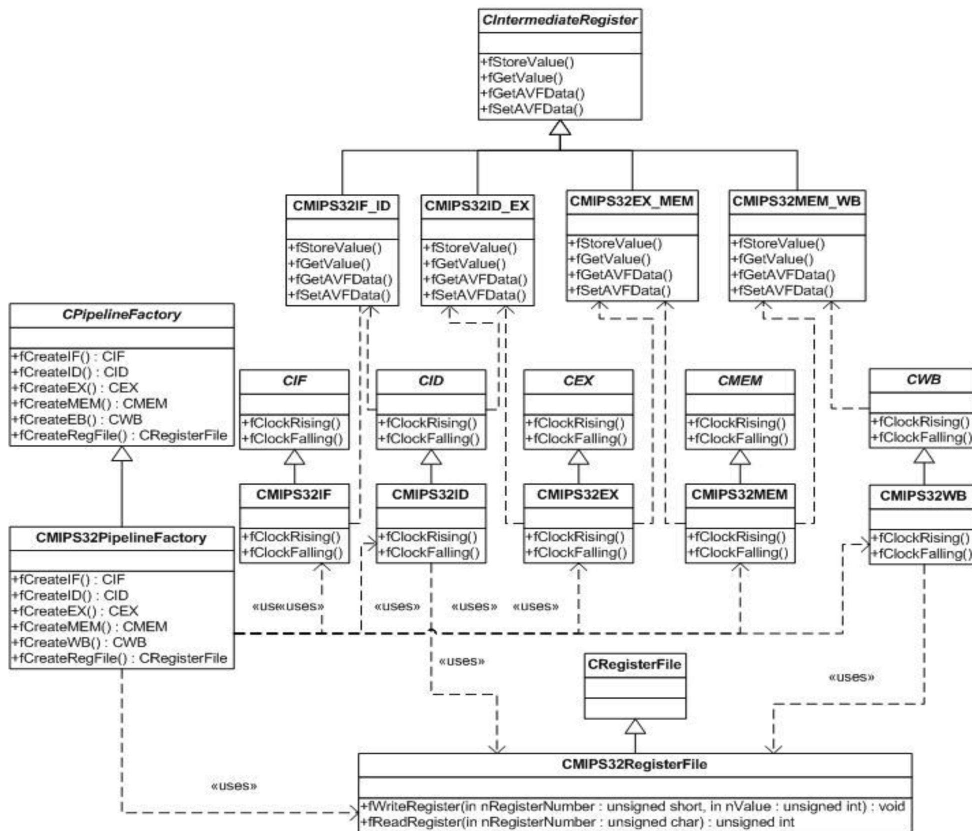
- Πέντε αντικείμενα τα οποία αντιπροσωπεύουν τα πέντε στάδια της διοχέτευσης του επεξεργαστή που μοντελοποιούμε.
- Τέσσερα αντικείμενα που αντιπροσωπεύουν τους ενδιάμεσους καταχωρητές.
- Ένα αντικείμενο το οποίο αντιπροσωπεύει το αρχείο καταχωρητών.
- Τις factory κλάσεις που δημιουργούν τα παραπάνω αντικείμενα.
- Την Facade κλάση η οποία αποτελεί την διεπαφή του μοντέλου με το υπόλοιπο λογισμικό.

Στην εικόνα 3 μπορούμε να δούμε καθαρά τις κλάσεις των παραπάνω αντικειμένων και το πώς αλληλεπιδρούν μεταξύ τους. Με ίσιες γραμμές φαίνονται οι διεπαφές της κάθε κλάσης και με διακεκομμένες φαίνεται το πώς αλληλεπιδρούν οι κλάσεις αυτές μεταξύ τους. Φαίνονται μόνο τα απαραίτητα τμήματα για την κατανόηση της λειτουργίας του μοντέλου. Διάφορες βοηθητικές μέθοδοι που μπορεί να έχει κάθε ένα από τα παραπάνω αντικείμενα δεν κρίνεται σκόπιμο να συμπεριληφθούν.

Κάθε ένα από τα στάδια της διοχέτευσης έχει μία μέθοδο fClockRising και μία μέθοδο fClockFalling. Αυτές οι μέθοδοι καλούνται για να προσημειώνεται η άνοδος και η κάθοδος του ρολογιού. Όλες σχεδόν οι διαδικασίες που εκτελεί το κάθε τμήμα, εκτελούνται μέσα σε αυτές τις δύο μεθόδους.

Κάθε ένας από τους ενδιάμεσους καταχωρητές περιέχει τις μεθόδους fStoreValue, fGetValue, fGetAVFData, fSetAVFData. Αυτές οι μέθοδοι χρησιμοποιούνται από τα αντικείμενα που αντιπροσωπεύουν τα στάδια της διοχέτευσης προκειμένου να αποθηκεύουν τα αποτελέσματα επεξεργασίας τους αλλά και να διαβάζουν τα δεδομένα εισόδου τους.

Το αρχείο καταχωρητών επικοινωνεί με τα στάδια ID και WB μέσω των μεθόδων fWriteRegister και fReadRegister.




**Εικόνα 3: UML διάγραμμα κλάσεων του μοντέλου της διοχέτευσης.**

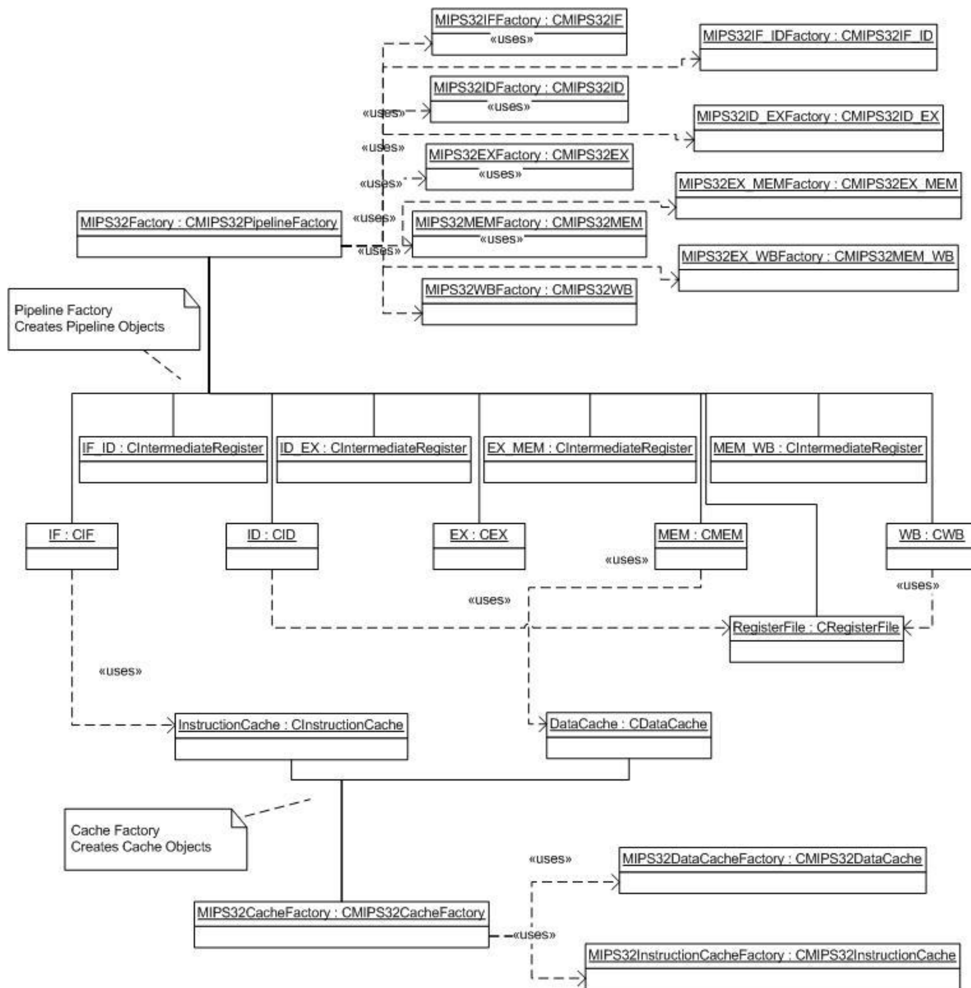
### 4.3.2 UML διάγραμμα μοντέλου επεξεργαστή

Η εικόνα 4 απεικονίζει τα σημαντικότερα αντικείμενα του τελικού μοντέλου που μελετάμε.

Εκτός από τα αντικείμενα της διοχέτευσης που αναλύσαμε προηγουμένως, φαίνονται και τα αντικείμενα που αποτελούν την κρυφή μνήμη προγράμματος και δεδομένων. Επίσης φαίνονται και τα αντικείμενα MIPS32PipelineFactory και MIPS32CacheFactory τα οποία χρησιμοποιούνται προκειμένου να δημιουργηθούν όλα τα υπόλοιπα αντικείμενα.

Βλέπουμε πως τα στάδια IF και MEM επικοινωνούν με τα αντικείμενα που αντιπροσωπεύουν την κρυφή μνήμη δεδομένων και προγράμματος.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση




**Εικόνα 4: UML διάγραμμα αντικειμένων του μοντέλου που μελετάμε.**

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

# 5 Πειράματα

## 5.1 ΕΙΣΑΓΩΓΗ

Ο προσομοιωτής που αναπτύχθηκε έχει τη δυνατότητα να εκτελεί προγράμματα στην αρχιτεκτονική που μοντελοποιούμε και να μας επιστρέφει ένα αρχείο με δεδομένα σχετικά με την εκτέλεση του προγράμματος καθώς και ένα αρχείο με τα δεδομένα της μνήμης του συστήματος και των καταχωρητών μετά το πέρας της εκτέλεσης. Αυτά τα δεδομένα μπορούν να χρησιμοποιηθούν για τον υπολογισμό του AVF αλλά και για την αποσφαλμάτωση του προγράμματος που τρέξαμε είτε για εντοπισμό πιθανών λαθών στην προσομοίωση.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## 5.2 ΜΟΡΦΗ ΑΡΧΕΙΟΥ ΠΕΙΡΑΜΑΤΟΣ

Το αρχείο πειράματος περιέχει τον κώδικα του προγράμματος που θέλουμε να τρέξουμε σε γλώσσα μηχανής σε δεκαεξαδική μορφή, τα δεδομένα της μνήμης συστήματος προτού ξεκινήσει η εκτέλεση του προγράμματος καθώς και κάποιες οδηγίες προς τον προσομοιωτή σχετικά με το πείραμα όπως τα επιθυμητά δεδομένα εξόδου, τυχόν πληροφορίες που θέλουμε να εμφανίζονται στην οθόνη κατά την εκτέλεση ή άλλες παραμέτρους του πειράματος. Πρέπει να σημειωθεί πως όποτε κάποιος σχεδιαστής αναπτύσσει μία νέα αρχιτεκτονική πρέπει εκτός από τα απαραίτητα μοντέλα και σχέδια, να αναπτύξει και εργαλεία ανάπτυξης λογισμικού όπως μεταγλωττιστές (compilers) και assemblers. Για την παραγωγή του κώδικα σε γλώσσα μηχανής, αναπτύξαμε ένα βοηθητικό πρόγραμμα στο οποίο ο χρήστης επιλέγει την εντολή, συμπληρώνει τα πεδία της και το πρόγραμμα παράγει την εντολή σε γλώσσα μηχανής σε δεκαεξαδική μορφή. Αυτό το πρόγραμμα συμπεριλαμβάνεται στο συνοδευτικό CD αλλά δεν αποτελεί αντικείμενο αυτής της εργασίας και για αυτό τον λόγο δεν αναλύεται.

Υποστηρίζονται οι παρακάτω εντολές-οδηγίες:

- cycles #αριθμός# - Ορίζει τον μέγιστο αριθμό κύκλων ρολογιού που θα τρέξει η προσομοίωση. Η εκτέλεση του προγράμματος που περιέχεται στο αρχείο πειράματος σταματάει όταν ο PC φτάσει να δείχνει μετά την τελευταία εντολή αλλά μπορεί για δικούς μας λόγους να θέλουμε να σταματάει η προσομοίωση νωρίτερα. Για αυτό τον λόγο, μπορούμε μέσω αυτής της οδηγίας να ορίσουμε συγκεκριμένο αριθμό κύκλων του ρολογιού που αν ξεπεραστούν, η εκτέλεση του προγράμματος σταματάει.
- saveavflog #αρχείο# - Η συγκεκριμένη οδηγία ορίζει πως θέλουμε να αποθηκεύσουμε ένα αρχείο το οποίο περιέχει δεδομένα σχετικά τα ACE bits που υπήρχαν σε κάθε καταχωρητή κατά τη διάρκεια του πειράματος. Συγκεκριμένα, το αρχείο περιέχει μία γραμμή κειμένου για κάθε εντολή που εκτελέστηκε και κάθε γραμμή περιέχει τον αριθμό των ACE bits ακολουθούμενο από τον αριθμό των κύκλων ρολογιού κατά τους οποίους παρέμειναν αυτά τα bit στον εκάστοτε καταχωρητή. Αν δηλαδή δούμε μία γραμμή να γράφει “32 1 87 2 43 2 39 1” σημαίνει πως η συγκεκριμένη εντολή είχε 32 ACE bits για 1 κύκλο στον 1ο εσωτερικό

καταχωρητή, 87 bits για 2 κύκλους στον 2ο καταχωρητή, 43 bits για 2 κύκλους στον 3ο καταχωρητή και 39 bits για 1 κύκλο στον 4ο καταχωρητή. Οι εντολές παραμένουν σε κάποιον καταχωρητή για περισσότερους από ένα κύκλο σε περιπτώσεις όπως μίας αστοχίας στην κρυφή μνήμη ή μίας φούσκας (bubble).

- printavf – Η συγκεκριμένη οδηγία κάνει τον προσομοιωτή να υπολογίζει την τιμή του AVF για όλο τον επεξεργαστή και να την τυπώνει στο τέλος της προσομοίωσης στην οθόνη. Πρέπει να σημειωθεί πως είναι προτιμότερο για λόγους μαθηματικής ακρίβειας ο AVF να εξάγεται από το αρχείο "avflog" με χρήση κάποιου προγράμματος όπως το MATLAB. Η συγκεκριμένη οδηγία όμως εξακολουθεί να έχει τη χρησιμότητά της αν θέλουμε κάποια γρήγορα αποτελέσματα χωρίς να μας ενδιαφέρει ιδιαίτερα η ακρίβειά τους.
- printavflog – Η συγκεκριμένη οδηγία εκτυπώνει μετά από κάθε φορά που κάποια εντολή φτάνει στο τέλος της διοχέτευσης την πληροφορία σχετική με τα ACE bit της εντολής αυτής, όπως αποθηκευόταν και στο αρχείο καταγραφής των δεδομένων του AVF με χρήση της εντολής saveavflog.
- printregs – Η οδηγία printregs εκτυπώνει μετά από κάθε φορά που κάποια εντολή φτάνει στο τέλος της διοχέτευσης τα περιεχόμενα των μη μηδενικών καταχωρητών στην οθόνη. Σκοπός αυτής της οδηγίας είναι να έχουμε μία πιο λεπτομερή εικόνα για την εκτέλεση του προγράμματος. Μπορεί να συνδυαστεί και με την οδηγία printavflog για ακόμα καλύτερη εικόνα των τεκταινόμενων στα εσωτερικά του μοντέλου.
- saveexecinfo #file# - Με την συγκεκριμένη οδηγία αποθηκεύεται σε ένα αρχείο πληροφορία για τα περιεχόμενα των καταχωρητών καθώς και της εντολής που εκτελέστηκε σε κάθε κύκλο ρολογιού.
- memdump #file# - Μετά το πέρας της εκτέλεσης του προγράμματος, αποθηκεύονται σε ένα αρχείο τα περιεχόμενα της μνήμης δεδομένων. Είναι ιδιαίτερα χρήσιμη εντολή και σε συνεργασία με την SaveExecInfo μπορούμε να επαληθεύσουμε τη σωστή λειτουργία του προσομοιωτή.
- data – Η συγκεκριμένη οδηγία ορίζει πως ακολουθούν τα δεδομένα του τμήματος δεδομένων κατά την αρχή του πειράματός μας. Το αρχείο δεδομένων ξεκινάει από τη διεύθυνση 0x08000000 και εκτείνεται μέχρι τη διεύθυνση 0x17FFFFFF. Κάθε γραμμή περιέχει μία λέξη των 32 bit σε 16δική μορφή. Είναι απαραίτητο να χρησιμοποιούμε 8 χαρακτήρες ανά εντολή, ακόμα και αν οι πρώτοι χαρακτήρες είναι μηδενικά.
- text – Η συγκεκριμένη οδηγία είναι και η σημαντικότερη. Ορίζει πως ακολουθεί ο κώδικας του προγράμματος προς εκτέλεση κατά το πείραμά μας. Ο κώδικας πρέπει να είναι σε γλώσσα μηχανής σε δεκαεξαδική μορφή. Είναι απαραίτητο να χρησιμοποιούμε 8 χαρακτήρες ανά εντολή, ακόμα και αν οι πρώτοι χαρακτήρες είναι μηδενικά. Επιτρέπεται μόνο μία εντολή ανά γραμμή κειμένου. Δίνεται η δυνατότητα προσθήκης σχολίων με χρήση του συμβόλου "//".

## ~~5.3~~ 5.3 ΠΕΙΡΑΜΑΤΑ

Μορφοποιημένο: Κουκκίδες και αρίθμηση

Ακολουθεί ένα σύνολο πειραμάτων που τρέξαμε με τον προσομοιωτή και τα αποτελέσματα τους. Τα προγράμματα επιλέχτηκαν έτσι ώστε να σχηματίσουμε μία καλή εικόνα για τον AVF της αρχιτεκτονικής που μοντελοποιούμε. Να αναφέρουμε πως δεν ακολουθούμε τις τυπικές συμβάσεις για τη χρήση των καταχωρητών μιας και τα συγκεκριμένα προγράμματα δεν έχουν καμία αξία πέρα των συγκεκριμένων πειραμάτων.

### ~~5.3.1~~ 5.3.1 Αριθμοί Fibonacci

Μορφοποιημένο: Κουκκίδες και αρίθμηση

Οι αριθμοί Fibonacci είναι οι αριθμοί μίας ακολουθίας ακεραίων αριθμών και ακολουθούν τον παρακάτω κανόνα:

- Οι πρώτοι δύο αριθμοί της ακολουθίας είναι οι 0, 1.

- Κάθε ένας από τους επόμενους αριθμού προκύπτει από το άθροισμα των δύο προηγούμενών του. Δηλαδή $F_n = F_{n-1} + F_{n-2}$.

Για παράδειγμα, οι πρώτοι 10 αριθμοί της ακολουθίας Fibonacci είναι οι:

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Ένα πρόγραμμα που υπολογίζει την ακολουθία Fibonacci είναι ιδιαίτερα εύκολο στην υλοποίησή του και το συγκεκριμένο πείραμα πραγματοποιεί διπλό σκοπό. Επειδή τα δεδομένα εισόδου καθώς και τα δεδομένα εξόδου είναι γνωστά και συγκεκριμένα, είναι ένα καλό παράδειγμα για να ελέγξουμε τόσο τη σωστή λειτουργία του προσομοιωτή, όσο και για να βγάλουμε κάποια συμπεράσματα σχετικά με τη συμπεριφορά της αρχιτεκτονικής μας σε περιπτώσεις όπου έχουμε πολλές προσπελάσεις στη μνήμη σε περιοχές τέτοιες ώστε έχουμε μία αστοχία στην κρυφή μνήμη ανά τρεις προσπελάσεις. Ας δούμε πρώτα το πρόγραμμα όπως το υλοποιούμε και στη συνέχεια θα το αναλύσουμε μαζί με τα αποτελέσματά του.

```
Τμήμα δεδομένων
00000000
00000001

Τμήμα κώδικα
3442000a // ori $2, $2, 0x000a
3c010800 // lui $1, $0, 0x0800
20210004 // addi $1, $1, 0x0004
8c25fffc // lw $5, 0xfffc($1)
8c260000 // lw $6, 0x0000($1)
00a63020 // add $6, $6, $5
ac260004 // sw $6, $1(0x0004)
20630001 // addi $3, $3, 0x0001
10430002 // beq $2, $3, 0x0002
00000000 // nop
08000002 // j 0x00000002
00000000 // nop
```

Το τμήμα δεδομένων περιέχει τα αρχικά δεδομένα μας. Συγκεκριμένα περιέχει τις λέξεις που αντιστοιχούν στους αριθμούς 0 και 1.

Ας δούμε τον κώδικα ξεκινώντας από τους καταχωρητές που χρησιμοποιούμε.

- $1 – Ο συγκεκριμένος καταχωρητής παίζει τον ρόλο του δείκτη προς τον (n-1) όρο της ακολουθίας Fibonacci. Χρησιμοποιείται προκειμένου να διαβαστούν από τη μνήμη του συστήματος οι δύο τελευταίοι αριθμοί της ακολουθίας αλλά και για να υπολογίσουμε τη διεύθυνση που θα αποθηκευτεί ο υπολογιζόμενος αριθμός.
- $5 – Περιέχει την τιμή του όρου n-2 της ακολουθίας Fibonacci.
- $6 – Περιέχει την τιμή του όρου n-1 της ακολουθίας Fibonacci πριν την εκτέλεση της πρόσθεσης των όρων n-2 και n1, και την τιμή του όρου n μετά την εκτέλεση της πρόσθεσης.
- $2 – Περιέχει τον αριθμό των επαναλήψεων που θέλουμε να τρέξει ο αλγόριθμος.
- $3 – Μετρητής που περιέχει των αριθμό της τρέχουσας επανάληψης.

Ο κώδικας είναι αρκετά απλός. Ξεκινάμε με το να ορίζουμε τον αριθμό των επιθυμητών επαναλήψεων στον καταχωρητή $2. Στη συνέχεια, φορτώνουμε στον καταχωρητή $1 τη διεύθυνση του στοιχείου n-1, στην περίπτωσή μας τη διεύθυνση του αριθμού 1. Στη συνέχεια ερχόμαστε στον κύριο βρόχο του προγράμματος. Διαβάζουμε από τη μνήμη τις τιμές των θέσεων n-1 και n-2 και τις αποθηκεύουμε στους καταχωρητές $5 και $6. Στη συνέχεια τους προσθέτουμε και αποθηκεύουμε το αποτέλεσμα στη θέση μνήμης n η οποία προκύπτει από την πράξη  $1 + 0χ0004. Τέλος, αυξάνουμε τον καταχωρητή $3 κατά 1 και εφόσον το περιεχόμενό του δεν είναι ίσο με τον $2 ο οποίος περιέχει τον

αριθμό των επιθυμητών επαναλήψεων, εκτελούμε ένα άλμα στην 3η εντολή του προγράμματος η οποία αυξάνει τον δείκτη του στοιχείου n-1 κατάλληλα ώστε να δείχνει στο σωστό στοιχείο. Ο βρόχος επαναλαμβάνεται μέχρι να φτάσουμε τον επιθυμητό αριθμό επαναλήψεων και να τερματίσουμε το πρόγραμμα. Η τελευταία εντολή nop είναι απαραίτητη μιας και στην αρχιτεκτονική MIPS εκτελείται πάντα η εντολή που ακολουθεί μία εντολή άλματος ή διακλάδωσης. Αν την παραλείψουμε, η εκτέλεση του προγράμματος θα σταματήσει χωρίς να πραγματοποιηθεί καμία επανάληψη.

Μετά την εκτέλεση του παραπάνω κώδικα, κοιτάμε το αρχείο καταγραφής των περιεχομένων της μνήμης δεδομένων:

```
08000000 - 00000000
08000004 - 00000001
08000008 - 00000001
0800000c - 00000002
08000010 - 00000003
08000014 - 00000005
08000018 - 00000008
0800001c - 0000000d
08000020 - 00000015
08000024 - 00000022
08000028 - 00000037
0800002c – 00000059
```

Το πρόγραμμα χρειάστηκε 125 κύκλους ρολογιού προκειμένου να ολοκληρωθεί και τα περιεχόμενά του ήταν αυτά που περιμέναμε. Δηλαδή οι πρώτοι 10 + 2 αριθμοί της ακολουθίας Fibonacci. Την σωστή εκτέλεση την επιβεβαιώνουμε κοιτώντας και το αρχείο καταγραφής της πληροφορίας εκτέλεσης, απόσπασμα του οποίου φαίνεται παρακάτω. Σε κάθε γραμμή έχουμε τον αριθμό της εντολής που εκτελέστηκε ο οποίος αυξάνει κατά 1 με κάθε νέα προσκόμιση, ακολουθούν οι καταχωρητές που περιέχουν μη μηδενικές τιμές και τέλος βρίσκουμε την διεύθυνση της εντολής, την εντολή που εκτελέστηκε καθώς και τα ACE bits και του κύκλους ρολογιού που παρέμεινε η εντολή σε κάθε καταχωρητή.:

| Καταχώρηση | Επεξήγηση |
|---|---|
| Cmd#:7, $1:8000004, $2:a,<br>0x8 [addi] - 32, 3 - 85, 3 - 41, 3 - 39, 3 | Αυξάνουμε το δείκτη του στοιχείου n κατά 1. |
| Cmd#:8, $1:8000004, $2:a,<br>0xc [lw] - 32, 3 - 87, 3 - 43, 3 - 39, 2 | Ανάγνωση του n-2. |
| Cmd#:9, $1:8000004, $2:a, $6:1,<br>0x10 [lw] - 32, 3 - 87, 3 - 43, 2 - 39, 3 | Ανάγνωση του n-1. |
| Cmd#:10, $1:8000004, $2:a, $6:1,<br>0x14 [stall] - 0, 0 - 27, 2 - 9, 3 - 7, 3 | Stall διότι την τελευταία ανάγνωση μνήμης ακολουθεί εντολή που χρησιμοποιεί τα δεδομένα της. |
| Cmd#:11, $1:8000004, $2:a, $6:1,<br>0x14 [R instr] - 28, 2 - 96, 3 - 41, 3 - 39, 3 | Προσθήκη των n-1 και n-2 |
| Cmd#:12, $1:8000004, $2:a, $6:1,<br>0x18 [sw] - 32, 3 - 117, 3 - 69, 3 - 1, 1 | Αποθήκευση του n. |
| Cmd#:13, $1:8000004, $2:a, $3:1, $6:1, | Αύξηση του n κατά 1. |

| Καταχώρηση | Επεξήγηση |
|---|---|
| 0x1c [addi] - 32, 3 - 85, 3 - 41, 1 - 39, 3 | |
| Cmd#:14, $1:8000004, $2:a, $3:1, $6:1,<br>0x20 [stall] - 0, 0 - 27, 1 - 9, 3 - 7, 3 | Stall διότι ακολουθεί άλμα βασισμένο στην τιμή του n η οποία μόλις άλλαξε. |
| Cmd#:15, $1:8000004, $2:a, $3:1, $6:1,<br>0x20 [beq] - 64, 1 - 80, 3 - 3, 3 - 1, 1 | Άλμα υπό συνθήκη. Δεν πραγματοποιείται στην περίπτωσή μας. |
| Cmd#:16, $1:8000004, $2:a, $3:1, $6:1,<br>0x24 [nop] - 17, 3 - 27, 3 - 9, 1 - 7, 1 | Εντολή nop αμέσως μετά το άλμα. |
| Cmd#:17, $1:8000004, $2:a, $3:1, $6:1,<br>0x28 [j] - 26, 3 - 70, 1 - 3, 1 - 1, 1 | Άλμα στην αρχή του βρόχου του προγράμματος για τον υπολογισμό του επόμενου αριθμού. |
| Cmd#:18, $1:8000004, $2:a, $3:1, $6:1,<br>0x2c [nop] - 17, 1 - 27, 1 - 9, 1 - 7, 1 | |
| Cmd#:19, $1:8000008, $2:a, $3:1, $6:1,<br>0x8 [addi] - 32, 1 - 85, 1 - 41, 1 - 39, 2 | |
| Cmd#:20, $1:8000008, $2:a, $3:1, $5:1, $6:1,<br>0xc [lw] - 32, 1 - 87, 1 - 43, 2 - 39, 2 | |
| Cmd#:21, $1:8000008, $2:a, $3:1, $5:1, $6:1,<br>0x10 [lw] - 32, 1 - 87, 2 - 43, 2 - 39, 1 | |
| Cmd#:22, $1:8000008, $2:a, $3:1, $5:1, $6:1,<br>0x14 [stall] - 0, 0 - 27, 2 - 9, 1 - 7, 1 | |
| Cmd#:23, $1:8000008, $2:a, $3:1, $5:1, $6:2,<br>0x14 [R instr] - 28, 2 - 96, 1 - 41, 1 - 39, 1 | |

**Πίνακας 17: Επεξήγηση μέρους πειράματος αριθμών Fibonacci.**

Από την εντολή #18 και πέρα, επαναλαμβάνεται ο βρόχος του προγράμματος. Μπορούμε να δούμε πως ο χρόνος παραμονής κάθε εντολής σε κάθε στάδιο της διοχέτευσης είναι πλέον 1 κύκλος μιας και έχουν όλες διαβαστεί στη κρυφή μνήμη εντολών οπότε δεν έχουμε αστοχίες ανάγνωσης στη συγκεκριμένη μνήμη. Επίσης, επειδή όλες οι διευθύνσεις της μνήμης δεδομένων που διαβάζουμε στο μέλλον έχουν ήδη γραφτεί από το ίδιο το πρόγραμμα κατά το προηγούμενο πέρασμα του βρόχου, ούτε σε αναγνώσεις μελλοντικών θέσεων έχουμε κάποια καθυστέρηση. Επομένως, όπως βλέπουμε, καθυστερήσεις έχουμε μόνο στην περίπτωση κάποιου stall.

Στη συνέχεια πήραμε το αρχείο καταγραφής δεδομένων σχετικά με τον AVF και το επεξεργαστήκαμε στο MATLAB. Οι τιμές του AVF είναι οι παρακάτω:

| Δομή | AVF |
|---|---|
| IF/ID | 47,64% |
| ID/EX | 42,39% |
| EX/MEM | 26,72% |
| MEM/WB | 31,07% |
| Διοχέτευση | 36,65% |

**Πίνακας 18: Τιμές AVF του πειράματος των 12 πρώτων αριθμών Fibonacci.**

Τα παραπάνω αποτελέσματα είναι και τα αναμενόμενα. Στο συγκεκριμένο πρόγραμμα οι εντολές άλματος είναι λίγες σε σχέση με τον υπόλοιπο κώδικα, επομένως ο AVF του καταχωρητή IF/ID είναι λογικός. Το γεγονός πως είναι μικρότερος του 50% οφείλεται στο γεγονός ύπαρξης εντολών nop καθώς και stalls. Επίσης λογικός είναι και ο AVF του καταχωρητή ID/EX μιας και σχεδόν όλες οι εντολές χρησιμοποιούν μόνο δύο από τα πεδία NPC, Data Read 1, Data Read 2 και Immediate. Το αποτέλεσμα για τον καταχωρητή EX/MEM έχει ενδιαφέρον μιας και οι περισσότερες εντολές χρησιμοποιούν το 50% του καταχωρητή αλλά πρέπει να δοθεί προσοχή στο ότι από τις 9 εντολές του κύριου βρόχου, οι 3 είναι εντολές οι οποίες δεν χρησιμοποιούν ούτε το πεδίο ALU result ούτε το πεδίο Read Data 2. Μετά από αυτή την παρατήρηση, γίνεται εμφανές πως το αποτέλεσμα θα έπρεπε να είναι αναμενόμενο. Το ίδιο ισχύει και για το αποτέλεσμα του καταχωρητή MEM/WB. Πρέπει να παρατηρήσει κανείς πως οι εντολές που εκτελούνται και δεν αποθηκεύουν το αποτέλεσμά τους σε κάποιον καταχωρητή είναι αρκετές για να δικαιολογούν την τιμή του AVF που είναι πολύ μικρότερη από 50%.

Από το αποτέλεσμα επιβεβαιώνεται η πρόταση πως είναι σημαντικό να γίνεται σωστή μελέτη της ευαισθησίας κάθε δομής πριν παρθεί κάποια απόφαση σχετικά με το ποιες δομές πρέπει να προστατευτούν και με ποιο τρόπο.

Έπειτα, εφόσον επιβεβαιώσαμε πως το πρόγραμμα και ο προσομοιωτής λειτουργεί σωστά, τρέξαμε το πρόγραμμα για το μεγαλύτερο δυνατό αριθμό επαναλήψεων, ώστε ο τελευταίος αριθμός Fibonacci να μπορεί να χωράει στα 32 bit της μίας λέξης. Εκτελέσαμε λοιπόν το πρόγραμμα για τους πρώτους 48 αριθμούς Fibonacci. Συνολικά χρειάστηκε 557 κύκλους ρολογιού και πήραμε τα εξής αποτελέσματα αναφορικά με τον AVF.

| **Δομή** | **AVF** |
|---|---|
| IF/ID | 48,13% |
| ID/EX | 42,09% |
| EX/MEM | 26,82% |
| MEM/WB | 30,70% |
| Διοχέτευση | 36,54% |

**Πίνακας 19: Τιμές AVF του πειράματος των 48 πρώτων αριθμών Fibonacci.**

Παρατηρούμε πως υπάρχουν κάποιες μικρές διαφορές στον AVF οι οποίες οφείλονται στα διαφορετικά δεδομένα εισόδου και εξόδου αλλά δεν πρόκειται για κάτι το αξιοσημείωτο μιας και στην ουσία αλλάζει μόνο ο αριθμός των επαναλήψεων και όχι οι αναλογίες των εντολών που εκτελέστηκαν.

### ~~5.3.2~~ 5.3.2 Αλγόριθμός Selection Sort [15]

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Ο αλγόριθμος Selection Sort είναι ένας αλγόριθμος ο οποίος ταξινομεί σε αύξουσα σειρά μία λίστα αριθμών. Χρησιμοποιεί τρεις δείκτες. Έναν ο οποίος δείχνει το πρώτο στοιχείο της λίστας προς ταξινόμηση, έναν ο οποίος δείχνει το τρέχον στοιχείο και έναν ο οποίος δείχνει το μικρότερο από τα στοιχεία που έχουμε ελέγξει. Στην αρχή κάθε περάσματος, και οι δείκτες του μικρότερου και του πρώτου στοιχείου δείχνουν το 1ο στοιχείο της λίστας ενώ ο δείκτης του τρέχοντος δείχνει το 2ο στοιχείο. Σε κάθε πέρασμα, ο δείκτης του τρέχοντος στοιχείου προχωράει ένα τα στοιχεία από το 1ο έως το τελευταίο. Κάθε φορά γίνεται έλεγχος για εάν το τρέχον στοιχείο είναι μικρότερο από αυτό που δείχνει ο δείκτης του μικρότερου στοιχείου. Εάν ισχύει αυτή η συνθήκη, τότε ο δείκτης του μικρότερου στοιχείου αλλάζει ώστε να δείχνει το τρέχον στοιχείο. Ο δείκτης του τρέχοντος στοιχείου προχωράει έως ότου φτάσει στο τέλος της λίστας. Στο τέλος του περάσματος, ανταλλάσσουν θέση τα περιεχόμενα των δεικτών του 1ου και του μικρότερου στοιχείου, ώστε το μικρότερο στοιχείο να βρεθεί στη χαμηλότερη θέση της λίστας. Έπειτα, ο δείκτης του 1ου στοιχείου προχωράει κατά μία θέση, ο δείκτης του μικρότερου στοιχείου δείχνει την ίδια θέση με τον δείκτη του 1ου και ο δείκτης του τρέχοντος στοιχείου δείχνει μία θέση μετά από τους δύο άλλους δείκτες. Ο αλγόριθμος εκτελείται έως ότου ο δείκτης του 1ου στοιχείου δείχνει το n-1 στοιχείο της λίστας, όπου γίνεται και ο τελευταίος έλεγχος μεταξύ των δύο τελευταίων στοιχείων.

***Ψευδοκώδικας Αλγορίθμου Selection Sort***

```
for i ← 0 to n-2 do
        min ← i
        for j ← (i + 1) to n-1 do
           if A[j] < A[min]
               min ← j
        swap A[i] and A[min]
```

Οι τρεις δείκτες στον παραπάνω ψευδοκώδικα είναι οι A[i], A[j] και A[min]. Ο A[i] δείχνει το 1ο στοιχείο από το "παράθυρο" μέσα στο οποίο κινούμαστε, ο A[j] δείχνει το τρέχον στοιχείο και ο A[min] δείχνει το μικρότερο στοιχείο. Ο πίνακας A είναι η λίστα με τα δεδομένα προς ταξινόμηση. Επειδή η μετατροπή του παραπάνω ψευδοκώδικα σε assembly δεν είναι προφανής, παρουσιάζουμε την μετατροπή του σε γλώσσα C γραμμένο έτσι ώστε να μπορεί να μετατραπεί εύκολα σε γλώσσα assembly.

Οι μεταβλητές pStart, pMin και pCurrent είναι οι τρεις δείκτες και όπως υπονοούν τα ονόματά τους, ο pStart δείχνει την αρχή του παραθύρου, ο pCurrent δείχνει το τρέχον στοιχείο και ο pMin δείχνει το μικρότερο στοιχείο. Ο δείκτης pEnd χρειάζεται για να ξέρουμε πότε φτάνουμε στο τελευταίο στοιχείο της λίστας. Επειδή η μνήμη στον MIPS είναι byte addressable, κάθε φορά που θέλουμε να δείξουμε σε κάποιο επόμενο στοιχείο, αυξάνουμε την τιμή του κάθε δείκτη κατά 4.

```
Init:       pStart = 0x080000
            pEnd = pStart + 28
Start:      if ( pStart == pEnd ) { goto End }
            pCurrent = pStart + 4
            pMin = pStart
Loop:       Min = *pMin
            Current = *pCurrent
            comp = ( Current < Min ? 1 : 0 )
            if (comp == 0) {
                    goto Point1
            } else {
                    pMin = pCurrent
            }
Point1:     comp = 0
            if (pCurrent != pEnd) {
                    goto Point2
            } else {
                    call swap
                    pStart += 4
                    goto Start
            }
Point2:     pCurrent += 4
            goto Loop
Swap:       temp = *pStart
            Min = *pMin
            *pStart = Min
            *pMin = temp
            return
```

End: nop

Ο παραπάνω κώδικας είναι πλέον εύκολο να μετατραπεί σε γλώσσα assembly.

```
3c010800    // lui $1, $0, 0x0800
20220018    // addi $2, $1, 0x0018
10220019    // beq $1, $2, End/
00000000    // nop
20240004    // addi $4, $1, 0x0004
00011820    // add $3, $0, $1
8c650000    // lw $5, 0x0000($3)
8c860000    // lw $6, 0x0000($4)
00c5382a    // slt $7, $6, $5
10070002    // beq $0, $7, Point_1
00000000    // nop
00041820    // add $3, $0, $4
00003820    // add $7, $0,$0
14820006    // bne $4, $2, Point_2
00000000    // nop
0c000017    // jal swap
00000000    // nop
20210004    // addi $1, $1, 0x0004
08000002    // j AdvanceStart
00000000    // nop
20840004    // addi $4, $4, 0x0004
08000006    // j InnerLoop
00000000    // nop
8c280000    // lw $8, 0x0000($1)
8c650000    // lw $5, 0x0000($3)
ac250000    // sw $5, 0x0000($1)
ac680000    // sw $8, 0x0000($3)
03e00008    // jr $31
00000000    // nop
```

Στον παραπάνω κώδικα, χρησιμοποιούμε τους εξής καταχωρητές:

- \$1 – Δείκτης αρχής του τρέχοντος παράθυρου.
- \$2 – Δείκτης τέλους της λίστας προς ταξινόμηση.
- \$3 – Δείκτης ελάχιστου στοιχείου.
- \$4 – Δείκτης τρέχοντος στοιχείου.
- \$5 – Τιμή ελάχιστου στοιχείου.
- \$6 – Τιμή τρέχοντος στοιχείου.
- \$7 – Αποτέλεσμα σύγκρισης \$6 < \$5
- \$8 – Καταχωρητής που χρησιμοποιείται για την εναλλαγή των τιμών του τρέχοντος με το ελάχιστο στοιχείο.

Εκτελέσαμε το πρόγραμμα με τρία διαφορετικά σύνολα δεδομένων εισόδου. Η πρώτη εκτέλεση περιείχε αριθμούς ταξινομημένους με αύξουσα σειρά, η δεύτερη εκτέλεση περιείχε τους ίδιους αριθμούς αλλά ταξινομημένους με φθίνουσα σειρά και η τρίτη εκτέλεση περιείχε τυχαίους αριθμούς, ίδιους όμως στο πλήθος με τα προηγούμενα δύο σύνολα.

***Σύνολο 1:***

00000000
00000001
00000002
00000004
00000008
0000000f
00000010
00000020
00000040
00000080
000000f0
000000ff
00000100
00000200
00000400
00000800
00000f00
00000fff
00001000
00002000
00004000
00008000
0000f000
0000ffff
00010000
00020000
00040000
00080000
000f0000
000fffff
00100000
00200000
00400000
00800000
00f00000
00ffffff
01000000
02000000
04000000

08000000
0f000000
0fffffff
10000000
20000000
40000000

***Σύνολο 2:***

40000000
20000000
10000000
0fffffff
0f000000
08000000
04000000
02000000
01000000
00ffffff
00f00000
00800000
00400000
00200000
00100000
000fffff
000f0000
00080000
00040000
00020000
00010000
0000ffff
0000f000
00008000
00004000
00002000
00001000
00000fff
00000f00
00000800
00000400
00000200
00000100
000000ff
000000f0
00000080
00000040

00000020
00000010
0000000f
00000008
00000004
00000002
00000001
00000000

***Σύνολο 3:***

00000010
00000031
0001025a
0000b110
0010ff01
502402ff
50240300
0000000f
0000ffff
00000101
10cbda56
bf48b0c1
5d6b4f08
641cc564
0cb6d8f0
1c52604d
504f86a5
0c1b68fd
0c156d8f
404a6df5
410c26a1
4f8bc504
b2a63dc4
0b56c4b0
56f74056
e45f6e04
56a2406a
2d40d2c6
2dfb0e86
a45fcve0
00000020
00000040
00000080
00000008
0000000f

d0c8af60
4b2c6a4d
0b2c6d0b
00000010
00000020
00000040
0c1b68fd
0c156d8f
404a6df5
410c26a1

Μετά το πέρας της εκτέλεσης ελέγχουμε το αρχείο καταγραφής των περιεχομένων της μνήμης δεδομένων και παίρνουμε τις παρακάτω τιμές, οι οποίες είναι όντως τα προηγούμενα δεδομένα ταξινομημένα με την επιθυμητή σειρά.

***Αποτελέσματα συνόλων 1 και 2:***

08000000 - 00000000
08000004 - 00000001
08000008 - 00000002
0800000c - 00000004
08000010 - 00000008
08000014 - 0000000f
08000018 - 00000010
0800001c - 00000020
08000020 - 00000040
08000024 - 00000080
08000028 - 000000f0
0800002c - 000000ff
08000030 - 00000100
08000034 - 00000200
08000038 - 00000400
0800003c - 00000800
08000040 - 00000f00
08000044 - 00000fff
08000048 - 00001000
0800004c - 00002000
08000050 - 00004000
08000054 - 00008000
08000058 - 0000f000
0800005c - 0000ffff
08000060 - 00010000
08000064 - 00020000
08000068 - 00040000
0800006c - 00080000
08000070 - 000f0000
08000074 - 000fffff

08000078 - 00100000
0800007c - 00200000
08000080 - 00400000
08000084 - 00800000
08000088 - 00f00000
0800008c - 00ffffff
08000090 - 01000000
08000094 - 02000000
08000098 - 04000000
0800009c - 08000000
080000a0 - 0f000000
080000a4 - 0fffffff
080000a8 - 10000000
080000ac - 20000000
080000b0 - 40000000

***Αποτέλεσμα συνόλου 3:***

08000000 - b2a63dc4
08000004 - bf48b0c1
08000008 - d0c8af60
0800000c - e45f6e04
08000010 - 00000008
08000014 - 0000000f
08000018 - 0000000f
0800001c - 00000010
08000020 - 00000010
08000024 - 00000020
08000028 - 00000020
0800002c - 00000031
08000030 - 00000040
08000034 - 00000040
08000038 - 00000080
0800003c - 00000101
08000040 - 0000b110
08000044 - 0000ffff
08000048 - 0001025a
0800004c - 000a45fc
08000050 - 0010ff01
08000054 - 0b2c6d0b
08000058 - 0b56c4b0
0800005c - 0c156d8f
08000060 - 0c156d8f
08000064 - 0c1b68fd
08000068 - 0c1b68fd
0800006c - 0cb6d8f0

08000070 - 10cbda56
08000074 - 1c52604d
08000078 - 2d40d2c6
0800007c - 2dfb0e86
08000080 - 404a6df5
08000084 - 404a6df5
08000088 - 410c26a1
0800008c - 410c26a1
08000090 - 4b2c6a4d
08000094 - 4f8bc504
08000098 - 502402ff
0800009c - 50240300
080000a0 - 504f86a5
080000a4 - 56a2406a
080000a8 - 56f74056
080000ac - 5d6b4f08
080000b0 - 641cc564

Στα αποτελέσματα του συνόλου 3, ο χρήστης πρέπει να προσέξει πως οι πρώτοι 4 αριθμοί είναι αρνητικοί αριθμοί και για αυτό κατέχουν τις εκάστοτε θέσεις.

Οι χρόνοι εκτέλεσης ήταν 13.452 κύκλοι για το σύνολο 1, 13.958 για το σύνολο 2 και 13.567 για το σύνολο 3.

Τέλος, πήραμε τα δεδομένα του αρχείου καταγραφής πληροφορίας σχετική με τον AVF και τα επεξεργαστήκαμε με το πρόγραμμα MATLAB. Τα αποτελέσματα ήταν τα παρακάτω:

| Δομή | AVF |
|---|---|
| IF/ID | 49,85% |
| ID/EX | 39,61% |
| EX/MEM | 21,86% |
| MEM/WB | 26,28% |
| Διοχέτευση | 33,60% |

**Πίνακας 20: Τιμές AVF πειράματος selection sort συνόλου 1.**

| Δομή | AVF |
|---|---|
| IF/ID | 49,61% |
| ID/EX | 40,28% |
| EX/MEM | 22,37% |
| MEM/WB | 27,19% |
| Διοχέτευση | 34,16% |

**Πίνακας 21: Τιμές AVF πειράματος selection sort συνόλου 2.**

| Δομή | AVF |
|---|---|
| IF/ID | 49,79% |
| ID/EX | 39,77% |

| Δομή | AVF |
|---|---|
| EX/MEM | 21,98% |
| MEM/WB | 26,50% |
| Διοχέτευση | 33,73% |

**Πίνακας 22: Τιμές AVF πειράματος selection sort συνόλου 3.**

Αναλύοντας τα αποτελέσματα για την περίπτωση του αλγορίθμου selection sort, κάποιος μπορεί να εξάγει τα ίδια συμπεράσματα με την περίπτωση του υπολογισμού των αριθμών Fibonacci. Παρατηρείται ελαφρώς μικρότερος AVF σε όλα τα στάδια πέραν του IF/ID. Η πτώση του AVF στα στάδια ID/EX, EX/MEM, MEM/WB οφείλεται πιθανότατα στο γεγονός πως έχουμε περισσότερα stall και nop σε σχέση με τους αριθμούς Fibonacci. Επίσης, επειδή προσπελαύνουμε τις ίδιες περιοχές μνήμης πολλές φορές, έχουμε πολύ λιγότερες αστοχίες στην κρυφή μνήμη οι οποίες αυξάνουν τον AVF εφόσον οι εντολές που υπάρχουν μέσα στη διοχέτευση την χρονική στιγμή του miss έχουν περισσότερα ACE bits από τον μέσο όρο του προγράμματος. Η ελαφρώς αυξημένη τιμή του AVF στον καταχωρητή IF/ID οφείλεται στο γεγονός πως έχουμε περισσότερες εντολές άλματος αναλογικά με τις υπόλοιπες εντολές που εκτελούνται αλλά εξακολουθεί να είναι κάτω του 50% λόγω των stalls και των εντολών nop.

### 5.3.3 Πολλαπλασιασμός ακεραίων με συνεχείς προσθέσεις

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

Σε ορισμένες περιπτώσεις χρειαζόμασταν την πράξη του πολλαπλασιασμού αλλά δεν την υλοποιούμε στο σύνολο εντολών του μοντέλου μας. Υλοποιήσαμε λοιπόν μία υπορουτίνα για αυτόν ακριβώς το λόγο η οποία δέχεται δύο ακέραιους αριθμούς και εκτελεί τον πολλαπλασιασμό ως εξής: χρησιμοποιεί έναν καταχωρητή ως αθροιστή και προσθέτει τον πρώτο τελεστέο τόσες φορές όσες και η τιμή του δεύτερου τελεστέου.

```
3c080800 // lui $0, $8, 0x0800
35080000 // ori $8, $8, 0x0000 -- t0 for 1st operand
21090004 // addi $8, $9, 0x0004 -- t1 for 2nd operand
212a0004 // addi $9, $10, 0x0004 -- t2 for result
8d040000 // lw $8, $4, 0x0000 -- 0x0005 to attribute 1
8d250000 // lw $9, $5, 0x0006 -- 0x0006 to attribute 2
0c00000b // jal mult -- call multiply function
00000000 // nop
ad420000 // sw $10, $2, 0x0000 -- store result to memory
08000014 // j 0x0019 -- End program
00000000 // nop
// --
// Multiply function
// $v0 ($2) as accumulator for result
// $a0 ($4) and $a1 ($5) for operands. $a1 MUST be positive.
// -- start --
00001020 // add $0, $0, $2 -- acc = 0
14a00003 // bne $5, $0, 0x0003 -- if b!=0, skip return
00000000 // nop
03e00008 // jr $31 -- return
00000000 // nop
00441020 // add $2, $4, $2 -- acc += a
20a5ffff // addi $5, $5, 0xffff -- b -= 1
```

```
1000fff9 // beq $0, $0, 0xfff9
00000000 // nop
// -- end --
00000000 // nop -- end of program
```

Στην πρώτη εκτέλεση, ο πρώτος τελεστέος ήταν ο αριθμός 0χ00018421 και ο δεύτερος ήταν ο 0χ0003ff. Χρειάστηκαν 98.319 κύκλοι ρολογιού για την ολοκλήρωση του προγράμματος. Τα αποτελέσματα σχετικά με τον AVF ήταν τα παρακάτω:

| Δομή | AVF |
|---|---|
| IF/ID | 57,80% |
| ID/EX | 41,67% |
| EX/MEM | 16,36% |
| MEM/WB | 22,07% |
| Διοχέτευση | 33,96% |

**Πίνακας 23: AVF πολλαπλασιασμού πολλών επαναλήψεων.**

Στην δεύτερη εκτέλεση, ο πρώτος τελεστέος ήταν ο αριθμός 0χ00018421 και ο δεύτερος ήταν ο 0χ0000ff. Χρειάστηκαν 1.551 κύκλοι ρολογιού για την ολοκλήρωση του προγράμματος. Τα αποτελέσματα σχετικά με τον AVF ήταν τα παρακάτω:

| Δομή | AVF |
|---|---|
| IF/ID | 57,30% |
| ID/EX | 41,80% |
| EX/MEM | 16,62% |
| MEM/WB | 22,21% |
| Διοχέτευση | 34,02% |

**Πίνακας 24: AVF πολλαπλασιασμού λίγων επαναλήψεων .**

Αξίζει να σημειωθεί πως ο AVF του καταχωρητή IF/ID είναι μεγαλύτερος του 50%, πράγμα το οποίο οφείλεται στο γεγονός πως το συγκεκριμένο πρόγραμμα έχει αρκετά μεγάλο αριθμό εντολών διακλάδωσης και άλματος σε σχέση με τα υπόλοιπα πειράματα που εκτελέσαμε.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

### ~~5.3.4~~ 5.3.4 Υπολογισμός n!

Τέλος, χρησιμοποιήσαμε την υπορουτίνα πολλαπλασιασμού για να υπολογίσουμε το n! διαφόρων αριθμών. Ο κώδικας που χρησιμοποιήσαμε ήταν ο παρακάτω (αλλάζαμε μόνο τα δεδομένα στην διεύθυνση 0χ08000000 τα οποία όριζαν το n):

```
data
0000000c // starting at address 0x08000000, little endian
text
// Compute the factorial of the word in address 0x08000000
// Save it to address 0x08000004
// $s0 ($16) is n
// $s1 ($17) is tempresult
// $t0 ($8) is &n
// $t1 ($9) is &result
//
```

```
// -- start --
3c080800 // lui $0, $8, 0x0800
35080000 // ori $8, $8, 0x0000
21090004 // addi $8, $9, 0x0004
8d100000 // lw $8, $16, 0x0000 -- load n
00108820 // add $0, $16, $17 -- set n as tempresult
// -- start main loop
2210ffff // addi $16, $16, 0xffff -- decrease n by 1
12000006 // beq $16, $0, 0x0006 -- if n == 0, exit loop
00112020 // add $0, $17, $4 -- tempresult to attribute 1
00102820 // add $0, $16, $5 -- decreased n to attribute 2
0c000010 // jal mult -- call multiply function
00000000 // nop
08000005 // j 0x0005
00028820 // add $0, $2, $17 -- store returned value to tempresult
ad310000 // sw $9, $17, 0x0000 -- store result to memory
08000019 // j 0x0019 -- End program
00000000 // nop
// --
// Multiply function
// $v0 ($2) as accumulator for result
// $a0 ($4) and $a1 ($5) for operands. $a1 MUST be positive.
// -- start --
00001020 // add $0, $0, $2 -- acc = 0
14a00003 // bne $5, $0, 0x0003 -- if b!=0, skip return
00000000 // nop
03e00008 // jr $31 -- return
00000000 // nop
00441020 // add $2, $4, $2 -- acc += a
20a5ffff // addi $5, $5, 0xffff -- b -= 1
1000fff9 // beq $0, $0, 0xfff9
00000000 // nop
// -- end --
00000000 // nop -- end of program
```

Η μεγαλύτερη τιμή του n που μπορούσαμε να ορίσουμε προκειμένου να μπορεί το n! να χωρέσει στα 32 bit που είναι το μέγεθος μίας λέξης, ήταν η 0x0000000c, δηλαδή το 12!. Για αυτήν την τιμή, χρειάστηκαν 568 κύκλοι ρολογιού και το 12! βρέθηκε να είναι ίσο με 0χ1c8cfc00. O AVF είχε τις εξής τιμές:

| **Δομή** | **AVF** |
|---|---|
| IF/ID | 53,78% |
| ID/EX | 43,56% |
| EX/MEM | 17,75% |
| MEM/WB | 23,72% |
| Διοχέτευση | 34,65% |

**Πίνακας 25: AVF κατά τον υπολογισμό του 12!.**

Στη συνέχεια υπολογίσαμε το 4!. Χρειάστηκαν μόλις 96 κύκλοι ρολογιού και βρέθηκε να είναι ίσο με 0χ00000018. Ο AVF είχε τις εξής τιμές:

| Δομή | AVF |
|---|---|
| IF/ID | 48,76% |
| ID/EX | 45,46% |
| EX/MEM | 19,90% |
| MEM/WB | 25,50% |
| Διοχέτευση | 35,44% |

**Πίνακας 26: AVF κατά τον υπολογισμό του 4!.**

Μορφοποιημένο: Κουκκίδες και αρίθμηση

## ~~5.4~~ 5.4 ΣΥΓΚΕΝΤΡΩΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

Στον παρακάτω πίνακα συγκεντρώνουμε τα αποτελέσματα των πειραμάτων.

| | 12 Fib. | 48 Fib. | Selsort αύξ. | Selsort φθίν. | Selsort τυχαία | Πολ/μος μεγάλος | Πολ/μος μικρός | 12! | 4! |
|---|---|---|---|---|---|---|---|---|---|
| IF/ID | 47,64% | 48,13% | 49,85% | 49,61% | 49,79% | 57,80% | 57,30% | 53,78% | 48,76% |
| ID/EX | 42,39% | 42,09% | 39,61% | 40,28% | 39,77% | 41,67% | 41,80% | 43,56% | 45,46% |
| EX/MEM | 26,72% | 26,82% | 21,86% | 22,37% | 21,98% | 16,36% | 16,62% | 17,75% | 19,90% |
| MEM/WB | 31,07% | 30,70% | 26,28% | 27,19% | 26,50% | 22,07% | 22,21% | 23,72% | 25,50% |
| Διοχέτευση | 36,65% | 36,54% | 33,60% | 34,16% | 33,73% | 33,96% | 34,02% | 34,65% | 35,44% |
| Εντολές που εκτελέστηκαν | 125 | 557 | 13.452 | 13.958 | 13.567 | 98.319 | 1.551 | 568 | 96 |

**Πίνακας 27: Συγκέντρωση αποτελεσμάτων AVF.**

Εκτελέσαμε ένα πλήθος προγραμμάτων τα οποία χρησιμοποιούν όλο το υποστηριζόμενο σύνολο εντολών και το κάθε ένα είχε αρκετά διαφορετικό χρόνο εκτέλεσης. Οι διαφορές στον AVF από πρόγραμμα σε πρόγραμμα για κάθε δομή φτάνουν μέχρι και το 10%. Για παράδειγμα, στην μονάδα EX/MEM η μικρότερη τιμή παρατηρήθηκε στο πρόγραμμα του πολλαπλασιασμού(16,36%) ενώ η μεγαλύτερη στον υπολογισμό των αριθμών Fibonacci (26,82%) ενώ τα δύο προγράμματα είχαν τελείως διαφορετικούς AVF στη μονάδα IF/ID. Συγκεκριμένα ήταν 48,13% για τους 48 αριθμούς Fibonacci ενώ ο μεγάλος πολλαπλασιασμό άγγιζε το 57,80%. Επίσης μικρές διαφοροποιήσεις στον AVF παρατηρήθηκαν ακόμα και για διαφορετικά σύνολα δεδομένων στο ίδιο πρόγραμμα. Αν και οι διαφορές αυτές ήταν μικρές για τα προγράμματα που εκτελέσαμε, είναι μία καλή ένδειξη πως για πολυπλοκότερες εφαρμογές τα δεδομένα εισόδου στον επεξεργαστή μπορεί να έχουν αισθητή επίδραση στον AVF.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

# ~~6~~ 6 Συμπεράσματα

## ~~6.1~~ 6.1 ΔΥΣΚΟΛΙΕΣ

Η μεγαλύτερη δυσκολία στον υπολογισμό του AVF ήταν η έλλειψη εξειδικευμένων και ανοιχτού κώδικα πακέτων λογισμικού. Όσα πακέτα υπήρχαν ήταν είτε εμπορικά, είτε συναντήσαμε σημαντικά προβλήματα κατά την εγκατάσταση και εκτέλεση. Για αυτούς τους λόγους χρειάστηκε να αναπτυχθεί από μηδενική βάση μία καινούρια μέθοδος προγραμματισμού μοντέλων επεξεργαστών καθώς και κώδικας που θα χρησιμοποιεί το εκάστοτε μοντέλο προκειμένου να συλλέγουν τα απαραίτητα δεδομένα για τον υπολογισμό του AVF.

Μία ακόμα δυσκολία που συναντήσαμε είναι πως επειδή το μοντέλο που αναπτύχθηκε είναι μοναδικό, δεν υπάρχουν εργαλεία ανάπτυξης λογισμικού για αυτό όπως κάποιος μεταγλωττιστής C και assembler. Επόμενο ήταν να χρειαστεί να γράψουμε τα δοκιμαστικά προγράμματα των πειραμάτων σε γλώσσα μηχανής στο χέρι. Αναπτύξαμε μεν μία βοηθητική εφαρμογή η οποία αν διαθέταμε περισσότερο χρόνο θα μπορούσε να μετατραπεί και σε πλήρη assembler αλλά δεν ήταν αυτό το αντικείμενο της εργασίας. Αυτό όμως το πρόβλημα είναι αναμενόμενο μιας και υποτίθεται πως η συγκεκριμένη εργασία απευθύνεται σε σχεδιαστές νέων ολοκληρωμένων κυκλωμάτων όπου έτσι και αλλιώς θα χρειάζονται νέα εργαλεία ανάπτυξης του λογισμικού τους, εκτός φυσικά και αν ο σχεδιαστής φτιάξει μοντέλο ενός επεξεργαστή του οποίου τα εργαλεία ήδη υπάρχουν.

Παρά τις παραπάνω δυσκολίες, πιστεύουμε πως η μέθοδος που ακολουθήσαμε είναι χρήσιμη και πρακτική για τον υπολογισμό του AVF οποιουδήποτε συστήματος και όχι μόνο επεξεργαστών, αρκεί το σύστημα να αποτελείται από κάποιες μονάδες επεξεργασίας στις οποίες παρεμβάλλονται κάποιες μονάδες αποθήκευσης, αν και αυτός ο περιορισμός μπορεί να αρθεί με λίγη εφευρετικότητα από πλευράς του σχεδιαστή.

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

## ~~6.2~~ 6.2 ΣΧΟΛΙΑ ΕΠΙ ΤΩΝ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

Στα διάφορα πειράματα που τρέξαμε, παρατηρήσαμε πως πράγματι, δεν είναι απαραίτητο να προστατευθούν όλα τα bit κάθε καταχωρητή του επεξεργαστή. Αυτό επιβεβαιώνει την σημασία του υπολογισμού του AVF πριν ληφθούν οι τελικές αποφάσεις για ένα σχέδιο ώστε να περιοριστεί το κόστος σχεδίασης των προστατευτικών δικλείδων που απαιτούνται για την αξιόπιστη λειτουργία του ολοκληρωμένου κυκλώματος ή όποιου άλλου συστήματος μελετάμε. Αποδεικτικό στοιχείο της παραπάνω δήλωσης είναι η διαφορά στον AVF μεταξύ των καταχωρητών IF/ID και ID/EX σε σχέση με τους καταχωρητές EX/MEM και MEM/WB, όπου στους πρώτους καταχωρητές ένα σφάλμα σε κάποιο bit έχει σχεδόν διπλάσια πιθανότητα να προκαλέσει λάθος στην εκτέλεση του προγράμματος σε σχέση με τους τελευταίους.

Επίσης ενδιαφέρον παρουσιάζει πως ο συνολικός AVF για το σύνολο της διοχέτευσης είναι σχετικά μικρός, περίπου 35%. Αυτό μας δείχνει πως αν ο επεξεργαστής αναμένεται να λειτουργεί σε κάποιο

περιβάλλον με μικρή ποσότητα ακτινοβολίας υποβάθρου και σε μη κρίσιμη εφαρμογή, μπορούμε να προστατεύσουμε μόνο κρίσιμες δομές όπως ο Program Counter χωρίς να έχουμε ιδιαίτερα προβλήματα κατά τη λειτουργία του συστήματος.

## 6.3 6.3 ΜΕΛΛΟΝΤΙΚΗ ΕΡΓΑΣΙΑ

Μορφοποιημένο: Κουκκίδες και αρίθμηση

Θα ήταν χρήσιμο για εκπαιδευτικούς λόγους αν αναπτυσσόταν ένα γραφικό περιβάλλον για την παρουσίαση της κατάστασης του μοντέλου και του προσομοιωτή. Αυτή τη στιγμή είναι δυνατή η προβολή πληροφορίας για την πορεία του εκάστοτε πειράματος αλλά η προβολή είναι περιορισμένη. Η σχεδίαση του λογισμικού και του μοντέλου δίνει την δυνατότητα πρόσβασης σε σχεδόν όλες τις παραμέτρους του μετά από κάθε κύκλο ρολογιού οπότε η ανάπτυξη ενός γραφικού περιβάλλοντος δεν θα επηρεάσει κανένα άλλο τμήμα του λογισμικού. Αυτός ήταν μάλιστα ένας από τους λόγους που κατά τη σχεδίαση και ανάπτυξη χρησιμοποιήσαμε το μοντέλο MVC (Model View Controller) [18].

Θα είχε ενδιαφέρον αν η μεθοδολογία που ακολουθήθηκε σε αυτή την εργασία εφαρμοζόταν στη σχεδίαση ενός μοντέλου και λογισμικού σε System C μιας και υπάρχουν ήδη αρκετά μοντέλα πραγματικών επεξεργαστών και ολοκληρωμένων κυκλωμάτων.

Επίσης, κάποιος θα μπορούσε να προσπαθήσει να μοντελοποιήσει κάποιο μεγαλύτερο υπολογιστικό σύστημα ώστε να εντοπίσει ποια υποσυστήματα του πρέπει να προστατευθούν. Θα μπορούσε κάποιος για παράδειγμα στη θέση της διοχέτευσης να βάλει μία διάταξη με συσκευές PLC ώστε να δει που χρειάζεται να εφαρμόσει τεχνικές όπως αυτή του πλεονασμού.

## 6.4 6.4 ΕΠΙΛΟΓΟΣ

Μορφοποιημένο: Κουκκίδες και αρίθμηση

Το θέμα του υπολογισμού του AVF αποδεικνύεται ένα ζήτημα με πολλές πτυχές και πολύ ενδιαφέρον τόσο από την πλευρά του σχεδιαστή, όσο και από την πλευρά του προγραμματιστή. Με το να υπολογίζει ένας σχεδιαστής τον AVF των σχεδίων του, εντοπίζει και μαθαίνει ποιες πρακτικές είναι επιρρεπείς σε σφάλματα και ποιες είναι ανθεκτικότερες. Επίσης καταφέρνει να δημιουργεί αξιόπιστα κυκλώματα και συστήματα χωρίς ιδιαίτερες θυσίες στην απόδοση ή στο κόστος. Ένας προγραμματιστής από την άλλη μπορεί να δει ποιες προγραμματιστικές πρακτικές είναι επιρρεπείς σε σφάλματα στην περίπτωση ενός μαλακού σφάλματος και ποιες όχι οπότε θα ξέρει τι να αποφύγει όταν αναπτύσσει μία εφαρμογή όπου κάποιο σφάλμα θα έχει σημαντικές συνέπειες.

Όπως αναφέρεται στις εργασίες [10] και [11], τα σφάλματα που οφείλονται στην κοσμική ακτινοβολία και στην ακτινοβολία από την συσκευασία των ολοκληρωμένων αναμένεται να αυξηθούν με γεωμετρική πρόοδο στις ερχόμενες γενεές επεξεργαστών οπότε η ανάγκη για έρευνα πάνω στο θέμα γίνετε όλο και πιο επιτακτική. Το πρόβλημα αυτή τη στιγμή είναι πως δεν υπάρχει κάποιος κοινός αποδεκτός και πρότυπος τρόπος για τον υπολογισμό της ευαισθησίας ενός σχεδίου και πιστεύουμε πως οι επόμενες προσπάθειες θα πρέπει να οδεύουν προς αυτή την κατεύθυνση. Για την ώρα, ελπίζουμε η συγκεκριμένη εργασία να διευκολύνει κάποιον ερευνητή στο έργο του ή τουλάχιστον, να δώσει κάποιες ιδέες προς τη σωστή κατεύθυνση.

Μορφοποιημένο: Κουκκίδες και αρίθμηση

# 7 7 Βιβλιογραφία


[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”, pp. 87-95

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”, pp. 185-193

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, (1996).
ISBN:0-471-95869-7.

[4] Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, (2000).
ISBN:0-471-60695-2.

[5] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, (2002).
ISBN:978-0321127426.

[6] Martin Fowler, *UML Distilled Third Edition: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley (2004).
ISBN:978-0-321-19368-1

[7] Gregor Hohpe, Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, (2003).
ISBN:0-321-20068-3.

[8] Eric T. Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. Head First Design Patterns. O'Reilly Media, (2004).
ISBN:0-596-00712-4.

[9] David A. Paterson, John L. Hennessy. “Computer Organisation and Design: The Hardware/Software Interface”. Morgan Kaufmann, 2009,
ISBN: 978-0-12-374493-7

[10] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, Todd Austin. “Measuring Architectural Vulnerability Factors”. IEEE Micro November 2003, pp. 70-75, 2003.

[11] Shubhendu S. Mukherjee, Christopher T. Weaver, Joel Emer, Steven K. Reinhardt, Todd Austin. “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor”. In 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03), pp. 29-40, San Diego, Ca, USA, December 2003

[12] Shubu Mukherjee, “Architecture Design for Soft Errors”, Morgan Kaufmann, 2008, ISBN: 978-0-12-369529-1

[13] Shubu Mukherjee, “Architecture Design for Soft Errors”, Morgan Kaufmann, 2008, ISBN: 978-0-12-369529-1, pp. 2-3

[14] Israel Koren and C. Mani Krishna, “Fault-Tolerant Systems”, Morgan Kaufmann, 2007, ISBN-13: 978-0-12-088525-1 / ISBN-10: 0-12-088525-5

[15] Donald Knuth, “The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition”. Addison-Wesley, 1997, ISBN: 0-201-89685-0, pp. 138-141

[16] Joel Emer, Pritpal Ahuja, Nathan Binkert, Eric Borch, Roger Espana, Toni Juan, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, “Asim: A Performance Model Framework”, IEEE Computer 35(2), pp. 68-76, Feb. 2002.

[17] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, “Quantitative System Performance”, Prentice Hall, p.p. 42-46, 1984.

[18] Steve Burbeck, “Applications Programming in Smalltalk-80™: How to use Model-View-Controller (MVC)”, University of Illinois at Urbana-Campaign, department of computer science, The UIUC Smalltalk Archive, http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

**Μορφοποιημένο:** Κουκκίδες και αρίθμηση

# 8 Ευρετήρια

## Ευρετήριο πινάκων

Πίνακας 1: Εντολές τύπου R που υποστηρίζει το μοντέλο μας. ........ 20
Πίνακας 2: Εντολές τύπου I που υποστηρίζει το μοντέλο μας. ........ 22
Πίνακας 3: Εντολές τύπου J που υποστηρίζει το μοντέλο μας. ........ 23
Πίνακας 4: Κωδικοποίηση εντολών (opcodes). ........ 30
Πίνακας 5: Bit ελέγχου ALU. ........ 32
Πίνακας 6: ALUOp bits. ........ 32
Πίνακας 7: Bit ελέγχου σταδίου εκτέλεσης. ........ 33
Πίνακας 8: Bit ελέγχου σταδίου προσπέλασης μνήμης. ........ 35
Πίνακας 9: Bit ελέγχου σταδίου επανεγγραφής. ........ 36
Πίνακας 10: Bit ελέγχου της διοχέτευσης. ........ 37
Πίνακας 11: ACE bits ελέγχου. ........ 38
Πίνακας 12: ACE bits καταχωρητή IF/ID. ........ 44
Πίνακας 13: ACE bits καταχωρητή ID/EX. ........ 51
Πίνακας 14: ACE bits καταχωρητή EX/MEM. ........ 57
Πίνακας 15: ACE bits καταχωρητή MEM/WB. ........ 62
Πίνακας 16: ACE bits καταχωρητών pipeline. ........ 63
Πίνακας 17: Επεξήγηση μέρους πειράματος αριθμών Fibonacci. ........ 77
Πίνακας 18: Τιμές AVF του πειράματος των 12 πρώτων αριθμών Fibonacci. ........ 77
Πίνακας 19: Τιμές AVF του πειράματος των 48 πρώτων αριθμών Fibonacci. ........ 78
Πίνακας 20: Τιμές AVF πειράματος selection sort συνόλου 1. ........ 86
Πίνακας 21: Τιμές AVF πειράματος selection sort συνόλου 2. ........ 86
Πίνακας 22: Τιμές AVF πειράματος selection sort συνόλου 3. ........ 87
Πίνακας 23: AVF πολλαπλασιασμού πολλών επαναλύψεων. ........ 88
Πίνακας 24: AVF πολλαπλασιασμού λίγων επαναλύψεων ........ 88
Πίνακας 25: AVF κατά τον υπολογισμό του 12!. ........ 90
Πίνακας 26: AVF κατά τον υπολογισμό του 4!. ........ 90
Πίνακας 27: Συγκέντρωση αποτελεσμάτων AVF. ........ 90

## Ευρετήριο εικόνων

Εικόνα 1: Εκτέλεση τριών εντολών σε διοχέτευση τριών σταδίων........ 24
Εικόνα 2: Εκτέλεση τριών εντολών χωρίς διοχέτευση........ 25
Εικόνα 3: UML διάγραμμα κλάσεων του μοντέλου του pipeline........ 71
Εικόνα 4: UML διάγραμμα αντικειμένων του μοντέλου που μελετάμε........ 72