**Εικόνα που περιέχει κείμενο, γραμματοσειρά, λογότυπο, έμβλημα

Περιγραφή που δημιουργήθηκε αυτόματα**

Πολυτεχνική Σχολή

Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών & Πληροφορικής

Διπλωματική Εργασία

Υλοποίηση του RISC-V με την τεχνική των μερικώς επικαλυπτόμενων λειτουργιών

Γιαννάκης Εμμανουήλ Δημήτριος

A.M. : 1067491

Επιβλέπων

Νικολός Δημήτριος, Καθηγητής

Μέλη Εξεταστικής Επιτροπής

Βέργος Χαρίδημος, Καθηγητής

Ζερβάκης Γεώργιος, Επίκουρος Καθηγητής

Ευχαριστίες

Θέλω να ευχαριστήσω τους ανθρώπους που με στήριξαν σε όλη τη διάρκεια των σπουδών μου. Χωρίς αυτούς η υλοποίηση αυτής της εργασίας δεν θα ήταν εφικτή.

Περίληψη

Στην παρούσα διπλωματική εργασία υλοποιείται ένα μοντέλο ενός επεξεργαστή με την τεχνική των μερικώς επικαλυπτόμενων λειτουργιών που βασίζεται στην αρχιτεκτονική RISC-V. Για την υλοποίηση επιλέχθηκε η γλώσσα περιγραφής υλικού Verilog. Για την επιβεβαίωση της σωστής λειτουργίας του μοντέλου έγιναν εξομοιώσεις μεμονωμένα για κάθε εντολή που μπορεί να εκτελέσει το μοντέλο καθώς και ακολουθίες εντολών οι οποίες περιείχαν όλες τις εξαρτήσεις που μπορούν να προκύψουν. Επιπλέον, ο σχεδιασμός φορτώθηκε σε FPGA και διαπιστώθηκε πως ολοκληρώνονται επιτυχώς οι διαδικασίες Synthesis και Implementation.

Abstract

This thesis implements a model of a processor with the pipelining technique based on the RISC-V architecture. The hardware description language Verilog was chosen for the implementation. To confirm the correct operation of the model, simulations were performed individually for each instruction that can be executed as well as instruction sequences that contained all the hazards that can arise. In addition, the design was loaded on FPGA and it was found that the Synthesis and Implementation processes were successfully completed.

ΠΕΡΙΕΧΟΜΕΝΑ

[1. ΕΙΣΑΓΩΓΗ 7](#_Toc169172589)

[1.1 Ο επεξεργαστής 7](#_Toc169172590)

[1.2 Αρχιτεκτονική συνόλου εντολών 7](#_Toc169172591)

[1.3 Αρχιτεκτονική RISC-V 8](#_Toc169172592)

[2. ΣΥΝΟΛΟ ΕΝΤΟΛΩΝ RV32I 11](#_Toc169172593)

[2.1 Τύποι κωδικοποίησης εντολών 11](#_Toc169172594)

[2.2 Εντολές αριθμητικών και λογικών πράξεων 15](#_Toc169172595)

[2.2.1 Εντολές για πράξεις μεταξύ δυο καταχωρητών 15](#_Toc169172596)

[2.2.2 Εντολές για πράξεις μεταξύ καταχωρητή – άμεσου δεδομένου 17](#_Toc169172597)

[2.3 Εντολές προσπέλασης της μνήμης 19](#_Toc169172598)

[2.3.1 Εντολές εγγραφής στη μνήμη (Store) 19](#_Toc169172599)

[2.3.2 Εντολές ανάγνωσης της μνήμης (Load) 20](#_Toc169172600)

[2.4 Εντολές διακλάδωσης 22](#_Toc169172601)

[2.4.1 Εντολές διακλάδωσης υπό συνθήκη (Conditional Branch) 22](#_Toc169172602)

[2.4.2 Εντολές άλματος (Jump) 23](#_Toc169172603)

[2.5 Εντολές ειδικών δεδομένων 24](#_Toc169172604)

[3. ΕΠΕΞΕΡΓΑΣΤΗΣ 26](#_Toc169172605)

[3.1 Τεχνική μερικώς επικαλυπτόμενων λειτουργιών (Pipelining) 26](#_Toc169172606)

[3.2 Λειτουργικές μονάδες του επεξεργαστή 29](#_Toc169172607)

[3.2.1 Κρυφή Μνήμη Εντολών (Instruction Cache Memory) 29](#_Toc169172608)

[3.2.2 Καταχωρητές Γενικού Σκοπού (Register File) 30](#_Toc169172609)

[3.2.3 Μονάδα Παραγωγής Άμεσων Δεδομένων (Immediate Generator) 31](#_Toc169172610)

[3.2.4 Μονάδα Ελέγχου (Control Unit) 33](#_Toc169172611)

[3.2.5 Μονάδα Ανίχνευσης Εξαρτήσεων (Hazard Detect Unit) 34](#_Toc169172612)

[3.2.6 Αριθμητική και Λογική Μονάδα (Arithmetic Logic Unit - ALU) 36](#_Toc169172613)

[3.2.7 Μονάδα Ελέγχου ΑΛΜ (ALU Control) 38](#_Toc169172614)

[3.2.8 Μονάδα Διαχείρισης Διακλαδώσεων (Branch Unit) 39](#_Toc169172615)

[3.2.9 Μονάδα Παροχέτευσης (Forwarding Unit) 40](#_Toc169172616)

[3.2.10 Κρυφή Μνήμη Δεδομένων (Data Cache Memory) 42](#_Toc169172617)

[3.3 Τελικός σχεδιασμός 44](#_Toc169172618)

[3.3.1 Στάδιο IF 44](#_Toc169172619)

[3.3.2 Στάδιο ID 45](#_Toc169172620)

[3.3.3 Στάδιο EX 47](#_Toc169172621)

[3.3.4 Στάδιο MEM 49](#_Toc169172622)

[3.3.4 Στάδιο WB 50](#_Toc169172623)

[3.4 Οι διαδικασίες Synthesis και Implementation 52](#_Toc169172624)

[4. ΕΠΑΛΗΘΕΥΣΗ ΣΧΕΔΙΑΣΜΟΥ 55](#_Toc169172625)

[4.1 Εξαρτήσεις από δεδομένα 55](#_Toc169172626)

[4.1.1 Ανάγνωση μετά από Εγγραφή (ΑμΕ) 55](#_Toc169172627)

[4.1.2 Ακολουθία εντολών load-use 57](#_Toc169172628)

[4.2 Διαδικασιακές εξαρτήσεις 59](#_Toc169172629)

[4.2.1 Εντολές διακλάδωσης υπό συνθήκη 60](#_Toc169172630)

[4.2.2 Εντολές άλματος 62](#_Toc169172631)

[5. ΜΕΛΛΟΝΤΙΚΕΣ ΒΕΛΤΙΩΣΕΙΣ 64](#_Toc169172632)

[5.1 Προσθήκη Επεκτάσεων 64](#_Toc169172633)

[5.2 Προσθήκη Κύριας Μνήμης 64](#_Toc169172634)

[5.3 Υλοποίηση τεχνικής πραγματικής πρόβλεψης για τις εντολές διακλάδωσης 65](#_Toc169172635)

[6. ΥΛΟΠΟΙΗΣΗ 66](#_Toc169172636)

[6.1 Αρχεία Verilog που περιγράφουν το Υλικό 66](#_Toc169172637)

[6.1.1 Περιγραφή Μετρητή Προγράμματος – PC.v 66](#_Toc169172638)

[6.1.2 Περιγραφή Κρυφής Μνήμης Εντολών – InstCache.v 66](#_Toc169172639)

[6.1.3 Περιγραφή Ενδιάμεσου Καταχωρητή IF\_ID – IF\_ID.v 68](#_Toc169172640)

[6.1.4 Περιγραφή Καταχωρητών Γενικού Σκοπού – RegFile.v 68](#_Toc169172641)

[6.1.5 Περιγραφή Μονάδας Ανίχνευσης Εξαρτήσεων – HazardDetectUnit.v 69](#_Toc169172642)

[6.1.6 Περιγραφή Μονάδας Ελέγχου – ControlUnit.v 70](#_Toc169172643)

[6.1.7 Περιγραφή Μονάδας Παραγωγής Άμεσων Δεδομένων – ImmGen.v 73](#_Toc169172644)

[6.1.7 Περιγραφή Ενδιάμεσου Καταχωρητή ID\_ΕΧ – ID\_ΕΧ.v 74](#_Toc169172645)

[6.1.8 Περιγραφή Μονάδας Παροχέτευσης – ForwadingUnit.v 77](#_Toc169172646)

[6.1.9 Περιγραφή Μονάδας Ελέγχου ΑΛΜ – ALU\_control.v 78](#_Toc169172647)

[6.1.10 Περιγραφή Αριθμητικής και Λογικής Μονάδας – ALU.v 81](#_Toc169172648)

[6.1.11 Περιγραφή Μονάδας Διαχείρισης Διακλαδώσεων – BranchUnit.v 83](#_Toc169172649)

[6.1.12 Περιγραφή Ενδιάμεσου Καταχωρητή ΕΧ\_MEM – ΕΧ\_MEM.v 84](#_Toc169172650)

[6.1.13 Περιγραφή Κρυφής Μνήμης Δεδομένων – DataCache.v 85](#_Toc169172651)

[6.1.13 Περιγραφή Ενδιάμεσου Καταχωρητή MEM\_WB – MEM\_WB.v 91](#_Toc169172652)

[6.1.14 Περιγραφή Μηχανισμού Μερικώς Επικαλυπτόμενων Λειτουργιών – PipelineDatapath.v 92](#_Toc169172653)

[6.2 Testbench αρχείo για την εξομοίωση του μηχανισμού 100](#_Toc169172654)

[7. ΠΑΡΑΡΤΗΜΑ – ΠΡΟΓΡΑΜΜΑΤΑ 101](#_Toc169172655)

[8. ΒΙΒΛΙΟΓΡΑΦΙΑ 102](#_Toc169172656)

# 1. ΕΙΣΑΓΩΓΗ

## 1.1 Ο επεξεργαστής

Οι επεξεργαστές αποτελούν τον πυρήνα των υπολογιστικών συστημάτων, εκτελούν ποικίλες εργασίες με μεγάλη ταχύτητα και αποτελεσματικότητα. Αναλαμβάνουν την επεξεργασία δεδομένων, την εκτέλεση εντολών και τον συντονισμό των λειτουργιών του υπολογιστικού συστήματος. Οι εξελίξεις στην τεχνολογία των επεξεργαστών έχουν οδηγήσει σε συνεχή αύξηση της ταχύτητας, της απόδοσης και της ενεργειακής αποδοτικότητας. Η σχεδίαση των επεξεργαστών γίνεται σύμφωνα με διάφορες αρχιτεκτονικές.

## 1.2 Αρχιτεκτονική συνόλου εντολών

H αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture - ISA) είναι μέρος του abstract μοντέλου ενός υπολογιστή που καθορίζει τον τρόπο με τον οποίο η CPU ελέγχεται από το λογισμικό. Η ISA αποτελεί τη σύνδεση μεταξύ του υλικού και του λογισμικού, καθορίζοντας το τι είναι ικανός να κάνει ο επεξεργαστής, αλλά και τον τρόπο με τον οποίο γίνεται αυτό. Μπορεί να θεωρηθεί ως το εγχειρίδιο του προγραμματιστή, επειδή είναι το τμήμα της μηχανής που είναι ορατό στον προγραμματιστή. Η ISA ορίζει τους υποστηριζόμενους τύπους δεδομένων, τους καταχωρητές, τον τρόπο με τον οποίο το υλικό διαχειρίζεται την μνήμη, ποιες εντολές μπορεί να εκτελέσει ένας επεξεργαστής. Τα αποτελέσματα της εκτέλεσης των εντολών μπορούν να είναι αριθμητικά αποτελέσματα, αλλαγή της ροής του προγράμματος με βάση μια συνθήκη ή κατάσταση, αλλαγή της κατάστασης που βρίσκεται ο επεξεργαστής. Οι κατηγορίες για τις αρχιτεκτονικές συνόλου εντολών είναι οι εξής:

1. Reduced Instruction Set Computer (RISC)

Το πλήθος των εντολών είναι περιορισμένο και οι εντολές αυτές εκτελούν βασικές λειτουργίες. Κάθε εντολή εκτελεί μια απλή λειτουργία γενικού σκοπού. Υπάρχουν ξεχωριστές εντολές για την επεξεργασία δεδομένων και την προσπέλαση της μνήμης.

1. Complex Instruction Set Computer (CISC)

Το πλήθος των εντολών είναι μεγάλο και οι εντολές αυτές εκτελούν λειτουργίες ειδικού σκοπού. Κάθε εντολή εκτελεί μια εξειδικευμένη λειτουργία.

## 1.3 Αρχιτεκτονική RISC-V

Η αρχιτεκτονική RISC-V ανήκει στην αρχιτεκτονική RISC και είναι η πρώτη open-source αρχιτεκτονική συνόλου εντολών. Δημιουργήθηκε το 2010 στο Πανεπιστήμιο της Καλιφόρνια, Berkeley από τους Krste Asanović, Andrew Waterman, David Patterson και άλλους. Από τη δημιουργία της έως και σήμερα πολλά άτομα έχουν συμβάλει στην ανάπτυξή της [[1](#_6._ΒΙΒΛΙΟΓΡΑΦΙΑ)]. Για τον σχεδιασμό της RISC-V δόθηκε έμφαση στην απλότητα και στην επεκτασιμότητα. Όσον αφορά την απλότητα, η RISC-V στη βάση της, ορίζεται ως μια μικρή αρχιτεκτονική συνόλου εντολών που αφορά ακέραιους αριθμούς και οι εντολές αυτές πρέπει να είναι παρούσες σε όλες τις υλοποιήσεις. Κάθε σύνολο εντολών για ακέραιους αριθμούς χαρακτηρίζεται από το μήκος των καταχωρητών και το μήκος των διευθύνσεων. Οι δύο βασικές παραλλαγές της RISC-V ISA είναι οι RV32I και RV64I οι οποίες αναφέρονται σε μήκος διεύθυνσης 32-bit και 64-bit αντίστοιχα. Όσον αφορά την επεκτασιμότητα, υπάρχουν επεκτάσεις ( “extensions” ) του συνόλου εντολών, όπως για παράδειγμα η «F» η οποία αφορά πράξεις με αριθμούς κινητής υποδιαστολής. Οι επεκτάσεις χωρίζονται σε standard και non-standard. Οι standard θεωρούνται εκείνες που είναι γενικά χρήσιμες και είναι σχεδιασμένες έτσι ώστε να διαχωρίζονται με σαφήνεια από τις υπόλοιπες standard. Οι non-standard θεωρούνται εκείνες που είναι υψηλά εξιδεικευμένες και ίσως να μην διαχωρίζονται με σαφήνεια από τις υπόλοιπες standard και non-standard [[2](#_6._ΒΙΒΛΙΟΓΡΑΦΙΑ)]. Παρακάτω παρουσιάζονται κάποιες standard επεκτάσεις:

RV32I/RV32E/RV64I/RV128I Base Integer Instruction Set

Το βασικό σύνολο εντολών για 32, 64, 128 bit αντίστοιχα. Περιλαμβάνει εντολές για αριθμητικές και λογικές πράξεις ακέραιων αριθμών, εντολές για ανάγνωση και προσπέλαση μνήμης.

“M” extension Multiplication - Division

Το σύνολο αυτό περιλαμβάνει εντολές για πράξεις πολλαπλασιασμού και διαίρεσης. Γίνεται διαχωρισμός από το Base Integer Instruction Set για να απλουστευθούν οι low-end υλοποιήσεις που δεν προβλέπουν τέτοιες πράξεις.

“A” extension Atomic Instructions

Το σύνολο αυτό περιλαμβάνει εντολές για ανάγνωση, τροποποίηση, εγγραφή της μνήμης για τον συγχρονισμό των πυρήνων ενός RISC-V επεξεργαστή που μοιράζονται την ίδια μνήμη.

“F” extension Single-Precision Floating-Point

Το σύνολο αυτό περιλαμβάνει εντολές για πράξεις με αριθμούς κινητής υποδιαστολής. Η επέκταση αυτή είναι συμβατή με το αριθμητικό πρότυπο IEEE 754-2008. Γίνεται διαχωρισμός από το Base Integer Instruction Set για να απλουστευθούν οι low-end υλοποιήσεις που δεν προβλέπουν τέτοιες πράξεις.

“D” extension Double-Precision Floating-Point

Το σύνολο αυτό περιλαμβάνει εντολές για πράξεις με αριθμούς κινητής υποδιαστολής διπλής ακρίβειας. Βασίζεται στην επέκταση «F».

“Q” extension Quad-Precision Floating-Point

Το σύνολο αυτό περιλαμβάνει εντολές για πράξεις με αριθμούς κινητής υποδιαστολής τετραπλής ακρίβειας. Βασίζεται στις επεκτάσεις «F» και «D».

“C” extension Compressed Instructions

Το σύνολο αυτό περιλαμβάνει συμπιεσμένες εντολές κωδικοποιημένες στα 16-bit οι οποίες αφορούν βασικές λειτουργίες. Αυτό έχει ως αποτέλεσμα τη μείωση του μεγέθους του κώδικα.

# 2. ΣΥΝΟΛΟ ΕΝΤΟΛΩΝ RV32I

Το σύνολο RV32I είναι το βασικό σύνολο εντολών της αρχιτεκτονικής RISC-V, όπου το 32 αντιπροσωπεύει το μέγεθος των διανυσμάτων των εντολών. Παρακάτω θα αναλυθούν οι εντολές του συνόλου, καθώς και τύποι δυαδικής κωδικοποίησης αυτών των εντολών.

## 2.1 Τύποι κωδικοποίησης εντολών

Στο σύνολο εντολών RV32I εμφανίζονται έξι διαφορετικοί τύποι κωδικοποίησης και καθένας προορίζεται για διαφορετική ομάδα εντολών. Πιο αναλυτικά:

R-type:

31 25 24 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| funct7 | rs2 | rs1 | funct3 | rd | opcode |

Σε αυτό τον τύπο κωδικοποίησης αξιοποιούνται τρία πεδία καταχωρητών rs1, rs2 και rd. Παρατηρούνται επίσης και τα πεδία funct7 και funct3. Η ομάδα εντολών στην οποία συναντάται αυτός ο τύπος είναι οι εντολές πράξεων μεταξύ δυο καταχωρητών (Register – Register).

I-type:

31 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| imm[11:0] | rs1 | funct3 | rd | opcode |

Σε αυτό τον τύπο κωδικοποίησης αξιοποιούνται δύο πεδία καταχωρητών rs1 και rd. Παρατηρούνται επίσης το πεδίο άμεσου δεδομένου imm και το funct3. Η ομάδα εντολών στην οποία συναντάται αυτός ο τύπος είναι οι εντολές πράξεων μεταξύ ενός καταχωρητή κι ενός άμεσου δεδομένου (Register - Immediate) καθώς και από τις εντολές ανάγνωσης της μνήμης δεδομένων (Load).

S-type:

31 25 24 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |

Σε αυτό τον τύπο κωδικοποίησης αξιοποιούνται δύο πεδία καταχωρητών rs1 και rs2. Παρατηρούνται επίσης δύο πεδία για το άμεσο δεδομένο imm και το funct3. Η ομάδα εντολών στην οποία συναντάται αυτός ο τύπος είναι οι εντολές εγγραφής στην μνήμη δεδομένων (Store).

B-type:

31 25 24 20 19 15 14 12 11 7 6 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| imm[12|10:5] | rs2 | rs1 | funct3 | imm[4:1|11] | opcode |

Σε αυτό τον τύπο κωδικοποίησης αξιοποιούνται δύο πεδία καταχωρητών rs1 και rs2. Παρατηρούνται επίσης δύο πεδία για το άμεσο δεδομένο imm και το funct3. Η ομάδα εντολών στην οποία συναντάται αυτός ο τύπος είναι οι εντολές διακλάδωσης υπό συνθήκη (Branch).

U-type:

31 12 11 7 6 0

|  |  |  |
| --- | --- | --- |
| imm[31:12] | rd | opcode |

Σε αυτό τον τύπο κωδικοποίησης αξιοποιείται ένα πεδίο για τον καταχωρητή rd. Παρατηρείται επίσης ένα πεδίο για το άμεσο δεδομένο imm. Η ομάδα εντολών στην οποία συναντάται αυτός ο τύπος είναι οι εντολές LUI (Load Upper Immediate) και AUIPC.

J-type:

31 12 11 7 6 0

|  |  |  |
| --- | --- | --- |
| imm[20|10:1|11|19:12] | rd | opcode |

Σε αυτό τον τύπο κωδικοποίησης αξιοποιείται ένα πεδίο για τον καταχωρητή rd. Παρατηρείται επίσης ένα πεδίο για το άμεσο δεδομένο imm. Η ομάδα εντολών στην οποία συναντάται αυτός ο τύπος είναι οι εντολές άλματος (Unconditional Jumps).

Παρατηρείται πως σε καθεμιά κατηγορία υπάρχει διαφορετικός αριθμός πεδίων που έχουν διάφορες χρήσεις. Όμως, τα κοινά πεδία μεταξύ των εντολών είναι στην ίδια θέση, αυτό το χαρακτηριστικό προσφέρει απλότητα στον σχεδιασμό και κάνει την παραγωγή των δυαδικών κωδικοποιήσεων των εντολών πιο εύκολη [[3](#_6._ΒΙΒΛΙΟΓΡΑΦΙΑ_1)]. Παρακάτω ακολουθεί πίνακας που επεξηγεί κάθε ξεχωριστό πεδίο από τους παραπάνω τύπους κωδικοποίησης.

|  |  |  |  |
| --- | --- | --- | --- |
| Πεδίο | Μέγεθος | Περιγραφή | Θέση |
| rs1 | 5 bit | Ο πρώτος καταχωρητής που θα γίνει ανάγνωσή του από τους καταχωρητές γενικού σκοπού | 19 – 15 bits |
| rs2 | 5 bit | Ο δεύτερος καταχωρητής που θα γίνει ανάγνωσή του από τους καταχωρητές γενικού σκοπού | 24 – 20 bits |
| rd | 5 bit | O καταχωρητής στον οποίο αποθηκεύεται το αποτέλεσμα | 11 – 7 bits |
| opcode | 7 bit | O μοναδικός κωδικός εκτέλεσης για κάθε ομάδα εντολών | 0 – 6 bits |
| funct3 | 3 bit | Εξειδικεύει περισσότερο τη λειτουργία της εντολής | 14 – 12 bits |
| funct7 | 7 bit | Εξειδικεύει περισσότερο τη λειτουργία της εντολής | 31 – 25 bits |
| imm | Ανάλογα με τον τύπο κωδικοποίησης | Άμεσο δεδομένο που χρησιμοποιείται στην εκτέλεση της εντολής | Ανάλογα με τον τύπο κωδικοποίησης |

Πίνακας 2.1: Πεδία της κωδικοποίησης των εντολών του συνόλου RV32I

## 2.2 Εντολές αριθμητικών και λογικών πράξεων

Σε αυτή την υποενότητα θα παρουσιαστούν όλες οι εντολές αριθμητικών και λογικών πράξεων που ανήκουν στο σύνολο RV32I.

### 2.2.1 Εντολές για πράξεις μεταξύ δυο καταχωρητών

Η κατηγορία αυτών των εντολών περιλαμβάνει όλες τις εντολές οι οποίες εκτελούν μια μαθηματική ή λογική πράξη μεταξύ των δεδομένων που περιέχουν δυο καταχωρητές (Register - Register). Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός c(x0), συμβολίζει το περιεχόμενο του καταχωρητή x0. Το ίδιο ισχύει και για τους υπόλοιπους καταχωρητές.
2. Οι καταχωρητές που χρησιμοποιούνται έχουν επιλεχθεί τυχαία.

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| ADD x0, x1, x2 | x0 ‹‒ x1 + x2 | Αριθμητική πρόσθεση των c(x1), c(x2). Αποθήκευση του αποτελέσματος στον x0. |
| SUB x0, x1, x2 | x0 ‹‒ x1 - x2 | Αριθμητική αφαίρεση των c(x1), c(x2). Αποθήκευση του αποτελέσματος στον x0. |
| SLL x0, x1, x2 | x0 ‹‒ x1 << x2[4:0] | Λογική ολίσθηση του c(x1) κατά X θέσεις αριστερά. Η τιμή Χ ισούται με τα 5 λιγότερο  σημαντικά bit της τιμής c(x2). Αποθήκευση του αποτελέσματος στον καταχωρητή x0. |
| SLT x0, x1, x2 | (x1 < x2) ? x0 ‹‒ 1 : 0 | Το c(x0) παίρνει την τιμή 1 αν c(x1) < c(x2),  αλλιώς παίρνει την τιμή 0. |
| SLTU x0, x1, x2 | (x1 < x2) ? x0 ‹‒ 1 : 0 | Ίδια λειτουργία με SLT. Οι αριθμοί θεωρούνται μη προσημασμένοι. |
| XOR x0, x1, x2 | x0 ‹‒ x1 ˆ x2 | Λογική πράξη XOR ανά bit μεταξύ των c(x1), c(x2). |
| SRL x0, x1, x2 | x0 ‹‒ x1 >> x2[4:0] | Λογική ολίσθηση του c(x1) κατά Χ θέσεις δεξιά. Η τιμή Χ ισούται με τα 5 λιγότερο  σημαντικά bit της τιμής c(x2). Αποθήκευση του αποτελέσματος στον x0. |
| SRA x0, x1, x2 | x0 ‹‒ x1 >>> x2[4:0] | Αριθμητική ολίσθηση του c(x1) κατά Χ θέσεις δεξιά. Η τιμή Χ ορίζεται ως τα 5 λιγότερο  σημαντικά bit της τιμής c(x2). Αποθήκευση  του αποτελέσματος στον x0. |
| OR x0, x1, x2 | x0 ‹‒ x1 | x2 | Λογική πράξη OR ανά bit μεταξύ των c(x1), c(x2). |
| AND x0, x1, x2 | x0 ‹‒ x1 & x2 | Λογική πράξη AND ανά bit μεταξύ των c(x1), c(x2). |

Πίνακας 2.2: Εντολές Register – Register

### 2.2.2 Εντολές για πράξεις μεταξύ καταχωρητή – άμεσου δεδομένου

Η κατηγορία αυτών των εντολών περιλαμβάνει όλες τις εντολές οι οποίες εκτελούν μια μαθηματική ή λογική πράξη μεταξύ του περιεχομένου ενός καταχωρητή και ενός άμεσου δεδομένου (Register - Immediate). Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός c(x0), συμβολίζει το περιεχόμενο του καταχωρητή x0. Το ίδιο ισχύει και για τους υπόλοιπους καταχωρητές.
2. Οι καταχωρητές που χρησιμοποιούνται έχουν επιλεχθεί τυχαία.

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| ADDI x0, x1, imm | x0 ‹‒ x1 + x2 | Αριθμητική πρόσθεση των c(x1), imm. Αποθήκευση του αποτελέσματος στον x0. |
| SLTI x0, x1, imm | (x1 < imm) ? x0 ‹‒ 1 : 0 | Το c(x0) παίρνει την τιμή 1 αν c(x1) < imm,  αλλιώς παίρνει την τιμή 0. |
| SLTIU x0, x1, imm | (x1 < imm) ? x0 ‹‒ 1 : 0 | Ίδια λειτουργία με SLTI. Οι αριθμοί θεωρούνται μη προσημασμένοι. |
| XORI x0, x1, imm | x0 ‹‒ x1 ˆ imm | Λογική πράξη XOR ανά bit μεταξύ των c(x1), imm. |
| ORI x0, x1, imm | x0 ‹‒ x1 ˆ imm | Λογική πράξη OR ανά bit μεταξύ των c(x1), imm. |
| ANDI x0, x1, imm | x0 ‹‒ x1 ˆ imm | Λογική πράξη AND ανά bit μεταξύ των c(x1), imm. |
| SLLI x0, x1, imm | x0 ‹‒ x1 << imm[4:0] | Λογική ολίσθηση του c(x1) κατά X θέσεις αριστερά. Η τιμή Χ ισούται με τα 5 λιγότερο  σημαντικά bit του imm. Αποθήκευση του αποτελέσματος στον καταχωρητή x0. |
| SRLI x0, x1, imm | x0 ‹‒ x1 >> imm[4:0] | Λογική ολίσθηση του c(x1) κατά X θέσεις δεξιά. Η τιμή Χ ισούται με τα 5 λιγότερο σημαντικά bit του imm. Αποθήκευση του αποτελέσματος στον καταχωρητή x0. |
| SRAΙ x0, x1, imm | x0 ‹‒ x1 >>> imm[4:0] | Αριθμητική ολίσθηση του c(x1) κατά Χ θέσεις δεξιά. Η τιμή Χ ισούται με τα 5 λιγότερο σημαντικά bit της τιμής imm. Αποθήκευση  του αποτελέσματος στον x0. |

Πίνακας 2.3: Εντολές Register – Immediate

## 2.3 Εντολές προσπέλασης της μνήμης

Στην αρχιτεκτονική RV32I μόνο οι εντολές load και store έχουν πρόσβαση στη μνήμη [[2](#_8._ΒΙΒΛΙΟΓΡΑΦΙΑ_1)].

### 2.3.1 Εντολές εγγραφής στη μνήμη (Store)

Η κατηγορία αυτών των εντολών περιλαμβάνει όλες τις εντολές οι οποίες εκτελούν μεταφορά δεδομένων από τους καταχωρητές στην μνήμη ώστε να αποθηκευτούν. Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός c(x0), συμβολίζει το περιεχόμενο του καταχωρητή x0. Το ίδιο ισχύει και για τους υπόλοιπους καταχωρητές.
2. Όπου εμφανίζεται ο συμβολισμός M(x), συμβολίζει το περιεχόμενο της θέσης μνήμης με διεύθυνση x.
3. Οι καταχωρητές που χρησιμοποιούνται έχουν επιλεχθεί τυχαία.
4. Στην Κρυφή Μνήμη Εντολών σε κάθε διεύθυνση αντιστοιχεί και μια λέξη των 32-bit (word).

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| SB x4, imm(x1) | M(c(x1) + imm) ‹‒ c(x4)[7:0] | Αποθήκευση του λιγότερου σημαντικού byte του c(x4) στην θέση μνήμης με διεύθυνση  (c(x1) + imm). |
| SH x4, imm(x1) | M(c(x1) + imm) ‹‒ c(x4)[15:0] | Αποθήκευση του λιγότερο σημαντικού byte του c(x4) στην θέση μνήμης με διεύθυνση  (c(x1) + imm). |
| SW x4, imm(x1) | M(c(x1) + imm) ‹‒ c(x4)[31:0] | Αποθήκευση του c(x4) στη θέση μνήμης με διεύθυνση  (c(x1) + imm). |

Πίνακας 2.4: Εντολές Store

### 2.3.2 Εντολές ανάγνωσης της μνήμης (Load)

Η κατηγορία αυτών των εντολών περιλαμβάνει όλες τις εντολές για την μεταφορά δεδομένων από την μνήμη στους καταχωρητές γενικού σκοπού. Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός c(x0), συμβολίζει το περιεχόμενο του καταχωρητή x0. Το ίδιο ισχύει και για τους υπόλοιπους καταχωρητές.
2. Όπου εμφανίζεται ο συμβολισμός M(x), συμβολίζει το περιεχόμενο της θέσης μνήμης με διεύθυνση x.
3. Οι καταχωρητές που χρησιμοποιούνται έχουν επιλεχθεί τυχαία.
4. Στην Κρυφή Μνήμη Εντολών σε κάθε διεύθυνση αντιστοιχεί και μια λέξη των 32-bit (word).

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| LB x4, imm(x1) | x4 ‹‒ M(c(x1) + imm) | Ανάγνωση του περιεχομένου της θέσης μνήμης με διεύθυνση (c(x1) + imm) και αποθήκευση στον x4 του λιγότερο σημαντικού byte αφού εφαρμοστεί επέκταση προσήμου. |
| LH x4, imm(x1) | x4 ‹‒ M(c(x1) + imm) | Ανάγνωση του περιεχομένου της θέσης μνήμης με διεύθυνση (c(x1) + imm) και αποθήκευση στον x4 του λιγότερο σημαντικού halfword αφού εφαρμοστεί επέκταση προσήμου. |
| LW x4, imm(x1) | x4 ‹‒ M(c(x1) + imm) | Ανάγνωση του περιεχομένου της θέσης μνήμης με διεύθυνση (c(x1) + imm) και αποθήκευση στον x4. |
| LBU x4, imm(x1) | x4 ‹‒ M(c(x1) + imm) | Ίδια λειτουργία με LB χωρίς την επέκταση προσήμου. |
| LHU x4, imm(x1) | x4 ‹‒ M(c(x1) + imm) | Ίδια λειτουργία με LH χωρίς την επέκταση προσήμου. |

Πίνακας 2.5: Εντολές Load

## 2.4 Εντολές διακλάδωσης

Η κατηγορία αυτών των εντολών περιλαμβάνει όλες τις εντολές που μπορούν να αλλάξουν την ροή του προγράμματος.

### 2.4.1 Εντολές διακλάδωσης υπό συνθήκη (Conditional Branch)

Στις εντολές διακλάδωσης υπό συνθήκη γίνεται κατάλληλη σύγκριση του περιεχομένου των δύο καταχωρητών που υποδεικνύονται από την εκάστοτε εντολή. Αν η συνθήκη ικανοποιείται τότε πραγματοποιείται η διακλάδωση, αλλιώς συνεχίζεται η κανονική ροή του προγράμματος. Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός c(x0), συμβολίζει το περιεχόμενο του καταχωρητή x0. Το ίδιο ισχύει και για τους υπόλοιπους καταχωρητές.
2. Όπου εμφανίζεται ο συμβολισμός PC, συμβολίζει το Μετρητή Προγράμματος.

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| BEQ x1, x2, offset | c(x1) == c(x2) ? PC += offset  : PC += 1 | Αν c(x1) ίσο με c(x2) τότε προστίθεται στην παρούσα τιμή του PC το offset. |
| BNE x1, x2, offset | c(x1) != c(x2) ? PC += offset  : PC += 1 | Αν c(x1) διαφορετικό του c(x2) τότε προστίθεται στην παρούσα τιμή του PC το offset. |
| BLT x1, x2, offset | c(x1) < c(x2) ? PC += offset  : PC += 1 | Αν c(x1) μικρότερο του c(x2) τότε προστίθεται στην παρούσα τιμή του PC το offset. |
| BGE x1, x2, offset | c(x1) >= c(x2) ? PC += offset  : PC += 1 | Αν c(x1) μεγαλύτερο ή ίσο του c(x2) τότε προστίθεται στην παρούσα τιμή του PC το offset. |
| BLTU x1, x2, offset | c(x1) < c(x2) ? PC += offset  : PC += 1 | Αντίστοιχο της BLT, όμως η σύγκριση γίνεται θεωρώντας πως οι αριθμοί είναι μη προσημασμένοι. |
| BGEU x1, x2, offset | c(x1) >= c(x2) ? PC += offset  : PC += 1 | Αντίστοιχο της BGE, όμως η σύγκριση γίνεται θεωρώντας πως οι αριθμοί είναι μη προσημασμένοι. |

Πίνακας 2.6: Εντολές Branch

### 2.4.2 Εντολές άλματος (Jump)

Η κατηγορία αυτών των εντολών περιλαμβάνει όλες τις εντολές που αλλάζουν την ροή του προγράμματος χωρίς την ικανοποίηση κάποιας συνθήκης. Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός c(x0), συμβολίζει το περιεχόμενο του καταχωρητή x0. Το ίδιο ισχύει και για τους υπόλοιπους καταχωρητές.
2. Όπου εμφανίζεται ο συμβολισμός PC, συμβολίζει το Μετρητή Προγράμματος.

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| JAL x1, offset | x1 ‹‒ PC + 1,  PC += offset | Αποθηκεύεται η διεύθυνση της επόμενης εντολής στον x1 και προστίθεται στην παρούσα τιμή του PC το offset. |
| JALR x1, offset(x2) | x1 ‹‒ PC + 1,  PC=c(x2)+offset | Αποθηκεύεται η διεύθυνση της επόμενης εντολής στον x1 και οδηγείται στον PC το (offset + c(x2)). |

# 

Πίνακας 2.7: Εντολές Jump

## 2.5 Εντολές ειδικών δεδομένων

Η κατηγορία αυτών των εντολών περιλαμβάνει δυο εντολές που χρησιμοποιούν την κωδικοποίηση U-type. Ακολουθεί πίνακας με αυτές τις εντολές:

Διευκρινήσεις:

1. Όπου εμφανίζεται ο συμβολισμός PC, συμβολίζει το Μετρητή Προγράμματος.

|  |  |  |
| --- | --- | --- |
| Εντολή | Πράξη | Λειτουργία |
| LUI x1, imm | x1 ‹‒ (imm << 12) | Αποθηκεύεται το 20-bit άμεσο δεδομένο στα 20 πιο σημαντικά bit του x1 και τα 12 λιγότερο σημαντικά bit συμπληρώνονται με μηδέν. |
| AUIPC x1, imm | x1 ‹‒ PC + (imm << 12) | Δημιουργείται μια σταθερά 32-bit από το 20-bit imm και συμπλήρωση των 12 λιγότερο σημαντικών bit με μηδέν. Η σταθερά αυτή προστίθεται στην παρούσα τιμή του PC και το αποτέλεσμα αποθηκεύεται στον x1. |

Πίνακας 2.8: Εντολές ειδικών δεδομένων

# 3. ΕΠΕΞΕΡΓΑΣΤΗΣ

Σε αυτή την ενότητα θα παρουσιαστούν οι σχεδιαστικές αποφάσεις που λήφθηκαν κατά την σχεδίαση του επεξεργαστή. Οι αποφάσεις πάρθηκαν με βάση την ανάλυση των εντολών του συνόλου RV32I που έγινε στην προηγούμενη ενότητα ([Ενότητα 2](#_2._ΣΥΝΟΛΟ_ΕΝΤΟΛΩΝ)).

## 3.1 Τεχνική μερικώς επικαλυπτόμενων λειτουργιών (Pipelining)

Είναι γνωστό ότι κατά την εκτέλεση μιας εντολής, ακολουθούνται κάποια βήματα που εκτελούνται το ένα μετά το άλλο. Η τεχνική των μερικώς επικαλυπτόμενων λειτουργιών ορίζει διαδοχικές βαθμίδες, οι οποίες εκτελούν η κάθε μια ένα από αυτά τα βήματα. Το σημαντικό χαρακτηριστικό της τεχνικής αυτής είναι ότι προσφέρει τη δυνατότητα να εκκινήσει η εκτέλεση μιας εντολής χωρίς να έχει ολοκληρωθεί η εκτέλεση μιας παλαιότερης, δηλαδή εκτελούνται παράλληλα τα στάδια διαφορετικών εντολών. Η τεχνική αυτή, έχει εφαρμοστεί σε επεξεργαστές RISC και CISC αρχιτεκτονικής.

Η τεχνική των μερικώς επικαλυπτόμενων λειτουργιών στην αρχιτεκτονική RISC ορίζει τις παρακάτω βαθμίδες [[4](#_6._ΒΙΒΛΙΟΓΡΑΦΙΑ_1)]:

1. ΙF – Instruction Fetching (Προσκόμιση εντολής):

Στο στάδιο αυτό προσκομίζεται από τη μνήμη εντολών η εντολή που τίθεται προς εκτέλεση. Ο μετρητής προγράμματος αυξάνεται κατάλληλα ώστε να υποδείξει τη διεύθυνση της επόμενης προς εκτέλεση εντολής.

1. ID – Instruction Decoding (Αποκωδικοποίηση εντολής):

Στο στάδιο αυτό γίνεται ανάγνωση των καταχωρητών και παράγονται τα ανάλογα σήματα ελέγχου για κάθε εντολή.

1. EX – Execution (Εκτέλεση πράξεων):

Στο στάδιο αυτό γίνεται η πράξη που έχει οριστεί από την εντολή. Το αποτέλεσμα της πράξης μπορεί να είναι η διεύθυνση προσπέλασης της μνήμης δεδομένων που βρίσκεται στο αμέσως επόμενο στάδιο (MEM) ή ένα αριθμητικό αποτέλεσμα το οποίο αποθηκεύεται σε κάποιο καταχωρητή σε μεταγενέστερο στάδιο (WB).

1. MEM – Memory Access (Προσπέλαση Μνήμης):

Στο στάδιο αυτό γίνεται ανάγνωση δεδομένων από τη μνήμη ή αποθήκευση δεδομένων σε αυτή.

1. WB – Write Back (Αποθήκευση αποτελεσμάτων):

Στο στάδιο αυτό που είναι και το τελευταίο, γίνεται αποθήκευση αποτελέσματος σε κάποιον από τους καταχωρητές.

Είναι αρκετά δύσκολο να σχεδιαστεί ένας επεξεργαστής με την τεχνική των μερικώς επικαλυπτόμενων λειτουργιών δίχως την ύπαρξη καταχωρητών μεταξύ των βαθμίδων που αναφέρθηκαν πριν. Αυτό οφείλεται στην ανάγκη κάθε βαθμίδα να έχει τον ίδιο χρόνο καθυστέρησης με τις υπόλοιπες. Η ανάγκη αυτή λοιπόν οδηγεί στη χρήση καταχωρητών μεταξύ των βαθμίδων οι οποίοι ονομάζονται ενδιάμεσοι καταχωρητές. Οι καταχωρητές λαμβάνουν το ίδιο σήμα χρονισμού (ρολόι) και η περίοδος αυτού του σήματος πρέπει να έχει διάρκεια που να επιτρέπει την εκτέλεση όλων των λειτουργιών που έχει αναλάβει κάθε βαθμίδα. Οι ενδιάμεσοι καταχωρητές που ορίστηκαν κατά τον σχεδιασμό του παρόντος επεξεργαστή είναι οι IF\_ID, ID\_EX, EX\_MEM, MEM\_WB. Η ονομασία κάθε καταχωρητή προέκυψε από τις βαθμίδες στις οποίες βρίσκεται ανάμεσα ο εκάστοτε καταχωρητής.

Όπως αναφέρθηκε πρωτύτερα, σε ένα μηχανισμό μερικώς επικαλυπτόμενων λειτουργιών τα διαφορετικά στάδια διαφορετικών εντολών εκτελούνται παράλληλα μέσα στο μηχανισμό. Αυτό το γεγονός προκαλεί καταστάσεις στις οποίες εμποδίζεται κάποιο στάδιο μιας εντολής να εκτελεστεί την χρονική περίοδο που έχει οριστεί. Αυτές οι καταστάσεις ονομάζονται εξαρτήσεις (hazards). Οι εξαρτήσεις που εγείρονται κατατάσσονται σε τρεις κατηγορίες οι οποίες αναφέρονται παρακάτω [[4](#_6._ΒΙΒΛΙΟΓΡΑΦΙΑ_2)]:

1. Structural Hazards (Δομικές εξαρτήσεις):

Αυτό το είδος εξάρτησης προκύπτει όταν κάποιες εντολές που έχουν εισαχθεί για εκτέλεση στον μηχανισμό μερικώς επικαλυπτόμενων λειτουργιών, απαιτούν την ταυτόχρονη χρήση μιας βαθμίδας του μηχανισμού. #Να αναφέρω ότι δεν υπάρχουν στην δική μου περίπτωση.

1. Data Hazards (Εξαρτήσεις από δεδομένα):

Αυτό το είδος εξάρτησης προκύπτει μεταξύ δυο εντολών που είναι κοντά στη σειρά εκτέλεσης ώστε η επικάλυψη των λειτουργιών τους να οδηγεί σε λανθασμένη προσπέλαση κάποιων δεδομένων. Για παράδειγμα αναφέρεται ένα ζεύγος διαδοχικών εντολών όπου η δεύτερη χρησιμοποιεί το αποτέλεσμα της εκτέλεσης της πρώτης.

1. Control Hazards (Διαδικασιακές εξαρτήσεις):

Αυτό το είδος εξάρτησης προκύπτει όταν η εκτέλεση μιας εντολής διακλάδωσης ([2.4](#_2.4_Εντολές_διακλάδωσης)) προκαλεί αλλαγή στη ροή του προγράμματος.

## 

## 3.2 Λειτουργικές μονάδες του επεξεργαστή

Ως λειτουργικές μονάδες του επεξεργαστή θεωρούνται οι μονάδες που υπάρχουν μέσα στον επεξεργαστή και υλοποιούνται ως ακολουθιακά κυκλώματα. Η θύρα εισόδου του σήματος ρολογιού (clk) καθώς και η θύρα εισόδου για το σήμα reset παραλείπεται από τις περιγραφές.

### 3.2.1 Κρυφή Μνήμη Εντολών (Instruction Cache Memory)

Η Κρυφή Μνήμη Εντολών (ΚΜΕ) είναι μία κρυφή μνήμη με οργάνωση μονοσήμαντης απεικόνισης (Direct Mapping). Λαμβάνει διευθύνσεις από τον Μετρητή Προγράμματος (Programm Counter) και με αυτό τον τρόπο κάνει ανάγνωση της εντολής η οποία τίθεται προς εκτέλεση. Ο τρόπος χρήσης των διευθύνσεων που λαμβάνει η ΚΜΕ είναι ο εξής:

31 8 7 3 2 0

|  |  |  |
| --- | --- | --- |
| tag | index | offset |

Το διάστημα offset καταλαμβάνει τα 3 λιγότερο σημαντικά bits της διεύθυνσης και προσδιορίζει τη διεύθυνση μιας λέξης μέσα στο πλαίσιο της κρυφής μνήμης. Στην συγκεκριμένη περίπτωση το πλαίσιο αποτελείται από 8 λέξεις. Το διάστημα index καταλαμβάνει τα 5 αμέσως επόμενα bits της διεύθυνσης και προσδιορίζει τη διεύθυνση του πλαισίου μέσα στην κρυφή μνήμη, η κρυφή μνήμη αποτελείται από 32 πλαίσια. Το διάστημα tag καταλαμβάνει τα υπόλοιπα 24 bits τα οποία αποτελούν την ετικέτα κάθε διεύθυνσης.

Η ΚΜΕ διαθέτει την θύρα εισόδου:

* addr: Θύρα εισόδου σήματος για την διεύθυνση προσπέλασης της μνήμης, με μέγεθος 32 bits.

Η ΚΜΕ διαθέτει την θύρα εξόδου:

* inst: Θύρα εξόδου σήματος για την εντολή που διαβάζεται από την μνήμη, με μέγεθος 32 bits.

### 3.2.2 Καταχωρητές Γενικού Σκοπού (Register File)

Οι Καταχωρητές Γενικού Σκοπού (ΚΓΣ) ορίζονται ως το σύνολο των καταχωρητών του επεξεργαστή και χρησιμεύουν στην προσωρινή αποθήκευση δεδομένων. Η αρχιτεκτονική συνόλου εντολών RV32I προδιαγράφει το πλήθος των καταχωρητών στους ΚΓΣ να είναι 32. Άλλο ένα χαρακτηριστικό είναι ότι ο καταχωρητής 0 δηλαδή ο πρώτος, πρέπει πάντα το περιεχόμενό του να είναι ίσο με μηδέν. Επιπλέον, με την τεχνική των μερικώς επικαλυπτόμενων λειτουργιών επιβάλλεται η ταυτόχρονη εγγραφή και ανάγνωση των ΚΓΣ κάτι το οποίο μπορεί να προκαλέσει δομική εξάρτηση. Η δομική εξάρτηση προκαλείται όταν μια μονάδα χρησιμοποιείται σε παραπάνω από μία βαθμίδες. Στην προκειμένη περίπτωση αυτές οι βαθμίδες είναι οι WB (εγγραφή σε καταχωρητή) και ID (ανάγνωση καταχωρητή). Το πρόβλημα αντιμετωπίζεται με την υλοποίηση των ΚΓΣ η οποία επιτρέπει στον ίδιο κύκλο ρολογιού την ταυτόχρονη εγγραφή και ανάγνωση στους καταχωρητές. Στην πιο ειδική περίπτωση όπου η εγγραφή και ανάγνωση αφορά τον ίδιο καταχωρητή, η εγγραφή πραγματοποιείται στο πρώτο μισό του σήματος ρολογιού και η ανάγνωση στο δεύτερο μισό.

Οι ΚΓΣ διαθέτουν τις παρακάτω θύρες εισόδου:

* rs1, rs2:Θύρες εισόδου για τη διεύθυνση των καταχωρητών προς ανάγνωση, με μέγεθος 5 bits.
* wr: Θύρα εισόδου για τη διεύθυνση του καταχωρητή προς εγγραφή, με μέγεθος 5 bits.
* wd: Θύρα εισόδου για τα δεδομένα προς εγγραφή στον καταχωρητή που υποδεικνύει η διεύθυνση wr, με μέγεθος 32 bits.
* RegWrite: Θύρα εισόδου με μέγεθος 1 bit που λειτουργεί ως είσοδος επίτρεψης για την εγγραφή του καταχωρητή που υποδεικνύει η διεύθυνση wr. Αν πάρει την τιμή 1 τότε γίνεται εγγραφή.

Οι ΚΓΣ διαθέτουν τις παρακάτω θύρες εξόδου:

* rd1, rd2: Θύρες εξόδου για τα δεδομένα που είναι αποθηκευμένα στους καταχωρητές που υποδεικνύονται από τις διευθύνσεις rs1, rs2 αντίστοιχα, με μέγεθος 32 bits.

Εικόνα που περιέχει κείμενο, διάγραμμα, στιγμιότυπο οθόνης, γραμμή

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.1: Καταχωρητές Γενικού Σκοπού

### 3.2.3 Μονάδα Παραγωγής Άμεσων Δεδομένων (Immediate Generator)

Η Μονάδα Παραγωγής Άμεσων Δεδομένων (ΜΠΑΔ) έχει ως σκοπό την αποκωδικοποίηση των άμεσων δεδομένων που είναι ενσωματωμένα στις εντολές. Άμεσα δεδομένα περιέχουν οι εντολές τύπου I, S, B, U, J όπως αναφέρθηκε στην [2.1](#_2.1_Τύποι_κωδικοποίησης) . Ο κάθε τύπος κωδικοποιεί το δεδομένο με διαφορετικό τρόπο μέσα στην εντολή. Έτσι, ανάλογα με τον opcode της εντολής η μονάδα μετατρέπει το δεδομένο σε κατάλληλη μορφή ώστε να μεταφερθεί στην επόμενη βαθμίδα για περαιτέρω επεξεργασία. Το δεδομένο που παράγεται πρέπει να οδηγηθεί στο στάδιο EX, άρα πρέπει η ΜΠΑΔ να τοποθετηθεί πριν από αυτό. Επίσης, η ΜΠΑΔ αντικαθιστά την ανάγνωση κάποιου καταχωρητή με την παραγωγή ενός άμεσου δεδομένου. Επομένως, η ΜΠΑΔ τοποθετείται στο στάδιο ID.

Η ΜΠΑΔ διαθέτει την θύρα εισόδου:

* Instruction: Θύρα εισόδου διανύσματος εντολής, με μέγεθος 32 bits.

Η ΜΠΑΔ διαθέτει την θύρα εξόδου:

* Immediate: Θύρα εξόδου για το αποτέλεσμα της αποκωδικοποίησης του άμεσου δεδομένου, με μέγεθος 32 bits.

Εικόνα που περιέχει γραμμή, διάγραμμα, στιγμιότυπο οθόνης, ορθογώνιο παραλληλόγραμμο

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.2: Μονάδα Παραγωγής Άμεσων Δεδομένων

### 

### 3.2.4 Μονάδα Ελέγχου (Control Unit)

Η Μονάδα Ελέγχου (ME) είναι η μονάδα που αναλαμβάνει την αποκωδικοποίηση της εντολής που προσκομίστηκε στον προηγούμενο κύκλο ρολογιού από την Μνήμη Εντολών, με αποτέλεσμα να τοποθετηθεί στο στάδιο ID. Η αποκωδικοποίηση βασίζεται αποκλειστικά στον κωδικό κάθε εντολής (opcode). Είναι ένα εξαιρετικά σημαντικό κύκλωμα διότι παράγει όλα τα σήματα που κρίνονται απαραίτητα για την εκτέλεση της εκάστοτε εντολής.

Η ΜΕ διαθέτει την θύρα εισόδου:

* Opcode: Θύρα εισόδου του κωδικού της εντολής, με μέγεθος 7 bits.

Η ΜΕ διαθέτει τις θύρες εξόδου:

* LUIorAUIPC: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο EX όπου εκεί καθορίζει τη ροή της λειτουργίας σύμφωνα με το αν εντολή που εκτελείται είναι η LUI ή η AUIPC.
* Jump: Θύρα εξόδου σήματος με μέγεθος 2 bits το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο EX όπου εκεί μπαίνει ως είσοδος στη Μονάδας Διαχείρισης Διακλαδώσεων.
* ALUop: Θύρα εξόδου σήματος με μέγεθος 2 bits το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο EX όπου εκεί μπαίνει ως είσοδος στη Μονάδας Ελέγχου ΑΛΜ.
* ALUsrc: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο EX όπου εκεί καθορίζει αν η είσοδος 1 (ALU1) της ΑΛΜ θα τροφοδοτηθεί από ένα άμεσο δεδομένο ή από το περιεχόμενο κάποιου καταχωρητή.
* MemRead: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο ΜΕΜ όπου εκεί καθορίζει αν θα γίνει ανάγνωση στην Μνήμη Δεδομένων.
* MemWrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο ΜΕΜ όπου εκεί καθορίζει αν θα γίνει εγγραφή στην Μνήμη Δεδομένων.
* RegWrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο WB όπου εκεί καθορίζει αν θα γίνει εγγραφή σε καταχωρητή γενικού σκοπού.
* MemToReg: Θύρα εξόδου σήματος με μέγεθος 2 bits το οποίο μέσω των ενδιάμεσων καταχωρητών διαδίδεται μέχρι το στάδιο WB όπου εκεί καθορίζει την έξοδο ενός πολυπλέκτη.

### 3.2.5 Μονάδα Ανίχνευσης Εξαρτήσεων (Hazard Detect Unit)

Η Μονάδα Ανίχνευσης Εξαρτήσεων (ΜΑΕ) είναι η μονάδα που αναλαμβάνει να ανιχνεύσει μια ειδική περίπτωση εξάρτησης από δεδομένα. Αυτή η εξάρτηση προκύπτει από μια συγκεκριμένη ακολουθία εντολών, πιο συγκεκριμένα όταν μια εντολή ανάγνωσης της μνήμης (Load) ακολουθείται από μια εντολή η οποία χρησιμοποιεί το περιεχόμενο του καταχωρητή που ανανεώνει η Load που αναφέρθηκε (ακολουθία εντολών Load-Use). Το επιθυμητό δεδομένο θα είναι διαθέσιμο στο στάδιο εκτέλεσης MEM της εντολής Load, αν δεν ληφθεί κάποιο μέτρο τότε η εντολή Use θα φτάσει στο στάδιο EX έχοντας διαβάσει λανθασμένο περιεχόμενο για τον καταχωρητή. Για αυτό κρίνεται απαραίτητη η καθυστέρηση (pipeline stall) της εντολής Use κατά ένα κύκλο ρολογιού όταν αυτή βρίσκεται στο στάδιο ID, επομένως όταν η Load βρίσκεται στο στάδιο EX. Επομένως, η ΜΑΕ τοποθετείται στο στάδιο ID.

Η ΜAΕ διαθέτει τις θύρες εισόδου:

* ID\_EXmemRead: Θύρα εισόδου σήματος με μέγεθος 1 bit το οποίο μαρτυρά αν η εντολή που βρίσκεται στο στάδιο EX είναι εντολή Load.
* ID\_EXrd: Θύρα εισόδου σήματος με μέγεθος 5 bits το οποίο πληροφορεί για τον καταχωρητή στον οποίο πρόκειται να αποθηκεύσει η εντολή που βρίσκεται στο στάδιο EX.
* IF\_IDrs1, IF\_IDrs2: Θύρες εισόδου σημάτων με μέγεθος 5 bits τα οποία πληροφορούν για τους καταχωρητές που θέλει να κάνει ανάγνωση η εντολή που βρίσκεται στο στάδιο ID.

Η ΜAΕ διαθέτει τις θύρες εξόδου:

* PCwrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο όταν είναι 1 τότε ο Μετρητής Προγράμματος (ΜΠ) θα ανανεώσει κατάλληλα την τιμή του στον επόμενο κύκλο ρολογιού, αλλιώς αν είναι 0 τότε ο ΜΠ θα διατηρήσει την τιμή που είχε στον προηγούμενο κύκλο ρολογιού.
* IF\_IDwrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο όταν είναι 1 τότε ο ενδιάμεσος καταχωρητής IF\_ID θα ανανεώσει κατάλληλα την τιμή του στον επόμενο κύκλο ρολογιού, αλλιώς αν είναι 0 τότε ο IF\_ID θα διατηρήσει την τιμή που είχε στον προηγούμενο κύκλο ρολογιού.
* ID\_EXwrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο όταν είναι 1 τότε ο ενδιάμεσος καταχωρητής ID\_EX θα ανανεώσει κατάλληλα την τιμή του στον επόμενο κύκλο ρολογιού, αλλιώς αν είναι 0 τότε ο ID\_EX θα διατηρήσει την τιμή που είχε στον προηγούμενο κύκλο ρολογιού
* regWrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο οδηγεί ένα πολυπλέκτη. Αν το σήμα έχει τιμή 1 τότε ο πολυπλέκτης θα διαδώσει την τιμή του σήματος RegWrite που παράγει η Μονάδα Ελέγχου και αναφέρθηκε στο [3.2.4](#_3.2.4_Μονάδα_Ελέγχου) . Αλλιώς, αν έχει τιμή 0 ο πολυπλέκτης θα διαδώσει την τιμή 0.
* memWrite: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο οδηγεί ένα πολυπλέκτη. Αν το σήμα έχει τιμή 1 τότε ο πολυπλέκτης θα διαδώσει την τιμή του σήματος MemWrite που παράγει η Μονάδα Ελέγχου και αναφέρθηκε στο [3.2.4](#_3.2.4_Μονάδα_Ελέγχου) . Αλλιώς, αν έχει τιμή 0 ο πολυπλέκτης θα διαδώσει την τιμή 0.

Οι τιμές των σημάτων στις θύρες εξόδου πρέπει να είναι αρχικοποιημένες στο 1, όταν αρχίζει η εκτέλεση ενός προγράμματος.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμμή, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.3: Μονάδα Ανίχνευσης Εξαρτήσεων

### 3.2.6 Αριθμητική και Λογική Μονάδα (Arithmetic Logic Unit - ALU)

Η Αριθμητική και Λογική Μονάδα (ΑΛΜ) είναι η μονάδα που εκτελούνται οι πράξεις που απαιτούνται από εντολές τύπου R, I, S, B. Τοποθετείται στο στάδιο EX. Στις εντολές τύπου R εφαρμόζεται μία πράξη σε 2 δεδομένα που προκύπτουν από την ανάγνωση 2 καταχωρητών. Στις εντολές τύπου I γίνεται μία πράξη σε 2 δεδομένα που προκύπτουν από την ανάγνωση ενός καταχωρητή και από ένα άμεσο δεδομένο. Επιπλέον, στις εντολές τύπου I κατατάσσονται και οι εντολές Load για τις οποίες η ΑΛΜ υπολογίζει τη διεύθυνση προσπέλασης της Μνήμης Δεδομένων. Οι εντολές τύπου S είναι οι εντολές Store για τις οποίες η ΑΛΜ υπολογίζει επίσης τη διεύθυνση προσπέλασης της Μνήμης Δεδομένων. Τέλος, για τις εντολές τύπου B, δηλαδή για τις εντολές διακλάδωσης υπό συνθήκη ([2.4.1](#_2.4.1_Εντολές_διακλάδωσης)) η ΑΛΜ εφαρμόζει τη σύγκριση που απαιτείται από την εκάστοτε εντολή.

Η ΑΛΜ διαθέτει τις θύρες εισόδου:

* data0, data1: Θύρες εισόδου σημάτων με μέγεθος 32 bits, τα οποία συμμετέχουν στην εκτέλεση της πράξης.
* ctrl: Θύρα εισόδου σήματος με μέγεθος 4 bits το οποίο προέρχεται από την Μονάδα Ελέγχου ΑΛΜ, το σήμα αυτό υποδεικνύει την πράξη ή την σύγκριση που χρειάζεται να εκτελέσει η ΑΛΜ.

Η ΑΛΜ διαθέτει τις θύρες εξόδου:

* result: Θύρα εξόδου σήματος με μέγεθος 32 bits το οποίο δίνει το αποτέλεσμα της πράξης που εκτελεί η ΑΛΜ.
* branch: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο οδηγείται στην Μονάδα Διαχείρισης Διακλαδώσεων. Αν το σήμα είναι 1 τότε αυτό σημαίνει πως ικανοποιείται η συνθήκη της εντολής διακλάδωσης.

Εικόνα που περιέχει διάγραμμα, γραμμή, σχεδίαση

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.4: Αριθμητική και Λογική Μονάδα

### 3.2.7 Μονάδα Ελέγχου ΑΛΜ (ALU Control)

Η Μονάδα Ελέγχου ΑΛΜ (ΜΕΑΛΜ) είναι η μονάδα που υποδεικνύει στην Αριθμητική και Λογική Μονάδα την πράξη ή την σύγκριση που πρέπει να εκτελέσει. Η ΜΕΑΛΜ εκμεταλλεύεται συγκεκριμένα πεδία κάθε εντολής.

Η ΜΕΑΛΜ διαθέτει τις θύρες εισόδου:

* ALUctrl\_f7: Θύρα εισόδου σήματος με μέγεθος 7 bits το οποίο αντιπροσωπεύει το πεδίο funct7 που αναφέρθηκε στον Πίνακα 1.
* ALUctrl\_f3: Θύρα εισόδου σήματος με μέγεθος 3 bits το οποίο αντιπροσωπεύει το πεδίο funct3 που αναφέρθηκε στον Πίνακα 1.
* ALUop: Θύρα εισόδου σήματος με μέγεθος 2 bits το οποίο προέρχεται από την Μονάδα Ελέγχου ([3.2.4](#_3.2.4_Μονάδα_Ελέγχου)).

Η ΜΕΑΛΜ διαθέτει τις θύρες εξόδου:

* ALUctrl\_lines: Θύρα εξόδου σήματος με μέγεθος 4 bits το οποίο οδηγείται στην Αριθμητική και Λογική Μονάδα.

Εικόνα που περιέχει κείμενο, διάγραμμα, κύκλος, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.5: Μονάδα Ελέγχου ΑΛΜ

### 3.2.8 Μονάδα Διαχείρισης Διακλαδώσεων (Branch Unit)

Η Μονάδα Διαχείρισης Διακλαδώσεων (ΜΔΔ) είναι η μονάδα που καθορίζει τις διαδικασίες που πρέπει να γίνουν κατά την εκτέλεση μιας εντολής διακλάδωσης ([2.4](#_2.4_Εντολές_διακλάδωσης)). Κάθε εντολή διακλάδωσης πρέπει να φτάσει στο στάδιο EX διότι χρειάζονται τιμές από τους καταχωρητές γενικού σκοπού για την εκτέλεση τους. Ειδικότερα στην περίπτωση των εντολών διακλάδωσης υπό συνθήκη, πρέπει να γίνει και σύγκριση μεταξύ των περιεχομένων των καταχωρητών. Με βάση τα παραπάνω η ΜΔΔ τοποθετείται στο στάδιο EX.

Η ΜΔΔ διαθέτει τις θύρες εισόδου:

* branch: Θύρα εισόδου σήματος με μέγεθος 1 bit το οποίο προέρχεται από την Αριθμητική και Λογική Μονάδα. Αν το σήμα είναι ίσο με 1, αυτό σημαίνει πως η συνθήκη της εντολής διακλάδωσης ικανοποιείται.
* jump: Θύρα εισόδου σήματος με μέγεθος 2 bit το οποίο προέρχεται από την Μονάδα Ελέγχου. Αν είναι ίσο με 00 τότε αυτό σημαίνει πως η εντολή που βρίσκεται στο στάδιο EX δεν είναι εντολή διακλάδωσης επομένως η ΜΔΔ δεν θα επηρεάσει τη ροή του προγράμματος.

Η ΜΔΔ διαθέτει τις θύρες εξόδου:

* ΙF\_Flush: Θύρα εξόδου σήματος με μέγεθος 1 bit το οποίο ισούται με 1 σε 2 περιπτώσεις. Πρώτον, όταν ικανοποιείται η συνθήκη της εντολής διακλάδωσης και δεύτερον όταν η εντολή είναι εντολή άλματος. Όταν λοιπόν ισούται με 1 τότε το περιεχόμενο του ενδιάμεσου καταχωρητή IF\_ID δεν διαδίδεται περαιτέρω καθώς περιέχει πληροφορίες για την εκτέλεση μιας εντολής (η δεύτερη μετά την εντολή διακλάδωσης) που δεν πρέπει να εκτελεστεί αφού η ροή του προγράμματος θα αλλάξει.
* ΙD\_Flush: Θύρα εξόδου σήματος με μέγεθος 1-bit το οποίο ισούται με 1 σε 2 περιπτώσεις. Πρώτον, όταν ικανοποιείται η συνθήκη της εντολής διακλάδωσης και δεύτερον όταν η εντολή είναι εντολή άλματος. Όταν λοιπόν ισούται με 1 τότε το περιεχόμενο του ενδιάμεσου καταχωρητή ID\_EX δεν διαδίδεται περαιτέρω καθώς περιέχει πληροφορίες για την εκτέλεση μιας εντολής (η πρώτη μετά την εντολή διακλάδωσης) που δεν πρέπει να εκτελεστεί αφού η ροή του προγράμματος θα αλλάξει.
* mux\_to\_pc: Θύρα εξόδου σήματος με μέγεθος 2 bits το οποίο οδηγεί τον πολυπλέκτη που τοποθετείται πριν τον Μετρητή Προγράμματος και καθορίζει το δεδομένο με το οποίο θα τροφοδοτηθεί ο MΠ.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, διάγραμμα

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.6: Μονάδα Διαχείρισης Διακλαδώσεων

### 3.2.9 Μονάδα Παροχέτευσης (Forwarding Unit)

Η Μονάδα Παροχέτευσης (ΜΠαρ) είναι η μονάδα που αναλαμβάνει να αντιμετωπίσει τις εξαρτήσεις από δεδομένα. Στην τεχνική των μερικώς επικαλυπτόμενων λειτουργιών, εξαρτήσεις από δεδομένα δημιουργούνται όταν μία εντολή αποθηκεύει ένα αποτέλεσμα σε ένα καταχωρητή και στη συνέχεια οι εντολές που ακολουθούν αναγιγνώσκουν το περιεχόμενο του ίδιου καταχωρητή. Αυτό συμβαίνει διότι η εγγραφή ενός καταχωρητή γίνεται στο στάδιο WB. Όταν η εντολή που αποθηκεύει σε ένα καταχωρητή βρίσκεται σε προγενέστερο στάδιο από το WB αυτό έχει ως αποτέλεσμα οι εντολές που ακολουθούν και επιθυμούν να κάνουν ανάγνωση του περιεχομένου του ίδιου καταχωρητή να μην διαβάσουν το νέο και έγκυρο περιεχόμενο διότι δεν θα έχει αποθηκευτεί ακόμη. Η λύση στο πρόβλημα δίνεται από την τεχνική της παροχέτευσης.

Η ΜΠαρ διαθέτει τις θύρες εισόδου:

* EX\_MEMregWrite: Θύρα εισόδου σήματος μεγέθους 1 bit που αντιπροσωπεύει το σήμα RegWrite από τη Μονάδα Ελέγχου για την εντολή που βρίσκεται στο στάδιο MEM.
* MEM\_WBregWrite: Θύρα εισόδου σήματος μεγέθους 1 bit που αντιπροσωπεύει το σήμα RegWrite από τη Μονάδα Ελέγχου για την εντολή που βρίσκεται στο στάδιο WB.
* ID\_EXrs1, ID\_EXrs2: Θύρες εισόδου σημάτων μεγέθους 5 bits που αντιπροσωπεύουν το περιεχόμενο των καταχωρητών rs1, rs2 τους οποίους έχει διαβάσει η εντολή που βρίσκεται στο στάδιο EX.
* EX\_MEMrd: Θύρα εισόδου σήματος μεγέθους 5 bits που αντιπροσωπεύει τον καταχωρητή που στοχεύει να εγγράψει η εντολή που βρίσκεται στο στάδιο MEM.
* MEM\_WBrd: Θύρα εισόδου σήματος μεγέθους 5 bits που αντιπροσωπεύει τον καταχωρητή που στοχεύει να εγγράψει η εντολή που βρίσκεται στο στάδιο WB.

Η ΜΠαρ διαθέτει τις θύρες εξόδου:

* FW0: Θύρα εξόδου σήματος μεγέθους 2 bits το οποίο οδηγεί τον πολυπλέκτη FW0 ο οποίος καθορίζει από ποιά βαθμίδα θα παρθεί το αποτέλεσμα που πρόκειται να εγγραφεί σε καταχωρητή. Διότι αυτό το αποτέλεσμα θα είναι το έγκυρο.
* FW1: Θύρα εξόδου σήματος μεγέθους 2 bits το οποίο οδηγεί τον πολυπλέκτη FW1 ο οποίος καθορίζει από ποιά βαθμίδα θα παρθεί το αποτέλεσμα που πρόκειται να εγγραφεί σε καταχωρητή. Διότι αυτό το αποτέλεσμα θα είναι το έγκυρο.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμμή, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.7: Μονάδα Παροχέτευσης

### 3.2.10 Κρυφή Μνήμη Δεδομένων (Data Cache Memory)

Η Κρυφή Μνήμη Δεδομένων (ΔΜΕ) είναι μία κρυφή μνήμη με οργάνωση μονοσήμαντης απεικόνισης (Direct Mapping). Σε αυτή τη μνήμη γίνεται είτε εγγραφή είτε ανάγνωση δεδομένων. Ο τρόπος χρήσης των διευθύνσεων που λαμβάνει η ΚΜΔ είναι ο εξής:

31 8 7 3 2 0

|  |  |  |
| --- | --- | --- |
| tag | index | offset |

Το διάστημα offset καταλαμβάνει τα 3 λιγότερο σημαντικά bits της διεύθυνσης και προσδιορίζει τη διεύθυνση μιας λέξης μέσα στο πλαίσιο της κρυφής μνήμης. Στην συγκεκριμένη περίπτωση το πλαίσιο αποτελείται από 8 λέξεις. Το διάστημα index καταλαμβάνει τα 5 αμέσως επόμενα bits της διεύθυνσης και προσδιορίζει τη διεύθυνση του πλαισίου μέσα στην κρυφή μνήμη, η κρυφή μνήμη αποτελείται από 32 πλαίσια. Το διάστημα tag καταλαμβάνει τα υπόλοιπα 24 bits τα οποία αποτελούν την ετικέτα κάθε διεύθυνσης.

Η ΚΜΔ διαθέτει τις παρακάτω θύρες εισόδου:

* addr: Θύρα εισόδου σήματος για την διεύθυνση προσπέλασης της μνήμης, με μέγεθος 32 bits.
* MemWrite: Θύρα εισόδου σήματος με μέγεθος 1 bit το οποίο όταν ισούται με 1 τότε θα γίνει εγγραφή της τιμής του σήματος WriteData στη θέση που υποδεικνύει το σήμα addr.
* MemRead: Θύρα εισόδου σήματος με μέγεθος 1 bit το οποίο όταν ισούται με 1 τότε θα γίνει ανάγνωση του περιεχομένου της θέσης που υποδεικνύει το σήμα addr.
* WriteData: Θύρα εισόδου σήματος με μέγεθος 32 bits το οποίο δίνει την τιμή που εγγραφεί στην διεύθυνση της μνήμης που έχει ορίσει το σήμα addr.
* funct3: Θύρα εισόδου σήματος με μέγεθος 3 bits το οποίο αντιπροσωπεύει το πεδίο funct3 που αναφέρθηκε στον Πίνακα 1. Το σήμα αυτό χρησιμοποιείται για να καθοριστεί η παραλλαγή των εντολών τύπου load ή store (π.χ. load byte, store halfword).

Η ΚΜΕ διαθέτει την θύρα εξόδου:

* output\_data: Θύρα εξόδου σήματος για τα δεδομένα που διαβάζονται από την μνήμη, με μέγεθος 32 bits.

## 3.3 Τελικός σχεδιασμός

Ακολουθώντας την τεχνική των μερικώς επικαλυπτόμενων λειτουργιών, σε συνδυασμό με τις μονάδες που αναφέρθηκαν στην ενότητα [3.2](#_3.2_Λειτουργικές_μονάδες) και τις εντολές του συνόλου RV32I καταλήγουμε σε ένα τελικό σχεδιασμό. Για να είναι ολοκληρωμένος και λειτουργικός ο σχεδιασμός χρειάζονται επιπλέον στοιχεία λογικού σχεδιασμού. Στην Εικόνα 3.13 φαίνεται ο τελικός σχεδιασμός. Ακολουθεί σχολιασμός των σταδίων της τεχνικής των μερικώς επικαλυπτόμενων λειτουργιών.

### 3.3.1 Στάδιο IF

Αρχικά, σε αυτό το στάδιο υπάρχει ένας πολυπλέκτης 4 σε 1 των 32 bits. Ο πολυπλέκτης τροφοδοτεί τον Μετρητή Προγράμματος (ΜΠ) με την κατάλληλη τιμή. Στην είσοδο 0 οδηγείται μια τιμή που προκύπτει από την πρόσθεση (Adder A0) της τιμής που έχει ο ΜΠ στον προηγούμενο κύκλο ρολογιού με τον κατάλληλο βαθμωτό αριθμό ώστε να διαβαστεί από την Κρυφή Μνήμη Εντολών η επομένη κατά σειρά εντολή του προγράμματος. Στην είσοδο 1 οδηγείται μια τιμή που όπως θα δούμε παρακάτω υπολογίζεται στο στάδιο ID και είναι η διεύθυνση διακλάδωσης για την περίπτωση που εκτελείται μία εντολή διακλάδωσης. Στην είσοδο 2 οδηγείται μια τιμή που όπως θα δούμε παρακάτω υπολογίζεται στο στάδιο EX και πρόκειται για την τιμή που πρέπει να λάβει ο ΜΠ όταν εκτελείται η εντολή JALR.

Επιπλέον, στο στάδιο IF διαβάζεται η [Κρυφή Μνήμη Εντολών](#_3.2.1_Κρυφή_Μνήμη) η οποία τροφοδοτεί τον ενδιάμεσο καταχωρητή IF-ID με την κωδικοποιημένη εντολή που τίθεται προς εκτέλεση.

Εικόνα που περιέχει διάγραμμα, τεχνικό σχέδιο, Σχέδιο, σχηματικό

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.8: Στάδιο IF

### 3.3.2 Στάδιο ID

Στο στάδιο ID υπάρχει η Μονάδα Ελέγχου η οποία αποκωδικοποιεί την εντολή. Ο Adder A1 που φαίνεται αναλαμβάνει τον υπολογισμό της διεύθυνσης διακλάδωσης. Επίσης, υπάρχουν οι Καταχωρητές Γενικού Σκοπού (ΚΓΣ) που δεν ανήκουν αποκλειστικά στο στάδιο ID καθώς όπως θα αναφερθεί παρακάτω μπορεί να γίνει εγγραφή των ΚΓΣ στο στάδιο WB. Στους ΚΓΣ πραγματοποιείται ανάγνωση τους από την εντολή που βρίσκεται στο στάδιο ID. Η τελευταία μονάδα σε αυτό το στάδιο είναι η Μονάδα Παραγωγής Άμεσων Δεδομένων η οποία παράγει το άμεσο δεδομένο που ορίζεται από κάθε εντολή της οποίας η κωδικοποίηση περιέχει άμεσα δεδομένα. Η Μονάδα Ανίχνευσης Εξαρτήσεων δεν ανήκει αποκλειστικά σε ένα στάδιο διότι τα σήματα που λαμβάνει ως είσοδο προέρχονται από δύο διαφορετικά στάδια (ID, EX).

Εικόνα που περιέχει κείμενο, διάγραμμα, Σχέδιο, τεχνικό σχέδιο

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.9: Στάδιο ID.

### 3.3.3 Στάδιο EX

Στο στάδιο EX υπάρχει η Αριθμητική και Λογική Μονάδα (ΑΛΜ) η οποία αναλαμβάνει την εκτέλεση πράξεων και συγκρίσεων μεταξύ αριθμών. Επιπλέον, υπάρχει η Μονάδα Ελέγχου ΑΛΜ η οποία παράγει ένα σήμα το οποίο υποδεικνύει στην ΑΛΜ την πράξη ή την σύγκριση που θα εκτελέσει. Οι πολυπλέκτες 4 σε 1 με ονόματα FW0, FW1 ελέγχονται από την Μονάδα Παροχέτευσης και καθορίζουν την τιμή που θα διαδοθεί, αναλόγως του αν υπάρχει κάποια εξάρτηση δεδομένων. Ο πολυπλέκτης 2 σε 1 Μ0 ελέγχεται από ένα σήμα που παράγει η Μονάδα Ελέγχου στο στάδιο ID και καθορίζει αν θα διαδοθεί το περιεχόμενο ενός καταχωρητή ή ένα άμεσο δεδομένο ανάλογα με τον τύπο της εντολής που εκτελείται. Ο Adder A2 αναλαμβάνει την πρόσθεση του άμεσου δεδομένου και του περιεχομένου που είχε ο Μετρητής Προγράμματος όταν άρχισε να εκτελείται η εντολή που βρίσκεται στο στάδιο EX. Το αποτέλεσμα της πρόσθεσης είναι απαραίτητο όταν εκτελείται η εντολή AUIPC. Ο πολυπλέκτης 2 σε 1 M1 ελέγχεται από ένα σήμα που παράγει η Μονάδα Ελέγχου και καθορίζει το δεδομένο που θα διαδοθεί, ανάλογα με το αν εκτελείται η εντολή LUI ή AUIPC. Η Μονάδα Διαχείρισης Διακλαδώσεων η οποία παράγει τα κατάλληλα σήματα όταν η εντολή που εκτελείται ανήκει στις εντολές διακλάδωσης. Η Μονάδα Παροχέτευσης δεν ανήκει αποκλειστικά σε ένα στάδιο διότι τα σήματα που λαμβάνει ως είσοδο προέρχονται από τρία διαφορετικά στάδια (EX, MEM, WB).

Εικόνα που περιέχει διάγραμμα, Σχέδιο, τεχνικό σχέδιο, σχηματικό

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.10: Στάδιο EX

### 3.3.4 Στάδιο MEM

Στο στάδιο MEM γίνεται ανάγνωση ή εγγραφή της Κρυφής Μνήμης Δεδομένων.

Εικόνα που περιέχει διάγραμμα, τεχνικό σχέδιο, Σχέδιο, γραμμή

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.11: Στάδιο MEM

### 3.3.4 Στάδιο WB

Στο στάδιο WB υπάρχει ένας πολυπλέκτης 4 σε 1 που καθορίζει το δεδομένο που θα σταλεί στους Καταχωρητές Γενικού Σκοπού ανάλογα με το είδος της εντολής που εκτελείται.

Εικόνα που περιέχει διάγραμμα, τεχνικό σχέδιο, γραμμή, Σχέδιο

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.12: Στάδιο WB

Εικόνα που περιέχει διάγραμμα, Σχέδιο, τεχνικό σχέδιο, σχηματικό

Περιγραφή που δημιουργήθηκε αυτόματα



## 3.4 Οι διαδικασίες Synthesis και Implementation

Οι διαδικασίες Synthesis [[5](#_8._ΒΙΒΛΙΟΓΡΑΦΙΑ)] και Implementation [[6](#_8._ΒΙΒΛΙΟΓΡΑΦΙΑ)] εκτελέστηκαν με στόχο τον προσδιορισμό της μεγαλύτερης συχνότητας που μπορεί να επιτευχθεί για το ρολόι χρονισμού που θα εφαρμοστεί στο σχεδιασμό. Οι δύο διαδικασίες είναι αλληλένδετες και για να ξεκινήσει η Implementation πρέπει να έχει ολοκληρωθεί επιτυχώς η Synthesis. Το FPGA που επιλέχθηκε ανήκει στην οικογένεια Spartan 7. Για να προσδιοριστεί η μέγιστη συχνότητα χρειάζεται να γίνει μια επαναληπτική διαδικασία όπου ο χρήστης σε κάθε επανάληψη των δύο διαδικασιών ορίζει μια συγκεκριμένη περίοδο ρολογιού (time constraints) την οποία και αυξάνει κάθε φορά. Στην επαναληπτική διαδικασία που έγινε για τον σχεδιασμό, τα αποτελέσματα ήταν τα εξής:

Εικόνα που περιέχει κείμενο, γραμματοσειρά, γραμμή, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.13: Χρονικά αποτελέσματα σχεδιασμού

Το Worst Negative Slack (WNS) είναι ουσιαστικά το Setup Slack οποίο πρέπει να είναι μεγαλύτερο ή ίσο του μηδενός. Ορίζεται ως η διαφορά Available Time μείον Arrival Time. Με τον όρο Available Time εννοείται ο χρόνος που είναι διαθέσιμος για να φτάσει ένα σήμα στον προορισμό του και ισούται με τη διαφορά της περιόδου του ρολογιού μείον το setup time (o ελάχιστος χρόνος που πρέπει η τιμή μιας εισόδου να είναι σταθερή πριν την ακμή του ρολογιού). Με τον όρο Arrival Time εννοείται ο πραγματικός χρόνος που χρειάζεται το σήμα για να φτάσει στον προορισμό του. Επομένως το WNS θα είναι μεγαλύτερο ή ίσο του μηδενός όταν το Available Time θα είναι μεγαλύτερο ή ίσο με το Arrival Time.

Το Worst Hold Slack (WHS) πρέπει και αυτό να είναι μεγαλύτερο ή ίσο του μηδενός. Είναι ουσιαστικά το Hold Slack και ορίζεται ως η διαφορά Arrival Time μείον Required Time. Με τον όρο Arrival Time εννοείται ο πραγματικός χρόνος που χρειάζεται το σήμα για να φτάσει στον προορισμό του. Με το όρο Required Time εννοείται το hold time (o ελάχιστος χρόνος που πρέπει η τιμή μιας εισόδου να είναι σταθερή μετά την ακμή του ρολογιού).

Ένα ακόμη αποτέλεσμα των διαδικασιών είναι η αναφορά για την κατανάλωση ενέργειας που υπάρχει στο FPGA. Στην εικόνα 3.14 φαίνονται η δυναμική και στατική κατανάλωση ενέργειας. Η δυναμική κατανάλωση αφορά το σχεδιασμό του χρήστη, ενώ η στατική αφορά το FPGA το οποίο απαιτεί ενέργεια για την ομαλή λειτουργία του.

Εικόνα που περιέχει κείμενο, γραμματοσειρά, αριθμός, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.14: Αποτελέσματα κατανάλωσης ενέργειας.

Τελικά, επιτυγχάνεται μέγιστη συχνότητα ρολογιού 289,855 MHz. Αυτή η συχνότητα προκύπτει από την περίοδο του ρολογιού που οριζόταν ως στόχος σε κάθε επανάληψη των διαδικασιών. Η μικρότερη συχνότητα που ικανοποιούσε όλες τις απαιτήσεις ήταν T = 3,45 ns. Ακολουθεί η Εικόνα 3.15 που επιβεβαιώνει τα παραπάνω.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, γραμμή

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3.15: Αποτελέσματα μέγιστης συχνότητας.

# 4. ΕΠΑΛΗΘΕΥΣΗ ΣΧΕΔΙΑΣΜΟΥ

Σε αυτό το κεφάλαιο θα γίνει επιβεβαίωση της σωστής λειτουργίας του επεξεργαστή μέσω απλών προγραμμάτων που περιέχουν όλες τις εξαρτήσεις που είναι δυνατόν να προκύψουν.

## 4.1 Εξαρτήσεις από δεδομένα

Σε αυτή την ενότητα θα παρουσιαστούν όλες οι εξαρτήσεις δεδομένων που αντιμετωπίζονται από τον μηχανισμό των μερικώς επικαλυπτόμενων λειτουργιών. Ένας τύπος εξάρτησης από δεδομένα είναι η Ανάγνωση μετά από Εγγραφή (ΑμΕ) η οποία προκύπτει όταν μια εντολή που διαβάζει ένα καταχωρητή εκτελείται μετά από μια εντολή που γράφει στον ίδιο καταχωρητή και μεταξύ αυτών των εντολών παρεμβάλλεται καμία ή μια εντολή. Ένας άλλος τύπος εξάρτησης από δεδομένα είναι η Εγγραφή μετά από Εγγραφή (ΕμΕ) η οποία όμως στην παρούσα υλοποίηση του μηχανισμού των μερικώς επικαλυπτόμενων λειτουργιών δεν προκύπτει επειδή γίνεται εγγραφή στους καταχωρητές ή στην κρυφή μνήμη εντολών σε μία μόνο βαθμίδα. Ακόμη ένας τύπος είναι η Εγγραφή μετά από Ανάγνωση (ΕμΑ) η οποία και αυτή δεν προκύπτει στην παρούσα υλοποίηση καθώς η ανάγνωση δεδομένων γίνεται σε προγενέστερο στάδιο από το στάδιο που γίνεται η εγγραφή δεδομένων.

### 4.1.1 Ανάγνωση μετά από Εγγραφή (ΑμΕ)

Παρακάτω ακολουθεί το πρόγραμμα που θα εκτελεστεί. Ο καταχωρητής x1 περιέχει την δεκαδική τιμή 2, ο x3 την δεκαδική τιμή 1 και ο x5 την δεκαδική τιμή 4. Όλοι οι υπόλοιποι έχουν αρχικοποιηθεί στην τιμή 0.

|  |  |
| --- | --- |
| Πρόγραμμα | Κωδικοποίηση (hex) |
| SUB x2, x1, x3  AND x12, x2, x5  OR x13, x6, x2  ADD x14, x2, x2 | 40308133  00517633  002366b3  00210733 |

Η πρώτη εξάρτηση εντοπίζεται μεταξύ της εντολής SUB και της AND αφού η SUB αποθηκεύει το αποτέλεσμα της στον x2 και η AND χρησιμοποιεί αυτό το αποτέλεσμα. Η δεύτερη εξάρτηση εντοπίζεται μεταξύ της εντολής SUB και της OR για τον ίδιο λόγο με παραπάνω. Οι εξαρτήσεις αυτές προκύπτουν διότι οι AND και OR φθάνουν στο στάδιο ID όπου γίνεται ανάγνωση των καταχωρητών προτού η SUB φτάσει στο στάδιο WB. Μεταξύ των εντολών SUB και ADD δεν υπάρχει εξάρτηση διότι η SUB βρίσκεται στο στάδιο WB όταν η ADD βρίσκεται στο ID. Δεν προκύπτει εξάρτηση διότι στους καταχωρητές γενικού σκοπού η εγγραφή των καταχωρητών προηγείται της ανάγνωσης αυτών, μέσα στον ίδιο κύκλο ρολογιού.

Αποτελέσματα εξομοίωσης:

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

Το σήμα time οδηγείται από το ρολόι. Τις χρονικές στιγμές 100, 200 έως και 500 το ρολόι βρίσκεται στην θετική του ακμή. Ενώ τις 50, 150 έως και 550 βρίσκεται στην αρνητική ακμή. Ισχύει ότι στην αρνητική ακμή του ρολογιού έχει επιλεγεί ο Μετρητής Προγράμματος και οι ενδιάμεσοι καταχωρητές να διαδίδουν το περιεχόμενό τους στην επόμενη βαθμίδα.

Ανάλυση αποτελεσμάτων εξομοίωσης:

Αρχικά, τη χρονική στιγμή 100 διαβάζεται η κωδικοποίηση της εντολής SUB x2, x1, x3 επομένως η SUB βρίσκεται στο στάδιο IF. Την 150 η SUB περνάει στο στάδιο ID και την 250 στο στάδιο EX, όπου εκεί γίνεται η εκτέλεση της πράξης. Το αποτέλεσμα της πράξης φαίνεται στο σήμα Alu\_res το οποίο στην περίπτωση της SUB είναι το αναμενόμενο (2 – 1 = 1). Τη χρονική στιγμή 350 η εντολή AND περνάει στο στάδιο EX και όπως φαίνεται το σήμα data0 λαμβάνει την σωστή τιμή χάρη στη λειτουργία της Μονάδας Παροχέτευσης η οποία θέτει το σήμα fw0 στο 10. Τη χρονική στιγμή 450 η εντολή OR περνάει στο στάδιο EX και όπως φαίνεται το σήμα data1 λαμβάνει την σωστή τιμή χάρη στη λειτουργία της Μονάδας Παροχέτευσης η οποία θέτει το σήμα fw1 στο 01. Τη χρονική στιγμή 550 η εντολή ADD περνάει στο στάδιο EX και όπως φαίνεται τα σήματα data0, data1 λαμβάνουν τη σωστή τιμή η οποία διαβάζεται από τους καταχωρητές γενικού σκοπού.

### 4.1.2 Ακολουθία εντολών load-use

Παρακάτω ακολουθεί το πρόγραμμα που θα εκτελεστεί. Ο καταχωρητής x1 περιέχει την δεκαδική τιμή 2, ο x2 την δεκαδική τιμή 6, ο x3 την δεκαδική τιμή 1 και ο x5 την δεκαδική τιμή 4. Όλοι οι υπόλοιποι έχουν αρχικοποιηθεί στην τιμή 0.

|  |  |
| --- | --- |
| Πρόγραμμα | Κωδικοποίηση (hex) |
| LB x3, 0(x8)  ORI x10, x3, 2  SLT x9, x2, x1 | 00040183  0021e513  001124b3 |

Σε αυτή την περίπτωση η εντολή LB x3, 0(x8) αποθηκεύει στον καταχωρητή x3 και η εντολή ORI x10, x3, 2 διαβάζει από αυτόν, με αποτέλεσμα να προκύπτει μία εξάρτηση δεδομένων. Η τιμή που θα αποθηκευτεί στον x3 εμφανίζεται πρώτη φορά τη στιγμή που η εντολή LB βρίσκεται στο στάδιο MEM διότι τότε γίνεται ανάγνωση της κρυφής μνήμης δεδομένων, όμως αυτή η τιμή χρειάζεται να γίνει διαθέσιμη την ίδια στιγμή στο στάδιο EX όπου εκεί βρίσκεται η εντολή ORI. Επομένως, είναι αναγκαίο μόλις αναγνωριστεί αυτή η εξάρτηση να «παγώσουν» οι βαθμίδες του μηχανισμού πριν το στάδιο MEM για ένα κύκλο ρολογιού, δηλαδή να καθυστερήσει η εκτέλεση των εντολών ORI, SLT.

Αποτελέσματα εξομοίωσης:

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, Μπελ ηλεκτρίκ

Περιγραφή που δημιουργήθηκε αυτόματα

Το σήμα time οδηγείται από το ρολόι. Τις χρονικές στιγμές 100, 200 έως και 500 το ρολόι βρίσκεται στην θετική του ακμή. Ενώ τις 50, 150 έως και 550 βρίσκεται στην αρνητική ακμή. Ισχύει ότι στην αρνητική ακμή του ρολογιού έχει επιλεγεί ο Μετρητής Προγράμματος και οι ενδιάμεσοι καταχωρητές να διαδίδουν το περιεχόμενό τους στην επόμενη βαθμίδα.

Ανάλυση αποτελεσμάτων εξομοίωσης:

Αρχικά, τη χρονική στιγμή 100 η εντολή LB βρίσκεται στο στάδιο IF διότι τότε διαβάζεται από την κρυφή μνήμη δεδομένων. Τη χρονική στιγμή 250 η εντολή LB περνάει στο στάδιο EX και η εντολή ORI στο στάδιο ID. Σε αυτή τη στιγμή η Μονάδα Ανίχνευσης Εξαρτήσεων ανιχνεύει τη συγκεκριμένη εξάρτηση με αποτέλεσμα να οδηγήσει τα σήματα PCwrite, IF\_IDwrite, ID\_EXwrite στο 0 για ένα κύκλο ρολογιού. Το οποίο σημαίνει ότι θα σταματήσει η εκτέλεση των εντολών που ακολουθούν μετά την LB.

## 4.2 Διαδικασιακές εξαρτήσεις

Οι διαδικασιακές εξαρτήσεις προκύπτουν όταν υπάρχει μια εντολή διακλάδωσης στην ροή εκτέλεσης του προγράμματος. Στην παρούσα υλοποίηση έχει θεωρηθεί ότι όταν εισέρχεται στο μηχανισμό μία εντολή διακλάδωσης τότε η ροή εκτέλεσης δεν θα αλλάζει. Η διεύθυνση διακλάδωσης υπολογίζεται στο στάδιο ID και ο έλεγχος της συνθήκης γίνεται στο στάδιο EX. Η απόφαση να μην αλλάζει η ροή εκτέλεσης έχει ως αποτέλεσμα οι εντολές μετά την εντολή διακλάδωσης να εφαρμόζουν αλλαγές στις τιμές των ενδιάμεσων καταχωρητών. Επομένως, κρίνεται αναγκαίο όταν η συνθήκη μιας εντολής διακλάδωσης ικανοποιείται, οι αλλαγές που έχει επιφέρει η εκτέλεση των επόμενων εντολών στους ενδιάμεσους καταχωρητές IF\_ID, ID\_EX να απαλειφθούν και να προσκομιστεί η εντολή που υποδεικνύει η διεύθυνση διακλάδωσης.

### 4.2.1 Εντολές διακλάδωσης υπό συνθήκη

Παρακάτω ακολουθεί το πρόγραμμα που θα εκτελεστεί. Ο καταχωρητής x1 περιέχει την δεκαδική τιμή 2, ο x2 την δεκαδική τιμή 6, ο x3 την δεκαδική τιμή 1 και ο x5 την δεκαδική τιμή 4. Όλοι οι υπόλοιποι έχουν αρχικοποιηθεί στην τιμή 0.

|  |  |
| --- | --- |
| Πρόγραμμα | Κωδικοποίηση (hex) |
| LB x3, 0(x8)  BNE x2, x18, 4  OR x20, x10, x5  XOR x4, x0, x3  LW x1, 5(x2)  XORI x3, x7, 4  OR x20, x10, x5  SLL x8, x9, x15 | 00040183  01211263  00556a33  00304233  00513083  0043c193  00556a33  00f49433 |

Σε αυτό το πρόγραμμα η εντολή BNE x2, x18, 4 είναι μια εντολή διακλάδωσης υπό συνθήκη και η συνθήκη της ικανοποιείται διότι ο καταχωρητής x2 έχει αποθηκευμένη διαφορετική τιμή από τον καταχωρητή x18. Επομένως, οι αλλαγές που θα εφαρμόσουν οι εντολές OR x20, x10, x5 και XOR x4, x0, x3 θα πρέπει να ακυρωθούν.

Αποτελέσματα εξομοίωσης:

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Το σήμα time οδηγείται από το ρολόι. Τις χρονικές στιγμές 100, 200 έως και 500 το ρολόι βρίσκεται στην θετική του ακμή. Ενώ τις 50, 150 έως και 550 βρίσκεται στην αρνητική ακμή. Ισχύει ότι στην αρνητική ακμή του ρολογιού έχει επιλεγεί ο Μετρητής Προγράμματος και οι ενδιάμεσοι καταχωρητές να διαδίδουν το περιεχόμενό τους στην επόμενη βαθμίδα.

Ανάλυση αποτελεσμάτων εξομοίωσης:

Αρχικά, τη χρονική στιγμή 200 η εντολή BNE βρίσκεται στο στάδιο IF διότι τότε διαβάζεται από την κρυφή μνήμη δεδομένων. Tη χρονική στιγμή 350 η BNE εισέρχεται στο στάδιο EX όπου εκεί γίνεται ο έλεγχος της συνθήκης και το σήμα branch λαμβάνει την τιμή 1 διότι η συνθήκη ικανοποιείται. Επίσης, παρατηρείται πως στο μηχανισμό έχουν εισέλθει οι δυο εντολές που ακολουθούν την BNE όμως οι αλλαγές που προκάλεσαν δεν διαδίδονται ποτέ στο στάδιο EX κάτι που διαπιστώνουμε κοιτώντας τις εισόδους της Αριθμητικής και Λογικής Μονάδας τη χρονική στιγμή 450. Τα σήματα data0 και data1 λαμβάνουν μηδενική τιμή κάτι που δεν θα συνέβαινε αν διαδίδονταν οι αλλαγές της εντολής OR x20, x10, x5 καθώς αυτή η εντολή θα έκανε το σήμα data1 να λάβει την δεκαδική τιμή 4 η οποία είναι αποθηκευμένη στον καταχωρητή x4. Τη χρονική στιμή 500 διαβάζεται από την κρυφή μνήμη εντολών οι εντολή XORI x3, x7, 4 που αναμενόμενα απέχει κατά 4 από την εντολή διακλάδωσης.

### 4.2.2 Εντολές άλματος

Παρακάτω ακολουθεί το πρόγραμμα που θα εκτελεστεί. Ο καταχωρητής x1 περιέχει την δεκαδική τιμή 2, ο x2 την δεκαδική τιμή 6, ο x3 την δεκαδική τιμή 1 και ο x5 την δεκαδική τιμή 4. Όλοι οι υπόλοιποι έχουν αρχικοποιηθεί στην τιμή 0.

|  |  |
| --- | --- |
| Πρόγραμμα | Κωδικοποίηση (hex) |
| LB x3, 0(x8)  JAL x7, 4  OR x20, x10, x5  XOR x4, x0, x3  LW x1, 5(x2)  XORI x3, x7, 4  OR x20, x10, x5  SLL x8, x9, x15 | 00040183  004003ef  00556a33  00304233  00513083  0043c193  00556a33  00f49433 |

Σε αυτό το πρόγραμμα η εντολή JAL x7, 4 είναι μια εντολή άλματος και όταν φτάσει στο στάδιο EX τότε θα αλλάξει η ροή του προγράμματος. Επομένως, οι αλλαγές που θα εφαρμόσουν οι εντολές OR x20, x10, x5 και XOR x4, x0, x3 θα πρέπει να ακυρωθούν.

Αποτελέσματα εξομοίωσης:

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, Μπελ ηλεκτρίκ

Περιγραφή που δημιουργήθηκε αυτόματα

Ανάλυση αποτελεσμάτων εξομοίωσης:

Αρχικά, τη χρονική στιγμή 200 η εντολή JAL βρίσκεται στο στάδιο IF διότι τότε διαβάζεται από την κρυφή μνήμη δεδομένων. Tη χρονική στιγμή 350 η JAL εισέρχεται στο στάδιο EX όπου εκεί το σήμα jump λαμβάνει την τιμή 1. Τη χρονική στιγμή 550 η JAL φτάνει στο στο στάδιο WB όπου στόχος είναι η αποθήκευση της διεύθυνσης της αμέσως επόμενης εντολής OR x20, x10, x5 στον καταχωρητή x7. Επίσης, παρατηρείται πως στο μηχανισμό έχουν εισέλθει οι δυο εντολές που ακολουθούν την JAL όμως οι αλλαγές που προκάλεσαν δεν διαδίδονται ποτέ στο στάδιο EX. Τη χρονική στιμή 500 διαβάζεται από την κρυφή μνήμη εντολών οι εντολή XORI x3, x7, 4 που αναμενόμενα απέχει κατά 4 από την εντολή άλματος.

# 5. ΜΕΛΛΟΝΤΙΚΕΣ ΒΕΛΤΙΩΣΕΙΣ

Στην παρούσα εργασία οι υλοποιήσεις που αναπτύχθηκαν χρησιμεύουν για την εξομοίωση της εκτέλεσης εντολών του υποσυνόλου RV32I και παρουσίαση των σταδίων εκτέλεσης μιας εντολής μέσα στο μηχανισμό των μερικώς επικαλυπτόμενων λειτουργιών. Τα παραπάνω συνθέτουν ένα λειτουργικό μοντέλο το οποίο όμως μπορεί να εμπλουτιστεί στο μέλλον για να αποκτήσει περισσότερα χαρακτηριστικά.

## 5.1 Προσθήκη Επεκτάσεων

Όπως αναφέρθηκε στην υποενότητα [1.3](#_1.3_Αρχιτεκτονική_RISC-V) υπάρχουν επεκτάσεις του συνόλου εντολών της αρχιτεκτονικής RISC V. Θα μπορούσαν να αναπτυχθούν υλοποιήσεις οι οποίες θα καθιστούσαν δυνατή την εκτέλεση εντολών που ανήκουν σε διάφορες επεκτάσεις, όπως είναι η επέκταση «Μ» που προβλέπει την εκτέλεση των πράξεων του πολλαπλασιασμού και διαίρεσης.

## 5.2 Προσθήκη Κύριας Μνήμης

Οι μνήμες που έχουν υλοποιηθεί αποτελούν Κρυφές Μνήμες (cache memories). Θα μπορούσε να προστεθεί μια Κύρια Μνήμη (RAM) η οποία θα συνεργάζεται με τις Κρυφές Μνήμες Εντολών και Δεδομένων. Για τη συνεργασία της Κύριας Μνήμης με τις Κρυφές είναι απαραίτητη η επιλογή μιας στρατηγικής για την απελευθέρωση πλαισίων των Κρυφών Μνημών ώστε να προσκομιστούν μπλοκ της Κύριας Μνήμης.

## 5.3 Υλοποίηση τεχνικής πραγματικής πρόβλεψης για τις εντολές διακλάδωσης

Στον μηχανισμό που υλοποιήθηκε έχει υιοθετηθεί η τεχνική ψευτοπρόβλεψης και θεωρείται ότι η ροή του προγράμματος δεν αλλάζει. Θα μπορούσε να υιοθετηθεί μια τεχνική πραγματικής πρόβλεψης η οποία να προσέφερε καλύτερη απόδοση.

# 6. ΥΛΟΠΟΙΗΣΗ

## 6.1 Αρχεία Verilog που περιγράφουν το Υλικό

### 6.1.1 Περιγραφή Μετρητή Προγράμματος – PC.v

1. module PC (clk, reset, hazDetect\_PC, pc\_input, pc\_output);
3. input clk, reset, hazDetect\_PC;
4. input [31:0] pc\_input;
5. output reg [31:0] pc\_output;
6. always @(negedge clk) begin //posedge
7. if(hazDetect\_PC) begin
8. if(reset) begin
9. pc\_output <= 32'b0;
10. end
11. else begin
12. pc\_output <= pc\_input;
13. end
14. end
15. end
17. endmodule

### 6.1.2 Περιγραφή Κρυφής Μνήμης Εντολών – InstCache.v

1. module InstCache (clk, addr, reset, inst);
2. input [31:0] addr;
3. input clk, reset;
5. output reg [31:0] inst;
6. wire [255:0] block;
8. //cache modules, 1KByte TAG\_WIDTH = 24, INDEX\_WIDTH = 5, OFFSET\_WIDTH = 3
9. reg [23:0] tags [31:0];
10. reg [255:0] inst\_data [31:0]; //1KByte
11. reg valid\_cache [31:0];
12. initial begin
13. $readmemh("tags.mem", tags);
14. $readmemh("data.mem", inst\_data);
15. $readmemh("valid\_bits.mem", valid\_cache);
16. end
18. assign block = inst\_data[addr[7:3]];
19. always @(posedge clk) begin
20. if ((valid\_cache[addr[7:3]] == 1) && (tags[addr[7:3]] == addr[31:8])) begin
21. case(addr[2:0])
22. 3'b000: begin
23. inst <= block[255:224];
24. end
25. 3'b001: begin
26. inst <= block[223:192];
27. end
28. 3'b010: begin
29. inst <= block[191:160];
30. end
31. 3'b011: begin
32. inst <= block[159:128];
33. end
34. 3'b100: begin
35. inst <= block[127:96];
36. end
37. 3'b101: begin
38. inst <= block[95:64];
39. end
40. 3'b110: begin
41. inst <= block[63:32];
42. end
43. 3'b111: begin
44. inst <= block[31:0]; //31
45. end
46. endcase
47. end
48. else begin
49. inst <= 32'hF;
50. end
51. end
52. endmodule

### 6.1.3 Περιγραφή Ενδιάμεσου Καταχωρητή IF\_ID – IF\_ID.v

1. module IF\_ID (clk, hazDetect\_IF\_ID, IF\_Flush, pcPlusFour\_i, pc\_i, inst\_i, pcPlusFour\_o, pc\_o, inst\_o);
2. input clk, hazDetect\_IF\_ID, IF\_Flush;
3. input [31:0] pcPlusFour\_i, pc\_i, inst\_i;
4. output reg [31:0] pcPlusFour\_o, pc\_o, inst\_o;
5. always @(negedge clk) begin
6. if(IF\_Flush == 0) begin
7. if(hazDetect\_IF\_ID) begin
8. pcPlusFour\_o <= pcPlusFour\_i;
9. pc\_o <= pc\_i;
10. inst\_o <= inst\_i;
11. end
12. end
13. else begin
14. pcPlusFour\_o <= 0;
15. pc\_o <= 0;
16. inst\_o <= 0;
17. end
18. end
19. endmodule

### 6.1.4 Περιγραφή Καταχωρητών Γενικού Σκοπού – RegFile.v

1. module RegFile (clk, RegWrite, rr1, rr2, wr, wd, rd1, rd2);
2. input clk, RegWrite;
3. input [4:0] rr1, rr2, wr;
4. input [31:0] wd;
5. output [31:0] rd1, rd2;
6. reg [31:0] register [31:0];
7. always @(posedge clk) begin
8. if(RegWrite) begin
9. register[wr] <= wd;
10. end
11. else begin
12. register[wr] <= register[wr];
13. end
14. end
15. assign rd1 = register[rr1];
16. assign rd2 = register[rr2];
18. endmodule

### 6.1.5 Περιγραφή Μονάδας Ανίχνευσης Εξαρτήσεων – HazardDetectUnit.v

1. module HazardDetectUnit (reset\_haz, ID\_EXrd, IF\_IDrs1, IF\_IDrs2, ID\_EXmemRead, PCwrite, IF\_IDwrite, ID\_EXwrite, regWrite, memWrite);
3. input ID\_EXmemRead, reset\_haz;
4. input [4:0] ID\_EXrd, IF\_IDrs1, IF\_IDrs2;
5. output reg PCwrite, IF\_IDwrite, ID\_EXwrite, regWrite, memWrite;
7. //for load-use sequences of instructions
8. always @(reset\_haz, ID\_EXmemRead, ID\_EXrd, IF\_IDrs1, IF\_IDrs2) begin //this reset signal is only for initialization
9. if (reset\_haz) begin
10. PCwrite = 1;
11. IF\_IDwrite = 1;
12. ID\_EXwrite = 1;
13. regWrite = 1;
14. memWrite = 1;
15. end
16. else begin
17. if ((ID\_EXmemRead == 1) && ((ID\_EXrd == IF\_IDrs1) || (ID\_EXrd == IF\_IDrs2)))
18. begin
19. PCwrite = 0;
20. IF\_IDwrite = 0;
21. ID\_EXwrite = 0;
22. regWrite = 0;
23. memWrite = 0;
24. end
25. else
26. begin
27. PCwrite = 1;
28. IF\_IDwrite = 1;
29. ID\_EXwrite = 1;
30. regWrite = 1;
31. memWrite = 1;
32. end
33. end
34. end
36. endmodule

### 6.1.6 Περιγραφή Μονάδας Ελέγχου – ControlUnit.v

1. module ControlUnit (opcode, ALUop, MemRead, MemWrite, RegWrite, MemToReg, ALUsrc, Jump, LUIorAUIPC);
3. input [6:0] opcode;
4. output reg [1:0] ALUop, MemToReg, Jump;
5. output reg MemRead, MemWrite, RegWrite, ALUsrc, LUIorAUIPC;
6. always @(opcode) begin
7. case(opcode)
8. 7'b0000011 : begin //LOAD
9. LUIorAUIPC <= 1'b0; //useless
10. Jump <= 2'b00;
11. ALUop <= 2'b01;
12. ALUsrc <= 1'b1;
13. MemRead <= 1'b1;
14. MemWrite <= 1'b0;
15. RegWrite <= 1'b1;
16. MemToReg <= 2'b00;
17. end
18. 7'b0100011 : begin //STORE
19. LUIorAUIPC <= 1'b0; //useless
20. Jump <= 2'b00;
21. ALUop <= 2'b01;
22. ALUsrc <= 1'b1;
23. MemRead <= 1'b0;
24. MemWrite <= 1'b1;
25. RegWrite <= 1'b0;
26. MemToReg <= 2'b00; //useless
27. end
28. 7'b0110011 : begin //R-type
29. LUIorAUIPC <= 1'bx; //useless
30. Jump <= 2'b00;
31. ALUop <= 2'b10;
32. ALUsrc <= 1'b0;
33. MemRead <= 1'b0;
34. MemWrite <= 1'b0;
35. RegWrite <= 1'b1;
36. MemToReg <= 2'b01;
37. end
38. 7'b1100011 : begin //Branch
39. LUIorAUIPC <= 1'b0; //useless
40. Jump <= 2'b00; //useless
41. ALUop <= 2'b11;
42. ALUsrc <= 1'b0;
43. MemRead <= 1'b0;
44. MemWrite <= 1'b0;
45. RegWrite <= 1'b0;
46. MemToReg <= 2'b00; //useless
47. end
48. 7'b0010011 : begin //Immediates
49. LUIorAUIPC <= 1'b0; //useless
50. Jump <= 2'b00;
51. ALUop <= 2'b00;
52. ALUsrc <= 1'b1;
53. MemRead <= 1'b0;
54. MemWrite <= 1'b0;
55. RegWrite <= 1'b1;
56. MemToReg <= 2'b01;
57. end
58. 7'b0110111 : begin //LUI
59. LUIorAUIPC <= 1'b1;
60. Jump <= 2'b00; //useless
61. ALUop <= 2'b00; //useless
62. ALUsrc <= 2'b0; //useless
63. MemRead <= 1'b0;
64. MemWrite <= 1'b0;
65. RegWrite <= 1'b1;
66. MemToReg <= 2'b11;
67. end
68. 7'b0010111 : begin //AUIPC
69. LUIorAUIPC <= 1'b0;
70. Jump <= 2'b00; //useless
71. ALUop <= 2'b00; //useless
72. ALUsrc <= 2'b0; //useless
73. MemRead <= 1'b0;
74. MemWrite <= 1'b0;
75. RegWrite <= 1'b1;
76. MemToReg <= 2'b11;
77. end
78. 7'b1100111 : begin //JALR
79. LUIorAUIPC <= 1'b0; //useless
80. Jump <= 2'b10;
81. ALUop <= 2'b01;
82. ALUsrc <= 2'b1;
83. MemRead <= 1'b0;
84. MemWrite <= 1'b0;
85. RegWrite <= 1'b1;
86. MemToReg <= 2'b10;
87. end
88. 7'b1101111 : begin //JAL
89. LUIorAUIPC <= 1'b0; //useless
90. Jump <= 2'b01;
91. ALUop <= 2'b01; //useless
92. ALUsrc <= 2'b1; //useless
93. MemRead <= 1'b0;
94. MemWrite <= 1'b0;
95. RegWrite <= 1'b1;
96. MemToReg <= 2'b10;
97. end
98. default : begin
99. LUIorAUIPC <= 1'b0;
100. Jump <= 2'b00;
101. ALUop <= 2'b00;
102. ALUsrc <= 2'b0;
103. MemRead <= 1'b0;
104. MemWrite <= 1'b0;
105. RegWrite <= 1'b0;
106. MemToReg <= 2'b00;
107. end
108. endcase
109. end
110. endmodule

### 6.1.7 Περιγραφή Μονάδας Παραγωγής Άμεσων Δεδομένων – ImmGen.v

1. module ImmGen (inst, imm);
2. input [31:0] inst;
3. output reg [31:0] imm;
4. always @(\*) begin
5. case(inst[6:0])
6. 7'b0010011 : begin //IMMEDIATES
7. imm <= $signed(inst[31:20]); // I-type
8. end
9. 7'b0000011 : begin //LOAD
10. imm <= $signed(inst[31:20]); // I-type
11. end
12. 7'b0100011 : begin
13. imm <= $signed({inst[31:25], inst[11:7]}); // S-type
14. end
15. 7'b1100011 : begin
16. imm <= $signed({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0}); // B-type
17. end
18. 7'b0010111 : begin
19. imm <= {inst[31:12], 12'b000000000000}; // U-type for AUIPC
20. end
21. 7'b0110111 : begin
22. imm <= {inst[31:12], 12'b000000000000}; // U-type for LUI
23. end
24. 7'b1101111 : begin //JAL
25. imm <= $signed({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0}); // I-type
26. end
27. 7'b1100111 : begin //JALR
28. imm <= $signed(inst[31:20]); // J-type
29. end
30. default : begin
31. imm <= 32'b0;
32. end
33. endcase
34. end
35. endmodule

### 6.1.7 Περιγραφή Ενδιάμεσου Καταχωρητή ID\_ΕΧ – ID\_ΕΧ.v

### 

1. module ID\_EX (
2. clk,
3. hazDetect\_ID\_EX,
4. ID\_Flush,
5. id\_ex\_LUIorAUIPC\_i,
6. id\_ex\_Jump\_i,
7. id\_ex\_RegWrite\_i ,
8. id\_ex\_MemToReg\_i ,
9. id\_ex\_MemRead\_i ,
10. id\_ex\_MemWrite\_i ,
11. id\_ex\_ALUop\_i ,
12. id\_ex\_ALUsrc\_i ,
13. branchAddr\_i ,
14. id\_ex\_pc\_i,
15. id\_ex\_pcPlusFour\_i ,
16. rd1\_i ,
17. rd2\_i ,
18. imm\_i ,
19. ALUctrl\_funct7\_i ,
20. ALUctrl\_funct3\_i ,
21. wr\_i ,
22. rs1\_i ,
23. rs2\_i ,
24. id\_ex\_LUIorAUIPC\_o,
25. id\_ex\_Jump\_o,
26. id\_ex\_RegWrite\_o ,
27. id\_ex\_MemToReg\_o ,
28. id\_ex\_MemRead\_o ,
29. id\_ex\_MemWrite\_o ,
30. id\_ex\_ALUop\_o ,
31. id\_ex\_ALUsrc\_o ,
32. branchAddr\_o ,
33. id\_ex\_pc\_o,
34. id\_ex\_pcPlusFour\_o ,
35. rd1\_o ,
36. rd2\_o ,
37. imm\_o ,
38. ALUctrl\_funct7\_o ,
39. ALUctrl\_funct3\_o ,
40. wr\_o ,
41. rs1\_o ,
42. rs2\_o
43. );
44. input clk, hazDetect\_ID\_EX, ID\_Flush, id\_ex\_RegWrite\_i, id\_ex\_MemRead\_i, id\_ex\_MemWrite\_i, id\_ex\_ALUsrc\_i, id\_ex\_LUIorAUIPC\_i;
45. input [1:0] id\_ex\_ALUop\_i, id\_ex\_MemToReg\_i, id\_ex\_Jump\_i;
46. input [31:0] id\_ex\_pcPlusFour\_i, branchAddr\_i, rd1\_i, rd2\_i, imm\_i, id\_ex\_pc\_i;
47. input [6:0] ALUctrl\_funct7\_i;
48. input [2:0] ALUctrl\_funct3\_i;
49. input [4:0] wr\_i, rs1\_i, rs2\_i;
50. output reg id\_ex\_RegWrite\_o, id\_ex\_MemRead\_o, id\_ex\_MemWrite\_o, id\_ex\_ALUsrc\_o, id\_ex\_LUIorAUIPC\_o;
51. output reg [1:0] id\_ex\_ALUop\_o, id\_ex\_MemToReg\_o, id\_ex\_Jump\_o;
52. output reg [31:0] id\_ex\_pcPlusFour\_o, branchAddr\_o, rd1\_o, rd2\_o, imm\_o, id\_ex\_pc\_o;
53. output reg [6:0] ALUctrl\_funct7\_o;
54. output reg [2:0] ALUctrl\_funct3\_o;
55. output reg [4:0] wr\_o, rs1\_o, rs2\_o;
56. always @(negedge clk) begin
57. if(ID\_Flush == 0) begin
58. if(hazDetect\_ID\_EX) begin
59. id\_ex\_LUIorAUIPC\_o <= id\_ex\_LUIorAUIPC\_i;
60. id\_ex\_Jump\_o <= id\_ex\_Jump\_i;
61. id\_ex\_RegWrite\_o <= id\_ex\_RegWrite\_i;
62. id\_ex\_MemToReg\_o <= id\_ex\_MemToReg\_i;
63. id\_ex\_MemRead\_o <= id\_ex\_MemRead\_i;
64. id\_ex\_MemWrite\_o <= id\_ex\_MemWrite\_i;
65. id\_ex\_ALUop\_o <= id\_ex\_ALUop\_i;
66. id\_ex\_ALUsrc\_o <= id\_ex\_ALUsrc\_i;
67. branchAddr\_o <= branchAddr\_i;
68. id\_ex\_pc\_o <= id\_ex\_pc\_i;
69. id\_ex\_pcPlusFour\_o <= id\_ex\_pcPlusFour\_i;
70. rd1\_o <= rd1\_i;
71. rd2\_o <= rd2\_i;
72. imm\_o <= imm\_i;
73. ALUctrl\_funct7\_o <= ALUctrl\_funct7\_i;
74. ALUctrl\_funct3\_o <= ALUctrl\_funct3\_i;
75. wr\_o <= wr\_i;
76. rs1\_o <= rs1\_i;
77. rs2\_o <= rs2\_i;
78. end
79. else begin
80. id\_ex\_LUIorAUIPC\_o <= 0;
81. id\_ex\_Jump\_o <= 0;
82. id\_ex\_RegWrite\_o <= 0;
83. id\_ex\_MemToReg\_o <= 0;
84. id\_ex\_MemRead\_o <= 0;
85. id\_ex\_MemWrite\_o <= 0;
86. id\_ex\_ALUop\_o <= 2'b01;
87. id\_ex\_ALUsrc\_o <= 0;
88. end
89. end
90. else begin
91. id\_ex\_LUIorAUIPC\_o <= 0;
92. id\_ex\_Jump\_o <= 0;
93. id\_ex\_RegWrite\_o <= 0;
94. id\_ex\_MemToReg\_o <= 0;
95. id\_ex\_MemRead\_o <= 0;
96. id\_ex\_MemWrite\_o <= 0;
97. id\_ex\_ALUop\_o <= 2'b01; //trick because i want branch and flush signals(ID\_flush....) to go to 0 in the next cycle if they are 1.
98. id\_ex\_ALUsrc\_o <= 0;
99. branchAddr\_o <= 0;
100. id\_ex\_pc\_o <= 0;
101. id\_ex\_pcPlusFour\_o <= 0;
102. rd1\_o <= 0;
103. rd2\_o <= 0;
104. imm\_o <= 0;
105. ALUctrl\_funct7\_o <= 0;
106. ALUctrl\_funct3\_o <= 0;
107. wr\_o <= 0;
108. rs1\_o <= 0;
109. rs2\_o <= 0;
110. end
111. end
112. endmodule

### 6.1.8 Περιγραφή Μονάδας Παροχέτευσης – ForwadingUnit.v

1. module ForwardingUnit (reset\_fw, ID\_EXrs1, ID\_EXrs2, EX\_MEMrd, EX\_MEMregWrite, MEM\_WBrd, MEM\_WBregWrite, FW0, FW1);
2. input EX\_MEMregWrite, MEM\_WBregWrite, reset\_fw;
3. input [4:0] ID\_EXrs1, ID\_EXrs2, EX\_MEMrd, MEM\_WBrd;
4. output reg [1:0] FW0, FW1;
5. always @(reset\_fw, ID\_EXrs1, EX\_MEMrd, EX\_MEMregWrite, MEM\_WBregWrite, MEM\_WBrd) begin
6. if ( reset\_fw == 1 ) begin
7. FW0 = 2'b00;
8. end
9. else begin
10. if ( (EX\_MEMregWrite == 1) && (EX\_MEMrd != 0) && (EX\_MEMrd == ID\_EXrs1) ) //EX\_MEMrd != 0 because want to avoid forwarding its possibly nonzero result value to x0. x0 must always be equal to 0 in RISC5.
11. begin
12. FW0 = 2'b10;
13. end
14. else if ( MEM\_WBregWrite == 1 && (MEM\_WBrd != 0)
15. && !(EX\_MEMregWrite == 1 && (EX\_MEMrd != 0) && (EX\_MEMrd == ID\_EXrs1))
16. && (MEM\_WBrd == ID\_EXrs1) )
17. begin
18. FW0 = 2'b01;
19. end
20. else
21. begin
22. FW0 = 2'b00;
23. end
24. end
25. end
26. always @(reset\_fw, ID\_EXrs2, EX\_MEMrd, EX\_MEMregWrite, MEM\_WBregWrite, MEM\_WBrd) begin
27. if ( reset\_fw == 1 ) begin
28. FW1 = 2'b00;
29. end
30. else begin
31. if ( (EX\_MEMregWrite == 1) && (EX\_MEMrd != 0) && (EX\_MEMrd == ID\_EXrs2) )
32. begin
33. FW1 = 2'b10;
34. end
35. else if ( MEM\_WBregWrite == 1 && (MEM\_WBrd != 0)
36. && !(EX\_MEMregWrite && (EX\_MEMrd != 0) && (EX\_MEMrd == ID\_EXrs2))
37. && (MEM\_WBrd == ID\_EXrs2) )
38. begin
39. FW1 = 2'b01;
40. end
41. else
42. begin
43. FW1 = 2'b00;
44. end
45. end
46. end
48. endmodule

### 6.1.9 Περιγραφή Μονάδας Ελέγχου ΑΛΜ – ALU\_control.v

1. module ALU\_control (ALUctrl\_f7, ALUctrl\_f3, ALUop, ALUctrl\_lines);
2. input [6:0] ALUctrl\_f7; //funct7[9:3] + funct3[2:0]
3. input [2:0] ALUctrl\_f3;
4. input [1:0] ALUop;
5. output reg [3:0] ALUctrl\_lines;
6. always @(\*) begin
7. case(ALUop)
8. 2'b00 : begin
9. case({ALUctrl\_f3, ALUctrl\_f7[5]})
10. 4'b0000 : begin //ADDI
11. ALUctrl\_lines <= 4'b0000;
12. end
13. 4'b0001 : begin //ADDI
14. ALUctrl\_lines <= 4'b0000;
15. end
16. 4'b0010 : begin //SLLI
17. ALUctrl\_lines <= 4'b0010;
18. end
19. 4'b1000 : begin //XORI
20. ALUctrl\_lines <= 4'b0011;
21. end
22. 4'b1001 : begin //XORI
23. ALUctrl\_lines <= 4'b0011;
24. end
25. 4'b1010 : begin //SRLI
26. ALUctrl\_lines <= 4'b0100;
27. end
28. 4'b1011 : begin //SRAI
29. ALUctrl\_lines <= 4'b0101;
30. end
31. 4'b1100 : begin //ORI
32. ALUctrl\_lines <= 4'b0110;
33. end
34. 4'b1101 : begin //ORI
35. ALUctrl\_lines <= 4'b0110;
36. end
37. 4'b1110 : begin //ANDI
38. ALUctrl\_lines <= 4'b0111;
39. end
40. 4'b1111 : begin //ANDI
41. ALUctrl\_lines <= 4'b0111;
42. end
43. 4'b0100 : begin //SLTI
44. ALUctrl\_lines <= 4'b1110;
45. end
46. 4'b0101 : begin //SLTI
47. ALUctrl\_lines <= 4'b1110;
48. end
49. 4'b0110 : begin //SLTIU
50. ALUctrl\_lines <= 4'b1111;
51. end
52. 4'b0111 : begin //SLTIU
53. ALUctrl\_lines <= 4'b1111;
54. end
55. default : begin
56. ALUctrl\_lines <= 4'b0000;
57. end
58. endcase
59. end
60. 2'b01 : begin
61. ALUctrl\_lines <= 4'b0000; //ADD
62. end
63. 2'b10 : begin
64. //ALUctrl\_lines = (ALUctrl\_f3[2]) ? 4'b0001 : 4'b0010;
65. case({ALUctrl\_f3, ALUctrl\_f7[5]})
66. 4'b0000 : begin //ADD
67. ALUctrl\_lines <= 4'b0000;
68. end
69. 4'b0001 : begin //SUB
70. ALUctrl\_lines <= 4'b0001;
71. end
72. 4'b0010 : begin //SLL
73. ALUctrl\_lines <= 4'b0010;
74. end
75. 4'b1000 : begin //XOR
76. ALUctrl\_lines <= 4'b0011;
77. end
78. 4'b1010 : begin //SRL
79. ALUctrl\_lines <= 4'b0100;
80. end
81. 4'b1011 : begin //SRA
82. ALUctrl\_lines <= 4'b0101;
83. end
84. 4'b1100 : begin //OR
85. ALUctrl\_lines <= 4'b0110;
86. end
87. 4'b1110 : begin //AND
88. ALUctrl\_lines <= 4'b0111;
89. end
90. 4'b0100 : begin //SLT
91. ALUctrl\_lines <= 4'b1110;
92. end
93. 4'b0110 : begin //SLTU
94. ALUctrl\_lines <= 4'b1111;
95. end
96. default : begin
97. ALUctrl\_lines <= 4'b0000;
98. end
99. endcase
100. end
101. 2'b11 : begin //write a case with func3 input for the branch conditions
102. case(ALUctrl\_f3)
103. 3'b000 : begin //equal, BEQ
104. ALUctrl\_lines <= 4'b1100;
105. end
106. 3'b001 : begin //not equal, BNE
107. ALUctrl\_lines <= 4'b1101;
108. end
109. 3'b100 : begin // BLT
110. ALUctrl\_lines <= 4'b1000;
111. end
112. 3'b101 : begin // BGE
113. ALUctrl\_lines <= 4'b1001;
114. end
115. 3'b110 : begin // BLTU
116. ALUctrl\_lines <= 4'b1010;
117. end
118. 3'b111 : begin // BGEU
119. ALUctrl\_lines <= 4'b1011;
120. end
121. default : begin
122. ALUctrl\_lines <= 4'b0000;
123. end
124. endcase
125. end
126. endcase
127. end
128. endmodule

### 6.1.10 Περιγραφή Αριθμητικής και Λογικής Μονάδας – ALU.v

1. module ALU (reset\_alu, data0, data1, ctrl, result, branch);
2. input reset\_alu;
3. input [31:0] data0, data1;
4. input [3:0] ctrl;
5. output reg [31:0] result;
6. output reg branch;
7. always@(\*) begin
8. if (reset\_alu) begin
9. branch <= 0;
10. end
11. case(ctrl)
12. 4'b0000 : begin //ADD
13. result <= data0 + data1;
14. branch <= 0;
15. end
16. 4'b0001 : begin //SUB
17. result <= data0 - data1;
18. branch <= 0;
19. end
20. 4'b0010 : begin //SLL
21. result <= data0 << data1[4:0];
22. branch <= 0;
23. end
24. 4'b0011 : begin //XOR
25. result <= data0 ^ data1;
26. branch <= 0;
27. end
28. 4'b0100 : begin //SRL
29. result <= data0 >> data1[4:0];
30. branch <= 0;
31. end
32. 4'b0101 : begin //SRA
33. result <= data0 >>> data1[4:0];
34. branch <= 0;
35. end
36. 4'b0110 : begin //OR
37. result <= data0 | data1;
38. branch <= 0;
39. end
40. 4'b0111 : begin //AND
41. result <= data0 & data1;
42. branch <= 0;
43. end
44. 4'b1100 : begin //BEQ // for branch-> 1: branch adress, 0: addr + 1.
45. branch <= (data0 == data1) ? 1 : 0;
46. result <= 0;
47. end
48. 4'b1101 : begin //BNE
49. branch <= (data0 != data1) ? 1 : 0;
50. result <= 0;
51. end
52. 4'b1000 : begin //BLT
53. branch <= ($signed(data0) < $signed(data1)) ? 1 : 0;
54. result <= 0;
55. end
56. 4'b1001 : begin //BGE
57. branch <= ($signed(data0) >= $signed(data1)) ? 1 : 0;
58. result <= 0;
59. end
60. 4'b1010 : begin //BLTU
61. branch <= (data0 < data1) ? 1 : 0;
62. result <= 0;
63. end
64. 4'b1011 : begin //BGEU
65. branch <= (data0 >= data1) ? 1 : 0;
66. result <= 0;
67. end
68. 4'b1110 : begin //SLT
69. result <= ($signed(data0) < $signed(data1)) ? 1 : 0;
70. branch <= 0;
71. end
72. 4'b1111 : begin //SLTU
73. result <= (data0 < data1) ? 1 : 0;
74. branch <= 0;
75. end
76. endcase
77. end
78. endmodule

### 6.1.11 Περιγραφή Μονάδας Διαχείρισης Διακλαδώσεων – BranchUnit.v

1. module BranchUnit (clk, reset\_br, jump, branch, mux\_to\_pc, IF\_Flush, ID\_Flush);
2. input clk, branch, reset\_br;
3. input [1:0] jump;
4. output reg IF\_Flush, ID\_Flush;
5. output reg [1:0] mux\_to\_pc;
6. //handling the occurence of load-use sequence after a conditional branch.
7. always @(posedge clk) begin //negedge clk reset\_br, branch, jump
8. if (reset\_br) begin
9. mux\_to\_pc <= 2'b00;
10. IF\_Flush <= 0;
11. ID\_Flush <= 0;
12. end
13. else begin
14. if (branch) begin
15. mux\_to\_pc <= 2'b01;
16. IF\_Flush <= 1;
17. ID\_Flush <= 1;
18. end
19. else begin
20. case(jump)
21. 2'b00 : begin
22. mux\_to\_pc <= 2'b00;
23. IF\_Flush <= 0;
24. ID\_Flush <= 0;
25. end
26. 2'b01 : begin
27. mux\_to\_pc <= 2'b01;
28. IF\_Flush <= 1;
29. ID\_Flush <= 1;
30. end
31. 2'b10 : begin
32. mux\_to\_pc <= 2'b10;
33. IF\_Flush <= 1;
34. ID\_Flush <= 1;
35. end
36. 2'b11 : begin
37. mux\_to\_pc <= 2'b00;
38. IF\_Flush <= 0;
39. ID\_Flush <= 0;
40. end
41. endcase
42. end
43. end
44. end
45. endmodule

### 6.1.12 Περιγραφή Ενδιάμεσου Καταχωρητή ΕΧ\_MEM – ΕΧ\_MEM.v

1. module EX\_MEM (
2. clk,
3. ex\_mem\_RegWrite\_i ,
4. ex\_mem\_MemToReg\_i ,
5. ex\_mem\_MemRead\_i ,
6. ex\_mem\_MemWrite\_i ,
7. ex\_mem\_pcPlusFour\_i,
8. ex\_mem\_Utype\_res\_i,
9. ALUresult\_i,
10. rd2\_i ,
11. wr\_i ,
12. funct3\_i,
13. ex\_mem\_RegWrite\_o ,
14. ex\_mem\_MemToReg\_o ,
15. ex\_mem\_MemRead\_o ,
16. ex\_mem\_MemWrite\_o ,
17. ex\_mem\_pcPlusFour\_o,
18. ex\_mem\_Utype\_res\_o,
19. ALUresult\_o,
20. rd2\_o ,
21. wr\_o,
22. funct3\_o
23. );
24. input clk, ex\_mem\_RegWrite\_i, ex\_mem\_MemWrite\_i, ex\_mem\_MemRead\_i;
25. input [1:0] ex\_mem\_MemToReg\_i;
26. input [31:0] ex\_mem\_pcPlusFour\_i, ALUresult\_i, rd2\_i, ex\_mem\_Utype\_res\_i;
27. input [4:0] wr\_i;
28. input [2:0] funct3\_i;
29. output reg ex\_mem\_RegWrite\_o, ex\_mem\_MemWrite\_o, ex\_mem\_MemRead\_o;
30. output reg [1:0] ex\_mem\_MemToReg\_o;
31. output reg [31:0] ex\_mem\_pcPlusFour\_o, ALUresult\_o, rd2\_o, ex\_mem\_Utype\_res\_o;
32. output reg [4:0] wr\_o;
33. output reg [2:0] funct3\_o;
34. always @(negedge clk) begin
35. ex\_mem\_RegWrite\_o <= ex\_mem\_RegWrite\_i;
36. ex\_mem\_MemToReg\_o <= ex\_mem\_MemToReg\_i;
37. ex\_mem\_MemRead\_o <= ex\_mem\_MemRead\_i;
38. ex\_mem\_MemWrite\_o <= ex\_mem\_MemWrite\_i;
39. ex\_mem\_pcPlusFour\_o <= ex\_mem\_pcPlusFour\_i;
40. ex\_mem\_Utype\_res\_o <= ex\_mem\_Utype\_res\_i;
41. ALUresult\_o <= ALUresult\_i;
42. rd2\_o <= rd2\_i;
43. wr\_o <= wr\_i;
44. funct3\_o <= funct3\_i;
45. end
46. endmodule

### 6.1.13 Περιγραφή Κρυφής Μνήμης Δεδομένων – DataCache.v

1. module DataCache (clk, addr, MemWrite, MemRead, WriteData, funct3, output\_data);
2. input clk, MemRead, MemWrite;
3. input [31:0] addr, WriteData;
4. input [2:0] funct3;
5. output reg [31:0] output\_data;
7. reg [31:0] ReadData;
9. wire [255:0] block;
10. wire [31:0] DataOut;
11. //cache modules, 1KByte TAG\_WIDTH = 24, INDEX\_WIDTH = 5, OFFSET\_WIDTH = 3
12. reg [23:0] tags [31:0];
13. reg [255:0] data\_cache [31:0]; //1KByte
14. reg valid\_cache [31:0];
15. reg dirty\_cache [31:0];
16. initial begin
17. $readmemh("tags.mem", tags);
18. $readmemh("cacheData.mem", data\_cache);
19. $readmemh("valid\_bits.mem", valid\_cache);
20. $readmemh("dirty\_bits.mem", dirty\_cache);
21. end
23. assign block = data\_cache[addr[7:3]];
25. always @(posedge clk) begin
26. if ((MemRead == 1) && (valid\_cache[addr[7:3]] == 1) && (tags[addr[7:3]] == addr[31:8])) begin
27. case(addr[2:0])
28. 3'b000: begin
29. ReadData <= block[255:224];
30. end
31. 3'b001: begin
32. ReadData <= block[223:192];
33. end
34. 3'b010: begin
35. ReadData <= block[191:160];
36. end
37. 3'b011: begin
38. ReadData <= block[159:128];
39. end
40. 3'b100: begin
41. ReadData <= block[127:96];
42. end
43. 3'b101: begin
44. ReadData <= block[95:64];
45. end
46. 3'b110: begin
47. ReadData <= block[63:32];
48. end
49. 3'b111: begin
50. ReadData <= block[31:0];
51. end
52. endcase
53. end
54. else if (MemWrite == 1) begin
55. case(addr[2:0])
56. 3'b000: begin
57. case(funct3[1:0])
58. 2'b00 : begin //store byte
59. data\_cache[addr[7:3]][231:224] <= WriteData[7:0];
60. end
61. 2'b01 : begin //store half
62. data\_cache[addr[7:3]][239:224] <= WriteData[15:0];
63. end
64. 2'b10 : begin //store word
65. data\_cache[addr[7:3]][255:224] <= WriteData[31:0];
66. end
67. 2'b11 : begin
69. end
70. endcase
71. end
72. 3'b001: begin
73. case(funct3[1:0])
74. 2'b00 : begin //store byte
75. data\_cache[addr[7:3]][199:192] <= WriteData[7:0];
76. end
77. 2'b01 : begin //store half
78. data\_cache[addr[7:3]][207:192] <= WriteData[15:0];
79. end
80. 2'b10 : begin //store word
81. data\_cache[addr[7:3]][223:192] <= WriteData[31:0];
82. end
83. 2'b11 : begin
85. end
86. endcase
87. end
88. 3'b010: begin
89. case(funct3[1:0])
90. 2'b00 : begin //store byte
91. data\_cache[addr[7:3]][167:160] <= WriteData[7:0];
92. end
93. 2'b01 : begin //store half
94. data\_cache[addr[7:3]][175:160] <= WriteData[15:0];
95. end
96. 2'b10 : begin //store word
97. data\_cache[addr[7:3]][191:160] <= WriteData[31:0];
98. end
99. 2'b11 : begin
101. end
102. endcase
103. end
104. 3'b011: begin
105. case(funct3[1:0])
106. 2'b00 : begin //store byte
107. data\_cache[addr[7:3]][135:128] <= WriteData[7:0];
108. end
109. 2'b01 : begin //store half
110. data\_cache[addr[7:3]][143:128] <= WriteData[15:0];
111. end
112. 2'b10 : begin //store word
113. data\_cache[addr[7:3]][159:128] <= WriteData[31:0];
114. end
115. 2'b11 : begin
117. end
118. endcase
119. end
120. 3'b100: begin
121. case(funct3[1:0])
122. 2'b00 : begin //store byte
123. data\_cache[addr[7:3]][103:96] <= WriteData[7:0];
124. end
125. 2'b01 : begin //store half
126. data\_cache[addr[7:3]][111:96] <= WriteData[15:0];
127. end
128. 2'b10 : begin //store word
129. data\_cache[addr[7:3]][127:96] <= WriteData[31:0];
130. end
131. 2'b11 : begin
133. end
134. endcase
135. end
136. 3'b101: begin
137. case(funct3[1:0])
138. 2'b00 : begin //store byte
139. data\_cache[addr[7:3]][71:64] <= WriteData[7:0];
140. end
141. 2'b01 : begin //store half
142. data\_cache[addr[7:3]][79:64] <= WriteData[15:0];
143. end
144. 2'b10 : begin //store word
145. data\_cache[addr[7:3]][95:64] <= WriteData[31:0];
146. end
147. 2'b11 : begin
149. end
150. endcase
151. end
152. 3'b110: begin
153. case(funct3[1:0])
154. 2'b00 : begin //store byte
155. data\_cache[addr[7:3]][39:32] <= WriteData[7:0];
156. end
157. 2'b01 : begin //store half
158. data\_cache[addr[7:3]][47:32] <= WriteData[15:0];
159. end
160. 2'b10 : begin //store word
161. data\_cache[addr[7:3]][63:32] <= WriteData[31:0];
162. end
163. 2'b11 : begin
165. end
166. endcase
167. end
168. 3'b111: begin
169. case(funct3[1:0])
170. 2'b00 : begin //store byte
171. data\_cache[addr[7:3]][7:0] <= WriteData[7:0];
172. end
173. 2'b01 : begin //store half
174. data\_cache[addr[7:3]][15:0] <= WriteData[15:0];
175. end
176. 2'b10 : begin //store word
177. data\_cache[addr[7:3]][31:0] <= WriteData[31:0];
178. end
179. 2'b11 : begin
181. end
182. endcase
183. end
184. endcase
185. end
186. else begin
187. ReadData <= 32'hFAFAFAFA;
188. end
190. end
191. assign DataOut = ReadData;
192. always @(DataOut, funct3) begin
193. case(funct3)
194. 3'b000: begin //load byte
195. output\_data <= $signed(DataOut[7:0]);
196. end
197. 3'b001: begin //load half
198. output\_data <= $signed(DataOut[15:0]);
199. end
200. 3'b010: begin //load word
201. output\_data <= DataOut[31:0];
202. end
203. 3'b011: begin //useless
204. output\_data <= 0;
205. end
206. 3'b100: begin //load byte unsigned
207. output\_data <= DataOut[7:0];
208. end
209. 3'b101: begin //load half unsigned
210. output\_data <= DataOut[15:0];
211. end
212. 3'b110: begin //useless
213. output\_data <= 0;
214. end
215. 3'b111: begin //useless
216. output\_data <= 0;
217. end
218. endcase
219. end
220. endmodule

### 6.1.13 Περιγραφή Ενδιάμεσου Καταχωρητή MEM\_WB – MEM\_WB.v

1. module MEM\_WB (
2. clk ,
3. readMem\_i ,
4. ALUresult\_i ,
5. wr\_i ,
6. mem\_wb\_RegWrite\_i ,
7. mem\_wb\_MemToReg\_i ,
8. mem\_wb\_pcPlusFour\_i,
9. mem\_wb\_Utype\_res\_i,
10. readMem\_o ,
11. ALUresult\_o ,
12. wr\_o ,
13. mem\_wb\_RegWrite\_o ,
14. mem\_wb\_MemToReg\_o ,
15. mem\_wb\_pcPlusFour\_o,
16. mem\_wb\_Utype\_res\_o
17. );
18. input clk;
19. input mem\_wb\_RegWrite\_i;
20. input [1:0] mem\_wb\_MemToReg\_i;
21. input [31:0] mem\_wb\_pcPlusFour\_i, readMem\_i, ALUresult\_i, mem\_wb\_Utype\_res\_i;
22. input [4:0] wr\_i;
23. output reg mem\_wb\_RegWrite\_o;
24. output reg [1:0] mem\_wb\_MemToReg\_o;
25. output reg [31:0] mem\_wb\_pcPlusFour\_o, readMem\_o, ALUresult\_o, mem\_wb\_Utype\_res\_o;
26. output reg [4:0] wr\_o;
27. always @(negedge clk) begin
28. readMem\_o <= readMem\_i;
29. ALUresult\_o <= ALUresult\_i;
30. wr\_o <= wr\_i;
31. mem\_wb\_RegWrite\_o <= mem\_wb\_RegWrite\_i;
32. mem\_wb\_MemToReg\_o <= mem\_wb\_MemToReg\_i;
33. mem\_wb\_pcPlusFour\_o <= mem\_wb\_pcPlusFour\_i;
34. mem\_wb\_Utype\_res\_o <= mem\_wb\_Utype\_res\_i;
35. end
36. endmodule

### 6.1.14 Περιγραφή Μηχανισμού Μερικώς Επικαλυπτόμενων Λειτουργιών – PipelineDatapath.v

1. module PipelineDatapath (clk, reset); // add outputs
2. //------------------------------I/O ports---------------------------------
3. input clk, reset;
4. /\* output [31:0] mux\_to\_pc, instMemOut, rd1\_id\_ex, ALU\_0, ALU\_2nd\_in, result, DataMem\_out;
5. output [1:0] fw0, fw1;
6. output [2:0] f3\_to\_dataMem;
7. output hazDetect\_IF\_ID, RegWrite\_to\_ex\_mem; \*/
8. //output [31:0] result, DataMem\_out, writeData\_to\_regFile, rd1\_id\_ex;
9. //------------------------Wires for datapath-----------------------------
10. //---IF
11. wire [31:0] add4, branchAddr, mux\_to\_pc, pc\_out, instMemOut, inst;
12. wire [255:0] block\_out;
14. //---ID
15. wire [31:0] if\_id\_pc\_o, if\_id\_inst\_o, if\_id\_pcPlusFour, rd1\_id\_ex, rd2\_id\_ex, Immed;
16. wire [1:0] ImmGenCtrl, ALUop, MemToReg, Jump;
17. wire Branch, MemRead, MemWrite, RegWrite, ALUsrc, LUIorAUIPC;
18. wire hazDetect\_PC, hazDetect\_IF\_ID, hazDetect\_ID\_EX, reg\_write, mem\_write;
19. wire regWrite\_to\_ID\_EX, memWrite\_to\_ID\_EX;
21. //---EX
22. wire zeroFlag, ALUsrc\_out, to\_branchUnit, IFreg\_flush, IDreg\_flush;
23. wire [31:0] rd1\_MUX, rd2\_MUX, imm\_MUX, branch\_address, ALU\_0, ALU\_1, result, adder\_res;
24. wire [31:0] ALU\_2nd\_in, id\_ex\_pcPlusFour, out\_AND, id\_ex\_pc\_out, pcPlusImm\_U\_type, Utype\_res;
25. wire [63:0] mul\_res;
26. wire [2:0] funct3\_to\_out;
27. wire [6:0] funct7\_to\_out;
28. wire [4:0] wr\_to\_EX\_MEM, rs1\_FW\_in, rs2\_FW\_in;
29. wire [1:0] ALUop\_out, MemToReg\_to\_ex\_mem, sel\_mux\_to\_pc, jump\_to\_branchUnit;
30. wire RegWrite\_to\_ex\_mem, Branch\_to\_ex\_mem, MemRead\_to\_ex\_mem, MemWrite\_to\_ex\_mem, LUIorAUIPC\_to\_mux;
31. wire [3:0] ALUctrl\_lines;
32. wire [1:0] fw0, fw1;
33. //---MEM
34. wire zero\_AND;
35. wire [1:0] MemToReg\_to\_mem\_wb;
36. wire RegWrite\_to\_mem\_wb, Branch\_out, MemRead\_out, MemWrite\_out;
37. wire [31:0] res\_to\_DataMem\_Addr, rd2\_to\_DataMem\_wd, DataMem\_out, Data\_from\_Mem, ALUres\_toMUX;
38. wire [31:0] ex\_mem\_pcPlusFour, ex\_mem\_Utype\_res;
39. wire [4:0] wr\_to\_MEM\_WB, wr\_to\_regFile;
40. wire [2:0] f3\_to\_dataMem;
41. //---WB
42. wire [31:0] writeData\_to\_regFile, mem\_wb\_pcPlusFour, mem\_wb\_Utype\_res;
43. wire [1:0] MemToReg\_out;
44. wire RegWrite\_out;
45. //----------------------------------------------MODULES---------------------------------------------------

48. //-------------------------------------------------IF-----------------------------------------------------
49. PC pc (
50. .clk (clk),
51. .reset (reset),
52. .pc\_input (mux\_to\_pc), //add4
53. .hazDetect\_PC (hazDetect\_PC),
54. .pc\_output (pc\_out)
55. );
56. Adder32 PCadd4 (
57. .data1 (pc\_out),
58. .data2 (32'd1), //4 when InstructionMem
59. .data\_o (add4)
60. );
61. // MUX32\_2to1 pc\_input\_select (
62. // .select\_i (sel\_mux\_to\_pc), //1'b0
63. // .data0\_i (add4),
64. // .data1\_i (branch\_address), //adder\_res
65. // .data\_o (mux\_to\_pc)
66. // );
67. MUX32\_4to1 pc\_input\_select (
68. .select\_i (sel\_mux\_to\_pc),
69. .data0\_i (add4),
70. .data1\_i (branch\_address),
71. .data2\_i (out\_AND),
72. .data3\_i (),
73. .data\_o (mux\_to\_pc)
74. );
75. InstCache instruction\_cache (
76. .clk (clk),
77. .addr (pc\_out),
78. .reset (reset),
79. .inst (instMemOut)
80. );
81. //--------------------------------------------------ID--------------------------------------------------
82. IF\_ID if\_id (
83. .clk (clk),
84. .hazDetect\_IF\_ID (hazDetect\_IF\_ID),
85. .IF\_Flush (IFreg\_flush),
86. .pcPlusFour\_i (add4),
87. .pc\_i (pc\_out),
88. .inst\_i (instMemOut),
89. .pcPlusFour\_o (if\_id\_pcPlusFour), //wire [31:0] if\_id\_pcPlusFour;
90. .pc\_o (if\_id\_pc\_o),
91. .inst\_o (if\_id\_inst\_o)
92. );
94. RegFile registers (
95. .clk (clk),
96. .RegWrite (RegWrite\_out),
97. .rr1 (if\_id\_inst\_o[19:15]),
98. .rr2 (if\_id\_inst\_o[24:20]),
99. .wr (wr\_to\_regFile),
100. .wd (writeData\_to\_regFile),
101. .rd1 (rd1\_id\_ex),
102. .rd2 (rd2\_id\_ex)
103. );
104. HazardDetectUnit hazDetection (
105. .reset\_haz (reset),
106. .ID\_EXrd (wr\_to\_EX\_MEM),
107. .IF\_IDrs1 (if\_id\_inst\_o[19:15]),
108. .IF\_IDrs2 (if\_id\_inst\_o[24:20]),
109. .ID\_EXmemRead (MemRead\_to\_ex\_mem),
110. .PCwrite (hazDetect\_PC), //wire hazDetect\_PC, hazDetect\_IF\_ID, reg\_write, mem\_write;
111. .IF\_IDwrite (hazDetect\_IF\_ID),
112. .ID\_EXwrite (hazDetect\_ID\_EX),
113. .regWrite (reg\_write),
114. .memWrite (mem\_write)
115. );
116. ControlUnit control (
117. .opcode (if\_id\_inst\_o[6:0]),
118. .ALUop (ALUop),
119. .MemRead (MemRead),
120. .MemWrite (MemWrite),
121. .RegWrite (RegWrite),
122. .MemToReg (MemToReg),
123. .ALUsrc (ALUsrc),
124. .Jump (Jump),
125. .LUIorAUIPC (LUIorAUIPC)
126. );
127. MUX\_2to1 regWrite (
128. .select\_i (reg\_write),
129. .data0\_i (1'b0),
130. .data1\_i (RegWrite),
131. .data\_o (regWrite\_to\_ID\_EX)
132. );
133. MUX\_2to1 memWrite (
134. .select\_i (mem\_write),
135. .data0\_i (1'b0),
136. .data1\_i (MemWrite),
137. .data\_o (memWrite\_to\_ID\_EX)
138. );
139. ImmGen immediates (
140. .inst (if\_id\_inst\_o),
141. .imm (Immed)
142. );
143. Adder32 branchAddress (
144. .data1 (Immed),
145. .data2 (if\_id\_pc\_o),
146. .data\_o (adder\_res)
147. );
148. //--------------------------------------------------------EX-------------------------------------------------------
149. ID\_EX id\_ex (
150. .clk (clk),
151. .hazDetect\_ID\_EX (hazDetect\_ID\_EX),
152. .ID\_Flush (IDreg\_flush),
153. .id\_ex\_LUIorAUIPC\_i (LUIorAUIPC),
154. .id\_ex\_Jump\_i (Jump),
155. .id\_ex\_RegWrite\_i (regWrite\_to\_ID\_EX),
156. .id\_ex\_MemToReg\_i (MemToReg),
157. .id\_ex\_MemRead\_i (MemRead),
158. .id\_ex\_MemWrite\_i (memWrite\_to\_ID\_EX),
159. .id\_ex\_ALUop\_i (ALUop),
160. .id\_ex\_ALUsrc\_i (ALUsrc),
161. .branchAddr\_i (adder\_res),
162. .id\_ex\_pc\_i (if\_id\_pc\_o),
163. .id\_ex\_pcPlusFour\_i (if\_id\_pcPlusFour),
164. .rd1\_i (rd1\_id\_ex),
165. .rd2\_i (rd2\_id\_ex),
166. .imm\_i (Immed),
167. .ALUctrl\_funct7\_i (if\_id\_inst\_o[31:25]),
168. .ALUctrl\_funct3\_i (if\_id\_inst\_o[14:12]),
169. .wr\_i (if\_id\_inst\_o[11:7]),
170. .rs1\_i (if\_id\_inst\_o[19:15]),
171. .rs2\_i (if\_id\_inst\_o[24:20]),
172. .id\_ex\_LUIorAUIPC\_o (LUIorAUIPC\_to\_mux),
173. .id\_ex\_Jump\_o (jump\_to\_branchUnit),
174. .id\_ex\_RegWrite\_o (RegWrite\_to\_ex\_mem),
175. .id\_ex\_MemToReg\_o (MemToReg\_to\_ex\_mem),
176. .id\_ex\_MemRead\_o (MemRead\_to\_ex\_mem),
177. .id\_ex\_MemWrite\_o (MemWrite\_to\_ex\_mem),
178. .id\_ex\_ALUop\_o (ALUop\_out),
179. .id\_ex\_ALUsrc\_o (ALUsrc\_out),
180. .branchAddr\_o (branch\_address), //branch\_address
181. .id\_ex\_pc\_o (id\_ex\_pc\_out),
182. .id\_ex\_pcPlusFour\_o (id\_ex\_pcPlusFour), //wire [31:0] id\_ex\_pcPlusFour;
183. .rd1\_o (rd1\_MUX),
184. .rd2\_o (rd2\_MUX),
185. .imm\_o (imm\_MUX),
186. .ALUctrl\_funct7\_o (funct7\_to\_out),
187. .ALUctrl\_funct3\_o (funct3\_to\_out),
188. .wr\_o (wr\_to\_EX\_MEM),
189. .rs1\_o (rs1\_FW\_in),
190. .rs2\_o (rs2\_FW\_in)
191. );
192. ForwardingUnit FW (
193. .reset\_fw (reset),
194. .ID\_EXrs1 (rs1\_FW\_in),
195. .ID\_EXrs2 (rs2\_FW\_in),
196. .EX\_MEMrd (wr\_to\_MEM\_WB),
197. .EX\_MEMregWrite (RegWrite\_to\_mem\_wb),
198. .MEM\_WBrd (wr\_to\_regFile),
199. .MEM\_WBregWrite (RegWrite\_out),
200. .FW0 (fw0),
201. .FW1 (fw1)
202. );
203. MUX32\_4to1 FW\_data0 (
204. .select\_i (fw0),
205. .data0\_i (rd1\_MUX),
206. .data1\_i (writeData\_to\_regFile),
207. .data2\_i (res\_to\_DataMem\_Addr),
208. .data3\_i (),
209. .data\_o (ALU\_0)
210. );
212. MUX32\_4to1 FW\_data1 (
213. .select\_i (fw1),
214. .data0\_i (rd2\_MUX),
215. .data1\_i (writeData\_to\_regFile),
216. .data2\_i (res\_to\_DataMem\_Addr),
217. .data3\_i (),
218. .data\_o (ALU\_1)
219. );
220. MUX32\_2to1 ALU\_2nd\_input (
221. .select\_i (ALUsrc\_out),
222. .data0\_i (ALU\_1),
223. .data1\_i (imm\_MUX),
224. .data\_o (ALU\_2nd\_in)
225. );
226. ALU\_control ALUcontrol (
227. .ALUctrl\_f7 (funct7\_to\_out),
228. .ALUctrl\_f3 (funct3\_to\_out),
229. .ALUop (ALUop\_out),
230. .ALUctrl\_lines (ALUctrl\_lines)
231. );
232. ALU alu (
233. .reset\_alu (reset),
234. .data0 (ALU\_0),
235. .data1 (ALU\_2nd\_in),
236. .ctrl (ALUctrl\_lines),
237. .result (result),
238. .branch (to\_branchUnit) //wire to\_branchUnit;
239. );
240. Adder32 pcPlusImm\_Utype (
241. .data1 (id\_ex\_pc\_out),
242. .data2 (imm\_MUX),
243. .data\_o (pcPlusImm\_U\_type)
244. );
245. MUX32\_2to1 LUIorAUIPC\_mux (
246. .select\_i (LUIorAUIPC\_to\_mux),
247. .data0\_i (pcPlusImm\_U\_type),
248. .data1\_i (imm\_MUX),
249. .data\_o (Utype\_res)
250. );
251. BranchUnit branch (
252. .clk (clk),
253. .reset\_br (reset),
254. .jump (jump\_to\_branchUnit),
255. .branch (to\_branchUnit),
256. .mux\_to\_pc (sel\_mux\_to\_pc), //wire sel\_mux\_to\_pc;
257. .IF\_Flush (IFreg\_flush), //wire IFreg\_flush, IDreg\_flush;
258. .ID\_Flush (IDreg\_flush)
259. );

262. //---------------------------------------------------------MEM-----------------------------------------------------
263. EX\_MEM ex\_mem (
264. .clk (clk),
265. .ex\_mem\_RegWrite\_i (RegWrite\_to\_ex\_mem),
266. .ex\_mem\_MemToReg\_i (MemToReg\_to\_ex\_mem),
267. .ex\_mem\_MemRead\_i (MemRead\_to\_ex\_mem),
268. .ex\_mem\_MemWrite\_i (MemWrite\_to\_ex\_mem),
269. .ex\_mem\_pcPlusFour\_i (id\_ex\_pcPlusFour),
270. .ex\_mem\_Utype\_res\_i (Utype\_res),
271. .ALUresult\_i(result),
272. .rd2\_i (ALU\_1),
273. .wr\_i (wr\_to\_EX\_MEM),
274. .funct3\_i (funct3\_to\_out), //input for f3
275. .ex\_mem\_RegWrite\_o (RegWrite\_to\_mem\_wb),
276. .ex\_mem\_MemToReg\_o (MemToReg\_to\_mem\_wb),
277. .ex\_mem\_MemRead\_o (MemRead\_out),
278. .ex\_mem\_MemWrite\_o (MemWrite\_out),
279. .ex\_mem\_pcPlusFour\_o (ex\_mem\_pcPlusFour), //ex\_mem\_pcPlusFour
280. .ex\_mem\_Utype\_res\_o (ex\_mem\_Utype\_res),
281. .ALUresult\_o(res\_to\_DataMem\_Addr),
282. .rd2\_o (rd2\_to\_DataMem\_wd),
283. .wr\_o (wr\_to\_MEM\_WB),
284. .funct3\_o (f3\_to\_dataMem) //output for f3 , wire [2:0] f3\_to\_dataMem
285. );
286. DataCache dataMemory (
287. .clk (clk),
288. .addr (res\_to\_DataMem\_Addr),
289. .MemWrite (MemWrite\_out),
290. .MemRead (MemRead\_out),
291. .WriteData (rd2\_to\_DataMem\_wd),
292. .funct3 (f3\_to\_dataMem),
293. .output\_data (DataMem\_out)
294. );
295. //------------------------------------------------------WB-----------------------------------------------------
296. MEM\_WB mem\_wb (
297. .clk (clk),
298. .readMem\_i (DataMem\_out),
299. .ALUresult\_i (res\_to\_DataMem\_Addr),
300. .wr\_i (wr\_to\_MEM\_WB),
301. .mem\_wb\_RegWrite\_i (RegWrite\_to\_mem\_wb),
302. .mem\_wb\_MemToReg\_i (MemToReg\_to\_mem\_wb),
303. .mem\_wb\_pcPlusFour\_i (ex\_mem\_pcPlusFour),
304. .mem\_wb\_Utype\_res\_i (ex\_mem\_Utype\_res),
305. .readMem\_o (Data\_from\_Mem),
306. .ALUresult\_o (ALUres\_toMUX),
307. .wr\_o (wr\_to\_regFile), //.WB\_o (WB\_out)
308. .mem\_wb\_RegWrite\_o (RegWrite\_out), //wire RegWrite\_out, MemToReg\_out;
309. .mem\_wb\_MemToReg\_o (MemToReg\_out),
310. .mem\_wb\_pcPlusFour\_o (mem\_wb\_pcPlusFour), //mem\_wb\_pcPlusFour
311. .mem\_wb\_Utype\_res\_o (mem\_wb\_Utype\_res)
312. );
313. MUX32\_4to1 writeData\_regFile (
314. .select\_i (MemToReg\_out),
315. .data0\_i (Data\_from\_Mem),
316. .data1\_i (ALUres\_toMUX),
317. .data2\_i (mem\_wb\_pcPlusFour),
318. .data3\_i (mem\_wb\_Utype\_res),
319. .data\_o (writeData\_to\_regFile)
320. );
321. endmodule

## 6.2 Testbench αρχείo για την εξομοίωση του μηχανισμού

1. module PipelineDatapathTest();
2. reg clk, reset;
3. PipelineDatapath pd0 (clk, reset);
4. initial begin
5. clk <= 1; reset <= 1; //clk = 0
6. end
7. initial begin
8. #56 reset <= 0;
9. end
11. always #50 clk <= !clk;
12. endmodule

# 7. ΠΑΡΑΡΤΗΜΑ – ΠΡΟΓΡΑΜΜΑΤΑ

Παρακάτω παρουσιάζονται τα προγράμματα που χρησιμοποιήθηκαν για την εκπόνηση της διπλωματική εργασίας:

* Visual Studio Code: Ένα πρόγραμμα που αποτελεί περιβάλλον προγραμματισμού και χρησιμοποιήθηκε για την ανάπτυξη του πηγαίου κώδικα.
* Modelsim SE 5.7f: Ένα πρόγραμμα που λειτούργησε ως compiler του πηγαίου κώδικα και ως εξομοιωτής για τον έλεγχο της σωστής λειτουργίας του μοντέλου.
* Vivado: Ένα πρόγραμμα που χρησιμοποιήθηκε για τις διαδικασίες synthesis και implementation.
* app.diagrams.net: Ένα πρόγραμμα που χρησιμοποιήθηκε για τη δημιουργία διαγραμμάτων που απεικονίζουν την αρχιτεκτονική του μηχανισμού των μερικώς επικαλυπτόμενων λειτουργιών.
* Github: Ένα πρόγραμμα για τον έλεγχο εκδόσεων που χρησιμοποιήθηκε για την αποθήκευση της προόδου της διπλωματικής εργασίας.

Ο πηγαίος κώδικας γράφτηκε στη γλώσσα περιγραφής υλικού Verilog.

Η παρακάτω ιστοσελίδα ήταν χρήσιμη:

* Rvcodec.js: Η ιστοσελίδα παρήγαγε άμεσα κωδικοποιήσεις εντολών στο δεκαεξαδικό σύστημα αρίθμησης. [[7](#_8._ΒΙΒΛΙΟΓΡΑΦΙΑ)]

# 8. ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Harris, S., & Harris, D. (2021). *Digital design and computer architecture: RISC-V Edition*. Morgan Kaufmann.

[2] “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2”, Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, May 2017.

[3] Patterson, D. A., & Hennessy, J. L. (n.d.). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.

[4] Nikolos Dimitrios. *Computer Architecture*. 2017. Pan. Papakonstantinou. ISBN: 978-618-83197-0-7.

[5] “Vivado Design Suite User Guide Synthesis (UG901, v2019.2).”, January 27, 2020. URL: <https://docs.amd.com/v/u/2019.2-English/ug901-vivado-synthesis> .

[6] “Vivado Design Suite User Guide Implementation (UG904, v2023.2).”, November 1, 2023. URL: <https://docs.amd.com/viewer/book-attachment/MpFvKAkfTBjBCiKmeo~Xkw/B2LYIvg~F0HYi4cI4QoWKw> .

[7] Rvcodec.js. URL: <https://luplab.gitlab.io/rvcodecjs/> .