ΑΛΕΞΑΝΔΡΕΙΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ
ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΚΑΤΑΣΚΕΥΗ ΠΡΟΣΟΜΟΙΩΤΗ ΑΝΑΛΥΣΗΣ ΤΩΝ PIPELINE HAZARDS ΜΕ ΤΗ ΧΡΗΣΗ ΤΗΣ ΓΛΩΣΣΑΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ JAVA 2

ΒΑΡΕΛΗΣ ΣΠΥΡΙΔΩΝ-ΝΙΚΟΛΑΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επιβλέπων: Καθ. Αντώνης Βαφειάδης

ΘΕΣΣΑΛΟΝΙΚΗ 2008

ΠΡΟΛΟΓΟΣ – ΕΥΧΑΡΙΣΤΙΕΣ

Η παρούσα εργασία αποτελεί την πτυχιακή μου εργασία με θέμα την ανάπτυξη ενός προσομοιωτή, ο οποίος θα βρίσκει και θα προτείνει μία λύση για την αποφυγή των pipeline hazards.

Η εργασία αυτή δημιουργήθηκε με βάση την γλώσσα προγραμματισμού Java 2 και το εργαλείο που χρησιμοποιήθηκε για την συγγραφή του κώδικα του προσομοιωτή, ήταν το Netbeans IDE 6.0.

Πραγματοποιήθηκε κατά το ακαδημαϊκό έτος 2007-2008 απο τον Νοέμβριο έως τον Ιούνιο.

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Αντώνη Βαφειάδη για την βοήθεια που μου πρόσφερε ώστε να φέρω εις πέρας την πτυχιακή μου εργασία.

Στους γονείς μου και την οικογένεια μου οφείλω ενα μεγάλο ευχαριστώ γιατί είναι πάντα δίπλα μου και με στηρίζουν σε οτι κάνω.

Τελος, ενα θερμό ευχαριστώ στους φίλους μου και γενικά σε όλους όσους βοήθησαν, ο καθένας με τον δικό του τρόπο, ώστε να πραγματοποιηθεί η πτυχιακή αυτή.

ΠΕΡΙΕΧΟΜΕΝΑ

Α. ΠΕΡΙΛΗΨΗ

Η τεχνική της Pipeline έχει κατορθώσει να επιταχύνει την απόδοση των επεξεργαστών και κατ' επέκταση την απόδοση των υπολογιστικών συστημάτων. Χώρίς τη μέθοδο αυτή, οι επεξεργαστές θα εκτελούσαν μια-μια τις εντολές των διαφόρων προγραμμάτων.

Η Pipeline είναι μια τεχνική που χρησιμοποιείται ευρέως στο κόσμο της επιστήμης των υπολογιστών αλλά και σε άλλες ψηφιακές, ηλεκτρονικές συσκευές με σκοπό να αυξήσει τον αριθμό των εντολών που μπορούν να εκτελεστούν στη μονάδα του χρόνου. Ο επεξεργαστής μπορεί να εκτελέσει πολλαπλές εντολές ταυτόχρονα. Αυτό επιτυγχάνεται με τη διάσπαση μιας διαδικασίας (ενός προς εκτέλεση προγράμματος) σε διαφορετικές φάσεις, οπότε διαφορετικές διαδικασίες μπορούν να εκτελούνται παράλληλα καθώς βρίσκονται σε διαφορετικές φάσεις η κάθε μια.

Παρόλο που η Pipeline προσφέρει πολλά σε ενα υπολογιστικό σύστημα, μπορεί να επιφέρει και κόστος. Το κόστος αυτό είναι είτε οικονομικό (προσθήκη επιπλέον μονάδων για να μπορεί να υπάρξει και να λειτουργήσει όπως ακριβώς έχει ορισθεί η Pipeline) είτε αποδοτικό, δηλαδή παρόλο που αυξάνεται η απόδοση, δεν επιτυγχάνεται το μέγιστο δυνατό αποτέλεσμα. Αυτό συμβαίνει λόγω κάποιων κινδύνων που «καραδωκούν». Αυτοί οι κίνδυνοι μπορεί να είναι δομικοί, κίνδυνοι δεδομένων ή κίνδυνοι ελέγχου.

Στην παρούσα εργασία γίνεται μια παρουσίαση των κινδύνων αυτών και με ποιούς τρόπους μπορούν να απαλειφθούν. Για να το πετύχουμε αυτό, εκτός από το θεωρητικό μέρος (θεωρητική προσέγγιση), αναπτύχθηκε και ένας προσομοιωτής, ο οποίος ανιχνεύει τέτοιου είδους κινδύνους και δίνει μία λύση, αν αυτό είναι εφικτό.

Ο προσομοιωτής αυτός κατασκεύαστηκε με τη γλώσσα προγραμματισμού Java 2.

Β. ΕΙΣΑΓΩΓΗ

Β1. Pipeline

Η pipeline, έχει ευρεία χρήση σε οποιδήποτε διαδικασία παράγει κάποιο αποτέλεσμα (προϊόν). Σε αυτή τη γενική περίπτωση, ένας ορισμός που θα ταίριαζε της είναι:

Pipeline *είναι η λειτουργία κατά την οποία η εκτέλεση μιας διαδικασίας χωρίζεται σε διαφορετικές φάσεις, οπότε διαφορετικές διαδικασίες μπορούν να εκτελούνται ταυτόχρονα καθώς βρίσκονται σε διαφορετικές φάσεις η κάθε μία.*

Το pipelining είναι μια φυσική έννοια που χρησιμοποιείται σε διάφορες φάσεις της καθημερινότητας μας. Για παράδειγμα, μπορούμε να αναφέρουμε τη γραμμή συναρμολόγησης αυτοκινήτων. Ας θεωρήσουμε ότι τα κύρια στάδια για τη συναρμολόγηση ενός αυτοκινήτου είναι η τοποθέτηση της μηχανής, η τοποθέτηση του αμαξώματος (πορτες, μπάρες προστασίας, κλπ), τοποθέτηση τροχών.

Στάδια που περιέχονται στη γραμμή συναρμολόγησης ενός αυτοκινήτου
1. Τοποθέτηση αμαξώματος
2. Τοποθέτηση μηχανής
3. Τοποθέτηση τροχών

Πίνακας 1. *Στάδια συναρμολόγησης αυτοκινήτων*

Ένα αυτοκίνητο, στη γραμμή συναρμολόγησης, μπορεί να βρίσκεται μόνο σε ένα απο τα τρία στάδια την κάθε στιγμή. Έστω τώρα οτι υπάρχουν και τρείς εργάτες, ένας ειδικευμένος για κάθε διαδικασία (στάδιο παραγωγής).

Η συναρμολόγηση ξεκινάει με την τοποθέτηση του αμαξώματος στο πρώτο αυτοκίνητο. Όταν τελειώσει αυτό το στάδιο, τότε το πρώτο αυτοκίνητο προχωράει προς το δεύτερο. Έτσι, ο εργάτης που απασχολείται στην τοποθέτηση του αμαξώματος μένει

διαθέσιμος για το επόμενο αυτοκινήτο που θα μπεί στην γραμμή συναρμολόγησης. Στη συνέχεια, το πρώτο αμάξι προχωράει στην τοποθέτηση της μηχανής και το δεύτερο στην τοποθέτηση του αμαξώματος. Καθώς το πρώτο αμάξι τελειώνει απο το στάδιο της τοποθέτησης της μηχανής και το δεύτερο τελειώνει την τοποθέτηση του αμαξώματος, ένα τρίτο αμάξι εισέρχεται στη ουρά και ξεκινάει την τοποθέτηση του αμαξώματος, Τα δύο πρώτα έχουν πλέον μεταφερθεί στα επόμενα στάδια, δηλαδή στην τοποθέτηση των τροχών (για το πρώτο αμαξι) και στην τοποθέτηση της μηχανής (για το δεύτερο αμάξι). Η λογική είναι οτι καθώς ένα αμάξι τελειώνει το στάδιο που βρίσκεται, ο εργάτης μένει ελεύθερος και διαθέσιμος για το αμάξι που ακολουθεί τη γραμμή συναρμολόγησης.

Αν το στάδιο της τοποθέτησης του αμαξώματος διαρκεί 40 λεπτά, το στάδιο της τοποθέτησης της μηχανής 20 λεπτά και το σταδιο της τοποθέτησης των τροχών 15 λεπτά, τότε ένα αυτοκίνητο χωρίς την εφαρμογή του pipelining, θα κατασκευαζόταν κάθε 75 λέπτα (40+20+15). Παρόλα αυτά, με τη χρήση του pipelining, το πρώτο αμάξι θα κατασκευαστεί σε 75 λεπτά (μέχρι να «γεμίσει» η pipeline), αλλά τα υπόλοιπα που θα ακολουθήσουν θα κατασκευαστούν σε συντομότερο χρονικό διάστημα καθώς ήδη βρίσκονται σε κάποιο στάδιο συναρμολόγησης. Παρακάτω ακολουθεί η σύγκριση της συναρμολόγησης αυτοκινήτων χώρις και με την εφαρμογή της pipeline. Τα κελιά των πινάκων που είναι με μπλέ φόντο δηλώνουν πότε ένα αυτοκίνητο ολοκληρώνει την κατασκευή του.

	Χρόνος								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Αμάξι 1	Στάδιο1	Στάδιο2	**Στάδιο3**						
Αμάξι 2				Στάδιο1	Στάδιο2	**Στάδιο3**			
Αμάξι 3							Στάδιο1	Στάδιο2	**Στάδιο3**

Πίνακας 2. *Συναρμολόγηση των αυτοκινήτων χωρίς την εφαρμογή της pipeline*

	Χρόνος				
	1	**2**	**3**	**4**	**5**
Αμάξι 1	Στάδιο 1	Στάδιο 2	**Στάδιο 3**		
Αμάξι 2		Στάδιο 1	Στάδιο 2	**Στάδιο 3**	
Αμάξι 3			Στάδιο 1	Στάδιο 2	**Στάδιο 3**

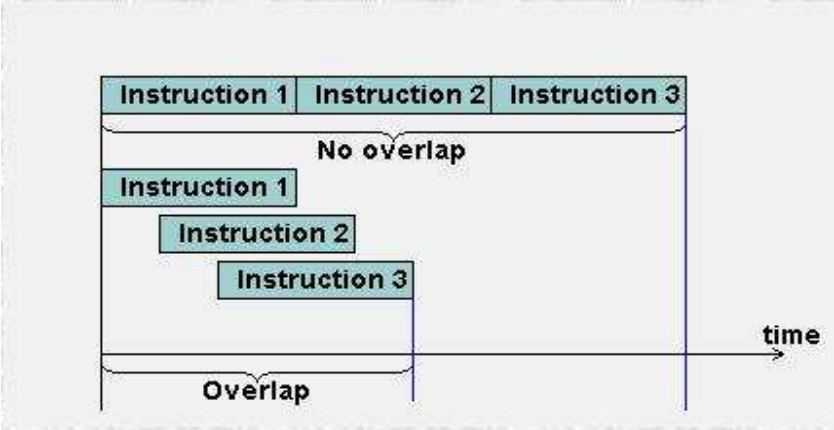
Πίνακας 3. *Συναρμολόγηση των αυτοκινήτων με την εφαρμογή της pipeline*

Β2. Το Pipelining στην επιστήμη των υπολογιστών

Στην τεχνολογία των επεξεργαστών η τεχνική του *Pipelining* είναι εξαιρετικά διαδεδομένη και πλεόν αποτελεί αναπόσπαστο κομμάτι της αρχιτεκτονικής των υπολογιστών.

Στην επιστήμη των υπολογιστών, η pipeline είναι ένα σύνολο δεδομένων, τα οποία πρόκειται να εκτελεστούν, τοποθετημένα σε μια σειρά, έτσι ώστε η έξοδος του ενός στοιχείου δεδομένων να είναι η είσοδος του επόμενου. Αυτα τα στοιχεία εκτελούνται παράλληλα. Η pipeline των υπολογιστών χωρίζεται σε ***φάσεις (stages)***. Σε κάθε φάση ολοκληρώνεται και ένα μέρος της εντολής. Οι φάσεις είναι συνδεδεμένες η μία μετά την άλλη ώστε να δημιουργηθεί μία ουρά (διασωληνωση).

Το pipelining δεν μειώνει τον χρόνο στον οποίο εκτελείται μια εντολή, αλλά αυξάνει την ***απόδοση της εντολής (instruction throughput)***. Η απόδοση μιας εντολής ορίζεται ώς ο ρυθμός εξαγωγής μιας εντολής απο την pipeline.

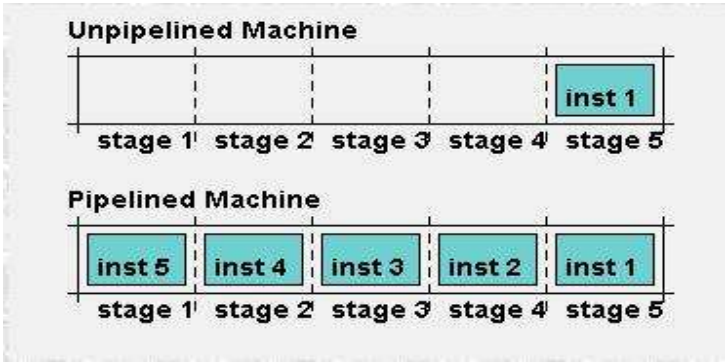


Εικόνα 1. *Απόδοση εντολής*

Ο στόχος ενός σχεδιαστή pipeline είναι να ισορροπήσει το μήκος της κάθε φάσης. Εάν οι φάσεις είναι τέλεια ισορροπημένες, τότε ο χρόνος για την παραμονή κάθε εντολής στην pipeline είναι:

$$\frac{\textit{Χρόνος για κάθε εντολή χωρίς pipeline}}{\textit{Αριθμός των φάσεων της pipeline}}$$

Σαν συμπέρασμα, προκύπτει ότι η επιτάχυνση της pipeline ισούται με τον αριθμό των φάσεων. Παρόλα αυτά, όμως, οι φάσεις δεν θα είναι ποτέ σχεδόν τέλεια ισορροπημένες. Στη συνέχεια, θα περιγράψουμε τις αρχές της pipeline με τη χρήση του υποθετικού επεξεργαστή DLX.



Εικόνα 2. *Σύγκριση μηχανής με τη χρήση της Pipeline και χωρίς αυτήν*

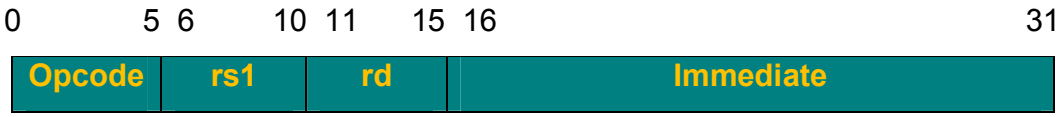
B3. Ο υποθετικος επεξεργαστής DLX

Σε αυτήν την ενότητα θα περιγράψουμε τον υποθετιό επεξεργαστή ***DLX***, τον επεξεργαστή που βασιστήκαμε για να περιγράψουμε την pipeline.

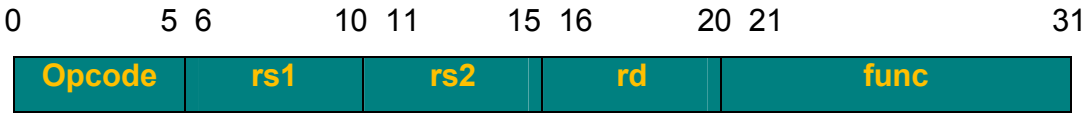
Ο DLX έχει 32 καταχωρητές (registers) γενικού σκοπού, μεγέθους 32 bits ο καθένας. Οι καταχωρητές αυτοί ονομάζονται R0, R1, R2, …, R31. Επίσης, υπάρχουν 32 καταχωρητές για πράξεις κινητής υποδιαστολής, F0, F1, F2, …, F31 του ιδίου μεγέθους.

Οι τρόποι διευθυνσιοδότησης στον DLX είναι δύο: α) ***ο άμεσος (immediate)*** και β) ***η μετατόπιση (displacement)***. Με αυτούς τους δύο τρόπους κλήσεως της μνήμης μπορούμε να υλοποιήσουμε ακόμα δύο τρόπους, την ***έμμεση κλήση (indirect)*** και την ***απόλυτη κλήση (absolute)*** χρησιμοποιώντας το μηδέν (0) ή το ειδικό καταχωρητή R0.

Οι εντολές στον DLX είναι του ίδιου μήκους, 32 bits. Συγκεκριμένα, οι εντολές είναι τριών τύπων και παρουσιάζονται στο ακόλουθο σχήμα (Σχήμα 1).



Εντολές τύπου I, π.χ. load, store



Εντολές τύπου R, π.χ. πράξεις μεταξύ καταχωρητών

Εντολές τύπου J, π.χ. άλματα

Σχήμα 1. *Οι τύποι των εντολών της γλώσσας DLX. Χωρίζονται σε εντολές τύπου I (immediate), τύπου R (register-to-register) και τύπου J (jumps)*

Β4. Οι φάσεις των εντολών

Γενικά, για να εκτελεστεί ένα πρόγραμμα, πρέπει η ΚΜΕ (Κεντρική Μονάδα επεξεργασίας, CPU) να ανακαλέσει τις εντολές του προγράμματος που είναι αποθηκευμένες στην κύρια μνήμη και να εκτελέσει τις λειτουργίες που αναφέρονται σε αυτές. Η διεύθυνση στην οποία βρίσκεται η προς εκτέλεση εντολή, είναι το περιεχόμενο του μετρητή προγράμματος (Program Counter). Οι εντολές ανακαλούνται απο τις θέσεις μνήμης διαδοχικά. Μετά την ανάκληση της εντολής, το περιεχόμενο του PC αλλάζει και δείχνει την επόμενη προς εκτέλεση εντολή.

Η pipeline του DLX επεξεργαστή χωρίζεται σε πέντε φάσεις. Σε κάθε φάση γίνεται και μια διαφορετική διαδικασία. Οι φάσεις αυτές είναι:

I. Φάση λήψης της εντολής (*Instruction Fetch*)
II. Φάση αποκωδικοποίησης/ανάγνωσης της εντολής (*Instruction Decode*)
III. Φάση εκτέλεσης (*Execution*)
IV. Φάση πρόσβασης στη μνήμη/ολοκλήρωσης αλμάτων (*Memory Access*)
V. Φάση εγγραφής καταχωρητών (*Write Back*)

Ακολουθεί μια σύντομη περιγραφή της κάθε φάσης.

Β4.1. Φάση λήψης εντολής

Η λειτουργία της φάσης αυτής περιγράφεται στη συνεχεια.

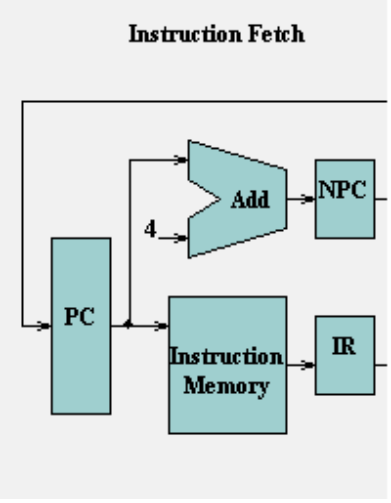
$$MAR \leftarrow Mem[PC]$$
$$IR \leftarrow Mem[MAR]$$
$$NPC \leftarrow PC + 4$$

- Γίνεται ανάκληση της εντολής από τη μνήμη, της οποίας την διεύθυνση την παίρνει από τον PC και αποθηκεύει αυτή στον καταχωρητή IR.
- Ο IR χρησιμοποιείται για να κρατήσει την εντολή που θα εκτελεστεί, κάτι το οποίο είναι απαραίτητο όταν έχουμε εκτέλεση εντολών σε πολλούς κύκλους.

- Αύξηση του καταχωρητή PC κατά 4 (NPC) που δείχνει την αρχή της διεύθυνσης στη μνήμη της επόμενης εντολής που θα εκτελεστεί.

Εδώ πρέπει να σημειώσουμε ότι ο καταχωρητής *MAR* είναι καταχωρητής ειδικής χρήσης και συνδέεται μέσω της αρτηρίας διευθύνσεων με τη μνήμη του επεξεργαστή. Επομένως, όποια διεύθυνση αποθηκευτεί στον καταχωρητή MAR, μεταφέρεται στη μνήμη. Ο IR είναι ο καταχωρητής εντολών του επεξεργαστή, δηλ. κάθε φορά εκτελείται η εντολή που βρίσκεται στον συγκεκριμένο καταχωρητή.

Η διεύθυνση της εντολής που πρόκειται να εκτελεστεί περιέχεται στον απαριθμητή προγράμματος (PC). Κατά την ανάκληση εντολής, η ΚΜΕ πρέπει να προσδιορίσει τη διεύθυνση της θέσης μνήμης που είναι αποθηκευμένη η εντολή και να ζητήσει τη λειτουργία ανάγνωσης. Έτσι, μεταφέρεται η διεύθυνση εντολής από τον PC στον καταχωρητή διευθύνσεων μνήμης (MAR), ο οποίος συνδέεται μέσω της αρτηρίας διευθύνσεων με τη μνήμη. Η ΚΜΕ χρησιμοποιεί τις γραμμές που μεταφέρουν σήματα ελέγχου για να υποδείξει στη μνήμη ότι ζητείται μία λειτουργία ανάγνωσης. Η εντολή ανακαλείται από τη μνήμη και αποθηκεύεται μέσω του MDR, στον καταχωρητή εντολών (IR). Αυτό ολοκληρώνει τη λειτουργία ανάκλησης εντολής.



Εικόνα 3. *Φάση λήψης εντολής (Instruction Fetch).*

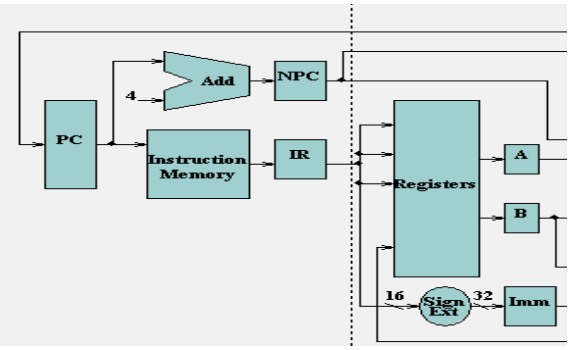
B4.2. Φάση αποκωδικοποίησης εντολής / Ανάγνωση των καταχωρητών

Λειτουργία της φάσης αυτής:

$$A \leftarrow Reg[IR_{6...10}]$$
$$B \leftarrow Reg[IR_{11...15}]$$
$$Imm \leftarrow [(IR16)16\#\#IR_{16...31}]$$

- Αποκωδικοποίηση της εντολής και πρόσβαση στο αρχείο καταχωρητών.
- Η έξοδος από τους καταχωρητές γενικής χρήσης αποθηκεύεται μέσα στους δύο προσωρινούς καταχωρητές A και B για να χρησιμοποιηθεί αργότερα στους επόμενους κύκλους του ρολογιού.
- Τα χαμηλότερα 16 bits του καταχωρητή IR επιδέχονται επέκταση πρόσημου και αποθηκεύονται μέσα στον προσωρινό καταχωρητή Imm για να χρησιμοποιηθούν σε επόμενο κύκλο.
- Η αποκωδικοποίηση γίνεται παράλληλα με την ανάγνωση από το αρχείο καταχωρητών, κάτι που είναι δυνατό λόγω της σταθερής θέσης των πεδίων στην εντολή του επεξεργαστή DLX (είναι γνωστή σαν fixed-field decoding τεχνική).

Στο βήμα αυτό, αποκωδικοποιείται η εντολή και ταυτόχρονα γίνεται πρόσβαση στο αρχείο των καταχωρητών για να διαβαστούν οι πηγαίοι καταχωρητές. Αυξάνεται ο απαριθμητής προγράμματος ώστε να δείχνει την επόμενη εντολή.



Εικόνα 4. *Φάση αποκωδικοποίησης / ανάγνωσης καταχωρητών (Instruction Decode)*

B4.3 Φάση της εκτέλεσης

Στη φάση αυτή, οι καταχωρητές που διαβάστηκαν στο προηγούμενο βήμα, βρίσκονται στην είσοδο της ALU. Η ALU πραγματοποιεί μία από τις τρείς λειτουργίες, ανάλογα με τον τύπο της εντολής του DLX.

Επικοινωνία με τη Μνήμη (εντολές φόρτωσης / αποθήκευσης):

$$MAR \leftarrow A + (IR16)^{16} \#\# IR_{16...31}$$
$$MDR \leftarrow B$$

Η ALU προσθέτει το περιεχόμενο του πηγαίου καταχωρητή (A) με το περιεχόμενο του καταχωρητή εντολών, το οποίο έχει υποστεί επέκταση προσήμου στα ψηφία 16 έως 31. Έτσι σχηματίζεται η ενεργός διεύθυνση, η οποία μεταφέρεται στον MAR. Στην περίπτωση εντολής αποθήκευσης ο MDR φορτώνεται με την τιμή του B.

Αριθμητική ή λογική εντολή:

$$ALU_{output} \leftarrow A \ op \ (B \ or \ (IR_{16})^{16} \#\# IR_{16...31})$$

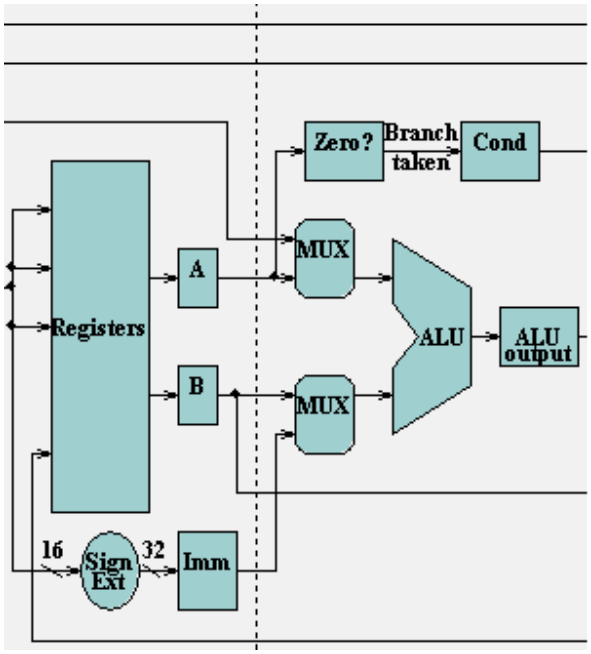
Η ALU εκτελεί τη λειτουργία που καθορίζεται από τον κωδικό λειτουργίας μεταξύ του A (Rs1) και του B ή μεταξύ του A και της τιμής που βρίσκεται στα τελευταία 16 bits του καταχωρητή εντολών ($(IR_{16})^{16}$ ## $IR_{16...31}$) .

Διακλάδωση με συνθήκη / Μεταπήδηση:

$$ALU_{output} \leftarrow PC + (IR_{16})^{16} \#\# IR_{16...31}$$
$$Cond \leftarrow (A \ op \ 0)$$

Η ALU προσθέτει στον PC την τιμή του offset για να υπολογίσει τη διεύθυνση του στόχου διακλάδωσης. Για διακλαδώσεις με συνθήκη, ο καταχωρητής A ελέγχεται για να

αποφασιστεί αν αυτή η διεύθυνση θα πρέπει να γίνει η νέα τιμή του PC. Η λειτουργία σύγκρισης op είναι ο σχετικός τελεστής που καθορίζεται από τον κωδικό λειτουργίας της εντολής. Για παράδειγμα, ο op είναι ο "==" για την εντολή BEQZ.



Εικόνα 5. *Φάση εκτέλεσης*

Β4.4 Φάση πρόσβασης στη μνήμη / ολοκλήρωση αλμάτων

Λειτουργία της φάσης της πρόσβασης στη μνήμη:

Επικοινωνία με τη Μνήμη:

$$LW: MDR \leftarrow Mem[MAR]$$

$$SW: Mem[MAR] \leftarrow MD$$

- Γίνεται πρόσβαση στη μνήμη όταν αυτό απαιτείται.
- Αν έχουμε εντολή φόρτωσης, τα δεδομένα διαβάζονται από τη μνήμη και τοποθετούνται στον καταχωρητή MDR.

- Αν έχουμε εντολή αποθήκευσης τα δεδομένα διαβάζονται από τον καταχωρητή B και εγγράφονται στη μνήμη.
- Σε κάθε μια από τις δύο προηγούμενες περιπτώσεις η διεύθυνση που χρησιμοποιείται είναι αυτή που έχει υπολογιστεί στον προηγούμενο κύκλο και αποθηκευτεί στον καταχωρητή ALUOutput.

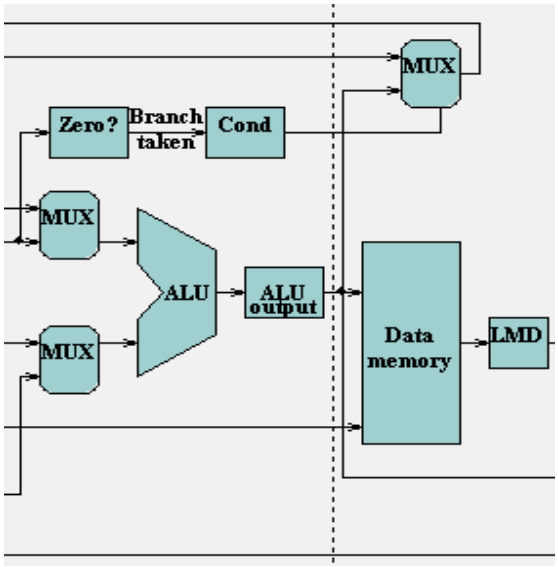
  Σε περίπτωση που η εντολή μας είναι άλμα (branch) τότε ισχύει ότι:

  Διακλάδωση με συνθήκη :

$$if\ (cond)\ PC \leftarrow ALU_{output}$$

$$else\ PC \leftarrow NPC$$

- Εάν έχουμε εντολή διακλάδωσης, το περιεχόμενο του καταχωρητή PC αντικαθίσταται με τη διεύθυνση του στόχου της διακλάδωσης που βρίσκεται στον καταχωρητή ALUOutput.
- Διαφορετικά ο PC παίρνει περιεχόμενο, αυτό του καταχωρητή NPC.



Εικόνα 6. *Φάση πρόσβασης στην μνήμη / ολοκλήρωσης αλμάτων*

Β4.5 Φάση εγγραφής καταχωρητών / μνήμης

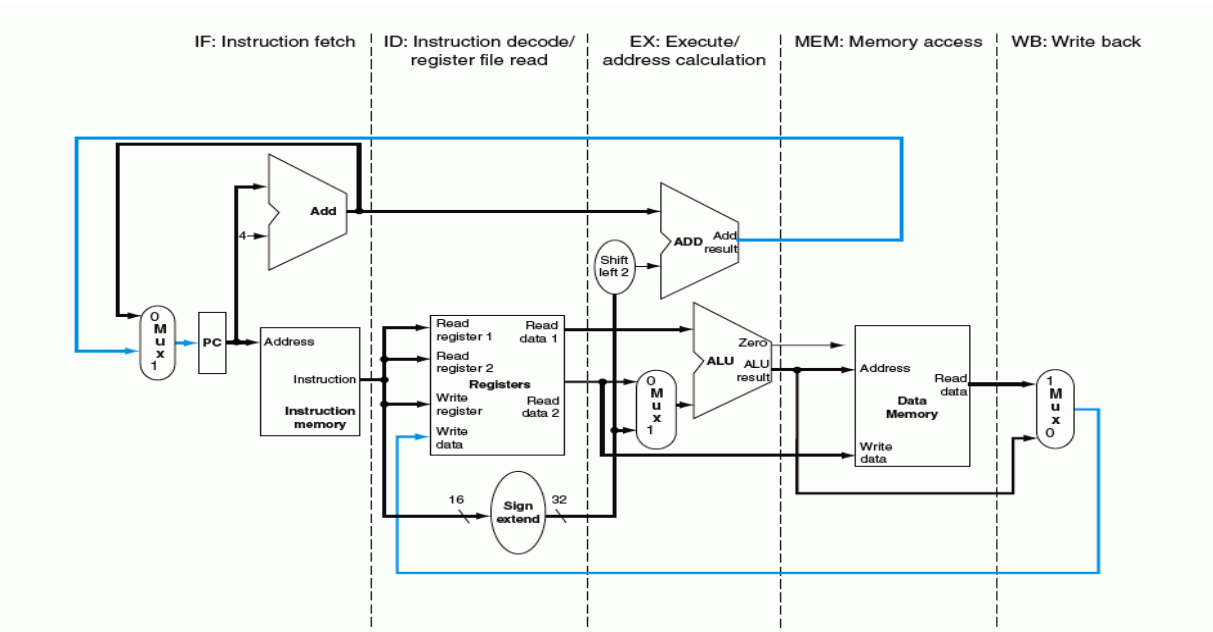
Η φάση της εγγραφής των καταχωρητών είναι και η τελευταία φάση εκτέλεσης των εντολών στην αρχιτεκτονική του DLX.

Γενικά:

$$Rd \leftarrow ALU_{output} \text{ ή } MDR$$

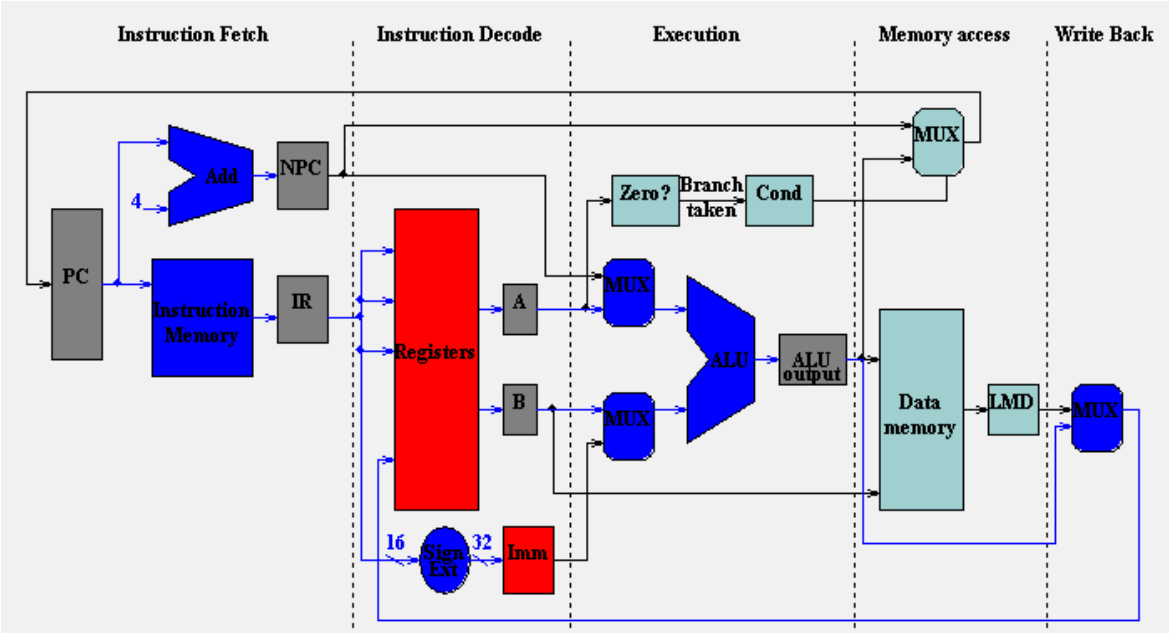
- Αριθμητική ή λογική εντολή (καταχωρητή - καταχωρητή)
  $Reg[IR_{16..20}] \neg ALUOutput$
- Αριθμητική ή λογική εντολή (καταχωρητή - Immediate)
  $Reg[IR_{11..15}] \neg ALUOutput$
- Εντολή φόρτωσης
  $Reg[IR_{11..15}] \neg MDR$

Η εντολή αυτή γράφει το αποτέλεσμα στο αρχείο καταχωρητών, το οποίο αποτέλεσμα προέρχεται είτε από τη μνήμη είτε από την ALU (ALUOutput). Ο καταχωρητής προορισμού εξαρτάται από το opcode.



Εικόνα 7. *Συνολική αποψη των φάσεων της Pipeline*

B5. Η υλοποίηση της Pipeline στον DLX



Εικόνα 8. *Η Pipeline του DLX κατά τη διάρκεια εκτέλεσης μιας Register-to-Register εντολής*

Στην παραπάνω εικόνα (Εικόνα 8), φαίνεται η βασική υλοποίηση της Pipeline του επεξεργαστη DLX. Οι πέντε φάσεις ξεχωρίζουν, καθώς επίσης φαίνονται και όλοι οι καταχωρητές, οι πολυπλέκτες (Multiplexers – MUX) και οι μονάδες Αριθμητικής και Λογικής επεξεργασίας (ALU). Η εικόνα δείχνει την εκτέλεση μιας εντολής Register-to-Register (ALU). Όπως παρατηρούμε, στους καταχωρητές γενικού σκοπού έχει αντιγραφεί το περιεχόμενο του καταχωρητή ALUOutput (η μονάδα Registers είναι κόκκινη).

Η Pipeline των εντολών του υποθετικού επξεργαστή DLX υλοποιείται με τη χρήση καταχωρητών μεταξύ των φάσεων, οι οποίοι φυλάσσουν όλη την πληροφορία που χρειάζεται η κάθε φάση απο την προηγούμενη. Κάθε φάση της pipeline διαρκεί έναν ***κύκλο CPU (cpu cycles).*** Κάθε εντολή χρειάζεται πέντε κύκλους ρολογιού για να ολοκληρωθεί η επεξεργασία της.

Στην ιδανική περίπτωση, αν υποθέσουμε οτι δεν υπάρχει καμία σύγκρουση μεταξύ των εντολών, η εκτέλεση τους θα γίνεται με τον τρόπο που φαίνεται στον παρακάτω πίνακα, όπου σε κάθε κύκλο μηχανής ολοκληρώνεται και απο μία εντολή.

	Κύκλοι CPU						
	1	2	3	4	5	6	7
Εντολη 1	IF	ID	EX	MEM	WB		
Εντολη 2		IF	ID	EX	MEM	WB	
Εντολη 3			IF	ID	EX	MEM	WB

Πίνακας 5. *Η ιδανική Pipeline*

Γ. ΟΙ ΚΙΝΔΥΝΟΙ ΤΗΣ PIPELINE

Γ1. Εισαγωγή

Στο κεφάλαιο αυτό θα γίνει αναφορά στο σημαντικότερο πρόβλημα που αντιμετωπίζει η εφαρμογή της Pipeline, τους ***κινδύνους (hazards)***. Υπάρχουν κάποιες καταστάσεις, στις οποίες η επόμενη εντολή στην pipeline, δεν μπορεί να εκτελεστεί στον επόμενο κύκλο ρολογιού. Οι κίνδυνοι (Hazards), όπως ονομάζονται αυτές οι καταστάσεις, προκύπτουν και κατηγοριοποιούνται ανάλογα, απο τρείς διαφορετικές αιτίες.

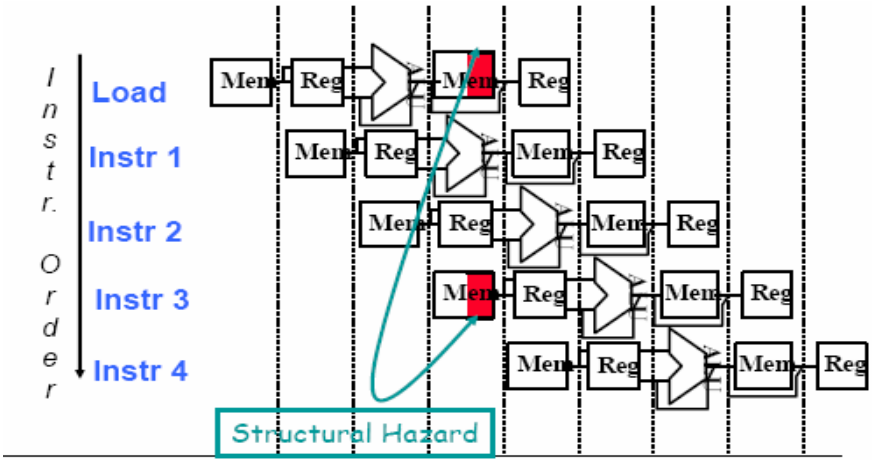
1. Οι **δομικοί κίνδυνοι (Structural hazards)**, προκύπτουν στις περιπτώσεις που το υλικό (hardware) δεν αρκεί για να καλύψει τις ανάγκες όλων των δυνατών συνδυασμών εντολών που μπορούν να προκύψουν στην Pipeline.
2. Οι **κίνδυνοι δεδομένων (Data hazards)**, προκύπτουν όταν η εκτέλεση μιας εντολής εξαρτάται απο το αποτέλεσμα μιας προηγούμενης εντολής.
3. Οι **κίνδυνοι ελέγχου (Control hazards)**, προκύπτουν απο τις εντολές άλματος, οι οποίες αλλάζουν τη ροή του προγράμματος.

Τέλος, με τη βοήθεια προσομοιωτή των Pipeline Hazards που κατασκευάσαμε, θα εκτελέσουμε κάποια παραδείγματα για κάθε κίνδυνο.

Γ2. Δομικοί Κίνδυνοι

Το πρώτο είδος κινδύνων ονομάζεται ***δομικοί κίνδυνοι***. Αυτό σημαίνει οτι το υλικό δεν μπορεί να υποστηρίξει των συνδυασμό των εντολών που θέλουν να εκτελεστούν στον ίδιο κύκλο ρολογιού. Ας υποθέσουμε για παράδειγμα, οτι είχαμε μία μόνο μνήμη αντί για δύο (όπως έχουμε κανονικά). Εάν σε μία pipeline είχαμε δύο εντολές, οι οποίες στον ίδιο κύκλο, η μεν πρώτη εγγράφει δεδομένα στη μνήμη και η δε δεύτερη φορτώνει δεδομένα απο την ίδια μνήμη. Χωρίς να έχουμε δύο μνήμες, η pipeline μας παρουσιάζει δομικό κίνδυνο.

Όταν μια μηχανή χρησιμοποιέι την pipeline, τότε η εκτέλεση των εντολών απαιτεί τη διασωλήνωση των λειτουργικών μονάδων (π.χ. ALU) καθώς και τις διπλές δομικές μονάδες (πόρους) ώστε να επιτρέψει να γίνουν όλοι οι πιθανοι συνδιασμοι των εντολών στην Pipeline.



Εικόνα 8. *Δομικοί κίνδυνοι*

Παράδειγμα

Έστω οτι έχουμε μια μονής μνήμης pipeline για δεδομένα και εντολές. Σαν αποτέλεσμα, όταν μια εντολή περιέχει μια αναφορά στη μνήμη δεδομένων (load), τότε θα συγκρουστεί με την αναφορά στη μνήμη των εντολών όταν έρθει μια νεα εντολή.

Εντολή	Κύκλοι ρολογιού							
	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

Για να το επιλύσουμε αυτό, τοποθετούμε νεκρούς κύκλους.

Εντολή	1. Κύκλοι ρολογιού								
	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Stall				Bubble	Bubble	Bubble	Bubble	bubble	
Instr 3					IF	ID	EX	MEM	WB

Πιο απλά, ο παραπάνω πίνακας μπορεί να γραφεί έτσι:

Εντολή	1. Κύκλοι ρολογιού								
	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				stall	IF	ID	EX	MEM	WB

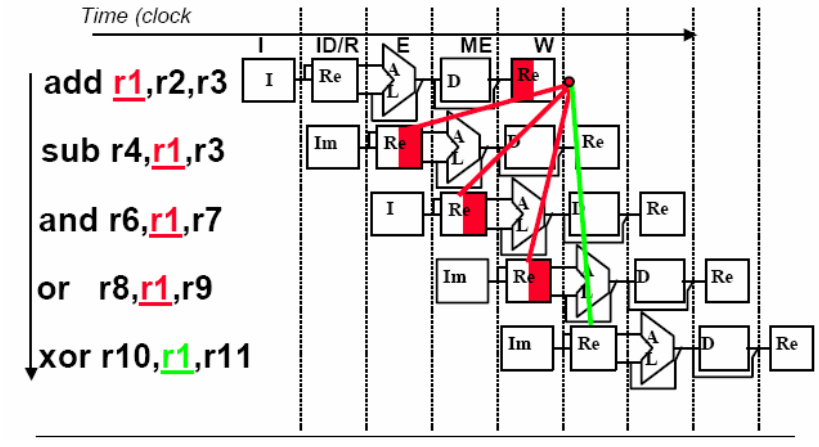
Το ερώτημα είναι το εξής, αφού μπορούμε να αποφύγουμε τους δομικούς κινδύνους, γιατί οι σχεδιαστές επιτρέπουν να υπάρχουν αυτοί οι κίνδυνοι;

Ο λόγος είναι η μείωση του κόστους. Για παράδειγμα, μηχανές που υποστηρίζουν την ταυτόχρονη πρόσβαση στη μνημη δεδομένων και εντολών (για την αποφυγη του κινδυνου όπως παραπάνω) χρειάζονται τουλάχιστον και instruction cache αλλά και data cache.

Γ3. Κίνδυνοι δεδομένων

Ο ***κίνδυνοι δεδομένων***, αποτελούν και το πιο σημαντικό πρόβλημα στην διαδικασία της Pipeline. Αυτού του είδους οι κίνδυνοι συμβαίνουν όταν η Pipeline αλλάζει τη σειρά πρόσβασης αναγνωσης/εγγραφής των τελεστών, ετσι ώστε η σειρά να διαφέρει απο αυτή που θα είχαμε σε μια εκτέλεση εντολών χωρίς Pipeline.

Η σημαντικότερη επιπλοκή στη χρήση της Pipeline είναι όταν υπάρχει χρονική συσχέτιση της εκτέλεσης των φάσεων των διαφορετικών εντολών. Για παράδειγμα, αν το αποτέλεσμα μιας πράξης που γίνεται σε μια εντολή x χρησιμοποιείται απο μία άλλη εντολή y, επόμενη στη σειρά, είναι δυνατόν λόγω του pipelining, να χρησιμοποιείται το αποτέλεσμα στην εντολή y πρίν παραχθεί απο την εντολή x.



Εικόνα 9. *Κίνδυνοι δεδομένων*

Παράδειγμα

Έστω ότι έχουμε να εκτελέσουμε τις ακόλουθες εντολές με τη χρήση της Pipeline.

ADD R1, R2, R3

SUB R4, R1, R5

Ο πίνακας που θα παραχθεί, χωρίς τη χρήση της Pipeline, θα έχει τη μορφή:

Κύκλοι CPU						
Εντολή	**1**	**2**	**3**	**4**	**5**	**6**
ADD R1, R2, R3	IF	ID	EX	MEM	WB	
SUB R4, R1, R5		IF	ID	EX	MEM	WB

Πίνακας 6. *Αρχική μορφή εκτέλεσης (χωρίς Pipeline)*

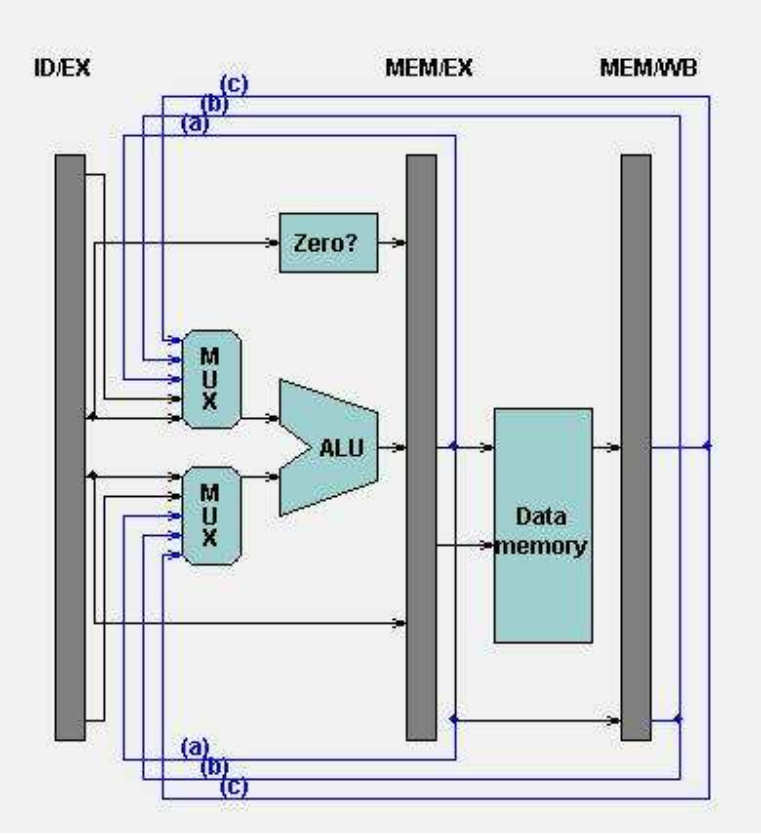
Από τον παραπάνω πίνακα μπορούμε να δούμε οτι η εντολή *SUB*, η οποία βρίσκεται μετά την *ADD*, θέλει να χρησιμοποιησεί το αποτέλεσμα της εντολής αυτής που είναι θα αποθηκευτεί στον καταχωρητή *R1*. Αυτό όμως δεν μπορεί να συμβεί και έτσι, προκύπτει κίνδυνος δεδομένων στην ακολουθία εντολών που δείνχει ο πίνακας και συγκεκριμένα μεταξύ της φάσης *WB* της εντολής *ADD*, η οποία παράγει το αποτέλεσμα που πρόκειται να εγγραφεί στον *R1*, και της φάσης *ID* της εντολής *SUB*, η οποία διαβάζει το αποτέλεσμα του *R1* πριν αυτό παραχθεί απο την *ADD*. Υπενθυμίζουμε ότι έτσι όπως έχουν σχεδιάστει οι φάσεις των εντολών, το αποτέλεσμα μιας πράξης γράφεται στη φάση *WB* στον κατάλληλο καταχωρητή, ενώ οι τελεστές μιας πράξης διαβάζονται στη φάση *ID*. Για το λόγο αυτό ακριβώς δημιουργείται και ο κίνδυνος δεδομένων που προαναφέραμε.

Για την επίλυση του προβλήματος χρησιμοποίειται μια τεχνική, που ονομάζεται ***Forwarding (προώθηση).*** Η τεχνική αυτή υλοποιείται απο το υλικό (hardware). Βασίζεται στην παρατήρηση ότι το αποτέλεσμα της πράξης ADD είναι έτοιμο μετά το πέρας της φάσης EX (της εντολής ADD). Οπότε, αν το αποτέλεσμα, μπορούσε με κάποιο τρόπο

να είναι διαθέσιμο και στην εντολή SUB τη στιγμή που το χρειαζεται, τότε δεν θα ήταν αναγκαίο να εισάγουμε νεκρούς κύκλους για να λυθεί το πρόβλημα.

Η λειτουργία του Forwarding είναι η εξής:

- Το αποτέλεσμα που παράγεται στην ALU, στη φάση EX/MEM, ανατροφοδοτείται απο τον καταχωρητή EX/MEM πίσω στην ALU.
- Αν το κύκλωμα που υλοποιεί το Forwarding ανιχνεύσει ότι κάποια επόμενη εντολή ζητάει το αποτέλεσμα μιας προηγούμενης εντολής, η μονάδα λογικής (Control Logic) επιλέγει το προωθούμενο αποτέλεσμα που εξάγεται απο την ALU παρά αυτό που βρίσκεται στον αντιστοιχο καταχωρητή γενικού σκοπού.



Εικόνα 10. *Η τεχνική της Προώθησης*

(a). Το αποτέλεσμα της ALU στο τέλος της φάσης EX

(b). Το αποτέλεσμα της ALU στο τέλος της φάσης MEM

(c). Το αποτέλεσμα της μνήμης στο τέλος της φάσης MEM.

Τελικά, με τη χρήση της τεχνικής της προώθησης οι εντολές μας θα εκτελούνταν όπως φαίνεται και στον πίνακα που εμφανίζει η παρακάτω εικόνα.

A	B	C	D	E	F	G	H	I
ADD R1, R2, R3	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		

Εικόνα 11. *Εκτέλεση του παραδείγματος με Forwarding*

Χωρίς την τεχνική που περιγράψαμε παραπάνω, οι εντολές από το παράδειγμα μας, θα εκτελούνταν σωστά μόνο με την είσοδο νεκρών κύκλων (stalls). Δηλαδή, η εκτέλεση θα γινόταν κάπως έτσι:

A	B	C	D	E	F	G	H	I	J
ADD R1, R2, R3	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	S	S	EX	MEM	WB	

Εικόνα 12. *Εκτέλεση του παραδείγματος με εισαγωγή νεκρών κύκλων*

Ένας κίνδυνος δημιουργείται όποτε υπάρχει κάποια εξάρτηση ανάμεσα στις εντολές που πρόκειται να εκτελεστούν και έτσι αλλάζει η σειρά της πρόσβασης στους τελεστές. Το προηγούμενο παράδειγμα, δείχνει αυτό ακριβώς το φαινόμενο. Παρόλα αυτά, είναι πολύ πιθανό να δημιουργηθεί μία εξάρτηση εγγραφής/ανάγνωσης της ίδιας θέσης της μνήμης. Στον DLX, οι αναφορές στη μνήμη βρίσκονται πάντα σε τάξη (σε σειρά), εμποδίζοντας έτσι να υπάρξει τέτοιου είδους κίνδυνος.

Όλοι οι τύποι των κινδύνων δεδομένων που περιγράφονται στην παρούσα εργασία αφορούν τους καταχωρητές στη CPU. Κάθε κίνδυνος ανήκει σε μια απο τις παρακάτω τρείς κατηγορίες, ανάλογα με τη σειρά των εγγραφών και των αναγνώσεων των δεδομένων. Για να γίνει πιο κατανοητό, ας θεωρήσουμε δύο εντολές i, j, όπου η i εκτελείται πρίν απο τη j. Έχουμε τους εξής κινδύνους:

- Ανάγνωση πρίν από εγγραφή (Read After Write, RAW).
- Εγγραφή μετά από εγγραφή (Write After Write, WAW).
- Εγγραφή μετά από εγγραφή (Write After Write, WAW).
- * Ανάγνωση μετά την ανάγνωση (Read After Read, RAR).

Γ3.1. Ανάγνωση πρίν από εγγραφή (Read After Write, RAW)

Αυτή την κατηγορία των data hazard, είναι η πιο συχνή στην εμφάνιση. Αντιμετωπίζεται με επιτυχία με την τεχνική της προώθησης. Σε αυτή, η εντολή j προσπαθεί να διαβάσει μια τιμή από κάποιον καταχωρητή χωρίς η i να έχει προηγουμένως εγγράψει σε αυτόν κάποια τιμή. Το προηγούμενο παράδειγμα ανήκει σε αυτήν την κατηγορία.

Γ3.2. Εγγραφή μετά από εγγραφή (Write After Write, WAW)

Όταν η εντολή j προσπαθεί να εγγράψει μια τιμή πάνω στον ίδιο καταχωρητή ή μνήμη όπου γράφει και η εντολή *i,* τότε προκύπτει data hazard που ανήκει σε αυτή την κατηγορία. Αυτός ο κίνδυνος συμβαίνει μόνο όταν η εγγραφή γίνεται σε περισσότερους του ενός κύκλου. Στον υποθετικό επεξεργαστή DLX, αυτο το είδος κινδύνου δεν υπάρχεί καθώς όλες οι εγγραφές διαρκούν έναν κύκλο.

Παράδειγμα

Ας υποθέσουμε οτι η φάση MEM διαρκεί δύο κύκλου ρολογιού. Έστω τώρα, οτι οι εντολές που θέλουμε να εκτελέσουμε είναι:

LD R1,R2(0)

ADD R1, R2, R3

Παρακάτω φαίνεται πως θα εκτελούνταν οι εντολές στην pipeline:

Κύκλοι CPU							
Εντολή	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LD R1,R2(0)	IF	ID	EX	MEM1	MEM2	WB	
ADD R1, R2, R3		IF	ID	EX	WB		

Πίνακας 7. *WAW Hazard*

Αυτό το πρόβλημα, αν δεν αντιμετωπιστεί, θα έχει ώς συνέπεια ότι η τιμή που θα μείνει στον R1 μετά το πέρας της εκτέλεσης θα είναι η τιμή που παίρνει στην πρώτη εγγραφή (LD) και όχι αυτή που πάιρνει από την δεύτερη (ADD).

Γ3.3. Εγγραφή μετά από εγγραφή (Write After Write, WAW)

Σε αυτήν την τελευταία κατηγορία κινδύνων δεδομένων, η εντολή j προσπαθεί να γράψει σε έναν καταχωρητή πριν η εντολή i τον διαβάσει. Έτσι, λανθασμένα, η εντολή i παίρνει τη νέα τιμή.

Αυτό όμως δεν μπορεί να συμβεί στην δική μας περίπτωση της pipeline επειδή όλες οι εγγραφές γίνονται νωρίς (στη φάση ID) και όλες οι εγγραφές αργά (στη φάση της WB). Αυτού του είδους ο κίνδυνος συμβαίνει όταν υπάρχουν κάποιες εντολές, οι οποίες γράφουν τα αποτελέσματα νωρίς και κάποιες άλλες, που διαβάζουν αργά μια πηγή (εναν καταχωρητή).

Λόγω της φυσικής δομής της Pipeline, η ανάγνωση γίνεται πριν απο μία εγγραφή, τέτοιοι κίνδυνοι είναι σπάνιοι. Εαν όμως τροποποιήσουμε την pipeline μας, όπως στο προηγούμενο παράδειγμα, και διαβάσουμε επίσης κάποιους τελεστές αργότερα, τότε ένα τέτοιο έιδος κινδύνου μπορεί να εμφανιστεί.

Παράδειγμα

Έστω οι εντολές που θέλουμε να εκτελέσουμε είναι:

ST R1, R2(0)

ADD R2, R3, R4

Ο πίνακας χρονισμού για τις παραπάνω εντολές είναι:

Κύκλοι CPU						
Εντολή	**1**	**2**	**3**	**4**	**5**	**6**
ST R1, R2(0)	IF	ID	EX	MEM1	MEM1	WB
ADD R2, R3, R4		IF	ID	EX	WB	

Πίνακας 8. *Πίνακας χρονισμού*

Εαν η εντολή ST διαβάζει τον R2 κατα τη διάρκεια της MEM2 φάσης της και η εντολή ADD γράφει στον R2 στο πρώτο μισό της WB φάσης της, τότε η ST θα διαβάσει και αποθηκεύσει (λανθασμένα), την τιμή που παρήγαγε η εντολή ADD.

Γ3.4. Εισαγωγή νεκρών κύκλων

Στην συνέχεια, θα αναφερθούμε στους κινδύνους που απαιτούν νεκρούς κύκλους για να τους αποφύγουμε μέσα απο παραδείγματα.

Στην pipeline του DLX είδαμε ότι υπάρχουν ουσιαστικά μόνο οι κίνδυνοι ανάγνωσης μετά από εγγραφή. Είδαμε επίσης πώς κάποιοι από αυτούς τους κινδύνους αντιμετωπίζονται με τη μέθοδο της προώθησης χωρίς εισαγωγή νεκρών κύκλων. Παρ' όλ' αυτά υπάρχουν ακόμα και στην περίπτωση του DLX κίνδυνοι δεδομένων οι οποίοι απαιτούν εισαγωγή νεκρών κύκλων.

Παράδειγμα

Έστω οι εντολές:

LD R1, R2(3)

SUB R4, R1, R5

Και ο πίνακας χρονισμού:

Κύκλοι CPU							
Εντολή	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LD R1, R2(3)	IF	ID	EX	MEM	WB		
SUB R4, R1, R5		IF	ID	EX	MEM	WB	

Πίνακας 9. *Πίνακας χρονισμού των εντολών*

	A	B	C	D	E	F	G	H	I	J
1	LD R1, R2, 3	IF	ID	EX	MEM	WB				
2	SUB R4, R1, R5		IF	ID	S	S	EX	MEM	WB	

Πίνακας 10. *Πίνακας χρονισμού των εντολών χωρίς Pipeline (εισαγωγή νεκρών κύκλων)*

Αυτή είναι η μοναδική περίπτωση που κινδύνων δεδομένων και συγκεκριμένα RAW (Read After Write), η οποία δεν μπορεί να επιλυθεί με μόνο με το Forwarding. Για αυτόν το λόγο θα χρειαστεί η εισαγωγή ενός νεκρού κύκλου (**stall**) για να εκτελεστεί σωστά.

Το πρόβλημα συμαίνει στη φάση MEM της εντολής LD και στη φάση EX της εντολής SUB. Η εντολή LD δεν έχει τα δεδομένα μέχρι το πέρας της φάσης MEM (στο τέλος του κύκλου 4), ενώ η εντολή SUB χρειάζεται αυτά τα δεδομένα ακριβώς στην αρχή αυτου του κύκλου (κυκλος 4). Έτσι θα χρειαστεί ένας νεκρός κύκλος μέχρι η LD αποκτήσει τα δεδομένα (στη φάση WB τα έχει). Οπότε και με τη βοήθεια του

Forwarding, η εντολή SUB θα πάρει και αυτή τα δεδομένα που χρειάζεται για να γίνει η πράξη (με τα σωστα δεδομένα). Συμπερασματικά, ο νέος πίνακας χρονισμού γίνεται:

A	B	C	D	E	F	G	H	I
LD R1, R2, 3	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	S	EX	MEM	WB	

Πίνακας 10. *Ο νέος πίνακας χρονισμού με τη χρήση Forwarding και νεκρών κύκλων*

Γ4. Κίνδυνοι ελέγχου

Σε αυτήν την ενότητα θα γίνει μια απλή αναφορά για τους ***κινδύνους ελέγχου*** (control hazards) και δεν θα εμβαθύνουμε στο θέμα, καθώς δεν αποτελεί μέρος του Προσομοιωτή των κινδύνων της Pipeline που κατασκευάσαμε.

Οι κίνδυνοι ελέγχου μπορούν να προκαλέσουν μεγαλύτερη μείωση της απόδοσης ενός επεξεργαστή απο όση θα προκαλούσαν οι κίνδυνοι δεδομένων και οι δομικοί κίνδυνοι. Αυτού του είδους οι κίνδυνοι προέρχονται απο την αλλαγή ροής του προγράμματος εξ'αιτίας κάποιου άλματος (με ή χωρίς συνθήκη).

Όταν ένα άλμα εκτελείται, ο PC μπορεί να αλλάξει την τιμή του σε κάτι διαφορετικό απο την τρέχουσα συν τέσσερα (PC + 4). Το άλμα που αλλάζει την τιμή του PC ώστε να δείχνει στην διεύθυνση στόχο, ονομάζεται *Taken Branch*. Αν όμως η συνθήκη δεν ικανοποιείται, τότε το άλμα λέγεται *Non Taken*.

Στην επόμενη σελίδα παραθέτουμε έναν πίνακα οπου φαίνεται η εκτέλεση των φάσεων των διαφόρων εντολών μετά απο μια εντολή ***branch (άλματος)***.

	Κύκλοι CPU									
	1	2	3	4	5	6	7	8	9	10
Branch	IF	ID	EX	MEM	WB					
Επόμενη εντολή		IF(stall)	stall	stall	IF	ID	EX	MEM	WB	
Επόμενη εντολη + 1						IF	ID	EX	MEM	WB

Πίνακας 15. *Εκτέλεση εντολών μετά απο branch*

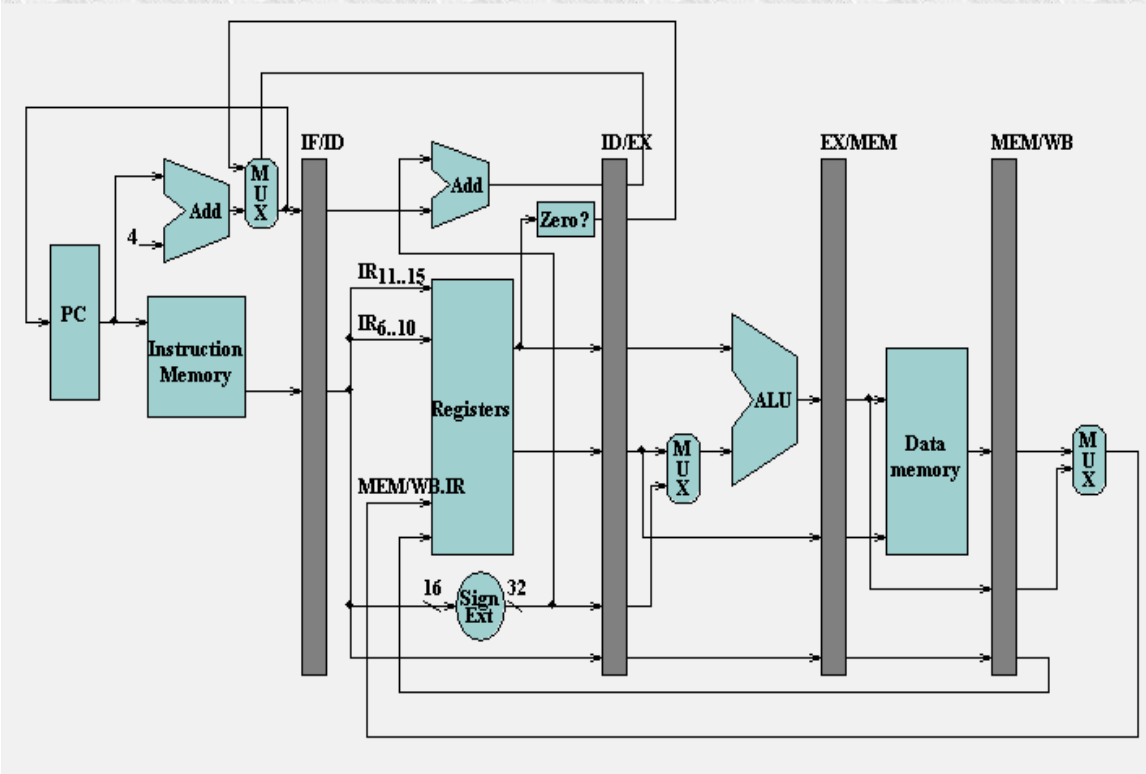
Σύμφωνα με τον αρχικό σχεδιασμό της pipeline, κατά την εκτέλεση ενός άλματος υπό συνθήκη γίνεται γνωστή η διεύθυνση της επόμενης εντολής αφού ελεγχθεί πρώτα η συνθήκη, δηλαδή στη φάση MEM. Αν η συνθήκη πετύχει τότε η διεύθυνση της επόμενης εντολής είναι η τιμή του ALUOutput διαφορετικά, η διεύθυνση της επόμενης εντολής είναι η τιμή του NPC. Αλλά και στην περίπτωση του άλματος χωρίς συνθήκη, η νέα τιμή του PC δίνεται στη φάση MEM. Και στις περιπτώσεις χάνονται τρείς κύκλοι μηχανής διότι η επόμενη εντολή πρέπει να ξαναρχίσει μετά απο την ολοκλήρωση της

φάσης MEM του άλματος. Τρείς νεκροί κύκλοι που χάνονται για κάθε άλμα είναι σημαντική απώλεια. Με μια συχνότητα περίπου 30% σε εντολέ - άλματα και ένα ιδανικό ***CPI=1 (Clocks per Instuction)***, ο μηχανισμός των νεκρών κύκλων των αλμάτων επιτυγχάνει μόνο τη μισή επιτάχυνση απο την Pipeline.

Η τεχνική που θα μπορούσαμε να χρησιμοποιήσουμε για να να μειώσουμε τον αριθμό των χαμένων κύκλων της pipeline λόγω των αλμάτων είναι εκτέλεση των ακόλουθων βημάτων:

1. Να βρούμε αν το άλμα θα γίνει ή όχι νωρίτερα από τη φάση MEM
2. Να υπολογίζουμε τη διεύθυνση του στόχου του άλματος νωρίτερα από τη φάση MEM

Αυτό επιτυγχάνεται αν παρατηρήσουμε ότι στα άλματα με συνθήκη η σύγκριση γίνεται μεταξύ ενός καταχωρητή γενικού σκοπού και του μηδενός (zero test). Οι καταχωρητές διαβάζονται κατά τη φάση ID οπότε η σύγκριση μπορεί να γίνει στη φάση αυτή. Επίσης, παρατηρούμε ότι ο υπολογισμός του στόχου του άλματος μπορεί να γίνει κατά τη φάση αποκωδικοποίησης, απλά, αν προσθέσουμε στον PC το offset. Θα πρέπει λοιπόν να μεταφέρουμε το κύκλωμα σύγκρισης με το μηδέν στη φάση ID και να προσθέσουμε μια ακόμα αριθμητική μονάδα στη φάση αυτή για τον υπολογισμό του αθροίσματος PC+Offset.



Εικόνα 12. *Η βελτιωμένη Pipeline. Τώρα ο στόχος του άλματος είναι γνωστός στη φάση ID*

Έτσι, μετα μετά τη βελτίωση της Pipeline, ο πίνακας χρονισμού γίνεται:

	Κύκλοι CPU									
	1	2	3	4	5	6	7	8	9	10
Branch	IF	ID	EX	MEM	WB					
Επόμενη εντολή		**IF**	IF	ID	EX	MEM	WB			
Επόμενη εντολη + 1				IF	ID	EX	MEM	WB		

Πίνακας 16. *Ο νέος πίνακας χρονισμού*

Μετά από μια εντολή άλματος, η επόμενη κάνει ένα κύκλο παραπάνω. Ο λόγος είναι ότι κατά τον κύκλο 2 εκτελέστηκε η φάση IF μιας λάθος εντολής. Κατά τον κύκλο 3 έχει ήδη ολοκληρωθεί η φάση ID του άλματος και επομένως γνωρίζουμε ποια είναι η σωστή επόμενη εντολή. Στον κύκλο 3 αρχίζει η εκτέλεση αυτής της εντολής.

Ακόμα και με αυτή τη βελτίωση υπάρχουν πράγματα που μπορούν να γίνουν για να βελτιωθεί η επίδοση της pipeline ακόμα περισσότερο. Υπάρχουν 4 διαφορετικές μέθοδοι διαχείρισης των αλμάτων με διαφορετικούς βαθμούς πολυπλοκότητας και διαφορετικές επιδόσεις ταχύτητας. Αυτές απλώς αναφέρονται παρακάτω χωρίς να γίνει εκτενής ανάλυση τους.

Α. Εισαγωγή πάντα ενός κύκλου καθυστέρησης
Β. Πρόβλεψη ότι το άλμα δεν θα γίνει (predict not taken)
Γ. Πρόβλεψη ότι το άλμα θα γίνει (predict taken)
Δ. Καθυστερημένο άλμα (delayed branch)

Δ. ΠΡΟΣΟΜΟΙΩΤΗΣ ΤΩΝ PIPELINE HAZARDS

Δ1. Μελέτη σκοπιμότητας – Απαιτήσεις συστήματος

Ο προσομοιωτής των Pipeline Hazards είναι ένα πρόγραμα , το οποίο θα δέχεται εντολές γλώσσας μηχανής (π.χ. LD R1,R2(9)), θα τις τοποθετεί σε έναν πίνακα, τον πίνακα που θα προσομοιώνει την εκτέλεση των εντολών με τη χρήση της τεχνικής pipeline και θα ανακαλύπτει αν υπάρχουν κίνδυνοι κατά την εκτέλεση των εντολών.

Στην περίπτωση που υπάρχουν, ο προσομοιωτής έχει τη δυνατότητα να τους αποφεύγει αυτούς τους κινδύνους με τη χρήση κάποιων μεθόδων. Οι μέθοδοι αυτοί μπορεί να είναι είτε η εισαγωγή νεκρών κύκλων είτε η τεχνική της προώθησης (Forwarding).

Για την σωστή λειτουργία της εφαρμογής πρέπει να επισημάνουμε κάποιες παραμέτρους, τόσο απο πλευράς Λογισμικού (Software) όσο και απο πλευράς Υλικού (Hardware) του υπολογιστή (Η/Υ).

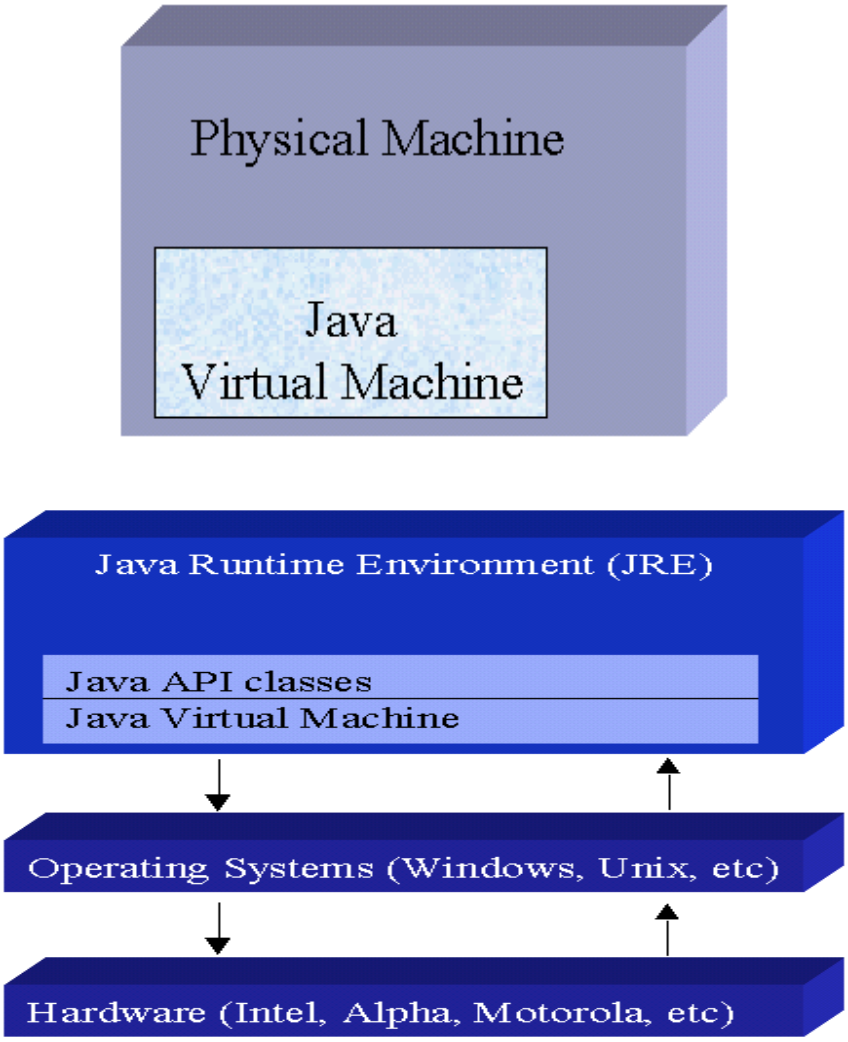
Οι ελάχιστες απαιτήσεις σε hardware είναι οι αρχικές εκδόσεις του επεξεργαστή Pentium III με συχνότητα τουλάχιστον στα 800MHz και μνήμη RAM, καθώς και οποιουδήποτε άλλου επεξεργαστή διαφορετικής αρχιτεκτονικής (Aplha, Sun Sparc, Motorola, κλπ).

Οι αντίστοιχες απαιτήσεις σε software είναι Λειτουργικό Σύστημα Windows ’98 και ανώ (αν προκειται για πλατφόρμα Windows) και οποιαδήποτε διανομή Linux και γενικότερα Λειτυργικού Συστήματος που βασίζεται στο Unix (Solaris OS, FreeBSD, κ.λ.π.).

Το πλεονέκτημα της εφαρμογής αυτής, καθώς και γενικά οπιασδήποτε εφαρμογής είναι γραμμένη σε Java, είναι οτι είναι ανεξάρτητη απο την πλατφόρμα του υπολογιστή που θα τρέξει το πρόγραμμα. Το μόνο που χρειάζεται είναι το αντίστοιχο για κάθε πλατφόρμα Java Virtual Machine (JVM). Το JVM είναι ένα εργαλείο, το οποίο διερμηνεύει τον bytecode που εχεί δημιουργήσει ο Java Compiler σε γλώσσα μηχανής κατανοητή απο την υπάρχουσα πλατφόρμα.

Συμπερασματικά λοιπόν, μπορούμε να πούμε οτι ο προσομοιωτής των Pipeline Hazards που υλοποιήσαμε, μπορεί να τρέξει κάτω σχεδόν απο οποιδήποτε

υπολογιστικο σύστημα, εφόσον υπάρχει ο JVM που αντιστοιχεί στο υπολογιστικό μας σύστημα.



Εικόνα 13. *Java Virtual Machine*

Δ2 Ανάλυση – Σχεδιασμός

Ανάλυση του συστήματος

Η κατασκεύη του προσομοιωτή Pipeline Hazards έγινε με τη χρήση της γώσσας προγραμματισμού Java 2. Στην ανάλυση και το σχεδιασμό του προγράμματος χρησιμοποιήθηκε το εργαλείο ArgoUML-0.24, το οποίο είναι open source πρόγραμμα.
Για συγγραφή του κώδικα χρεισιμοποιήθηκε εξ'ολοκλήρου το Netbeans IDE. Η ανάλυση έγινε με την γλώσσα μοντελοποίησης UML.

Τα UML διαγράμματα που δημιουργούνται στη φάση της ανάλυσης εστιάζουν στην περιοχή του προβλήματος και όχι σε μια συγκεκριμένη τεχνική λύση.

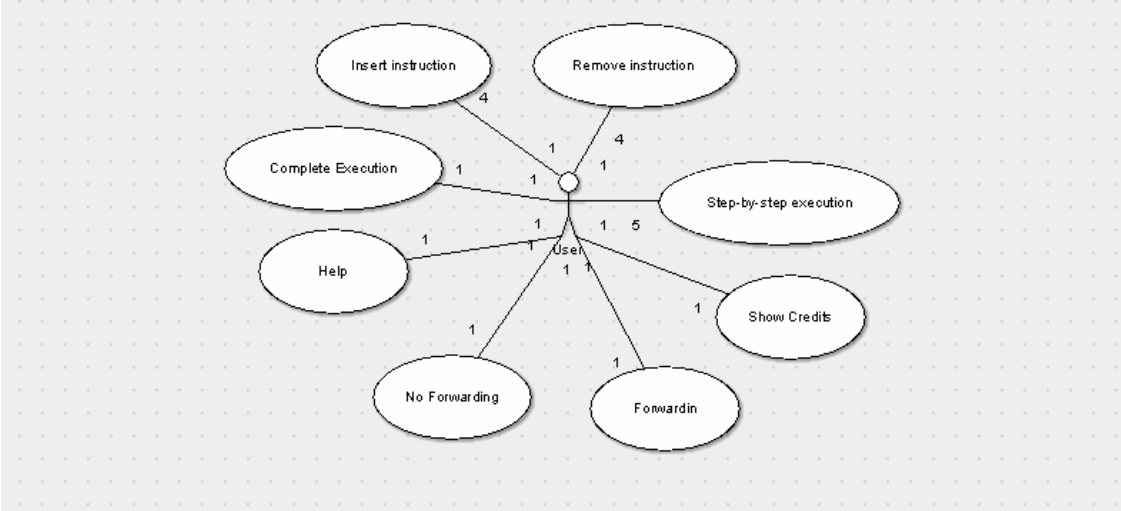
Γενικά, η κατασκευή των διαγραμμάτων δεν είναι μια σειριακή διαδικασία δημιουργίας του ενός διαγράμματος μετά το άλλο. Τα διαγράμματα γίνονται παράλληλα και λεπτομέρειες προστίθενται σε συνεχόμενες επαναλήψεις. Εκτός και αν έχει επιβληθεί μια αυστηρή διαδικασία ανάπτυξης στους developers, το ποια διαγράμματα πρέπει να γίνουν είναι κυρίως προσωπική απόφαση του αναλυτή.

Πρέπει να κάνουμε σαφές το γεγονός οτι η ανάλυση και ο σχεδιασμός ενός συστήματος είναι καθαρά υποκειμενικος όπως προαναφέραμε. Στην περίπτωση μας δημιουργήθηκαν τρία διαγράμματα. Αυτά είναι το δίαγραμμα περιπτώσεων χρήσης, το διάγραμμα τάξεων και τέλος το διάγραμμα πακέτων.

Περιπτώσεις Χρήσης – Use Cases

Αρχικά λοιπόν, χρησιμοποιήσαμε την μέθοδο των Περιπτώσεων Χρήσης (Use Cases). Η μέθοδος αυτή είναι ένα ενδιαφέρον φαινόμενο και τόσο σημαντική ώστε να παίζει πρωταρχικό ρόλο στην ανάπτυξη και το σχεδιασμό ενός έργου (*I.Jacobson, εφευρέτης των περιπτώσεων χρήσης*).

Έτσι, δημιουργήθηκε το ομώνυμο διάγραμμα που σε μεγάλο βαθμό εξηγεί τι θα κάνει το σύστημα. Το σύνολο των περιπτώσεων χρήσης είναι η εξωτερική εικόνα του συστήματος. Παρακάτω φαίνεται το διάγραμμα Περιπτώσεων Χρήσης για τον PHS.



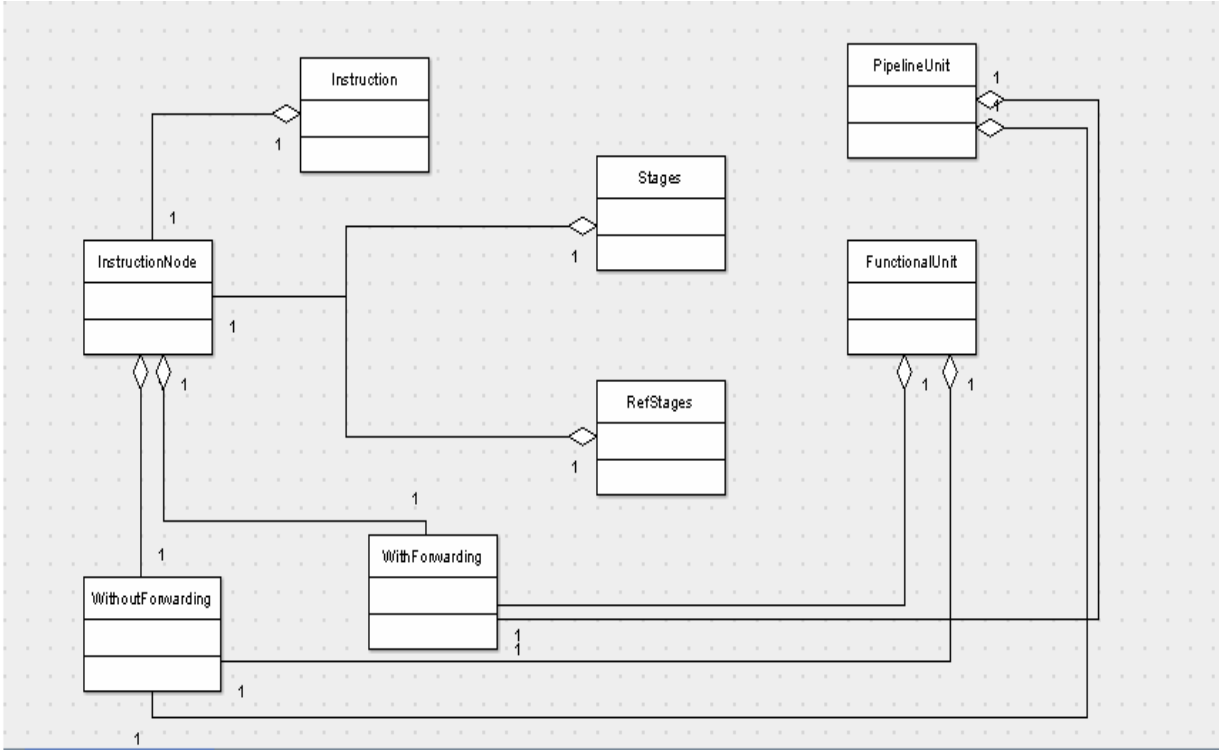
Εικόνα 14. *Διάγραμμα Περιπτώσεων Χρήσης (Use Case Diagram)*

Όπως μπορούμε να παρατηρήσουμε και απο το διάγραμμα των περιπτώσεων χρήσης, ο χρήστης του PHS θα μπορεί να εισάγει απο μια έως τέσσερεις εντολές (Insert instruction) καθώς επίσης και να τις αφαιρεί (Remove instruction). Η επεξεργασία αυτών θα γίνεται είτε βήμα προς βήμα είτε ολοκληρωτικά με μία μόνο κίνηση. Επιπλέον, βλεπουμε οτι υπάρχει η δυνατότητα ο χρήστης να επιλέξει την τεχνική αποφυγής των pipeline hazards, δηλαδή το forwarding (with or without forwarding). Τέλος, θα υπάρχει και μια αναφορά στους συντελεστές αυτού του προγράμματος.

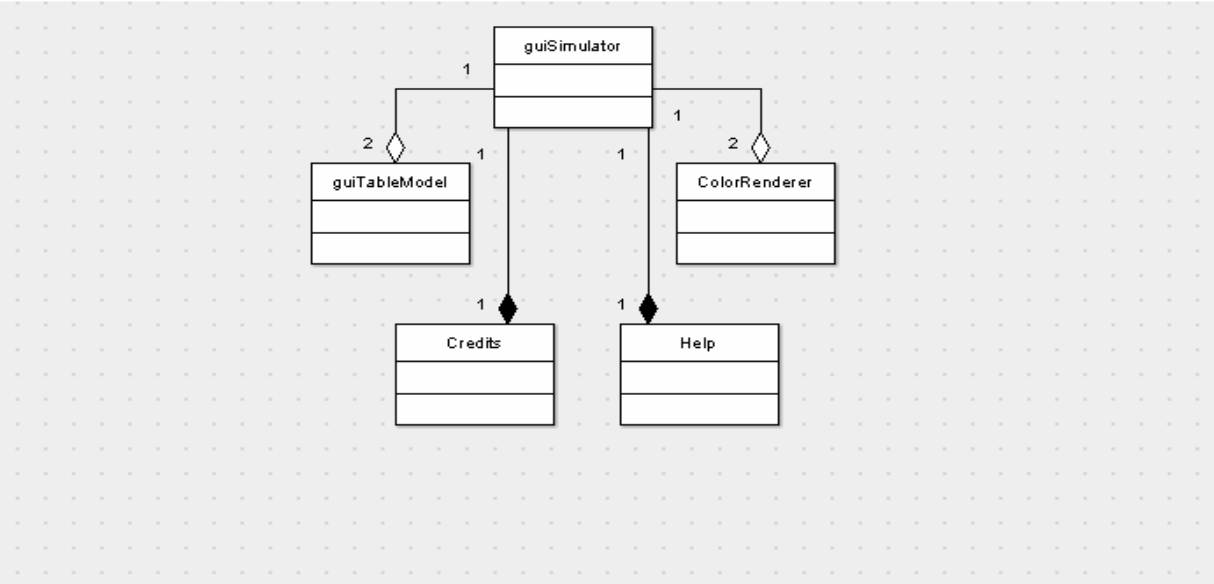
Διαγράμματα Τάξεων – Class Diagrams

Στη συνέχεια της ανάλυσης και του σχεδιασμου του προσομοιωτή, δημιουργήθηκε το Διάγραμμα Τάξεων. Σε αυτό το σημείο πρέπει να αναφέρουμε οτι οι τάξεις διαιρεθηκαν σε δύο μέρη, ανάλογα με το τι αποσκοπούν να κάνουν. Τα δύο αυτά μέρη είναι ο αλγόριθμος και το γραφικο περιβάλλον. Οι τάξεις αυτές ομαδοποιήθηκαν σε μονάδες υψηλότερων επιπέδων, οι οποίες είναι γνωστές στην UML σαν πακετα. Επομένως, οι μεν τάξεις του αλγόριθμου ομοδοποιήθηκαν και δημιούργησαν το πακέτο *algor* και οι δε του γραφικού περιβάλλοντος δημιούργησαν το πακέτο *GUI*.

Το Διάγραμμα Τάξεων, δίνει μια αφηρημένη περιγραφή των τάξεων που χρησιμοποιήθηκαν για την κατασκευή του συστήματος μας. Παρακάτω φαίνεται πως είναι το διάγραμμα του μοντέλου αυτού.



Εικόνα 15. *Το Διάγραμμα Τάξεων – Οι τάξεις του αλγορίθμου*



Εικόνα 16. *Το Διάγραμμα Τάξεων – Οι τάξεις του γραφικού περιβάλλοντος*

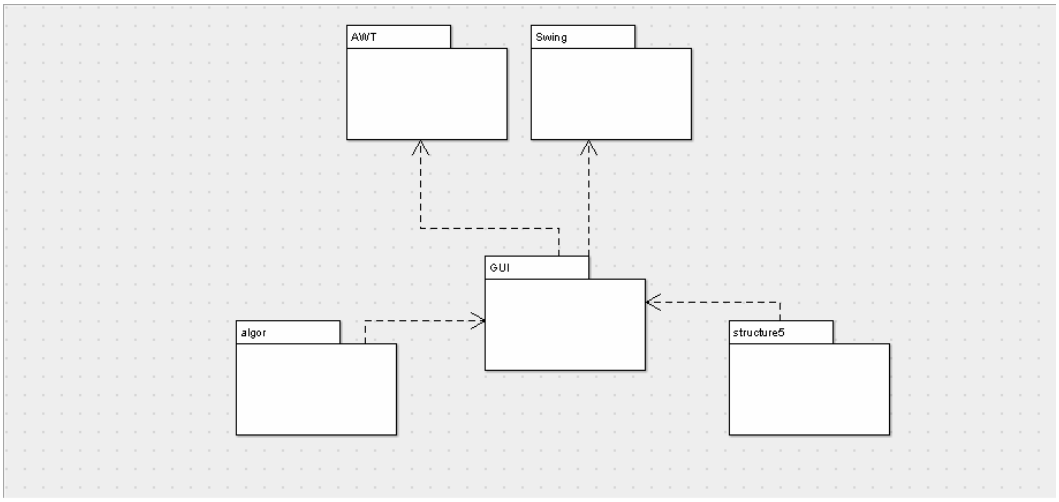
Τα διαγράμματα Ταξεων, περιγράφουν τους τύπους των αντικειμένων στο σύστημα και τα διάφορα είδη στατικών σχέσεων που υπάρχουν μεταξύ τους. Οι στατικές σχέσεις που μπορούν να προκύψουν είναι οι συσχετίσεις (associations) και οι υποτύποι (ή δευτερεύοντες τύποι). Τα διαγράμματα Τάξεων δείχνουν επίσης τις ιδιότητες και τις λειτουργίες μιας Τάξης αλλά και τους περιορισμούς που υπάρχουν στον τρόπο με τον οποίον συνδέονται τα αντικείμενα.

Διαγράμματα Πακετων – Package Diagrams

Μια απο τις παλαιότερες ερωτήσεις στις μεθόδους ανάπτυξης λογισμικού είναι: πώς διαιρει κανείς ένα μεγάλο σύστημα σε μικρότερα; Η απάντηση είναι η ομαδοποίηση των τάξεων σε μονάδες υψηότερων επιπέδων. Στη UML, ο μηχανισμός ομαδοποίησης ονομάζεται πακέτο (package).

Η ιδέα των πακέτων μπορεί να εφαρμοστεί σε οποιοδήποτε στοιχείο μοντελοποίησης και όχι μόνο στις τάξεις. Η ομοδοποίηση είναι αυθέρετη. Το στοιχείο που είναι πιο χρήσιμο και αυτό που τονίζεται με περισσότερη έμφαση και στη UML, είναι η εξάρτηση (dependency).

Έτσι λοιπόν, τα διαγράμματα πακέτων αποικονίζουν πακέτα τάξεων και τις εξαρτήσεις μεταξύ τους. Στην παρακάτω εικόνα μπορούμε να παρατηρήσουμε το διάγραμμα πακέτων που αφορά την εφαρμογή μας.



Εικόνα 17. Το *Διάγραμμα Πακέτων (Packets Diagram)*

Παρατηρούμε οτι υπάρχουν πέντε πακέτα, τα πακετα AWT και Swing που τα εισάγουμε απο τη γλώσσα προγραμμτισμου Java 2 και τα παλέτα algor, GUI και structure5.

Τα πακέτα AWT και Swing, περιλαμβάνουν όλες εκείνες τις κλάσεις και τις διεπαφές ώστε να κατασκευάσουμε το γραφικό περιβάλλον της εφαρμογής μας και να του δώσουμε ζωή (events).

Το πακέτο algor, όπως έχει ήδη αναφερθέι, περιλαμβάνει τις κλάσεις που αφορούν τον αλγόριθμο, δηλαδή τις κλάσεις Instruction, Stages, RefStages, FunctionalUnit, PipelineUnit, InstructionNode, WithoutForwarding και WithForwarding.

Το πακέτο GUI, περιλαμβάνει τις κλάσεις που αφορούν το γραφικό περιβάλλον της εφαρμογής μας. Οι τάξεις που βρίσκονται μέσα σε αυτό είναι: guiSimulator, ColorRenderer1, Credits, InstructionCode και guiTableModel.

Τέλος, το πακέτο structure5 αποτετελεί μια συλλογή απο κλάσεις, οι οποίες περιγράφουν δομές δεδομένων. Το πακέτο αυτό βοήθησε στην κατασκευή δομών δεδομένων για το πρόγραμμά μας.

Όσον αφορά τον σχεδιασμό και την υλοποίηση του συστήματος μας, παραθέτουμε τους ακόλουθους πίνακες που δείχνουν το σχεδιασμό των τάξεων.

Τάξεις που αφορούν τον αλγόριθμο

Instruction	type: String destination: String source1: String source2: String numberOfCycles: int
	Instruction(String, String, String, String, int)
	getType(), getDestination(), getSource1(), getSource2(), getNumberOfCycles() setType(String), setDestination(String), setSource1(String), setSource2(String), setNumberOfCycles(int)

Stages	iFetch: int decode: int exeStart: int exeEnd: int wBack: int
	getIFecth(), getDecode(), getExeStart(), getExeEnd(),getWBack(), getSatgeCycle() setIFecth(int), setDecode(int), setExeStart(int), setExeEnd(int), setWBack(int)

RefStages	iFetch[]:int i: int decode[]:int d:int execEnd[]:int e:int memory[]:int m:int wBack[]:int w:int structural[]:int s:int max:int
	RefStages()
	addIFetchDept(int), addDecodeDept(int), addExecEndDept(int), addMemoryDept(int), addWBackDept(int), getIFetch(), setIFetch(int),getDecode(), setDecode(int),getExecEnd(), setExecEnd(int),getMemory(), setMemory(int),getWBack(), setWBack(int),getStructural(), setStructural(int),getMax(), setMax(int),getI(), setI(int),getD(), setD(int),getE(), setE(int),getM(), setM(int),getW(), setW(int),getS(), setS(int)

InstructionNode	number:int instruction:Instruction indegree:int stages:Stages refNode:RefStages refStages:RefStages
	InstructionNode(Instruction,int)
	getNumber(), setNumber(int),getInstruction(), setInstruction(Instruction),getIndegree(), setIndegree(int),getStages(), setStages(Stages),getRefNode(), setRefNode(RefStages),getRefStages(), setRefStages(RefStages)

PipelineUnit	load:int add:int multiply:int divide:int
	PipelineUnit()
	getValue(String), getLoad(), setLoad(int),getAdd(), setAdd(int),getMultiply(), setMultiply(int),getDivide(), setDivide(int)

FunctionalUnit	integer:int add:int multiply:int divide:int
	FunctionalUnit(int, int, int,int)
	getNoUnit(String), getInteger(), setInteger(int), getAdd(), setAdd(int), getMultiply(), setMultiply(int), getDivide(), steDivide(int)

WithoutForwarding	numInstruction:int instrNode[]:InstructionNode count:int maxCycle:int totalCycle:int pipeline:int pUnit:PipelineUnit graph: GraphListDirected fUnit: FunctionalUnit timeChart[][]:String
	WithoutForwarding(int, PipelineUnit, FunctionalUnit)
	addInstruction(InstructionNode), generatedHazardsGraph(), topologicalTraversal(), findNodeZeroIndegree(), getNumInstruction(), getInstrNode(), getCount(), getMax(), getMaxCycle(), getTotalCycle(), getPipeline(), getPUnit(), getGraph(), getFUnit(), getTimeChart(), setGraph(GraphListDirected), setInstrNode(InstructionNode), setNumInstruction(int), setTotalCycle(int)

WithForawarding	numInstruction:int instrNode[]:InstructionNode count:int maxCycle:int totalCycle:int pipeline:int pUnit:PipelineUnit graph: GraphListDirected fUnit: FunctionalUnit timeChart[][]:String
	WithoutForwarding(int, PipelineUnit, FunctionalUnit)
	addInstruction(InstructionNode), generatedHazardsGraph(), topologicalTraversal(), findNodeZeroIndegree(), getNumInstruction(), getInstrNode(), getCount(), getMax(), getMaxCycle(), getTotalCycle(), getPipeline(), getPUnit(), getGraph(), getFUnit(), getTimeChart(), setGraph(GraphListDirected), setInstrNode(InstructionNode), setNumInstruction(int), setTotalCycle(int)

Τάξεις που αφορούν το γραφικό περιβάλλον

ColorRenderer1	selectedBorder:Border
	getTableCellRendererComponent(JTable, Object, boolean, boolean,int, int)

guiTableModel	columnName[]:String colSize:int data[][]:Object
	setData(int), setRowCol(String, String), getRowCount(), getColumnCount(), getColumnNames(int), getValueAt(int, int), getColumnClass(int), setValueAt(object, int, int)

Credits	label1:JLabel label2:JLabel label3:JLabel label4:JLabel label5:JLabel label6:JLabel QUIT:String applet:guiSimulator
	credits(guiSimulator)
	main(String), actionPerformed(ActionEvent)

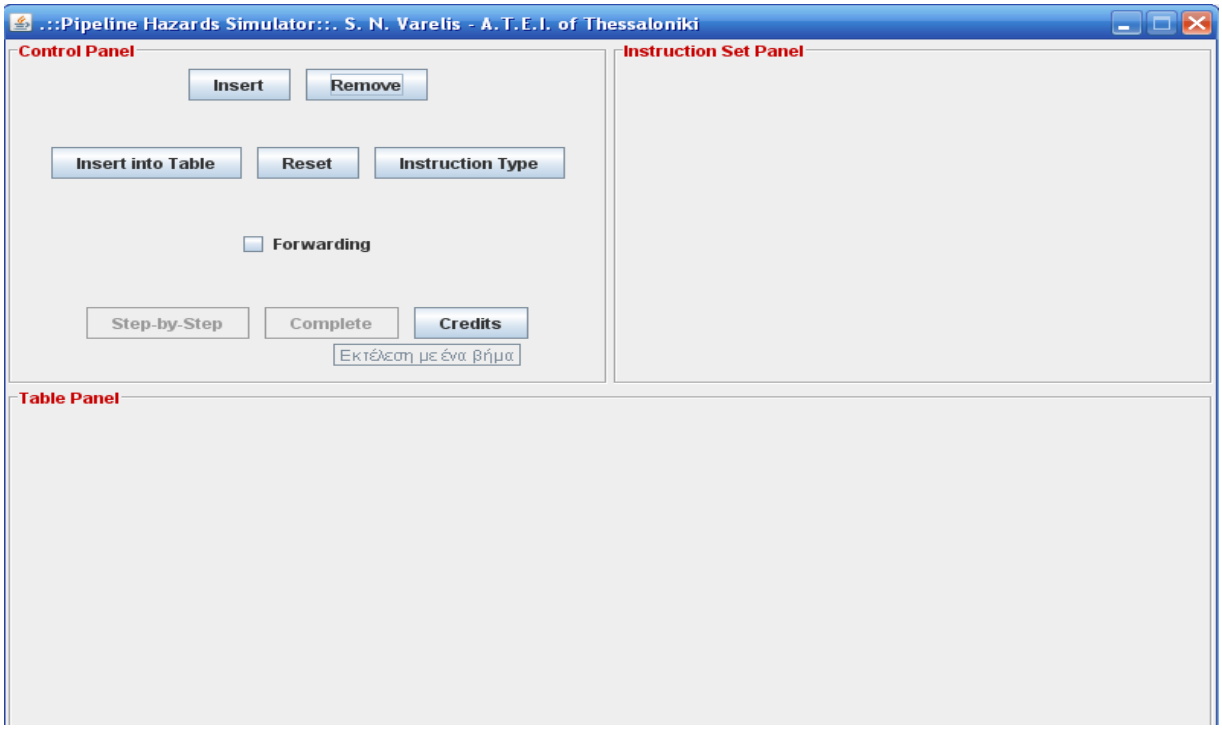
guiSimulator	numMaxInstr:int numInstr:int int cycle String instrSet int totalCycle int numColumn int current_numInstr WithForwarding withforwardingInstrGraph WithoutForwarding withoutforwardingInstrGraph InstructionNode withforwardingInstrNode InstructionNode withoutforwardingInstrNode String instrCode[] String destReg[] String srcReg[] String offset[] String registerInt[] String numberCycles[] JComboBox instr[] JComboBox dest[]

	JComboBox src1[] JComboBox src2[] JButton insertButton JButton removeButton JButton execute JButton reset JCheckBox withForwarding JButton stepExecution JButton finalExecution JButton credits JButton exitButton JTable withForwardingTable JTable withoutForwardingTable JScrollPane sp1, sp2 guiTableModel tm1, tm2 instrPanel tablePanel cpuPanel hazardsPanel JLabel hazardsLabel JLabel cpuCyclesLabel JLabel instrLabel Container c
	guiSimulator()
	actionPerformed(ActionEvent), itemStateChanged(ItemEvent),getNumInstrCycle(String), pressReset(), pressExecute(), executeWithForwarding(PipelineUnit, FunctionalUnit), executeWithouForwarding(PipelineUnit, FunctionalUnit), main(String)

InstructionCode	String QUIT Container c guiSimulator gui JPanel InstrPanel, InstrPanel2, InstrPanel3 JLabel l, label, panelLabel JButton okButton
	InstructionCode(guiSimulator)
	actionPerformed(ActionEvent), assemblyCode(), createImageIcon(String, String), main(String)

Ε3. Εγχειρίδιο χρήσης

Με την έναρξη της εφαρμογής ***Pipeline Hazards Simulator*** (PHS), εμφανίζεται η ακόλουθη φόρμα.

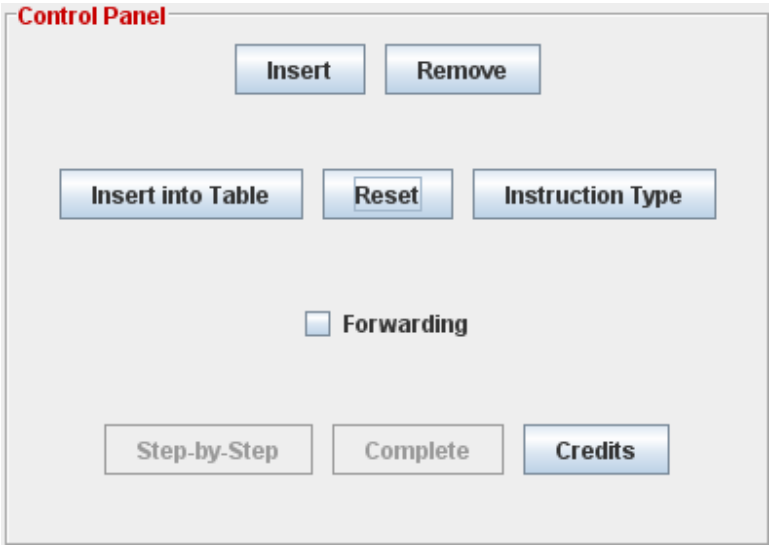


Εικόνα 18. *Το γραφικό περιβάλλον του PHS*

Με την παραπάνω φόρμα μπορούμε να πάρουμε διάφορες πληροφορίες για τις λειτουργίες του προσομοιωτή. Χωρίζεται σε τρία panels, τα οποία έχουν τις ονομασίες Instruction Panel, Control Panel και Table Panel αντίστοιχα. Κάθε ένα απο τα panel αυτά εξυπηρετεί το δικό του σκοπό. Ας τα δούμε ένα ένα ξεχωριστά.

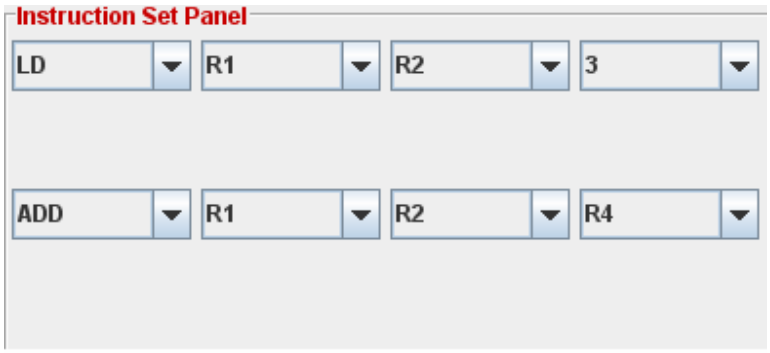
Το Control Panel, περιλαμβάνει τα κουμπιά για την εισαγωγή και αφαίρεση μιας εντολής καθώς και την εισαγωγή αυτής στον πίνακα της pipeline. Επιπλέον, σε αυτό το panel ευρίσκονται τα κουμπιά εκτέλεσης των εντολών, ένα κουμπί που εμφανίζει τη μορφή των εντολών γλώσσας μηχανής, ένα κουμπί που εμφανίζει πληροφορίες για την εφαρμογή και ένα πλαίσιο ελέγχου (checkbox), το οποίο δίνει την επιλογή στον χρήστη ώστε να επιλέξει ή να αποεπιλέξει την μέθοδο παράκαμψης των κινδύνων της Pipeline,

την Προώθηση. Όπως μπορει να γίνει αντιληπτό, πρόκειται για ένα panel που δίνει τον έλεγχο στον χρήστη για το τι θέλει να κάνει.



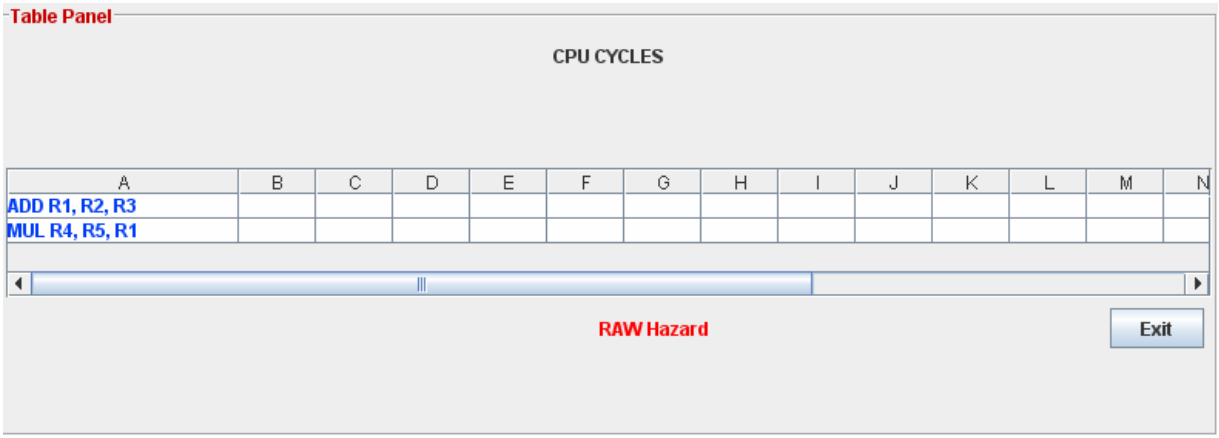
Εικόνα 19. *Control Panel*

Το Instruction Panel, δεν περιλαμβάνει αρχικά τίποτε. Όταν ο χρήστης είσάγει κάποια εντολή, πατώντας το πλήκτρο “Insert”, τότε εμφανίζονται τέσσερα πτυσσόμενα πλαίσια (comboboxes), που μπορεί ο χρήστης να διατυπώση την εντολή που θέλει για να εκτελεστεί. Μπορούν να υπάρχουν μέχρι το πολύ τέσσερεις γραμμές απο τέτοια πτυσσόμενα πλαίσια (όσες και οι μέγιστες επιτρεπόμενες εντολες). Αυτό το panel μπορούμε να πούμε οτι αφορά μόνο τη σύνταξη της προς εκτέλεση εντολής απο το χρήστη.



Εικόνα 20. *Instruction Panel*

Τέλος, το Table Panel, αφορά τον πίνακα που θα εισάγονται οι εντολές και θα εκτελούνται με την τεχνική της Pipeline. Ο πίνακας αυτός, κατα την εκκίνηση της εφαρμογής δεν υπάρχει. Απο τη στιγμή όμως, που ο χρήστης θα πατήσει το κουμπί “Instert into Table”, τοτε θα δημιουργηθεί δυναμικά, με την προσθήκη της ή των εντολών. Αν υπάρχουν κίνδυνοι (hazards), τότε στον πίνακα αυτόν θα φαίνεται ο τρόπος αποφυγής τους. Επιπλέον, στο panel υπάρχει και μία ετικέτα που αναγράφει το είδος του pipeline hazard που προέκυψε (αν προέκυψε).

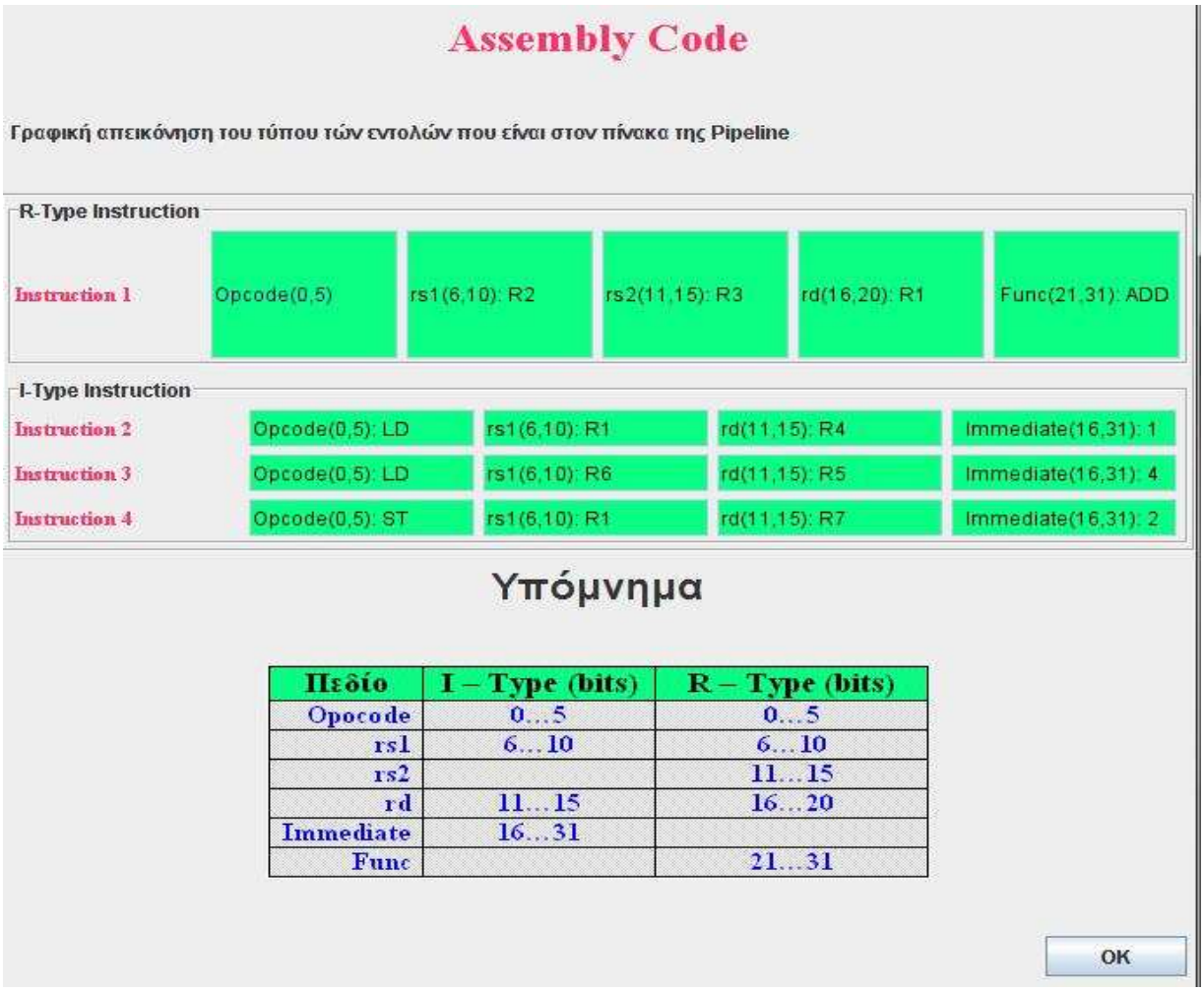


Εικόνα 21. *Table Panel*

Για να εισάγει ο χρήστης μια εντολή στον πίνακα της Pipeline, θα πρέπει να πατήσει το κουμπί “Instert” που βρίσκεται στο Instruction Panel. Αυτομάτως, εμφανίζονται τέσσερα πτυσσόμενα πλαίσια απο τα οποία ο χρήστης θα συντάξει την εντολή. Αυτό μπορεί να γίνει έως και το πολύ τέσσερεις φορές (μέγιστος επιτρεπόμενος αριθμός εντολών).

Στην συνέχεια, με το πάτημα του “Instert into Table” πλήκτρου, η/οι εντολή/ές θα ειισάγωνται στον πίνακα της Pipeline. Αφού ο χρήστης έχει συντάξει τις εντολές που θέλει να εκτελέσει, εχει τη δνατότητα με το πάτημα του πλήκτρου “Instruction Type” να δει με γραφικό τρόπο τον τύπο της κάθε εντολής μαζί με τα αντίστοιχα πεδία. Τέλος, για να αρχίσει η εκτέλεση του προσομοιωτή, ο χρήστης θα πρέπει να πατήσει ένα απο τα κουμπιά “Step-by-Step” ή “Complete”. Αν πατηθεί το κουμπί “Step-by-Step” η εκτέλεση θα γίνεται βήμα-βήμα. Αν όμως πατηθεί το “Complete” η εκτέλεση θα γίνει κατευθείαν,

περνώντας απο όλα τα στάδια και εμφανίζοντας τους κινδύνους (αν υπάρχουν) καθώς και τους τρόπους για την αποφυγή τους. Σε αυτό το σημείο, πρέπει να προσθέσουμε οτι ο χρήστης έχει την επιλογή, η εκτέλεση να γίνει με τη χρήση της μεθόδου του Forwarding ή χωρίς αυτήν.



Εικόνα . *Με το πάτημα του "Instruction Type εμφανίζεται ο τύπος των εντολών*

Ε. ΠΑΡΑΡΤΗΜΑ

Ε1. Τεκμηρίωση του προγράμματος

Η κατασκεύη του προσομοιωτή Pipeline Hazards έγινε με τη χρήση της γώσσας προγραμματισμού Java 2. Στην ανάλυση και το σχεδιασμό του προγράμματος χρησιμοποιήθηκε το εργαλείο ArgoUML-0.24, το οποίο είναι open source πρόγραμμα. Για συγγραφή του κώδικα χρεισιμοποιήθηκε εξ'ολοκλήρου το Netbeans IDE. Η ανάλυση έγινε με την γλώσσα μοντελοποίησης UML.

Στην ενότητα αυτή, παρουσιάζουμε την τεκμηρίωση του προγράμματος μας που υλοποιήσαμε.

Η τεκμηρίωση (Documentation), περιγράφει και κατα συνέπεια δίνει μια πλήρη εικόνα στον προγραμματιστή και κυριώς στον συντηρητή της εφαρμογής (αν δεν είναι ο ίδιος ο προγραμματιστης που την εφτιαξε) ώστε να κατανοήση καλυτερα το πρόγραμμα και σε περίπτωση μελλοντικής επέμβασης να γνωρίζει τις πιθανές κινήσεις του χωρίς να χαθεί στην ποικιλία των κλάσεων και τον μεθόδων.

Παρακάτω ακολουθεί η σχηματική μορφή της τεκμηρίωσης όπως την εξήγαγε το πρόγραμμα NetBeans IDE 6.0.

ΠΑΚΕΤΑ ΕΦΑΡΜΟΓΉΣ

Packages	
algor	
GUI	
structure5	

ΙΕΡΑΡΧΙΑ ΟΛΩΝ ΤΩΝ ΠΑΚΕΤΩΝ ΚΑΙ ΤΩΝ ΤΑΞΕΩΝ

Package Hierarchies:

algor, GUI, structure5

Class Hierarchy

- java.lang.Object
  - structure5.**AbstractIterator**<E> (implements java.util.Enumeration<E>, java.lang.Iterable<T>, java.util.Iterator<E>)
    - structure5.**AbstractListIterator**<E> (implements java.util.ListIterator<E>)
    - structure5.**ArrayIterator**<E>
    - structure5.**DoublyLinkedListIterator**<E>
  - structure5.**AbstractMap**<K,V> (implements structure5.Map<K,V>)
    - structure5.**ChainedHashtable**<K,V> (implements java.lang.Iterable<T>, structure5.Map<K,V>)
    - structure5.**Table**<K,V> (implements structure5.OrderedMap<K,V>)
  - structure5.**AbstractStructure**<E> (implements structure5.Structure<E>)
    - structure5.**AbstractLinear**<E> (implements structure5.Linear<E>)
      - structure5.**AbstractQueue**<E> (implements structure5.Queue<E>)
        - structure5.**QueueArray**<E> (implements structure5.Queue<E>)
        - structure5.**QueueList**<E> (implements structure5.Queue<E>)
        - structure5.**QueueVector**<E> (implements structure5.Queue<E>)
      - structure5.**AbstractStack**<E> (implements structure5.Stack<E>)
        - structure5.**StackArray**<E> (implements structure5.Stack<E>)
        - structure5.**StackList**<E> (implements structure5.Stack<E>)
        - structure5.**StackVector**<E> (implements structure5.Stack<E>)
    - structure5.**AbstractList**<E> (implements structure5.List<E>)
      - structure5.**CircularList**<E>
      - structure5.**DoublyLinkedList**<E>
      - structure5.**SinglyLinkedList**<E>
      - structure5.**Vector**<E> (implements java.lang.Cloneable)
    - structure5.**AbstractSet**<E> (implements structure5.Set<E>)
      - structure5.**SetList**<E>
      - structure5.**SetVector**<E>

- structure5.**BinarySearchTree**<E> (implements structure5.OrderedStructure<K>)
  - structure5.**SplayTree**<E> (implements structure5.OrderedStructure<K>)
- structure5.**GraphList**<V,E> (implements structure5.Graph<V,E>)
  - structure5.**GraphListDirected**<V,E>
  - structure5.**GraphListUndirected**<V,E>
- structure5.**GraphMatrix**<V,E> (implements structure5.Graph<V,E>)
  - structure5.**GraphMatrixDirected**<V,E>
  - structure5.**GraphMatrixUndirected**<V,E>
- structure5.**OrderedList**<E> (implements structure5.OrderedStructure<K>)
- structure5.**OrderedVector**<E> (implements structure5.OrderedStructure<K>)
- structure5.**RedBlackSearchTree**<E> (implements structure5.OrderedStructure<K>)

- javax.swing.table.AbstractTableModel (implements java.io.Serializable, javax.swing.table.TableModel)
  - GUI.**guiTableModel**
- structure5.**Assert**
- structure5.**Association**<K,V> (implements java.util.Map.Entry<K,V>)
  - structure5.**ComparableAssociation**<K,V> (implements java.lang.Comparable<T>, java.util.Map.Entry<K,V>)
  - structure5.**HashAssociation**<K,V>
- structure5.**BinaryTree**<E>
- structure5.**BitSet**
- structure5.**CharSet**
- structure5.**Clock**
- java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
  - java.awt.Container
    - javax.swing.JComponent (implements java.io.Serializable)
      - javax.swing.JLabel (implements javax.accessibility.Accessible, javax.swing.SwingConstants)
        - GUI.**ColorRenderer1** (implements javax.swing.table.TableCellRenderer)
        - GUI.**ColorRenderer2** (implements javax.swing.table.TableCellRenderer)
    - java.awt.Window (implements javax.accessibility.Accessible)
      - java.awt.Frame (implements java.awt.MenuContainer)
        - javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)

        - GUI.**Credits** (implements java.awt.event.ActionListener)
        - GUI.**guiApplet** (implements java.awt.event.ActionListener, java.awt.event.ItemListener)
    - structure5.**DoublyLinkedNode**<E>
    - structure5.**Edge**<V,E>
        - structure5.**ComparableEdge**<V,E> (implements java.lang.Comparable<T>)
    - structure5.**Entry**<K,V> (implements java.util.Map.Entry<K,V>)
    - algor.**FunctionalUnit**
    - structure5.**Hashtable**<K,V> (implements java.lang.Iterable<T>, structure5.Map<K,V>)
    - java.io.InputStream (implements java.io.Closeable)
        - structure5.**FileStream**
        - java.io.FilterInputStream
            - structure5.**ReadStream**
    - algor.**Instruction**
    - algor.**InstructionNode**
    - structure5.**MapList**<K,V> (implements structure5.Map<K,V>)
    - structure5.**Matrix**<E>
    - structure5.**NaturalComparator**<E> (implements java.util.Comparator<T>)
    - structure5.**Node**<E>
    - algor.**PipelineUnit**
    - structure5.**PriorityVector**<E> (implements structure5.PriorityQueue<E>)
    - structure5.**RedBlackTree**<E>
    - algor.**RefStages**
    - structure5.**ReverseComparator**<E> (implements java.util.Comparator<T>)
    - structure5.**SkewHeap**<E> (implements structure5.MergeableHeap<E>)
    - algor.**Stages**
    - structure5.**StructCollection**<E> (implements java.util.Collection<E>)
    - structure5.**VectorHeap**<E> (implements structure5.PriorityQueue<E>)
    - structure5.**Version**
    - algor.**WithForwarding**
    - algor.**WithoutForwarding**

Interface Hierarchy

- java.lang.Iterable<T>
    - structure5.**Graph**<V,E>
    - structure5.**Linear**<E>
        - structure5.**Queue**<E>
        - structure5.**Stack**<E>
    - structure5.**List**<E>
    - structure5.**OrderedStructure**<K>

- structure5.**Queue**<E>
- structure5.**Set**<E>
- structure5.**Stack**<E>
- structure5.**Structure**<E>
  - structure5.**Graph**<V,E>
  - structure5.**Linear**<E>
    - structure5.**Queue**<E>
    - structure5.**Stack**<E>
  - structure5.**List**<E>
  - structure5.**OrderedStructure**<K>
  - structure5.**Queue**<E>
  - structure5.**Set**<E>
  - structure5.**Stack**<E>
- structure5.**Map**<K,V>
  - structure5.**OrderedMap**<K,V>
- structure5.**PriorityQueue**<E>
  - structure5.**MergeableHeap**<E>

ΠΑΚΕΤΟ algor

Class Summary	
FunctionalUnit	
Instruction	
InstructionNode	
PipelineUnit	
RefStages	
Stages	
WithForwarding	
WithoutForwarding	

algor

Class FunctionalUnit

java.lang.Object
└ **algor.FunctionalUnit**

public class **FunctionalUnit**
extends java.lang.Object

Constructor Summary

FunctionalUnit(int integer, int add, int multiply, int divide)

Method Summary

int	**getAdd**()
int	**getDivide**()
int	**getInteger**()
int	**getMultiply**()
int	**getNoUnit**(java.lang.String operation)
int	**setAdd**(int add)
int	**setDivide**(int divide)
int	**setInteger**(int integer)
int	**setMultiply**(int multiply)

Constructor Detail

FunctionalUnit

public **FunctionalUnit**(int integer,
int add,
int multiply,
int divide)

Method Detail

getNoUnit

public int **getNoUnit**(java.lang.String operation)

getInteger
public int **getInteger**()

getAdd
public int **getAdd**()

getMultiply
public int **getMultiply**()

getDivide
public int **getDivide**()

setInteger
public int **setInteger**(int integer)

setAdd
public int **setAdd**(int add)

setMultiply
public int **setMultiply**(int multiply)

setDivide
public int **setDivide**(int divide)

algor

Class Instruction

java.lang.Object
└ **algor.Instruction**

public class **Instruction**
extends java.lang.Object

Constructor Summary

Instruction()
Instruction(java.lang.String type, java.lang.String destination, java.lang.String source1, java.lang.String source2, int numberOfCycles)

Method Summary

java.lang.String	**getDestination**()
int	**getNumberOfCycles**()
java.lang.String	**getSource1**()
java.lang.String	**getSource2**()
java.lang.String	**getType**()
void	**setDestination**(java.lang.String destination)
void	**setNumberOfCycles**(int numberOfCycles)
void	**setSource1**(java.lang.String source1)
void	**setSource2**(java.lang.String source2)
void	**setType**(java.lang.String type)

Constructor Detail

Instruction
public **Instruction**()

Instruction
public **Instruction**(java.lang.String type,
java.lang.String destination,
java.lang.String source1,
java.lang.String source2,
int numberOfCycles)

Method Detail

getType
public java.lang.String **getType**()

---

getDestination
public java.lang.String **getDestination**()

---

getSource1
public java.lang.String **getSource1**()

---

getSource2
public java.lang.String **getSource2**()

---

getNumberOfCycles
public int **getNumberOfCycles**()

---

setType
public void **setType**(java.lang.String type)

---

setDestination
public void **setDestination**(java.lang.String destination)

---

setSource1
public void **setSource1**(java.lang.String source1)

---

setSource2
public void **setSource2**(java.lang.String source2)

setNumberOfCycles
public void **setNumberOfCycles**(int numberOfCycles)

algor

Class InstructionNode

java.lang.Object
  └ **algor.InstructionNode**

public class **InstructionNode**
extends java.lang.Object

Field Summary

int	**indegree**

Constructor Summary

InstructionNode(Instruction instruction, int number)

Method Summary

Instruction	**getInstruction**()
int	**getNumber**()
RefStages	**getRefNode**()
RefStages	**getRefStages**()
Stages	**getStages**()
void	**setInstruction**(Instruction instruction)

void	**setNumber**(int number)
void	**setRefNode**(RefStages refNode)
void	**setRefStages**(RefStages refStages)
void	**setStages**(Stages stages)

Field Detail

indegree
public int **indegree**

Constructor Detail

InstructionNode
public **InstructionNode**(Instruction instruction,
int number)

Method Detail

getNumber
public int **getNumber**()

---

getInstruction
public Instruction **getInstruction**()

---

getStages
public Stages **getStages**()

---

getRefNode
public RefStages **getRefNode**()

---

getRefStages
public RefStages **getRefStages**()

---

setNumber
public void **setNumber**(int number)

setInstruction
public void **setInstruction**(Instruction instruction)

setStages
public void **setStages**(Stages stages)

setRefNode
public void **setRefNode**(RefStages refNode)

setRefStages
public void **setRefStages**(RefStages refStages)

algor

Class PipelineUnit

java.lang.Object
└ **algor.PipelineUnit**

public class **PipelineUnit**
extends java.lang.Object

Constructor Summary

PipelineUnit()	

Method Summary

int	**getAdd**()
int	**getDivide**()
int	**getLoad**()

int	**getMultiply**()
int	**getValue**(java.lang.String operation)
int	**setAdd**(int add)
int	**setDivide**(int divide)
int	**setLoad**(int load)
int	**setMultiply**(int multiply)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PipelineUnit
public **PipelineUnit**()

Method Detail

getValue
public int **getValue**(java.lang.String operation)

---

getLoad
public int **getLoad**()

---

getAdd
public int **getAdd**()

---

getMultiply
public int **getMultiply**()

---

getDivide
public int **getDivide**()

---

setLoad
public int **setLoad**(int load)

---

setAdd
public int **setAdd**(int add)

---

setMultiply
public int **setMultiply**(int multiply)

---

setDivide
public int **setDivide**(int divide)

algor

Class RefStages

java.lang.Object
  └ **algor.RefStages**

---

public class **RefStages**
extends java.lang.Object

Constructor Summary

RefStages()

Method Summary

void	**addDecodeDep**(int nodeNo)
void	**addExecuteDep**(int nodeNo)
void	**addIFetchDep**(int nodeNo)
void	**addStructuralDep**(int nodeNo)

void	**addWBackDep**(int nodeNo)
int	**getD**()
int	**getDecode**(int i)
int	**getE**()
int	**getExecEnd**(int i)
int	**getI**()
int	**getIFetch**(int i)
int	**getS**()
int	**getStructural**(int i)
int	**getW**()
int	**getWBack**(int i)
void	**setD**(int d)
void	**setDecode**(int[] is)
void	**setE**(int e)
void	**setExecEnd**(int[] is)
void	**setI**(int i)
void	**setIFetch**(int[] is)
void	**setS**(int s)
void	**setStructural**(int[] is)

void	**setW**(int w)
void	**setWBack**(int[] is)

Constructor Detail

RefStages
public **RefStages**()

Method Detail

addIFetchDep
public void **addIFetchDep**(int nodeNo)

---

addDecodeDep
public void **addDecodeDep**(int nodeNo)

---

addExecuteDep
public void **addExecuteDep**(int nodeNo)

---

addWBackDep
public void **addWBackDep**(int nodeNo)

---

addStructuralDep
public void **addStructuralDep**(int nodeNo)

---

getIFetch
public int **getIFetch**(int i)

---

getDecode
public int **getDecode**(int i)

---

getExecEnd
public int **getExecEnd**(int i)

---

getWBack
public int **getWBack**(int i)

---

getStructural
public int **getStructural**(int i)

---

getI
public int **getI**()

---

getD
public int **getD**()

---

getE
public int **getE**()

---

getW
public int **getW**()

---

getS
public int **getS**()

---

setIFetch
public void **setIFetch**(int[] is)

---

setDecode
public void **setDecode**(int[] is)

---

setExecEnd
public void **setExecEnd**(int[] is)

---

setWBack
public void **setWBack**(int[] is)

setStructural
public void **setStructural**(int[] is)

setI
public void **setI**(int i)

setD
public void **setD**(int d)

setE
public void **setE**(int e)

setW
public void **setW**(int w)

setS
public void **setS**(int s)

algor

Class Stages

java.lang.Object
  └ **algor.Stages**

public class **Stages**
extends java.lang.Object

Constructor Summary

Stages()

Method Summary

int	**getDecode**()

int	**getExecEnd**()
int	**getExecStart**()
int	**getIfetch**()
int	**getStageCycle**(int cycle)
int	**getWback**()
void	**setDecode**(int decode)
void	**setExecEnd**(int memory)
void	**setExecStart**(int execute)
void	**setIfetch**(int iFetch)
void	**setWback**(int wBack)

Constructor Detail

Stages
public **Stages**()

Method Detail

getStageCycle
public int **getStageCycle**(int cycle)

---

getIfetch
public int **getIfetch**()

---

getDecode
public int **getDecode**()

getExecStart
public int **getExecStart**()

getExecEnd
public int **getExecEnd**()

getWback
public int **getWback**()

setIfetch
public void **setIfetch**(int iFetch)

setDecode
public void **setDecode**(int decode)

setExecStart
public void **setExecStart**(int execute)

setExecEnd
public void **setExecEnd**(int memory)

setWback
public void **setWback**(int wBack)

algor

Class WithForwarding

java.lang.Object
  └ **algor.WithForwarding**

public class **WithForwarding**
extends java.lang.Object

Field Summary

java.lang.String[][]	**timeChart**

Constructor Summary

WithForwarding(int numInstruction, PipelineUnit pUnit, FunctionalUnit fUnit)

Method Summary

void	**addInstruction**(InstructionNode node)
InstructionNode	**findNodeZeroIndegree**()
java.lang.String	**generatedHazardsGraph**()
int	**getCount**()
FunctionalUnit	**getFUnit**()
GraphListDirected	**getGraph**()
InstructionNode	**getInstrNode**(int index)
int	**getMaxCycle**()
int	**getNumInstruction**()
int	**getPipeline**()
PipelineUnit	**getPUnit**()
int	**getTotalCycle**()
void	**setGraph**(GraphListDirected graph)
void	**setInstrNode**(InstructionNode[] node)

void	**setNumInstruction**(int number)
void	**setTotalCycle**(int cycles)
void	**topologicalTraversal**()

Field Detail

timeChart
public java.lang.String[][] **timeChart**

Constructor Detail

WithForwarding
public **WithForwarding**(int numInstruction,
PipelineUnit pUnit,
FunctionalUnit fUnit)

Method Detail

addInstruction
public void **addInstruction**(InstructionNode node)

---

generatedHazardsGraph
public java.lang.String **generatedHazardsGraph**()

---

topologicalTraversal
public void **topologicalTraversal**()

---

findNodeZeroIndegree
public InstructionNode **findNodeZeroIndegree**()

---

getCount
public int **getCount**()

getFUnit
public FunctionalUnit **getFUnit**()

getGraph
public GraphListDirected **getGraph**()

getInstrNode
public InstructionNode **getInstrNode**(int index)

getMaxCycle
public int **getMaxCycle**()

getNumInstruction
public int **getNumInstruction**()

getPUnit
public PipelineUnit **getPUnit**()

getPipeline
public int **getPipeline**()

getTotalCycle
public int **getTotalCycle**()

setGraph
public void **setGraph**(GraphListDirected graph)

setInstrNode
public void **setInstrNode**(InstructionNode[] node)

setNumInstruction
public void **setNumInstruction**(int number)

setTotalCycle
public void **setTotalCycle**(int cycles)

algor
Class WithoutForwarding
java.lang.Object
└ **algor.WithoutForwarding**

public class **WithoutForwarding**
extends java.lang.Object

Field Summary

java.lang.String[][]	**timeChart**

Constructor Summary

WithoutForwarding(int numInstruction, PipelineUnit pUnit, FunctionalUnit fUnit)

Method Summary

void	**addInstruction**(InstructionNode node)
InstructionNode	**findNodeZeroIndegree**()
java.lang.String	**generatedHazardsGraph**()
int	**getCount**()
FunctionalUnit	**getFUnit**()
GraphListDirected	**getGraph**()
InstructionNode	**getInstrNode**(int index)
int	**getMaxCycle**()

int	**getNumInstruction**()
int	**getPipeline**()
PipelineUnit	**getPUnit**()
int	**getTotalCycle**()
void	**setGraph**(GraphListDirected graph)
void	**setInstrNode**(InstructionNode[] node)
void	**setNumInstruction**(int number)
void	**setTotalCycle**(int cycles)
void	**topologicalTraversal**()

Field Detail

timeChart
public java.lang.String[][] **timeChart**

Constructor Detail

WithoutForwarding
public **WithoutForwarding**(int numInstruction,
PipelineUnit pUnit,
FunctionalUnit fUnit)

Method Detail

addInstruction
public void **addInstruction**(InstructionNode node)

---

generatedHazardsGraph
public java.lang.String **generatedHazardsGraph**()

---

topologicalTraversal
public void **topologicalTraversal**()

---

findNodeZeroIndegree
public InstructionNode **findNodeZeroIndegree**()

---

getCount
public int **getCount**()

---

getFUnit
public FunctionalUnit **getFUnit**()

---

getGraph
public GraphListDirected **getGraph**()

---

getInstrNode
public InstructionNode **getInstrNode**(int index)

---

getMaxCycle
public int **getMaxCycle**()

---

getNumInstruction
public int **getNumInstruction**()

---

getPUnit
public PipelineUnit **getPUnit**()

---

getPipeline
public int **getPipeline**()

---

getTotalCycle
public int **getTotalCycle**()

---

setGraph
public void **setGraph**(GraphListDirected graph)

setInstrNode
public void **setInstrNode**(InstructionNode[] node)

setNumInstruction
public void **setNumInstruction**(int number)

setTotalCycle
public void **setTotalCycle**(int cycles)

ΠΑΚΕΤΟ GUI

Class Summary	
ColorRenderer1	
Credits	
guiSimulator	
guiTableModel	
InstructionCode	

GUI

Class ColorRenderer1

java.lang.Object
  └ java.awt.Component
    └ java.awt.Container
      └ javax.swing.JComponent
        └ javax.swing.JLabel
          └ **GUI.ColorRenderer1**

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.SwingConstants, javax.swing.table.TableCellRenderer

public class **ColorRenderer1**
extends javax.swing.JLabel
implements javax.swing.table.TableCellRenderer

Constructor Summary

ColorRenderer1()

Method Summary

java.awt.Component	**getTableCellRendererComponent**(javax.swing.JTable table, java.lang.Object value, boolean isSelected, boolean hasFocus, int row, int column)

Constructor Detail

ColorRenderer1

public **ColorRenderer1**()

Method Detail

getTableCellRendererComponent

public java.awt.Component **getTableCellRendererComponent**(javax.swing.JTable table,
java.lang.Object value,
boolean isSelected,
boolean hasFocus,
int row,
int column)

Specified by:

getTableCellRendererComponent in interface javax.swing.table.TableCellRenderer

GUI

Class Credits

java.lang.Object
└ java.awt.Component
└ java.awt.Container
└ java.awt.Window
└ java.awt.Frame
└ javax.swing.JFrame
└ **GUI.Credits**

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

public class **Credits**
extends javax.swing.JFrame
implements java.awt.event.ActionListener

Method Summary

void	**actionPerformed**(java.awt.event.ActionEvent e)
static void	**main**(java.lang.String[] args)

Method Detail

actionPerformed

public void **actionPerformed**(java.awt.event.ActionEvent e)

Specified by:

actionPerformed in interface java.awt.event.ActionListener

---

main

public static void **main**(java.lang.String[] args)

Class InstructionCode

java.lang.Object
  └ java.awt.Component
    └ java.awt.Container
      └ java.awt.Window
        └ java.awt.Frame
          └ javax.swing.JFrame
            └ **GUI.InstructionCode**

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener,

javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

public class **InstructionCode**
extends javax.swing.JFrame
implements java.awt.event.ActionListener

Field Summary

java.awt.Container	**c**

Constructor Summary

InstructionCode(guiSimulator gui)

Method Summary

void	**actionPerformed**(java.awt.event.ActionEvent e)
void	**assemblyCode**()
protected javax.swing.ImageIcon	**createImageIcon**(java.lang.String path, java.lang.String description) Returns an ImageIcon, or null if the path was invalid.
static void	**main**(java.lang.String[] args)

Field Detail

c

public java.awt.Container **c**

Constructor Detail

InstructionCode

public **InstructionCode**(guiSimulator gui)

Method Detail

assemblyCode

public void **assemblyCode**()

---

createImageIcon

protected javax.swing.ImageIcon **createImageIcon**(java.lang.String path,
java.lang.String description)

Returns an ImageIcon, or null if the path was invalid.

---

actionPerformed

public void **actionPerformed**(java.awt.event.ActionEvent e)

Specified by:

actionPerformed in interface java.awt.event.ActionListener

---

main

public static void **main**(java.lang.String[] args)

Class guiSimulator

java.lang.Object
└ java.awt.Component
└ java.awt.Container
└ java.awt.Window
└ java.awt.Frame
└ javax.swing.JFrame
└ **GUI.guiSimulator**

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.event.ItemListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

public class **guiSimulator**
extends javax.swing.JFrame
implements java.awt.event.ActionListener, java.awt.event.ItemListener

---

Field Summary

Constructor Summary

guiSimulator()	

Method Summary

void	**actionPerformed**(java.awt.event.ActionEvent e)
void	**executeWithForwarding**(PipelineUnit p, FunctionalUnit unit)
void	**executeWithoutForwarding**(PipelineUnit p, FunctionalUnit unit)
void	**itemStateChanged**(java.awt.event.ItemEvent e)
static void	**main**(java.lang.String[] args)
void	**pressExecute**()
void	**pressReset**()

Constructor Detail

guiSimulator

public **guiSimulator**()
throws javax.swing.UnsupportedLookAndFeelException

Throws:

javax.swing.UnsupportedLookAndFeelException

Method Detail

actionPerformed

public void **actionPerformed**(java.awt.event.ActionEvent e)

Specified by:

actionPerformed in interface java.awt.event.ActionListener

---

itemStateChanged

public void **itemStateChanged**(java.awt.event.ItemEvent e)

Specified by:

itemStateChanged in interface java.awt.event.ItemListener

---

pressReset

public void **pressReset**()

---

pressExecute

public void **pressExecute**()

---

executeWithForwarding

public void **executeWithForwarding**(PipelineUnit p,
FunctionalUnit unit)

---

executeWithoutForwarding

public void **executeWithoutForwarding**(PipelineUnit p,
FunctionalUnit unit)

---

main

public static void **main**(java.lang.String[] args)
throws javax.swing.UnsupportedLookAndFeelException

Throws:

javax.swing.UnsupportedLookAndFeelException

GUI

Class guiTableModel

java.lang.Object
└ javax.swing.table.AbstractTableModel
└ **GUI.guiTableModel**

All Implemented Interfaces:

java.io.Serializable, javax.swing.table.TableModel

public class **guiTableModel**
extends javax.swing.table.AbstractTableModel

Constructor Summary

guiTableModel()	

Method Summary

java.lang.Class	**getColumnClass**(int c)
int	**getColumnCount**()
java.lang.String	**getColumnNames**(int col)
int	**getRowCount**()
java.lang.Object	**getValueAt**(int rowIndex, int columnIndex)
void	**setData**(int row)
void	**setRowCol**(java.lang.String[] name, java.lang.String col)
void	**setValueAt**(java.lang.Object value, int row, int col)

Constructor Detail

guiTableModel
public **guiTableModel**()

Method Detail

setData
public void **setData**(int row)

setRowCol
public void **setRowCol**(java.lang.String[] name,
java.lang.String col)

---

getRowCount
public int **getRowCount**()

---

getColumnCount
public int **getColumnCount**()

---

getColumnNames
public java.lang.String **getColumnNames**(int col)

---

getValueAt
public java.lang.Object **getValueAt**(int rowIndex,
int columnIndex)

---

getColumnClass
public java.lang.Class **getColumnClass**(int c)

Specified by:

getColumnClass in interface javax.swing.table.TableModel

Overrides:

getColumnClass in class javax.swing.table.AbstractTableModel

---

setValueAt
public void **setValueAt**(java.lang.Object value,
int row,
int col)

Specified by:

setValueAt in interface javax.swing.table.TableModel

Overrides:

setValueAt in class javax.swing.table.AbstractTableModel

Ε2. Ευρετήριο όρων

ΣΤ. ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] David A. Patterson, John L.Hennesy: "Computer Organization & Design", Third Edition, Elsevier Inc., 2005

[2] Mostafa Abo-El-Barr & Hesham El-Rewini: "Funtamentals of Computer Organization & Architecture",Wiley-Interscience, 2005

[3] Αντ. Βαφειάδης & Κ. Διαμαντάρας: "Προηγμένες Αρχιτεκτονικές Υπολογιστών", 2006

[4] Martin Fowler & Kendal Scott: "Εισαγωγή στη UML, Δεύτερη Αμερικανική Έκδοση, Συνοπτικός οδηγός της Πρότυπης Γλώσσας Μοντελοποίησης Αντικειμένων", Εκδόσεις Κλειδάριθμος 2001

[5] Shally Shlaer & Stephen J. Meillor: "Recursive Design of an Application Independent Architrcture" IEEE Software, Vol. 14, No 1, 1997

[6] Γιώργος Λιακέας: "Εισαγωγή στην Java 2", Εκδόσεις Κλειδάριθμος, 2003

[7] Ted Young: "Java 2 Bible", John Willey and Sons, 2000

[8] John R. Hubbard: "Java, θεωρία και προβήματα", Εκδόσεις Κλειδάριθμος, 1999

[9] Duane A. Bailey: "Java Structures, Data Stuctures in Java for the Principled Programmer", Williams College, 2007

[10] Robert Eckstein, Mark Loy & Dave Wood: "Java Swing", O'Reilly, 1998

Σελίδες στον ιστοχώρο με βιβλιογραφικό υλικό

http://en.wikipedia.org/wiki/Pipeline_%28computing%29

http://en.wikipedia.org/wiki/Instruction_pipeline

http://www.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/hazards.html

http://www.cs.nmsu.edu/~pfeiffer/classes/473/notes/hazards.html

http://en.wikipedia.org/wiki/Hazard_(computer_architecture)

http://www.ecs.umass.edu/ece/koren/architecture/windlx/main.html

http://java.sun.com/docs/books/tutorial/reallybigindex.html

http://java.sun.com/javase/6/docs/api/

http://www.cs.umd.edu/class/fall2001/cmsc411/proj01/DLX/index.html

http://www.cs.bu.edu/~best/courses/cs550/fridman/pdlx/dlxsim.html

http://architecture.di.uoa.gr/5sect13.html#%F3%E5%EB0

http://www.cs.umbc.edu/~plusquel/611/slides/chap3_3.html