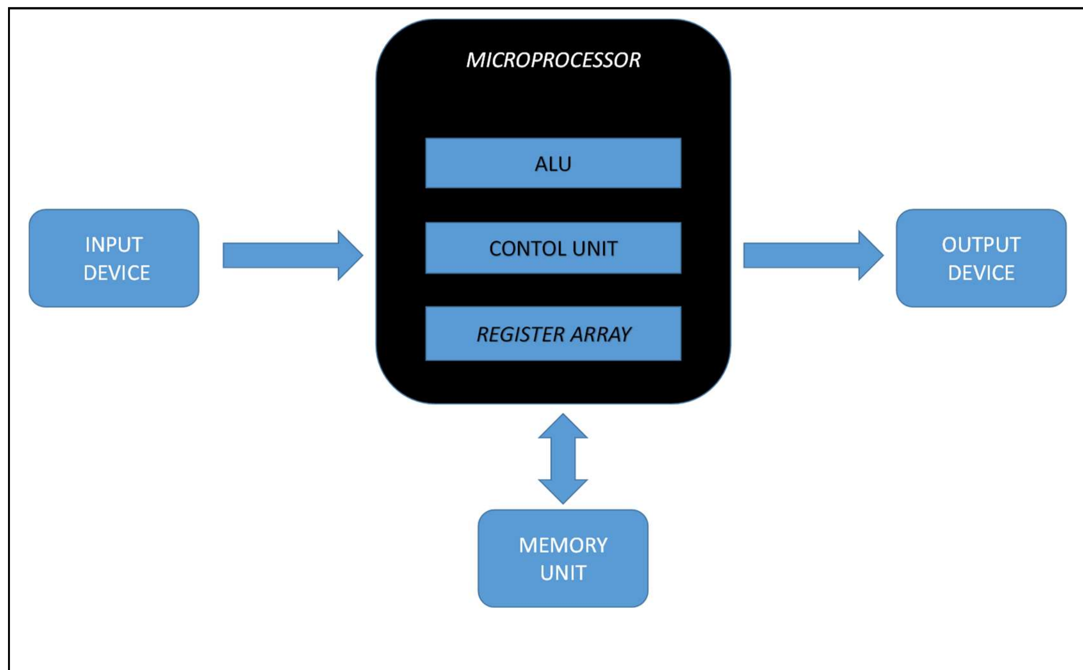
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ & ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ

Διπλωματική Εργασία

Σχεδίαση Μονάδας Αριθμητικής – Λογικής (ALU) σε HDL



Φοιτητής: Γιώργος Οικονόμου
AM: ee06695

Επιβλέπων Καθηγητής

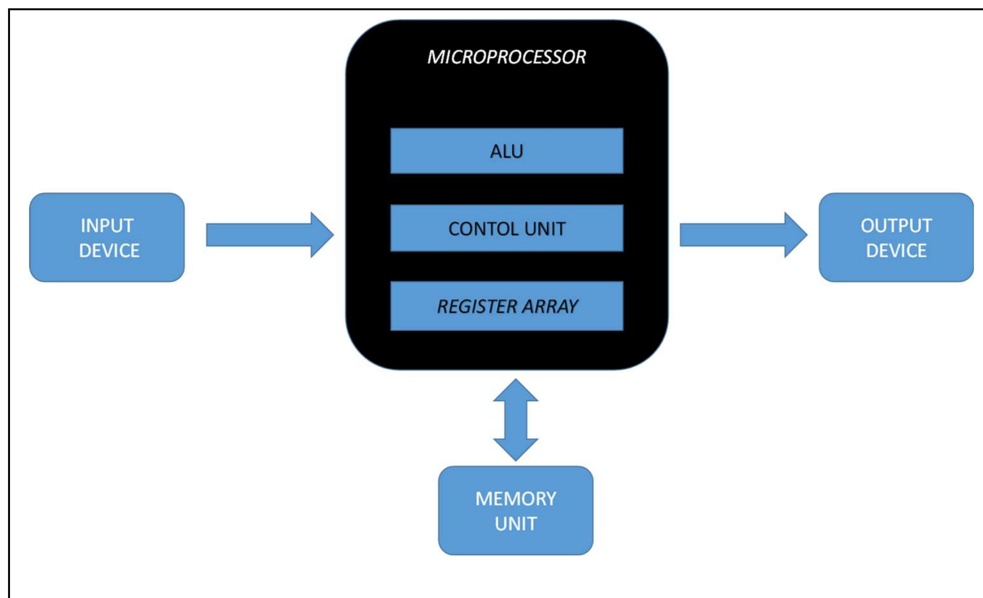
Οδυσσεύς Τσακιρίδης

ΑΘΗΝΑ-ΑΙΓΑΛΕΩ, ΙΟΥΛΙΟΣ 2024

UNIVERSITY OF WEST ATTICA
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

Diploma Thesis

Design of Arithmetic – Logic Unit using HDL



Student: George Economou
Registration Number: ee06695

Supervisor

Odyssefs Tsakiridis

ATHENS-EGALEO, JULY 2024

Η Διπλωματική Εργασία έγινε αποδεκτή και βαθμολογήθηκε από την εξής τριμελή επιτροπή:

Τσακιρίδης Οδυσσεύς, Επίκουρος Καθηγητής	Φωτόπουλος Παναγιώτης, Αναπληρωτής Καθηγητής	Γαλατά Σωτηρία, Επίκουρη Καθηγήτρια

Copyright © Με επιφύλαξη παντός δικαιώματος. All rights reserved.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ και Γιώργος Οικονόμου,
Ιούλιος 2024

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τους συγγραφείς.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον/την συγγραφέα του και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις θέσεις του επιβλέποντος, της επιτροπής εξέτασης ή τις επίσημες θέσεις του Τμήματος και του Ιδρύματος.

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Ο κάτωθι υπογεγραμμένος **Γιώργος Οικονόμου** του Νικολάου με αριθμό μητρώου ee06695 φοιτητής του Πανεπιστημίου Δυτικής Αττικής της Σχολής ΜΗΧΑΝΙΚΩΝ του Τμήματος ΗΛΕΚΤΡΟΛΟΓΩΝ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ,

δηλώνω υπεύθυνα ότι:

«Είμαι συγγραφέας αυτής της διπλωματικής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος.
Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του διπλώματός μου.
Επιθυμώ την απαγόρευση πρόσβασης στο πλήρες κείμενο της εργασίας μου μέχρι ……………………… και έπειτα από αίτησή μου στη Βιβλιοθήκη και έγκριση του επιβλέποντος καθηγητή.»

Ο/Η Δηλών/ούσα
Γιώργος Οικονόμου

Περίληψη

Τα τελευταία χρόνια η ραγδαία εξέλιξη της τεχνολογίας έχει οδηγήσει στην πραγμάτωση όλο και πολυπλοκότερων συστημάτων σε διάφορους τομείς της καθημερινότητας, όπως της Πληροφορικής, της Ενέργειας, των Τηλεπικοινωνιών και της Οικονομίας. Πυρήνας κάθε τέτοιου συστήματος είναι τα ψηφιακά κυκλώματα. Με τον όρο ψηφιακό κύκλωμα, αναφερόμαστε σε ηλεκτρονικά συστήματα που επεξεργάζονται διακριτά επίπεδα ηλεκτρικής τάσης.

Η παρούσα διπλωματική εργασία λοιπόν παρουσιάζει την προσομοίωση ενός ψηφιακού συστήματος, το οποίο έχει υλοποιηθεί με χρήση της γλώσσας περιγραφής υλικού VHDL και αναλύει επίσης τις βασικές αρχές των ψηφιακών κυκλωμάτων. Στα κεφάλαια που ακολουθούν γίνεται μια αναλυτική παρουσίαση των αρχών του σχεδιασμού ψηφιακών συστημάτων, της μεθοδολογίας σχεδιασμού επεξεργαστή ενός κύκλου και λεπτομερούς ερμηνείας του κώδικα του επεξεργαστή που υλοποιήθηκε.

Πιο συγκεκριμένα στο πρώτο κεφάλαιο υπάρχει η δομή της εργασίας, ο σκοπός και οι μεθοδολογίες που χρησιμοποιήθηκαν για την παρούσα διπλωματική εργασία. Στο δεύτερο κεφάλαιο αναφέρονται οι βασικές αρχές για τον σχεδιασμό και την υλοποίηση ψηφιακών κυκλωμάτων. Παρουσιάζονται τα δομικά χαρακτηριστικά διάφορων γλωσσών περιγραφής υλικού καθώς επίσης και τα πλεονεκτήματα των ψηφιακών συστημάτων συγκριτικά με άλλες τεχνολογίες.

Στην συνέχεια, στο τρίτο κεφάλαιο γίνεται μια λεπτομερής παρουσίαση της γλώσσας VHDL, των τύπων δεδομένων που χρησιμοποιεί, των μεταβλητών και των χρήσιμων συνθηκών που απαιτούνται για την συγγραφή κώδικα με VHDL. Στο τέταρτο κεφάλαιο υπάρχει ένας οδηγός ανάλυσης των βασικών στοιχείων ενός επεξεργαστή ενός κύκλου, στα οποία συγκαταλέγονται ο μετρητής προγράμματος, ο δείκτης και καταχωρητής εντολών, οι μνήμες Cache και RAM, η μονάδα διαχείρισης μνήμης, η αριθμητική λογική μονάδα (ALU), ο Controller και η βαθμίδα αποκωδικοποίησης εντολών.

Το πέμπτο κεφάλαιο είναι η ανάλυση του κώδικα που έχει γραφτεί, για την υλοποίηση του επεξεργαστή ενός κύκλου στα πλαίσια της διπλωματικής εργασίας. Επιπλέον, γίνεται πρακτική υλοποίηση της αρχιτεκτονικής με παρόμοια μορφή στο PCB Mojo v.3. Ακολουθεί το τελευταίο κεφάλαιο με τα συμπεράσματα και την βιβλιογραφία.

Λέξεις – κλειδιά

Ολοκληρωμένα κυκλώματα, VHDL, Επεξεργαστής, Μονάδα Ελέγχου, Μνήμη, Εκτέλεση εντολών, Αριθμητική Λογική Μονάδα, Ελεγκτής, Γλώσσα Περιγραφής Υλικού, Αποκωδικοποίηση, Καταχωρητής, Κώδικας, Ψηφιακό Σύστημα, Mojo_V3

Abstract

In recent years the rapid development of technology has led to the need for implementation of even more complex systems in various fields of everyday life, such as IT, Energy, Telecommunications and Financials. At the core of any such system are the digital circuits. By the term digital circuit, we refer to electronic systems that process distinct levels of electrical voltage.

This thesis presents the simulation of a digital system, which has been implemented using the hardware description language VHDL and also analyzes the basic principles of digital circuits. The following chapters provide a detailed presentation of the principles of digital system design, the single-cycle processor design methodology, and a detailed interpretation of the implemented processor code.

More specifically, in the first chapter there is the structure, the purpose and the methodologies used for this thesis. In the second chapter, the basic principles for the design and implementation of digital circuits are mentioned. The structural features of various hardware description languages are presented as well as the advantages of digital systems compared to other technologies.

Then, in the third chapter, there is a detailed presentation of the VHDL language, the data types it uses, the variables and the useful conditions required for writing code with VHDL. In the fourth chapter there is an analysis guide to the basic components of a single-cycle processor, which include the program Counter, the Instruction Pointer Register, the Cache and RAM memories, the Memory Management Unit (MMU), the Arithmetic Logic Unit (ALU), the Controller, and the Instruction Decoder.

The fifth chapter is the analysis of the code that has been written, for the implementation of the single-cycle processor in the context of the diploma thesis. Furthermore, a similar architecture is being implemented on the printed circuit board of Mojo v.3. This is followed by the last chapter with the conclusions and bibliography.

Keywords

Integrated Circuits, VHDL, Processor, Control Unit, Memory, Instruction Execution, Arithmetic Logic Unit, Controller, Hardware Description Language, Decoding, Register, Code, Digital System, Mojo_V3

Περιεχόμενα

Περίληψη 5
Λέξεις – κλειδιά 5
Abstract 6
Keywords 6
Κατάλογος Διαγραμμάτων 10
Κατάλογος Εικόνων 10
Αλφαβητικό Ευρετήριο 11
1 Εισαγωγή 12
1.1 Αντικείμενο της διπλωματικής εργασίας 12
1.2 Σκοπός και στόχοι 12
1.3 Μεθοδολογία 13
1.4 Καινοτομία 14
1.5 Δομή 15
2 Ψηφιακά συστήματα 17
2.1 Προσωπικός Υπολογιστής & Bit 17
2.2 Η γλώσσα μηχανής 17
2.3 Ολοκληρωμένα κυκλώματα 19
2.4 Σχεδίαση ολοκληρωμένων ψηφιακών συστημάτων 20
2.5 Πλεονεκτήματα ψηφιακών συστημάτων 23
2.6 Γλώσσες περιγραφής υλικού 23
3 Η γλώσσα VHDL 25
3.1 Ο τρόπος λειτουργίας της VHDL 25
3.2 Αριθμοί και χαρακτήρες στη VHDL 27
3.3 Δομή κώδικα 27
3.4 Λεκτικά στοιχεία 29
3.5 Η «σύγχρονη» εντολή SELECT 30
3.6 Η δομή PROCESS 30
3.7 Αντικείμενα και τύποι δεδομένων στη VHDL 30
3.7.1 Αντικείμενα – objects δεδομένων 30
3.7.2 Δήλωση αντικειμένων 31
3.7.3 Τύποι δεδομένων 32
3.7.4 Προκαθορισμένοι τύποι δεδομένων 32
3.7.5 Τύποι standard logic 33
3.8 Τελεστές και πράξεις 33
3.8.1 Τελεστές ανάθεσης 34
3.8.2 Λογικοί τελεστές 34

3.8.3 Σχεσιακοί τελεστές 34
3.8.4 Τελεστές αριθμητικών πράξεων 35
3.8.5 Τελεστές ολίσθησης 35
3.8.6 Τελεστής συνένωσης 36
3.9 Χρήσιμες συνθήκες 36
3.10 Η εντολή WAIT 38
3.11 Η εντολή LOOP 38
3.12 Η εντολή CASE 39
4 Σχεδίαση επεξεργαστή ενός κύκλου 41
4.1 Μετρητής προγράμματος 41
4.2 Δείκτης & καταχωρητής εντολών 42
4.3 Cache & RAM 42
4.4 Μονάδα διαχείρισης μνήμης 43
4.5 Αριθμητική λογική μονάδα (ALU) 43
4.6 Ελεγκτής (Controller) 44
4.7 Βαθμίδα αποκωδικοποίησης εντολών 44
4.8 Μονάδα καταχωρητών 45
4.9 CPU clock 46
5 Διαγράμματα & Ερμηνεία του κώδικα ακολουθούμενη από πρακτική υλοποίηση στο Mojo v.3 47
5.1 Αθροιστής (Adder) 47
5.2 Εκτέλεση πράξεων 49
5.3 Decoder 53
5.4 Controller 54
5.5 Μνήμη 56
5.6 Entity MEM 56
5.7 Βαθμίδα πρόσβασης στη μνήμη 59
5.8 Καταχωρητής 60
5.9 RF οντότητα 61
5.10 EXSTAGE & IFSTAGE οντότητα 63
5.11 Το PCB του Mojo v.3 και η δομή του 64
6 Συμπεράσματα 76
7 Βιβλιογραφία – Αναφορές - Διαδικτυακές Πηγές 78
Παράρτημα Α 79
Παράρτημα Β – entity ALU 80
Παράρτημα Γ - Decoder 82
Παράρτημα Δ - Controller 83
Παράρτημα Ε 85

Παράρτημα Ζ....88
Παράρτημα Η....90
Παράρτημα Θ....91
Παράρτημα Ι....93
Παράρτημα Κ....96
Παράρτημα Λ....98
Παράρτημα Μ....99
Παράρτημα Ν....100
Παράρτημα Ξ....102

Κατάλογος Διαγραμμάτων

Διάγραμμα 2.1: Νόμος του Moore 18
Διάγραμμα 2.2: Σχεδίαση ψηφιακών συστημάτων 19
Διάγραμμα 2.3: Κυματομορφή ψηφιακού σήματος 20
Διάγραμμα 2.4: Typical diagram of a CPU 31
Διάγραμμα 3.1: Ιεραρχική σύνδεση οντοτήτων 24
Διάγραμμα 5.1: Η πράξη της πρόσθεσης 41
Διάγραμμα 5.2: Η πράξη της πρόσθεσης με αρνητικούς αριθμούς 42
Διάγραμμα 5.3: Η πράξη της αφαίρεσης 42
Διάγραμμα 5.4: Εκτέλεση λογικών & αριθμητικών πράξεων με bin & hex αριθμούς 43
Διάγραμμα 5.5: Decoder 45
Διάγραμμα 5.6: Η μνήμη του επεξεργαστή 47
Διάγραμμα 5.7: Κατάσταση της μνήμης 51
Διάγραμμα 5.8: IFSTAGE, DECSTAGE, EXSTAGE, MEMSTAGE οντότητες 55
Διάγραμμα 6.1: Ακολουθιακά & Συνδυαστικά κυκλώματα 58

Κατάλογος Εικόνων

Εικόνα 2.1: Πεδία Εντολών Γλώσσας Μηχανής 16
Εικόνα 2.2: Εσωτερική δομή Τρανζίστορ 17
Εικόνα 2.3: PCB πλακέτα 18
Εικόνα 2.4: Λογικές πύλες 20
Εικόνα 5.1: Ο αθροιστής 38
Εικόνα 5.2: Εκτέλεση αριθμητικών & λογικών πράξεων 40
Εικόνα 5.3: Κώδικας Decoder 44
Εικόνα 5.4: Τμήμα του κώδικα για τον Controller 46
Εικόνα 5.5: Είσοδοι στο τσιπάκι της μνήμης 48
Εικόνα 5.6: Η MEMSTAGE αρχιτεκτονική 49
Εικόνα 5.7: Πρόσβαση στη μνήμη 50

Εικόνα 5.8: Η αρχικοποίηση ενός καταχωρητή 52
Εικόνα 5.9: RF οντότητα 53
Εικόνα 5.10: Mojo v.3 PCB 67

Αλφαβητικό Ευρετήριο

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
RF	Radio Frequency
VHDL	VHSIC Hardware Description Language
RAM	Random Access Memory
DRAM	Dynamic Random Access Memory
CU	Control Unit
MMU	Memory Management Unit
ISA	Instruction Set Architecture
PCB	Printed Circuit Board
IEEE	Institute of Electrical and Electronics Engineers
Ovf	Overflow
IoT	Internet of Things

1 Εισαγωγή

Στο συνεχώς εξελισσόμενο περιβάλλον των ψηφιακών ηλεκτρονικών, ο σχεδιασμός και η προσομοίωση ψηφιακών συστημάτων διαδραματίζουν καθοριστικό ρόλο στη διαμόρφωση της τεχνολογικής προόδου της εποχής μας. Καθώς η ζήτηση για ολοένα πιο πολύπλοκα και αποτελεσματικά ηλεκτρονικά συστήματα αυξάνεται σημαντικά, η ανάγκη για ισχυρές μεθοδολογίες στον ψηφιακό σχεδιασμό είναι ύψιστης σημασίας. Η ταχεία ανάπτυξη της τεχνολογίας, που χαρακτηρίζεται από την ενσωμάτωση ποικίλων λειτουργιών σε συμπαγείς ηλεκτρονικές συσκευές, απαιτεί τη βαθιά κατανόηση του σχεδιασμού ψηφιακών συστημάτων.

Σε αυτήν την αδιάκοπη επιδίωξη της καινοτομίας λοιπόν, το «οικοδόμημα» των ψηφιακών ηλεκτρονικών, αποτελεί ακρογωνιαίο λίθο, πιέζοντας συνεχώς τα όρια αυτού που είναι τεχνολογικά εφικτό. Στην καρδιά αυτού του δυναμικού τοπίου βρίσκεται η περίπλοκη διαδικασία σχεδιασμού ψηφιακών συστημάτων, μια προσπάθεια που απαιτεί ακρίβεια, αποτελεσματικότητα και προσαρμοστικότητα. Αυτή η διπλωματική εργασία ξεκινά ένα ταξίδι στον κόσμο της VHDL (VHSIC Hardware Description Language), βασικό παράγοντα στον τομέα του ψηφιακού σχεδιασμού, με έμφαση στην εφαρμογή του στην προσομοίωση συστημάτων και απώτερο σκοπό την υλοποίηση μιας Αριθμητικής – Λογικής Μονάδας, χρησιμοποιώντας τις κατάλληλες μεθοδολογίες σχεδίασης.

1.1 Αντικείμενο της διπλωματικής εργασίας

Η παρούσα διπλωματική εργασία εμβαθύνει στο πεδίο σχεδιασμού ψηφιακών συστημάτων με έμφαση στη VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language), μια ισχυρή γλώσσα για μοντελοποίηση και προσομοίωση ψηφιακών κυκλωμάτων, η οποία έχει αναδειχθεί ως ακρογωνιαίος λίθος σε αυτόν τον τομέα. Η σημαντικότητα της εκτείνεται σε διάφορους τομείς, όπως η βιομηχανία αυτοκινήτων, οι τηλεπικοινωνίες, τα ηλεκτρονικά είδη ευρείας κατανάλωσης και όχι μόνο.

1.2 Σκοπός και στόχοι

Σκοπός της διπλωματικής εργασίας είναι να διερευνήσει τις μεθοδολογίες και τις αρχές του σχεδιασμού ψηφιακών συστημάτων χρησιμοποιώντας VHDL και αξιοποιώντας τεχνικές προσομοίωσης να παρουσιάσει την υλοποίηση ενός ολοκληρωμένου ψηφιακού κυκλώματος. Χρησιμοποιώντας τη VHDL ως την κύρια γλώσσα περιγραφής υλικού, η μελέτη θα εμβαθύνει στη δημιουργία ενός αποτελεσματικού και αξιόπιστου ψηφιακού κυκλώματος. Η εργασία περιλαμβάνει τη μοντελοποίηση της λογικής σύνθεσης ψηφιακών σχεδίων και την ολοκληρωμένη προσομοίωση

χρησιμοποιώντας προτυποποιημένα βιομηχανικά εργαλεία, συγκεκριμένα το EDA Playground Editor.

Οι βασικοί στόχοι της συγκεκριμένης εργασίας παρουσιάζονται εν συντομία παρακάτω:

1. Η εις βάθος κατανόηση των βασικών αρχών και των πρακτικών σχεδιασμού της γλώσσας VHDL από τον αναγνώστη.
2. Η διερεύνηση των διάφορων μεθοδολογιών σχεδιασμού και βέλτιστων πρακτικών με απώτερο σκοπό την υλοποίηση ενός ψηφιακού συστήματος.
3. Η υλοποίηση και η ανάλυση μίας αριθμητικής – λογικής μονάδας (ALU), με χρήση της γλώσσας VHDL.

1.3 Μεθοδολογία

Η υλοποίηση της εργασίας βασίστηκε στην παρακάτω μεθοδολογία επτά φάσεων:

1. Έρευνα - Συλλογή δεδομένων.

   Γενικότερη έρευνα για τις βασικές αρχές των Ψηφιακών Συστημάτων, τα πλεονεκτήματα και τους περιορισμούς της χρήσης τους. Στην συνέχεια κατηγοριοποίηση υπαρχουσών συστημάτων βάσει διάφορων παραμέτρων, με σκοπό την απόκτηση μιας σφαιρικής εικόνας σχετικά με το ζητούμενο της εργασίας που είναι η υλοποίηση ενός Επεξεργαστή με χρήση της γλώσσας VHDL. Τέλος συλλογή αξιόπιστων δεδομένων από papers και επιστημονικά site μέσω κατάλληλων μηχανών αναζήτησης επικεντρωμένα στα ζητούμενα του θέματος.

2. Διασταύρωση και κατηγοριοποίηση πληροφοριών.

   Επαλήθευση της ορθότητας της συλλεγόμενης πληροφορίας, διασταυρώνοντας την αξιοπιστία των πηγών προέλευσης της και συγκρίνοντας τα δεδομένα με εκείνα άλλων αξιόπιστων πηγών.

3. Υλοποίηση του Επεξεργαστή και ολοκλήρωση του τεχνικού μέρους της εργασίας.

4. Σχεδιασμός Δομής της Διπλωματικής εργασίας.

   Σχεδιασμός της δομής της διπλωματικής εργασίας, με τέτοιο τρόπο ώστε να είναι ευανάγνωστη, λογικά κατανεμημένη η καταγεγραμμένη πληροφορία και ομαλή η μετάβαση μεταξύ των κεφαλαίων, σύμφωνα με το περιεχόμενο τους.

5. Επεξεργασία της συγκεντρωμένης πληροφορίας και αντιστοίχιση της στα κεφάλαια της Διπλωματικής.

   Ομαδοποίηση, επεξεργασία και διαχωρισμός της συγκεντρωμένης πληροφορίας βάσει των

κεφαλαίων στα οποία θα ενταχθούν.

6. Διεξαγωγή Συμπερασμάτων από την υλοποίηση του επεξεργαστή και την ερευνητική διαδικασία.

   Παρατήρηση και επεξεργασία των δεδομένων που προέκυψαν από την ερευνητική διαδικασία και την υλοποίηση του Επεξεργαστή. Στην συνέχεια καταγραφή των συμπερασμάτων της εργασίας βάσει της επεξεργασμένης πληροφορίας που συλλέχθηκε.

7. Συγγραφή διπλωματικής και παρουσίαση αποτελεσμάτων.

   Συγγραφή της διπλωματικής σύμφωνα με τα παραπάνω στοιχεία.

1.4 Καινοτομία

Η καινοτομία της συγκεκριμένης εργασίας έγκειται στο γεγονός ότι σχεδιάζει ανεξάρτητες δομές κώδικα, μικρής κλίμακας. Αυτές οι δομές μπορούν να συντεθούν μεταξύ τους και να δημιουργήσουν πολυπλοκότερες και πιο σύνθετες δομές. Αυτό το χαρακτηριστικό στον προγραμματισμό είναι γνωστό ως κλιμάκωση (scalable).

Μέσα από το συνολικό αποτέλεσμα μπορούν να προκύψουν χρήσιμα συμπεράσματα και ιδέες για μελλοντικές έρευνες – σχεδιασμούς επεξεργαστών.

Αρχικά ένας βελτιστοποιημένος επεξεργαστής μπορεί να περιλαμβάνει παράλληλη επεξεργασία, καθώς και προηγμένα σύνολα εντολών. Η βελτιστοποίηση αυτή μπορεί να επιτευχθεί μέσω κατάλληλων εφαρμογών και διεπαφών. Για παράδειγμα η χρησιμοποίηση μιας εφαρμογής κρυπτογράφησης, θα μπορούσε να ενισχύσει τα αδύναμα σημεία ασφαλείας ενός ψηφιακού συστήματος. Η αποθήκευση – διαχείριση των κλειδιών και των πρωτοκόλλων ασφαλείας θα γινόταν εξολοκλήρου από την εφαρμογή.

Επιπλέον θα μπορούσε να δημιουργηθεί μια προσαρμοσμένη διεπαφή για να εξυπηρετεί κάποιο συγκεκριμένο σκοπό. Ένα τέτοιο παράδειγμα είναι η ανάπτυξη ενός πρωτοκόλλου επικοινωνίας ή ενός αισθητήρα για την απόκτηση δεδομένων.

Ακόμα η ενεργειακή απόδοση ενός επεξεργαστή είναι άλλο ένα επίκαιρο ζήτημα, το οποίο απασχολεί όλο και περισσότερο την επιστημονική κοινότητα. Ιδιαίτερο ενδιαφέρον παρουσιάζει ο σχεδιασμός του ρολογιού για τον επεξεργαστή με τέτοιο τρόπο, ώστε να μειωθεί η κατανάλωση της ενέργεια του, χωρίς να υποβαθμιστεί η απόδοσή του.

Τέλος η συμμετοχή των μικρό-επεξεργαστών στα δίκτυα αισθητήρων (IoT), είναι πλέον πραγματικότητα. Η παρούσα εργασία θα μπορούσε να συμβάλλει στην πυροδότηση ιδεών για ένα

σχεδιασμό τέτοιου είδους, καθώς τα δίκτυα αυτά απαντώνται συχνά σε πολλές εφαρμογές της καθημερινής ζωής.

1.5 Δομή

Η συγγραφή της εργασίας βασίστηκε στη δομή που ακολουθεί, η οποία αποτελεί ένα ολοκληρωμένο πλαίσιο για την παρουσίαση της έρευνάς, σχετικά με το σχεδιασμό και την προσομοίωση ψηφιακών συστημάτων με χρήση VHDL.

Περίληψη

Σύντομη περίληψη του περιεχομένου της εργασίας, των βασικών ευρημάτων και της σημαντικότητας της.

Κεφάλαιο 1: Εισαγωγή

Σε αυτό το κεφάλαιο παρουσιάζεται το αντικείμενο της εργασίας, καθώς επίσης ο σκοπός και οι στόχοι της. Αναφέρονται η μεθοδολογία που ακολουθήθηκε για την συγγραφή της και αναλύεται η σημαντικότητα της συγκεκριμένης έρευνας για τον κλάδο των Ψηφιακών Συστημάτων, τόσο σε ακαδημαϊκό, όσο και σε βιομηχανικό επίπεδο. Τέλος στο εισαγωγικό κεφάλαιο παρουσιάζεται και η δομή της εργασίας.

Κεφάλαιο 2: Ψηφιακά Συστήματα

Το Κεφάλαιο 2, ξεκινά με ιστορική αναδρομή στις πρώτες εμφανίσεις των προσωπικών υπολογιστών και κατ' επέκταση στις βασικές αρχές λειτουργίας τους. Αναλύονται κάποια από τα θεμελιώδη συστατικά της σύνθεσης τους, με βασικότερο τη Γλώσσα Μηχανής. Ακολουθεί ανάλυση των Ολοκληρωμένων Κυκλωμάτων , των τεχνικών σχεδιασμού τους και παρουσίαση των πλεονεκτημάτων τους. Στο τελευταίο υποκεφάλαιο ερευνώνται οι Γλώσσες Περιγραφής Υλικού.

Κεφάλαιο 3: Η γλώσσα VHDL

Βασικό θέμα του συγκεκριμένου κεφαλαίου είναι ο τρόπος λειτουργίας της γλώσσας VHDL. Παρουσιάζεται η δομή του κώδικα σε VHDL, καθώς επίσης και τα δομικά στοιχεία της γλώσσας, όπως οι τύποι δεδομένων, οι μεταβλητές, οι σταθερές και οι χρήσιμες συνθήκες από τις οποίες αποτελείται.

Κεφάλαιο 4: Σχεδίαση επεξεργαστή ενός κύκλου

Παρουσίαση και ανάλυση των στοιχείων που συντελούν στον σχεδιασμό ενός επεξεργαστή ενός κύκλου. Τέτοια στοιχεία είναι ο Μετρητής Προγράμματος, ο Δείκτης και καταχωρητής εντολών, οι μνήμες Cache και RAM, η Μονάδα Διαχείρισης Μνήμης, η Αριθμητική Λογική Μονάδα (ALU), ο Ελεγκτής, το σύστημα Αποκωδικοποίησης εντολών, η Μονάδα Καταχωρητών και η Βαθμίδα εκτέλεσης εντολών. Κάθε ένα από αυτά τα στοιχεία, αντιστοιχεί και παρουσιάζεται σε ένα υποκεφάλαιο

Κεφάλαιο 5: Διαγράμματα και ερμηνεία του κώδικα ακολουθούμενη από πρακτική υλοποίηση στο Mojo v.3

Σε αυτήν την ενότητα επεξηγούνται τα τμήματα του VHDL κώδικα που χρησιμοποιήθηκε για την διεκπεραίωση αυτής της εργασίας. Πραγματοποιείται η τεχνική ανάλυση των μεμονωμένων τμημάτων που αποτελούν τον επεξεργαστή. Δίνεται επίσης η ερμηνεία των αποτελεσμάτων που προκύπτουν από τη μελέτη των ψηφιακών διαγραμμάτων. Στόχος όλων των διαγραμμάτων που παρουσιάζονται είναι η επιβεβαίωση της σωστής λειτουργίας κάθε κυκλώματος. Τέλος, η επαλήθευση των θεμελιωδών στόχων του κυκλώματος γίνεται με υλοποίηση της αρχιτεκτονικής σε παρόμοια μορφή με χρήση γλώσσας Verilog στο Mojo v.3/

Κεφάλαιο 6: Συμπεράσματα

Συμβολή της εργασίας στο βιομηχανικό και τον ακαδημαϊκό κόσμο.

Κεφάλαιο 7: Βιβλιογραφία

Παραπομπές για όλες τις πηγές που αναφέρονται στην εργασία.

Παραρτήματα

Συμπληρωματικό υλικό, αποσπάσματα κώδικα, πρόσθετα δεδομένα κλπ.

2 Ψηφιακά συστήματα

Τα ψηφιακά συστήματα απαντώνται παντού γύρω μας στη σημερινή εποχή και η συνεχής εξέλιξη της τεχνολογίας έχει καταστήσει άμεση προτεραιότητα την κατανόηση του τρόπου λειτουργίας τους. Ως ψηφιακό σύστημα νοείται κάθε ηλεκτρονικό σύστημα του οποίου τα σήματα μπορούν να πάρουν μόνο 2 διακριτές τιμές, για παράδειγμα 0 ή 1. Κατά συνέπεια μπορούν να επεξεργαστούν ένα ευρύ σύνολο διακριτών πληροφοριών, σε αντίθεση με τα αναλογικά ηλεκτρονικά συστήματα, όπου εκεί οι πληροφορίες είναι συνεχής με τη μορφή των σημάτων. Κάθε ψηφιακό σύστημα, προκειμένου να εκτελέσει τις απαραίτητες διεργασίες, είναι απαραίτητο να συνδυάσει το λογισμικό (software) με το υλικό (hardware).

2.1 Προσωπικός Υπολογιστής & Bit

Οι προσωπικοί υπολογιστές αποτελούν μια μορφή ψηφιακών συστημάτων, με μόνη διαφορά ότι χρησιμοποιούν το δυαδικό σύστημα αρίθμησης αντί για το δεκαδικό. Το δυαδικό σύστημα αρίθμησης αναπαριστά αριθμητικές τιμές χρησιμοποιώντας τα ψηφία 0 και 1, ενώ έχει ως βάση το 2. Παρακάτω δίνεται ως παράδειγμα η αναπαράσταση της αριθμητικής τιμής 73 στο δυαδικό σύστημα:

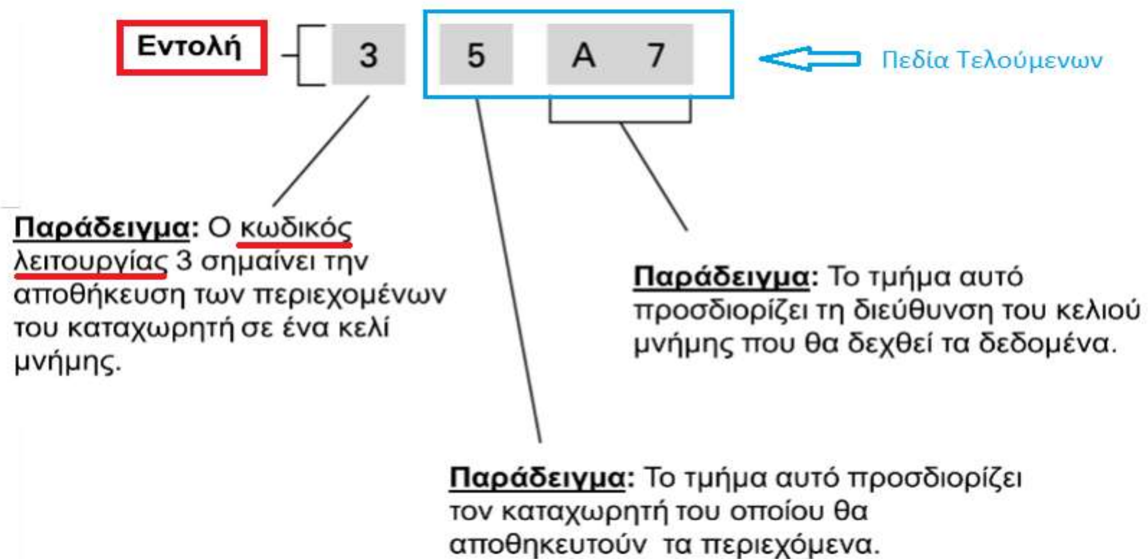
$$(73)_{10} = (1001001)_2 = 1*2^6+0*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0$$

Το bit ή αλλιώς δυαδικό ψηφίο είναι η ποσότητα της πληροφορίας που μπορεί να αποθηκευτεί και να επεξεργαστεί από μια συσκευή διακριτών καταστάσεων, όπως για παράδειγμα ένας προσωπικός υπολογιστής. Το όνομα bit προέρχεται από τις λέξεις binary digit. Όπως αναφέρθηκε και παραπάνω η ποσότητα αυτή, συγκεκριμένα στην επιστήμη των υπολογιστών, μπορεί να υπάρχει σε δύο διακριτές καταστάσεις, το 0 ή το 1.

2.2 Η γλώσσα μηχανής

Τα ηλεκτρονικά συστήματα ενός υπολογιστή, προκειμένου να λειτουργήσουν σωστά, απαιτείται να πάρουν και τις κατάλληλες εντολές. Οι εντολές αυτές δίνονται από το χρήστη και είναι απαραίτητο να «μεταφραστούν» πριν φτάσουν στον τελικό αποδέκτη. Ως γλώσσα μηχανής αναφέρονται όλες οι εντολές που αναγνωρίζονται και εκτελούνται από τον ηλεκτρονικό υπολογιστή. Παρακάτω παρουσιάζονται συνοπτικά οι κατηγορίες των εντολών που διακρίνονται με βάση το περιεχόμενό τους:

- <u>Εντολές Μεταφοράς Δεδομένων:</u> επιτρέπουν την επικοινωνίας μεταξύ των καταχωρητών.
- <u>Αριθμητικές Εντολές:</u> περιλαμβάνουν τις εντολές της πρόσθεσης, της αφαίρεσης, καθώς και άλλων αριθμητικών πράξεων.
- <u>Εντολές Λογικών Πράξεων:</u> εκτελούνται μεταξύ των δεδομένων των καταχωρητών και αφορούν τις λογικές πράξεις (AND, NOR, OR, NOT, κλπ.).
- <u>Εντολές Ελέγχου της Ροής του Προγράμματος:</u> οι εντολές αυτές μπορούν να αλλάξουν τη ροή του προγράμματος. Κάθε εντολή που εκτελείται μετά από αυτές, δεν είναι εντολή του προγράμματος.
- <u>Εντολές Εισόδου / Εξόδου:</u> πρόκειται για τις εντολές που μεταφέρουν δεδομένα από τις διατάξεις εισόδου / εξόδου προς / από τον επεξεργαστή.



Εικόνα 2.1: Πεδία Εντολών Γλώσσας Μηχανής

Όλες οι εντολές διακρίνονται σε πεδία, στα οποία περιέχονται ο κωδικός λειτουργίας, οι διευθύνσεις των καταχωρητών, καθώς και άλλες πληροφορίες, όπως ο αριθμός ολίσθησης. Πάντα στο πρώτο πεδίο γράφεται ο κωδικός λειτουργίας, ο οποίος είναι μια ακολουθία 6 bit. Με βάση αυτόν καθορίζεται και ο αριθμός των υπόλοιπων πεδίων. Ο κωδικός λειτουργίας διαβάζεται από την κρυφή μνήμη του επεξεργαστή και με αυτό τον τρόπο γίνεται γνωστό ποιες εντολές πρέπει να εκτελεστούν. Οι τιμές που παίρνει σχετίζονται με την εντολή που πρέπει να εκτελεστεί. Οι επιλογές δίνονται παρακάτω:

LOAD: 000001

STORE: 000010

ADD: 000011

SUB: 000111

MUL: 001000

BRE: 001010

SHIFT: 001100

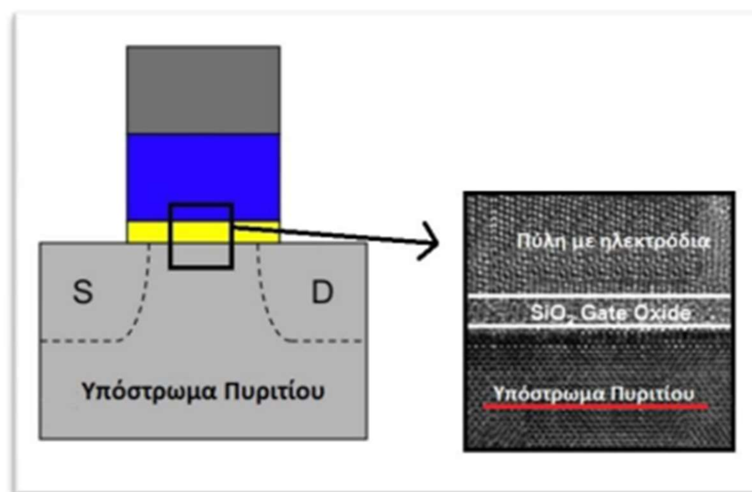
AND: 001101

JUMP: 010001

Τα υπόλοιπα πεδία της εντολής περιέχουν τις διευθύνσεις των καταχωρητών, καθώς και άλλες χρήσιμες πληροφορίες οι οποίες πρέπει να ληφθούν υπόψη από τον επεξεργαστή. Ο αριθμός των πεδίων που θα έχει η κάθε εντολή, ποικίλει με βάση των τύπο της. Για παράδειγμα εντολές πρόσθεσης ή πολλαπλασιασμού συνήθως έχουν περισσότερα πεδία από μια απλή εντολή μεταφοράς δεδομένων μεταξύ δύο καταχωρητών.

2.3 Ολοκληρωμένα κυκλώματα

Η υλοποίηση των ψηφιακών συστημάτων γίνεται με τα ολοκληρωμένα κυκλώματα, τα οποία περιέχουν τα λογικά κυκλώματα. Η εξέλιξη της τεχνολογίας έχει επηρεάσει σημαντικά τον τρόπο δημιουργίας αυτών των κυκλωμάτων. Πριν από μια δεκαετία η κατασκευή των λογικών κυκλωμάτων βασιζόταν σε ανεξάρτητα κομμάτια, τα οποία περιείχαν τρανζίστορ (Εικόνα 2.2). Η εικόνα που ακολουθεί αφορά την τυπική δομή ενός τρανζίστορ το οποίο επιτρέπει τη μεταφορά φορτίων από την πηγή (Source) προς τον απαγωγό (Drain), μέσω ενός καναλιού. Η διάταξη αυτή μπορεί να βρίσκεται μέσα σε κάποιο ημιαγωγό. Η σύνδεση των τρανζίστορ μεταξύ τους δημιουργούσε τα ολοκληρωμένα κυκλώματα. Η ονομασία chip περιγράφει τα πρώτα ολοκληρωμένα κυκλώματα ενός τεμαχίου.



Εικόνα 2.2: Εσωτερική δομή Τρανζίστορ

Η εξέλιξη αυτής της τεχνολογίας έγινε με την προσθήκη όλων των απαραίτητων στοιχείων για τη δημιουργία ενός μικροεπεξεργαστή στο ίδιο chip. Τα πρώτα χρόνια η ισχύς των μικροεπεξεργαστών ήταν πολύ μικρή και αποτελούνταν από λίγα τρανζίστορ. Ωστόσο οι ρυθμοί ανάπτυξης των ολοκληρωμένων κυκλωμάτων ήταν εντυπωσιακοί, γεγονός που οδήγησε τον Gordon Moore, ιδρυτή και πρόεδρο της εταιρείας Intel, στην πρόβλεψη ότι τα τρανζίστορ που χωράνε σ' ένα ολοκληρωμένο κύκλωμα μπορούν να διπλασιάζονται κάθε 2 χρόνια [1]. Η πρόβλεψη αυτή επαληθεύτηκε και σήμερα είναι γνωστή ως «Νόμος του Moore». Θεωρείται ότι ο Νόμος του Moore θα επαληθεύεται μέχρι τα τέλη της δεκαετίας αυτής, ενώ οι μελλοντικές έρευνες πάνω στο νόμο συνεχίζονται ακόμη και σήμερα [21].

Στον άξονα y του παρακάτω διαγράμματος φαίνεται ο αριθμός τον τρανζίστορ που περιλαμβάνονται πάνω σε κάθε chip, ενώ στον άξονα x δίνονται τα χρονολογικά έτη ανά δύο. Η γραμμικότητα του διαγράμματος επιβεβαιώνει το συνεχόμενο διπλασιασμό, τόσο των τιμών στον άξονα y, όσο και των τιμών στον άξονα x.

Σήμερα κάθε chip είναι δυνατόν να περιέχει δισεκατομμύρια τρανζίστορ και ο αριθμός αυτός συνεχώς αυξάνεται. Ωστόσο ο νόμος του Moore τείνει να μην μπορεί να επιβεβαιωθεί πλέον, γιατί ο χρόνος που χρειάζεται για να ενσωματωθεί διπλάσιος αριθμός κυκλωμάτων σε ένα chip ξεπερνάει τα 2 έτη.

Διάγραμμα 2.1: Νόμος του Moore

2.4 Σχεδίαση ολοκληρωμένων ψηφιακών συστημάτων

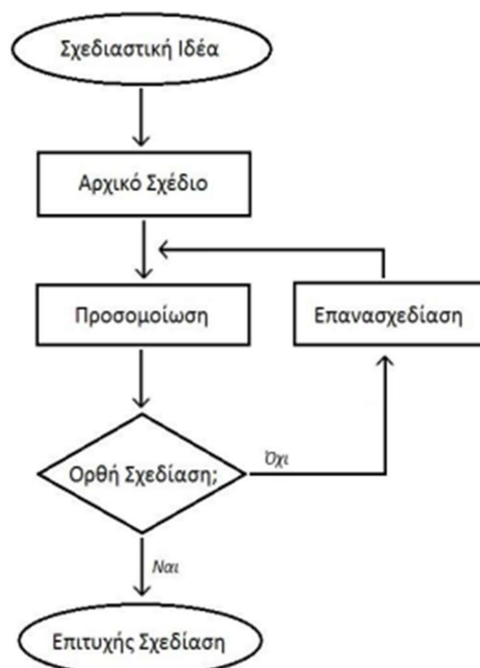
Βασική αρχή της σχεδίασης είναι η τοποθέτηση των λογικών κυκλωμάτων σ’ ένα chip, και κατ’ επέκταση η σχεδίαση και η τοποθέτηση πολλών ολοκληρωμένων κυκλωμάτων σε μια πλακέτα τυπωμένου κυκλώματος (Printed Circuit Board, PCB). Η εικόνα που ακολουθεί απεικονίζει μια τέτοια πλακέτα, πάνω στην οποία συγκεντρώνονται ολοκληρωμένα κυκλώματα.

Εικόνα 2.3: PCB πλακέτα

Μετέπειτα, προς διευκόλυνσή τους οι σχεδιαστές λογικών κυκλωμάτων, προχώρησαν στην υλοποίηση τους, χρησιμοποιώντας ήδη έτοιμα που υπήρχαν στην αγορά. Η απλοποίηση αυτή μπορεί να μείωσε το χρόνο παραγωγής τους, ωστόσο έφερε στο προσκήνιο και ορισμένα ζητήματα.

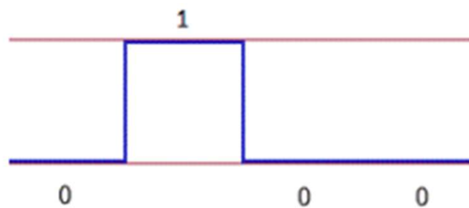
Το πρώτο ζήτημα αφορά τον προσδιορισμό, από το σχεδιαστή, της λειτουργίας του κυκλώματος και στη συνέχεια τη χρησιμοποίηση και την ανάπτυξη του. Η διαδικασία αυτή είναι γνωστή ως ανάλυση του λογικού κυκλώματος. Το δεύτερο ζήτημα αφορά τη διαδικασία της σύνθεσης, όπου ο σχεδιαστής πρέπει να υλοποιήσει εκ του μηδενός ένα κύκλωμα, καθώς δεν υπάρχει κάποιο ήδη έτοιμο που να εκτελεί τις απαιτούμενες λειτουργίες. Συνήθως η διαδικασία της ανάλυσης είναι πιο προσιτή για έναν σχεδιαστή, ωστόσο δεν είναι πάντα εφικτή η εφαρμογή της.

Στη φάση της σχεδίασης είναι απαραίτητο από τους σχεδιαστές να τηρήσουν ορισμένες προθεσμίες, να πετύχουν το μικρότερο κόστος υλοποίησης, προκειμένου να είναι προσιτό στην αγορά το τελικό προϊόν, καθώς και να εντοπίσουν τυχόν λάθη που υπάρχουν στη διαδικασία. Ενδεικτικά στο παρακάτω διάγραμμα απεικονίζονται τα βασικά βήματα σχεδίασης ψηφιακών κυκλωμάτων που πρέπει να ακολουθούνται από τους σχεδιαστές.



Διάγραμμα 2.2: Σχεδίαση ψηφιακών συστημάτων

Δυαδικά ψηφιακά κυκλώματα ονομάζονται αυτά που δέχονται σήματα δύο διακριτών τιμών. Αυτές οι διακριτές τιμές είναι το δυαδικό 0 και το δυαδικό 1. Με άλλα λόγια δέχονται ψηφιακά σήματα, όπως αυτά της ακόλουθης εικόνας.



Διάγραμμα 2.3: Κυματομορφή ψηφιακού σήματος

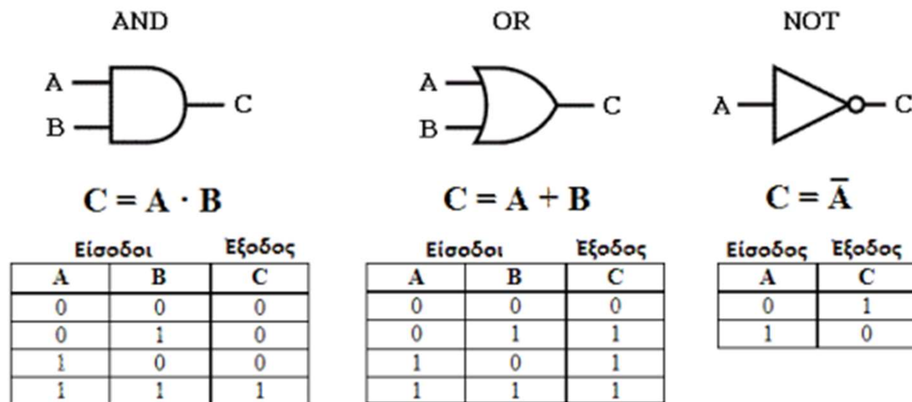
Ένα ακόμα βασικό χαρακτηριστικό της σχεδίασης λογικών κυκλωμάτων, εκτός από τα σήματα, είναι οι λογικές πύλες (logic gates) και τα δίκτυα πυλών. Τα δικτυώματα δημιουργούνται συνδυάζοντας μεμονωμένες λογικές πύλες. Αυτός ο συνδυασμός οδηγεί στην υλοποίηση πολύπλοκων λογικών συναρτήσεων.

Κάθε λογική πύλη μπορεί να έχει μία ή περισσότερες εισόδους, αλλά υποχρεωτικά μία έξοδο η οποία είναι το αποτέλεσμα των πράξεών της. Όλες οι τιμές των εισόδων και των εξόδων αποθηκεύονται σε δυαδικές μεταβλητές και μπορούν να αναπαρασταθούν στον πίνακα αλήθειας (truth table). Με πιο απλά λόγια ο προαναφερόμενος πίνακας αποτελεί μια απεικόνιση της αντιστοιχίας των εισόδων – εξόδων.

Οι 3 βασικές πύλες δίνονται παρακάτω και ο συνδυασμός τους μπορεί να οδηγήσει σε νέες πύλες ή ακόμα και σε λογικά κυκλώματα:

- Σύζευξη – AND
- Διάζευξη – OR
- Λογική άρνηση – NOT

Παρακάτω δίνεται ο πίνακας αλήθειας και η μαθηματική συνάρτηση για κάθε μία από τις λογικές πύλες που ήδη αναφέρθηκαν. Για κάθε περίπτωση λαμβάνονται δύο είσοδοι.



Είσοδοι		Έξοδος
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Είσοδοι		Έξοδος
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Είσοδος	Έξοδος
A	C
0	1
1	0

Εικόνα 2.4: Λογικές πύλες

2.5 Πλεονεκτήματα ψηφιακών συστημάτων

Τα ψηφιακά συστήματα χρησιμοποιούνται ευρέως στις τεχνολογίες υλικού καθώς διαθέτουν σημαντικά πλεονεκτήματα. Τα βασικότερα εξ’ αυτών παρουσιάζονται παρακάτω:

- Η μικρή ευαισθησία τους παρέχει ακρίβεια και αξιοπιστία.
- Η λογική σχεδίαση καθιστά ευκολότερο το σχεδιασμό τους.
- Είναι εφικτός ο προγραμματισμός τους.
- Η υπολογιστική ισχύς είναι αυξημένη, λόγω των υψηλών συχνοτήτων λειτουργίας.

Επιτυγχάνεται υψηλή απόδοση με χαμηλό κόστος.

2.6 Γλώσσες περιγραφής υλικού

Η μοντελοποίηση των ψηφιακών συστημάτων γίνεται μέσα από τις γλώσσες περιγραφής υλικού (Hardware Description Languages – HDLs) , οι οποίες σε πολλές περιπτώσεις θυμίζουν τις γλώσσες προγραμματισμού. Ο κώδικας που γράφεται τηρεί ορισμένους συντακτικούς και γραμματικούς κανόνες και είναι εφικτό να μοντελοποιήσει από μια πύλη, ως ένα περίπλοκο λογικό κύκλωμα. Η ειδοποιός τους διαφορά με τις γλώσσες προγραμματισμού είναι ότι δεν αποσκοπούν στην περιγραφή – προσομοίωση λογισμικού.

Οι γλώσσες περιγραφής υλικού επιτρέπουν την ιεραρχική σχεδίαση που χρησιμοποιείται στο υλικό και αφορά κυκλώματα, τα οποία συνθέτονται από απλούστερα – μικρότερα κυκλώματα. Επιπλέον μπορούν να υποστηρίζουν την παραλληλία και την ταυτόχρονη εκτέλεση κάποιων στοιχείων, που απαρτίζουν τα ψηφιακά συστήματα. Αντίθετα οι γλώσσες προγραμματισμού ακολουθούν μια ακολουθιακή εκτέλεση των εντολών.

Η ευαισθησία που παρουσιάζουν στο χρόνο είναι μια ακόμα σημαντική διαφορά μεταξύ των γλωσσών περιγραφής υλικού και λογισμικού. Ο χρονισμός των διαφόρων στοιχείων, στην περίπτωση της

περιγραφής υλικού, αποτελεί κρίσιμο παράγοντα για το εξαγόμενο αποτέλεσμα. Η λειτουργία του κυκλώματος προσομοιάζει τη λειτουργία ενός κυκλώματος σε κάποιο πραγματικό υλικό. Αυτό σημαίνει ότι η εκτέλεση των λειτουργιών μπορεί, είτε να γίνεται ακαριαία, είτε να εμπεριέχει κάποιες καθυστερήσεις. Ειδικά σήματα εισόδου χρησιμοποιούνται όταν είναι επιθυμητή η εισαγωγή καθυστερήσεων.

Οι βασικότερες λειτουργίες μιας γλώσσας περιγραφής υλικού είναι η προσομοίωση (simulation) και η σύνθεση (synthesis) ενός κυκλώματος. Ο όρος προσομοίωση αναφέρεται στη διαδικασία πιστοποίησης της συμπεριφοράς ενός κυκλώματος, με τη χρήση ενός προσομοιωτή (simulator). Οι προσομοιωτές είναι κατάλληλα λογισμικά τα οποία έχουν δημιουργηθεί γι’ αυτό το σκοπό. Δέχονται την περιγραφή του κυκλώματος, λαμβάνουν τα κατάλληλα σήματα εισόδου και παράγουν τις λογικές τιμές εξόδου του κυκλώματος. Με αυτό τον τρόπο ελέγχονται όλα τα κυκλώματα, πριν προχωρήσουν στην πραγματική τους υλοποίηση και παραγωγή.

Από την άλλη πλευρά, ο όρος σύνθεση αναφέρεται στην αυτοματοποιημένη διαδικασία παραγωγής του δικτυώματος που αποτελεί το κύκλωμα. Τα ειδικά εργαλεία που επιτελούν αυτό το σκοπό δέχονται ως είσοδο την περιγραφή του κυκλώματος και τις συνδέσεις των στοιχείων και ως έξοδο το δικτύωμα με πύλες.

Οι πιο διαδεδομένες γλώσσες περιγραφής υλικού είναι η Verilog και η VHDL. Οι δύο γλώσσες αμφότερες αποτελούν πρότυπα του IEEE (Institute of Electrical and Electronics Engineers), χρησιμοποιούνται ευρέως στη βιομηχανία και υποστηρίζονται από όλες τις εταιρείες που κατασκευάζουν ψηφιακό εξοπλισμό.

Η κώδικας της Verilog είναι συνοπτικός, καθώς διαθέτει ένα μικρό σύνολο στοιχείων για την περιγραφή των λογικών κυκλωμάτων. Η αποτελεσματική περιγραφή του υλικού την καθιστά γλώσσα χαμηλότερου επιπέδου. Ωστόσο η SystemVerilog είναι μια νεότερη επέκταση της, η οποία αφενός βελτιώνει κάποιες από τις υπάρχουσες ατέλειες της αρχικής γλώσσας, αφετέρου εισάγει κάποια νέα χαρακτηριστικά. Η νέα αυτή επέκταση ανταγωνίζεται τη VHDL, που θα αναλυθεί στην επόμενη ενότητα.

3 Η γλώσσα VHDL

Η γλώσσα VHDL (Very High Speed Integrated Circuit Hardware Description Language) εμφανίστηκε πρώτη φορά το 1987 με το πρότυπο IEEE 1076-1987. Το 1993 δημιουργήθηκε μια βελτιωμένη έκδοση της με το πρότυπο IEEE 1164-1993. Η βασική έκδοση της VHDL (VHDL 4.0) που κυριαρχεί μέχρι και σήμερα, προτάθηκε το 2008 από την τεχνική επιτροπή της Accellera και τελικά δημοσιοποιήθηκε το 2009 με το πρότυπο IEEE 1076-2008. Η πιο σύγχρονη έκδοση της VHDL είναι το πρότυπο IEEE 1076-2019.

Είναι μια ντετερμινιστική και πλούσια σε στοιχεία γλώσσα και υποστηρίζει τον προγραμματισμό του κυκλώματος σε υψηλότερο επίπεδο. Διαθέτει αναλυτικό κώδικά, ο οποίος έχει την ικανότητα να αναδεικνύει σφάλματα τα οποία η Verilog αγνοεί. Το βασικότερο πλεονέκτημα της και ένας από τους βασικούς λόγους της μακροζωίας της είναι η φορητότητά της (portability). Αυτό σημαίνει πως ο κώδικας δεν εξαρτάται από την τεχνολογία υλοποίησης τους ψηφιακού κυκλώματος, γεγονός που τον καθιστά ευρέως συμβατό. Κατ' αυτόν τον τρόπο ο σχεδιαστής είναι ικανός να επικεντρωθεί αποκλειστικά στη λειτουργική σχεδίαση του κυκλώματος.

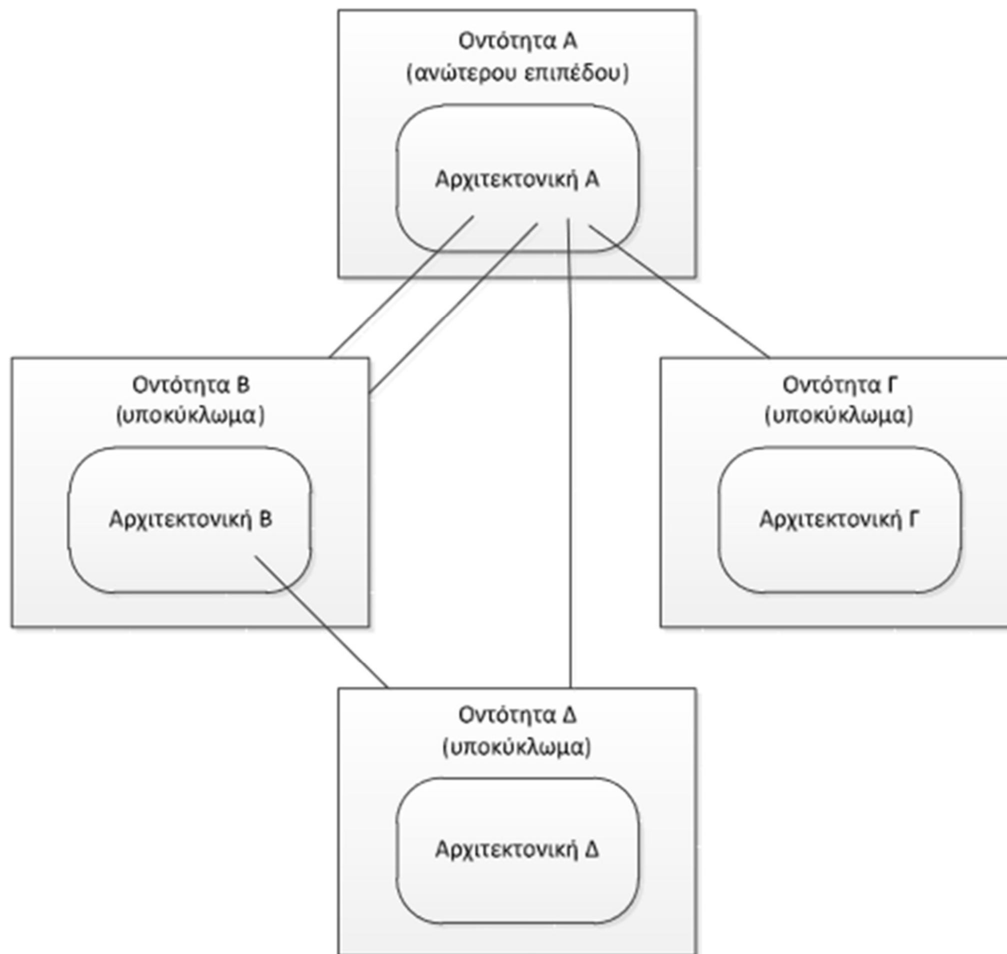
3.1 Ο τρόπος λειτουργίας της VHDL

Η VHDL ανήκει στις παράλληλες γλώσσες προγραμματισμού, και το γεγονός αυτό επιβεβαιώνεται από το ότι κρατάει από τη γλώσσα Ada, η οποία επιτρέπει τον προγραμματισμό παράλληλων διεργασιών. Επιπλέον υπάρχει η δυνατότητα να αναπτυχθούν δομημένα κυκλώματα, καθώς υπακούει στις αρχές του δομημένου προγραμματισμού. Το πλεονέκτημα της είναι ότι μπορεί να υλοποιήσει, αφενός τις λειτουργίες κυκλωμάτων, αφετέρου τους χρόνους στους οποίους παράγονται αποτελέσματα από τις λειτουργίες.

Ο κώδικας της VHDL είναι από προεπιλογή ταυτόχρονος (concurrent). Αυτό σημαίνει ότι όλες οι λειτουργίες εκτελούνται ταυτόχρονα, εκτός και αν υπάρχει κάποια διαφορετική οδηγία. Κάθε κομμάτι του κώδικα περιγράφει λειτουργίες, οι οποίες παράγουν αποτελέσματα ταυτόχρονα μαζί με άλλες λειτουργίες. Με άλλα λόγια αυτό σημαίνει ότι η σειρά με την οποία τοποθετούνται τα τμήματα του κώδικα δεν επηρεάζει τις τελικές εξόδους. Ο ταυτόχρονος κώδικας μπορεί να χρησιμοποιηθεί για την υλοποίηση συνδυαστικών κυκλωμάτων, όπως αθροιστές, πολυπλέκτες και κωδικοποιητές. Αυτά είναι κυκλώματα των οποίων η έξοδοι εξαρτώνται μόνο από την αρχικά δεδομένη είσοδο.

Διευκρινίζεται ωστόσο, ότι η VHDL μπορεί να περιγράψει και ακολουθιακές λογικές λειτουργίες. Αυτό πρακτικά σημαίνει ότι υπάρχουν κάποια συμβάντα, όπως οι παλμοί του ρολογιού τα οποία συγχρονίζονται και παράγουν το ζητούμενο αποτέλεσμα. Ο διαδοχικός κώδικας χρησιμοποιείται όταν η προηγούμενη είσοδος επηρεάζει την έξοδο. Συνδυαστικά κυκλώματα μεγαλύτερης πολυπλοκότητας χρησιμοποιούν το διαδοχικό κώδικα. Συνοψίζοντας γίνεται κατανοητό ότι η σύνταξη του κώδικα μπορεί

να γίνει, είτε με σύγχρονες δομές κώδικα (concurrent code), είτε με ακολουθιακές δομές (sequential code).



Διάγραμμα 3.1: Ιεραρχική σύνδεση οντοτήτων

Ο διαχωρισμός του κώδικα μεταξύ οντότητας και αρχιτεκτονικής, όπως θα αναλυθεί σε επόμενη ενότητα, επιτρέπει τον ιεραρχικό τρόπο σχεδίασης στη VHDL. Αυτό σημαίνει ότι κάποιο κύκλωμα που έχει σχεδιαστεί, μπορεί να ενσωματωθεί σε ένα περισσότερο πολύπλοκο, απλώς με μια αναφορά της οντότητας του, δίχως να χρειάζεται να περιγραφεί εξαρχής η αρχιτεκτονική του.

Όπως φαίνεται και στο παραπάνω διάγραμμα της ιεραρχικής σχεδίασης, κάθε κύκλωμα που ανώτερου επιπέδου χρησιμοποιεί κυκλώματα που βρίσκονται σε κατώτερα επίπεδα. Για παράδειγμα, η οντότητα Α χρησιμοποιεί τις οντότητες Β, Γ και Δ, ενώ η οντότητα Β χρησιμοποιεί την οντότητα Δ. με βάση τις αρχές της ιεραρχικής σχεδίασης δεν είναι δυνατόν μια κατώτερη οντότητα (π.χ Δ) να χρησιμοποιεί – καλεί μια ανώτερη (π.χ Γ). Προκειμένου να επιτευχθεί αυτή η ιεραρχία δημιουργούνται κατάλληλα instances (στιγμιότυπα), ο αριθμός των οποίων δεν έχει κάποιον περιορισμό. Με τον τρόπο αυτό η περιγραφή του υλικού (hardware) καθίσταται επαναχρησιμοποιήσιμη (reusable), γεγονός που προσδίδει ευελιξία αντίστοιχη με του λογισμικού (software). Η επαναχρησιμοποίηση του υλικού επιτρέπει στα κυκλώματα να

διαμοιράζονται σε πολλούς χρήστες μέσω βιβλιοθηκών. Με αυτό τον τρόπο ενισχύεται η δυνατότητα ανάπτυξης μεγάλων συστημάτων, τα οποία κάθε φορά προσαρμόζονται στις ανάγκες του κάθε σχεδιαστή.

3.2 Αριθμοί και χαρακτήρες στη VHDL

Οι πληροφορία που μεταφέρεται από τα σήματα και τις μεταβλητές της γλώσσας περιέχει αριθμητικό περιεχόμενο. Στις περισσότερες περιπτώσεις η πληροφορία αυτή έχει, είτε τη μορφή ακέραιων, είτε τη μορφή δεκαδικών τιμών. Επιπλέον υποστηρίζονται, τόσο προσημασμένοι, όσο και μη προσημασμένοι αριθμοί.

Οι ακέραιοι αριθμοί παριστάνονται στο δεκαδικό σύστημα, ωστόσο μπορούν να χρησιμοποιηθούν και άλλα συστήματα όπως το δυαδικό και το δεκαεξαδικό. Για τις 2 τελευταίες περιπτώσεις η αναπαράσταση του αριθμού γίνεται βάζοντας πρώτα τον αριθμό της βάσης και στη συνέχεια τον αριθμό περιβαλλόμενος από το σύμβολο της δίεσης. Για παράδειγμα η αναπαράσταση του δεκαδικού αριθμού 14 στο δυαδικό σύστημα της VHDL είναι 2#1110#. Αντίστοιχα η αναπαράσταση του δεκαδικού 5000 στο δεκαεξαδικό σύστημα της VHDL είναι 16# 1388#.

Για την αναπαράσταση των δυαδικών τιμών χρησιμοποιούνται τα μονά ή τα διπλά εισαγωγικά. Όταν μια τιμή περιλαμβάνει πολλά bits, τότε προτιμάται η δεκαεξαδική απεικόνιση.

Η VHDL μπορεί να χρησιμοποιεί, τόσο προσημασμένους, όσο και μη προσημασμένους αριθμούς. Οι μη προσημασμένοι αριθμοί, είναι με απλά λόγια οι μη αρνητικοί και η περιοχή των τιμών τους είναι από 0 έως 2^{N}-1. Αντίθετα η περιοχή των τιμών στους προσημασμένους αριθμούς είναι από -2^{N-1} έως 2^{N-1}-1. Σε αυτούς τους αριθμούς το πρόσημο παριστάνεται από το σημαντικότερο bit.

Τέλος σημειώνεται ότι οι χαρακτήρες της γλώσσας μπορούν να δίνονται ως τιμές σε μεταβλητές, είτε με απλή μορφή (π.χ “k” ή “u”), είτε να συνθέτουν ακολουθίες χαρακτήρων, δηλαδή πιο σύνθετες μορφές, γνωστές ως strings. Τέτοιο παραδείγματα είναι οι ακολουθίες “first” και “arrow”.

3.3 Δομή κώδικα

Στην ενότητα αυτή θα αναφερθούν οι κανόνες σύνταξης και η βασική δομή ενός VHDL προγράμματος. Ο κώδικας της κατά κύριο λόγο αποτελείται από 3 μέρη διακριτά μεταξύ τους. Αυτά είναι η δήλωση των βιβλιοθηκών (Libraries), η οντότητα (Entity) και η αρχιτεκτονική (Architecture). Το σχήμα που ακολουθεί παρουσιάζει αυτά τα 3 μέρη.

Με τη δήλωση των βιβλιοθηκών (Libraries) και των πακέτων (Packages) επιτρέπεται η χρήση («κάλεσμα») τμημάτων κώδικα που έχουν ήδη κατασκευασθεί στο παρελθόν και χρησιμοποιούνται συχνά. Πρόκειται επομένως για χρήσιμα τμήματα κώδικα που τοποθετούνται σε μια βιβλιοθήκη και μπορούν εύκολα να ενσωματωθούν και σε μελλοντικούς κώδικες, απλώς με την αναφορά της συγκεκριμένης βιβλιοθήκης στην αρχή του κώδικα.

Δήλωση Βιβλιοθηκών: Στο πρώτο μέρος του κώδικα συμπεριλαμβάνονται όλες οι βιβλιοθήκες που θα χρησιμοποιηθούν στο κυρίως μέρος. Οι συνηθέστερες χρησιμοποιούμενες βιβλιοθήκες είναι η IEEE και η std. Η βιβλιοθήκη IEEE έχει συγκεκριμένα υπό-τμήματα, τα οποία ονομάζονται πακέτα, τα οποία πρέπει κάθε φορά να καθορίζονται, προκειμένου να μπορούν να επιτελέσουν συγκεκριμένες λειτουργίες. Η βιβλιοθήκη std χρειάζεται σε λειτουργίες εισόδου – εξόδου κειμένου, καθώς και στον καθορισμό του τύπου των δεδομένων. Επιπλέον πάντα υπάρχει και μια βιβλιοθήκη εργασίας, όπου αποθηκεύονται όλα τα αρχεία κώδικα. Τέλος αναφέρεται ότι ο σχεδιαστής μπορεί να δημιουργήσει και τα δικά του πακέτα και να τα συμπεριλάβει στο πρόγραμμα. Παρακάτω δίνονται 2 παραδείγματα εισαγωγής βιβλιοθηκών:

`library ieee;` → δήλωση βιβλιοθήκης

`use ieee.std_logic_1164.all;` → εισαγωγή του πακέτου std_logic_1164

Οντότητα: Στη δήλωση της οντότητας ορίζονται όλα τα σήματα εισόδου και εξόδου του σχεδιαζόμενου κυκλώματος και δεν περιλαμβάνονται στοιχεία που φανερώνουν τη λειτουργία του. Το όνομα της οντότητας συνήθως συμπίπτει με το όνομα του αρχείου για λόγους διευκόλυνσης. Κάθε σήμα μπορεί να λειτουργεί ως είσοδος (in), έξοδος (out), είσοδος / έξοδος (inout) ή ως έξοδος η οποία μπορεί να διαβαστεί και εσωτερικά της οντότητας (buffer). Κάτι το οποίο είναι ανέφικτο και δεν επιτρέπεται είναι οι θύρες εξόδου να βρίσκονται στο δεξί μέλος μιας εντολής ανάθεσης. Με άλλα λόγια δεν μπορεί η κατάστασή τους να εκχωρηθεί σε κάποιο άλλο σήμα. Πολύπλοκές καταστάσεις που δημιουργούνται εξαιτίας αυτής της αδυναμίας, επιλύονται με τη χρήση εσωτερικών σημάτων. Παρακάτω δίνεται η σύνταξη της οντότητας:

```
entity ENTITY_NAME is
port [signal_input_name: in data_type,
signal_output_name: out data_type];
end ENTITY_NAME;
```

Αρχιτεκτονική: Αυτό είναι το κυρίως μέρος του κώδικα, όπου τα δεδομένα διαβάζονται από ένα αρχείο, στη συνέχεια επεξεργάζονται κατάλληλα, προκειμένου να προκύψουν τα επιθυμητά αποτελέσματα, και τέλος αυτά εξάγονται σε ένα αρχείο τύπου txt. Πριν από τον κυρίως κώδικα τα σήματα που χρησιμοποιούνται πρέπει να δηλωθούν, καθώς και ο τύπος των δεδομένων. Κάθε αρχιτεκτονική αναφέρεται σε μία μοναδική οντότητα. Αντίθετα μια οντότητα μπορεί να παρουσιάζεται σε περισσότερες από μία αρχιτεκτονικές.

```
Architecture Design of ENTITY_NAME is
signal declaration;
type declaration;
components declaration;
Begin
```

```
        statements;
End Design
```

Η γλώσσα VHDL δεν είναι τόσο ευαίσθητη κατά την εγγραφή της λέξης “Begin”, είτε γραφεί “Begin”, είτε “BEGIN”, είτε “BegiN”, δεν επηρεάζει τη μεταγλώττιση (compile).

Το κυρίως μέρος της αρχιτεκτονικής μπορεί να σχεδιαστεί σε 3 επίπεδα. Αυτά είναι το επίπεδο συμπεριφοράς (behavioral), το επίπεδο ροής δεδομένων (dataflow) και το δομικό επίπεδο (structural).

Επίπεδο Συμπεριφοράς: η συμπεριφορά του κυκλώματος περιγράφεται και μοντελοποιείται σε γενικό – αφηρημένο επίπεδο (abstract), χωρίς να υπάρχει αναλυτική περιγραφή των λογικών εξισώσεων. Συνηθέστερη χρησιμοποιούμενη τεχνική είναι η process, όπου εντός αυτής η εκτέλεση όλων των εντολών γίνεται ακολουθιακά. Πιο συγκεκριμένα, υπάρχει μια λίστα με σήματα και κάθε φορά που συμβαίνει μια μεταβολή στην κατάσταση ενός σήματος εκτελείται η process.

Επίπεδο Ροής Δεδομένων: τα δεδομένα μεταβαίνουν από την είσοδο προς την έξοδο και στο σύστημα εκτελούνται παράλληλες εντολές. Μπορεί να υλοποιηθεί, είτε χρησιμοποιώντας απλές εντολές εκχώρησης σημάτων, είτε χρησιμοποιώντας εντολές εκχώρησης σημάτων υπό συνθήκη. Σε αρχιτεκτονικές που περιγράφονται με το συγκεκριμένο επίπεδο, καμία εντολή δεν μπορεί να βρίσκεται μέσα σε process.

Δομικό Επίπεδο: το σύστημα σχεδιάζεται ιεραρχικά. Τα επιμέρους δομικά στοιχεία που το συνθέτουν, ορίζονται ξεχωριστά και συνδέονται μεταξύ τους προκειμένου να συνθέσουν το συνολικό κύκλωμα. Κάθε δομική μονάδα απαιτείται αν δηλωθεί πριν από το begin, όπως δηλώνονται και οι οντότητες.

3.4 Λεκτικά στοιχεία

Στον VHDL κώδικα χρησιμοποιούνται λεκτικά στοιχεία, πρόκειται για ειδικά διαμορφωμένους αλφαριθμητικούς χαρακτήρες, μέσω των οποίων επιτυγχάνεται η ανάθεση τιμών, η ονοματοδοσία τμημάτων κώδικα, η σύνταξη εντολών και ο ορισμός τελεστών πράξεων. Ως λεκτικά στοιχεία αναγνωρίζονται και κάποιες δεσμευμένες λέξεις, οι οποίες αποτελούν λέξεις-κλειδιά για τον VHDL κώδικα. Μερικές από αυτές είναι: LIBRARY, ENTITY, ARCITECTURE, IF, ELSE, WHILE, WHEN, SELECT, WAIT, VARIABLE, SIGNAL, CONSTANT κ.α. οι λέξεις αυτές έχουν συγκεκριμένη σημασία, και δεν μπορούν να χρησιμοποιηθούν με διαφορετικό τρόπο από αυτό που έχουν ήδη οριστεί.

Χρησιμοποιώντας αντίστοιχες λεκτικές διατυπώσεις, πρέπει να δηλωθούν και τα σήματα που χρησιμοποιούνται για τη μεταφορά πληροφορίας από – προς το κύκλωμα, ή ακόμα και μέσα στο ίδιο κύκλωμα. Με άλλα λόγια όλες οι οντότητες και τα σήματα της γλώσσας ονοματίζονται κατάλληλα, χρησιμοποιώντας κάποιο αναγνωριστικό.

Λόγω της ανελαστικότητας κατά τη μεταγλώττιση της γλώσσας, απαιτείται οι συντακτικές διατυπώσεις να είναι αυστηρές, προκειμένου να αποφεύγονται δυσλειτουργίες.

Ιδιαίτερη σημασία έχει το ελληνικό ερωτηματικό (;) το οποίο τοποθετείται στο τέλος κάθε γραμμής και δηλώνει την ολοκλήρωση μιας VHDL εντολής. Συνηθίζεται επίσης τα τμήματα του κώδικα να διακρίνονται μεταξύ τους, χρησιμοποιώντας διακεκομμένες γραμμές. Ακόμα κάθε γραμμή που στην αρχή της περιέχει δυο ή περισσότερες παύλες θεωρείται σχόλιο και δεν λαμβάνεται υπόψη κατά τη μεταγλώττιση.

3.5 Η «σύγχρονη» εντολή SELECT

Η λειτουργία που περιγράφει η εντολή SELECT υλοποιείται άμεσα, ανεξάρτητα από το σημείο του κώδικα στο οποίο βρίσκεται. Οι αναθέσεις των τιμών ανανεώνονται ταυτόχρονα για όλες τις σύγχρονες εντολές που υπάρχουν σε έναν κώδικα. Όπως έχει αναφερθεί και παραπάνω η VHDL επιτρέπει το συγχρονισμό, κατά την εκτέλεση των διεργασιών, ανεξάρτητα από τη θέση των εντολών στον κώδικα. Για παράδειγμα αν ένα σήμα στην αρχή του κώδικα τροποποιείται από μια λειτουργία που βρίσκεται στο τέλος του, τότε αυτό θα πρέπει να ενημερωθεί μετά την ολοκλήρωση της λειτουργίας. Οι αλλαγές των σημάτων θα σταματήσουν, όταν όλα τα σήματα σταθεροποιηθούν. Κάθε κύκλος προσομοίωσης ολοκληρώνεται όταν εκτελεστούν όλες οι προβλεπόμενες διεργασίες. Αν υπάρξει τροποποίηση κάποιων σημάτων, τότε ξεκινάει ένας νέος κύκλος προσομοίωσης με την επανάληψη όλων των διεργασιών.

3.6 Η δομή PROCESS

Ορισμένες λειτουργίες στα ψηφιακά κυκλώματα επιτελούνται μόνο σε συγχρονισμό με συγκεκριμένα συμβάντα. Αυτό πρακτικά συμβαίνει στα ακολουθιακά κυκλώματα. Με πιο απλά λόγια οι αναθέσεις των εξόδων γίνονται μόνο κατά τις μεταβάσεις σημάτων. Υπάρχει επομένως η ανάγκη για ύπαρξη δομών που ανανεώνουν τις τιμές των σημάτων μόνο σε συγκεκριμένες χρονικές στιγμές. Μια τέτοια δομή είναι η PROCESS, η οποία ανανεώνει τις αναθέσεις των τιμών των σημάτων, μόνο όταν αλλάξουν οι τιμές των σημάτων που συμπεριλαμβάνονται στη λίστα ευαισθησία της.

Ο χρήστης αντιλαμβάνεται ότι η PROCESS διεργασία αναθέτει τιμές στα σήματα με σύγχρονο τρόπο, με τη λεπτομέρεια ότι οι αναθέσεις αυτές ανανεώνονται μόνο κατά τις μεταβάσεις των σημάτων που περιλαμβάνονται στη λίστα ευαισθησία της. Κατά τη σύνθεση του κυκλώματος η PROCESS διεργασία μεταφράζεται από το εργαλείο σχεδίασης ως ένα ακολουθιακό κύκλωμα.

3.7 Αντικείμενα και τύποι δεδομένων στη VHDL

Παρακάτω θα περιγραφούν τα βασικά αντικείμενα δεδομένων που χρησιμοποιούνται στη σύνταξη του κώδικα. Ακόμα θα περιγραφούν και οι βασικοί τύποι δεδομένων – βιβλιοθήκες που απαντώνται σε ορισμένα αντικείμενα.

3.7.1 Αντικείμενα – objects δεδομένων

Συνολικά υπάρχουν 3 αντικείμενα δεδομένων (data objects) που μπορούν να μεταφέρουν πληροφορία μέσα στο σύστημα και να αποδίδουν τιμές σε διάφορα σημεία. Αυτά είναι τα σήματα (SIGNALS), οι μεταβλητές (VARIABLES) και οι σταθερές (CONSTANTS). Κάθε αντικείμενο λαμβάνει και ένα συγκεκριμένο όνομα, το οποίο ορίζεται με βάση κάποιων τύπο δεδομένων (data types). Επίσης σημειώνεται ότι τα

αντικείμενα περιλαμβάνουν πάντα μια τιμή και δεν είναι ποτέ κενά. Ενημερωτικά αναφέρεται ότι τα αρχεία (FILES) είναι το τέταρτο αντικείμενο δεδομένων, αλλά δεν εξετάζονται στα πλαίσια αυτής της εργασίας.

Τα σήματα είναι ιδιαίτερα σημαντικά και μπορούν να χρησιμοποιηθούν, τόσο σε τμήματα με σύγχρονο κώδικα, όσο και σε τμήματα με ακολουθιακό. Πρακτικά χρησιμοποιούνται για να αποδώσουν τιμές στα καλώδια του κυκλώματος και αποτελούν τις ενώσεις ανάμεσα σε διαφορετικές μονάδες κυκλωμάτων.

Οι μεταβλητές χρησιμοποιούνται για την αποθήκευση προσωρινών τιμών που προκύπτουν από την εκτέλεση αριθμητικών ή λογικών πράξεων. Χρησιμοποιούνται σε τμήματα του κώδικα που περιλαμβάνουν ακολουθιακές εντολές. Τα τμήματα αυτά του κώδικα δηλώνονται ως διεργασίες (PROCESS), όπως αναφέρθηκε και παραπάνω. Συνήθως περιγράφουν ακολουθιακά κυκλώματα, όπως είναι οι καταχωρητές. Εκτός από μια διεργασία, μια μεταβλητή μπορεί να χρησιμοποιηθεί και σε ένα υποπρόγραμμα. Οι μεταβλητές επηρεάζουν πρακτικά το κύκλωμα, όταν σε αυτές αποδίδεται κάποιο σήμα.

Στην περίπτωση των σταθερών η τιμή τους παραμένει αμετάβλητη, καθ' όλη τη λειτουργία του κυκλώματος. Αυτό πρακτικά σημαίνει ότι η τιμή που δηλώνεται αρχικά, παραμένει σταθερή. Γίνεται επομένως κατανοητό ότι οι τιμές αυτές δεν αντιπροσωπεύουν κάποιο καλώδιο του συστήματος, παρά μόνο αποθηκεύουν αριθμητικές τιμές.

3.7.2 Δήλωση αντικειμένων

Όλα τα παραπάνω αντικείμενα απαιτείται να δηλωθούν (declarations) προκειμένου να χρησιμοποιηθούν μέσα στον κώδικα. Για κάθε αντικείμενο ορίζεται το όνομα του και ο τύπος του. Στην περίπτωση των σταθερών ορίζεται και η τιμή τους, ενώ προαιρετικά το ίδιο μπορεί να γίνει, τόσο για τα σήματα, όσο και για τις μεταβλητές. Η ονοματοδοσία των αντικειμένων γίνεται ακολουθώντας ορισμένους κανόνες. Συνολικά επιτρέπεται η χρησιμοποίηση όλων των αλφαριθμητικών χαρακτήρων (πεζών & κεφαλαίων), καθώς και η χρησιμοποίηση του underscore (_). Τα ονόματα δεν γίνεται να ξεκινάνε και να τελειώνουν με «_», καθώς επίσης δεν γίνεται να ξεκινάνε και με κάποιον αριθμό. Επιπλέον δεν μπορούν να χρησιμοποιηθούν συνεχόμενα 2 underscore χαρακτήρες «__». Τέλος το όνομα δεν μπορεί να ταυτίζεται με κάποια λέξη κλειδί, για παράδειγμα ENTITY. Τέλος διευκρινίζεται ότι στη VHDL δεν μπορεί να υπάρχει διάκριση ανάμεσα στα πεζά και στα κεφαλαία γράμματα.

Παρακάτω φαίνεται πως δηλώνονται οι σταθερές, τα σήματα και οι μεταβλητές:

- CONSTANT όνομα: τύπος: τιμή;
- SIGNAL όνομα: τύπος;
- VARIABLE όνομα: τύπος;

Τα σήματα μπορούν να δηλωθούν σε τρία διαφορετικά σημεία μέσα στον κώδικα. Αυτά είναι στη δήλωση της οντότητας (entity), στη δήλωση της αρχιτεκτονικής, καθώς και στη δήλωση των βιβλιοθηκών – πακέτων (libraries & packages).

3.7.3 Τύποι δεδομένων

Στη γλώσσα VHDL διακρίνονται οι προκαθορισμένοι τύποι δεδομένων (predefined data types) και αυτοί που δημιουργούνται από τους χρήστες (user defined data types). Οι προκαθορισμένοι τύποι είναι προτυποποιημένοι και οι βιβλιοθήκες τους υπάρχουν στα σχεδιαστικά εργαλεία. Αυτό σημαίνει ότι μπορούν εύκολα να χρησιμοποιηθούν, καλώντας τα πακέτα που τους περιγράφουν. Στα πλαίσια αυτής της εργασίας θα εξεταστούν μόνο οι προκαθορισμένοι τύποι δεδομένων.

Οι τύποι των δεδομένων είναι σημαντικοί και πρέπει να γίνεται κατανοητό πως πρέπει να χρησιμοποιούνται. Για παράδειγμα στα σήματα ο τύπος των δεδομένων είναι αυτός που καθορίζει τις τιμές που μπορεί να λάβει το σήμα, καθώς και τις πράξεις που υποστηρίζει. Ο τύπος των δεδομένων που χρησιμοποιείται συνδέεται και με τους αντίστοιχους τελεστές. Οι τελεστές θα αναφερθούν σε επόμενη υποενότητα αυτού του κεφαλαίου.

3.7.4 Προκαθορισμένοι τύποι δεδομένων

Ο βασικότερος τύπος δεδομένων ανήκει στην αρχική προτυποποίηση της γλώσσας (βιβλιοθήκη std) και χρησιμοποιείται, χωρίς να χρειάζεται αναφορά σε βιβλιοθήκες. Οι τύποι αυτοί είναι οι ακόλουθοι:

BIT: ο τύπος αυτός αποτελείται από ένα bit, δηλαδή τα σήματα μπορούν να λάβουν τις τιμές «0» ή «1». Υποστηρίζει λογικούς και σχεσιακούς τελεστές και δηλώνεται ως εξής: *SIGNAL star : bit ;*

BIT_VECTOR: ο τύπος αυτός ορίζεται ως πίνακας μιας διάστασης (array), που περιέχει BIT στοιχεία. Το τελικό αποτέλεσμα αποτελείται από οκτώ στοιχεία και καλείται διάνυσμα. Υποστηρίζει λογικούς και σχεσιακούς τελεστές, καθώς και τελεστές συνένωσης και ολίσθησης. Μπορεί να οριστεί ως εξής: *SIGNAL star : bit_vector (7 downto 0) ;*

INTEGER: τα σήματα αυτά μεταφέρουν ακέραιες τιμές και από το χρήστη γίνονται κατανοητά ως βαθμωτά μεγέθη. Αντίθετα από τη σκοπιά των σημάτων οι ακέραιες τιμές περιλαμβάνουν ένα σύνολο από bits. Έχουν μήκος 32 bits και το εύρος τιμών τους είναι από $-(2^{31}-1)$ έως $(2^{31}-1)$. Για παράδειγμα μια μεταβλητή με αρχική τιμή το 5 και με όρια τιμών το -64 και το 64, δηλώνεται με τον ακόλουθο τρόπο:

variable number : integer range -64 to 64 :=5;

Διευκρινίζεται ότι το εύρος των τιμών μπορεί να αλλάξει, χρησιμοποιώντας τη λέξη RANGE. Αν τα όρια των ακέραιων τιμών δεν προσδιοριστούν, τότε ο μεταγλωτιστής προσδίδει αυτόματα εύρος 32 bit. Οι ακέραιοι τύποι υποστηρίζουν αριθμητικές πράξεις και πράξεις σύγκρισης.

NATURAL: πρακτικά πρόκειται για υποκατηγορία των ακέραιων τύπων, καθώς περιλαμβάνει μη αρνητικούς αριθμούς. Υποστηρίζει τις ίδιες πράξεις με τους ακέραιους τύπους, δηλώνεται ως εξής: *signal k : natural range 10 to 20 ;*

BOOLEAN: πρόκειται για βαθμωτό τύπο δεδομένων που μπορεί να πάρει μόνο την τιμή TRUE ή FALSE. Υποστηρίζει λογικούς και σχεσιακούς τελεστές.

CHARACTER: οι τιμές που λαμβάνουν αυτά τα σήματα προέρχονται από ένα σύνολο 256 χαρακτήρων των 8 bit. Οι χαρακτήρες έχουν εύρος 8 bits, ωστόσο ο τύπος αυτός θεωρείται βαθμωτός γιατί διαχειρίζονται κάθε φορά έναν χαρακτήρα. Υποστηρίζει σχεσιακούς τελεστές, ενώ οι πίνακες χαρακτήρων υποστηρίζουν και τη συνένωση.

TIME: ο τύπος αυτός προορίζεται μόνο για προσομοιώσεις και όχι για σύνθεση. Οι ακέραιοι που ορίζονται, παριστάνουν χρονικές στιγμές. Υποστηρίζει αριθμητικούς και σχεσιακούς τελεστές.

3.7.5 Τύποι standard logic

Οι τύποι std_logic και std_logic_vector αποτελούν συνήθως τη βάση για τις περισσότερες σχεδιάσεις, ενώ πρόκειται για πρότυπα της ηλεκτρονικής βιομηχανίας. Οι τύποι αυτοί ορίζονται στο πακέτο std_logic_1164 που προστέθηκε στο πρότυπο IEEE 1174 το 1993. Έχει αναφερθεί σε προηγούμενη υποενότητα αυτού του κεφαλαίου, το πώς γίνεται η δήλωση αυτής της βιβλιοθήκης και του προτύπου std_logic_1164.

Ένα σήμα του τύπου std_logic μπορεί να πάρεις τις τιμές 0 και 1, όπως και ένα bit. Ωστόσο μπορεί να λάβει και κάποιες ακόμα τιμές που αντιπροσωπεύουν ορισμένες καταστάσεις. Οι τιμές αυτές είναι:

- «Z» → υψηλή εμπέδηση
- «-» → αδιάφορη κατάσταση
- «U» → μη αρχικοποιημένη κατάσταση
- «X» → άγνωστη κατάσταση
- «L» → ασθενής χαμηλή
- «H» → ασθενής υψηλή
- «W» → ασθενής άγνωστη

Όσον αφορά τα σήματα std_logic_vector, διευκρινίζεται ότι μπορούν να πάρουν τις ίδιες τιμές τα σήματα std_logic, με τη διαφορά ότι αφού πρόκειται για διανύσματα αποτελούνται από έναν πίνακα τιμών.

3.8 Τελεστές και πράξεις

Στη VHDL μπορούν να εκτελεστούν αριθμητικές, λογικές και σχεσιακές πράξεις. Απαραίτητη προϋπόθεση για να γίνει αυτό είναι η χρησιμοποίηση των κατάλληλων τελεστών. Τα είδη που υποστηρίζονται είναι αριθμητικοί τελεστές, τελεστές σύγκρισης, τελεστές ολίσθησης και ο τελεστής συνένωσης. Στις επόμενες υποενότητες αναλύονται λεπτομερώς.

3.8.1 Τελεστές ανάθεσης

Αποδίδουν τιμές σε σήματα, μεταβλητές και σταθερές. Ένα τέτοιο παράδειγμα είναι *VARIABLE v4 : bit_vector (7 downto 0) ;*

Η χρήση τους επεξήγεται στον ακόλουθο πίνακα:

Τελεστής	**Λειτουργία**
<=	Αναθέτει τιμές σε σήματα
:=	Αναθέτει τιμές σε μεταβλητές Δίνει αρχικές τιμές σε σήματα κατά τη δήλωσή τους
=>	Δίνει τιμές σε στοιχεία πινάκων

Πίνακας 3.1: Τελεστές ανάθεσης

3.8.2 Λογικοί τελεστές

Χρησιμοποιούνται για την πραγματοποίηση λογικών πράξεων. Τα αποτελέσματα των λογικών πράξεων είναι του ίδιου τύπου και μεγέθους με τους τελεστέους.

Τελεστής	**Λειτουργία**
Not	Αντιστροφή
And	Και
Nand	Όχι και
Or	Ή
Nor	Ούτε
Xor	Αποκλειστικό Η
Xnor	Αποκλειστικό Ούτε

Πίνακας 3.2: Λογικοί τελεστές

Με εξαίρεση τον τελεστή Not, οι υπόλοιπες λογικές πράξεις έχουν την ίδια προτεραιότητα. Για το λόγο αυτό είναι καλό όταν σε δήλωση υπάρχουν πολλές λογικές να χρησιμοποιούνται παρενθέσεις για να εξασφαλίζονται οι προτεραιότητες.

3.8.3 Σχεσιακοί τελεστές

Πραγματοποιούν συγκρίσεις μεταξύ σημάτων ή μεταβλητών και ελέγχουν την ύπαρξη ή μη της ισότητας. Όλες οι σχεσιακές πράξεις έχουν την ίδια προτεραιότητα. Τα αποτελέσματά τους είναι πάντα TRUE ή FALSE, δηλαδή Boolean.

Τελεστής	Λειτουργία
=	Ίσο
/=	Διάφορο

<	Μικρότερο
>	Μεγαλύτερο
<=	Μικρότερο ή ίσο
>=	Μεγαλύτερο ή ίσο

Πίνακας 3.3: Σχεσιακοί τελεστές

3.8.4 Τελεστές αριθμητικών πράξεων

Στον πίνακα που ακολουθεί παρουσιάζονται οι αριθμητικοί τελεστές, ενώ στην τρίτη στήλη του πίνακα γίνεται κατανοητός ο τρόπος με τον οποίο συντάσσονται. Ως x και y ορίζονται τυχαίες μεταβλητές, ενώ τα αποτελέσματα των πράξεων εκχωρούνται στη μεταβλητή *result*.

Τελεστής	**Λειτουργία**	**Σύνταξη**
+	Πρόσθεση	result<=x + y;
-	Αφαίρεση	result<=x - y;
*	Πολλαπλασιασμός	result<=x * y;
/	Διαίρεση	result<=x / y;
**	Ύψωση σε δύναμη	result<=x ** 3;
ABS	Απόλυτη τιμή	result<=ABS y;
REM	Υπόλοιπο	result<=x REM y;
MOD	Modulo	result<=x MOD y;

Πίνακας 3.4: Αριθμητικοί τελεστές

Οι παραπάνω τελεστές χρησιμοποιούνται με προκαθορισμένους τύπους δεδομένων, είτε integer, είτε natural. Πολλαπλασιαστές και αθροιστές είναι συνηθισμένες περιπτώσεις κυκλωμάτων που χρησιμοποιούν τους παραπάνω τελεστές και επιτελούν αριθμητικές πράξεις.

3.8.5 Τελεστές ολίσθησης

Αυτοί οι τελεστές χρησιμοποιούνται σε διανυσματικούς τύπους δεδομένων (bit_vector). Με άλλα λόγια επιτρέπουν την ολίσθηση των bits. Οι τελεστές και η λειτουργία τους παρουσιάζονται στον ακόλουθο πίνακα, ενώ και πάλι με τυχαίο τρόπο ορίζονται οι μεταβλητές x και y. Το n στη σύνταξη των εντολών δηλώνει τον αριθμό των θέσεων που ολισθαίνει ο αριθμός.

Τελεστής	**Λειτουργία**	**Σύνταξη**
SLL	Λογική ολίσθηση αριστερά κατά n θέσεις. Οι θέσεις στα δεξιά γεμίζουν με 0.	y<=x SLL n
SRL	Λογική ολίσθηση δεξιά κατά n θέσεις. Οι θέσεις στα αριστερά	y<=x SRL n

	γεμίζουν με 0.	
SLA	Ολίσθηση προς τα αριστερά. Οι θέσεις στα δεξιά γεμίζουν με το δεξιότερο bit.	y<=x SLA n
SRA	Ολίσθηση προς τα δεξιά. Οι θέσεις στα αριστερά γεμίζουν με το αριστερότερο bit.	y<=x SRA n
ROL	Περιστροφική ολίσθηση προς τα αριστερά.	y<=x ROL n
ROR	Περιστροφική ολίσθηση προς τα δεξιά.	y<=x ROR n

Πίνακας 3.5: Τελεστές ολίσθησης

3.8.6 Τελεστής συνένωσης

Σκοπός του είναι να ομαδοποιεί τις τιμές, προκειμένου να επεκτείνει το εύρος ενός σήματος ή μιας πράξης ολίσθησης. Έστω ότι έχουμε τα σήματα x και y:

SIGNAL x,y : std_logic_vector (3 downto 0);

Η πράξη *y<= x(1 downto 0) & "00" ;* ισοδυναμεί με ολίσθηση κατά δύο θέσεις αριστερά.

3.9 Χρήσιμες συνθήκες

Μια βασική χρήσιμη συνθήκη συντάσσεται με την <u>εντολή SELECT</u>. Παρακάτω φαίνεται ο τρόπος με τον οποίο δομείται ο κώδικας:

```
WITH αναγνωριστικό SELECT
Ανάθεση_τιμής WHEN τιμή
Ανάθεση_τιμής WHEN τιμή
Ανάθεση_τιμής WHEN OTHERS;
```

Το αναγνωριστικό μπορεί να συμβολίζει το όνομα κάποιου σήματος. Επειδή η εντολή SELECT απαιτείται να καλύπτει όλες τις πιθανές δυνατές τιμές που μπορεί να λάβει το σήμα (αναγνωριστικό), συχνά χρησιμοποιείται η έκφραση WHEN OTHERS η οποία καλύπτει όλες τις εναπομείναντες περιπτώσεις, που δεν καλύπτονται από τις προηγούμενες εκφράσεις. Η εντολή SELECT υλοποιεί κυκλώματα με εισόδους επιλογής, όπως είναι οι πολυπλέκτες και οι κωδικοποιητές.

<u>Η συνθήκη WHEN…ELSE</u> πραγματοποιεί ανάθεση τιμής σε κάποιο σήμα, με την προϋπόθεση ότι ισχύει μια συγκεκριμένη συνθήκη. Ο τρόπος λειτουργίας της είναι παρόμοιος με την ακολουθιακή συνθήκη IF, και συντάσσεται όπως φαίνεται παρακάτω:

```
Ανάθεση_τιμής WHEN συνθήκη ELSE
```

```
Ανάθεση_τιμής WHEN συνθήκη ELSE
…
Ανάθεση_τιμής;
```

Μπορεί οι δύο τελευταίες συνθήκες να φαίνονται ίδιες, ωστόσο αυτό δεν ισχύει και υπάρχουν διαφορές μεταξύ τους. Οι συνθήκες ανάθεσης στη σύγχρονη εντολή SELECT είναι αποκλειστικές. Αυτό σημαίνει ότι δεν μπορούν να γίνουν 2 αναθέσεις τιμών σε συνθήκες που ικανοποιούνται ταυτόχρονα. Αντίθετα στη συνθήκη WHEN…ELSE υπάρχει σειρά προτεραιότητας μεταξύ των συνθηκών, και έτσι δεν αποκλείονται μεταξύ τους. Επομένως οι συνθήκες ικανοποιούνται με σειρά προτεραιότητας. Τέλος μια ακόμα σημαντική διαφορά μεταξύ τους, είναι ότι η συνθήκη WHEN…ELSE συντάσσεται ακόμα και όταν δεν καλύπτονται όλες οι περιπτώσεις (πιθανές τιμές). Αντίθετα η εντολή SELECT απαιτεί να καλύπτονται όλες οι πιθανές τιμές που μπορεί να λάβει το αναγνωριστικό (`WHEN OTHERS`).

Η συνθήκη IF επιτρέπει να οριστούν εντολές διακλάδωσης. Γράφεται μέσα σε ακολουθιακά τμήματα κώδικα, συνήθως σε κάποια διεργασία (PROCESS). Στόχος της είναι να ελέγξει πότε μια συνθήκη είναι αληθής, και όταν αυτό συμβαίνει να εκτελέσει το τμήμα του κώδικα που βρίσκεται μέσα σε αυτήν. Όταν η αρχική συνθήκη δεν είναι αληθής, προχωράει στον έλεγχο της επόμενης, μέχρι να ικανοποιηθεί κάποια από τις συνθήκες. Η σύνταξή της γίνεται, όπως φαίνεται παρακάτω:

```
IF συνθήκη THEN
      Ανάθεση_τιμής;
ELSEIF συνθήκη THEN
      Ανάθεση_τμής;
ELSEIF συνθήκη THEN
      Ανάθεση_τμής ;
ELSE
      Ανάθεση_τιμής;
ENDIF;
```

Η περίπτωση που περιγράφεται παραπάνω αφορά μια περίπλοκη περίπτωση με την ύπαρξη πολλών εμφωλευμένων ELSEIF εντολών. Η συνθήκη IF μπορεί να συνταχθεί και με απλούστερο τρόπο, όπως:

```
IF συνθήκη THEN
      Ανάθεση_τιμής;
ENDIF;
```

Η συνθήκη FOR…GENERATE χρησιμοποιεί την εντολή GENERATE και επιτρέπει σε ένα τμήμα κώδικα να επαναλαμβάνεται, με τη βοήθεια ενός δείκτη. Ο βρόχος επανάληψης που δημιουργείται μπορεί να περιλαμβάνει από απλές αναθέσεις σημάτων και αριθμητικές πράξεις, μέχρι ολόκληρα στιγμιότυπα (instances) κυκλωμάτων. Πρακτικά πρόκειται για ένα δυναμικό τρόπο δημιουργίας μεγαλύτερων κυκλωμάτων, χρησιμοποιώντας μικρότερα – απλούστερα κυκλώματα. Ο κώδικας που παράγεται είναι

συμπυκνωμένος και ισχυρότερος. Εκτός από τη συνθήκη FOR…GENERATE, υπάρχει και η συνθήκη IF…GENERATE, αλλά δεν συνηθίζεται να χρησιμοποιείται. Η συνθήκη FOR…GENERATE διατυπώνεται ως εξής:

```
FOR αναγνωριστικό IN περιοχή_τιμών GENERATE
       Τμήμα_κώδικα
END GENERATE;
```

3.10 Η εντολή WAIT

Πρόκειται για μια ακολουθιακή εντολή, η οποία χρησιμοποιείται αποκλειστικά μέσα σε διεργασίες και υποπρογράμματα. Είναι ένας εναλλακτικός τρόπος για να εισαχθούν καθυστερήσεις κατά την εκτέλεση μιας διεργασίας ή διαφορετικά να οριστούν συγκεκριμένες χρονικές στιγμές τις οποίες μια διεργασία θα ανταποκριθεί σε αλλαγές των σημάτων. Η εντολή WAIT δεν μπορεί να περιλαμβάνει λίστα ευαισθησίας, μπορεί ωστόσο να συνοδεύεται από συνθήκες, λίστες σημάτων και χρονικές εκφράσεις. Παρακάτω περιγράφονται οι τρεις τρόποι με τους οποίους μπορεί να συνταχθεί:

- WAIT UNTIL συνθήκη;
- WAIT FOR χρονική_έκφραση;
- WAIT ON λίστα_σημάτων;

3.11 Η εντολή LOOP

Η συγκεκριμένη εντολή είναι υπεύθυνη για τη δημιουργία βρόχων επανάληψης, στους οποίους υλοποιούνται πολλά στιγμιότυπα ενός κυκλώματος. Για παράδειγμα με την εντολή LOOP μπορούμε να γεμίσουμε όλες τις θέσεις ενός καταχωρητή. Αντίστοιχα σε έναν απαριθμητή, μπορεί να χρησιμοποιηθεί για την παραγωγή των διαδοχικών αυξήσεων της εξόδου, μέχρι να συμπληρωθεί ο απαραίτητος αριθμός καταστάσεων. Δεδομένου ότι η LOOP είναι μια ακολουθιακή πρόταση, πρέπει να βρίσκεται πάντα μέσα σε μια διεργασία ή κάποιο υποπρόγραμμα. Η σύγχρονη εντολή GENERATE αποτελεί την αντιστοιχία της, όσον αφορά την εκτέλεση ενός τμήματος κώδικα κατ' επανάληψη. Παρακάτω παρουσιάζονται οι μορφές με τις οποίες μπορεί να συνταχθεί:

LOOP χωρίς συνθήκη

Αυτή η συνθήκη πρακτικά δεν ικανοποιεί τις βασικές αρχές του προγραμματισμού, διότι επαναλαμβάνεται στο άπειρο. Συνήθως είναι απαραίτητο να προβλέπεται και κάποια δυνατότητα εξόδου από τον ατέρμονα βρόχο, ο οποίος επαναλαμβάνεται στο άπειρο (infinite loop).αυτή η περίπτωση της LOOP συνθήκης μπορεί να συνταχθεί ως εξής:

```
LOOP
Ακολουθιακός_κώδικας
END LOOP;
```

LOOP με εντολή EXIT

Πρακτικά η εντολή EXIT διακόπτει την εκτέλεση των εντολών μέσα στο βρόχο, και μεταφέρει τη σειρά εκτέλεσης έξω από αυτόν. Δίνεται το ακόλουθο παράδειγμα σύνταξης αυτής της συνθήκης χρησιμοποιώντας τη φυσική (natural) μεταβλητή k.

```
LOOP
      k:= k+1;
      exit when k>10;
END LOOP;
```

Η εκτέλεση των εντολών μεταφέρεται έξω από το LOOP βρόχο όταν το k πάρει τιμή μεγαλύτερη από το 10.

LOOP με FOR

Είναι ο πιο γνωστός τρόπος δημιουργίας επαναληπτικού βρόχου, ο οποίος επιτρέπει τη δημιουργία πολυπλοκότερων υποπρογραμμάτων, όπως αναφέρθηκε και παραπάνω. Συντάσσεται ως εξής:

```
FOR αναγνωριστικό IN περιοχή_τιμών LOOP
      Ακολουθιακός_κώδικας
END LOOP;
```

Το αναγνωριστικό είναι ευρέως γνωστό ως μετρητής και συνήθως ο χαρακτήρας i χρησιμοποιείται στον προγραμματισμό για αυτό το σκοπό. Πρακτικά ο μετρητής είναι υπεύθυνος για την καταγραφή των επαναλήψεων που υλοποιούνται. Όταν το αναγνωριστικό εξέλθει από την αρχικά ορισμένη περιοχή των τιμών, τότε η εκτέλεση του βρόχου ολοκληρώνεται.

LOOP με WHILE

Στην περίπτωση αυτή πριν από κάθε επανάληψη ελέγχεται η ικανοποίηση μιας συνθήκης. Κάθε φορά που η συνθήκη ικανοποιείται ο ακολουθιακός βρόχος εκτελείται. Όταν σταματήσει να ικανοποιείται ο ακολουθιακός βρόχος παρακάμπτεται και το πρόγραμμα συνεχίζει με τις επόμενες εντολές. Συντάσσεται ως εξής:

```
WHILE συνθήκη LOOP
Ακολουθιακός_κώδικας
END LOOP;
```

3.12 Η εντολή CASE

Η λειτουργίας της είναι αντίστοιχη με αυτή της συνθήκης IF, καθώς επιλέγει ποια ομάδα ακολουθιακών εντολών θα εκτελέσει, έχοντας ως κριτήριο την ικανοποίηση κάποιας συνθήκης. Η διαφορά μεταξύ τους είναι ότι η IF εξετάζει διαδοχικά την ισχύ όλων των λογικών συνθηκών, σε αντίθεση με την CASE η οποία επιλέγει με βάση την τιμή που λαμβάνει μια μοναδική έκφραση. Συντάσσεται ως εξής:

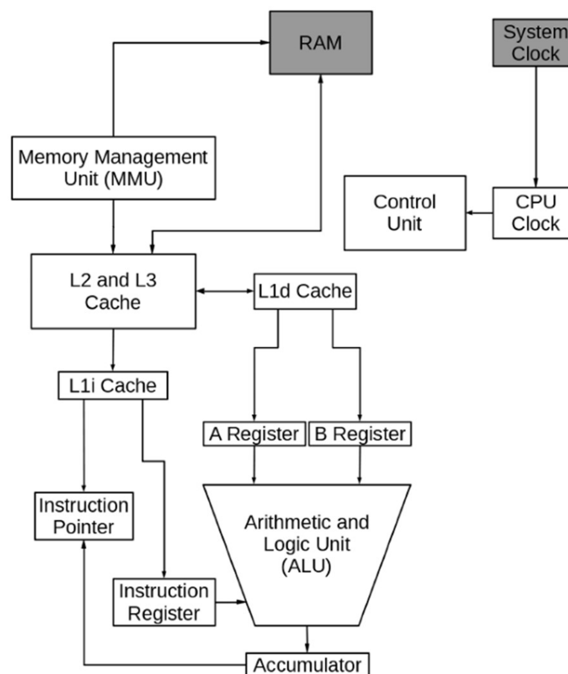
```
CASE έκφραση IS
      WHEN τιμή => ανάθεση;
      WHEN τιμή=> ανάθεση;
      WHEN OTHERS=> ανάθεση;
END CASE;
```

Η τελευταία έκφραση (WHEN OTHERS) χρησιμοποιείται, προκειμένου να ληφθούν υπόψη όλες οι δυνατές τιμές της αρχικά επιλεγμένης έκφρασης.

Ιδιαίτερα αυξημένη είναι η χρησιμότητα της εντολής CASE στην περιγραφή μηχανών πεπερασμένων καταστάσεων (finite state machines). Οι μηχανές αυτές υλοποιούνται μέσω ακολουθιακών κυκλωμάτων τα οποία χρησιμοποιούν συνδυαστικά και κάποιο καταχωρητή. Το σύστημα προκειμένου να μεταβεί στην επόμενη κατάσταση εξετάζει την τρέχουσα κατάσταση, καθώς και την τιμή του σήματος εισόδου. Μια διεργασία (PROCESS), περιλαμβάνοντας το σήμα του ρολογιού στη λίστα ευαισθησίας μπορεί να περιγράψει μια μηχανή πεπερασμένων καταστάσεων. Τα σήματα εξόδου παράγονται με τη βοήθεια σύγχρονων (concurrent) εντολών.

4 Σχεδίαση επεξεργαστή ενός κύκλου

Στην ενότητα που ακολουθεί θα παρουσιασθούν και θα αναλυθούν τα βασικότερα τμήματα – συστατικά που αποτελούν μια κεντρική μονάδα επεξεργασίας. Κάθε συστατικό πραγματοποιεί και μια διαφορετική λειτουργία και παράγει διαφορετικά σήματα εξόδου, τα οποία πρόκειται να χρησιμοποιηθούν σε επόμενες υπηρεσίες. Τα βασικότερα στοιχεία μιας CPU (Διάγραμμα 2.4) είναι ο μετρητής προγράμματος (Program Counter), η κρυφή μνήμη εντολών και δεδομένων, οι καταχωρητές, η Αριθμητική Λογική Μονάδα (Arithmetic Logic Unit), ο Controller και η βαθμίδα αποκωδικοποίησης εντολών. Το διάγραμμα που ακολουθεί δείχνει πως συνδέονται και πως επικοινωνούν τα στοιχεία αυτά μεταξύ τους. Για παράδειγμα η μνήμη RAM συνδέεται με τις μνήμες επιπέδου L2 & L3, καθώς επίσης και με τη μονάδα διαχείρισης της μνήμης.



Διάγραμμα 2.4: Typical diagram of a CPU

4.1 Μετρητής προγράμματος

Η απλούστερη λειτουργία αυτού του στοιχείου είναι η καταχώρηση, δηλαδή χρησιμοποιεί μια είσοδο για να παράγει μία έξοδο, μέσω μιας πύλης ενεργοποίησης. Όταν η πύλη ενεργοποίησης έχει την τιμή 1, τότε δίνεται ως έξοδος η νέα επεξεργασμένη τιμή. Αντίθετα όταν έχει την τιμή 0 η έξοδος θα παραμείνει όπως ήταν αρχικά. Επιπλέον διαθέτει και ένα σήμα ρολογιού, το οποίο όταν η πύλη ενεργοποίησης λαμβάνει την τιμή 1, τότε κατά τη θετική ακμή του ρολογιού γίνεται η προσκόμιση της εντολής και στην έξοδο του μετρητή παίρνουμε τη διεύθυνση της εντολής.

Ο μετρητής προγράμματος επιλύει ένα κοινό πρόβλημα στον τομέα των ψηφιακών συστημάτων. Επιτρέπει την τροποποίηση, κάθε φορά που η πύλη εξόδου πρόκειται να παράγει ένα νέο σήμα. Αυτό σημαίνει ότι θα υπάρξει αυτόματη αλλαγή στις εισόδους των ακόλουθων στοιχείων, επεξεργάζοντας τα νέα δεδομένα και δημιουργώντας τη νέα έξοδο. Τέλος είναι σημαντικό οι έξοδοι από διαφορετικά

στοιχεία να φτάνουν στο επόμενο στάδιο της επεξεργασίας στον ίδιο χρόνο, διότι διαφορετικά ενέχει ο κίνδυνος να χρησιμοποιηθούν ως είσοδοι μη ενημερωμένα στοιχεία.

4.2 Δείκτης & καταχωρητής εντολών

Ο δείκτης εντολών (Instruction Pointer) καθορίζει τη θέση στη μνήμη, όπου περιέχεται η επόμενη εντολή και πρόκειται να εκτελεστεί από τη CPU. Όταν η CPU ολοκληρώσει την εκτέλεση της τρέχουσας εντολής, η επόμενη εντολή φορτώνεται στον καταχωρητή εντολών (Instruction Register) από τη θέση μνήμης που δείχνει ο δείκτης εντολών. Μετά τη φόρτωση της εντολής, ο δείκτης του μητρώου εντολών αυξάνεται κατά μία διεύθυνση εντολής. Η αύξηση του επιτρέπει να είναι έτοιμος να μετακινήσει την επόμενη εντολή στον καταχωρητή εντολών.

4.3 Cache & RAM

Οι σύγχρονοι επεξεργαστές (CPUs) διαθέτουν περισσότερα του ενός επίπεδα μνήμης (cache). Η CPU έχει την ικανότητα να εκτελεί υπολογισμούς πολύ γρηγορότερα από την ικανότητα της μνήμης RAM να τροφοδοτεί με δεδομένα τον επεξεργαστή. Η προσωρινή μνήμη (cache) είναι ταχύτερη από τη μνήμη RAM του συστήματος, διότι βρίσκεται στο τσιπ του επεξεργαστή, επομένως πιο κοντά στη CPU. Η κρυφή μνήμη παρέχει αποθήκευση δεδομένων για να εμποδίσει την CPU να περιμένει την ανάκτηση δεδομένων από τη μνήμη RAM. Όταν η CPU χρειάζεται δεδομένα η κρυφή μνήμη καθορίζει εάν τα δεδομένα υπάρχουν ήδη και τα παρέχει στην CPU. Σημειώνεται ότι οι οδηγίες προγράμματος θεωρούνται επίσης δεδομένα και μπορούν και αυτά να παρέχονται από την cache.

Στην περίπτωση που τα ζητούμενα δεδομένα δεν βρίσκονται στη μνήμη cache, τότε ανακτώνται από τη μνήμη RAM και χρησιμοποιούνται αλγόριθμοι πρόβλεψης για τη μεταφορά περισσότερων δεδομένων από τη RAM στην κρυφή μνήμη. Ο ελεγκτής κρυφής μνήμης αναλύει τα απαιτούμενα δεδομένα και προσπαθεί να προβλέψει τα επόμενα που θα ζητηθούν από τη RAM. Φορτώνει τα δεδομένα που έχουν προβλεφθεί στην κρυφή μνήμη. Διατηρώντας ορισμένα δεδομένα πιο κοντά στη CPU σε μια cache, η CPU μπορεί να παραμείνει απασχολημένη και να μην σπαταλά τους κύκλους αναμονής για δεδομένα.

Σήμερα, ένας τυπικός υπολογιστής διαθέτει ιεραρχία μνήμης 6 επιπέδων. Πιο συγκεκριμένα:

- Registers: αυτή είναι η ταχύτερη μνήμη που μπορεί να χρησιμοποιηθεί από τον χρήστη. Θεωρητικά η πρόσβαση στους καταχωρητές είναι σχεδόν ελεύθερη, αλλά υπάρχουν μόνο λίγοι από αυτούς διαθέσιμοι στους επεξεργαστές.
- L1 Cache: Αυτό είναι το επίπεδο μνήμης πιο κοντά στον επεξεργαστή. Προσφέρει πολύ γρήγορη πρόσβαση (κοντά στον χρόνο πρόσβασης των registers) αλλά το μέγεθός της είναι συνήθως μικρό, περίπου 8 έως 64 KB.
- L2 Cache: Αυτό είναι το επίπεδο μνήμης πάνω από την μνήμη L1 και είναι κάπως πιο αργό από L1 Cache, αλλά είναι επίσης σχετικά γρήγορο. Το μέγεθός της κυμαίνεται από 256KB έως 8MB.
- L3 Cache: Αυτό είναι το επίπεδο μνήμης πάνω από το L2 Cache. Το μέγεθός της ποικίλλει μεταξύ 10 και 64 MB.

- DRAM: Αυτή είναι η κύρια μνήμη του υπολογιστή όπως όλοι γνωρίζουμε. Η DRAM είναι 10 έως 100 φορές πιο αργή από μια cache μνήμη. Σε έναν τυπικό υπολογιστή το μέγεθός του είναι μεταξύ 4 έως 128 GB.
- Μαγνητικός δίσκος: Αυτό είναι (μερικές φορές) το τελευταίο επίπεδο μνήμης σε έναν υπολογιστή. Η πρόσβαση σε δεδομένα από το δίσκο είναι 100.000 (!) πιο αργή από την πρόσβαση σε δεδομένα από DRAM.

4.4 Μονάδα διαχείρισης μνήμης

Η μονάδα αυτή (Memory Management Unit) διαχειρίζεται τη ροή των δεδομένων μεταξύ της RAM και της μνήμη του επεξεργαστή. Επιπλέον παρέχει προστασία στη μνήμη, η οποία είναι απαραίτητη σε περιβάλλοντα πολλαπλών διεργασιών (multitasking). Τέλος καθιστά δυνατή τη μετατροπή διευθύνσεων εικονικής μνήμης σε φυσικές διευθύνσεις και αντίστροφα.

4.5 Αριθμητική λογική μονάδα (ALU)

Η (ΑΛΜ) αριθμητική λογική μονάδα ή αλλιώς ALU (Arithmetic Logic Unit) είναι ένα βασικό υποσύστημα του επεξεργαστή το οποίο πράττει το μετασχηματισμό του αριθμητικού και του λογικού από απλές μέχρι και πολυσύνθετες πράξεις. Η ΑΛΜ διαθέτει αποθηκευτικούς χώρους για τελεστές και αποτελέσματα λογικών πράξεων που αποθηκεύονται σε αντίστοιχους τελεστές. Η αριθμητική λογική μονάδα χρησιμεύει ως ένα ψηφιακό συνδυαστικό κύκλωμα που εκτελεί λογικές πράξεις μεταξύ bit. Με λίγα λόγια είναι μέρος της δομής μονάδας ελέγχου και υπολογιστικών κυκλωμάτων ακόμη και των κεντρικών μονάδων επεξεργασίας (CPU) και γραφικών (GPU).

Ο μαθηματικός John von Neumann πρότεινε την έννοια της ALU το 1945 σε μια έκθεση σχετικά με τα θεμέλια για έναν νέο υπολογιστή που ονομαζόταν EDVAC. Ένας από τους πρώτους υπολογιστές που διέθεταν πολλαπλά διακριτά κυκλώματα ALU ενός bit ήταν ο Whirlwind I του 1948, ο οποίος χρησιμοποίησε δεκαέξι τέτοιες "μαθηματικές μονάδες" για να του επιτρέψει να λειτουργεί με λέξεις των 16 bit. Το 1967, η Fairchild παρουσίασε την πρώτη ALU που υλοποιήθηκε ως ολοκληρωμένο κύκλωμα, το Fairchild 3800, που αποτελείται από μια ALU 8-bit με συσσωρευτή.

Μερικά παραδείγματα πράξεων που υποστηρίζει η ALU είναι:

1. <u>Αριθμητικές πράξεις:</u>

- Πρόσθεση ή Addition

  Προσθέτει τα A και B με μεταφορά στο Y ή με άθροισμα μεταφοράς

- Αφαίρεση ή Subtraction

  Αφαιρεί το B από το A ή το αντίστροφο με τη διαφορά στο Y και τη μεταφορά ή την εκτέλεση.

- Αύξηση

  Όπου το A ή το B αυξάνεται κατά ένα και το Y αντιπροσωπεύει τη νέα τιμή.

- Μείωση

  Όπου το A ή το B μειώνεται κατά ένα και το Y αντιπροσωπεύει τη νέα τιμή.

2. Λογικές πράξεις:

- ΚΑΙ ή AND

  Η bitwise λογική AND των A και B αντιπροσωπεύεται από το Y.

- Ή ή OR

  Η bitwise λογική OR των A και B αντιπροσωπεύεται από το Y.

- Αποκλειστικό-Ή ή XOR

  Η Bitwise λογική XOR των A και B αντιπροσωπεύεται από το Y.

Μία ALU μπορεί να σχεδιαστεί για τον υπολογισμό οποιασδήποτε πράξης. Η πολυπλοκότητα δεν είναι ένας από τους τομείς που απασχολούν έναν μηχανικό. Το πρόβλημα έγκειται στο ότι όσο πιο πολύπλοκος είναι ένας υπολογισμός, τόσο πιο ακριβή γίνεται η ALU, καταλαμβάνει περισσότερο χώρο μέσα στον επεξεργαστή και καταναλώνει περισσότερη ενέργεια.

Για αυτό τον λόγο οι μηχανικοί πάντοτε θέτουν ένα συμβιβασμό, έτσι ώστε να παρέχουν στον επεξεργαστή μια ALU αρκετά δυνατή για να κάνει τον επεξεργαστή γρήγορο, αλλά και όχι τόσο πολύπλοκο για να μην γίνεται το κόστος και το μέγεθος του απαγορευτικό.

4.6 Ελεγκτής (Controller)

Η μονάδα ελέγχου (Control Unit) είναι το τμήμα του επεξεργαστή που κατευθύνει τη λειτουργία του. Λέει στη μνήμη, την αριθμητική και λογική μονάδα και τις συσκευές εισόδου και εξόδου του υπολογιστή πώς να ανταποκρίνονται στις οδηγίες που έχουν σταλεί στον επεξεργαστή.

Κατευθύνει τη λειτουργία των άλλων μονάδων παρέχοντας σήματα χρονισμού και ελέγχου. Η διαχείριση των περισσότερων πόρων υπολογιστών γίνεται από το CU. Κατευθύνει τη ροή δεδομένων μεταξύ της CPU και των άλλων συσκευών. Ο John von Neumann συμπεριέλαβε τη μονάδα ελέγχου ως μέρος της αρχιτεκτονικής von Neumann. Στα σύγχρονα σχέδια υπολογιστών, η μονάδα ελέγχου είναι συνήθως ένα εσωτερικό μέρος της CPU με τον συνολικό ρόλο και τη λειτουργία της να παραμένουν αμετάβλητες από την εισαγωγή της.

4.7 Βαθμίδα αποκωδικοποίησης εντολών

Η οδηγία που λαμβάνει ο επεξεργαστής από τη μνήμη καθορίζει το τι θα κάνει. Στο βήμα αποκωδικοποίησης, που εκτελείται από κύκλωμα δυαδικού αποκωδικοποιητή γνωστό ως αποκωδικοποιητή εντολών (Instructions Decoder), η εντολή μετατρέπεται σε σήματα που ελέγχουν άλλα μέρη του επεξεργαστή.

Ο τρόπος με τον οποίο ερμηνεύεται η εντολή ορίζεται από την αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture, ISA) της CPU. Συχνά, μια ομάδα bit εντός της εντολής, που ονομάζεται opcode, υποδεικνύει ποια λειτουργία είναι να εκτελεστεί, ενώ τα υπόλοιπα πεδία συνήθως παρέχουν συμπληρωματικές πληροφορίες που απαιτούνται για τη λειτουργία, όπως οι τελεστές (operands). Αυτοί οι τελεστές μπορούν να καθοριστούν ως μια σταθερή τιμή ή ως η θέση μιας τιμής που μπορεί να είναι ένας καταχωρητής επεξεργαστή ή μια διεύθυνση μνήμης, όπως καθορίζεται από κάποιο τρόπο διευθυνσιοδότησης.

Σε ορισμένα σχέδια CPU, ο αποκωδικοποιητής εντολών υλοποιείται ως ένα ενσύρματο, αμετάβλητο κύκλωμα δυαδικού αποκωδικοποιητή. Σε άλλα, ένα μικροπρόγραμμα χρησιμοποιείται για τη μετάφραση εντολών σε σύνολα σημάτων διαμόρφωσης CPU, που εφαρμόζονται διαδοχικά σε πολλαπλούς παλμούς ρολογιού. Σε ορισμένες περιπτώσεις, η μνήμη που αποθηκεύει το μικροπρόγραμμα μπορεί να επανεγγραφεί, καθιστώντας δυνατή την αλλαγή του τρόπου με τον οποίο η CPU αποκωδικοποιεί τις οδηγίες.

4.8 Μονάδα καταχωρητών

Ο καταχωρητής (register) είναι τύπος μικρής, αλλά πολύ γρήγορης μνήμης που βρίσκεται μέσα στο τσιπ του επεξεργαστή. Η μνήμη αυτή χρησιμοποιείται για την βελτίωση της ταχύτητας εκτέλεσης των διαφόρων προγραμμάτων, αφού σε αυτήν συνήθως αποθηκεύονται δεδομένα που χρησιμοποιούνται συνέχεια από τα προγράμματα. Ο καταχωρητής παρέχει πολύ γρήγορη πρόσβαση σε αυτά τα δεδομένα και έτσι το πρόγραμμα εκτελείται πιο γρήγορα. Οι περισσότεροι από τους σύγχρονους ηλεκτρονικούς υπολογιστές λειτουργούν σύμφωνα με την εξής λογική:

Μεταφέρουν δεδομένα από την κεντρική μνήμη στους καταχωρητές, κάνουν τις διάφορες πράξεις πάνω στα δεδομένα και στην συνέχεια μεταφέρουν το αποτέλεσμα από τους καταχωρητές πίσω στην κύρια μνήμη. Η τεχνική αυτή ονομάζεται load-store architecture.

Η ΚΜΕ (Κεντρική Μονάδα Επεξεργασίας) περιέχει πολλούς καταχωρητές, από τους οποίους άλλοι είναι γενικής χρήσης, ενώ άλλοι επιτελούν μια συγκεκριμένη λειτουργία (ειδικής χρήσης). Οι σημαντικότεροι από τους καταχωρητές ειδικής χρήσης είναι ο μετρητής προγράμματος (Program Counter) και ο καταχωρητής εντολών (Instruction Register). Ο μετρητής προγράμματος δείχνει την επόμενη εντολή που πρόκειται να εκτελεστεί, ενώ ο καταχωρητής εντολών περιέχει την εντολή που εκτελείται εκείνη τη στιγμή.

Στην ψηφιακή ηλεκτρονική, ειδικά στην επιστήμη υπολογιστών, οι hardware registers είναι κυκλώματα που συνήθως αποτελούνται από flip flops, συχνά με πολλά χαρακτηριστικά παρόμοια με τη μνήμη, όπως:

1. Η ικανότητα ανάγνωσης ή εγγραφής πολλαπλών δυαδικών ψηφίων ταυτόχρονα και

2. Χρήση μιας διεύθυνσης για την επιλογή ενός συγκεκριμένου καταχωρητή με τρόπο παρόμοιο με μια διεύθυνση μνήμης.

Το χαρακτηριστικό τους, ωστόσο, που τους κάνει να ξεχωρίζουν, είναι ότι έχουν ειδικές λειτουργίες που σχετίζονται με το υλικό πέρα από αυτές της συνηθισμένης μνήμης. Έτσι, οι hardware registers είναι σαν τη μνήμη με πρόσθετες λειτουργίες που σχετίζονται με το υλικό, δηλαδή τα κυκλώματα μνήμης είναι σαν hardware registers που αποθηκεύουν απλώς δεδομένα. Ανάλογα με την πολυπλοκότητά τους, οι σύγχρονες συσκευές υλικού μπορούν να έχουν πολλούς καταχωρητές.

4.9 CPU clock

Το ρολόι της CPU βρίσκεται σε άμεση σύνδεση με τον ελεγκτή (Controller or Control Unit), προκειμένου να συνδράμει στο συγχρονισμό και στην ομαλή λειτουργία όλων των στοιχείων της CPU. Ο ρυθμός με τον οποίο η μονάδα ελέγχου εκτελεί τις εντολές και τα σήματα καθορίζεται από την ταχύτητα του ρολογιού. Πρακτικά όπως έχει αναφερθεί και παραπάνω, κάθε «χτύπος» του ρολογιού ισοδυναμεί και με μια αλλαγή κατάστασης στη CPU.

5 Διαγράμματα & Ερμηνεία του κώδικα ακολουθούμενη από πρακτική υλοποίηση στο Mojo v.3

Στην ενότητα αυτή θα πραγματοποιηθεί μια επεξήγηση των παραρτημάτων στα οποία περιέχονται τα τμήματα του VHDL κώδικα που χρησιμοποιήθηκε για την διεκπεραίωση αυτής της εργασίας. Με άλλα λόγια θα επεξηγηθούν από τεχνικής σκοπιάς τα μεμονωμένα τμήματα που αποτελούν τον επεξεργαστή. Εκτός από την επεξήγηση των περισσότερων τμημάτων του κώδικα, θα δοθεί και μια πρώτη ερμηνεία των αποτελεσμάτων που προέκυψαν από τη μελέτη των ψηφιακών διαγραμμάτων. Στόχος όλων των διαγραμμάτων που παρουσιάζονται είναι η επιβεβαίωση της σωστής λειτουργίας κάθε κυκλώματος.

Ο επεξεργαστής που κατασκευάστηκε πρακτικά αποτελείται από καταχωρητές, από τη Λογική πράξη την ALU η οποία μέσα έχει τον Adder, από τον Decoder και από την πράξη για την αποθήκευση των εντολών. Στη συνέχεια ο επεξεργαστής συνδέεται με μία μνήμη, προκειμένου να αποθηκεύεται σε αυτή η σειρά των εντολών. Τέλος η μνήμη ενημερώνει τον επεξεργαστή ποια εντολή πρέπει να εκτελέσει.

5.1 Αθροιστής (Adder)

Στην υποενότητα αυτή θα αναλυθεί η οντότητα του αθροιστή ή αλλιώς Adder, όπως ονομάζεται. Το παράρτημα A με τον κώδικα περιγράφει τη λειτουργία αυτής της οντότητας. Πιο συγκεκριμένα η μονάδα του αθροιστή (Adder) που ορίζεται, εκτελεί προσθέσεις των 32 bit, καθώς επίσης χρησιμοποιεί και κατάλληλα σήματα μεταφοράς. Στο επίπεδο συμπεριφοράς του αθροιστή ορίζονται τα 3 ακόλουθα σήματα:

- Άθροισμα (sum): σήμα μεγέθους 32 bit που αντιπροσωπεύει το άθροισμα των A, B και της μεταφοράς.
- Μεταφορά (carry): σήμα μεγέθους 33 bit που αντιπροσωπεύει τη μεταφορά που παράγεται κατά την προσθήκη.
- carry_in: ένα ακέραιο σήμα που διατηρεί την τιμή 1 εάν η μεταφορά είναι 1, διαφορετικά η τιμή του παραμένει 0.

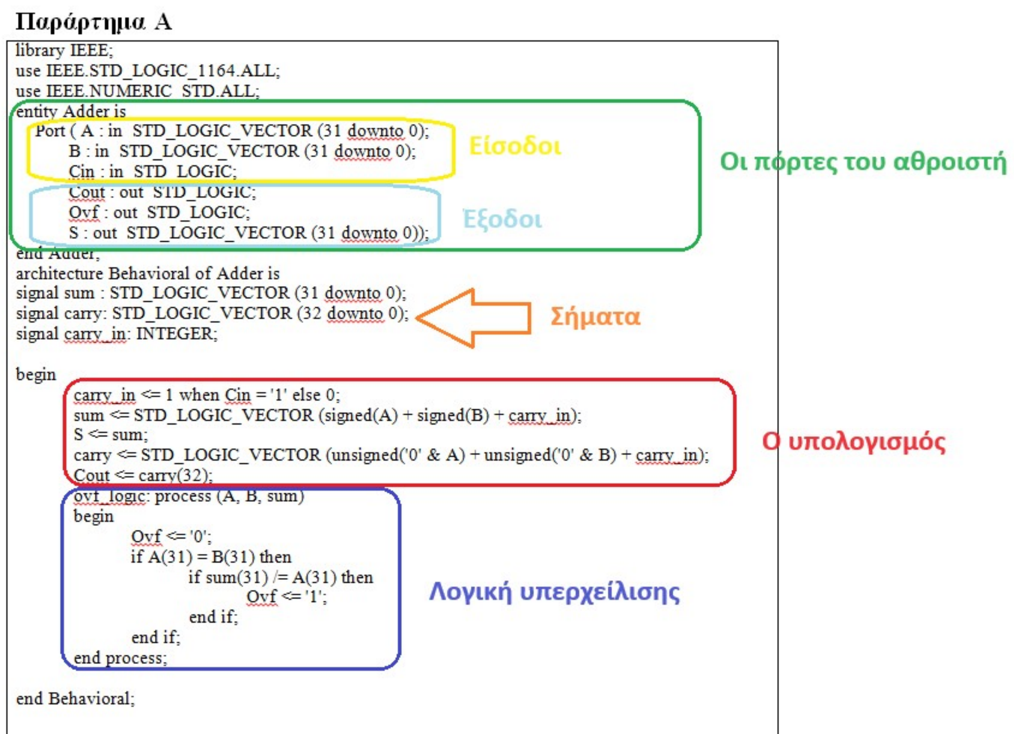
Κατά τον υπολογισμό, πραγματοποιείται η πράξη της πρόσθεσης στην οποία λαμβάνονται υπόψη και κάποια ακόμα σήματα. Παρακάτω παρουσιάζονται αναλυτικά όλες οι εκχωρήσεις τιμών που υλοποιούνται:

- Το carry_in έχει τιμή 1 εάν το Cin είναι 1. Διαφορετικά, εκχωρείται η τιμή 0. Το Cin έχει το ρόλο του κρατούμενου, όπως συμβαίνει και κατά το αριθμητικό άθροισμα.
- Το αποτέλεσμα υπολογίζεται ως το άθροισμα των A, B, και carry_in. Διευκρινίζεται ότι τα A & B θεωρούνται προσημασμένοι αριθμοί.
- Στο S αποδίδεται η τιμή του αθροίσματος.
- Η μεταφορά υπολογίζεται ως το άθροισμα των A, B και carry_in.
- Στο Cout εκχωρείται η τιμή του 33ου bit μεταφοράς.

Επιπλέον στην οντότητα του Adder, συμπεριλαμβάνεται και μια λογική υπερχείλισης (overflow logic), η οποία είναι υπεύθυνη στο να ανιχνεύει διαφορές – αλλαγές μεταξύ του A, B και του αθροίσματος. Πιο συγκεκριμένα ελέγχει για συνθήκες υπερχείλισης συγκρίνοντας το πιο σημαντικό bit του A, B με αυτό του αθροίσματος. Εάν τα πιο σημαντικά bit του A και του B είναι τα ίδια, αλλά διαφορετικά από το πιο σημαντικό bit του αθροίσματος, τότε ανιχνεύεται μια συνθήκη υπερχείλισης και διαμορφώνεται το αντίστοιχο σήμα.

Γενικότερα η μονάδα Adder εκτελεί προσθέσεις 32-bit, καθώς επίσης ανιχνεύει και συνθήκες υπερχείλισης με βάση τις τιμές των εισόδων και του αθροίσματος. Τα σήματα διεξαγωγής και υπερχείλισης παρέχονται ως έξοδοι για τη διευκόλυνση της περαιτέρω επεξεργασίας σε ένα μεγαλύτερο σύστημα, το οποίο στη συγκεκριμένη περίπτωση είναι ο επεξεργαστής που έχει δημιουργηθεί.

Στην εικόνα που ακολουθεί σημειώνονται πάνω στον κώδικα όσα έχουν ήδη αναφερθεί σε αυτή την υποενότητα.



Παράρτημα Α

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Adder is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
           B : in  STD_LOGIC_VECTOR (31 downto 0);
           Cin : in  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Ovf : out  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR (31 downto 0));
end Adder;
architecture Behavioral of Adder is
signal sum : STD_LOGIC_VECTOR (31 downto 0);
signal carry: STD_LOGIC_VECTOR (32 downto 0);
signal carry_in: INTEGER;

begin
        carry_in <= 1 when Cin = '1' else 0;
        sum <= STD_LOGIC_VECTOR (signed(A) + signed(B) + carry_in);
        S <= sum;
        carry <= STD_LOGIC_VECTOR (unsigned('0' & A) + unsigned('0' & B) + carry_in);
        Cout <= carry(32);
        ovf_logic: process (A, B, sum)
        begin
                Ovf <= '0';
                if A(31) = B(31) then
                        if sum(31) /= A(31) then
                                Ovf <= '1';
                        end if;
                end if;
        end process;

end Behavioral;
```


Εικόνα 5.1: Ο αθροιστής

5.2 Εκτέλεση πράξεων

Για την υλοποίηση της ALU χρησιμοποιήθηκε ένας Adder των 32 bit με Carry in, Carry out, overflow out. Η λογική για το overflow είναι η εξής: όταν προσθέτουμε αριθμούς με ίδιο πρόσημο και το αποτέλεσμα που θα προκύψει έχει διαφορετικό πρόσημο, τότε έχουμε overflow. Το carry out το υπολογίζεται κάνοντας μία δεύτερη πρόσθεση μεταξύ των ίδιων σημάτων αλλά σε 33 bit αναπαράσταση με zero fill:

carry <= STD_LOGIC_VECTOR (unsigned('0' & A) + unsigned('0' & B) + carry_in); Cout <= carry(32);

όπου το carry_in είναι integer με τιμή 0 ή 1 ανάλογη του Cin (είσοδος στον adder).

Η ALU όταν θα κάνει αφαίρεση θα αντιστρέφει τα bit του δεύτερου operant και θα βάζει '1' στο Cin του adder. Στην ουσία θα κάνει πρόσθεση με το two's complement του δεύτερου operant.

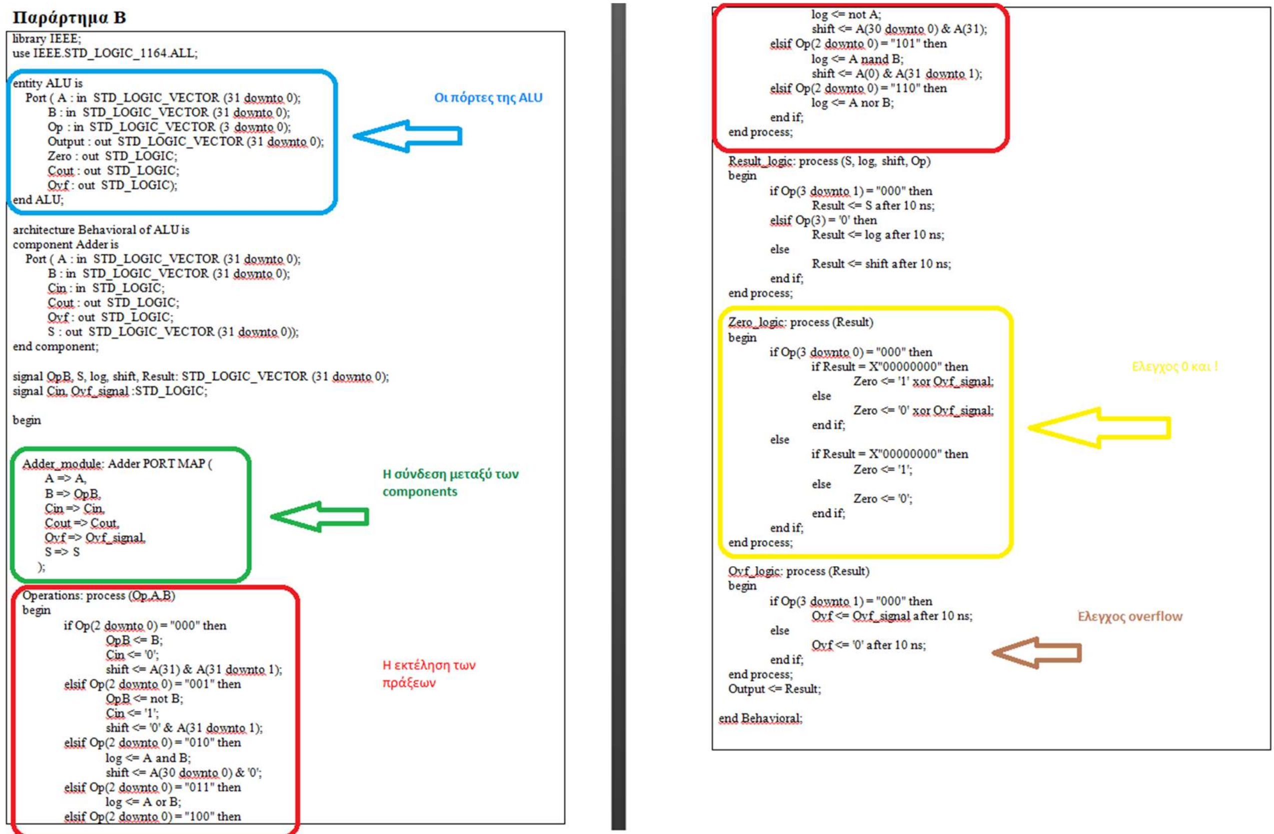
Στην εικόνα που ακολουθεί φαίνεται ο κώδικας του παραρτήματος Β, η λειτουργία του οποίου είναι η πραγματοποίηση αριθμητικών και λογικών πράξεων. Όλες οι πράξεις υλοποιούνται από την οντότητα ALU, η οποία είναι και αυτή ένα κύκλωμα.

Με μπλε χρώμα φαίνεται η οντότητα ALU στην οποία δηλώνονται όλες οι πόρτες για το συγκεκριμένο τμήμα του κώδικα. Συγκεκριμένα δηλώνεται το A και το B, η πράξη που θα πραγματοποιηθεί, η έξοδος, αν το αποτέλεσμα είναι 0 ή όχι, αν υπάρχει κρατούμενο και αν υπάρχει overflow. Η πράξη που ελέγχει το 0 είναι διαφορετική από αυτή που ελέγχει το overflow. Συγκεκριμένα η πρώτη από αυτές φαίνεται με το κίτρινο βέλος, ενώ η δεύτερη με το καφέ. Όλες οι πράξεις για να ολοκληρωθούν χρειάζονται χρόνο 10 ns.

Το τμήμα του κώδικα μέσα στο πράσινο πλαίσιο είναι υπεύθυνο για την προετοιμασία των A και B, πριν από την εκτέλεση των πράξεων. Το S που φαίνεται μέσα στο τμήμα αυτό αφορά το Sum, δηλαδή το αποτέλεσμα της πράξης A + B. Ως Cin ορίζεται το κρατούμενο εισόδου και ως Cout το κρατούμενο εξόδου.

Το κόκκινο τμήμα του κώδικα είναι η εκτέλεση των πράξεων. Ο τελεστής Op επειδή είναι 3 bit (2^3) μπορεί να δεχθεί μέχρι 8 πράξεις. Η πράξη 000 είναι η πρόσθεση. Κάθε φορά που ο operator είναι της μορφής 100, τότε εκτελείται η πράξη του shift, δηλαδή ο πολλαπλασιασμός και η διαίρεση.

Ως notB ορίζεται το αντίστροφο του B και *Cin <= '1'* σημαίνει ότι του προτίθεται ένα κρατούμενο.



Παράρτημα Β

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

entity ALU is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
           B : in  STD_LOGIC_VECTOR (31 downto 0);
           Op : in  STD_LOGIC_VECTOR (3 downto 0);
           Output : out  STD_LOGIC_VECTOR (31 downto 0);
           Zero : out  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Ovf : out  STD_LOGIC);
end ALU;

architecture Behavioral of ALU is
component Adder is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
           B : in  STD_LOGIC_VECTOR (31 downto 0);
           Cin : in  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Ovf : out  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

signal OpB, S, log, shift, Result: STD_LOGIC_VECTOR (31 downto 0);
signal Cin, Ovf_signal :STD_LOGIC;

begin

  Adder_module: Adder PORT MAP (
       A => A,
       B => OpB,
       Cin => Cin,
       Cout => Cout,
       Ovf => Ovf_signal,
       S => S
     );

  Operations: process (Op,A,B)
  begin
       if Op(2 downto 0) = "000" then
            OpB <= B;
            Cin <= '0';
            shift <= A(31) & A(31 downto 1);
       elsif Op(2 downto 0) = "001" then
            OpB <= not B;
            Cin <= '1';
            shift <= '0' & A(31 downto 1);
       elsif Op(2 downto 0) = "010" then
            log <= A and B;
            shift <= A(30 downto 0) & '0';
       elsif Op(2 downto 0) = "011" then
            log <= A or B;
       elsif Op(2 downto 0) = "100" then
            log <= not A;
            shift <= A(30 downto 0) & A(31);
       elsif Op(2 downto 0) = "101" then
            log <= A nand B;
            shift <= A(0) & A(31 downto 1);
       elsif Op(2 downto 0) = "110" then
            log <= A nor B;
       end if;
  end process;

  Result_logic: process (S, log, shift, Op)
  begin
       if Op(3 downto 1) = "000" then
            Result <= S after 10 ns;
       elsif Op(3) = '0' then
            Result <= log after 10 ns;
       else
            Result <= shift after 10 ns;
       end if;
  end process;

  Zero_logic: process (Result)
  begin
       if Op(3 downto 0) = "000" then
            if Result = X"00000000" then
                 Zero <= '1' xor Ovf_signal;
            else
                 Zero <= '0' xor Ovf_signal;
            end if;
       else
            if Result = X"00000000" then
                 Zero <= '1';
            else
                 Zero <= '0';
            end if;
       end if;
  end process;

  Ovf_logic: process (Result)
  begin
       if Op(3 downto 1) = "000" then
            Ovf <= Ovf_signal after 10 ns;
       else
            Ovf <= '0' after 10 ns;
       end if;
  end process;
  Output <= Result;

end Behavioral;
```


Εικόνα 5.2: Εκτέλεση αριθμητικών & λογικών πράξεων

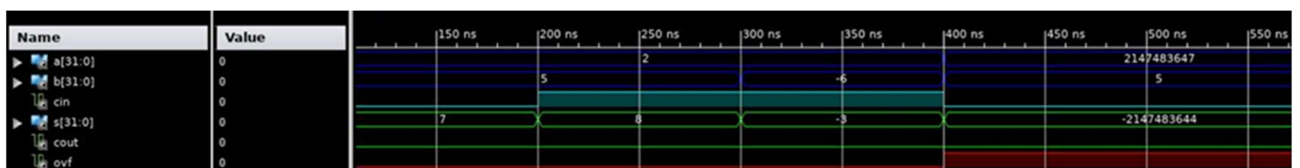
Παρακάτω εμφανίζεται η λέξη log. Αυτή δηλώνει το αποτέλεσμα μιας λογικής πράξης. Για παράδειγμα το *log <= A nand B;* καταγράφει το αποτέλεσμα του A not AND B.

Η δήλωση *A (31 downto 1)* δείχνει πως ο αριθμός A έχει όλα τα ψηφία από το 31 έως το 1, με εξαίρεση το τελευταίο – δεξιότερο bit.

Η λειτουργία του *shift* μεταφέρει όλα τα bit κατά ένα προς τα δεξιά.

Η δήλωση *shift <= A(31) & A(31 downto 1);* πρακτικά αφορά την πράξη της διαίρεσης.

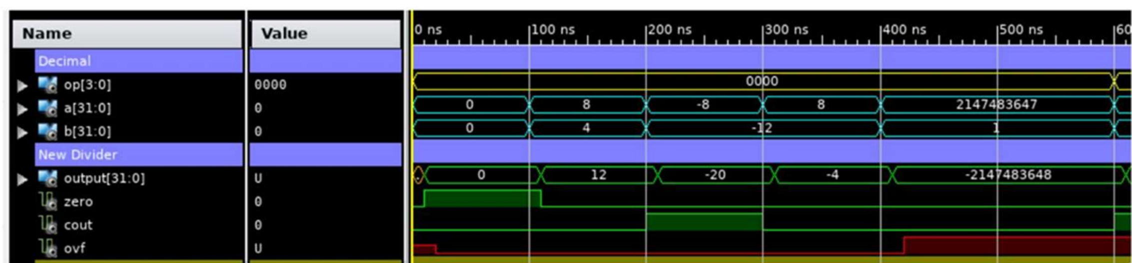
Παρακάτω δίνονται κάποια διαγράμματα τα οποία περιλαμβάνουν την εκτέλεση, τόσο αριθμητικών, όσο και λογικών πράξεων, καθώς και τα αντίστοιχα αποτελέσματά τους. Στο επάνω μέρος όλων των διαγραμμάτων που παρουσιάζονται φαίνεται ο χρόνος σε ns διήρκησε η κάθε πράξη – ενέργεια. Πραγματοποιείται επεξήγηση του τρόπου με τον οποίο προέκυψε το κάθε αποτέλεσμα.



Διάγραμμα 5.1: Η πράξη της πρόσθεσης

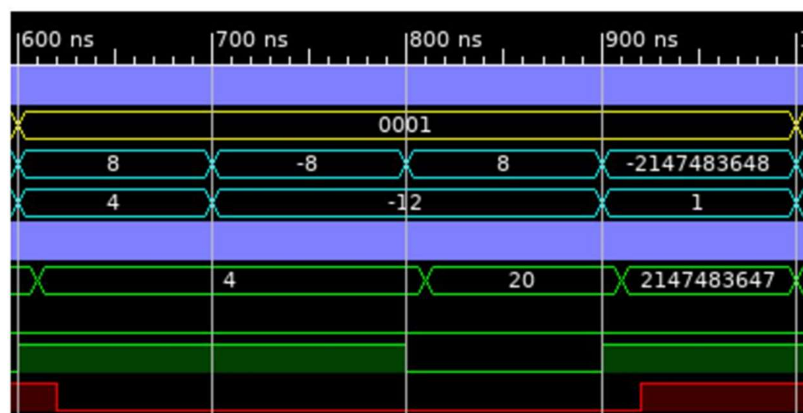
Όλες οι πράξεις που εκτελούνται στο παραπάνω διάγραμμα χρησιμοποιούν τον τελεστή της πρόσθεσης. Αρχικά φαίνεται ότι εκτελείται η πράξη *2 + 5* και το αποτέλεσμα είναι το *7*. Έπειτα εμφανίζεται και 1 κρατούμενο (Cin τιμή – γαλάζια γραμμή) και το αποτέλεσμα της ίδιας πράξης γίνεται 8. Ακολουθεί η πράξη *2 - 6* που ισούται με -4, ωστόσο προσθέτοντας και πάλι το κρατούμενο το τελικό αποτέλεσμα, όπως φαίνεται είναι το *-3*.

Τέλος φαίνεται να εκτελείται μια πράξη με έναν αρκετά μεγάλο αριθμό. Ο αριθμός αυτός έχει μπροστά τη μονάδα, πράγμα που σημαίνει ότι έχει αρνητικό πρόσημο. Γι' αυτό άλλωστε και το τελικό αποτέλεσμα είναι αρνητικό. Στο σημείο αυτό παρατηρείται ότι κάτω από το τελικό αποτέλεσμα υπάρχει μια παχιά κόκκινη γραμμή. Αυτή αντιστοιχεί στη μονάδα και δείχνει ότι υπάρχει overflow. Όταν η γραμμή είναι λεπτή αντιστοιχεί σε 0.



Διάγραμμα 5.2: Η πράξη της πρόσθεσης με αρνητικούς αριθμούς

Οι πράξεις του ανωτέρου διαγράμματος χρησιμοποιούν και πάλι τον τελεστή της πρόσθεσης (0000), με τη διαφορά ότι αυτή τη φορά χρησιμοποιούνται και αρνητικοί αριθμοί. Αρχικά είναι εμφανές ότι το αποτέλεσμα της πράξης *0 + 0* είναι *0*. Έπειτα πραγματοποιείται η πράξη *8 + 4* και προκύπτει το *12*. Η πράξη *-8 + (-12)* έχει ως αποτέλεσμα το *-20* και εμφανίζει ένα κρατούμενο εξόδου, δηλαδή Cout=1 γιατί αυξάνονται τα bit. Αυτό ωστόσο δεν συμβαίνει σε όλα τα αρνητικά αποτελέσματα. Η πράξη *8 + (-12)* δίνει το τελικό *-4*. Τέλος παρατηρείται ότι υπάρχει Overflow (κόκκινη γραμμή), κατά την εκτέλεση της πράξης με τον μεγάλο αριθμό.



Διάγραμμα 5.3: Η πράξη της αφαίρεσης

Ακολουθεί το παραπάνω διάγραμμα το οποίο περιλαμβάνει κάποιες πράξεις που χρησιμοποιούν τον αφαιρετικό τελεστή, ο οποίος είναι το 0001. Η πράξη *8 – 4* επιστρέφει το αποτέλεσμα *4*, ενώ το ίδιο αποτέλεσμα επιστρέφεται και από την επόμενη πράξη που είναι η *-8 – (-12)*. Στη συνέχεια η πράξη *8 – (-12)* επιστρέφει το θετικό *20*. Κλείνοντας από τον μεγάλο αριθμό αφαιρείται η μονάδα και προκύπτει το τελικό αποτέλεσμα. Υπενθυμίζεται ότι ο αρχικός μεγάλος αριθμός μπορεί να έχει αρνητικό πρόσημο, αλλά με τη μονάδα που έχει μπροστά, αυτό μετατρέπεται σε θετικό (-1 * -1 = 1).

Διάγραμμα 5.4: Εκτέλεση λογικών & αριθμητικών πράξεων με bin & hex αριθμούς

- Ο πρώτος τελεστής είναι ο 0010 και αντιστοιχεί στη λογική πράξη AND. Εκτελείται η πράξη *ffff0000 AND ff00ff00* και το αποτέλεσμα είναι *ff000000*.
- Ο δεύτερος τελεστής είναι το 0011 και αντιπροσωπεύει το OR. Εκτελείται η πράξη *ffff0000 OR ff00ff00* και το αποτέλεσμα είναι *ffffff00*.
- Ο τελεστής 0100 αντιστοιχεί στο NOTA, δηλαδή υπολογίζεται το αντίστροφο του A, το οποίο είναι *0000ffff*.
- Ο τελεστής 0101 υποδηλώνει την πράξη NAND, δηλαδή not AND. Επομένως το αποτέλεσμα *ff000000* αντιστρέφεται και γίνεται *00ffffff*.
- Ο τελεστής 0110 υποδηλώνει την πράξη NOR, δηλαδή το αντίστροφο του OR. Επομένως το αποτέλεσμα *ffffff00* αντιστρέφεται και γίνεται *000000ff*.
- Ο τελεστής 1000 που ακολουθεί πραγματοποιεί διαίρεση με πρόσθεση κατά 1 (ολίσθηση με συμπλήρωμα). Με άλλα λόγια στον αριθμό *80000003* που είναι η είσοδος πραγματοποιείται shift όλων των ψηφιών μία θέση προς τα δεξιά. Στην αρχή του αριθμού μπαίνει το 1 και έτσι προκύπτει ο αριθμός 12, ο οποίος στο δεκαεξαδικό σύστημα είναι το c. Το τελικό αποτέλεσμα της πράξης είναι *c0000001*.
- Ο τελεστής 1001 πραγματοποιεί και πάλι διαίρεση, χωρίς την προσθήκη του 1. Με άλλα λόγια η πρώτη θέση γεμίζει με 0. Το τελικό αποτέλεσμα είναι *40000001*.
- Ο τελεστής 1010 αντιστοιχεί στον πολλαπλασιασμό του αριθμού με το 2. Το αρχικό 8 μετακινείται μια θέση προς τα αριστερά, επομένως απομακρύνεται εντελώς από το τελικό αποτέλεσμα. Το τελικό αποτέλεσμα είναι *00000006*.
- Ο τελεστής 1100 αντιστοιχεί στον πολλαπλασιασμό του αριθμού με το 2 και προστίθεται και μία μονάδα στο τέλος του. Παρομοίως με την προηγούμενη περίπτωση το αρχικό 8 μετακινείται μια θέση προς τα αριστερά. Το τελικό αποτέλεσμα είναι *00000007*.
- Ο τελεστής 1101 πραγματοποιεί μια πράξη shift (ολίσθησης) προς τα δεξιά (διαίρεση) με συμπλήρωση κατά 1, παρόμοια με τον 1000.

Σημειώνεται ότι σε καμία από τις ανωτέρω πράξεις δεν εμφανίζεται κρατούμενο ή κάποιο overflow.

5.3 Decoder

Το παράρτημα Γ με τον κώδικα αφορά τον Decoder, ο οποίος αποτελεί άλλο ένα ξεχωριστό κύκλωμα. Η λειτουργία του με απλά λόγια είναι να δέχεται έναν δεκαδικό αριθμό και να τον μετατρέπει σε δυαδικό. Κάθε είσοδος που λαμβάνει είναι 5 bit, ενώ κάθε έξοδος είναι 32 bit.

Παρακάτω παρουσιάζονται 2 παραδείγματα με το αποτέλεσμα που θα προκύψει αν δοθούν ως είσοδοι οι αριθμοί 12 και 17 αντίστοιχα.

12 → 0000 0000 0000 0000 0000 1000 0000 0000
17 → 0000 0000 0000 0001 0000 0000 0000 0000

Στην εικόνα που ακολουθεί φαίνεται το τμήμα του κώδικα, ενώ με κίτρινο χρώμα επισημαίνεται η είσοδος που δίνεται στην περίπτωση του νέου επεξεργαστή.

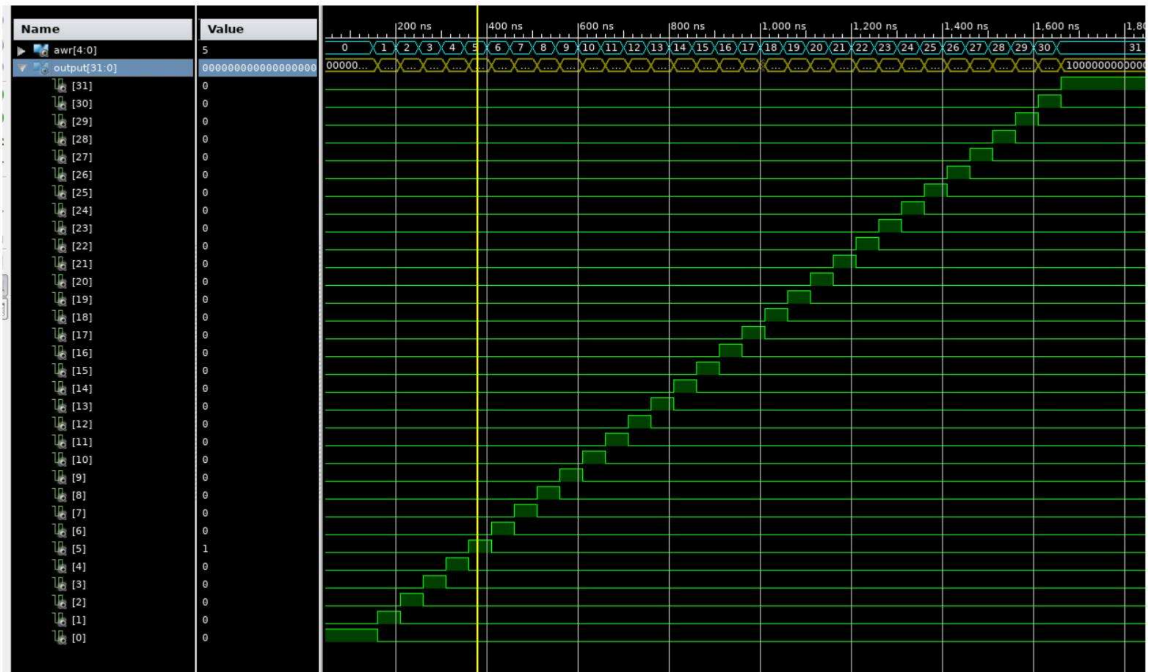
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Decoder is
  Port ( Awr : n  STD_LOGIC_VECTOR (4 downto 0);
         Output : out  STD_LOGIC_VECTOR (31 downto 0));
end Decoder;

architecture Behavioral  of Decoder is

signal  dec_out: STD_LOGIC_VECTOR (31 downto 0);

begin
        dec_logic: process (Awr)
        begin
                dec_out <= (others => '0');
                for i in 0 to 31 loop
                        if i = unsigned(Awr) then
                                dec_out(i) <= '1';
                        end if;
                end loop;
        end process;
        Output <= dec_out after 10 ns;

end Behavioral;
```

Πόρτες αποκωδικοποιητή

CPU-Επεξεργαστής

Συμπεριφορά αποκωδικοποιητή

Εικόνα 5.3: Κώδικας Decoder



Διάγραμμα 5.5: Decoder

Η λειτουργία του decoder απεικονίζεται και στο παραπάνω διάγραμμα. Είναι εμφανές ότι για κάθε αριθμό που δίνεται εμφανίζεται και ένα αντίστοιχο πράσινο «σκαλοπάτι» στο αντίστοιχο bit. Για παράδειγμα ο αριθμός 1 εμφανίζει το προαναφερθέν σκαλοπάτι στο $1^{ο}$ bit, ενώ ο αριθμός 5 στο $5^{ο}$ bit (κίτρινη γραμμή).

5.4 Controller

Ο ρόλος του επεξεργαστή είναι να διαχειρίζεται μια σειρά από εντολές, οι οποίες υπάρχουν στη μνήμη. Με βάση τις οδηγίες που διαβάζει από τη μνήμη εκτελεί και τις αντίστοιχες ενέργειες. Για παράδειγμα αν η μνήμη του πει να διαβάσει την τιμή A, τότε αφού τη διαβάσει θα πάει και θα την «τοποθετήσει» σε κάποιο καταχωρητή.

Ο ρόλος του controller είναι να ελέγχει βήμα προς βήμα τι κάνει ο επεξεργαστής και ανάλογα με την κάθε περίπτωση να λαμβάνει και κάποιες αποφάσεις, για παράδειγμα την υλοποίηση μιας πράξης. Συνδυάζεται με τη μνήμη 2048, η οποία περιέχει 2 πράξεις τη διάβασε (read) και τη γράψε(write).

Ως γενική λειτουργία του controller είναι η λήψη των σημάτων και η επιστροφή των αντίστοιχων bit, προκειμένου να γίνουν οι αντίστοιχες πράξεις ή να υλοποιηθούν κάποιες εγγραφές στη μνήμη.

Το παράρτημα Δ αντιπροσωπεύει τον κώδικα για τον controller. Παρακάτω παρουσιάζεται επιλεκτικά ένα μέρος του κώδικα, προκειμένου να εξηγηθεί η λειτουργία του.

```
entity CONTROL is
   Port ( Instr : in  STD_LOGIC_VECTOR (31 downto 0);
         PC_sel : out  STD_LOGIC;
         PC_LdEn : out  STD_LOGIC;
         RF_WrEn : out  STD_LOGIC;
         RF_WrData_sel : out  STD_LOGIC;
         RF_B_sel : out  STD_LOGIC;
         ImmExt : out  STD_LOGIC_VECTOR (1 downto 0);
         ALU_Bin_sel : out  STD_LOGIC;
         ALU_func : out  STD_LOGIC_VECTOR (3 downto 0);
         ALU_zero : in  STD_LOGIC;
         ByteOp : out  STD_LOGIC;
         Mem_WrEn : out  STD_LOGIC);
end CONTROL;

architecture Behavioral of CONTROL is

signal branch: STD_LOGIC;

begin

      is_branch: process (Instr)
      begin
            if Instr(31 downto 26) = "111111" then
                  branch <= '1';
                  PC_sel <= '1';
            elsif Instr(31 downto 26) = "000000" then
                  branch <= '1';
                  PC_sel <= '1' and ALU_zero;
            elsif Instr(31 downto 26) = "000001" then
                  branch <= '1';
                  PC_sel <= '1' and ALU_zero;
            else
                  branch <= '0';
                  PC_sel <= '0';
            end if;
      end process;

      mem_wr: process (Instr)
      begin
            if Instr(31 downto 26) = "000111" then
                  Mem_WrEn <= '1';
            elsif Instr(31 downto 26) = "011111" then
                  Mem_WrEn <= '1';
            else
                  Mem_WrEn <= '0';
            end if;
      end process;
```

Σειριακή διεργασία

καταχώρηση τιμής

Εγγραφή στην μνήμη

Εικόνα 5.4: Τμήμα του κώδικα για τον Controller

Το Instr, που φαίνεται μέσα στο κίτρινο πλαίσιο, προκύπτει από τη λέξη Instruction και αφορά κάποια οδηγία. Συγκεκριμένα εκεί που υπάρχει το κίτρινο βέλος ο controller ελέγχει αν τα bit από το 31ο μέχρι το 26ο είναι ίσα με τη μονάδα. Αν αυτό ισχύει, τότε κάποιοι καταχωρητές, όπως ο PC_sel (μπλε βέλος) λαμβάνουν και συγκεκριμένες τιμές.

Μέσα στο πορτοκαλί πλαίσιο φαίνεται η διεργασία mem_wr η οποία ελέγχει αν θα πραγματοποιηθεί κάποια εγγραφή στη μνήμη. Όταν η mem_wr λαμβάνει την τιμή 1 (Mem_WrEn <= '1') τότε σημαίνει πως πραγματοποιείται εγγραφή. Αντίθετα στην περίπτωση που Mem_WrEn <= '0', δεν γίνεται καμία εγγραφή.

5.5 Μνήμη

Η μνήμη είναι ένα κύκλωμα το οποίο χρειάζεται 3 στοιχεία για να λειτουργήσει σωστά. Αρχικά χρειάζεται τη διεύθυνση (address) που θα πραγματοποιήσει κάποια ανάγνωση (read) ή κάποια εγγραφή (write). Έπειτα χρειάζεται την πληροφορία για το αν θα επιτελέσει εγγραφή ή ανάγνωση (Write or Read). Τέλος χρειάζεται τα δεδομένα που θα χρησιμοποιήσει για να υλοποιήσει την αντίστοιχη ενέργεια.

Το μέγεθος της είναι 2^{32}, δηλαδή μπορεί να αποθηκεύσει 32 bit. Όπως γίνεται κατανοητό το μέγεθος αυτό είναι αρκετά μικρό, αν σκεφτεί κανείς ότι κάποιοι απλοί επεξεργαστές σημερινών υπολογιστών διαθέτουν μνήμη 34359738368 bit ή 4GB.

Διάγραμμα 5.6: Η μνήμη του επεξεργαστή

Το διάγραμμα 5.6 αφορά τη μνήμη του επεξεργαστή.

5.6 Entity MEM

Το τσιπάκι της μνήμης αντιπροσωπεύεται από την οντότητα MEM (Entity MEM), μέσα στο οποίο περιλαμβάνεται μια διεύθυνση (address) για να γράφονται όλα τα δεδομένα. Επιπρόσθετα από αυτό δίνεται η πληροφορία της εξόδου που θα προκύψει.

Τα στοιχεία της μνήμης είναι οι διαθέσιμες διευθύνσεις της. Η μορφή της είναι πινακοειδής, καθώς αποτελείται από γραμμές και στήλες. Με άλλα λόγια η μνήμη διαθέτει 1024 γραμμές και 1024 στήλες.

Στην εικόνα που ακολουθεί το data_dout (κίτρινο βέλος) προσδιορίζει την κατάσταση της μνήμης, καθώς και τη διεύθυνσή της. Με πορτοκαλί βέλος σημειώνεται το inst_dout στο οποίο περιλαμβάνονται όλες οι οδηγίες – εντολές που πρόκειται να δοθούν στη μνήμη. Τέλος το πράσινο βέλος υποδεικνύει την προσθήκη των δεδομένων στη μνήμη.

Παρακάτω με μπλε βέλος σημειώνεται το αρχείο των δεδομένων που φορτώνεται στη μνήμη και η ονομασία του είναι “rom.data”.

Με γαλάζια υπογράμμιση (highlight) επισημαίνεται το ρολόι που διαθέτει ο επεξεργαστής για την εκτέλεση των πράξεων μετά από ένα χρονικό διάστημα. Αυτό μπορεί να είναι, είτε 10ns, είτε 12ns. Στην περίπτωση του συγκεκριμένου επεξεργαστή επιλέχθηκε το πρώτο μέγεθος.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity MEM is
   Port ( clk : in  STD_LOGIC;
         inst_addr : in  STD_LOGIC_VECTOR (10 downto 0);
         inst_dout : out  STD_LOGIC_VECTOR (31 downto 0);
         data_we : in  STD_LOGIC;
         data_addr : in  STD_LOGIC_VECTOR (10 downto 0);
         data_din : in  STD_LOGIC_VECTOR (31 downto 0);
         data_dout : out  STD_LOGIC_VECTOR (31 downto 0));
end MEM;

architecture syn of MEM is
   type ram_type is array (2047 downto 0) of std_logic_vector (31 downto 0);
   impure function InitRamFromFile (RamFileName : in string) return ram_type is
   FILE ramfile : text is in RamFileName;
   variable RamFileLine : line;
   variable ram : ram_type;
   begin
      for i in 0 to 1023 loop
         readline(ramfile, RamFileLine);
         read (RamFileLine, ram(i));
      end loop;
      for i in 1024 to 2047 loop
         ram(i) := x"00000000";
      end loop;
   return ram;
   end function;

   signal MEM:ram_type := InitRamFromFile("rom.data");

begin
   process (clk)
   begin
      if clk'event and clk = '1' then
         if data_we = '1' then
            MEM(conv_integer(data_addr)) <= data_din;
         end if;
      end if;
   end process;
```

ΜΝΗΜΗ
οδηγίες-εντολές
κατάσταση-διεύθυνση
χρόνος απόκρισης CPU 10ns
αρχείο φόρτωσης

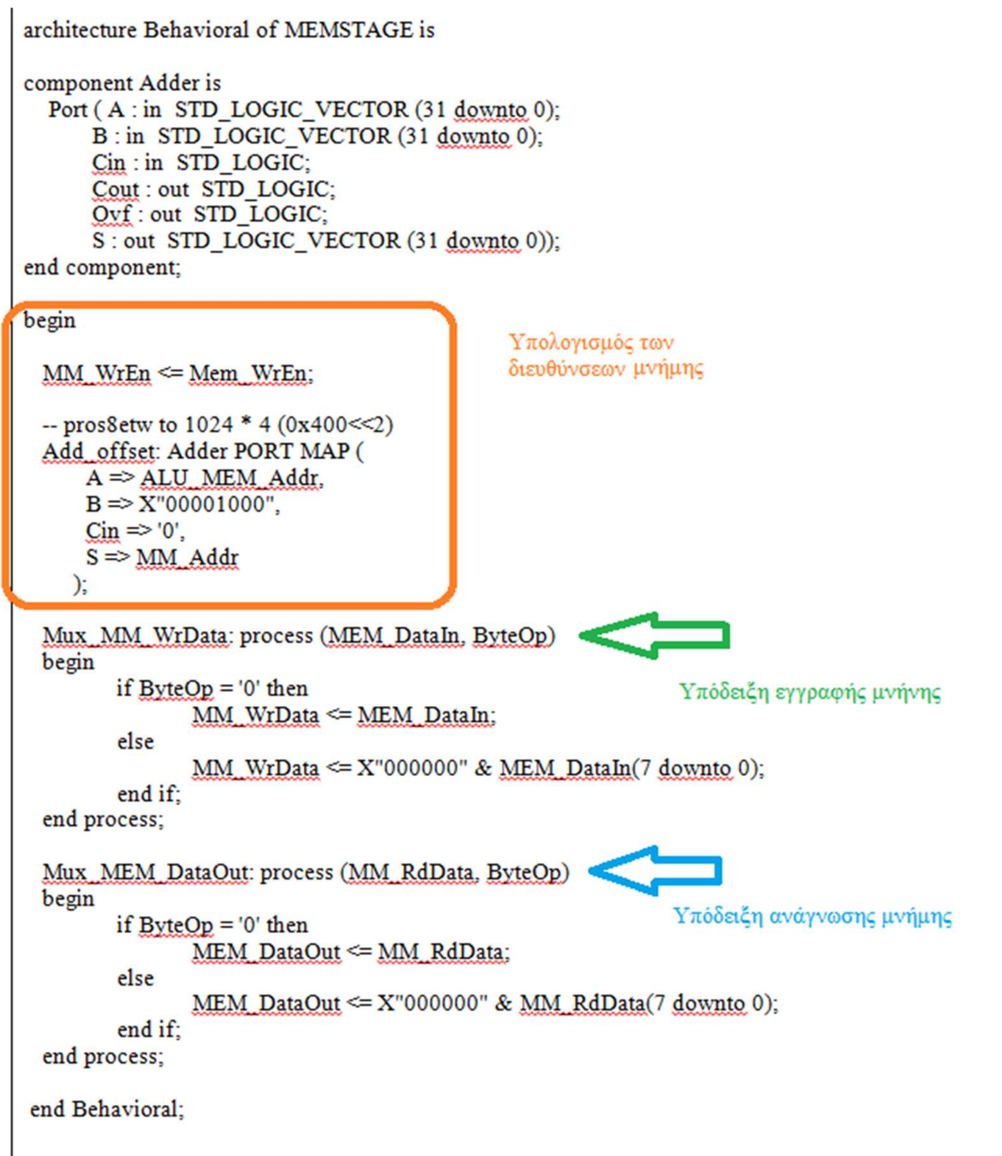
Εικόνα 5.5: Είσοδοι στο τσιπάκι της μνήμης

Στη συνέχεια θα παρουσιαστεί η αρχιτεκτονική της MEMSTAGE, σκοπός της οποίας είναι να ελεγχθεί η ύπαρξη δεδομένων σε μια διεύθυνση της μνήμης. Πρακτικά πρόκειται για άλλο ένα κύκλωμα, το οποίο επιτρέπει στον controller να γνωρίζει αν μπορεί ή όχι να γράψει δεδομένα σε μια συγκεκριμένη θέση.

Μέσα στο πορτοκαλί πλαίσιο της επόμενης εικόνας γίνεται ο υπολογισμός των διευθύνσεων της μνήμης και αντίστοιχα με επόμενες διεργασίες αυτού του κώδικα, αυτές οι διευθύνσεις ελέγχονται, όπως αναφέρθηκε και παραπάνω.

Η διεργασία Mux_MM_WrData (πράσινο βέλος) είναι υπεύθυνη για να υποδείξει αν γράφτηκαν δεδομένα ή όχι σε μια περιοχή της μνήμης.

Η διεργασία Mux_MEM_DataOut (μπλε βέλος) είναι υπεύθυνη για να υποδείξει αν διαβάστηκαν ή όχι δεδομένα από μια περιοχή της μνήμης.



```
architecture Behavioral of MEMSTAGE is

component Adder is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
         B : in  STD_LOGIC_VECTOR (31 downto 0);
         Cin : in  STD_LOGIC;
         Cout : out  STD_LOGIC;
         Ovf : out  STD_LOGIC;
         S : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

begin

  MM_WrEn <= Mem_WrEn;

  -- pros8etw to 1024 * 4 (0x400<<2)
  Add_offset: Adder PORT MAP (
        A => ALU_MEM_Addr,
        B => X"00001000",
        Cin => '0',
        S => MM_Addr
      );

  Mux_MM_WrData: process (MEM_DataIn, ByteOp)
  begin
        if ByteOp = '0' then
                MM_WrData <= MEM_DataIn;
        else
                MM_WrData <= X"000000" & MEM_DataIn(7 downto 0);
        end if;
  end process;

  Mux_MEM_DataOut: process (MM_RdData, ByteOp)
  begin
        if ByteOp = '0' then
                MEM_DataOut <= MM_RdData;
        else
                MEM_DataOut <= X"000000" & MM_RdData(7 downto 0);
        end if;
  end process;

 end Behavioral;
```


Εικόνα 5.6: Η MEMSTAGE αρχιτεκτονική

Στο σημείο αυτό καλό είναι να διευκρινιστούν 2 όροι που απαντώνται ευρέως στον κώδικα του επεξεργαστή. Το DATAPATH λέει στον επεξεργαστή αν πρέπει να διαβάσει στη μνήμη ή στους καταχωρητές. Διευκρινίζεται ότι και οι τελευταίοι είναι ένα είδος μνήμης για την αποθήκευση προσωρινών τιμών.

Το PC_sel ενημερώνει τον controller σε ποια εντολή βρισκόμαστε κάθε φορά. Κάθε φορά που αυτό αλλάζει, τροποποιείται και η οδηγία (instruction), μέχρις ότου να μην υπάρχει κάποια οδηγία σε αναμονή για εκτέλεση.

5.7 Βαθμίδα πρόσβασης στη μνήμη

Η υλοποίηση αυτού του μέρους της εργασίας πραγματοποιείται από τον κώδικα που βρίσκεται στο παράρτημα Ζ και τα αποτελέσματα αυτής της υλοποίησης αντικατοπτρίζονται στο παρακάτω διάγραμμα, στο οποίο εμφανίζονται οι μεταβολές στην κατάσταση της μνήμης. Στην εικόνα που ακολουθεί μέσα στο μπλε πλαίσιο σημειώνεται το Port μέσα στο οποίο περιλαμβάνεται η διεύθυνση των δεδομένων, τα in και out, καθώς και αν πρόκειται για εγγραφή.

Παράρτημα Ζ

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity MEM is
   Port ( clk : in  STD_LOGIC;
          inst_addr : in  STD_LOGIC_VECTOR (10 downto 0);
          inst_dout : out  STD_LOGIC_VECTOR (31 downto 0);
          data_we : in  STD_LOGIC;
          data_addr : in  STD_LOGIC_VECTOR (10 downto 0);
          data_din : in  STD_LOGIC_VECTOR (31 downto 0);
          data_dout : out  STD_LOGIC_VECTOR (31 downto 0));
end MEM;

architecture syn of MEM is
   type ram_type is array (2047 downto 0) of std_logic_vector (31 downto 0);
   impure function InitRamFromFile (RamFileName : in string) return ram_type is
   FILE ramfile : text is in RamFileName;
   variable RamFileLine : line;
   variable ram : ram_type;
   begin
     for i in 0 to 1023 loop
        readline(ramfile, RamFileLine);
        read (RamFileLine, ram(i));
     end loop;
     for i in 1024 to 2047 loop
        ram(i) := x"00000000";
     end loop;
```

Λειτουργίες μνήμης

Εγγραφή Μνήμης

Μέσο ανάγνωσης αρχείου

Καταχώρηση '0..0'

Εικόνα 5.7: Πρόσβαση στη μνήμη

Με τα κίτρινα βέλη σημειώνεται ο τρόπος με τον οποίο ορίζεται το RAM. Υπάρχει η δυνατότητα το περιεχόμενο του, είτε να γεμίσει διαβάζοντας κάποιο αρχείο (επάνω βέλος), είτε να περιλαμβάνει μόνο μηδενικά (κάτω βέλος).

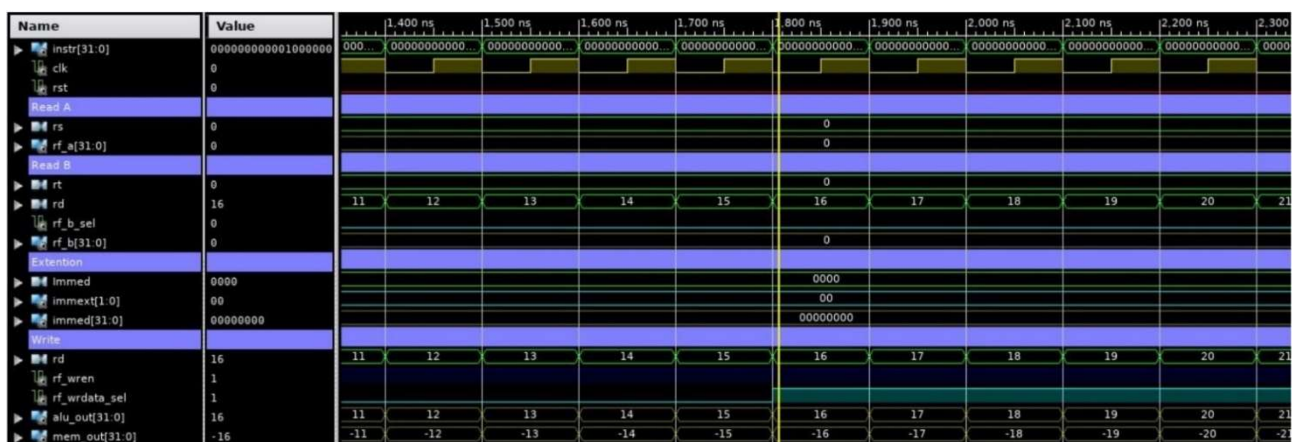
Κάθε φορά που δίνεται ένα σήμα write, η εγγραφή πραγματοποιείται σε συγκεκριμένη διεύθυνση η οποία μπορεί να έχει δηλωθεί εξαρχής ή να έχει υπολογιστεί μετέπειτα. Πριν από την εγγραφή, προηγείται η εκτέλεση μιας πράξης η οποία απαιτεί την ανάγνωση από τη μνήμη συγκεκριμένων τιμών και παραμέτρων.

Για την υλοποίηση αυτού του μέρους θεωρούμε ότι ο διαθέσιμος χώρος για τα data βρίσκεται από την θέση 1024 της μνήμης και έπειτα, καθώς ο κώδικας της RAM αρχικοποιεί την μνήμη από το αρχείο rom.data μέχρι και την 1023. Έτσι προσθέσαμε το 1024 στην διεύθυνση λέξης. Άρα στην υλοποίησή η διεύθυνση 0 στο ALU_MEM_Addr μεταφράζεται ως 1024 στην MM_Addr.

5.8 Καταχωρητής

Ο κώδικας που βρίσκεται στο παράρτημα Η της εργασίας αναπαριστά έναν καταχωρητή. Αρχικά στην οντότητα Register (πορτοκαλί βέλος), γίνεται η αρχικοποίηση των απαραίτητων παραμέτρων. Στη συνέχεια μέσα στο πράσινο πλαίσιο περιλαμβάνεται η βασική λειτουργία του καταχωρητή. Αυτό σημαίνει, είτε ότι το περιεχόμενο του επαναφέρεται – μηδενίζεται (reset), είτε διατηρείται το περιεχόμενο του προσωρινά. Όλη αυτή η διεργασία, εντός του πλαισίου είναι ευαίσθητη στις αλλαγές του ρολογιού (clock sensitive), και κάθε φορά που υπάρχει αλλαγή στο σήμα του επαναλαμβάνεται ξανά.

Γενικότερα η λειτουργία των καταχωρητών είναι απλή και όπως έχει αναφερθεί και παραπάνω συμβάλλουν στην προσωρινή αποθήκευση τιμών. Οι καταχωρητές αποτελούν και αυτοί μια μορφή προσωρινής μνήμης.



Διάγραμμα 5.7: Κατάσταση της μνήμης

Παράρτημα Η

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


entity REG is
   Port ( RST : in  STD_LOGIC;
          CLK : in  STD_LOGIC;
          WE : in  STD_LOGIC;
          Datain : in  STD_LOGIC_VECTOR (31 downto 0);
          Dataout : out  STD_LOGIC_VECTOR (31 downto 0));
end REG;

architecture  Behavioral of REG is

signal reg_out: STD_LOGIC_VECTOR (31 downto 0);

begin

        reg_logic: process (CLK)
        begin
                if CLK'event and CLK = '1' then
                        if RST = '1' then
                                reg_out <= (others => '0');
                        elsif WE = '1' then
                                reg_out <= Datain;
                        end if;
                end if;
        end process;

        Dataout <= reg_out after 10 ns;

end Behavioral;
```

Αρχικοποίηση παραμέτρων Καταχωρητή

Βασική λειτουργία Καταχωρητή

Εικόνα 5.8: Η αρχικοποίηση ενός καταχωρητή

5.9 RF οντότητα

Η RF οντότητα αναπαριστάται από τον κώδικα στο παράρτημα Θ και αποτελείται από 2 βασικά συστατικά (components), τον Decoder και από το Register. Το πρώτο από αυτά λαμβάνει ως είσοδο διευθύνσεις 5 bit

και παράγει εξόδους 32 bit. Το δεύτερο συστατικό αντιπροσωπεύει ένα βασικό καταχωρητή με είσοδο (Datain) και έξοδο (Dataout) μεγέθους 32 bit, καθώς και σήματα ενεργοποίησης ρολογιού και εγγραφής.

Αρχικά στην επόμενη εικόνα σημειώνονται με πορτοκαλί βέλη τα 2 components που αναφέρθηκαν παραπάνω. Με μπλε βέλος σημειώνεται το array3232 το οποίο είναι ένας πίνακας 32 στοιχείων, καθένα από τα οποία περιλαμβάνει ένα διάνυσμα 32-bit. Η γραμμή με το πράσινο βέλος περιλαμβάνει μια σειρά από σήματα:

- reg_array: Πίνακας καταχωρητών, με κάθε καταχωρητή να έχει μέγεθος 32 bit.
- dec_out: Έξοδος 32-bit που προκύπτει από το decoder component.
- and_out: αυτή η εκχώρηση σήματος πραγματοποιεί τη λογική πράξη AND μεταξύ του dec_out και του WrEn_vector.
- WrEn_vector: διάνυσμα 32-bit δημιουργείται χρησιμοποιώντας την τιμή του WrEn.
- Rst_vector: Διάνυσμα 32 bit που δημιουργήθηκε λαμβάνοντας υπόψη την τιμή του Reset.

Παράρτημα Θ

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


use IEEE.NUMERIC_STD.ALL;



entity RF is
   Port ( Ard1 : in  STD_LOGIC_VECTOR (4 downto 0);
        Ard2 : in  STD_LOGIC_VECTOR (4 downto 0);
        Awr : in  STD_LOGIC_VECTOR (4 downto 0);
        Dout1 : out  STD_LOGIC_VECTOR (31 downto 0);
        Dout2 : out  STD_LOGIC_VECTOR (31 downto 0);
        Din : in  STD_LOGIC_VECTOR (31 downto 0);
        WrEn : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        Rst : in  STD_LOGIC);
end RF;

architecture Behavioral of RF is

component Decoder is
   Port ( Awr : in  STD_LOGIC_VECTOR (4 downto 0);
        Output : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component REG is
   Port ( RST : in  STD_LOGIC;
        CLK : in  STD_LOGIC;
        WE : in  STD_LOGIC;
        Datain : in  STD_LOGIC_VECTOR (31 downto 0);
        Dataout : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

type array3232 is array (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
signal reg_array: array3232;
signal dec_out, and_out, WrEn_vector, Rst_vector: STD_LOGIC_VECTOR (31 downto 0);
```


Εικόνα 5.9: RF οντότητα

Ολοκληρώνοντας παρατηρείται ότι στο τέλος του κώδικα δημιουργείται ένα μπλοκ καταχωρητών (registers) οι οποίοι είναι υπεύθυνοι για την κατασκευή 32 περιπτώσεων (instances) για το συστατικό REG. Κάθε μία περίπτωση συνδέεται με τα αντίστοιχα στοιχεία των πινάκων και των σημάτων που αναφέρθηκαν παραπάνω.

Σχετικά με τα μπλοκ Mux_dout1 και μπλοκ Mux_dout2 που παρατηρούνται μετά από το μπλοκ των καταχωρητών διευκρινίζεται ότι πρόκειται για μπλοκ που χρησιμοποιούνται στην πολυπλεξία των δεδομένων εξόδου (Dout1 και Dout2) με βάση τις τιμές των Ard1 και Ard2 για την επιλογή των κατάλληλων δεδομένων από τη συστοιχία reg_array.

5.10 EXSTAGE & IFSTAGE οντότητα

Πρόκειται για το παράρτημα I του κώδικα και οι οντότητες που περιλαμβάνονται σε αυτό, το EXSTAGE και το IFSTAGE, αντιπροσωπεύουν το στάδιο εκτέλεσης και το στάδιο ανάκτησης εντολών, αντίστοιχα, στο επίπεδο του επεξεργαστή.

Η αρχιτεκτονική EXSTAGE είναι υπεύθυνη αρχικά για τη δημιουργία ενός στοιχείου, χρησιμοποιώντας την ALU, το οποίο κατέχει και τις συνδέσεις με τις αντίστοιχες θύρες. Επιπλέον περιέχει τη διεργασία Mux_AluB που επιλέγει την είσοδο B για την ALU με βάση την τιμή του ALU_Bin_sel και την εκχωρεί στο σήμα B.

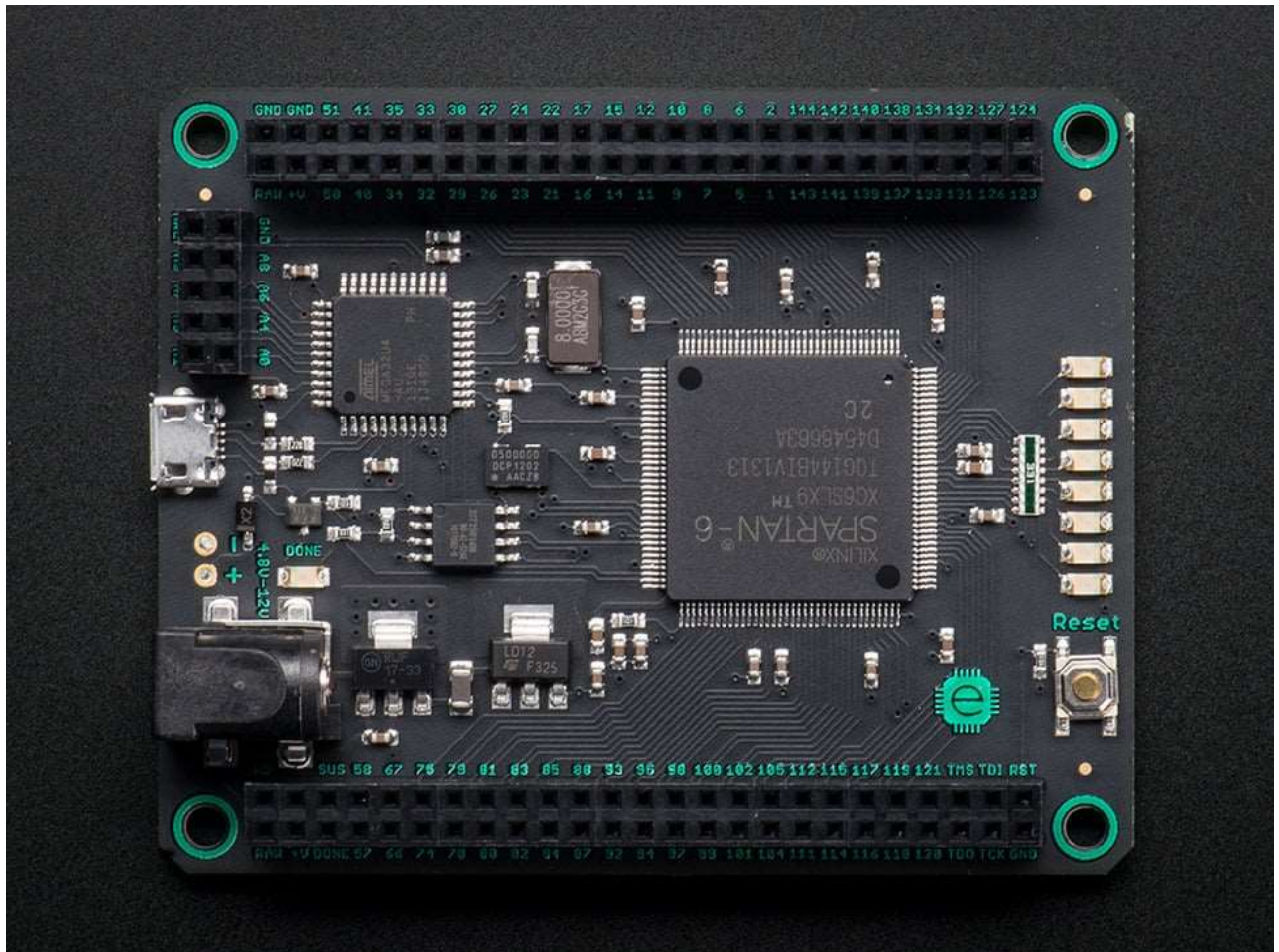
Η αρχιτεκτονική IFSTAGE κάνει τα εξής δημιουργεί τα στοιχεία REG και Adder με συνδέσεις στις αντίστοιχες θύρες. Εκτελεί την πράξη της πρόσθεσης για να αυξήσει την τιμή του PC καταχωρητή κατά 4 και στη συνέχεια προσθέσει την άμεση τιμή στο προηγούμενο αποτέλεσμα. Διευκρινίζεται ότι ως PC εννοείται ένας καταχωρητής για την προσωρινή αποθήκευση τιμών. Τέλος χρησιμοποιεί μια διαδικασία πολυπλέκτη Mux για να επιλεχθεί η κατάλληλη είσοδος (PC_in) για τον PC καταχωρητή με βάση την τιμή του PC_sel.

Στο παραπάνω διάγραμμα αποτελεσμάτων αναπαρίσταται η λειτουργία των προηγούμενων οντοτήτων μέσα από την εκτέλεση πράξεων. Στα αριστερά του διαγράμματος δίνονται τα ονόματα των οντοτήτων και των μεταβλητών. Στη συνέχεια, ακριβώς δεξιότερα ορίζονται οι τιμές τους (values). Έπειτα στο υπόλοιπο διάγραμμα υπάρχει η γραφική αναπαράσταση, η οποία στο επάνω μέρος της (πράσινα πλαίσια) περιλαμβάνει τους αριθμούς που δίνονται ως είσοδοι και στο κάτω μέρος της (κίτρινα πλαίσια) τους αντίστοιχους τελεστές που χρησιμοποιήθηκαν. Διευκρινίζεται ότι οι γραμμές που έχουν πλάτος αντιστοιχούν σε κάποια λειτουργία που εκτελείται ή υπάρχει ή κάποιο σήμα έχει την τιμή 1. Αντίθετα οι γραμμές δίχως πλάτος δείχνουν ότι η τιμή ενός σήματος είναι 0 ή ότι μια λειτουργία δεν εκτελέστηκε. Κατά μήκος των γραμμών παρατηρούνται οι μεταβολές στο πέρασμα του χρόνου.

Διάγραμμα 5.8: IFSTAGE, DECSTAGE, EXSTAGE, MEMSTAGE οντότητες

5.11 Το PCB του Mojo v.3 και η δομή του

Το Mojo v.3 αποτελεί μία ολοκληρωμένη λύση που προσφέρεται σε μία μόνο πλακέτα PCB. Η χρήση του για πρωτότυπες εφαρμογές και εξοικείωση με τα FPGAs είναι καθοριστική καθώς δίνεται η δυνατότητα προγραμματισμού του FPGA της Xilinx Spartan 6. Το Mojo IDE παρέχεται ως ελεύθερο λογισμικό και μπορεί κανείς να προγραμματίσει σε αυτό , την πλακέτα. Ο προγραμματισμός επιτυγχάνεται με τη βοήθεια του μικροελεγκτή ATMEGA32U4 που μεσολαβεί στην επικοινωνία από τη USB θύρα έως το FPGA. Γενικά, στην πλακέτα υπάρχουν ελεύθερα αρκετά IOs pins, 8 leds, button για reset και θύρα εξωτερικής τροφοδοσία για standalone χρήση.



Εικόνα 5.10: Mojo v.3 PCB

Με την αρχικοποίηση ενός project στο περιβάλλον Mojo IDE, το αρχείο mojo_top.v καθορίζει την βασική δομή του συστήματος, δηλαδή τα σήματα επικοινωνίας με τον ATMEGA (π.χ. SPI σήματα επικοινωνίας) και τα leds που μπορούν να ανάβουν ή να σβήνουν κατά το δοκούν (όταν λαμβάνει τιμή 1 ο buffer led στο index i, το αντίστοιχο led ανάβει, ενώ για τη τιμή 0 σβήνει). Η αρχιτεκτονική της ALU υλοποιείται με διαφορετικό τρόπο σχετικά, κυρίως εξαιτίας του διαφορετικού τρόπου σύνταξης της Verilog και των διαθέσιμων CPLDs του FPGA για λογικές πράξεις.

Επομένως, η αρχιτεκτονική που υλοποιείται στο πρακτικό μέρος που αφορά το Mojo v.3, φέρει τις παρακάτω υπομονάδες :

- ✓ And των 32 bits
- ✓ Or των 32 bits
- ✓ Xor των 32 bits
- ✓ Shift του 1 bit αριστερά για μήκος λέξης 32 bits

- ✓ Shift του 1 bit δεξιά για μήκος λέξης 32 bits
- ✓ Αθροιστής των 32 bits
- ✓ Αφαιρέτης των 32 bits
- ✓ Πολλαπλασιαστής των 16 bits προκειμένου να δώσει ως έξοδο 32 bit λέξη στην οριακή περίπτωση που πολλαπλασιάζονται οι μεγαλύτεροι αριθμοί μεταξύ τους.

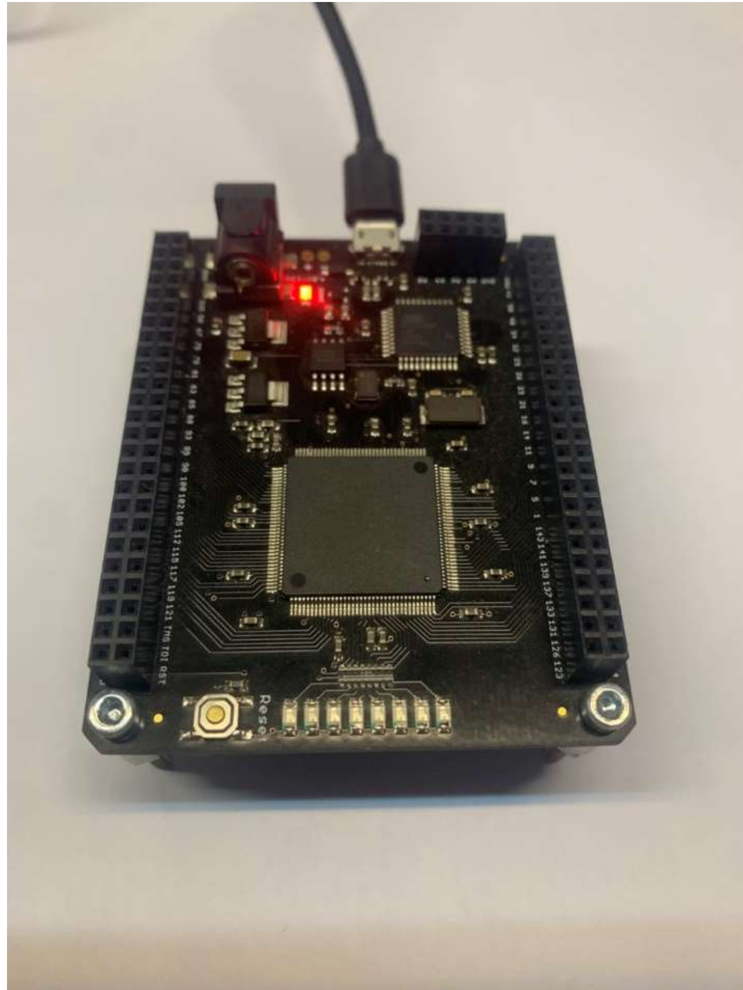
Η λογική της εξόδου της αρχιτεκτονικής αυτής, προκειμένου να κριθεί η ορθότητά της, βασίζεται στη χρήση των leds. Επομένως, μετά το τέλος των πράξεων, τα αποτελέσματα αφορούν στα 8 λιγότερο σημαντικά ψηφία των λέξεων εισόδου / εξόδου για τις περιπτώσεις των 32 bits λέξεων, και στα 4 λιγότερο σημαντικά ψηφία των 16 bits εισόδων για τη περίπτωση του πολλαπλασιασμού και μόνο.

Οι πράξεις εκτελούνται όλες και αποφασίζεται σύμφωνα με την εντολή Opcode εάν θα καταχωρηθεί στην μνήμη/διαβαστεί και επίσης επιλέγεται η αντίστοιχη τιμή ανάλογα με την πράξη που θέλουμε να γίνει. Συνεπώς, τα υποσυστήματα που περιγράφονται παραπάνω παράγουν εξόδους των 32 bits οι οποίες καταλήγουν στην μονάδα Mux4, όπου το Opcode αποφασίζει την εκάστοτε λειτουργεία:

- Το MSB του Opcode για τιμή 0 δηλώνει πως το αποτέλεσμα της πράξης απλά καταχωρείται σε register και δεν εμφανίζεται κάποιο αποτέλεσμα (άρα όλα τα leds είναι σβηστά). Για τιμή 1, δηλώνει πως το αποτέλεσμα όχι μόνο καταχωρείται, αλλά διαβάζεται παράλληλα, οπότε τα leds δείχνουν το αποτέλεσμα που βγάζει η πράξη.
- Τα 3 υπόλοιπα bits του Opcode καθορίζουν το αποτέλεσμα που έχουμε ως έξοδο ανάλογα με την επιθυμητή πράξη. Επομένως, 8 διαφορετικές πράξεις μπορούν να εκτελεσθούν σύμφωνα με τα διαθέσιμα υπο-blocks που δηλώσαμε αρχικά (καταστάσεις 000 έως 111). Συνεπώς η εντολή 0011 σημαίνει πως εκτελείται η πράξη Shift32R (μετακίνηση ένα ψηφίο δεξιά) χωρίς διάβασμα της τιμής στην έξοδο ενώ η εντολή 1101 δηλώνει πως εκτελείται η πράξη άθροισης με διάβασμα της τιμής στην έξοδο.

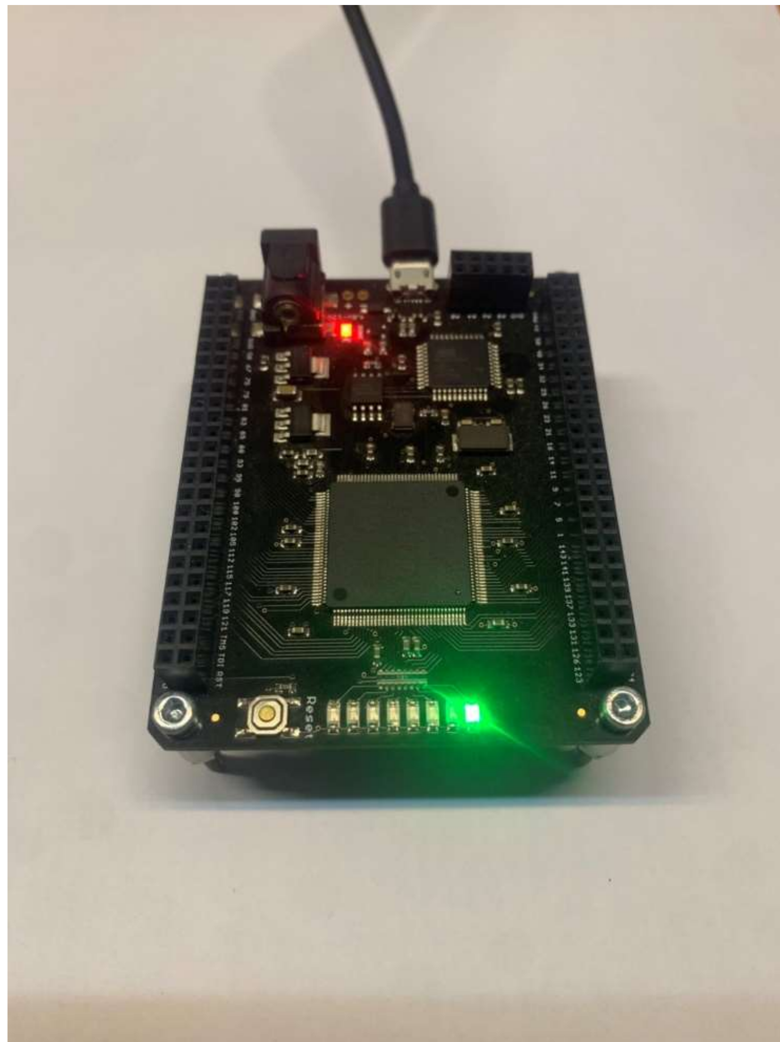
Τέλος, η δομή map32to8.v εκτελεί την καταχώρηση των 8 λιγότερο σημαντικών ψηφιών εξόδου στον αντίστοιχο buffer. Αυτό γίνεται έτσι ώστε να μπορούμε στο τέλος να εμφανίσουμε τα 8 bits αποτελέσματα στα leds. Τα σήματα εισόδου όπως το κρατούμενο και οι buffers με τις λέξεις των 32 bits και 16 bits για πολλαπλασιασμό, μπορούν να αρχικοποιηθούν στο mojo_top.v ενώ όλα τα υπόλοιπα συστήματα υλοποιούνται σε ξεχωριστά Verilog αρχεία των οποίων τους κώδικες μπορεί κανείς να συμβουλευτεί από το αντίστοιχο παράρτημα. Επιπλέον, στο mojo_top.v τοποθετείται και η τιμή του opcode που θα καθορίσει το αποτέλεσμα κάθε εξομοίωσης. Γενικά, δοκιμάζονται διάφορες τιμές και παρακάτω ακολουθούν ενδεικτικά αποτελέσματα.

```
40 initial begin
41  a = 32'b00000000000000000000000001010101;
42  b = 32'b00000000000000000000000010101011;
43  mula = 16'b0000000000001111;
44  mulb = 16'b0000000000001111;
45  cin = 1'b0;
46  opcode = 4'b0000;
47 end
```



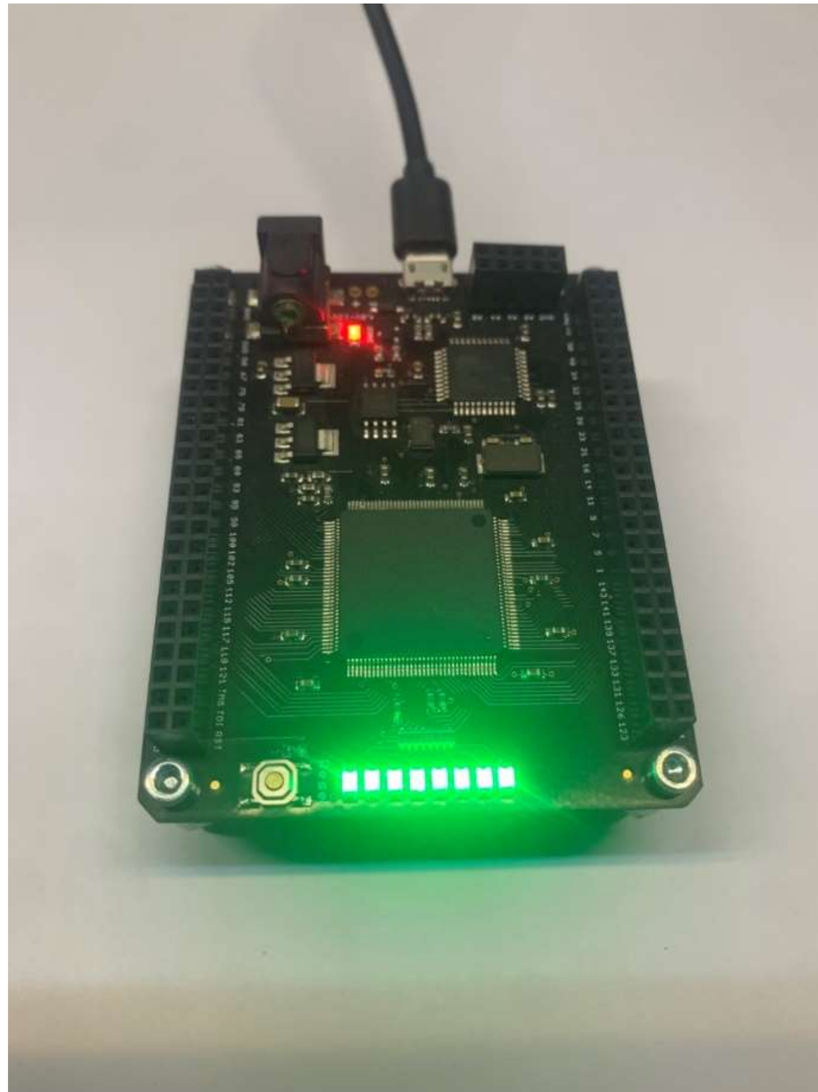
Η εντολή Opcode δηλώνει εμμέσως την επιλογή μας να αποθηκεύεται το αποτέλεσμα στον register και να διαβάζεται (MSB με τιμή 1) ή να μην διαβάζεται (MSB με τιμή 0). Στο παράδειγμα παραπάνω δεν ανοίγουν τα leds επειδή η εντολή δηλώνει 0 στην τιμή του MSB, δηλαδή δεν διαβάζεται ο register.

```
40 initial begin
41  a = 32'b00000000000000000000000001010101;
42  b = 32'b00000000000000000000000010101011;
43  mula = 16'b0000000000001111;
44  mulb = 16'b0000000000001111;
45  cin = 1'b0;
46  opcode = 4'b1000;
47 end
```



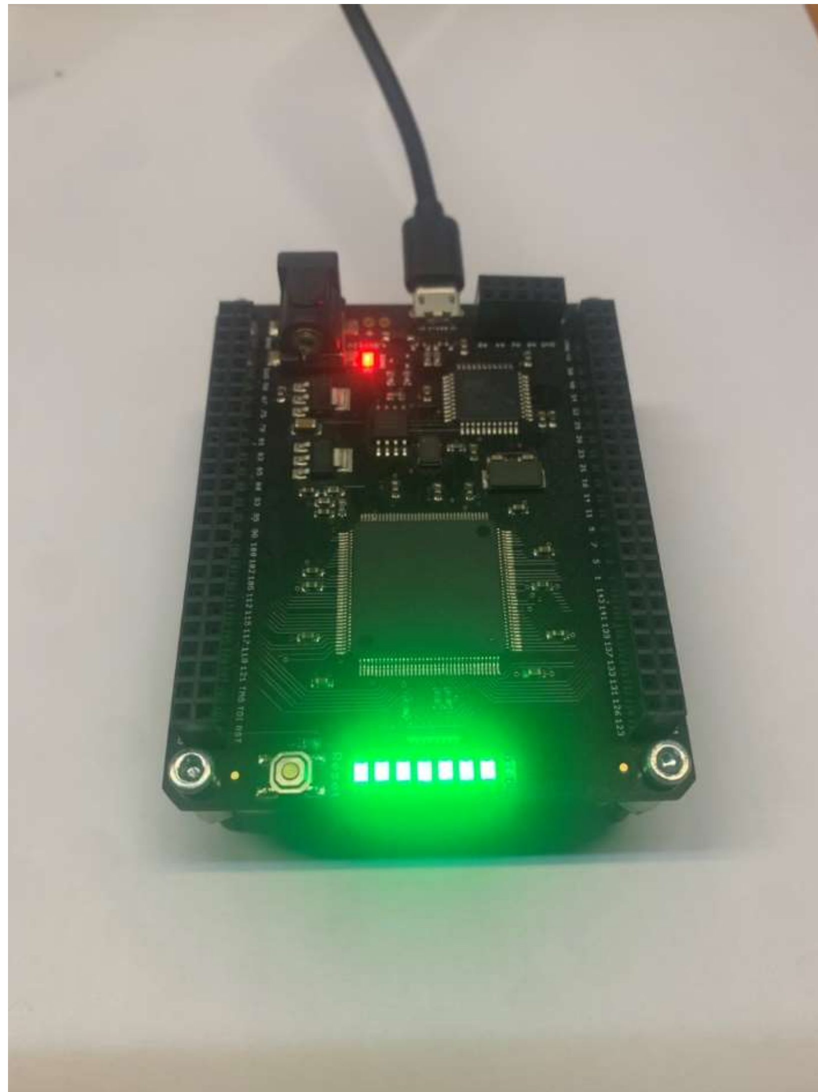
Αντίθετα με το προηγούμενο παράδειγμα, εδώ εμφανίζεται το αποτέλεσμα της AND λογικής. Συγκρίνοντας ένα προς ένα τα bits, βλέπουμε το πρώτο bit να είναι λογικό 1 (αναμμένο led).

```
40 initial begin
41   a = 32'b00000000000000000000000001010101;
42   b = 32'b00000000000000000000000010101010;
43   mula = 16'b0000000000001111;
44   mulb = 16'b0000000000001111;
45   cin = 1'b0;
46   opcode = 4'b1001;
47 end
```



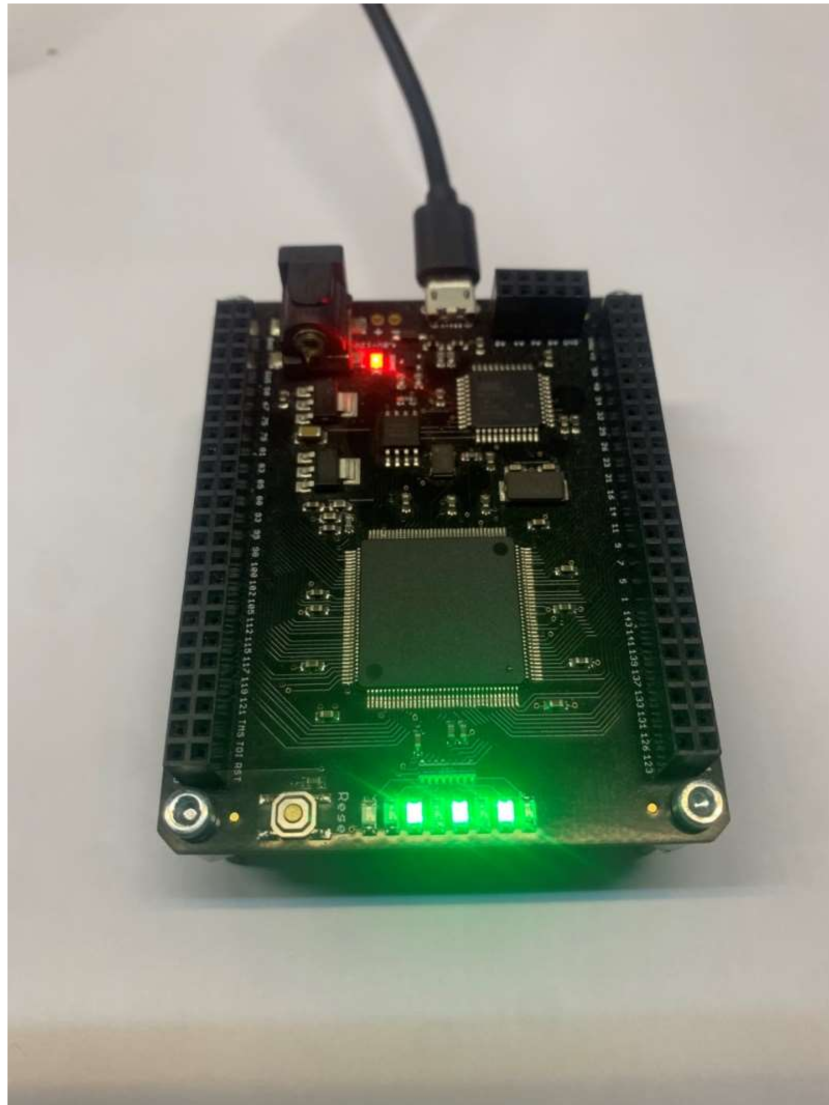
Το παράδειγμα αυτό αποδεικνύει την λογική OR των 8 bits.

```
40 initial begin
41   a = 32'b00000000000000000000000001010101;
42   b = 32'b00000000000000000000000010101011;
43   mula = 16'b0000000000001111;
44   mulb = 16'b0000000000001111;
45   cin = 1'b0;
46   opcode = 4'b1010;
47 end
```



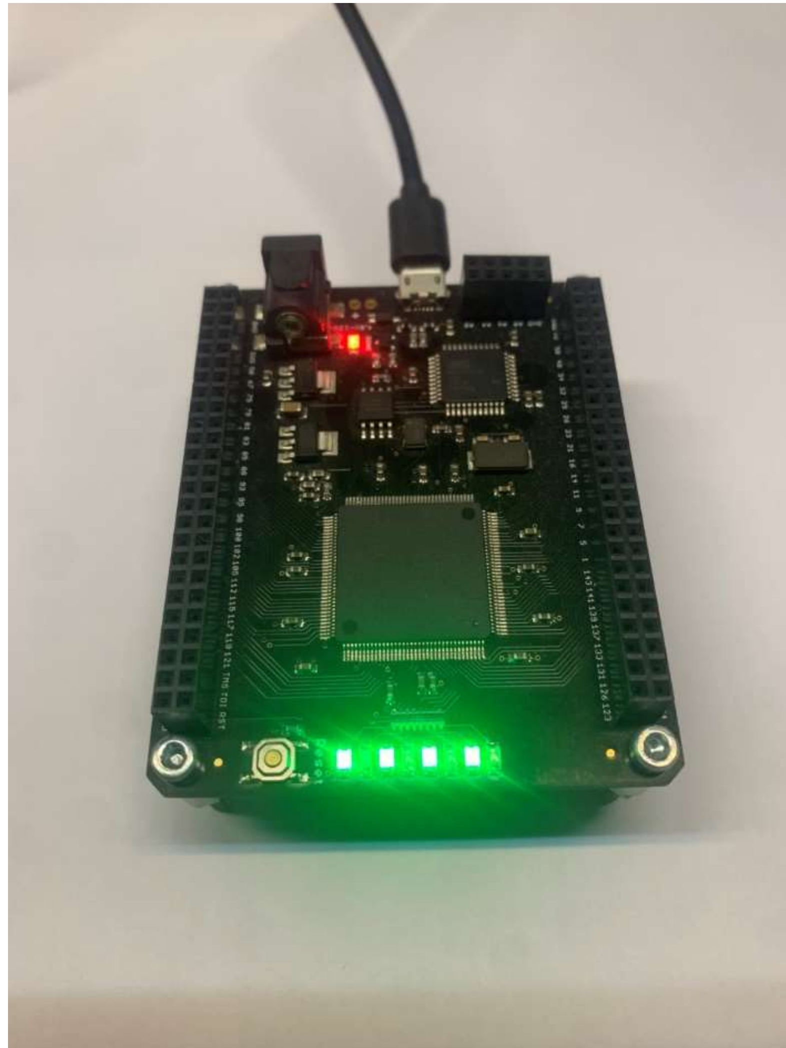
Το παράδειγμα αυτό αποδεικνύει την λογική XOR των 8 bits. Τα bits με ίδια τιμή δίνουν αποτέλεσμα 0 ενώ με διαφορετική δίνουν αποτέλεσμα 1.

```
40 initial begin
41  a = 32'b00000000000000000000000001010101;
42  b = 32'b00000000000000000000000010101011;
43  mula = 16'b0000000000001111;
44  mulb = 16'b0000000000001111;
45  cin = 1'b0;
46  opcode = 4'b1011;
47 end
```



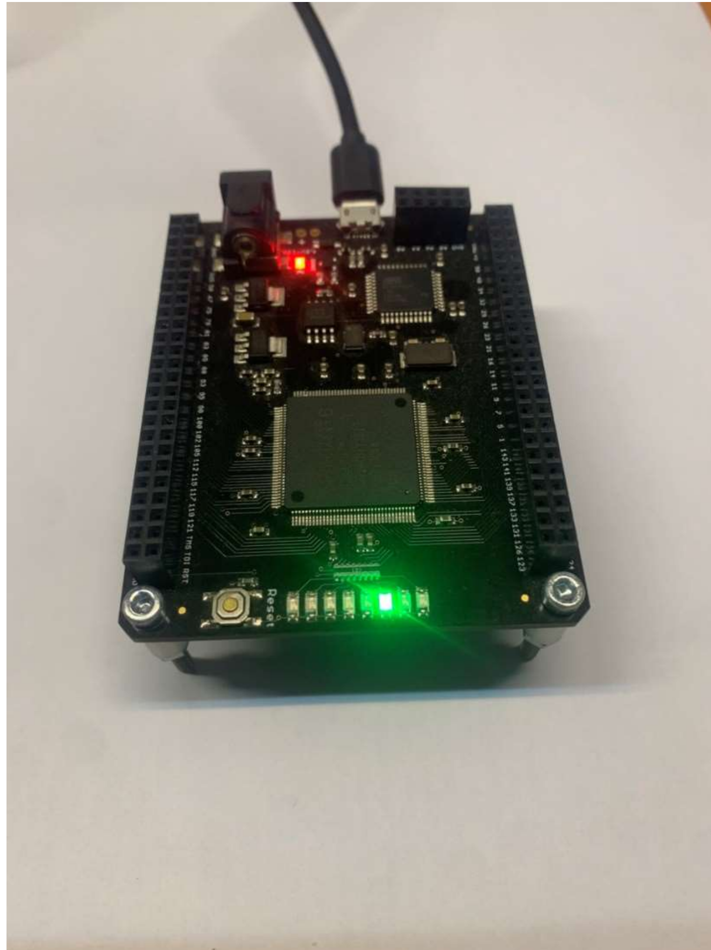
Το παράδειγμα αυτό αποδεικνύει την λογική μετατόπισης δεξιά κατά 1 bit.

```
40 initial begin
41   a = 32'b00000000000000000000000001010101;
42   b = 32'b00000000000000000000000010101011;
43   mula = 16'b0000000000001111;
44   mulb = 16'b0000000000001111;
45   cin = 1'b0;
46   opcode = 4'b1100;
47 end
```



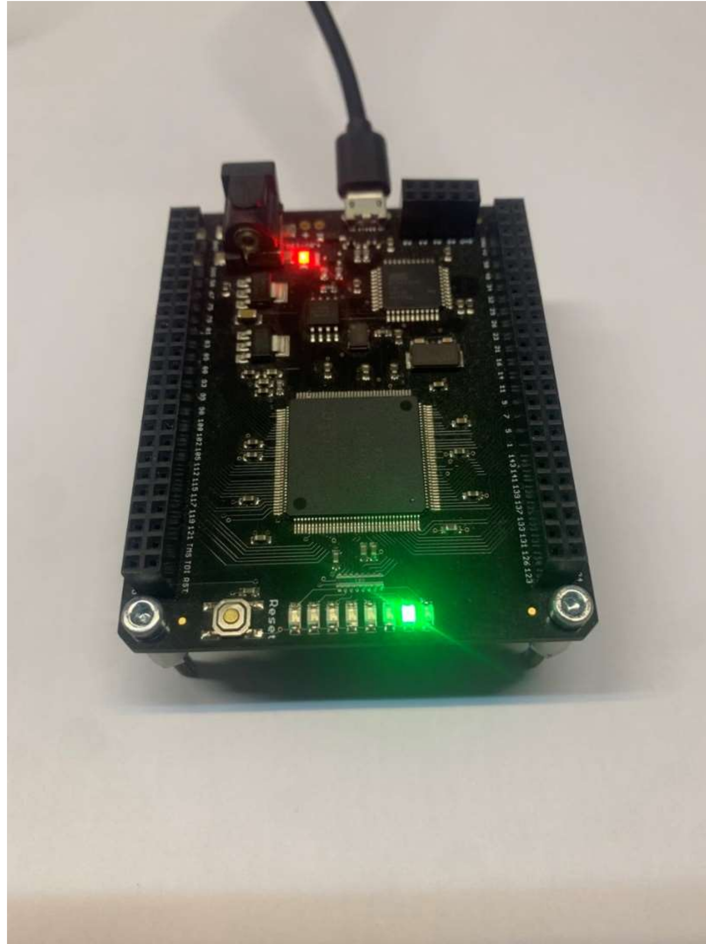
Το παράδειγμα αυτό αποδεικνύει την λογική μετατόπισης αριστερά κατά 1 bit.

```
40 initial begin
41  a = 32'b00000000000000000000000000000001;
42  b = 32'b00000000000000000000000000000011;
43  mula = 16'b0000000000001111;
44  mulb = 16'b0000000000001111;
45  cin = 1'b0;
46  opcode = 4'b1101;
47 end
```



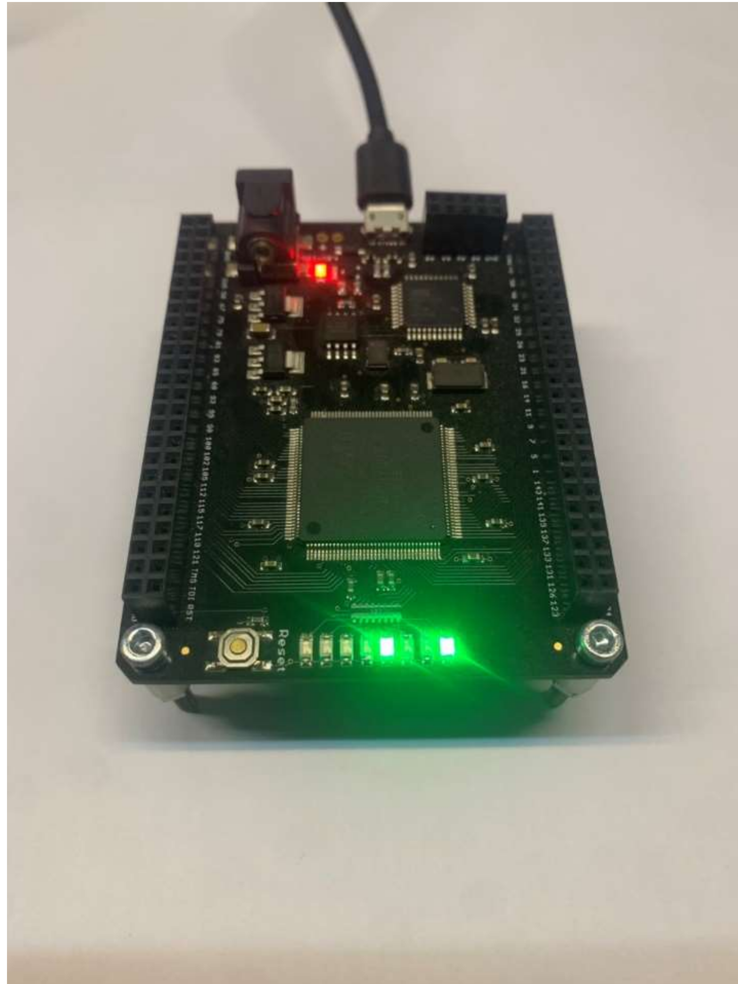
Το παράδειγμα αυτό αποδεικνύει την λογική αθροίσματος λέξεων μήκους 8 bits. Για το παράδειγμα θεωρείται κρατούμενο εισόδου ίσο με 0.

```
40 initial begin
41  a = 32'b00000000000000000000000000000011;
42  b = 32'b00000000000000000000000000000001;
43  mula = 16'b0000000000001111;
44  mulb = 16'b0000000000001111;
45  cin = 1'b0;
46  opcode = 4'b1110;
47 end
```



Το παράδειγμα αυτό αποδεικνύει την λογική αφαίρεσης λέξεων μήκους 8 bits.

```
40 initial begin
41  a = 32'b00000000000000000000000000000011;
42  b = 32'b00000000000000000000000000000001;
43  mula = 16'b0000000000000011;
44  mulb = 16'b0000000000000011;
45  cin = 1'b0;
46  opcode = 4'b1111;
47 end
```



Το παράδειγμα αυτό αποδεικνύει την λογική πολλαπλασιασμού λέξεων μήκους 4 bits. Αυτό γίνεται έτσι ώστε σε ακραία περίπτωση να προκύψει αποτέλεσμα με μήκος 8 bits που να μπορεί να αποτυπωθεί στα leds που υπάρχουν στο PCB.

6 Συμπεράσματα

Με την ολοκλήρωση αυτής της εργασίας είναι σημαντικό να επισημανθούν όλα τα χρήσιμα αποτελέσματα που προέκυψαν από τη συγγραφή της. Ένας από τους βασικούς στόχους της εργασίας, ο οποίος και επιτεύχθηκε ήταν η κατανόηση της σημαντικότητας της γλώσσας VHDL στη σχεδίαση των ψηφιακών συστημάτων. Μια παράλληλη γλώσσα προγραμματισμού με πολλά στοιχεία και λειτουργίες, που παρέχει ευελιξία στον προγραμματισμό κυκλωμάτων σε υψηλότερο επίπεδο.

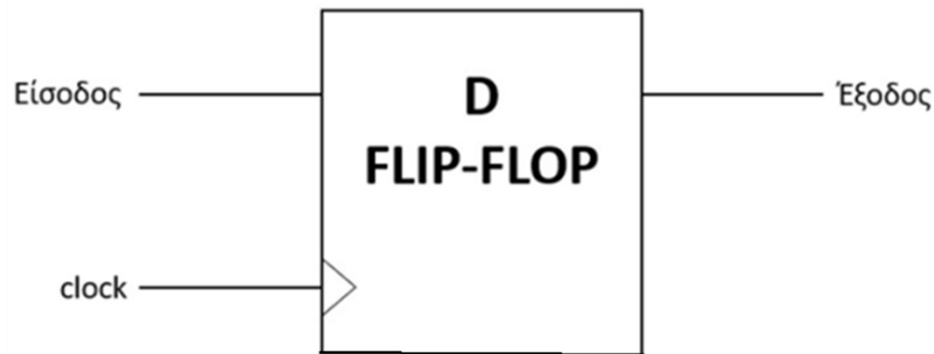
Η δομή της εργασίας είναι αναλυτική, συνεκτική και προσαρμοσμένη στην βασική κατεύθυνση, η οποία είναι η σχεδίαση – προσομοίωση ενός επεξεργαστή με χρήση της VHDL. Συνολικά αποτελείται από 6 ενότητες, με τις πρώτες από αυτές να αποτελούν το θεωρητικό γνωστικό υπόβαθρο σχετικά με το αντικείμενο μελέτης και την προτελευταία να αποτελεί τη δημιουργία στην πράξη ενός επεξεργαστή με χρήση της VHDL.

Μέσα από την εργασία επιβεβαιώνονται τα χαρακτηριστικά και οι λειτουργίες της γλώσσας που περιγράφονται στο τρίτο κεφάλαιο. Η ευελιξία και η χρηστικότητα της είναι εμφανή από τα ανεξάρτητα τμήματα κώδικα που γράφονται και μπορούν να συνθέσουν πολυπλοκότερες δομές – οντότητες.

Το πρακτικό τμήμα αυτής της εργασίας μπορεί να αξιολογηθεί από 2 σκοπιές. Η πρώτη είναι η ακολουθιακή πλευρά και η δεύτερη η συνδυαστική. Συμπεραίνεται πως όταν γίνεται αναφορά σε ακολουθιακή λογική γίνεται κατανοητό ότι πρόκειται για απλούστερες μορφές κώδικα με συγκεκριμένη λειτουργία. Όταν γίνεται αναφορά σε συνδυαστικές λογικές τότε η πολυπλοκότητα αυξάνεται και μπορεί να φτάσει στο επίπεδο του επεξεργαστή.

Σε μια συνδυαστική υλοποίηση το σύστημα αποτελείται από λογικά και αριθμητικά κυκλώματα και οι έξοδοι του δεν επηρεάζονται από τις αρχικές ή τις τρέχουσες εισόδους. Αυτές οι υλοποιήσεις συνήθως αποτελούνται από απλούστερα κυκλώματα, όπως είναι ο αθροιστής (Adder), οι αφαιρέτης (Subtractor) και ο αποκωδικοποιητής (Decoder). Η σχεδίαση και η διαχείρισή τους είναι εφικτή, αρκεί να είναι κατανοητός και σαφείς ο λόγος για τον οποίο χρειάζεται ένα κύκλωμα.

Στην ακολουθιακή υλοποίηση όλες οι μεταβάσεις μεταξύ των καταστάσεων γίνονται σειριακά. Αυτό σημαίνει ότι κάθε επόμενη κατάσταση είναι διαδοχική της τρέχουσας. Επίσης οι έξοδοι καθορίζονται από τις εισόδους του συστήματος. Αυτό καθιστά απαραίτητη την αποθήκευση τιμών σε κάποιο στοιχείο μνήμης. Ο επαναπροσδιορισμός του περιεχομένου της μνήμης γίνεται με βάση το ρολόι που διαθέτει το σύστημα. Η μνήμη αυτής της υλοποίησης θα μπορούσε να παρομοιαστεί με έναν καταχωρητή D flip – flop (Διάγραμμα 6.1), ο οποίος αποδίδει σε κάθε έξοδο το περιεχόμενο της τρέχουσας εισόδου. Ο καταχωρητής αυτός ανανεώνεται σε κάθε κύκλο του ρολογιού.



Διάγραμμα 6.1: Καταχωρητής flip – flop

Στον πίνακα που ακολουθεί συγκεντρώνονται τα χαρακτηριστικά των ακολουθιακών και των συνδυαστικών κυκλωμάτων.

Ακολουθιακά Κυκλώματα	**Συνδυαστικά Κυκλώματα**
Η έξοδος εξαρτάται από όλες τις προηγούμενες εισόδους.	Η έξοδος εξαρτάται μόνο από την τρέχουσα είσοδο.
Περιέχουν στοιχεία μνήμης.	Δεν περιέχουν στοιχεία μνήμης.
Περιέχουν σήματα ρολογιού.	Δεν περιέχουν σήματα ρολογιού.
Χαμηλή ταχύτητα εκτέλεσης, λόγω της πολυπλοκότητας.	Γρήγορα σε ταχύτητα εκτέλεσης.

Πίνακας 6.1: Ακολουθιακά & Συνδυαστικά κυκλώματα

Η προσομοίωση του επεξεργαστή που ήταν και ο αρχικός στόχος αυτής της εργασίας είναι περισσότερο κοντά με την ακολουθιακή υλοποίηση. Ο επεξεργαστής επιτελεί διάφορες λειτουργίες καθεμία με το δικό της σκοπό, οι οποίες τελικώς συμβάλλουν στην εκτέλεση πράξεων και στη λήψη αποφάσεων μέσα σε μερικά κλάσματα του δευτερολέπτου.

Ως επέκταση των δυνατοτήτων του επεξεργαστή που σχεδιάστηκε θα μπορούσε να θεωρηθεί η αύξηση της υπολογιστικής του ικανότητας. Για παράδειγμα αυτό θα μπορούσε να επιτευχθεί με την εκτέλεση πράξεων, χρησιμοποιώντας αριθμούς μεγαλύτερους των 32-bit. Κάτι τέτοιο ωστόσο θα απαιτούσε και περισσότερους πόρους, όπως για παράδειγμα μεγαλύτερο χώρο αποθήκευσης.

Εν κατακλείδι διευκρινίζεται ότι, ο τρόπος με τον οποίο σχεδιάστηκαν τα επιμέρους κυκλώματα θα μπορούσε να ήταν διαφορετικός, ωστόσο η τελική τους λειτουργία ήταν σαφώς καθορισμένη εξαρχής. Αυτή επιβεβαιώθηκε μέσω των πολλαπλών δοκιμών που έλαβαν χώρα στο λογισμικό της προσομοίωσης.

7 Βιβλιογραφία – Αναφορές - Διαδικτυακές Πηγές

1. ‘A Comparison of VHDL and Verilog HDL Through Design of a General-Purpose Microprocessor’, International Journal of Emerging Technologies and Innovative Research (www.jetir.org), ISSN:2349-5162, Vol.8, Issue 3, page no.2716-2722, March-2021.
2. G. E. Moore, "Progress in digital integrated electronics [Technical literature, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]," in IEEE Solid-State Circuits Society Newsletter, vol. 11, no. 3, pp. 36-37, Sept. 2006, doi: 10.1109/N-SSC.2006.4804410.
3. Cong, Jason & Wei, Peng & Yu, Hao & Zhang, Peng. (2018), ‘Automated accelerator generation and optimization with composable, parallel and pipeline architecture’, 1-6. 10.1145/3195970.3195999.
4. Markus Schütz. 1995, ‘How to efficiently build VHDL test benches’, In Proceedings of the conference on European design automation (EURO-DAC '95/EURO-VHDL '95), IEEE Computer Society Press, Washington, DC, USA, 554–559.
5. De, Outubro & Augusto, Nelson & Côrtes, Mario & Centoducatte, Paulo. (1996), ‘A CPU for Educational Applications Designed with VHDL and FPGA’.
6. Pedroni, Volnei A., ‘Circuit Design with VHDL’, Massachusetts, MIT Press, 2004.
7. Peter J. Ashenden, ‘Ψηφιακή σχεδίαση : Ενσωματωμένα συστήματα με VHDL’, Εκδόσεις Νέων Τεχνολογιών, 2010.
8. Καλομοίρος Ι., ‘Εισαγωγή στη γλώσσα VHDL’, Τεχνολογικό Εκπαιδευτικό Ίδρυμα Σερρών, 2012.
9. Vranesic Z. & Brown S., ‘Σχεδίαση Ψηφιακών Συστημάτων με τη γλώσσα VHDL’, Εκδόσεις Τζιόλα, 2011.
10. IEEE Standard VHDL Language Reference Manual (2008), Design Automation Standards Committee , IEEE Standard 1076-2008 (Revision of IEEE Standard 1076-2002).
11. Ouarda Hachour, “VHDL Circuits Hardware Description Language : Notes”, International Journal Of Circuits, Systems And Signal Processing, Issue: 5, Volume: 5, 2011.
12. Han, Xiaojing & Liu, Liang & Zhang, Zhe & Sun, Yufeng & Zhou, Jiahui & Cai, Hao. (2023). Design of a High Performance Vector Processor Based on RISIC-V Architecture. Journal of Physics: Conference Series. 2560. 012027. 10.1088/1742-6596/2560/1/012027.
13. Δασυγένης Μ., ‘Ενσωματωμένα Συστήματα Ενότητα 3: Η γλώσσα περιγραφής υλικού VHDL’, Εκπαιδευτικό Υλικό, Πανεπιστήμιο Δυτικής Μακεδονίας, 2016.

Ιστιογραφία (Last Access October 2023)

14. https://www.wsj.com/articles/BL-DGB-42647
15. https://www.redhat.com/sysadmin/cpu-components-functionality
16. https://www.fpga4student.com/

Παράρτημα Α

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Adder is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          Cin : in  STD_LOGIC;
          Cout : out  STD_LOGIC;
          Ovf : out  STD_LOGIC;
          S : out  STD_LOGIC_VECTOR (31 downto 0));
end Adder;
architecture Behavioral of Adder is
signal sum : STD_LOGIC_VECTOR (31 downto 0);
signal carry: STD_LOGIC_VECTOR (32 downto 0);
signal carry_in: INTEGER;

begin
	carry_in <= 1 when Cin = '1' else 0;
	sum <= STD_LOGIC_VECTOR (signed(A) + signed(B) + carry_in);
	S <= sum;
	carry <= STD_LOGIC_VECTOR (unsigned('0' & A) + unsigned('0' & B) + carry_in);
	Cout <= carry(32);
	ovf_logic: process (A, B, sum)
	begin
		Ovf <= '0';
		if A(31) = B(31) then
			if sum(31) /= A(31) then
				Ovf <= '1';
			end if;
		end if;
	end process;

end Behavioral;
```

Παράρτημα B – entity ALU

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

entity ALU is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          Op : in  STD_LOGIC_VECTOR (3 downto 0);
          Output : out  STD_LOGIC_VECTOR (31 downto 0);
          Zero : out  STD_LOGIC;
          Cout : out  STD_LOGIC;
          Ovf : out  STD_LOGIC);
end ALU;

architecture Behavioral of ALU is
component Adder is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          Cin : in  STD_LOGIC;
          Cout : out  STD_LOGIC;
          Ovf : out  STD_LOGIC;
          S : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

signal OpB, S, log, shift, Result: STD_LOGIC_VECTOR (31 downto 0);
signal Cin, Ovf_signal :STD_LOGIC;

begin


  Adder_module: Adder PORT MAP (
        A => A,
        B => OpB,
        Cin => Cin,
        Cout => Cout,
        Ovf => Ovf_signal,
        S => S
      );

  Operations: process (Op,A,B)
  begin
            if Op(2 downto 0) = "000" then
                      OpB <= B;
                      Cin <= '0';
                      shift <= A(31) & A(31 downto 1);
            elsif Op(2 downto 0) = "001" then
                      OpB <= not B;
                      Cin <= '1';
                      shift <= '0' & A(31 downto 1);
            elsif Op(2 downto 0) = "010" then
                      log <= A and B;
                      shift <= A(30 downto 0) & '0';
            elsif Op(2 downto 0) = "011" then
                      log <= A or B;
            elsif Op(2 downto 0) = "100" then
                      log <= not A;
```

```
                    shift <= A(30 downto 0) & A(31);
            elsif Op(2 downto 0) = "101" then
                    log <= A nand B;
                    shift <= A(0) & A(31 downto 1);
            elsif Op(2 downto 0) = "110" then
                    log <= A nor B;
            end if;
    end process;

    Result_logic: process (S, log, shift, Op)
    begin
            if Op(3 downto 1) = "000" then
                    Result <= S after 10 ns;
            elsif Op(3) = '0' then
                    Result <= log after 10 ns;
            else
                    Result <= shift after 10 ns;
            end if;
    end process;

    Zero_logic: process (Result)
    begin
            if Op(3 downto 0) = "000" then
                    if Result = X"00000000" then
                            Zero <= '1' xor Ovf_signal;
                    else
                            Zero <= '0' xor Ovf_signal;
                    end if;
            else
                    if Result = X"00000000" then
                            Zero <= '1';
                    else
                            Zero <= '0';
                    end if;
            end if;
    end process;

    Ovf_logic: process (Result)
    begin
            if Op(3 downto 1) = "000" then
                    Ovf <= Ovf_signal after 10 ns;
            else
                    Ovf <= '0' after 10 ns;
            end if;
    end process;
    Output <= Result;

end Behavioral;
```

Παράρτημα Γ - Decoder

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

entity Decoder is
   Port ( Awr : in  STD_LOGIC_VECTOR (4 downto 0);
          Output : out  STD_LOGIC_VECTOR (31 downto 0));
end Decoder;

architecture Behavioral of Decoder is

signal dec_out: STD_LOGIC_VECTOR (31 downto 0);

begin
	dec_logic: process (Awr)
	begin
		dec_out <= (others => '0');
		for i in 0 to 31 loop
			if i = unsigned(Awr) then
				dec_out(i) <= '1';
			end if;
		end loop;
	end process;
	Output <= dec_out after 10 ns;

end Behavioral;
```

Παράρτημα Δ - Controller

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



entity CONTROL is
   Port ( Instr : in  STD_LOGIC_VECTOR (31 downto 0);
          PC_sel : out  STD_LOGIC;
          PC_LdEn : out  STD_LOGIC;
          RF_WrEn : out  STD_LOGIC;
          RF_WrData_sel : out  STD_LOGIC;
          RF_B_sel : out  STD_LOGIC;
          ImmExt : out  STD_LOGIC_VECTOR (1 downto 0);
          ALU_Bin_sel : out  STD_LOGIC;
          ALU_func : out  STD_LOGIC_VECTOR (3 downto 0);
          ALU_zero : in  STD_LOGIC;
          ByteOp : out  STD_LOGIC;
          Mem_WrEn : out  STD_LOGIC);
end CONTROL;

architecture Behavioral of CONTROL is

signal branch: STD_LOGIC;

begin

	is_branch: process (Instr)
	begin
		if Instr(31 downto 26) = "111111" then
			branch <= '1';
			PC_sel <= '1';
		elsif Instr(31 downto 26) = "000000" then
			branch <= '1';
			PC_sel <= '1' and ALU_zero;
		elsif Instr(31 downto 26) = "000001" then
			branch <= '1';
			PC_sel <= '1' and ALU_zero;
		else
			branch <= '0';
			PC_sel <= '0';
		end if;
	end process;

	mem_wr: process (Instr)
	begin
		if Instr(31 downto 26) = "000111" then
			Mem_WrEn <= '1';
		elsif Instr(31 downto 26) = "011111" then
			Mem_WrEn <= '1';
		else
			Mem_WrEn <= '0';
		end if;
	end process;
```

```
	reg_wr: process (Instr, branch)
	begin
		if (branch = '0') and (Instr(31) = '1') then
			RF_WrEn <= '1';
		elsif Instr(31 downto 26) = "000011" then
			RF_WrEn <= '1';
		elsif Instr(31 downto 26) = "001111" then
			RF_WrEn <= '1';
		else
			RF_WrEn <= '0';
		end if;
	end process;

	ext: process (Instr)
	begin
		if Instr(30 downto 27) = "1001" then
			ImmExt <= "00";
		elsif ((Instr(31) xor Instr(27)) and (Instr(31) xor Instr(26))) = '1' then
			ImmExt <= "01";
		elsif Instr(31 downto 26) = "111001" then
			ImmExt <= "11";
		else
			ImmExt <= "10";
		end if;
	end process;

	func: process (Instr, branch)
	begin
		if Instr(31 downto 26) = "100000" then
			ALU_func <= Instr(3 downto 0);			-- reg type
		elsif (Instr(31) nor branch) = '1' then
			ALU_func <= "0000";				-- memory
		elsif Instr(31 downto 29) = "111" and (branch = '0') then
			ALU_func <= "0000";				-- li, lui
		elsif branch = '1' then
			ALU_func <= "0001";				-- branch
		elsif Instr(31 downto 26) = "110010" then
			ALU_func <= "0101";				-- nandi
		else
			ALU_func <= Instr(29 downto 26);	-- addi, ori
		end if;
	end process;

	PC_LdEn <= '1';
	RF_WrData_sel <= not Instr(31);
	RF_B_sel <= not Instr(31);

	ALU_Bin_sel <= Instr(30) or Instr(27);
	ByteOp <= not (Instr(31) or Instr(30) or Instr(29));

end Behavioral;
```

Παράρτημα Ε

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



entity DATAPATH is
   Port ( Rst : in  STD_LOGIC;
          Clk : in  STD_LOGIC;
          PC_sel : in  STD_LOGIC;
          PC_LdEn : in  STD_LOGIC;
          PC : out  STD_LOGIC_VECTOR (31 downto 0);
          Instr : in  STD_LOGIC_VECTOR (31 downto 0);
          RF_WrEn : in  STD_LOGIC;
          RF_WrData_sel : in  STD_LOGIC;
          RF_B_sel : in  STD_LOGIC;
          ImmExt : in  STD_LOGIC_VECTOR (1 downto 0);
          ALU_Bin_sel : in  STD_LOGIC;
          ALU_func : in  STD_LOGIC_VECTOR (3 downto 0);
          ALU_zero : out  STD_LOGIC;
          ByteOp : in  STD_LOGIC;
          Mem_WrEn : in  STD_LOGIC;
          MM_WrEn : out  STD_LOGIC;
          MM_Addr : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_WrData : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_RdData : in  STD_LOGIC_VECTOR (31 downto 0));
end DATAPATH;

architecture Behavioral of DATAPATH is

signal Immed : STD_LOGIC_VECTOR (31 downto 0);
signal ALU_out : STD_LOGIC_VECTOR (31 downto 0);
signal MEM_out : STD_LOGIC_VECTOR (31 downto 0);
signal RF_A, RF_B : STD_LOGIC_VECTOR (31 downto 0);

component IFSTAGE is
   Port ( PC_Immed : in  STD_LOGIC_VECTOR (31 downto 0);
          PC_sel : in  STD_LOGIC;
          PC_LdEn : in  STD_LOGIC;
          Rst : in  STD_LOGIC;
          Clk : in  STD_LOGIC;
          PC : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component DECSTAGE is
   Port ( Instr : in  STD_LOGIC_VECTOR (31 downto 0);
          RF_WrEn : in  STD_LOGIC;
          ALU_out : in  STD_LOGIC_VECTOR (31 downto 0);
          MEM_out : in  STD_LOGIC_VECTOR (31 downto 0);
          RF_WrData_sel : in  STD_LOGIC;
          RF_B_sel : in  STD_LOGIC;
          ImmExt : in  STD_LOGIC_VECTOR (1 downto 0);
          Clk : in  STD_LOGIC;
          Rst : in  STD_LOGIC;
          Immed : out  STD_LOGIC_VECTOR (31 downto 0);
```

```
          RF_A : out  STD_LOGIC_VECTOR (31 downto 0);
          RF_B : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component EXSTAGE is
    Port ( RF_A : in  STD_LOGIC_VECTOR (31 downto 0);
          RF_B : in  STD_LOGIC_VECTOR (31 downto 0);
          Immed : in  STD_LOGIC_VECTOR (31 downto 0);
          ALU_Bin_sel : in  STD_LOGIC;
          ALU_func : in  STD_LOGIC_VECTOR (3 downto 0);
          ALU_out : out  STD_LOGIC_VECTOR (31 downto 0);
          ALU_zero : out  STD_LOGIC);
end component;

component MEMSTAGE is
    Port ( ByteOp : in  STD_LOGIC;
          Mem_WrEn : in  STD_LOGIC;
          ALU_MEM_Addr : in  STD_LOGIC_VECTOR (31 downto 0);
          MEM_DataIn : in  STD_LOGIC_VECTOR (31 downto 0);
          MEM_DataOut : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_WrEn : out  STD_LOGIC;
          MM_Addr : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_WrData : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_RdData : in  STD_LOGIC_VECTOR (31 downto 0));
end component;

begin

            IFSTAGE_module: IFSTAGE PORT MAP (
         PC_Immed => Immed,
         PC_sel => PC_sel,
         PC_LdEn => PC_LdEn,
         Rst => Rst,
         Clk => Clk,
         PC => PC
       );

   DECSTAGE_module: DECSTAGE PORT MAP (
         Instr => Instr,
         RF_WrEn => RF_WrEn,
         ALU_out => ALU_out,
         MEM_out => MEM_out,
         RF_WrData_sel => RF_WrData_sel,
         RF_B_sel => RF_B_sel,
         ImmExt => ImmExt,
         Clk => Clk,
         Rst => Rst,
         Immed => Immed,
         RF_A => RF_A,
         RF_B => RF_B
       );

   EXSTAGE_module: EXSTAGE PORT MAP (
         RF_A => RF_A,
         RF_B => RF_B,
         Immed => Immed,
         ALU_Bin_sel => ALU_Bin_sel,
```

```
        ALU_func => ALU_func,
        ALU_out => ALU_out,
        ALU_zero => ALU_zero
      );


          -- Instantiate the Unit Under Test (UUT)
  MEMSTAGE_module: MEMSTAGE PORT MAP (
        ByteOp => ByteOp,
        Mem_WrEn => Mem_WrEn,
        ALU_MEM_Addr => ALU_out,
        MEM_DataIn => RF_B,
        MEM_DataOut => MEM_out,
        MM_WrEn => MM_WrEn,
        MM_Addr => MM_Addr,
        MM_WrData => MM_WrData,
        MM_RdData => MM_RdData
      );




end Behavioral;
```

Παράρτημα Ζ

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity MEM is
   Port ( clk : in  STD_LOGIC;
          inst_addr : in  STD_LOGIC_VECTOR (10 downto 0);
          inst_dout : out  STD_LOGIC_VECTOR (31 downto 0);
          data_we : in  STD_LOGIC;
          data_addr : in  STD_LOGIC_VECTOR (10 downto 0);
          data_din : in  STD_LOGIC_VECTOR (31 downto 0);
          data_dout : out  STD_LOGIC_VECTOR (31 downto 0));
end MEM;

architecture syn of MEM is
   type ram_type is array (2047 downto 0) of std_logic_vector (31 downto 0);
   impure function InitRamFromFile (RamFileName : in string) return ram_type is
   FILE ramfile : text is in RamFileName;
   variable RamFileLine : line;
   variable ram : ram_type;
   begin
      for i in 0 to 1023 loop
         readline(ramfile, RamFileLine);
         read (RamFileLine, ram(i));
      end loop;
      for i in 1024 to 2047 loop
         ram(i) := x"00000000";
      end loop;
   return ram;
   end function;

   signal MEM:ram_type := InitRamFromFile("rom.data");

begin
   process (clk)
   begin
      if clk'event and clk = '1' then
         if data_we = '1' then
            MEM(conv_integer(data_addr)) <= data_din;
         end if;
      end if;
   end process;

   data_dout <= MEM(conv_integer(data_addr)) after 12ns;
   inst_dout <= MEM(conv_integer(inst_addr)) after 12ns;

end syn;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity MEMSTAGE is
```

```
   Port ( ByteOp : in  STD_LOGIC;
          Mem_WrEn : in  STD_LOGIC;
          ALU_MEM_Addr : in  STD_LOGIC_VECTOR (31 downto 0);
          MEM_DataIn : in  STD_LOGIC_VECTOR (31 downto 0);
          MEM_DataOut : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_WrEn : out  STD_LOGIC;
          MM_Addr : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_WrData : out  STD_LOGIC_VECTOR (31 downto 0);
          MM_RdData : in  STD_LOGIC_VECTOR (31 downto 0));
end MEMSTAGE;

architecture Behavioral of MEMSTAGE is

component Adder is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          Cin : in  STD_LOGIC;
          Cout : out  STD_LOGIC;
          Ovf : out  STD_LOGIC;
          S : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

begin

  MM_WrEn <= Mem_WrEn;

  -- pros8etw to 1024 * 4 (0x400<<2)
  Add_offset: Adder PORT MAP (
       A => ALU_MEM_Addr,
       B => X"00001000",
       Cin => '0',
       S => MM_Addr
      );

  Mux_MM_WrData: process (MEM_DataIn, ByteOp)
  begin
          if ByteOp = '0' then
                  MM_WrData <= MEM_DataIn;
          else
                  MM_WrData <= X"000000" & MEM_DataIn(7 downto 0);
          end if;
  end process;

  Mux_MEM_DataOut: process (MM_RdData, ByteOp)
  begin
          if ByteOp = '0' then
                  MEM_DataOut <= MM_RdData;
          else
                  MEM_DataOut <= X"000000" & MM_RdData(7 downto 0);
          end if;
  end process;

 end Behavioral;
```

Παράρτημα Η

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


entity REG is
   Port ( RST : in  STD_LOGIC;
          CLK : in  STD_LOGIC;
          WE : in  STD_LOGIC;
          Datain : in  STD_LOGIC_VECTOR (31 downto 0);
          Dataout : out  STD_LOGIC_VECTOR (31 downto 0));
end REG;

architecture Behavioral of REG is

signal reg_out: STD_LOGIC_VECTOR (31 downto 0);

begin

        reg_logic: process (CLK)
        begin
                if CLK'event and CLK = '1' then
                        if RST = '1' then
                                reg_out <= (others => '0');
                        elsif WE = '1' then
                                reg_out <= Datain;
                        end if;
                end if;
        end process;

        Dataout <= reg_out after 10 ns;


end Behavioral;
```

Παράρτημα Θ

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


use IEEE.NUMERIC_STD.ALL;



entity RF is
   Port ( Ard1 : in  STD_LOGIC_VECTOR (4 downto 0);
          Ard2 : in  STD_LOGIC_VECTOR (4 downto 0);
          Awr : in  STD_LOGIC_VECTOR (4 downto 0);
          Dout1 : out  STD_LOGIC_VECTOR (31 downto 0);
          Dout2 : out  STD_LOGIC_VECTOR (31 downto 0);
          Din : in  STD_LOGIC_VECTOR (31 downto 0);
          WrEn : in  STD_LOGIC;
          Clk : in  STD_LOGIC;
          Rst : in  STD_LOGIC);
end RF;

architecture Behavioral of RF is

component Decoder is
   Port ( Awr : in  STD_LOGIC_VECTOR (4 downto 0);
          Output : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component REG is
   Port ( RST : in  STD_LOGIC;
          CLK : in  STD_LOGIC;
          WE : in  STD_LOGIC;
          Datain : in  STD_LOGIC_VECTOR (31 downto 0);
          Dataout : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

type array3232 is array (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
signal reg_array: array3232;
signal dec_out, and_out, WrEn_vector, Rst_vector: STD_LOGIC_VECTOR (31 downto 0);

begin

  Decoder_module: Decoder PORT MAP (
        Awr => Awr,
        Output => dec_out
      );

  WrEn_vector <= (others => WrEn);
  and_out <= dec_out and WrEn_vector after 2 ns;

  rst_v: process (Rst)
  begin
           Rst_vector <= (others => Rst);
           Rst_vector(0) <= '1';
  end process;
```

```
  Registers: for i in 0 to 31 generate
        r: REG PORT MAP (
                    RST => Rst_vector(i),
                    CLK => Clk,
                    WE => and_out(i),
                    Datain => Din,
                    Dataout => reg_array(i)
            );
  end generate;

  Mux_dout1: block
  begin
        Dout1 <= reg_array(to_integer(unsigned(Ard1))) after 10 ns;
  end block;

  Mux_dout2: block
  begin
        Dout2 <= reg_array(to_integer(unsigned(Ard2))) after 10 ns;
  end block;

end Behavioral;
```

Παράρτημα Ι

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


entity EXSTAGE is
    Port ( RF_A : in  STD_LOGIC_VECTOR (31 downto 0);
           RF_B : in  STD_LOGIC_VECTOR (31 downto 0);
           Immed : in  STD_LOGIC_VECTOR (31 downto 0);
           ALU_Bin_sel : in  STD_LOGIC;
           ALU_func : in  STD_LOGIC_VECTOR (3 downto 0);
           ALU_out : out  STD_LOGIC_VECTOR (31 downto 0);
           ALU_zero : out  STD_LOGIC);
end EXSTAGE;

architecture Behavioral of EXSTAGE is

component ALU is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
           B : in  STD_LOGIC_VECTOR (31 downto 0);
           Op : in  STD_LOGIC_VECTOR (3 downto 0);
           Output : out  STD_LOGIC_VECTOR (31 downto 0);
           Zero : out  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Ovf : out  STD_LOGIC);
end component;

signal B: STD_LOGIC_VECTOR (31 downto 0);

begin

  ALU_modle: ALU PORT MAP (
        A => RF_A,
        B => B,
        Op => ALU_func,
        Output => ALU_out,
        Zero => ALU_zero
      );

  Mux_AluB: process (RF_B, Immed, ALU_Bin_sel)
  begin
            if ALU_Bin_sel = '0' then
                    B <= RF_B;
            else
                    B <= Immed;
            end if;
  end process;

end Behavioral;


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity IFSTAGE is
    Port ( PC_Immed : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
        PC_sel : in  STD_LOGIC;
        PC_LdEn : in  STD_LOGIC;
        Rst : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        PC : out  STD_LOGIC_VECTOR (31 downto 0));
end IFSTAGE;

architecture Behavioral of IFSTAGE is

component REG is
   Port ( RST : in  STD_LOGIC;
        CLK : in  STD_LOGIC;
        WE : in  STD_LOGIC;
        Datain : in  STD_LOGIC_VECTOR (31 downto 0);
        Dataout : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component Adder is
   Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
        B : in  STD_LOGIC_VECTOR (31 downto 0);
        Cin : in  STD_LOGIC;
        Cout : out  STD_LOGIC;
        Ovf : out  STD_LOGIC;
        S : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

signal PC_out, PC_in, PC_p4, PC_p4i: STD_LOGIC_VECTOR (31 downto 0);
begin

  PC <= PC_out;

  Add_4: Adder PORT MAP (
       A => PC_out,
       B => X"00000004",
       Cin => '0',
       S => PC_p4
     );

  Add_Immed: Adder PORT MAP (
       A => PC_p4,
       B => PC_Immed,
       Cin => '0',
       S => PC_p4i
     );

  Mux: process (PC_p4, PC_p4i, PC_sel)
  begin
          if PC_sel = '0' then
                  PC_in <= PC_p4;
          else
                  PC_in <= PC_p4i;
          end if;
  end process;

  PC_module: REG PORT MAP (
       RST => Rst,
       CLK => Clk,
```

```
    WE => PC_LdEn,
    Datain => PC_in,
    Dataout => PC_out
   );


end Behavioral;
```

Παράρτημα Κ

```
module mojo_top(
   // 50MHz clock input
   input clk,
   // Input from reset button (active low)
   input rst_n,
   // cclk input from AVR, high when AVR is ready
   input cclk,
   // Outputs to the 8 onboard LEDs
   output[7:0]led,
   // AVR SPI connections
   output spi_miso,
   input spi_ss,
   input spi_mosi,
   input spi_sck,
   // AVR ADC channel select
   output [3:0] spi_channel,
   // Serial connections
   input avr_tx, // AVR Tx => FPGA Rx
   output avr_rx, // AVR Rx => FPGA Tx
   input avr_rx_busy // AVR Rx buffer full
   );

wire rst = ~rst_n; // make reset active high

// these signals should be high-z when not used
assign spi_miso = 1'bz;
assign avr_rx = 1'bz;
assign spi_channel = 4'bzzzz;

reg cin;  //είσοδος για carry
reg[31:0] a; //είσοδος για 32bit λέξη a
reg[31:0] b; //είσοδος για 32bit λέξη b
// οι πολλαπλασιασμοί 16bit δίνουν max αριθμό 32 bits
// επομένως επειδή η ALU διαχειρίζεται έως 32 bits,
// κάνουμε τη παραδοχή για εισόδους των 16 bits
reg[15:0] mula; //είσοδος για 16bit λέξη a
reg[15:0] mulb; //είσοδος για 16bit λέξη b

reg[3:0] opcode; //δίνει την εντολή που εκτελείτε
// επίσης το MSB δείχνει αν θα διαβαστεί από τον register της ALU (MSB = 1)
// ή αν απλά θα εκτελεστεί πράξη χωρίς να έχουμε αποτέλεσμα στη μνήμη (MSB = 0)
// τα υπόλοιπα λιγότερο σημαντικά bits δείχνουν την επιλεγμένη εντολή όπως ορίζεται στο
//mux4
wire[31:0] out32; //η 32bit έξοδος
wire[7:0] out8; //η 8 bit έξοδος που πάει στο led
wire cout; //carry εξόδου αν υπάρχει




initial begin
```

```
 a = 32'b10000000000000000000000000001111;     //παράδειγμα a
 b = 32'b00000000000000000000000000001111;     //παράδειγμα b
 mula = 16'b0000000000001111;       //παράδειγμα a για πολλαπλασιασμό
 mulb = 16'b0000000000001111; //παράδειγμα b για πολλαπλασιασμό
 cin = 1'b0; //παράδειγμα για carry εισόδου
 opcode = 4'b1101; //εδώ βλέπουμε την εντολή πράξης AandB με καταχώρηση στον register
end

ALU alu_F(opcode, a, b, mula, mulb, cin, cout, out32, out8); //αρχικοποίηση της ALU με τα σήματα

assign led = out8; //τοποθέτηση 8 bit αποτελέσματος στο led
```

Παράρτημα Λ

```
module ALU(
    input[3:0] Op,
    input[31:0] A,
    input[31:0] B,
    input[15:0] mulA,
    input[15:0] mulB,
    input Cin,
    output Cout,
    output[31:0] Out32,
    output[7:0] Out8
);

wire[31:0] w1,w2,w3,w4,w5,w6,w8,w10;
wire w7,w9;
//εδώ φτιάχνονται σήματα με καλώδια (wire) γιατί θα μπουν όλα με μορφή block σχήματος και
//όχι με χρήση κώδικα σε always ή assigns.δηλαδή εδώ πλέον υλοποιείτε αρχιτεκτονική όχι
//απλά λογική λειτουργείας όπως στα υποblocks
And32 f1(A,B,w1);
Or32 f2(A,B,w2);
Xor32 f3(A,B,w3);
Shift32R f4(A,w4);
Shift32L f5(A,w5);
Adder32 f6(A,B,Cin,w6,w7);
Substr32 f7(A,B,w8,w9);
Mult16 f8(mulA,mulB,w10);
Mux4 f(Op,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,Out32,Cout);
map32to8 g(Out32,Out8);

endmodule
```

Παράρτημα Μ

```
module Mux4(
  input[3:0] Op,
  input[31:0] And32,
  input[31:0] Or32,
  input[31:0] Xor32,
  input[31:0] Shift32R,          //τα μικρά σύμβολα κρατάνε τα αποτελέσματα
  input[31:0] Shift32L,          // που πράγματι εκτελούνται
  input[31:0] Adder32,           // τα μεγάλα σύμβολα κρατάνε αποτελέσματα αν τυχόν εγγράφουν
  input adder32Carry,            // και διαβαστούν στη μνήμη του register.
  input[31:0] Sub32,             // Άρα για να γίνει αυτό το opcode πρέπει να φέρει 1 στο MSB.
  input sub32Carry,
  input[31:0] Mult32,
  output[31:0] out,
  output carryOut
);
reg[31:0] w,W;
reg c,C;

always @(*) begin
 C = 1'b0;                  //αρχικοποιούνται τα σήματα πάντα σε κάθε πράξη
 W = 32'b0;
 case(Op[2:0])              //εδώ διακρίνουμε τις περιπτώσεις των πράξεων που θα εκτελεστούν
  3'b000 :  w = And32;      // βάση των 3 τελευταίων bits της εντολής
  3'b001 :  w = Or32;
  3'b010 :  w = Xor32;
  3'b011 :  w = Shift32R;
  3'b100 :  w = Shift32L;
  3'b101 :  begin
              w = Adder32;
              c = adder32Carry;
            end
  3'b110 :  begin
              w = Sub32;
              c = sub32Carry;
            end
  3'b111 :  w = Mult32;
 endcase

// εδώ ελέγχουμε το MSB της εντολής για να δούμε αν θα κάνει εγγραφή και διάβασμα του
//register. Αν γίνει, τα leds θα ανάψουν ανάλογα με τη πράξη, διαφορετικά δεν θα ανάψουν
 if(Op[3] == 1'b1) begin
   W = w;
   C = c;
 end
end

assign out = W;
assign carryOut = C;

endmodule
```

Παράρτημα Ν

```
module Adder32(
  input[31:0] A,
  input[31:0] B,
  input Cin,
  output[31:0] S,
  output Cout
);

integer i;
reg[31:0] w; //Χρήση ενδιάμεσου σήματος για αναδρομική καταχώρηση αποτελέσματος. Αυτό
//απαιτητέ λόγο της υλοποίησης με loop και θέλουμε τέτοιου τύπου δεδομένα στο always block
//(τύπου reg). Στο τέλος αποθηκεύεται με το assign.
reg[32:0] carry;

// εκτελείτε αναδρομικά η πράξη της πρόσθεσης βασισμένη στην λογική του full adder.
//Βασισμένη στην ισοδύναμη εξίσωση bool για full adder για το πως υπολογίζει sum και carry.
always @(*) begin
 carry[0] = Cin;
 for(i=0;i<32;i=i+1) begin
   w[i] = A[i]^B[i]^carry[i];
   carry[i+1] = (A[i]&B[i]) | (B[i]&carry[i]) | (A[i]&carry[i]);
 end
end

assign S = w;
assign Cout = carry[32];          //στο τέλος το carry στο 33 bit δείχνει αν
endmodule                         //υπερχείλισε το αποτέλεσμα ή όχι



module Substr32(
  input[31:0] A,
  input[31:0] B,
  output[31:0] S,
  output Cout
);

integer i;
reg[31:0] xorw;
reg[31:0] w;
reg[32:0] carry;

// η λογική της αφαίρεσης είναι όταν έχουμε πρόσθεση με το συμπλήρωμα
// οπότε δημιουργούμε τα συμπληρώματα βάση του carry εισόδου (1) και μετά εκτελούμε τις
//πράξεις πρόσθεσης όπως στον αθροιστή. Κανονικά ο αθροιστής γίνεται και αφαιρέτης σε άλλα
//κυκλώματα, αλλά εδώ κάνουμε χωριστές τις λειτουργίες για να φαίνεται η διάκριση
```

```
always @(*) begin
 carry[0] = 1'b1;
 for(i=0;i<32;i=i+1) begin
   xorw[i] = B[i]^carry[0];
   w[i] = A[i]^xorw[i]^carry[i];
   carry[i+1] = (A[i]&xorw[i]) | (xorw[i]&carry[i]) | (A[i]&carry[i]);
 end
end

assign S = w;
assign Cout = carry[32];


endmodule


module Mult16(
  input[15:0] A,
  input[15:0] B,
  output[31:0] S
);

integer i;
reg[31:0] w;
//εκτελούμε τον πολλαπλασιασμό με βάση το shift επειδή έτσι δίνεται στην αρχιτεκτονική του ο
//πολλαπλασιαστής. Έχει την λογική να κάνεις τόσα shifts στον πολλαπλασιαστέο όσα ορίζει το
//κάθε ψηφίο του πολλαπλασιαστή.
always @(*) begin
  w = 32'b0;
  for (i = 0; i < 16; i = i + 1) begin
    if (B[i]) begin
      w = w + (A << i);
    end
  end
end

assign S = w;

endmodule
```

Παράρτημα Ξ

```
module And32(
  input[31:0] A,
  input[31:0] B,
  output[31:0] S
);

assign S = A&B; //εκτελούμε απλά την and πράξη

endmodule


module Or32(
  input[31:0] A,
  input[31:0] B,
  output[31:0] S
);

assign S = A | B; // απλή λογική or

endmodule


module Shift32L(
  input[31:0] A,
  output[31:0] S
);

integer i;
reg[31:0] x;

assign S[0] = A[31];
//γίνεται αναδρομική εκχώρηση κυκλικά προς τα αριστερά. Άρα κοιτάμε ποιο bit είχαμε στο
//τέλος για να μπει στην αρχή
always @(*) begin
for(i=1;i<32;i=i+1) begin
   x[i] = A[i-1];
end
end

assign S[31:1] = x[31:1];

endmodule



module Shift32R(
  input[31:0] A,
  output[31:0] S
);

integer i;
```

```
reg[30:0] x;
// γίνεται αναδρομική εκχώρηση κυκλικά προς τα δεξιά. Άρα κοιτάμε ποιο bit είχαμε στην αρχή
//για να μπει στο τέλος

always @(*) begin
for(i=30;i>=0;i=i-1) begin
  x[i] = A[i+1];
end
end

assign S[30:0] = x;
assign S[31] = A[0];

endmodule



module Xor32(
  input[31:0] A,
  input[31:0] B,
  output[31:0] S
);

assign S = A^B;    // απλό XOR

endmodule



module map32to8(
    input [31:0] data_in,
    output [7:0] data_out
);

assign data_out = data_in [7:0];     //εδώ κρατάμε τα λιγότερο  σημαντικά 8 bits

endmodule
```