ΚΕΦΑΛΑΙΟ 1

ΕΙΣΑΓΩΓΗ ΣΤΙΣ ΠΡΟΓΡΑΜΜΑΤΙΖΟΜΜΕΝΕΣ ΔΙΑΤΑΞΕΙΣ ΠΥΛΩΝ

(FPGAS)

1.1 Εισαγωγή

Οι προγραμματιζόμενες λογικές συσκευές (PLDs) είναι ψηφιακά ολοκληρωμένα κυκλώματα (ICs), διαμορφούμενα από τον χρήστη που χρησιμοποιούνται για την υλοποίηση λογικών συναρτήσεων. Τα PLDs μπορούν να πραγματοποιήσουν οποιαδήποτε Boolean έκφραση ή συνάρτηση χρησιμοποιώντας τις λογικές δομές που είναι ήδη υλοποιημένες. Σε αντίθεση, τα συνήθη ολοκληρωμένα κυκλώματα, όπως για παράδειγμα οι TTL συσκευές, παρέχουν μία συγκεκριμένη λογική συνάρτηση που δεν μπορεί να τροποποιηθεί έτσι ώστε να ικανοποιήσει συγκεκριμένη απαίτηση σε κάποιο σχεδιασμό. Τα τελευταία χρόνια, τα PLDs αποτελούν την περισσότερο προτιμούμενη επιλογή, καθώς το κόστος τους έχει μειωθεί σημαντικά με την χρήση νέων τεχνολογιών. Οι κατασκευαστές των PLDs έχουν την δυνατότητα να προσφέρουν τις συσκευές τους που έχουν υψηλότερα επίπεδα ολοκλήρωσης, καλύτερη απόδοση λειτουργίας και χαμηλότερο κόστος από τις συνήθεις διακριτές συσκευές.

Η προγραμματιζόμενη λογική συμπεριλαμβάνει όλα τα λογικά κυκλώματα που διαμορφώνονται από τον χρήστη, περιέχοντας απλές 20-pin PAL συσκευές, Field Programmable Gate Arrays (FPGAs), και σύνθετα PLDs (CPLDs). Τα PLDs προσφέρονται σε διάφορες αρχιτεκτονικές και με ένα μεγάλο πλήθος από τεχνολογίες μνημών για τον προγραμματισμό τους.

Τα CPLDs και τα FPGAs έχουν διαφορετική δομή διασυνδέσεων. Η δομή των διασυνδέσεων (διασυνδέσεις χωρισμένες σε τμήματα) των FPGAs χρησιμοποιεί μεταβλητά μήκη μεταλλικών γραμμών που συνδέονται με τα λογικά κελιά (cells) μέσο pass transistors. Σε αντίθεση, η συνεχής δομή των διασυνδέσεων των CPLDs χρησιμοποιεί μεταλλικές γραμμές που είναι όλες του ίδιου μήκους για να παρέχουν σύνδεση μεταξύ λογικής και κελιών. Η συνεχής δομή διασύνδεσης αποκλείει την μεταβλητότητα των χρονισμών που σχετίζεται με την τμηματική δομή των διασυνδέσεων, και έτσι παρέχει γρήγορα και καθορισμένου μήκους μονοπάτια καθυστερήσεων ανάμεσα στα λογικά κελιά. Η δομή αυτή κάνει πιο εύκολη την υλοποίηση ενός κυκλώματος και ταυτόχρονα μειώνει τον κύκλο ανάπτυξης.

Οι σχεδιαστές συνήθως αναπτύσσουν ένα λογικό κύκλωμα με μία από τις τρεις διαφορετικές επιλογές υλοποιήσεων που είναι: διακριτή λογική (TTL, CMOS), custom ή semi-custom συσκευές (ASICs) και PLDs. Η καλύτερη επιλογή είναι αυτή που εκπληρώνει τον μεγαλύτερο αριθμό των απαιτήσεων ενός σχεδιασμού. Ο πίνακας 1 δείχνει κάποιες από τις σχεδιαστικές απαιτήσεις καθώς και τον βαθμό με τον οποίο αυτές επιτυγχάνονται στις τρεις διαφορετικές επιλογές σχεδιάσεων.

Απαίτηση	PLD	Διακριτή Λογική	Custom συσκευές
Ταχύτητα	***	*	***
Πυκνότητα	***	*	***
Κόστος	***	*	***(1)
Χρόνος Ανάπτυξης	***	**	*
Χρόνος Εξομοίωσης	***	*	*
Χρόνος Κατασκευής	***	**	*
Ευκολία Χρήσης	***	**	*
Μελλοντική Τροποποίηση	***	**	*
Εργαλεία Ανάπτυξης	***	*	***

Πίνακας 1

Σημειώσεις : *(1)* Αποτελεσματικά μόνο σε μεγάλη κλίμακα παραγωγής.
*** Πολύ αποτελεσματικά
** Ικανοποιητικά
* Όχι ικανοποιητικά

1.2 Πλεονεκτήματα των PLDs της Altera

Τα PLDs της Altera δεν προσφέρουν μόνο τα γενικά πλεονεκτήματα της τεχνολογίας των PLDs, αλλά και άλλα πλεονεκτήματα που στηρίζονται στην αρχιτεκτονική τους καθώς και στο σχεδιαστικό περιβάλλον ανάπτυξης του MAX+PLUS II. Τα πλεονεκτήματα αυτά περιλαμβάνουν:

a) Καλύτερη Απόδοση
b) Μεγαλύτερη Κλίμακα Ολοκλήρωσης
c) Καλύτερο λόγο Κόστους/Αποτελεσματικότητα
d) Μικρότερος κύκλος σχεδίασης χρησιμοποιώντας το MAX+PLUS II λογισμικό.

1.2.1Καλύτερη Απόδοση

Η απόδοση είναι συνάρτηση της αρχιτεκτονικής. Οι συσκευές της Altera κατασκευάζονται με CMOS τεχνολογία, που προσφέρουν τις χαμηλότερες δυνατές καθυστερήσεις. Ακόμα, η συνεχής δομή διασυνδέσεων παρέχει γρήγορα με σταθερές καθυστερήσεις σήματα σε όλο το μήκος της συσκευής.

1.2.2 Μεγαλύτερη Κλίμακα Ολοκλήρωσης

Οι σχεδιαστές συχνά αναζητούν την υψηλότερη δυνατή ολοκλήρωση για τα σχέδια τους, έτσι ώστε να μειώσουν το κόστος και το μέγεθος του τυπωμένου κυκλώματος. Στην περίπτωση αυτή, τα PLDs με υψηλή κλίμακα ολοκλήρωσης προσφέρουν την καλύτερη λύση. Οι συσκευές της Altera, με πυκνότητα από 300 έως 50.000 χρησιμοποιήσιμες πύλες, μπορούν εύκολα να ολοκληρώσουν οποιαδήποτε υπάρχουσα λογική. Η δυνατότητα αυτή της μεγάλης κλίμακας ολοκλήρωσης παρέχει καλύτερη απόδοση και αξιοπιστία καθώς και χαμηλότερο κόστος.

1.2.3 Καλύτερος λόγος Κόστους/Αποτελεσματικότητα

Με την βελτίωση των τεχνικών παραγωγής και ανάπτυξης αυξάνεται η απόδοση των προγραμματιζόμενων λογικών συσκευών και αναβαθμίζεται ο λόγος Κόστους/Αποτελεσματικότητα.

1.2.4 Μικρότερος κύκλος σχεδίασης

Κατά την διάρκεια της σχεδίασης ενός κυκλώματος, ο σημαντικότερος παράγοντας είναι ο χρόνος που απαιτείται για την ολοκλήρωσή της. Επομένως, όσο μικρότερος είναι ο πλήρης κύκλος μιας σχεδίασης, τόσο το καλύτερο. Με το λογισμικό MAX+PLUS II της Altera, ο κύκλος σχεδίασης μειώνεται σημαντικά. Η εισαγωγή ενός σχεδίου, η πιστοποίηση της ορθής λειτουργίας του και ο προγραμματισμός της συσκευής ολοκληρώνονται μέσα σε λίγες ώρες επιτρέποντας έτσι πολλαπλές ανασχεδιάσεις μέσα σε μία μέρα.

1.3 Οικογένειες συσκευών της Altera

Η Altera προσφέρει επτά οικογένειες γενικής χρήσης PLDs που είναι, όπως δείχνεται στον πίνακα 2, οι: FLEX 10K, FLEX 8000, MAX 9000, MAX 7000, FLASHlogic, MAX 5000, Classic.

Λογική Οικογένεια	Δομή του Λογικού Κελιού	Δομή Διασυνδέσεων	Επαναπρο/μενο Στοιχείο
FLEX 10K	Look- up-Table	Συνεχής	SRAM
FLEX 8000	Look- up-Table	Συνεχής	SRAM
FLEX 9000	Product Term	Συνεχής	EEPROM
FLEX 7000	Product Term	Συνεχής	EEPROM
FLASHlogic	Product Term	Συνεχής	SRAM & Flash
MAX 500	Product Term	Συνεχής	EPROM
Classic	Product Term	Συνεχής	EPROM

Πίνακας 2
Λογικές οικογένειες συσκευών της Altera.

Όλες οι λογικές οικογένειες κατασκευάζονται με χρήση CMOS τεχνολογίας, που παρέχει χαμηλότερη κατανάλωση και μεγαλύτερη αξιοπιστία από τις διπολικές τεχνολογίες.

1.4 Το σχεδιαστικό πακέτο MAX + PLUSII

Στην ιδανική κατάσταση, ένα περιβάλλον σχεδίασης προγραμματιζόμενης λογικής ικανοποιεί μία πληθώρα από σχεδιαστικές απαιτήσεις όπως: θα πρέπει να υποστηρίζει συσκευές με διαφορετικές αρχιτεκτονικές , να τρέχει σε διάφορες πλατφόρμες, να είναι απλό στις διαδικασίες που επιτελεί και να επικοινωνεί με τον χρήστη με εύκολο τρόπο. Το **MAX + PLUSII** της Altera, αποτελεί ένα ολοκληρωμένο περιβάλλον σχεδίασης που εκπληρώνει όλες τις παραπάνω απαιτήσεις.

Το MAX + PLUS II έχει τα ακόλουθα χαρακτηριστικά :

- *Ανεξαρτησία από την Αρχιτεκτονική :* Το MAX PLUS II υποστηρίζει τις οικογένειες FLEX 10K, FLEX 8000, FLEX 6000, MAX 9000, MAX 7000, MAX 5000 και Classic της Altera. Ο συμβολομεταφραστής του MAX PLUS II παρέχει ακόμα, λογική σύνθεση και βελτιστοποίηση δημιουργώντας έτσι αρκετά αποδοτικά κυκλώματα.
- *Πολλαπλές πλατφόρμες Εργασίας :* Το MAX PLUS II “ τρέχει “ σε Windows NT 3.51 ή 4.0, Windows 95 σε 486 - ή Pentium PCs και σε Sun SPARCstationw, HP9000 Series 700/800, IBM RISC System/6000 σταθμούς εργασίας.
- *Πλήρως Ολοκληρωμένο :* Ο τρόπος με τον οποίο γίνεται η εισαγωγή ενός κυκλώματος καθώς και όλη η διαδικασία σχεδιασμού επιτρέπουν την γρήγορη εκσφαλμάτωση ενός σχεδιασμού με αποτέλεσμα να απαιτούνται λιγότεροι κύκλοι ανάπτυξης.
- *Γλώσσες Περιγραφής Υλικού :* Το MAX PLUS II υποστηρίζει αρκετές γλώσσες περιγραφής υλικού όπως είναι η VHDL, VerilogHDL καθώς και την γλώσσα περιγραφής υλικού της Altera (AHDL).
- *MegaCore συναρτήσεις* : Οι MegaCore συναρτήσεις είναι HDL αρχεία σε netlist μορφή που υλοποιούν σύνθετες συναρτήσεις και έχουν βελτιστοποιηθεί για τις οικογένειες FLEX10K, FLEX 8000, FLEX 6000, MAX 9000 και MAX 7000.

Το MAX PLUS II προσφέρει ένα μεγάλο φάσμα από σχεδιαστικές λογικές δυνατότητες. Ο σχεδιαστής μπορεί να συνδυάζει κώδικα, σχηματικά διαγράμματα, καθώς και κυματομορφές που παράγονται με την εφαρμογή του εξομοιωτή σε συγκεκριμένο κύκλωμα, δημιουργώντας έτσι ένα πλήρως ιεραρχημένο σύστημα. Ο συμβολομεταφραστής πραγματοποιεί την λογική σύνθεση και βελτιστοποίηση, τοποθετεί το κύκλωμα σε μία ή περισσότερες συσκευές και παράγει τα απαιτούμενα δεδομένα για τον προγραμματισμό. Προσφέρεται ακόμα η δυνατότητα της επιβεβαίωσης του σχεδιασμού χρησιμοποιώντας λογική και χρονική εξομοίωση καθώς και πρόβλεψη των καθυστερήσεων για τα κρίσιμα ως προς τους χρονισμούς μονοπάτια

ΚΕΦΑΛΑΙΟ 2

Η ΠΡΟΓΡΑΜΜΑΤΙΖΟΜΕΝΗ ΛΟΓΙΚΗ ΟΙΚΟΓΕΝΕΙΑ FLEX 8000

2.1 Εισαγωγή

Τα γενικά χαρακτηριστικά της οικογένειας FLEX 8000 της Altera δίνονται στην συνέχεια:

a) Χαμηλό κόστος, υψηλή πυκνότητα, CMOS προγραμματιζόμενη λογική οικογένεια με 2500 έως 16000 λογικές πύλες και 282 έως 1500 καταχωρητές.
b) Χαμηλή κατανάλωση καθώς και ενσωματωμένα κυκλώματα JTAG BST (Boundary Scan Test) σύμφωνα με το πρότυπο IEEE Std 1149.1- 1990.
c) Έχει συνεχή δομή διασυνδέσεων έτσι ώστε να επιτυγχάνονται μικρές και προβλέψιμες καθυστερήσεις. Ακόμα έχει αφιερωμένα carry chains που χρησιμοποιούνται για την υλοποίηση γρήγορων μετρητών, συγκριτών και αθροιστών.
d) Δυναμικούς I/O ακροδέκτες
e) Υποστηρίζεται από το σχεδιαστικό πακέτο MAX PLUS II που παρέχει αυτόματη τοποθέτηση (*placement*) των εξαρτημάτων και διασύνδεση (*routing*) μεταξύ τους.

2.2 Γενική Περιγραφή

Η λογική οικογένεια FLEX (Flexible Logic Element Matrix) της Altera συνδυάζει τα πλεονεκτήματα τόσο των προγραμματιζόμενων λογικών συσκευών με δυνατότητα σβησίματος (EPLDs), όσο και των FPGAs. Πιο συγκεκριμένα, η οικογένεια FLEX 8000 είναι ιδανική για ένα μεγάλο πλήθος εφαρμογών, αφού συνδυάζει την ευέλικτη αρχιτεκτονική των FPGAs με την υψηλή ταχύτητα, και τις προβλέψιμες καθυστερήσεις στις διασυνδέσεις των EPLDs. Η λογική υλοποιείται με σταθερά look-up-tables (LUTs) των 4 εισόδων και προγραμματιζόμενους καταχωρητές. Η υψηλή τους απόδοση εξασφαλίζεται από την συνεχή δομή που ακολουθείται στις διασυνδέσεις.

Οι FLEX 8000 συσκευές παρέχουν ένα μεγάλο αριθμό από στοιχεία αποθήκευσης για εφαρμογές όπως ψηφιακή επεξεργασία σήματος (DSP) και μετασχηματισμό δεδομένων. Η λογική και οι διασυνδέσεις στην FLEX 8000 αρχιτεκτονική διαμορφώνονται με CMOS SRAM στοιχεία.

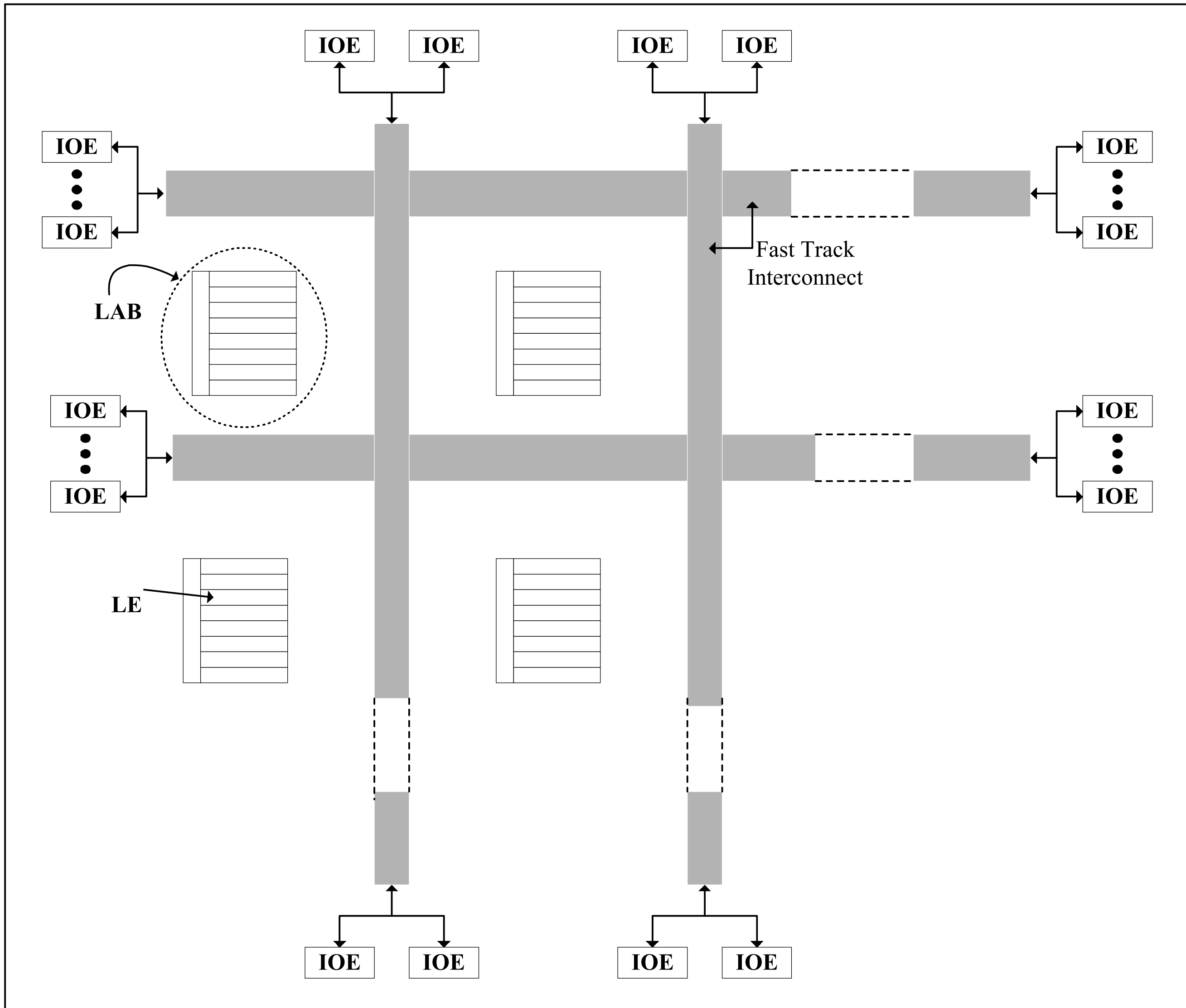
Η οικογένεια FLEX 8000 υποστηρίζεται από το σχεδιαστικό εργαλείο ανάπτυξης της Altera, MAX + PLUSII που προσφέρει την δυνατότητα σχεδίασης με χρήση σχηματικών διαγραμμάτων και με Γλώσσα Περιγραφής Υλικού (AHDL, VHDL, VerilogVHDL), καθώς και την εξομοίωση των κυκλωμάτων που έχουν προκύψει μετά από συμβολομετάφραση και λογική σύνθεση.

2.3 Λειτουργική Περιγραφή

Η αρχιτεκτονική της οικογένειας FLEX 8000 έχει ενσωματωμένο ένα μεγάλο πίνακα από συμπαγή blocks, που αποτελούν τα λογικά στοιχεία (**LEs**). Κάθε τέτοιο στοιχείο περιέχει ένα **LUT** των 4 εισόδων που δίνει την δυνατότητα για υλοποίηση συνδυαστικής λογικής και ένα προγραμματιζόμενο καταχωρητή που παρέχει την δυνατότητα υλοποίησης ακολουθιακής λογικής.

Οκτώ (8) LEs ομαδοποιούνται και έτσι αποτελούν ένα λογικό πίνακα από blocks (**LAB**). Κάθε FLEX 8000 LAB αποτελεί μια ανεξάρτητη δομή με κοινές εισόδους, διασυνδέσεις και σήματα ελέγχου. Η LAB αρχιτεκτονική παρέχει συσκευές που έχουν υψηλή απόδοση και απλές διασυνδέσεις.

ο σχήμα 1 δείχνει την αρχιτεκτονική της οικογένειας FLEX 8000 . Τα LABs ταξινομούνται σε γραμμές και στήλες. Οι I/O ακροδέκτες υποστηρίζονται από I/O στοιχεία (**IOEs**) που βρίσκονται στο τέλος κάθε γραμμής και στήλης. Κάθε IOE περιέχει έναν δικατευθυντήριο I/O απομονωτή και ένα flipflop που μπορεί να χρησιμοποιηθεί σαν καταχωρητής εισόδου ή εξόδου.



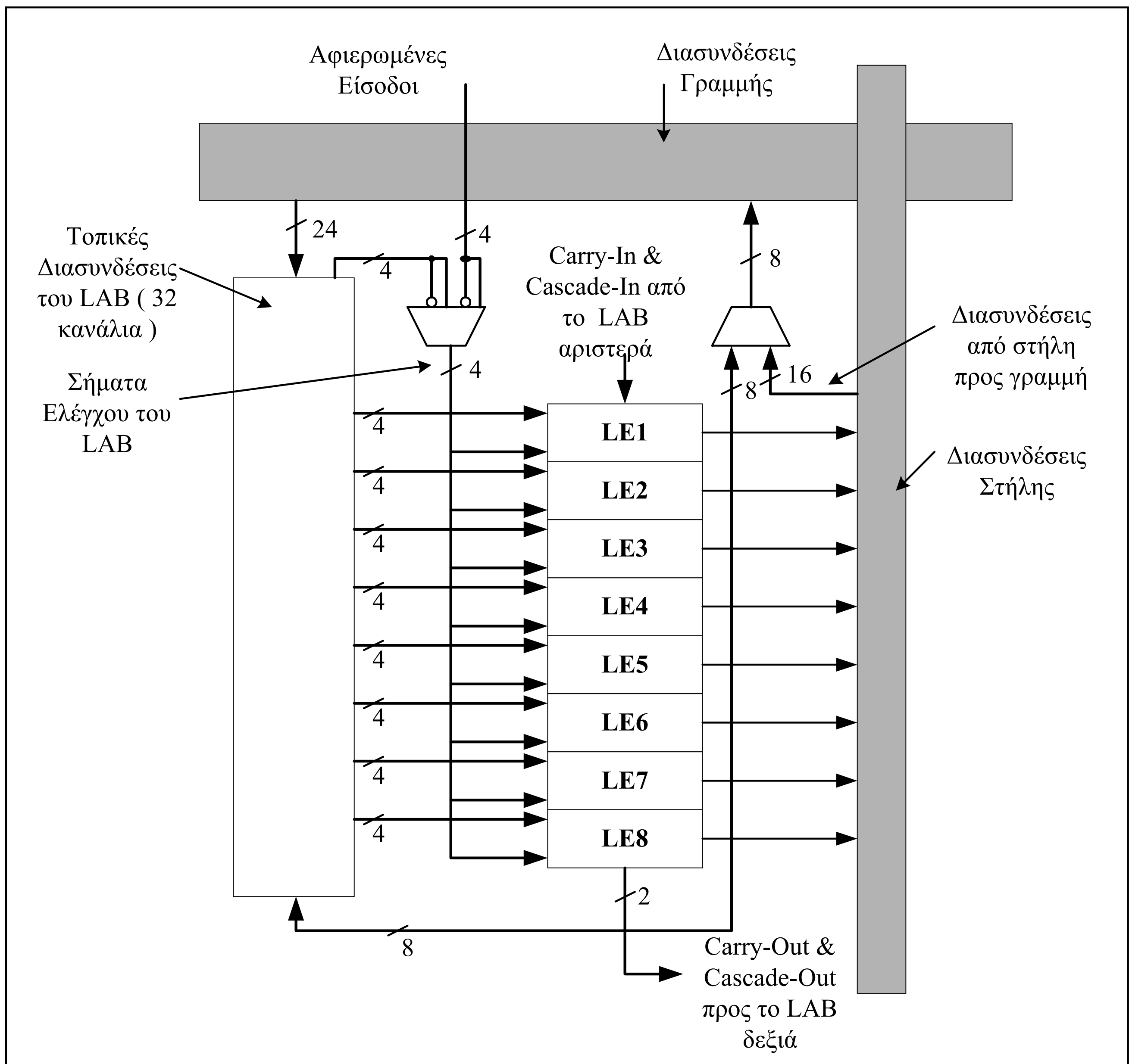
Σχήμα 1
Αρχιτεκτονική της Οικογένειας FLEX 8000.

Οι διασυνδέσεις σημάτων στις συσκευές τις οικογένειας FLEX 8000 καθώς και μεταξύ των ακροδεκτών της συσκευής γίνονται με χρήση του **FastTrack Interconnect**, που είναι μια σειρά από γρήγορα και συνεχή κανάλια που διανύουν όλο το μήκος και το πλάτος της συσκευής. Τα IOEs είναι τοποθετημένα στο τέλος κάθε γραμμής (οριζόντια) και στήλης (κάθετα) του FastTrack Interconnect μονοπατιού.

2.4 Δομή του LAB

Ένας λογικός πίνακας από blocks (**LAB**) αποτελείται από οκτώ LEs, τις σχετιζόμενες με αυτά carry και cascade αλυσίδες, τα σήματα ελέγχου του LAB και τις τοπικές διασυνδέσεις του LAB. Η δομή αυτή επιτρέπει στις συσκευές της οικογένειας FLEX 8000 να έχουν αποδοτικές διασυνδέσεις και υψηλή απόδοση λειτουργίας. Το σχήμα 2 δείχνει την δομή του LAB για τις συσκευές της οικογένειας 8000.

Κάθε LAB δίνει τέσσερα σήματα ελέγχου που μπορούν να χρησιμοποιηθούν από όλα τα LEs. Δύο απ΄ αυτά τα σήματα μπορούν να χρησιμοποιηθούν σαν ρολόγια και τα άλλα δύο σαν clear ή preset σήματα. Τα σήματα ελέγχου του LAB μπορούν να οδηγηθούν απευθείας είτε από κάποια αφιερωμένη είσοδο, ή από ένα I/O ακροδέκτη είτε από κάποιο εσωτερικό σήμα. Οι αφιερωμένες είσοδοι χρησιμοποιούνται συνήθως, για καθολικά ρολόγια, *clear* και *preset* σήματα επειδή παρέχουν σύγχρονο έλεγχο με χαμηλό *skew* σ΄ όλο το μήκος της



Σχήμα 2
Η δομή του LAB για την οικογένεια FLEX 8000.

συσκευής. Εάν λογική απαιτείται πάνω σ΄ ένα σήμα ελέγχου, τότε αυτή μπορεί να δημιουργηθεί σ΄ ένα ή περισσότερα LEs σ΄ ένα LAB και να οδηγηθεί στο τμήμα των τοπικών διασυνδέσεων του επιθυμητού LAB.

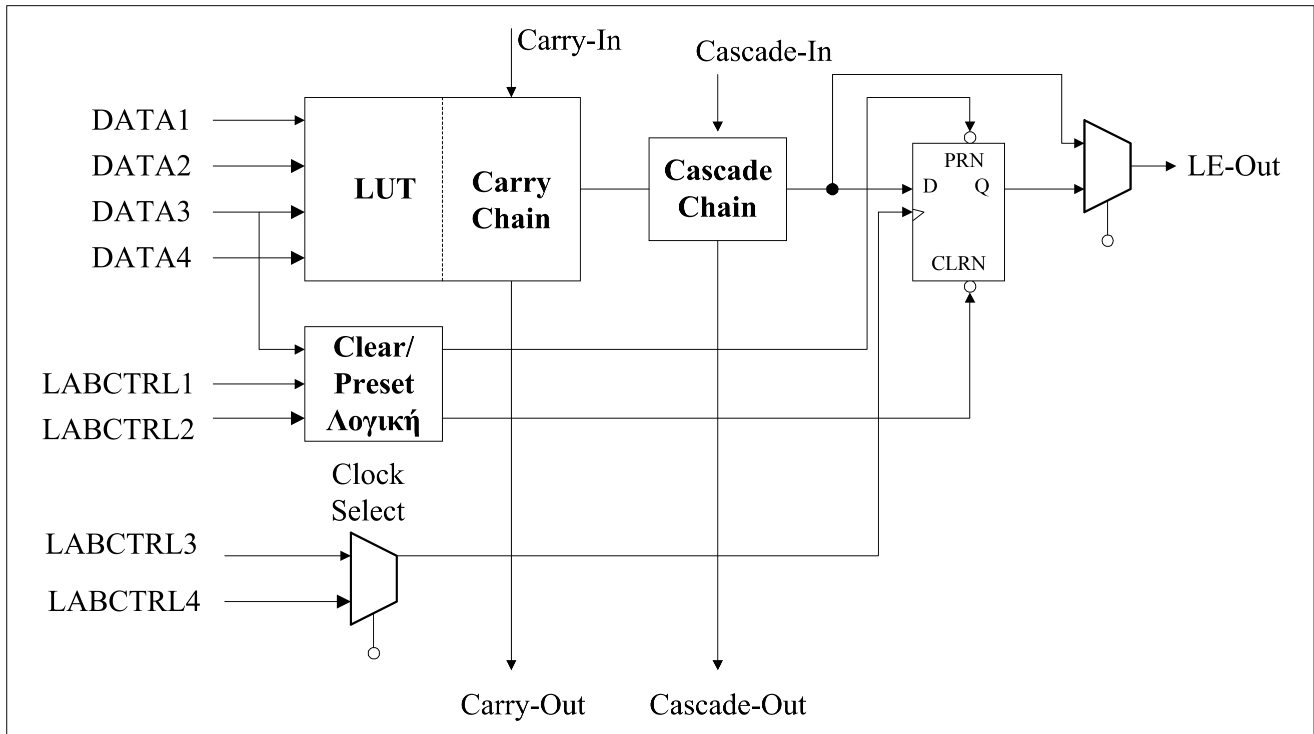
2.5 Τα Λογικά Στοιχεία

Το λογικό στοιχείο (**LE**) είναι η στοιχειώδης μονάδα λογικής στην αρχιτεκτονική της οικογένειας FLEX 8000 και έχει σταθερό μέγεθος. Κάθε LE περιέχει ένα LUT τεσσάρων εισόδων, ένα προγραμματιζόμενο flipflop, μία carry αλυσίδα και μία cascade αλυσίδα. Στο σχήμα 3 δίνεται το διάγραμμα ενός LE.

Το LUT μπορεί να θεωρηθεί σαν μία γεννήτρια συναρτήσεων, που γρήγορα υπολογίζει οποιαδήποτε συνάρτηση τεσσάρων μεταβλητών. Το προγραμματιζόμενο flipflop μπορεί να διαμορφωθεί για να λειτουργεί

σαν D, T, JK ή SR flipflop. Για καθαρά συνδυαστικά κυκλώματα το flipflop παρακάμπτεται και η έξοδος του LUT περνά κατευθείαν στην έξοδο του LE.

Η αρχιτεκτονική της οικογένειας FLEX 8000 περιέχει ακόμα, δύο αφιερωμένα υψηλής ταχύτητας μονοπάτια (τις carry και cascade αλυσίδες), που διασυνδέουν γειτονικά LEs χωρίς να χρησιμοποιούν



Σχήμα 3
Η δομή του LE για την οικογένεια FLEX 8000.

τα τοπικά μονοπάτια διασυνδέσεων. Η carry αλυσίδα υποστηρίζει την σχεδίαση μετρητών και αθροιστών υψηλής ταχύτητας, ενώ η cascade αλυσίδα υλοποιεί συναρτήσεις με πολλές εισόδους με τις μικρότερες καθυστερήσεις. Η χρήση των αλυσίδων αυτών πρέπει να περιορίζεται μόνο σε κυκλώματα με κρίσιμους χρονισμούς γιατί η εκτεταμένη χρήση τους οδηγεί σε μείωση των δυνατοτήτων διασυνδέσεων.

2.5.1 Η Carry Αλυσίδα

Η αλυσίδα αυτή παρέχει μια πολύ γρήγορη (μικρότερη από 1ns) carry συνάρτηση μεταξύ των LEs. Η είσοδος Carry-In από το χαμηλότερης τάξης bit μετακινείται προς το υψηλότερης τάξης bit μέσω της Carry αλυσίδας και τροφοδοτεί το LUT και το επόμενο τμήμα της αλυσίδας αυτής. Ο συμβολομεταφραστής του MAX + PLUSII μπορεί να δημιουργήσει αυτόματα τις αλυσίδες αυτές κατά την διάρκεια του σχεδιασμού, αλλά αυτές μπορούν να εισαχθούν και από τον σχεδιαστή κατά την σχεδίαση χρησιμοποιώντας το κατάλληλο primitive (**carry**).

2.5.2 Η Cascade Αλυσίδα

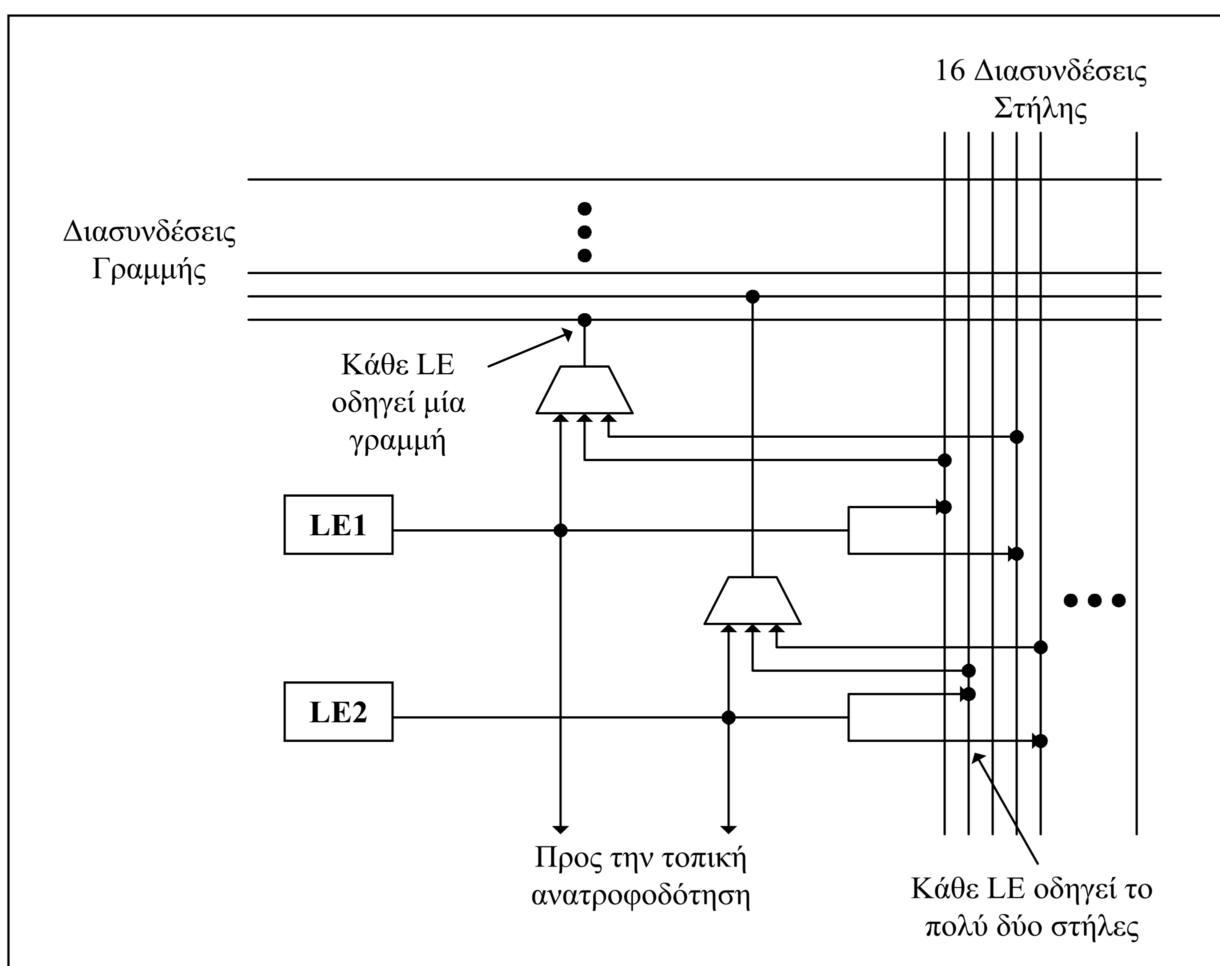
Με την Cascade αλυσίδα, η αρχιτεκτονική της οικογένειας FLEX 8000 μπορεί να υλοποιήσει συναρτήσεις με μεγάλο fan-in. Γειτονικά LUTs μπορούν να χρησιμοποιηθούν για να υπολογιστούν τμήματα από κάποια συνάρτηση. Για να διασυνδεθούν οι έξοδοι γειτονικών LEs η cascade αλυσίδα μπορεί να χρησιμοποιήσει μία λογική συνάρτηση AND ή OR. Κάθε επιπλέον LE δίνει τέσσερις ακόμα εισόδους στο ενεργό μήκος της συνάρτησης με καθυστέρηση 0.6ns για κάθε LE. Ο συμβολομεταφραστής του MAX + PLUSII μπορεί να δημιουργήσει αυτόματα τις αλυσίδες αυτές κατά την διάρκεια του σχεδιασμού, αλλά αυτές μπορούν να εισαχθούν και από τον σχεδιαστή κατά την σχεδίαση. Αλυσίδες μεγαλύτερες από οκτώ LEs δημιουργούνται αυτόματα, διασυνδέοντας LABs μεταξύ τους.

2.6 FastTrack Interconnect

Στην αρχιτεκτονική της οικογένειας FLEX 8000, οι διασυνδέσεις μεταξύ των LEs και των ακροδεκτών κάθε συσκευής παρέχονται από το FastTrack Interconnect που είναι μια σειρά από συνεχή οριζόντια και κάθετα κανάλια διασυνδέσεων που διασχίζουν όλη την συσκευή. Η δομή αυτή εξασφαλίζει αναμενόμενη απόδοση λειτουργίας ακόμα και σε σύνθετα κυκλώματα.

Τα LAB στην οικογένεια FLEX 8000, είναι κατασκευασμένα έτσι ώστε να δημιουργούν ένα πίνακα από γραμμές και στήλες. Κάθε γραμμή ενός LAB έχει μια αφιερωμένη γραμμή διασυνδέσεων που χρησιμοποιείται για την σύνδεση των σημάτων τόσο μέσα όσο και έξω από αυτό. Η γραμμή διασυνδέσεων μπορεί στην συνέχεια να οδηγήσει τους I/O ακροδέκτες ή να τροφοδοτήσει άλλα LAB μέσα στην ίδια συσκευή.

Κάθε LE σ' ένα LAB μπορεί να οδηγήσει το πολύ δύο διαφορετικές στήλες διασυνδέσεων. Γι' αυτό, οι δεκαέξι διαθέσιμες στήλες μπορούν να οδηγηθούν από το LAB. Τα κανάλια διασυνδέσεων διατρέχουν κάθετα την συσκευή και δίνουν πρόσβαση σε LABs στην ίδια στήλη αλλά σε διαφορετικές γραμμές. Ο συμβολομεταφραστής του MAX + PLUSII επιλέγει ποια LEs πρέπει να διασυνδεθούν σε μία στήλη. Μία γραμμή διασύνδεσης οδηγείται είτε από την έξοδο ενός LE είτε από δύο στήλες διασυνδέσεων. Αυτά τα τρία σήματα θέτονται στις εισόδους ενός πολυπλέκτη η έξοδος του οποίου συνδέεται σε κάποια συγκεκριμένη γραμμή διασύνδεσης. Κάθε LE είναι συνδεδεμένο σε ένα 3 -σε-1 πολυπλέκτη. Σ' ένα LAB, οι πολυπλέκτες εξασφαλίζουν ότι οι 16 στήλες διασυνδέσεων προσπελαύνουν 8 γραμμές διασυνδέσεων. Η δομή των διασυνδέσεων σ' ένα LAB δείχνονται στο σχήμα 4.



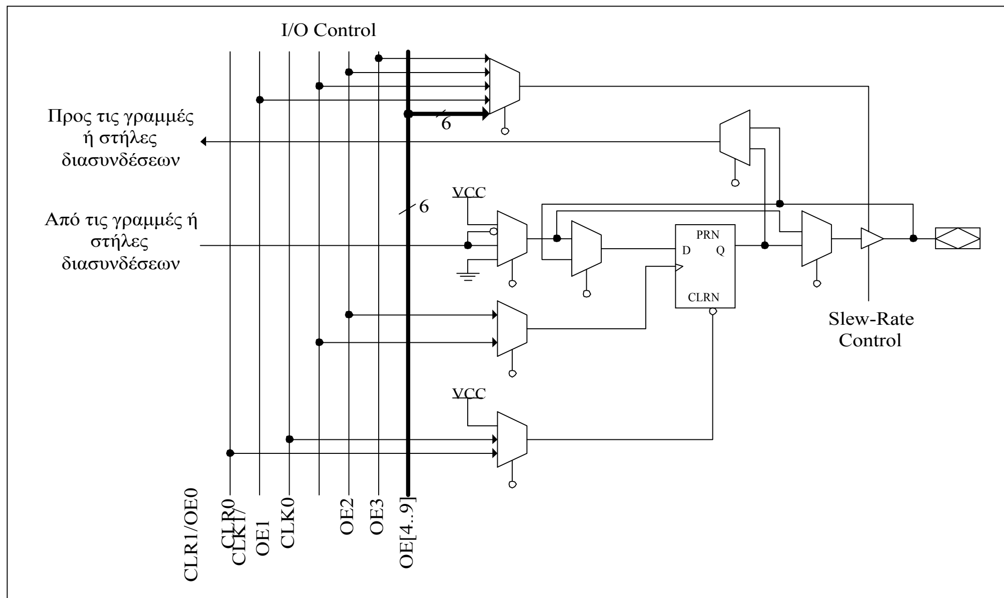
Σχήμα 4
Διασυνδέσεις γραμμής και στήλης σ' ένα LAB.

Κάθε στήλη ενός LAB έχει μια αφιερωμένη στήλη διασύνδεσης στην οποία συνδέονται σήματα που προέρχονται από άλλα LABs. Οι διασυνδέσεις στήλης μπορούν στην συνέχεια να οδηγήσουν I/O ακροδέκτες ή να τροφοδοτηθούν μέσα στις διασυνδέσεις γραμμής έτσι ώστε να προκύψουν τα σήματα που θα οδηγήσουν άλλα LABs. Ένα σήμα από τις διασυνδέσεις στήλης, που

μπορεί να είναι είτε η έξοδος του LE είτε μία είσοδος από ένα I/O ακροδέκτη, πρέπει πρώτα να τροφοδοτήσει κάποια διασύνδεση γραμμής πριν να εισέλθει σ' ένα LAB.

2.7 Στοιχεία Εισόδου/Εξόδου

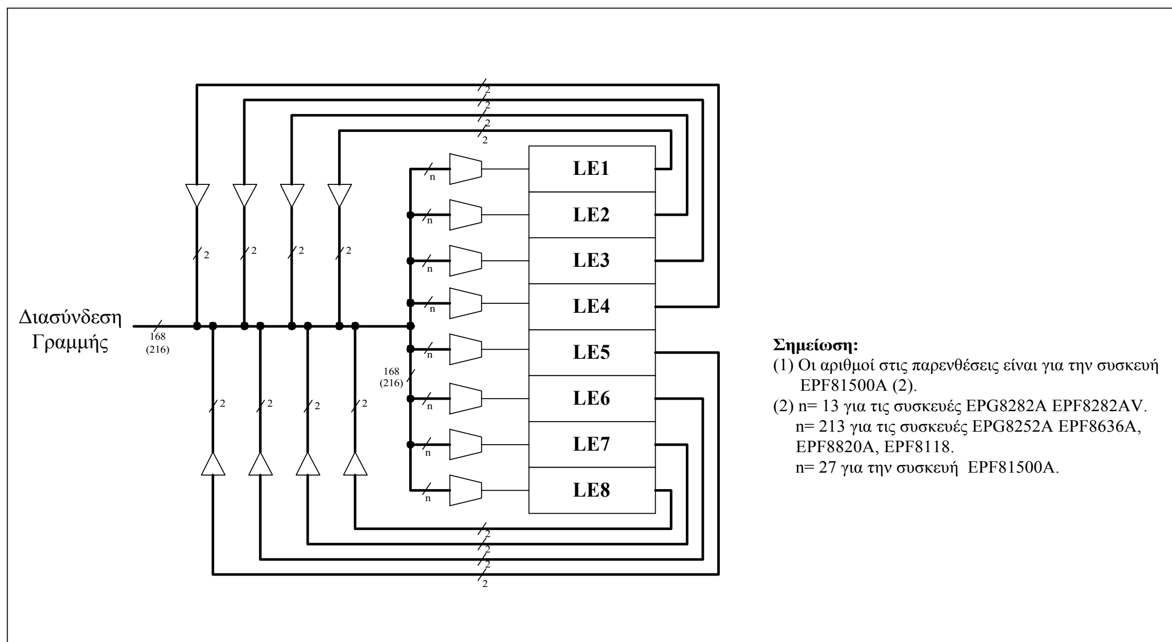
Ένα στοιχείο εισόδου/εξόδου (**IOE**) περιέχει ένα δικατευθυντήριο απομονωτή εισόδου/εξόδου και ένα καταχωρητή που μπορεί να χρησιμοποιηθεί είτε σαν καταχωρητής εισόδου για εξωτερικά δεδομένα που απαιτούν γρήγορους χρόνους αποκατάστασης, είτε σαν καταχωρητής εξόδου για δεδομένα που απαιτούν γρήγορους χρόνους απόκρισης (με βάση το ρολόι). Τα IOEs μπορούν να χρησιμοποιηθούν σαν είσοδοι, έξοδοι ή δικατευθυντήριοι ακροδέκτες. Το σχήμα 5 δείχνει το block διάγραμμα κάθε IOE.



Σχήμα 5
Η δομή του IOE.

2.7.1 Διασυνδέσεις από γραμμή σε στοιχείο Εισόδου/Εξόδου

Το σχήμα 6 δείχνει την σύνδεση μεταξύ IOEs και γραμμών. Ένα σήμα εισόδου από ένα IOE μπορεί να

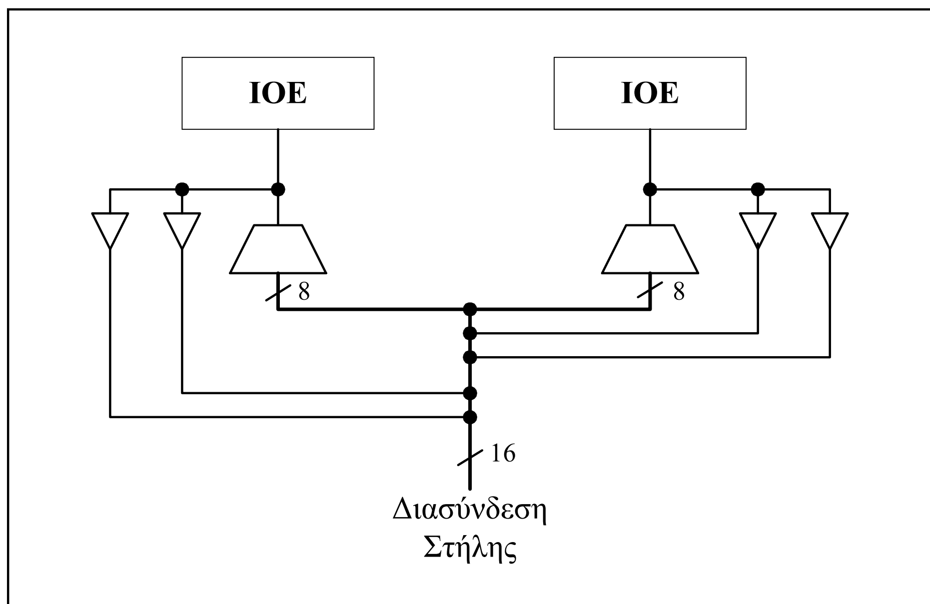


Σχήμα 6
Διασυνδέσεις από γραμμή σε στοιχείο Εισόδου/Εξόδου στην οικογένεια FLEX 8000.

οδηγήσει δύο διαφορετικά κανάλια γραμμών διασυνδέσεων. Όταν ένα IOE χρησιμοποιείται σαν έξοδος, το σήμα οδηγείται από ένα n-1 πολυπλέκτη που επιλέγει τα κανάλια γραμμών. Το μέγεθος μεταβάλλεται ανάλογα με τον αριθμό των στηλών σε μία συσκευή.

2.7.2 Διασυνδέσεις από στήλη σε στοιχείο Εισόδου/Εξόδου

Δύο IOEs εντοπίζονται στην αρχή και στο τέλος των καναλιών στηλών διασυνδέσεων, όπως δείχνεται στο σχήμα 7. Όταν ένα IOE χρησιμοποιείται σαν είσοδος, μπορεί να οδηγήσει το πολύ δύο διαφορετικές στήλες. Το σήμα εξόδου σ' ένα IOE μπορεί να επιλέξει το πολύ οκτώ από τι δεκαέξι στήλες χρησιμοποιώντας ένα 8-σε-1 πολυπλέκτη.



Σχήμα 7
Διασυνδέσεις από στήλη σε στοιχείο Εισόδου/Εξόδου στην οικογένεια FLEX 8000.

Συμπληρώνοντας τους γενικής χρήσης I/O ακροδέκτες, στις συσκευές της οικογένειας FLEX8000 υπάρχουν ακόμα τέσσερις αφιερωμένοι ακροδέκτες εισόδου. Αυτές οι είσοδοι προσφέρουν χαμηλό skew, καλή κατανομή σημάτων μέσα στην συσκευή και γενικά χρησιμοποιούνται για την σύνδεση του ρολογιού και των σημάτων αρχικοποίησης της λειτουργίας του σχεδιαζόμενου κυκλώματος.

ΚΕΦΑΛΑΙΟ 3

ΣΧΕΔΙΑΣΗ ΜΕ ΧΡΗΣΗ ΤΗΣ ΓΛΩΣΣΑΣ AHDL

3.1 Εισαγωγή

Η Γλώσσα Περιγραφής Υλικού της Altera (**AHDL**) είναι μία γλώσσα υψηλού επιπέδου που είναι ενσωματωμένη στο περιβάλλον του MAX + PLUSII. Είναι περισσότερο αποδοτική για την σχεδίαση σύνθετων συνδυαστικών κυκλωμάτων, πινάκων αληθείας και μηχανών κατάστασης. Για την δημιουργία AHDL αρχείων (***.tdf***) μπορεί να χρησιμοποιηθεί ο κειμενογράφος του MAX + PLUSII ή οποιοσδήποτε άλλος κειμενογράφος. Στην συνέχεια, τα TDF μπορούν να συμβολομεταφραστούν και να εξομοιωθούν μέσα στο ίδιο κέλυφος του προγράμματος.

Οι AHDL δηλώσεις είναι αρκετά δυναμικές και απλές στην χρήση. Μπορούν να σχεδιαστούν μεγάλα, ιεραρχημένα κυκλώματα αποκλειστικά με χρήση της AHDL ή μεικτά με χρήση σχηματικών και AHDL. Από τον κειμενογράφο, μέσα στο περιβάλλον, δίνεται η δυνατότητα για την αυτόματη δημιουργία συμβόλου που αναπαριστάνει ένα TDF αρχείο έτσι ώστε να μπορεί να χρησιμοποιηθεί σε σχηματικά. Κατά τον ίδιο τρόπο, μπορούν να δημιουργηθούν συναρτήσεις σε οποιοδήποτε TDF. Παρέχονται επίσης, Include αρχεία (***.inc***) για όλες τις συναρτήσεις στην βιβλιοθήκη του MAX + PLUSII ή αυτές που δημιουργούνται από τον σχεδιαστή. Γενικά η AHDL, αποτελείται από μια πληθώρα από εξαρτήματα που χρησιμοποιούνται σε δηλώσεις συμπεριφοράς έτσι ώστε να περιγράψουν κάποια λογική λειτουργία. Το κεφάλαιο αυτό περιέχει πληροφορίες για το τρόπο διαχείρισης αυτών των δηλώσεων.

3.2 Χρήση Αριθμών

Οι αριθμοί χρησιμοποιούνται για να ορίσουν σταθερές τιμές σε Boolean εκφράσεις και εξισώσεις. Η AHDL υποστηρίζει την χρήση δεκαδικών, δυαδικών, οκταδικών και δεκαεξαδικών αριθμών. Το συντακτικό για κάθε βάση δείχνεται στην συνέχεια:

Radix:	**Values:**
Δεκαδικοί	<ψηφία από 0 εώς 9>
Δυαδικοί	B"<σειρά από 0's, 1's, X's>" (όπου X = "don't care")
Οκταδικοί	O"< ψηφία από 0 έως 7>" ή Q"< ψηφία από 0 έως 7>"
Δεκαεξαδικοί	X"< ψηφία από 0 έως 9, A έως F>" ή H"< ψηφία από 0 έως 9, A to F>"

Στο παρακάτω παράδειγμα δείχνονται έγκυροι αριθμοί στην AHDL:

B"0110X1X10"
Q"4671223"
H"123AECF"

Αριθμοί δεν μπορούν να ανατεθούν σε απλούς κόμβους σε Boolean εξισώσεις. Στην περίπτωση αυτή, χρησιμοποιούνται τα `VCC` και `GND` στοιχεία. Το αρχείο *decode1.tdf* που δείχνεται στο παρακάτω παράδειγμα

δείχνει έναν αποκωδικοποιητή διεύθυνσης που δημιουργεί ένα ενεργό σε high σήμα enable όταν η διεύθυνση είναι 370Hex.

```
SUBDESIGN decode1
(
   address[15..0] : INPUT;
   chip_enable     : OUTPUT;
)
BEGIN
   chip_enable = (address[15..0] == H"0370");
END;
```

Στο παράδειγμα αυτό, οι δεκαδικοί αριθμοί 15 και 0 χρησιμοποιούνται για να ορίσουν τα bit του address bus. Ο δεκαεξαδικός αριθμός `H"0370"` καθορίζει την διεύθυνση που κωδικοποιείται.

3.3 Δήλωση σταθερών

Μία δήλωση σταθερών επιτρέπει την αντικατάσταση ενός αριθμού μ' ένα συμβολικό όνομα. Το όνομα αυτό αναπαριστά επομένως τον αριθμό αυτό. Στην AHDL, οι σταθερές υλοποιούνται με **Constant** δηλώσεις. Στα παρακάτω παραδείγματα δείχνονται δηλώσεις σταθερών.

```
CONSTANT UPPER_LIMIT = 130;
CONSTANT BAR = 1 + 2 DIV 3 + LOG2(256);
CONSTANT FOO = 1;
```

Μια δήλωση σταθεράς έχει τα ακόλουθα χαρακτηριστικά:

a) Μία δήλωση σταθεράς αρχίζει με την λέξη CONSTANT που ακολουθείται από ένα συμβολικό όνομα, το σύμβολο της ισότητας (=) και ένα αριθμό.
b) Η δήλωση τελειώνει με το σύμβολο ;.
c) Όταν μία σταθερά δηλωθεί, τότε μπορεί να χρησιμοποιηθεί κατά την αναπαράσταση του αριθμού σ' ένα TDF αρχείο.

Οι δηλώσεις σταθερών πρέπει να ικανοποιούν τους παρακάτω κανόνες:

a) Μία σταθερά μπορεί να χρησιμοποιηθεί μόνο μετά την δήλωσή της.
b) Το όνομα μιας σταθεράς πρέπει να είναι μοναδικό.
c) Μία δήλωση σταθεράς μπορεί να χρησιμοποιηθεί πολλές φορές σ' ένα TDF αρχείο.

Το *decode2.tdf* αρχείο που δείχνεται στην συνέχεια έχει την ίδια λειτουργία με το *decode1.tdf* αρχείο, με την διαφορά ότι χρησιμοποιεί την σταθερά `IO_ADDRESS` αντί για τον αριθμό `H"0370"`.

```
CONSTANT IO_ADDRESS = H"0370";
SUBDESIGN decode2
(
   a[15..0] : INPUT;
   ce       : OUTPUT;
)
BEGIN
   ce = (a[15..0] == IO_ADDRESS);
END;
```

Οι σταθερές είναι ιδιαίτερα χρήσιμες εάν κάποιος αριθμός επαναλαμβάνεται πολλές φορές σε ένα αρχείο (εάν ο αριθμός αλλάζει, μόνο η Constant δήλωση πρέπει να μεταβληθεί).

3.4 Κυκλώματα συνδυαστικής λογικής

Κάποιο κύκλωμα είναι συνδυαστικής λογικής εάν οι έξοδοι σε καθορισμένη χρονική στιγμή είναι συνάρτηση μόνο των εισόδων. Κυκλώματα συνδυαστικής λογικής υλοποιούνται στην AHDL με Boolean

εκφράσεις και εξισώσεις, πίνακες αληθείας και ένα πλήθος από συναρτήσεις. Παραδείγματα κυκλωμάτων συνδυαστικής λογικής είναι οι πολυπλέκτες, οι αθροιστές, οι κωδικοποιητές, κ.ο.κ.

3.4.1 Υλοποίηση Boolean συναρτήσεων και εκφράσεων

Οι Boolean εκφράσεις είναι ένα σύνολο από κόμβους, αριθμούς, σταθερές και άλλες εκφράσεις που διαχωρίζονται από τελεστές ή σύμβολα σύγκρισης. Μία Boolean εξίσωση θέτει ένα κόμβο ή ένα σύνολο από κόμβους ίσο με την τιμή κάποιας Boolean έκφρασης. Μία Boolean εξίσωση μπορεί να είναι μία από τις ακόλουθες:

a) Ένας τελεστής (Παράδειγμα: `a, b[5..1], 7, VCC`).
b) Αναφορά σε εξάρτημα ή συνάρτηση (Παράδειγμα: `out[15..0] = 16dmux(q[3..0]);`).
c) Ένας τελεστής (Παράδειγμα: `!c`).
d) Δύο Boolean εκφράσεις που διαχωρίζονται από ένα δυαδικό τελεστή (Παράδειγμα: `d1 $ d3`).
e) Μία Boolean έκφραση που περιέχεται σε παρενθέσεις (Παράδειγμα: `(!foo & bar)`).

Το *boole1.tdf* που δείχνεται στην συνέχεια, αναπαριστάνει δύο λογικές πύλες.

```
SUBDESIGN boole1
(
   a0, a1, b  : INPUT;
   out1, out2 : OUTPUT;
)
BEGIN
   out1 = a1 & !a0;
   out2 = out1 # b;
END;
```

Στο αρχείο αυτό, η έξοδος `out1` οδηγείται από μια λογική πύλη `AND` που έχει σαν εισόδους το `a1` και το αντίστροφο του `a0`, και το σήμα `out2` οδηγείται από μια λογική `OR` που έχει σαν εισόδους το `out1` και το `b`.

3.4.2 Δηλώσεις κόμβων

Ένας κόμβος, που δηλώνεται με μια **Node** δήλωση στο τμήμα δηλώσεων μεταβλητών μπορεί να χρησιμοποιηθεί για να διατηρήσει την τιμή κάποιου σήματος. Μία μεταβλητή τύπου Node μπορεί να χρησιμοποιηθεί είτε στο δεξιό είτε στο αριστερό τμήμα μίας συνάρτησης. Το *boole2.tdf* αρχείο, που δείχνεται στην συνέχεια, υλοποιεί την ίδια λογική με το *boole1.tdf* αλλά έχει μόνο μία είσοδο.

```
SUBDESIGN boole2
(
   a0, a1, b   : INPUT;
   out         : OUTPUT;
)
VARIABLE
   a_equals_2  : NODE;
BEGIN
   a_equals_2 = a1 & !a0;
   out = a_equals_2 # b;
END;
```

Στο αρχείο αυτό, δηλώνεται ο κόμβος `a_equals_2` στον οποίο ανατίθεται η έκφραση `a1 & !a0`. Η χρήση κόμβων μπορεί να ελαττώσει των αριθμό των LE που απαιτούνται σε μία σχεδίαση όταν ο κόμβος αυτός χρησιμοποιείται σε πολλαπλές εκφράσεις.

3.4.3 Δήλωση συνόλων

Ένα σύνολο που περιέχει το πολύ 256 μέλη, αντιμετωπίζεται σαν μια συλλογή από κόμβους και επεξεργάζεται σαν μία μονάδα. Σε Boolean εξισώσεις, ένα σύνολο μπορεί να τεθεί ίσο με μια Boolean έκφραση, ένα άλλο σύνολο, ένα κόμβο, `VCC`, `GND`, 1, ή 0. Σε κάθε περίπτωση, η τιμή του συνόλου είναι διαφορετική.

Όταν ένα σύνολο ορίζεται, η χρήση του συμβόλου [ ] δίνει ένα σύντομο τρόπο για να οριστεί όλη η κλίμακα. Το *group1.tdf* αρχείο δείχνει μία απλή Boolean εξίσωση που ορίζει δύο σύνολα.

```
OPTIONS BIT0 = MSB;
CONSTANT MAX_WIDTH = 1+2+3-3-1;  % MAX_WIDTH = 2 %
SUBDESIGN group1
(
   a[1..2], use_exp_in[1+2-2..MAX_WIDTH]        : INPUT;
   d[1..2], use_exp_out[1+2*2-4..MAX_WIDTH]     : OUTPUT;
   dual_range[5..4][3..2]                       : OUTPUT;
)
BEGIN
   d[ ] = a[ ] + B"10";
   use_exp_out[ ] = use_exp_in[ ];
   dual_range[ ][ ] = VCC;
END;
```

3.4.4 Υλοποίηση Λογικής υπό συνθήκη

Η υπό συνθήκη λογική επιλέγει ανάμεσα από διαφορετικές συμπεριφορές ανάλογα με τις τιμές εισόδου. Οι **If** και **Case** δηλώσεις είναι ιδανικές για την υλοποίηση λογικής υπό συνθήκη.

a) Οι **If** δηλώσεις ελέγχουν μία ή περισσότερες Boolean εκφράσεις και στην συνέχεια περιγράφουν την επιθυμητή συμπεριφορά για διαφορετικές τιμές της έκφρασης.

b) Οι **Case** δηλώσεις δίνουν μια λίστα από εναλλακτικές συμπεριφορές για κάθε τιμή μιας έκφρασης.

3.4.4.1 Η δήλωση If

Η **If** δήλωση δημιουργεί μια λίστα από δηλώσεις συμπεριφοράς που ενεργοποιούνται μετά από την θετική επιβεβαίωση μιας Boolean έκφρασης. Η δήλωση αυτή έχει τα ακόλουθα χαρακτηριστικά:

a) Οι λέξεις `If` και `Then` περικλείουν την Boolean έκφραση που θα ελέγχεται και ακολουθούνται από μία ή περισσότερες δηλώσεις συμπεριφοράς, κάθε μία από τις οποίες “κλείνει” με το σύμβολο (;).

b) Οι λέξεις `Elsif` και `Then` περικλείουν επιπλέον Boolean εκφράσεις που θα ελεγχθούν και επίσης ακολουθούνται από μία ή περισσότερες δηλώσεις συμπεριφοράς.

c) Οι δηλώσεις συμπεριφοράς που ακολουθούν την λέξη `Then` ενεργοποιούνται για την πρώτη έκφραση που ανιχνεύεται ότι είναι αληθής.

Το *priority.tdf* αρχείο, που δείχνεται στην συνέχεια δείχνει ένα κωδικοποιητή προτεραιοτήτων που μετατρέπει το επίπεδο της εισόδου με την υψηλότερη προτεραιότητα σε μια τιμή. Δημιουργεί ένα 2-bit κώδικα που υποδεικνύει την είσοδο υψηλότερης προτεραιότητας που οδηγείται από το `VCC`.

```
SUBDESIGN priority
(
   low, middle, high   : INPUT;
   highest_level[1..0] : OUTPUT;
)
BEGIN
   IF high THEN
      highest_level[ ] = 3;
   ELSIF middle THEN
      highest_level[ ] = 2;
   ELSIF low THEN
      highest_level[ ] = 1;
   ELSE
      highest_level[ ] = 0;
   END IF;
END;
```

Στο παράδειγμα αυτό, οι είσοδοι `middle`, `high` και `low` επιβλέπονται έτσι ώστε να αποφασιστεί εάν οδηγούνται από το `VCC`.

3.4.4.2 Η Δήλωση Case

Η **Case** δήλωση δημιουργεί μία λίστα από λειτουργίες που ενεργοποιούνται ανάλογα με την τιμή μιας μεταβλητής ή έκφρασης που ακολουθεί τη λέξη Case. Η δήλωση αυτή έχει τα ακόλουθα χαρακτηριστικά:

a) Οι λέξεις `Case` και `Is` περιέχουν μία Boolean έκφραση ή μηχανή κατάστασης.
b) Η `Case` δήλωση ολοκληρώνεται με τις λέξεις `END CASE` και το σύμβολο (`;`).
c) Εάν μία Boolean έκφραση που ακολουθεί την `Case` δήλωση πάρει μια συγκεκριμένη τιμή τότε οι δηλώσεις συμπεριφοράς που ακολουθούν, ενεργοποιούνται.
d) Κάθε δήλωση συμπεριφοράς ολοκληρώνεται με το σύμβολο (`;`).

Το αρχείο *decoder.tdf* που δείχνεται στην συνέχεια, δείχνει ένα 2-bit-σε-4bit κωδικοποιητή. Μετατρέπει μία δυαδική κωδική λέξεις εισόδου σε μία “one-hot” κωδική λέξη.

```
SUBDESIGN decoder
(
      code[1..0]  : INPUT;
      out[3..0]   : OUTPUT;
)
BEGIN
      CASE code[] IS
            WHEN 0 => out[] = B"0001";
            WHEN 1 => out[] = B"0010";
            WHEN 2 => out[] = B"0100";
            WHEN 3 => out[] = B"1000";
      END CASE;
END;
```

Στο παράδειγμα αυτό, η είσοδος `code[1..0]` έχει μία από τις τιμές 0, 1, 2 ή 3. Η εξίσωση που ακολουθεί το σύμβολο => ορίζει την τιμή της εξόδου ανάλογα με την τιμή της εισόδου.

3.4.5 Υλοποίηση Δικατευθυντήριων ακροδεκτών

Το MAX + PLUSII επιτρέπει I/O ακροδέκτες να διαμορφωθούν σαν δικατευθυντήριοι ακροδέκτες. Οι δικατευθυντήριοι ακροδέκτες ορίζονται με ένα **BIDIR** port που συνδέεται στην έξοδο ενός **TRI** στοιχείου. Το σήμα μεταξύ του ακροδέκτη και του `TRI` είναι ένα δικατευθυντήριο σήμα που μπορεί να χρησιμοποιηθεί για να οδηγήσει άλλη λογική στο σχεδιαζόμενο κύκλωμα.
Το bus_reg2.tdf αρχείο, που δείχνεται στην συνέχεια, υλοποιεί ένα καταχωρητή που δειγματοληπτεί την τιμή που έχει το tri-state bus. Μπορεί επίσης να οδηγήσει την αποθηκευμένη τιμή πίσω στο bus.

```
SUBDESIGN bus_reg2
(
   clk : INPUT;
   oe  : INPUT;
   io  : BIDIR;
)
VARIABLE
   dff_out : NODE;
BEGIN
   dff_out = DFF(io, clk, ,);
   io = TRI(dff_out, oe);
END;
```

Το δικατευθυντήριο `io` σήμα, που οδηγείται από το `TRI`, χρησιμοποιείται σαν η `d` είσοδος σ’ ένα D flipflop. Τα κόμματα χρησιμοποιούνται στην θέση των clrn και prn σημάτων του flipflop.

3.4.6 Δήλωση Πινάκων Αληθείας

Η δήλωση πινάκων αληθείας χρησιμοποιείται για την υλοποίηση συνδυαστικής λογικής και των καταστάσεων μιας μηχανής κατάστασης. Σ’ ένα πίνακα αληθείας στην AHDL, κάθε καταχώρηση στον πίνακα

περιέχει ένα συνδυασμό από τιμές εισόδου που θα δίνουν καθορισμένες τιμές εξόδου. Οι τιμές αυτές εξόδου μπορούν να χρησιμοποιηθούν σαν ανατροφοδότηση για τον ορισμό μεταβάσεων μεταξύ καταστάσεων καθώς και των εξόδων μιας μηχανής κατάστασης. Το παρακάτω παράδειγμα δείχνει μία δήλωση ενός πίνακα αληθείας:

```
TABLE
      a0,         f[4..1].q   =>          f[4..1].d,  control;
      0,          B"0000"           =>    B"0001",           1;
      0,          B"0100"           =>    B"0010",           0;
      1,          B"0XXX"           =>    B"0100",           0;
      X,          B"1111"           =>    B"0101",           1;
END TABLE;
```

Η δήλωση ενός πίνακα αληθείας έχει τα ακόλουθα χαρακτηριστικά:

a) Η επικεφαλίδα του πίνακα αληθείας αποτελείται από την λέξη `TABLE`, που ακολουθείται από μία λίστα από εισόδους που διαχωρίζονται από το σύμβολο (,), το σύμβολο (`=>`) και μια λίστα από εξόδους που διαχωρίζονται μεταξύ τους από το σύμβολο (,). Η επικεφαλίδα "κλείνει" με το σύμβολο (`;`).

b) Οι είσοδοι στον πίνακα αληθείας είναι Boolean εκφράσεις σε αντίθεση με τις εξόδους του πίνακα αληθείας που είναι μεταβλητές.

c) Κάθε σήμα έχει μία ένα -προς -ένα αντιστοίχηση με τις τιμές σε κάθε καταχώρηση.

d) Οι τιμές εισόδου και εξόδου μπορούν να είναι αριθμοί, σταθερές (`VCC` ή `GND`), συμβολικά ονόματα σταθερών ή σύνολα από αριθμούς ή σταθερές.

e) Οι λέξεις `END TABLE`, ακολουθούμενες από το σύμβολο (`;`) "κλείνουν" την δήλωση του λογικού πίνακα.

Οι ακόλουθοι κανόνες εφαρμόζονται κατά την δήλωση ενός πίνακα αληθείας:

a) Τα ονόματα στις επικεφαλίδες του πίνακα μπορούν να είναι είτε απλοί κόμβοι είτε σύνολα κόμβων.

b) Κάθε πιθανός συνδυασμός εισόδων δεν είναι απαραίτητο να υπάρχει στην λίστα.

c) Ο αριθμός των αντικειμένων, που διαχωρίζονται μεταξύ τους με το σύμβολο (,) σε μια γραμμή του πίνακα αληθείας πρέπει να ισούται με τον αριθμό των αντικειμένων στην επικεφαλίδα του πίνακα αληθείας.

3.5 Ακολουθιακή Λογική

Όλα τα ακολουθιακά κυκλώματα πρέπει να περιέχουν ένα ή περισσότερα flipflop. Η ακολουθιακή λογική υλοποιείται συνήθως, στην AHDL, κατά την σχεδίαση καταχωρητών, μηχανών κατάστασης, μετρητών, ελεγκτών ή μανδαλωτών.

3.5.1 Δήλωση Καταχωρητών

Το τμήμα δηλώσεων μεταβλητών χρησιμοποιείται για την δήλωση καταχωρητών , περιέχοντας D, T, JK και SR flipflop. Οι καταχωρητές αποθηκεύουν δεδομένα και τα συγχρονίζουν μ' ένα σήμα ρολογιού. Μπορεί να υλοποιηθεί ένας καταχωρητής με την χρήση της δήλωσης Register στο τμήμα δηλώσεων. Όταν έχει δηλωθεί ένας καταχωρητής, τότε αυτός μπορεί να συνδεθεί σε οποιαδήποτε άλλη λογική σ' ένα TDF αρχείο χρησιμοποιώντας τα διαθέσιμα κανάλια. Ένα κανάλι είναι η είσοδος ή έξοδος ενός primitive, μιας συνάρτησης, μιας μηχανής κατάστασης και είναι συνώνυμο με το όνομα ενός ακροδέκτη σε ένα σχηματικό (**.gdf**) ή σε οποιοδήποτε άλλο αρχείο. Ένα κανάλι κάποιου συγκεκριμένου στοιχείου χρησιμοποιείται με χρήση της ακόλουθης μορφής:

```
<instance name>.<port name>
```

Το *bur_reg.tdf* αρχείο που δείχνεται στην συνέχεια, περιέχει ένα καταχωρητή ενός byte που θέτει τις τιμές στις d εισόδους στις q εξόδους κατά την ανερχόμενη παρυφή του ρολογιού όταν η είσοδος load γίνει high.

```
SUBDESIGN bur_reg
(
```

```
   clk, load, d[7..0]    : INPUT;
   q[7..0]               : OUTPUT;
)

VARIABLE
   ff[7..0]            : DFFE;

BEGIN

   ff[ ].clk = clk;
   ff[ ].ena = load;
   ff[ ].d = d[ ];
   q[ ] = ff[ ].q;

END;
```

Ο καταχωρητής δηλώνεται σαν ένα D flipflop με enable είσοδο (**DFFE**), στο τμήμα δηλώσεων των μεταβλητών. Η πρώτη Boolean εξίσωση στο λογικό τμήμα, συνδέει την είσοδο `clk` με την είσοδο clk των οκτώ flipflops. Η δεύτερη εξίσωση συνδέει την enable είσοδο των flipflop με το σήμα εισόδου load. Η τρίτη εξίσωση συνδέει τις εισόδους `d[7..0]` στις d εισόδους των flipflop. Η τέταρτη δήλωση συνδέει τις εξόδους `q[7..0]` στις εξόδους q των flipflop. Και οι τέσσερις δηλώσεις υπολογίζονται ταυτόχρονα.

Στην περίπτωση που ο καταχωρητής πρέπει να φορτωθεί σε κάποια συγκεκριμένη παρυφή του ρολογιού, προτείνεται η χρήση της enable εισόδου του **DFFE, TFFE, SRFFE ή JKFFE** flipflop έτσι ώστε να γίνεται ο έλεγχος κατά την φόρτωση του καταχωρητή.

3.5.2 Δήλωση Εξόδων Καταχωρητών

Μπορούν να δηλωθούν έξοδοι καταχωρητών ενός υποσχεδίου δηλώνοντας τις εξόδους σαν D flipflop σε μια Register δήλωση στο τμήμα των μεταβλητών. Το *reg_out.tdf* αρχείο έχει την ίδια λειτουργία με το *bur_reg.tdf* με την διαφορά ότι έχει την δήλωση εξόδου καταχωρητών.

```
SUBDESIGN reg_out
(
   clk, load, d[7..0] : INPUT;
   q[7..0]            : OUTPUT;
)
VARIABLE
   q[7..0]            : DFFE; % outputs also declared as registers %
BEGIN
   q[ ].clk = clk;
   q[ ].ena = load;
   q[ ] = d[ ];
END;
```

Όταν ανατίθεται μια τιμή σε μια τέτοια έξοδο, στο λογικό τμήμα δηλώσεων, η τιμή αυτή οδηγεί τις d εισόδους των καταχωρητών. Η έξοδος των καταχωρητών δεν αλλάζει μέχρι να ανιχνευθεί ανερχόμενη παρυφή του ρολογιού. Για τον ορισμό της εισόδου ρολογιού στον καταχωρητή, χρησιμοποιείται η δήλωση `<register name>.clk` στο τμήμα λογικών δηλώσεων.

3.5.3 Μετρητές

Οι μετρητές χρησιμοποιούν ακολουθιακή λογική για να μετρήσουν παλμούς ενός ρολογιού. Μερικοί μετρητές μπορούν να μετρούν προς τα πάνω και με αντίστροφη φορά και μπορούν να φορτωθούν με δεδομένα και να τεθούν στην τιμή μηδέν. Οι μετρητές υλοποιούνται με τη χρήση συνήθως D flipflop (**DFF ή DFFE**) και με **If** δηλώσεις. Το *ahdlcnt.tdf* αρχείο που δείχνεται στην συνέχεια, υλοποιεί ένα 16-bit μετρητή προς τα πάνω που μπορεί να φορτωθεί με συγκεκριμένη τιμή και να “ καθαρισθεί “ στο μηδέν.

```
SUBDESIGN ahdlcnt
(
   clk, load, ena, clr, d[15..0]    : INPUT;
   q[15..0]                         : OUTPUT;
)
```

```
VARIABLE
   count[15..0]                    : DFF;
BEGIN
   count[ ].clk = clk;
   count[ ].clrn = !clr;
   IF load THEN
      count[ ].d = d[ ];
   ELSIF ena THEN
      count[ ].d = count[ ].q + 1;
   ELSE
      count[ ].d = count[ ].q;
   END IF;
   q[ ] = count[ ];
END;
```

Στο αρχείο αυτό, 16 flipflop δηλώνονται στο τμήμα δηλώσεων των μεταβλητών με το όνομα `count0` έως `count15`. Η If δήλωση προσδιορίζει την τιμή που φορτώνεται στα flipflop στην ανερχόμενη παρυφή του ρολογιού.

3.6 Μηχανές Κατάστασης

Μία μηχανή κατάστασης δημιουργείται δηλώνοντας το όνομα της μηχανής κατάστασης, τις καταστάσεις της και προαιρετικά τα bit ανάθεσης. Κάθε κατάσταση της μηχανή κατάστασης αναπαρίσταται από ένα μοναδικό συνδυασμό από high και low εξόδους flipflop. Τα bit κατάστασης είναι τα flipflop που απαιτούνται από την μηχανή κατάστασης για την αποθήκευση των καταστάσεων της. Μία δήλωση μηχανής κατάστασης έχει τα ακόλουθα χαρακτηριστικά:

a) Το όνομα της μηχανής κατάστασης είναι ένα συμβολικό όνομα.
b) Το όνομα της μηχανής κατάστασης ακολουθείται από το σύμβολο (:) και την λέξη **MACHINE**.
c) Η δήλωση της μηχανής κατάστασης πρέπει να περιέχει μία λίστα από καταστάσεις.
d) Οι καταστάσεις ορίζονται με τις λέξεις **WITH STATES**, που ακολουθούνται από μία λίστα από συμβολικά ονόματα που διαχωρίζονται μεταξύ τους με το σύμβολο (,).
e) Η δήλωση της μηχανής κατάστασης "κλείνει" με το σύμβολο (;).

Οι μηχανές κατάστασης, όπως οι πίνακες αληθείας και οι Boolean εξισώσεις, υλοποιούνται με απλό τρόπο στην AHDL. Η γλώσσα είναι δομημένη κατά τέτοιο τρόπο έτσι ώστε να μπορούν να ανατεθούν bit κατάστασης και τιμές κατάστασης από τον χρήστη. Ο συμβολομεταφραστής χρησιμοποιεί ειδικούς αλγόριθμους για την αυτόματη ανάθεση καταστάσεων έτσι ώστε να ελαχιστοποιείται ο αριθμός των LE που χρησιμοποιούνται για την υλοποίηση της μηχανής κατάστασης. Ο συμβολομεταφραστής πραγματοποιεί τις ακόλουθες λειτουργίες:

a) Αναθέτει bit, επιλέγοντας είτε T είτε D flipflop για κάθε bit.
b) Αναθέτει τις τιμές στις καταστάσεις.
c) Πραγματοποιεί λογική σύνθεση.

Για τον ορισμό μιας μηχανής κατάστασης στην AHDL, η δήλωσή της πρέπει να περιέχει τα ακόλουθα στοιχεία:

a) Δήλωση της μηχανής κατάστασης (στο τμήμα δηλώσεων μεταβλητών).
b) Boolean εξισώσεις ελέγχου (στο λογικό τμήμα δηλώσεων).
c) Μεταβάσεις μεταξύ των καταστάσεων (στο λογικό τμήμα δηλώσεων).

3.6.1 Υλοποίηση Μηχανών Κατάστασης

Μία μηχανή κατάστασης μπορεί να σχεδιαστεί δηλώνοντας το όνομα της μηχανής κατάστασης, τις καταστάσεις της και προαιρετικά τα bit της μηχανής κατάστασης που θα χρησιμοποιούνται, με την χρήση μιας **State Machine** δήλωσης στο τμήμα δηλώσεων μεταβλητών.

Το *simple.tdf* αρχείο που δείχνεται στην συνέχεια έχει την ίδια λειτουργία μ' ένα D flipflop (DFF).

```
SUBDESIGN simple
(
   clk, reset, d : INPUT;
   q             : OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s0, s1);
BEGIN
   ss.clk = clk;
   ss.reset = reset;
   CASE ss IS
      WHEN s0 =>
         q = GND;
         IF d THEN
            ss = s1;
         END IF;
      WHEN s1 =>
         q = VCC;
         IF !d THEN
            ss = s0;
         END IF;
   END CASE;
END;
```

Στο *simple.tdf* αρχείο, μία μηχανή κατάστασης με το όνομα `ss` δηλώνεται στο τμήμα δηλώσεων μεταβλητών. Οι καταστάσεις της μηχανής ορίζονται σαν `s0` και `s1` και δεν ορίζονται bit κατάστασης.

3.6.2 Ενεργοποίηση των σημάτων Clock, Reset & Enable

Τα σήματα clock, reset & enable ελέγχουν τα flipflop του καταχωρητή κατάστασης στην μηχανή κατάστασης. Τα σήματα αυτά καθορίζονται με την χρήση Boolean εξισώσεων στο λογικό τμήμα δηλώσεων. Στο *simple1.tdf* αρχείο που δείχνεται στην συνέχεια, το ρολόι της μηχανής κατάστασης οδηγείται από την είσοδο `clk`. Το ασύγχρονο reset σήμα οδηγείται από την είσοδο `reset`, που είναι ενεργή σε high. Η εξίσωση (`ss.ena = ena;`) στο λογικό τμήμα δηλώσεων συνδέει την `ena` είσοδο των flipflop κατάστασης με την είσοδο `ena`.

```
SUBDESIGN simple
(
   clk, reset, ena, d : INPUT;
   q                  : OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s0, s1);
BEGIN
   ss.clk = clk;
   ss.reset = reset;
   ss.ena = ena;
   CASE ss IS
      WHEN s0 =>
         q = GND;
         IF d THEN
            ss = s1;
         END IF;
      WHEN s1 =>
         q = VCC;

         IF !d THEN
            ss = s0;
         END IF;
   END CASE;
END;
```

3.6.3 Ανάθεση των τιμών και bit της μηχανής κατάστασης

Ένα bit κατάστασης είναι μία έξοδος ενός flipflop που χρησιμοποιείται από την μηχανή κατάστασης για να αποθηκεύσει ένα bit της τιμής της μηχανής κατάστασης. Στις περισσότερες περιπτώσεις, είναι καλύτερο η διαδικασία αυτή της ανάθεσης των bit και τιμών κατάστασης να γίνεται από τον συμβολομεταφραστή του MAX + PLUSII έτσι ώστε να ελαχιστοποιείται η λογική που χρησιμοποιείται. Ωστόσο, οι αναθέσεις αυτές μπορούν να γίνουν και κατά την δήλωση της μηχανής κατάστασης από τον χρήστη, εάν συγκεκριμένα bit επιθυμούνται να είναι έξοδοι της μηχανής κατάστασης. Το stepper.tdf αρχείο που δείχνεται στην συνέχεια, υλοποιεί ένα βηματικό ελεγκτή κινητήρα.

```
SUBDESIGN stepper
(
   clk, reset   : INPUT;
   ccw, cw      : INPUT;
   phase[3..0]  : OUTPUT;
)
VARIABLE
   ss: MACHINE OF BITS (phase[3..0])
      WITH STATES  (
         s0 = B"0001",
         s1 = B"0010",
         s2 = B"0100",
         s3 = B"1000");
BEGIN
   ss.clk   = clk;
   ss.reset = reset;
   TABLE
      ss,   ccw,   cw   =>   ss;
      s0,   1,      x   =>   s3;
      s0,   x,      1   =>   s1;
      s1,   1,      x   =>   s0;
      s1,   x,      1   =>   s2;
      s2,   1,      x   =>   s1;
      s2,   x,      1   =>   s3;
      s3,   1,      x   =>   s2;
      s3,   x,      1   =>   s0;
   END TABLE;
END;
```

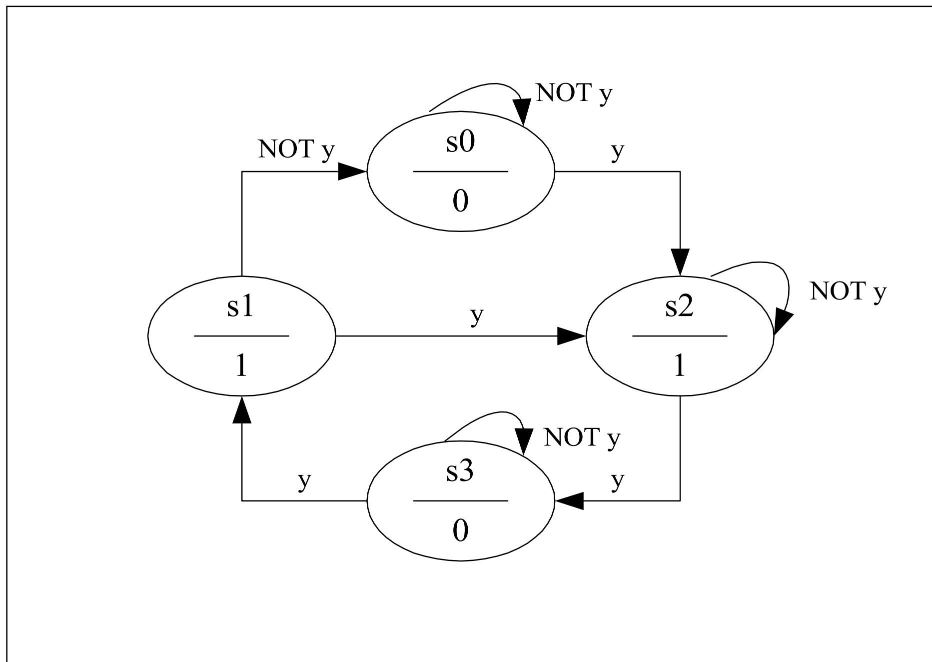
Στο παράδειγμα αυτό, οι phase[3..0]έξοδοι δηλώνονται επίσης και σαν bit της μηχανής κατάστασης ss στο τμήμα δηλώσεων των μεταβλητών. Σημειώνεται επίσης , ότι οι είσοδοι ccw, cw δεν πρέπει ποτέ να έχουν ταυτόχρονα την τιμή ‘1’ στον ίδιο πίνακα. Η AHDL θεωρεί ότι κάθε φορά μόνο μία συνθήκη είναι αληθής και επομένως επικαλυπτόμενοι συνδυασμοί από bit μπορούν να προκαλέσουν απρόβλεπτα αποτελέσματα.

3.6.4 Μηχανές Κατάστασης με Σύγχρονες Εισόδους

Εάν οι έξοδοι μιας μηχανής κατάστασης εξαρτώνται μόνο από την κατάσταση στην οποία αυτή βρίσκεται, τότε οι έξοδοι της μπορούν να καθοριστούν στο WITH STATES τμήμα της δήλωσης της μηχανής κατάστασης. Αυτές οι αναθέσεις κάνουν την εισαγωγή της μηχανής κατάστασης λιγότερο επιρρεπή σε σφάλματα και σε μερικές περιπτώσεις δημιουργούν λογική με λιγότερα χρησιμοποιούμενα λογικά κελιά.

Στο επόμενο σχήμα δείχνεται το διάγραμμα καταστάσεων μιας τεσσάρων καταστάσεων Moore μηχανής κατάστασης. Στις Moore μηχανές κατάστασης , η παρούσα κατάσταση εξαρτάται μόνο από τις εισόδους και την προηγούμενη κατάσταση και οι έξοδοι εξαρτώνται μόνο από την προηγούμενη κατάσταση της μηχανής κατάστασης.

Το *moore1.tdf* αρχείο που δείχνεται στην συνέχεια υλοποιεί την λειτουργία που δείχνεται στο παραπάνω διάγραμμα..



Σχήμα 1
Διάγραμμα Κατάστασης μιας Moore μηχανής κατάστασης

```
SUBDESIGN moore1
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
VARIABLE
                % current   current %
                % state     output  %
   ss: MACHINE OF BITS (z)
       WITH STATES (s0    =   0,
                    s1    =   1,
                    s2    =   1,
                    s3    =   0);
BEGIN
   ss.clk   = clk;
   ss.reset = reset;
   TABLE
   %  current  current      next  %
   %  state    input        state %
      ss,      y      =>    ss;
      s0,      0      =>    s0;
      s0,      1      =>    s2;
      s1,      0      =>    s0;
      s1,      1      =>    s2;
      s2,      0      =>    s2;
      s2,      1      =>    s3;
      s3,      0      =>    s3;
      s3,      1      =>    s1;
   END TABLE;
END;
```

Στο παράδειγμα αυτό ορίζονται οι καταστάσεις της μηχανής κατάστασης και οι μεταβάσεις δίνονται στον πίνακα καταστάσεων. Η μηχανή κατάστασης `ss` έχει τέσσερις καταστάσεις αλλά μόνο ένα bit κατάστασης (z). Ο συμβολομεταφραστής του MAX + PLUSII αυτόματα προσθέτει ακόμα ένα bit και κάνει τις απαραίτητες αναθέσεις κατά την σύνθεση ώστε να παραχθεί μια μηχανή κατάστασης τεσσάρων καταστάσεων. Η μηχανή αυτή κατάστασης απαιτεί τουλάχιστον δύο bit.

Όταν οι τιμές των καταστάσεων χρησιμοποιούνται σαν έξοδοι, όπως στο *moore1.tdf* αρχείο, η σχεδίαση μπορεί να χρησιμοποιήσει λιγότερα λογικά κελιά, αλλά τα λογικά κελιά μπορεί να απαιτούν περισσότερη λογική για να οδηγήσουν τις εισόδους των flipflop.

Άλλος τρόπος για την σχεδίαση μηχανών κατάστασης με σύγχρονες εξόδους είναι η παράλειψη των αναθέσεων των τιμών καταστάσεως και η σαφής δήλωση των εξόδων των flipflop. Το αρχείο *moore2.tdf* δείχνει αυτή την εναλλακτική υλοποίηση.

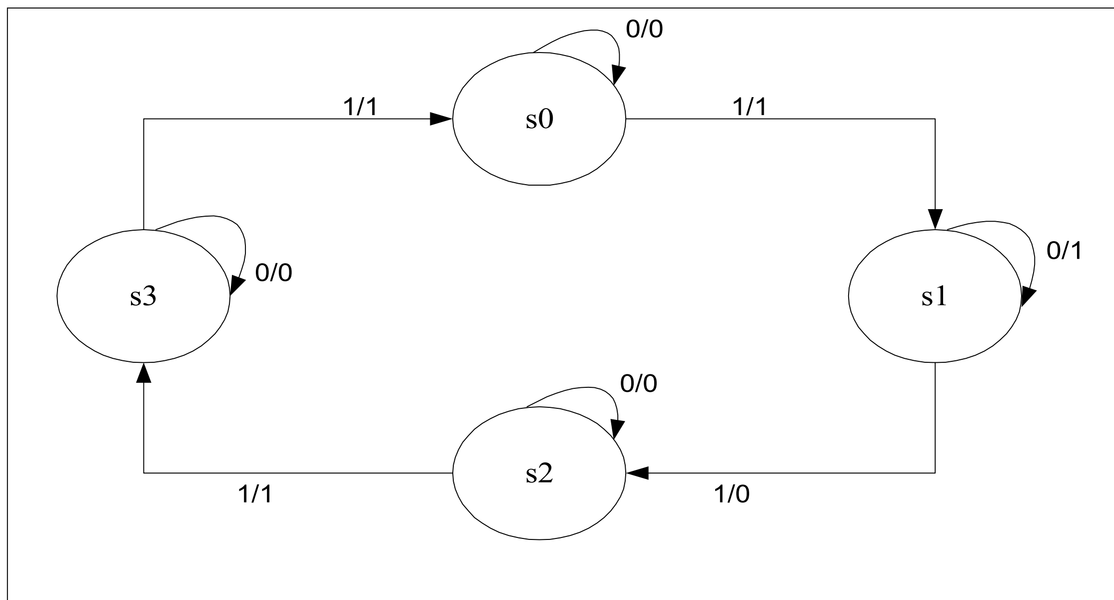
```
SUBDESIGN moore2
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
VARIABLE

   ss: MACHINE WITH STATES (s0, s1, s2, s3);
   zd: NODE;

BEGIN

   ss.clk   = clk;
   ss.reset = reset;
   z = DFF(zd, clk, VCC, VCC);
   TABLE
   %  current  current     next     next   %
   %  state    input       state    output %
      ss,      y      =>   ss,      zd;
      s0,      0      =>   s0,      0;
      s0,      1      =>   s2,      1;
      s1,      0      =>   s0,      0;
      s1,      1      =>   s2,      1;
      s2,      0      =>   s2,      1;
      s2,      1      =>   s3,      0;
      s3,      0      =>   s3,      0;
      s3,      1      =>   s1,      1;
   END TABLE;

END;
```

Αντί να οριστεί η έξοδος με ανάθεση των τιμών κατάστασης στο τμήμα δήλωσης της μηχανής κατάστασης, αυτό το παράδειγμα περιέχει μία `next output` στήλη μετά την `next state` στήλη. Η μέθοδος αυτή χρησιμοποιεί ένα D flipflop (DFF), για τον συγχρονισμό των εξόδων με το ρολόι.

3.6.5 Μηχανές Κατάστασης με Ασύγχρονες Εξόδους

Η AHDL υποστηρίζει την υλοποίηση μηχανών κατάστασης με ασύγχρονες εξόδους. Οι έξοδοι των μηχανών καταστάσεων αυτού του τύπου μπορούν να αλλάζουν οποτεδήποτε οι είσοδοι αλλάζουν, ανεξάρτητα από τις μεταβάσεις του ρολογιού. Στο επόμενο σχήμα δείχνεται μία Mealy μηχανή κατάστασης τεσσάρων καταστάσεων. Στις Mealy μηχανές κατάστασης, οι έξοδοι είναι συνάρτηση των εισόδων και της παρούσας κατάστασης.

Το mealy.tdf αρχείο που δείχνεται στην συνέχεια υλοποιεί την λειτουργία που δείχνεται στο παραπάνω διάγραμμα.



Σχήμα 2
Διάγραμμα Κατάστασης μιας Mealy μηχανής κατάστασης.

```
SUBDESIGN mealy
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3);
BEGIN
   ss.clk = clk;
   ss.reset = reset;
   TABLE
   %  current  current      current  next  %
   %  state    input        output   state %
      ss,      y       =>   z,       ss;
      s0,      0       =>   0,       s0;
      s0,      1       =>   1,       s1;
      s1,      0       =>   1,       s1;
      s1,      1       =>   0,       s2;
      s2,      0       =>   0,       s2;
      s2,      1       =>   1,       s3;
      s3,      0       =>   0,       s3;
      s3,      1       =>   1,       s0;
   END TABLE;
END;
```

3.7 Υλοποίηση ενός Ιεραρχημένου σχεδίου

Τα TDF αρχεία μπορούν να αναμιχθούν με άλλα σχεδιαστικά αρχεία μέσα σε μια ιεραρχημένη δομή. Τα αρχεία που βρίσκονται χαμηλά στην ιεραρχία μπορούν να είναι είτε συναρτήσεις που δίνονται από την Altera είτε συναρτήσεις που ορίζονται από τον χρήστη.

3.7.1 Χρήση των συναρτήσεων της Altera

Μία αναφορά σε συνάρτηση ή σε κάποιο εξάρτημα είναι μια Boolean έκφραση που υλοποιεί την συνάρτηση ή το εξάρτημα. Για παράδειγμα, το ακόλουθο υπόδειγμα συνάρτησης έχει τρεις εισόδους `a[3..0]` και `b[3..0]` και εξόδους `less, equal` και `greater`:

```
FUNCTION compare (a[3..0], b[3..0])
```

```
RETURNS (less, equal, greater);
```

Η αναφορά στην συνάρτηση `compare` δείχνεται στο δεξιό τμήμα της παρακάτω δήλωσης:

```
(clockwise, , counterclockwise) = compare(position[], target[]);
```

Η αναφορά σε μία συνάρτηση έχει τα ακόλουθα χαρακτηριστικά:

a) Το όνομα της συνάρτησης ή του εξαρτήματος ακολουθείται από μία λίστα από σήματα που περικλείονται σε παρενθέσεις. Η λίστα αυτή μπορεί να περιέχει συμβολικά ονόματα, δεκαδικούς αριθμούς ή σύνολα από κόμβους.
b) Οι έξοδοι μιας συνάρτησης συνδέονται σε μεταβλητές σε ανάλογες θέσεις στο αριστερό τμήμα κλήσης της συνάρτησης.
c) Η τιμή των μεταβλητών, που καθορίζεται στο Λογικό τμήμα δηλώσεων, τροφοδοτεί τις εισόδους και εξόδους μιας συνάρτησης.

Το MAX + PLUSII περιέχει μια μεγάλη βιβλιοθήκη από συναρτήσεις που χρησιμοποιούνται για την δημιουργία ενός ιεραρχημένου σχεδίου. Υπάρχουν δύο τρόποι για την κλήση μιας συνάρτησης στην AHDL:

a) Ορισμός μιας μεταβλητής για την συνάρτηση, δηλαδή δήλωση ενός ονόματος στο τμήμα δηλώσεων των μεταβλητών και χρήση καναλιών της συνάρτησης στο λογικό τμήμα δηλώσεων. Με τη μέθοδο αυτή, τα ονόματα των καναλιών διασύνδεσης είναι σημαντικά ανεξάρτητα από την σειρά τους.
b) Αναφορά της συνάρτησης στο λογικό τμήμα δηλώσεων του TDF. Με την μέθοδο αυτή, η σειρά των καναλιών διασύνδεσης είναι σημαντική ανεξάρτητα από τα ονόματά τους.

Οι είσοδοι και οι έξοδοι της συνάρτησης δείχνονται με μία *Function Prototype* δήλωση. Μία τέτοια λίστα μπορεί επίσης να σωθεί σ' ένα include αρχείο και να εισαχθεί σ' ένα TDF αρχείο με μία *Include* δήλωση (ένα include αρχείο μπορεί να δημιουργηθεί αυτόματα με την *Create Include File* εντολή). Το macro1.tdf αρχείο που δείχνεται στην συνέχεια υλοποιεί ένα μετρητή των 4-bit που συνδέεται σ' ένα 4-bit-binary-to-16-line αποκωδικοποιητή.

```
INCLUDE "4count";
INCLUDE "16dmux";
SUBDESIGN macro1
(
   clk        : INPUT;
   out[15..0] : OUTPUT;
)
VARIABLE
   counter : 4count;
   decoder : 16dmux;
BEGIN
   counter.clk = clk;
   counter.dnup = GND;
   decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
   out[15..0] = decoder.q[15..0];
END;
```

Το αρχείο αυτό χρησιμοποιεί *Include* δηλώσεις για να εισάγει τις σχεδιασμένες από την Altera συναρτήσεις `4count` και `16dmux`. Στο τμήμα δηλώσεων μεταβλητών, η μεταβλητή `counter` δηλώνεται σαν ένα στοιχείο της 4count συνάρτησης και η μεταβλητή `decoder` δηλώνεται σαν στοιχείο της `16dmux` συνάρτησης. Τα κανάλια εισόδου και για τις δύο συναρτήσεις δηλώνονται στο αριστερό τμήμα των Boolean εξισώσεων, ενώ οι έξοδοι ορίζονται στο δεξιό τμήμα.

Το αρχείο *macro2.tdf* έχει την ίδια λειτουργία όπως το *macro1.tdf* μόνο που υλοποιεί τις συναρτήσεις με απευθείας αναφορά.

```
INCLUDE "4count";

INCLUDE "16dmux";

SUBDESIGN macro2
```

```
(
   clk          : INPUT;
   out[15..0]   : OUTPUT;
)
VARIABLE
   q[3..0]      : NODE;
BEGIN
   (q[3..0], ) = 4count (clk, , , , , GND, , , , );
   out[15..0]  = 16dmux (.(d, c, b, a)=q[3..0]);
END;
```

Τα πρωτότυπα των συναρτήσεων που βρίσκονται στα *4count.inc* και *16mux.inc* αρχεία δείχνονται στην συνέχεια:

```
FUNCTION 4count (clk, clrn, setn, ldn, cin, dnup, d, c, b, a)
   RETURNS (qd, qc, qb, qa, cout);

FUNCTION 16dmux (d, c, b, a)
   RETURNS (q[15..0]);
```

Τα κανάλια εισόδου, εξόδου των συναρτήσεων συνδέονται στο λογικό τμήμα δηλώσεων όπως δείχνεται στο *macro2.tdf* αρχείο. Η σειρά δήλωσης των καναλιών είναι σημαντική επειδή υπάρχει μία ένα προς ένα αντιστοίχηση μεταξύ της σειράς των καναλιών όπως δηλώνονται στο πρωτότυπο της συνάρτησης και των καναλιών όπως δηλώνονται στο λογικό τμήμα.

3.7.2 Εισαγωγή και Εξαγωγή Μηχανών Κατάστασης

Μπορεί να γίνει εισαγωγή και εξαγωγή μηχανών κατάστασης μεταξύ TDF και άλλων σχεδιαστικών αρχείων ορίζοντας μία είσοδο ή έξοδο σαν **MACHINE INPUT** ή **MACHINE OUTPUT**. Το πρωτότυπο της συνάρτησης που αναπαριστάνει την μηχανή κατάστασης πρέπει να δείχνει ποιες από τις εισόδους και τις εξόδους είναι από την μηχανή κατάστασης θέτοντας στο όνομα του σήματος το πρόθεμα **MACHINE**.

Το *ss_def.tdf* αρχείο που δείχνεται στην συνέχεια, ορίζει και εξάγει την μηχανή κατάστασης `ss` με την έξοδο από μηχανή κατάστασης `ss_out`.

```
SUBDESIGN ss_def
(
   clk, reset, count : INPUT;
   ss_out            : MACHINE OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s1, s2, s3, s4, s5);
BEGIN

 ss_out = ss;
   CASE ss IS
      WHEN s1=>
         IF count THEN ss = s2; ELSE ss = s1; END IF;
      WHEN s2=>
         IF count THEN ss = s3; ELSE ss = s2; END IF;
      WHEN s3=>
         IF count THEN ss = s4; ELSE ss = s3; END IF;
      WHEN s4=>
         IF count THEN ss = s5; ELSE ss = s4; END IF;
      WHEN s5=>
         IF count THEN ss = s1; ELSE ss = s5; END IF;
   END CASE;
   ss.(clk, reset) = (clk, reset);

END;
```

Το *ss_use.tdf* αρχείο που δείχνεται στην συνέχεια εισάγει μία μηχανή κατάστασης με την είσοδο από μηχανή κατάστασης `ss_in`.

```
SUBDESIGN ss_use
```

```
(
   ss_in : MACHINE INPUT;
   out   : OUTPUT;
)
BEGIN
   out = (ss_in == s2) OR (ss_in == s4);
END;
```

Το *top1.tdf* αρχείο, εισάγει στοιχεία από τις συναρτήσεις `ss_def` και `ss_use`.

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);

FUNCTION ss_use (MACHINE ss_in) RETURNS (out);

SUBDESIGN top1
(
      sys_clk, /reset, hold : INPUT;
      sync_out              : OUTPUT;
)
VARIABLE
      ss_ref: MACHINE;  % Machine Alias Declaration %
BEGIN
      ss_ref = ss_def(sys_clk, !/reset, !hold);
      sync_out = ss_use(ss_ref);
END;
```

Το *top2.tdf* αρχείο που δείχνεται στην συνέχεια έχει την ίδια λειτουργία με το *top1.tdf* αρχείο, μόνο που χρησιμοποιεί δηλώσεις των εξαρτημάτων για την κλήση των συναρτήσεων.

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);

SUBDESIGN top2
(
      sys_clk, /reset, hold : INPUT;
      sync_out              : OUTPUT;
)
VARIABLE
      sm_macro : ss_def;
      sync     : ss_use;
BEGIN
   sm_macro.(clk, reset, count) = (sys_clk, !/reset, !hold);
   sync.ss_in = sm_macro.ss_out;
   sync_out = sync.out;
END;
```

3.8 Έλεγχος της Λογικής Σύνθεσης

Η AHDL δίνει ένα μερικό έλεγχο στον τρόπο που γίνεται η σύνθεση ενός κυκλώματος. Στο τμήμα αυτό περιγράφονται τα στοιχεία της AHDL και οι δηλώσεις, που επιτρέπουν στον σχεδιαστή να καθορίσει την διεξαγωγή της λογικής σύνθεσης.

3.8.1 Υλοποίηση των LCELL & SOFT πρωτογενών στοιχείων

Μπορεί να περιοριστεί η έκταση της λογικής σύνθεσης αλλάζοντας τις μεταβλητές τύπου **NODE** με **SOFT** και **LCELL** εξαρτήματα. Οι **NODE** μεταβλητές και τα **LCELL** πρωτογενή εξαρτήματα παρέχουν τον μεγαλύτερο έλεγχο κατά την διάρκεια της λογικής σύνθεσης. Τα **SOFT** πρωτογενή εξαρτήματα δίνουν λιγότερο έλεγχο κατά την λογική σύνθεση.

Οι NODE μεταβλητές, θέτουν πολύ λίγους περιορισμούς στην λογική σύνθεση. Κατά την διάρκεια της σύνθεσης, ο Λογικός Συνθέτης αντικαθιστά κάθε εξάρτημα μιας NODE μεταβλητής με την λογική που η μεταβλητή που αναπαριστά. Στην συνέχεια ελαχιστοποιείται η λογική έτσι ώστε να ταιριάζει σ' ένα λογικό κελί. Αυτή η μέθοδος συνήθως οδηγεί σε υψηλές ταχύτητες, αλλά μπορεί να οδηγήσει σε λογική που είναι αρκετά πολύπλοκη ή δύσκολη για να προσαρμοστεί σε κάποια συσκευή.

Οι SOFT απομονωτές παρέχουν μεγαλύτερο έλεγχο των πόρων της συσκευής από τις NODE μεταβλητές. Ο Λογικός Συνθέτης επιλέγει πότε θα αντικαταστήσει πρωτογενή SOFT εξαρτήματα με LCELL εξαρτήματα. Οι SOFT απομονωτές μπορούν να ελαχιστοποιήσουν λογική που είναι αρκετά σύνθετη, αλλά μπορούν να αυξήσουν την χρησιμοποίηση των λογικών κελιών και να μειώσουν την ταχύτητα απόκρισης.

Τα LCELL πρωτογενή εξαρτήματα παρέχουν τον μεγαλύτερο έλεγχο. Ο Λογικός Συνθέτης ελαχιστοποιεί όλη την λογική που οδηγεί ένα LCELL εξάρτημα έτσι ώστε η λογική να προσαρμόζεται σ' ένα μοναδικό λογικό κελί. Τα LCELL εξαρτήματα υλοποιούνται πάντα σ' ένα λογικό κελί και δεν απομακρύνονται ποτέ ακόμα και αν οδηγούνται από μία μόνο είσοδο.

Ο συμβολομεταφραστής του MAX + PLUSII αυτόματα εισάγει SOFT απομονωτές σε πλεονεκτικές θέσεις σ' ένα κύκλωμα εάν η *SOFT Buffer Insertion* λογική επιλογή έχει ενεργοποιηθεί. Στην συνέχεια δείχνονται δύο εκδοχές ενός TDF αρχείου: η μία είναι υλοποιημένη με NODE μεταβλητές και η άλλη με SOFT εξαρτήματα. Στο `nodevar` αρχείο, η μεταβλητή `odd_parity` δηλώνεται σαν NODE μεταβλητή και στην συνέχεια της ανατίθεται η τιμή της Boolean έκφρασης. Στο `softbuf` αρχείο, ο συμβολομεταφραστής θα αντικαταστήσει μερικά από τα *SOFT* εξαρτήματα με *LCELL* εξαρτήματα έτσι ώστε να βελτιωθεί το ποσοστό κάλυψης μιας συσκευής.

TDF με NODE μεταβλητές:

```
SUBDESIGN nodevar
(
)
VARIABLE
      odd_parity : NODE;
BEGIN
odd_parity = d0 $ d1 $ d2
             $ d3 $ d4 $ d5
             $ d6 $ d7 $ d8;
END;
```

TDF με SOFT εξαρτήματα:

```
SUBDESIGN softbuf
(
)
VARIABLE
      odd_parity : NODE;
BEGIN
      odd_parity = SOFT(d0 $ d1 $ d2)
            $ SOFT(d3 $ d4 $ d5)
            $ SOFT(d6 $ d7 $ d8);
END;
```

3.9 Υλοποίηση RAM & ROM Μνημών

Το MAX + PLUSII και η AHDL δίνουν πολλές συναρτήσεις που επιτρέπουν την υλοποίηση RAM και ROM μνημών. Οι συναρτήσεις που χρησιμοποιούνται για την υλοποίησή τους είναι:

Όνομα:	**Περιγραφή:**
lpm_ram_dq	Σύγχρονη ή ασύγχρονη μνήμη με ξεχωριστές εισόδους, εξόδους
lpm_ram_io	Σύγχρονη ή ασύγχρονη μνήμη με ένα μοναδικό I/O ακροδέκτη
lpm_rom	Σύγχρονη ή ασύγχρονη μνήμη, μόνο για διάβασμα
csdpram	Μνήμη διπλής κατεύθυνσης
csfifo	FIFO απομονωτής

Στις συναρτήσεις αυτές, παράμετροι χρησιμοποιούνται για τον καθορισμό των μηκών των δεδομένων εισόδου και εξόδου.

3.10 Χρήση Επαναλαμβανόμενης Λογικής

Όταν επιθυμείται η χρήση πολλαπλών block που είναι ίδια μεταξύ τους τότε χρησιμοποιείται η **For Generate** δήλωση για να δημιουργηθεί επαναλαμβανόμενη λογική που βασίζεται στην τιμή μιας μεταβλητής. Το *iter_add.tdf* αρχείο που δείχνεται στην συνέχεια, δείχνει την υλοποίηση μιας τέτοιας λογικής.

```
CONSTANT NUM_OF_ADDERS = 8;
SUBDESIGN iter_add
(
   a[NUM_OF_ADDERS..1], b[NUM_OF_ADDERS..1], cin : INPUT;
   c[NUM_OF_ADDERS..1], cout                     : OUTPUT;
)

VARIABLE
   sum[NUM_OF_ADDERS..1], carryout[(NUM_OF_ADDERS+1)..1] : NODE;
BEGIN
 carryout[1] = cin;
```

```
   FOR i IN 1 TO NUM_OF_ADDERS GENERATE
      sum[i] = a[i] $ b[i] $ carryout[i];   % Full Adder %
      carryout[i+1] = a[i] & b[i] # carryout[i] & (a[i] $ b[i]);
   END GENERATE;
   cout = carryout[NUM_OF_ADDERS+1];
   c[] = sum[];
END;
```

Στο *iter_add.tdf* αρχείο η `For Generate` δήλωση χρησιμοποιείται για την υλοποίηση πλήρων αθροιστών, που ο καθένας εκτελεί ενός bit πρόσθεση.

3.11 Χρήση Επαναλαμβανόμενης Λογικής υπό συνθήκη

Μπορεί να δημιουργηθεί επαναλαμβανόμενη λογική υπό συνθήκη με την χρήση **If Generate** δηλώσεων, εάν για παράδειγμα επιθυμείται η υλοποίηση διαφορετικών λειτουργιών σύμφωνα με την τιμή μιας αριθμητικής έκφρασης. Το *conlog1.tdf* αρχείο που δείχνεται στην συνέχεια, χρησιμοποιεί μία `If Generate` δήλωση για να υλοποιήσει διαφορετική συμπεριφορά στην έξοδο `output_b` βάση της λογικής οικογένειας που χρησιμοποιείται.

```
PARAMETERS (DEVICE_FAMILY);
SUBDESIGN condlog1
(
   input_a : INPUT;
   output_b : OUTPUT;
)
BEGIN
   IF DEVICE_FAMILY == "FLEX8K" GENERATE
      output_b = input_a;
   ELSE GENERATE
      output_b = LCELL(input_a);
   END GENERATE;
END;
```

Η `If Generate` δήλωση είναι ιδιαίτερα χρήσιμη με `For Generate` δηλώσεις που χειρίζονται ειδικές περιπτώσεις με διαφορετικό τρόπο.

3.12 Αφιερωμένες Λέξεις

Αφιερωμένες λέξεις χρησιμοποιούνται κατά την έναρξη και την λήξη AHDL δηλώσεων. Υπάρχουν επίσης, και αφιερωμένα πρωτογενή στοιχεία που είναι ουσιαστικά, ονόματα για ειδικές χρήσεις στην AHDL και δεν μπορούν να οριστούν από τον χρήστη. Τέτοια πρωτογενή στοιχεία περιλαμβάνουν απομονωτές, flipflop, μανδαλωτές, κ.ο.κ.

Στην παρακάτω λίστα δείχνονται οι αφιερωμένες λέξεις στην AHDL:

```
AND               FUNCTION        OUTPUT
ASSERT            GENERATE        PARAMETERS
BEGIN             GND             REPORT
BIDIR             HELP_ID         RETURNS
BITS              IF              SEGMENTS
BURIED            INCLUDE         SEVERITY
CASE              INPUT           STATES
CLIQUE            IS              SUBDESIGN
CONNECTED_PINS    LOG2            TABLE
CONSTANT          MACHINE         THEN
DEFAULTS          MOD             TITLE
DEFINE            NAND            TO
DESIGN            NODE            TRI_STATE_NODE
DEVICE            NOR             VARIABLE
DIV               NOT             VCC
ELSE              OF              WHEN
ELSIF             OPTIONS         WITH
END               OR              XNOR
FOR               OTHERS          XOR
```

Στην παρακάτω λίστα δείχνονται τα αφιερωμένα πρωτογενή στοιχεία στην AHDL:

```
CARRY        JKFFE      SRFFE
CASCADE      JKFF       SRFF
CEIL         LATCH      TFFE
DFFE         LCELL      TFF
DFF          MCELL      TRI
EXP          MEMORY     USED
FLOOR        OPENDRN    WIRE
GLOBAL       SOFT       X
```

3.13 Ονόματα

Τρεις τύποι ονομάτων υπάρχουν στην AHDL:

a) Συμβολικά ονόματα που χρησιμοποιούνται για να ονομάσουν τα ακόλουθα τμήματα σ' ένα TDF αρχείο:

- **i)** Εσωτερικοί και εξωτερικοί κόμβοι.
- **ii)** Σταθερές.
- **iii)** Μεταβλητές μηχανών κατάστασης, ονόματα καταστάσεων.
- **iv)** Εξαρτήματα.

b) Ονόματα υποσχεδίων, που δηλώνονται από τον χρήστη και πρέπει να είναι τα ίδια με τα ονόματα των TDF αρχείων.

c) Ονόματα ακροδεκτών που διαχωρίζουν την είσοδο ή έξοδο τόσο πρωτογενών στοιχείων όσο και συναρτήσεων.

Οι επιτρεπόμενοι χαρακτήρες σ' ένα όνομα μπορούν να είναι οι: A-Z, a-z, 0-9, _ , /, -.

3.14 Λογικοί Τελεστές

Οι λογικοί τελεστές για Boolean εκφράσεις δείχνονται στον παρακάτω πίνακα:

Τελεστής	Παράδειγμα	Περιγραφή
! (NOT)	!tob (NOT tob)	Αντιστροφή
& (AND)	bread & butter (bread AND butter)	AND
!& (NAND)	a[3..1] !& b[5..3] (a[3..1] NAND b[5..3])	NAND
# (OR)	trick # treat (trick OR treat)	OR
!# (NOR)	c[8..5] !# d[7..4] (c[8..5] NOR d[7..4])	NOR
$ (XOR)	foo $ bar (foo XOR bar)	Αποκλειστικό OR
!$ (XNOR)	x2 !$ x4(x2 XNOR x4)	Αποκλειστικό XOR

Πίνακας 1
Λογικοί Τελεστές

Κάθε τελεστής αναπαριστάνει μία δύο εισόδων λογική πύλη, εκτός από τον `NOT` τελεστή που έχει μόνο μία είσοδο. Για να χρησιμοποιηθεί ο τελεστής αρκεί να δηλωθεί το όνομά του ή το σύμβολό του.

3.14.1 Εκφράσεις που χρησιμοποιούν τον NOT τελεστή

Η συμπεριφορά του `NOT` τελεστή εξαρτάται από τον τελεστέο στον οποίο εφαρμόζεται. Τρεις τύποι τελεστέων μπορούν να χρησιμοποιηθούν με τον `NOT` τελεστή:

a) Εάν ο τελεστής είναι ένας μοναδικός κόμβος, `GND` ή `VCC` πραγματοποιείται η λειτουργία της αναστροφής. Για παράδειγμα, `!a` σημαίνει ότι το σήμα `a` περνά μέσα από ένα αναστροφέα.

b) Εάν ο τελεστέος είναι ένα σύνολο από κόμβους, κάθε μέλος του συνόλου περνά από ένα αναστροφέα. Για παράδειγμα, το σύνολο `!a[4..1]` ερμηνεύεται σαν `(!a4, !a3, !a2, !a1 )`.

c) Εάν ο τελεστέος είναι ένας αριθμός, τότε αντιμετωπίζεται σαν ένας δυαδικός αριθμός με τόσο bits, όσα ορίζονται από το σύνολο στο οποίο χρησιμοποιείται και κάθε bit αντιστρέφεται. Για παράδειγμα, `!9` σε ένα σύνολο των τεσσάρων μελών ερμηνεύεται σαν `!B"1001"`, που είναι το ίδιο με το `!B"0110"`.

3.14.2 Εκφράσεις με τους τελεστές AND, NAND, OR, NOR, XOR, & XNOR

Τέσσερις συνδυασμοί τελεστέων υπάρχουν για τους δυαδικούς τελεστές και καθένας από αυτούς ερμηνεύεται διαφορετικά.

a) Εάν όλοι οι τελεστέοι είναι απλοί κόμβοι ή `GND, VCC`, ο τελεστής εκτελεί την λογική λειτουργία στα δύο στοιχεία (παράδειγμα : `a & b`).

b) Εάν όλοι οι τελεστέοι είναι σύνολα από κόμβους, ο τελεστής δρα στους αντίστοιχους κόμβους κάθε συνόλου (παράδειγμα : `( a, b, c ) # ( d, e, f ) = ( a#d, b#e , c#f )`).

c) Εάν ο ένας τελεστέος είναι απλός κόμβος ή `GND, VCC`, και ο άλλος είναι ένα σύνολο από κόμβους, τότε ο κόμβος ή η σταθερά αντιγράφεται έτσι ώστε να σχηματίσει ένα σύνολο που θα έχει τόσα μέλη όσα και το άλλο σύνολο. Στην συνέχεια η έκφραση αντιμετωπίζεται σαν μία πράξη μεταξύ συνόλων (παράδειγμα : `( a &( b[4..1] )= α&b4, a&b3,a&b2, a&b1 )`).

d) Εάν και οι δύο τελεστέοι είναι αριθμοί, ο μικρότερος αριθμός επεκτείνεται έτσι ώστε να έχει το μήκος του άλλου αριθμού. Η έκφραση στην συνέχεια, αντιμετωπίζεται σαν μία πράξη μεταξύ συνόλων.

e) Εάν ο ένας τελεστέος είναι αριθμός και ο άλλος είναι ένα σύνολο από κόμβους, τότε ο αριθμός αποκόπτεται ή επεκτείνεται έτσι ώστε να αποκτήσει το μήκος του συνόλου. Στην συνέχεια η έκφραση αντιμετωπίζεται σαν μία πράξη μεταξύ συνόλων.

3.15 Αριθμητικοί Τελεστές

Οι αριθμητικοί τελεστές χρησιμοποιούνται για την εκτέλεση πρόσθεσης και αφαίρεσης σε σύνολα από κόμβους και αριθμούς. Στον παρακάτω πίνακα δείχνονται οι αριθμητικοί τελεστές στην AHDL.

Τελεστής	Παράδειγμα	Περιγραφή
+(δυαδικό)	+1	θετικό πρόσημο
-(δυαδικό)	-a[4..1]	αρνητικό πρόσημο
+	count[7..0] + delta[7..0]	πρόσθεση
-	rightmost_x[ ] - leftmost_x[ ]	αφαίρεση

Πίνακας 2
Αριθμητικοί Τελεστές

Το (+) και (-) πρόσημο δεν επηρεάζουν τον τελεστέο αλλά χρησιμοποιούνται για την δήλωση θετικών και αρνητικών ποσοτήτων. Οι παρακάτω κανόνες εφαρμόζονται στους άλλους αριθμητικούς τελεστές:

a) Οι πράξεις εκτελούνται μεταξύ δύο τελεστέων, που πρέπει να είναι σύνολα από κόμβους ή αριθμούς.

b) Εάν και οι δύο τελεστέοι είναι σύνολα από κόμβους, τα σύνολα αυτά πρέπει να είναι του ίδιου μεγέθους.

c) Εάν και οι δύο τελεστέοι είναι αριθμοί, ο μικρότερος αριθμός επεκτείνεται έτσι ώστε να έχει το ίδιο μέγεθος με τον άλλο τελεστή.

d) Εάν ο ένας τελεστής είναι αριθμός και ο άλλος ένα σύνολο από κόμβους, ο αριθμός αποκόπτεται ή επεκτείνεται έτσι ώστε να αποκτήσει το μήκος του συνόλου.

3.16 Συγκριτές

Δύο τύποι συγκριτών χρησιμοποιούνται για τη σύγκριση απλών κόμβων ή λογικών, αριθμητικών συνόλων. Στον επόμενο πίνακα δείχνονται οι συγκριτές στην AHDL.

Συγκριτής	Τύπος	Παράδειγμα	Περιγραφή
= =	Λογικός	addr[19..4] == B"B800"	Ίσο με
!=	Λογικός	b1 != b3	Άνισο με
<	Αριθμητικός	fame[ ] < power[ ]	Μικρότερο από
<=	Αριθμητικός	money[ ] <= power[ ]	Μικρότερο από ή ίσο με
>	Αριθμητικός	love[ ] > money[ ]	Μεγαλύτερο από
>=	Αριθμητικός	delta[ ] >= 0	Μεγαλύτερο από ή ίσο με

Πίνακας 3
Συγκριτές.

Ο τελεστής (= =) είναι ένας τελεστής ισότητας που χρησιμοποιείται αποκλειστικά σε Boolean εκφράσεις. Οι λογικοί συγκριτές συγκρίνουν απλούς κόμβους, σύνολα από κόμβους και αριθμούς. Εάν σύνολα από κόμβους ή αριθμοί συγκρίνονται, τότε τα σύνολα πρέπει να είναι του ίδιου μεγέθους. Οι αριθμητικοί τελεστές μπορούν να συγκρίνουν μόνο σύμβολα από κόμβους και αριθμούς και τα σύνολα από κόμβους πρέπει να είναι του ίδιου μεγέθους.

3.17 Προτεραιότητες των Boolean τελεστών και συγκριτών

Τελεστέοι που χωρίζονται από λογικούς και αριθμητικούς τελεστές και συγκριτές υπολογίζονται σύμφωνα με τις προτεραιότητες που δίνονται στον παρακάτω πίνακα. Οι πράξεις με ίσες προτεραιότητες υπολογίζονται από αριστερά προς τα δεξιά. Με την χρήση παρενθέσεων ( ) αλλάζει η σειρά υπολογισμού των προτεραιοτήτων.

Προτεραιότητα	Τελεστής/Συγκριτής
1	- (αρνητικό πρόσημο)
1	! (NOT)
2	+ (πρόσθεση)
2	- (αφαίρεση)
3	= = (ίσο με)
3	!= (άνισο με)
3	< (μικρότερο από)
3	<= (μικρότερο από ή ίσο με)
3	> (μεγαλύτερο από)
3	>= (μεγαλύτερο από ή ίσο με)
4	& (AND)
4	!& (NAND)
5	$ (XOR)
5	!$ (XNOR)
6	# (OR)
6	!# (NOR)

Πίνακας 4
Προτεραιότητες των Boolean τελεστών και συγκριτών

3.18 Σύνταξη των Εντολών της AHDL

Στο τμήμα αυτό περιγράφεται με σύντομο τρόπο η σύνταξη των εντολών της AHDL.

a) Η δήλωση σταθεράς έχει την ακόλουθη σύνταξη:

```
CONSTANT <symbolic name> = <expression>;
```

b) Η δήλωση μιας συνάρτησης έχει την ακόλουθη σύνταξη:

```
FUNCTION <subdesign> ( <input list> )
      RETURNS ( <output list> );
      FUNCTION <primitive> ( <input list> )
```

c) Η δήλωση παραμέτρου έχει την ακόλουθη σύνταξη:

```
PARAMETERS ( <parameter list> );
```

d) Η Include δήλωση έχει την ακόλουθη σύνταξη:

```
INCLUDE <filename> ;
```

e) Η δήλωση ενός υποσχεδίου έχει την ακόλουθη σύνταξη:

```
SUBDESIGN <design name>
      (
            <signal list> ;
            { <signal list> ; }
      )
<signal list> ::=
      <port list> : <port type>
<port type> ::=
      INPUT [ = VCC | = GND ]
      |     OUTPUT
      |     BIDIR [ = VCC | = GND ]
   |  MACHINE INPUT
      |     MACHINE OUTPUT
```

f) Το τμήμα δήλωσης μεταβλητών έχει την ακόλουθη σύνταξη:

```
VARIABLE
            <port list> : <variable type> ;
            { <port list> : <variable type> ; }

<variable type> ::=
      NODE
      TRI_STATE_NODE
      |     <subdesign>
      |     <primitive>
      |     <state machine>
      |     <machine alias>
<state machine> ::=
      MACHINE [ OF BITS <bits> ] WITH STATES ( <state> { , <state> } )
<state> ::=
      <symbolic name> [ = <state value> ]

<state value> ::=
      <number>
      |     <symbolic name>
<machine alias> ::=
      <symbolic name> : MACHINE;
```

g) Οι Boolean εξισώσεις έχουν την ακόλουθη σύνταξη:

```
<group> = <rgroup>;
```

h) Οι Boolean εξισώσεις ελέγχου έχουν την ακόλουθη σύνταξη:

```
<symbolic name>.clk = <rgroup>;
[<symbolic name>.reset = <rgroup>;]
[<symbolic name>.ena = <rgroup>;]
```

i) Η Case δήλωση έχει την ακόλουθη σύνταξη:

```
CASE <rgroup> IS
            WHEN <choices> => <statements>
            { WHEN <choices> => <statements> }
            [ WHEN OTHERS => <statements> ]
      END CASE;

<choices> ::=
      <constant group> {  , <constant group> }
<statements> ::=
      <statement>
      { <statement> }
<statement> ::=
      <boolean equation>
      |     <boolean control equation>
      |     <case statement>
      |     <if then statement>
      |     <if generate statement>
      |     <in-line logic function reference>
      |     <for generate statement>
      |     <truth table statement>
```

j) Η If δήλωση έχει την ακόλουθη σύνταξη:

```
IF <rgroup> THEN
            <statements>
      { ELSIF <rgroup> THEN
            <statements> }
      [ ELSE <rgroup> THEN
            <statements> ]
      END IF;

<statements> ::=
      <statement>
      { <statement> }
<statement> ::=
      <boolean equation>
      |     <boolean control equation>
      |     <case statement>
      |     <if then statement>
      |     <if generate statement>
      |     <in-line logic function reference>
      |     <for generate statement>
      |     <truth table statement>
```

k) Η If Generate δήλωση έχει την ακόλουθη σύνταξη:

```
IF <expression> GENERATE
            <statements>
      { ELSE
            <statements> }
      END GENERATE;

<statements> ::=
      <statement>
```

```
      { <statement> }
<statement> ::=
      <boolean equation>
      |     <boolean control equation>
      |     <case statement>
      |     <if then statement>
      |     <if generate statement>
      |     <in-line logic function reference>
      |     <for generate statement>
      |     <truth table statement>
```

l) Η For Generate δήλωση έχει την ακόλουθη σύνταξη:

```
FOR ( <symbolic name> ) IN <expression> TO <expression> GENERATE
            <statements>
      END GENERATE;

<statements> ::=
      <statement>
      { <statement> }
<statement> ::=
      <boolean equation>
      |     <boolean control equation>
      |     <case statement>
      |     <if then statement>
      |     <if generate statement>
      |     <in-line logic function reference>
      |     <for generate statement>
      |     <truth table statement>
```

m) Σε μία Boolean εξίσωση, η αναφορά σε μία συνάρτηση έχει την ακόλουθη σύνταξη:

```
<subdesign> ( <rgroup list> ) [ WITH ( <parameter list> ) ]
```

n) Η δήλωση ενός πίνακα αληθείας έχει την ακόλουθη σύνταξη:

```
TABLE <inputs> => <outputs> ;
            <input values> => <output values> ;
            { <input values> => <output values> ; }
      END TABLE;

<inputs> ::=
      <rgroup> { , <rgroup> }
<outputs> ::=
      <lgroup> { , <lgroup> }
<input values> ::=
      <constant group> { , <constant group> }
<output values> ::=
      <constant group> { , <constant group> }
```

o) Η δήλωση μιας εισόδου ή μιας εξόδου σ' ένα σχεδιασμό, έχει την ακόλουθη σύνταξη:

```
<port name> : <port type> [ = <default port value> ]

<port type> ::=
      INPUT
      |     OUTPUT
      |     BIDIR
      |     MACHINE INPUT
      |     MACHINE OUTPUT

<default port value> ::=
      VCC
      |     GND
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Altera Data Book 1998
[2] Altera MAX + PLUSII AHDL
[3] "Field Programmable Gate Array Technology", Stephen M. Trimberger, Kluwer Academic Publishers, 1994.
[4] "Field Programmable Gate Arrays", Stephen D. Brown, Robert J. Francis, Jonathan Rose, Zvonko G. Vranesic, Kluwer Academic Publishers, 1992.
[5] "Digital Design Using Field Programmable Gate Arrays", Pak K. Chan, Samiha Mourad, Prentice Hall Series in Innovative Technology, 1994.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 1 1

ΕΙΣΑΓΩΓΗ ΣΤΙΣ ΠΡΟΓΡΑΜΜΑΤΙΖΟΜΜΕΝΕΣ ΔΙΑΤΑΞΕΙΣ ΠΥΛΩΝ 1

(FPGAS) 1

1.1 ΕΙΣΑΓΩΓΗ 1

1.2 ΠΛΕΟΝΕΚΤΗΜΑΤΑ ΤΩΝ PLDS ΤΗΣ ALTERA 2

1.2.1ΚΑΛΥΤΕΡΗ ΑΠΟΔΟΣΗ 2
1.2.2 ΜΕΓΑΛΥΤΕΡΗ ΚΛΙΜΑΚΑ ΟΛΟΚΛΗΡΩΣΗΣ 2
1.2.3 ΚΑΛΥΤΕΡΟΣ ΛΟΓΟΣ ΚΟΣΤΟΥΣ/ΑΠΟΤΕΛΕΣΜΑΤΙΚΟΤΗΤΑ 2
1.2.4 ΜΙΚΡΟΤΕΡΟΣ ΚΥΚΛΟΣ ΣΧΕΔΙΑΣΗΣ 2

1.3 ΟΙΚΟΓΕΝΕΙΕΣ ΣΥΣΚΕΥΩΝ ΤΗΣ ALTERA 3

1.4 ΤΟ ΣΧΕΔΙΑΣΤΙΚΟ ΠΑΚΕΤΟ MAX + PLUSII 3

ΚΕΦΑΛΑΙΟ 2 4

Η ΠΡΟΓΡΑΜΜΑΤΙΖΟΜΕΝΗ ΛΟΓΙΚΗ ΟΙΚΟΓΕΝΕΙΑ FLEX 8000 4

2.1 ΕΙΣΑΓΩΓΗ 4

2.2 ΓΕΝΙΚΗ ΠΕΡΙΓΡΑΦΗ 4

2.3 ΛΕΙΤΟΥΡΓΙΚΗ ΠΕΡΙΓΡΑΦΗ 4

2.4 ΔΟΜΗ ΤΟΥ LAB 5

2.5 ΤΑ ΛΟΓΙΚΑ ΣΤΟΙΧΕΙΑ 6

2.5.1 Η CARRY ΑΛΥΣΙΔΑ 7
2.5.2 Η CASCADE ΑΛΥΣΙΔΑ 7

2.6 FASTTRACK INTERCONNECT 8

2.7 ΣΤΟΙΧΕΙΑ ΕΙΣΟΔΟΥ/ΕΞΟΔΟΥ 9

2.7.1 ΔΙΑΣΥΝΔΕΣΕΙΣ ΑΠΟ ΓΡΑΜΜΗ ΣΕ ΣΤΟΙΧΕΙΟ ΕΙΣΟΔΟΥ/ΕΞΟΔΟΥ 9
2.7.2 ΔΙΑΣΥΝΔΕΣΕΙΣ ΑΠΟ ΣΤΗΛΗ ΣΕ ΣΤΟΙΧΕΙΟ ΕΙΣΟΔΟΥ/ΕΞΟΔΟΥ 10

ΚΕΦΑΛΑΙΟ 3 11

ΣΧΕΔΙΑΣΗ ΜΕ ΧΡΗΣΗ ΤΗΣ ΓΛΩΣΣΑΣ AHDL 11

3.1 ΕΙΣΑΓΩΓΗ 11

3.2 ΧΡΗΣΗ ΑΡΙΘΜΩΝ 11

3.3 ΔΗΛΩΣΗ ΣΤΑΘΕΡΩΝ 12

3.4 ΚΥΚΛΩΜΑΤΑ ΣΥΝΔΥΑΣΤΙΚΗΣ ΛΟΓΙΚΗΣ **12**

3.4.1 ΥΛΟΠΟΙΗΣΗ BOOLEAN ΣΥΝΑΡΤΗΣΕΩΝ ΚΑΙ ΕΚΦΡΑΣΕΩΝ 13
3.4.2 ΔΗΛΩΣΕΙΣ ΚΟΜΒΩΝ 13
3.4.3 ΔΗΛΩΣΗ ΣΥΝΟΛΩΝ 13
3.4.4 ΥΛΟΠΟΙΗΣΗ ΛΟΓΙΚΗΣ ΥΠΟ ΣΥΝΘΗΚΗ 14
3.4.4.1 Η δήλωση If *14*
3.4.4.2 Η Δήλωση Case *15*
3.4.5 ΥΛΟΠΟΙΗΣΗ ΔΙΚΑΤΕΥΘΥΝΤΗΡΙΩΝ ΑΚΡΟΔΕΚΤΩΝ 15
3.4.6 ΔΗΛΩΣΗ ΠΙΝΑΚΩΝ ΑΛΗΘΕΙΑΣ 15

3.5 ΑΚΟΛΟΥΘΙΑΚΗ ΛΟΓΙΚΗ **16**

3.5.1 ΔΗΛΩΣΗ ΚΑΤΑΧΩΡΗΤΩΝ 16
3.5.2 ΔΗΛΩΣΗ ΕΞΟΔΩΝ ΚΑΤΑΧΩΡΗΤΩΝ 17
3.5.3 ΜΕΤΡΗΤΕΣ 17

3.6 ΜΗΧΑΝΕΣ ΚΑΤΑΣΤΑΣΗΣ **18**

3.6.1 ΥΛΟΠΟΙΗΣΗ ΜΗΧΑΝΩΝ ΚΑΤΑΣΤΑΣΗΣ 18
3.6.2 ΕΝΕΡΓΟΠΟΙΗΣΗ ΤΩΝ ΣΗΜΑΤΩΝ CLOCK, RESET & ENABLE 19
3.6.3 ΑΝΑΘΕΣΗ ΤΩΝ ΤΙΜΩΝ ΚΑΙ BIT ΤΗΣ ΜΗΧΑΝΗΣ ΚΑΤΑΣΤΑΣΗΣ 20
3.6.4 ΜΗΧΑΝΕΣ ΚΑΤΑΣΤΑΣΗΣ ΜΕ ΣΥΓΧΡΟΝΕΣ ΕΙΣΟΔΟΥΣ 20
3.6.5 ΜΗΧΑΝΕΣ ΚΑΤΑΣΤΑΣΗΣ ΜΕ ΑΣΥΓΧΡΟΝΕΣ ΕΞΟΔΟΥΣ 22

3.7 ΥΛΟΠΟΙΗΣΗ ΕΝΟΣ ΙΕΡΑΡΧΗΜΕΝΟΥ ΣΧΕΔΙΟΥ **23**

3.7.1 ΧΡΗΣΗ ΤΩΝ ΣΥΝΑΡΤΗΣΕΩΝ ΤΗΣ ALTERA 23
3.7.2 ΕΙΣΑΓΩΓΗ ΚΑΙ ΕΞΑΓΩΓΗ ΜΗΧΑΝΩΝ ΚΑΤΑΣΤΑΣΗΣ 25

3.8 ΈΛΕΓΧΟΣ ΤΗΣ ΛΟΓΙΚΗΣ ΣΥΝΘΕΣΗΣ **26**

3.8.1 ΥΛΟΠΟΙΗΣΗ ΤΩΝ LCELL & SOFT ΠΡΩΤΟΓΕΝΩΝ ΣΤΟΙΧΕΙΩΝ 26

3.9 ΥΛΟΠΟΙΗΣΗ RAM & ROM ΜΝΗΜΩΝ **27**

3.10 ΧΡΗΣΗ ΕΠΑΝΑΛΑΜΒΑΝΟΜΕΝΗΣ ΛΟΓΙΚΗΣ **27**

3.11 ΧΡΗΣΗ ΕΠΑΝΑΛΑΜΒΑΝΟΜΕΝΗΣ ΛΟΓΙΚΗΣ ΥΠΟ ΣΥΝΘΗΚΗ **28**

3.12 ΑΦΙΕΡΩΜΕΝΕΣ ΛΕΞΕΙΣ **28**

3.13 ΟΝΟΜΑΤΑ **29**

3.14 ΛΟΓΙΚΟΙ ΤΕΛΕΣΤΕΣ **29**

3.14.1 ΕΚΦΡΑΣΕΙΣ ΠΟΥ ΧΡΗΣΙΜΟΠΟΙΟΥΝ ΤΟΝ NOT ΤΕΛΕΣΤΗ 29
3.14.2 ΕΚΦΡΑΣΕΙΣ ΜΕ ΤΟΥΣ ΤΕΛΕΣΤΕΣ AND, NAND, OR, NOR, XOR, & XNOR 30

3.15 ΑΡΙΘΜΗΤΙΚΟΙ ΤΕΛΕΣΤΕΣ **30**

3.16 ΣΥΓΚΡΙΤΕΣ **31**

3.17 ΠΡΟΤΕΡΑΙΟΤΗΤΕΣ ΤΩΝ BOOLEAN ΤΕΛΕΣΤΩΝ ΚΑΙ ΣΥΓΚΡΙΤΩΝ **31**

3.18 ΣΥΝΤΑΞΗ ΤΩΝ ΕΝΤΟΛΩΝ ΤΗΣ AHDL **31**

ΒΙΒΛΙΟΓΡΑΦΙΑ **35**

ΠΕΡΙΕΧΟΜΕΝΑ **36**