



ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ  
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ  
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ  
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ  
8<sup>Ο</sup> ΕΞΑΜΗΝΟ

ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ ΗΩ ΣΕ ΕΠΙΠΕΔΑ ΧΑΜΗΛΗΣ ΛΟΓΙΚΗΣ II

ΑΝΑΦΟΡΑ ΕΡΓΑΣΙΑΣ

*ΚΩΝΣΤΑΝΤΙΝΟΣ ΒΛΑΧΑΚΟΣ*

*ΑΕΜ: 10403*

## Περιεχόμενα

Εισαγωγή.....	3
Main Module.....	4
Normalization Module .....	5
Rounding Module.....	5
Exception Module .....	6
Testbench.....	8
Assertions .....	10
Επίλογος/Συμπεράσματα.....	11

## Εισαγωγή

Στην παρούσα εργασία επιχειρείται η κατασκευή ενός module που χειρίζεται την floating point αριθμητική, και πιο συγκεκριμένα τον πολλαπλασιασμό 32bit αριθμών βασισμένο στο πρότυπο IEEE-754. Το module αυτό αποτελείται από 3 διαφορετικά τμήματα (normalization/rounding/exception) τα οποία χειρίζονται ξεχωριστά τα βήματα για την ορθή εκτέλεση της πράξης.

Αρχικά γνωρίζουμε από το πρότυπο πως η αναπαράσταση ενός floating point αριθμού, γίνεται μέσω 3 τμημάτων του στο δυαδικό σύστημα. Για την περίπτωση των 32 bit έχουμε: το MSB που ορίζεται ως sign bit, τα επόμενα 8 που αφορούν το exponent ενώ τα υπόλοιπα 23 bits αφορούν το mantissa. Για τον πολλαπλασιασμό δύο αριθμών οφείλουμε να προσθέσουμε (με τη χρήση κατάλληλου bias) τα exponents και να πολλαπλασιάσουμε μεταξύ τους τα mantissa.

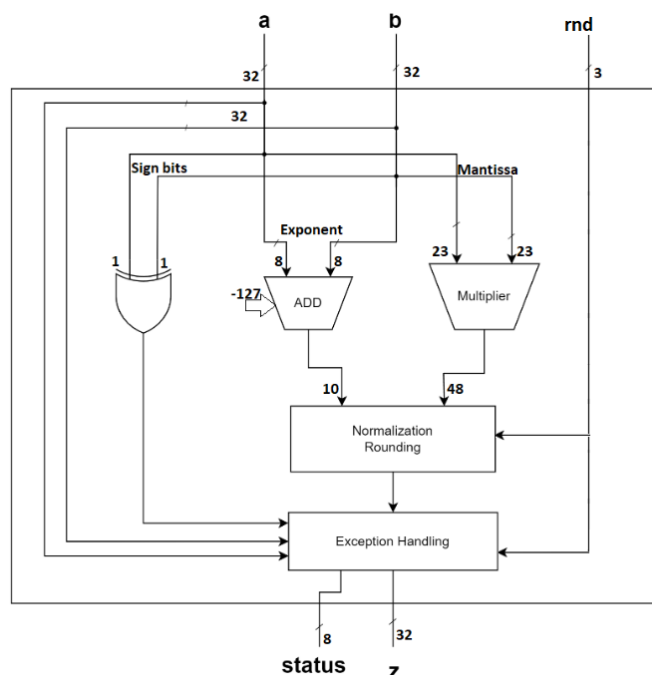
Ωστόσο, γίνεται φανερό το ότι ο πολλαπλασιασμός των mantissa θα οδηγήσει σε έναν αριθμό περισσότερων από 46 bits, ενώ το αποτέλεσμα θα χωράει αντίστοιχα σε μόλις 23. Επιπρόσθετα, η πρόσθεση των exponents ενδέχεται να οδηγήσει σε κάποιον 9 bit αριθμό, όπου σε αυτή την περίπτωση έχουμε overflow.

Μιας και το αποτέλεσμα οφείλει πάντα να τηρεί τη μορφή των εισόδων, τα 3 τμήματα του floating point module, χειρίζονται ακριβώς όλες αυτές τις καταστάσεις με τη μέγιστη δυνατή ακρίβεια και με σήματα ελέγχου και κατάστασης εξόδου τα οποία προσπαθούν να κρατήσουν όσο περισσότερη πληροφορία σχετικά με τις στρογγυλοποιήσεις και την πληροφορία που «χάνεται» από την διαδικασία προσαρμογής του αριθμού αποτελέσματος σε μορφή IEEE-754.

Όσο αφορά τα εργαλεία, επιχειρήθηκε η χρήση του αντίστοιχου open source Icarus Verilog, ωστόσο λόγω των περιορισμών του στο συντακτικό και στην κατανόηση της SystemVerilog, έγινε αλλαγή στο πρόγραμμα Questa, από το οποίο παίρνονται και τα screenshots κυματομορφών.

## Main Module

Στο συγκεκριμένο module επιχειρείται η περιγραφή του σχηματικού που φαίνεται στην εικόνα, και αποτελεί το επίπεδο το οποίο ενώνει όλα τα υπόλοιπα καθώς χειρίζεται τις εισόδους και τις εξόδους, εκτελεί την πράξη του πολλαπλασιασμού και βρίσκει τα βασικά ζητήματα που προκύπτουν από αυτή όπως το overflow, το underflow ή την προσαύξηση του exponent σε περίπτωση που έχει overflow η mantissa.



Έτσι ακολουθώντας και το σχηματικό, μέσα σε ένα procedural block συνδυαστικής λογικής της SystemVerilog (always\_comb) λαμβάνονται από τους 32 bit αριθμούς τα κατάλληλα σήματα logic για τα signs, τα exponents και την mantissa. Έπειτα περιγράφεται η πύλη XOR για το σήμα sign του z καθώς και οι αθροιστές και ο πολλαπλασιαστής για τα τμήματα exponent και mantissa. Για την περίπτωση του exponent, επιθυμούμε να λάβουμε την unbiased τιμή του, επομένως έχουμε  $e1+e2=e3=E1+E2-256=E3-127 \rightarrow E3=E1+E2-127$ , ενώ από τα 8 bits προστίθενται 2 και έχει μήκος 10, ένα bit για την περίπτωση του overflow από την πρώτη άθροιση και ένα bit για την δεύτερη. Το exponent και το mantissa περνούν ως είσοδοι στο normalization module, το οποίο επιστρέφει το normalized exponent και το normalized mantissa τα οποία περνούν ως είσοδοι στο rounding module. Το rounding module από τη μεριά του χειρίζεται τη στρογγυλοποίηση του αριθμού βασισμένο στον κανόνα στρογγυλοποίησης που δίνεται ως είσοδος και μπορεί ή να επιστρέψει την είσοδο ή να προσαυξήσει την είσοδο κατά 1. Ο στρογγυλοποιημένος αριθμός αποτελείται από 25 bits, το πρώτο εξ' αυτών είναι η πιθανή θέση για mantissa overflow, ενώ το επόμενο αφορά το leading one.

Στον αριθμό αυτό γίνεται ο έλεγχος για το mantissa overflow ( $1.mantissa+1=2.0$ ) στο MSB του και στην περίπτωση που υπάρχει overflow, κάνουμε την δεύτερη πρόσθεση στο exponent (+1) και shiftάρουμε το mantissa μια θέση δεξιά ώστε πλέον το leading one να βρίσκεται μια θέση πριν το MSB. Το αποτέλεσμα που προκύπτει και από τη μεριά του exponent και από τη μεριά του mantissa μετά από αυτή τη διαδικασία χρησιμοποιείται για την παραγωγή του z\_calc με απλή σύνθεση του αριθμού από τα LSB των τιμών (SIGN|EXP[7:0]|MANT[22:0]). Επιπλέον, το exponent έπειτα από τον χειρισμό του mantissa overflow χρησιμοποιείται για τη σηματοδότηση του overflow και του underflow (για τους normal αριθμούς), κάνοντας μια

signed σύγκριση μεταξύ του μέγιστου (normal) exponent που μπορεί να λάβει και του ελάχιστου αντίστοιχα. Η σύγκριση γίνεται signed, καθώς στην περίπτωση του underflow από την πράξη  $e1 + e2 - 127$  μπορεί να προκύψει αρνητικός αριθμός, ο οποίος αν δε διαβαστεί ως αρνητικός δεν θα σηματοδοτήσει το γεγονός ότι προέκυψε underflow, αντ' αυτού διαβάζοντας το MSB (sign bit) να θεωρήσει πως πρόκειται για κάποιον μεγάλο αριθμό δίνοντας τα ακριβώς αντίθετα από τα επιθυμητά αποτελέσματα.

Τέλος, δημιουργούνται τα instances από τα 3 επόμενα modules, και η λειτουργικότητα του main module τελειώνει με τη συνδεσμολογία αυτών όπως περιγράφονται από το σχηματικό.

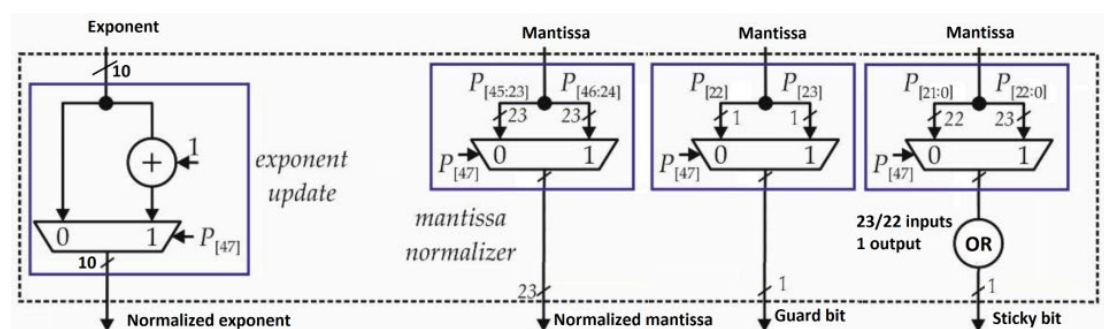
## Normalization Module

Στο συγκεκριμένο module έχουμε για εισόδους το 48 bit αποτέλεσμα του πολλαπλασιασμού των mantissa και το 10 bit αποτέλεσμα της πρόσθεσης και οφείλουμε να εξάγουμε το normalized mantissa καθώς και το normalized exponent.

Για να γίνει αυτό ελέγχουμε το MSB αυτού του 48 bit διανύσματος το οποίο εκφράζει το αποτέλεσμα της πράξης, γνωρίζοντας πως η υποδιαστολή θα βρίσκεται μεταξύ των bits 45 και 46. Στην περίπτωση που, το MSB είναι ίσο με 1, τότε σημαίνει πως το αποτέλεσμα είναι της μορφής 10.M ή 11.M, συνεπώς η υποδιαστολή χρειάζεται να μετακινηθεί μια θέση αριστερά (διαίρεση /2) και για να διατηρηθεί η τιμή του αριθμού ίδια αυξάνουμε κατά 1 τον εκθέτη. Αν είναι 0, τότε είναι στην κανονική του μορφή και δε χρειάζεται κάποια αλλαγή.

Τα bits εξόδου που υπολογίζονται από το μέρος που κόβεται, είναι τα guard και sticky με το guard να είναι το πρώτο bit του μέρους που κόπηκε και το sticky να είναι 1 σε περίπτωση που υπάρχει 1 σε κάποιο από όλα τα υπόλοιπα. Τα δύο αυτά bits, είναι πολύ χρήσιμα στο να βρούμε τη θέση που είμαστε στο rounding ώστε να εφαρμόσουμε σωστά τους κανόνες.

Ο κώδικας ακολουθεί τη συνδυαστική λογική αυτού του σχηματικού, όπως παρουσιάζεται και στην εκφώνηση της εργασίας



## Rounding Module

Στο παρόν module υλοποιούμε την στρογγυλοποίηση η οποία εξαρτάται από τα guard και sticky bits, δηλαδή τους αριθμούς που κόπηκαν. (Αντίστοιχο παράδειγμα στο δεκαδικό 2,723 με δύο δεκαδικά ψηφία  $\rightarrow$  2,720). Για τον χειρισμό της εισόδου του κανόνα rounding (rnd), χρησιμοποιήθηκε ένα enum, στο οποίο κάνουμε typecast το logic input και έπειτα χειριζόμαστε την κάθε περίπτωση μέσω ενός case. Έτσι έχουμε:

- **IEEE\_near**: Στρογγυλοποιούμε στην κοντινότερη δυνατή τιμή, αν βρισκόμαστε ακριβώς στη μέση, στρογγυλοποιούμε προς το άρτιο mantissa (even significant). Το guard bit μας δείχνει το πρώτο στοιχείο που κόπηκε, ενώ το sticky αν υπάρχει κάτι περισσότερο στα υπόλοιπα στοιχεία. Συνεπώς αν το guard bit είναι 0, ο αριθμός βρίσκεται ήδη πιο κοντά στον χαμηλότερο από τους δύο στρογγυλούς οπότε δε

χρειάζεται προσαύξηση. Στην περίπτωση που είναι 1, εξετάζουμε το sticky και αν είναι και αυτό 1 τότε βρίσκεται προς τη μεριά του υψηλότερου, επομένως εκτελούμε την προσαύξηση. Αν το sticky είναι 0 ενώ το guard bit 1, είμαστε στην περίπτωση όπου βρισκόμαστε ακριβώς στη μέση, επομένως θα πρέπει να κοιτάζουμε τον αριθμό που ήδη έχουμε και τον επόμενο. Εφ' όσον θέλουμε προς τον άρτιο, ελέγχουμε το LSB του mantissa και αν είναι 0, τότε δεν εκτελούμε την προσαύξηση καθώς είμαστε ήδη στον άρτιο, ενώ αν είναι 1 την εκτελούμε για να οδηγηθούμε σε 0.

- **IEEE\_zero:** Στρογγυλοποιούμε πάντα με κατεύθυνση προς το 0, δηλαδή προς τον χαμηλότερο. Εφ' όσον τα ψηφία που κόβονται μπορούν μόνο να προσδώσουν τιμή στον αριθμό, για να κρατήσουμε τον χαμηλότερο αρκεί να μην προσαυξάνουμε ποτέ (χωρίς έλεγχο των guard, sticky).
- **IEEE\_pinf:** Στρογγυλοποιούμε προς τα θετικά, δηλαδή εφ' όσον το mantissa μειώνεται από το κόψιμο (inexact), στους θετικούς αριθμούς προσαυξάνουμε την mantissa για να οδηγηθούμε σε κάποιον ακόμα πιο θετικό αριθμό, ενώ στους αρνητικούς την αφήνουμε ίδια καθώς με το αρνητικό πρόσημο η προσαύξηση θα οδηγούσε προς ακόμα πιο αρνητικό.
- **IEEE\_ninf:** Με παρόμοια λογική με το παραπάνω, προσαυξάνουμε εφ' το mantissa μειώνεται από το κόψιμο και το πρόσημο του είναι αρνητικό για να οδηγηθούμε στο πιο αρνητικό αποτέλεσμα.
- **near\_up:** Στρογγυλοποιούμε προς την κοντινότερη δυνατή τιμή, ενώ αν βρισκόμαστε στη μέση, προς τα θετικά. Με παρόμοια λογική με το IEEE\_near η στρογγυλοποίηση γίνεται μόνο εφ' όσον το guard bit είναι 1, ενώ στην περίπτωση που το sticky είναι 0 και βρισκόμαστε ακριβώς στη μέση, αντί να εξετάσουμε το LSB της mantissa που κάνουμε στο IEEE\_near εξετάζουμε το πρόσημο του αριθμού το οποίο απαιτείται να είναι 0 προκειμένου να γίνει η προσαύξηση (θετικός ώστε να δώσει κάτι ακόμα πιο θετικό).
- **away\_zero:** Στρογγυλοποιούμε προς τον πιο μακρινό από το 0, συνεπώς το mantissa θα προσαυξάνεται ανεξάρτητα του προσήμου, εφ' όσον έχει μειωθεί από το κόψιμο (inexact).

Η παρούσα λογική υλοποιείται μέσω του enum, που κάνει mapping την είσοδο rnd στις αντίστοιχες περιπτώσεις και ενός απλού always\_comb block, το οποίο μέσα από ένα case οδηγεί έναν πολυπλέκτη αποτελέσματος και δίνει αντίστοιχα την είσοδο στην έξοδο αυτούσια ή προσαυξημένη κατά ένα, με βάση την εκάστοτε περίπτωση. (Για default τιμή στο case δόθηκε το IEEE\_near)

## Exception Module

Το συγκεκριμένο module είναι υπεύθυνο για τον χειρισμό ακραίων περιπτώσεων, μεγάλων αριθμών που οδηγούν σε overflow, μικρών αριθμών που οδηγούν σε underflow ή περιπτώσεις εισόδων NaN, και INF καθώς και για την οδήγηση των status bits εξόδου. Για την ευκολία στον χειρισμό ορίζουμε το προηγούμενο enum με τα rounding modes, καθώς και ένα δεύτερο με τις όλες τις περιπτώσεις εισόδου όπως ZERO, INF, NORM, MIN\_NORM, MAX\_NORM. Θεωρούμε τα denormals ως μηδενικά και τα NaNs ως άπειρα, επομένως δε χρήζουν επιπλέον ορισμό. Έπειτα υλοποιούνται δύο συναρτήσεις οι οποίες κάνουν τις μεταβάσεις από το enum εισόδου σε 32 bit αριθμούς, ώστε να είναι πιο εύκολη η περιγραφή των καταστάσεων. Οι συναρτήσεις ορίζονται ως automatic καθώς επιθυμούμε να καλούνται συνεχώς με νέες τιμές και να μην αποθηκεύουν τις παλιές (σε αντίθεση με τις static).

Η πρώτη συνάρτηση επιστρέφει αριθμό με βάση το enum και χρειαζόμαστε μόνο τις ακραίες και ορισμένες περιπτώσεις τις οποίες το exception module θα οδηγήσει σε έξοδο, δηλαδή ZERO, INF, MIN\_NORM, MAX\_NORM (σε περίπτωση που έχω NORM στην έξοδο θα

βγάλω την υπολογισμένη τιμή και όχι μια default από τη συνάρτηση). Η δεύτερη συνάρτηση εκτελεί την αντίθετη δουλειά, δηλαδή επιστρέφει το enum type του δοθέντος αριθμού. Στη συγκεκριμένη περίπτωση οφείλουμε να συμπεριλάβουμε και το norm καθώς επιθυμούμε μια ένα προς ένα αντιστοιχία των αριθμών στο enum περιπτώσεων. Έχοντας κάνει όλα αυτά, μέσα σε ένα `always_comb`, αρχικοποιούμε τα status bits τα οποία προτίθενται να αλλάξουν στη συνέχεια του κώδικα, κάνουμε τα απαραίτητα initializations και typecasts των enums που ορίσαμε και προχωράμε στην ανάγνωση των αριθμών a και b από τη συνάρτηση που επιστρέφει τον τύπο σε enum. Μέσω αυτών των τύπων χειριζόμαστε όλα τα ζευγάρια ακραίων περιπτώσεων μέσω ενός case για τον αριθμό a και εμφωλευμένων if για τον b.

Στην περίπτωση που ο a είναι μηδέν (η denormal) το αποτέλεσμα μπορεί να είναι θετικό άπειρο εφ' όσον πολλαπλασιαστεί με άπειρο, ή μηδέν με πρόσημο σε οποιαδήποτε άλλη περίπτωση (με normal ή zero). Έτσι εάν ο b είναι άπειρο (ή NaN), ανοίγουμε το `inf_f` καθώς και το `nan_f` καθώς εάν προέκυψε από denormal ή NaN και τα λαμβάναμε υπόψη, αυτή θα ήταν η μοναδική περίπτωση που θα έδινε NaN στο pre rounded αποτέλεσμα, και δίνουμε στην τιμή του z το +INF. Σε διαφορετική περίπτωση δίνουμε 0 με πρόσημο και ανοίγουμε το `zero_f`. Με ακριβώς τον ίδιο τρόπο χειριζόμαστε την περίπτωση του απείρου όπου αν ο b είναι 0, το αποτέλεσμα θα είναι το +INF με την παραπάνω λογική ενώ αν είναι και αυτός άπειρο επιστρέφουμε άπειρο με πρόσημο, ανοίγοντας σε κάθε περίπτωση και τα αντίστοιχα flags. Τέλος στην περίπτωση του normal, οι πράξεις με 0 ή άπειρο θα δώσουν αντίστοιχα 0 και άπειρο, ενώ μένει προς εξέταση η περίπτωση του normal με normal, όπου εκεί θα χρειαστούν οι κανόνες και οι μέθοδοι του rounding.

Στο normal επί normal, αν έχω overflow (το αποτέλεσμα είναι μεγαλύτερο του MAX\_NORM), δε μπορεί να αναπαρασταθεί με 8 bits exponent, μπορώ να στείλω το αποτέλεσμα είτε MAX\_NORM («rounding προς τα κάτω») είτε INF («rounding προς τα επάνω»). Ανοίγω τα huge και inexact flags και με βάση τα rounding modes έχω:

- IEEE\_near: Στρογγυλοποιώ προς τον αριθμό με το άρτιο mantissa δηλαδή INF
- IEEE\_zero: Στρογγυλοποιώ προς τον πιο κοντά στο 0 αριθμό, δηλαδή MAX\_NORM
- IEEE\_pinf: Στρογγυλοποιώ προς τα θετικά, άρα αν είναι αρνητικός στο MAX\_NORM και αν είναι θετικός στο INF
- IEEE\_ninf: Στρογγυλοποιώ προς τα αρνητικά, άρα αν είναι αρνητικός στο INF και αν είναι θετικός στο MAX\_NORM

(οι δύο αυτές συνθήκες γράφτηκαν μαζί ως ένα else if που στρογγυλοποιεί στο INF αν ταιριάζουν τα πρόσημα με τους κανόνες, ενώ σε περίπτωση που δεν ταιριάζουν πηγαίνει σε ένα else για το MAX\_NORM)

- near\_up: Στρογγυλοποιώ προς τον θετικό αριθμό δηλαδή INF
- away\_zero: Στρογγυλοποιώ προς την κατεύθυνση μακριά από το 0 δηλαδή INF

Αντίστοιχα αν έχω underflow, μπορώ να στείλω στο αποτέλεσμα είτε MIN\_NORM είτε ZERO. Ανοίγω τα tiny και inexact flags και με βάση τα rounding modes έχω:

- IEEE\_near: Στρογγυλοποιώ προς τον αριθμό με το άρτιο mantissa δηλαδή ZERO
- IEEE\_zero: Στρογγυλοποιώ προς τον πιο κοντά στο 0 αριθμό, δηλαδή ZERO
- IEEE\_pinf: Στρογγυλοποιώ προς τα θετικά, άρα αν είναι αρνητικός στο ZERO και αν είναι θετικός στο MIN\_NORM
- IEEE\_ninf: Στρογγυλοποιώ προς τα αρνητικά, άρα αν είναι αρνητικός στο MIN\_NORM και αν είναι θετικός στο ZERO

(οι δύο αυτές συνθήκες γράφτηκαν μαζί ως ένα `else if` που στρογγυλοποιεί στο `MIN_NORM` αν ταιριάζουν τα πρόσημα με τους κανόνες, ενώ σε περίπτωση που δεν ταιριάζουν πηγαίνει σε ένα `else` για το `MAX_NORM`)

- `near_up`: Στρογγυλοποιώ προς τον θετικό αριθμό δηλαδή `ZERO`
- `away_zero`: Στρογγυλοποιώ προς την κατεύθυνση μακριά από το 0 δηλαδή `MIN_NORM`

Τέλος, αν δεν βρίσκομαι σε καμία από αυτές τις περιπτώσεις δίνω στο αποτέλεσμα το ήδη υπολογισμένο από το rounding module και οδηγώ το `inexact flag` ανάλογα με το σήμα `inexact` του rounding.

Καθώς τα overflow και underflow υπολογίζονται στη `main`, είναι εύλογο να αναρωτηθούμε εάν επηρεάζουν τις περιπτώσεις του `zero` και του `inf`, καθώς με βάση τον έλεγχο θα φαίνονται και αυτά ως underflow ή overflow αντίστοιχα. Ωστόσο, μέσα από το exception handling τα σήματα αυτά δε θα επηρεάσουν το αποτέλεσμα καθώς θα οδηγηθούν στα `cases ZERO` και `INF` αντίστοιχα και όχι στο `norm` επί `norm` όπου γίνεται ο έλεγχος των σημάτων.

## Testbench

Οι προδιαγραφές του testbench είναι να δοκιμαστούν random αριθμοί για όλες τις μεθόδους rounding καθώς και όλοι οι συνδυασμοί ακραίων περιπτώσεων (144) για κάθε τύπο rounding. Το παρόν testbench ελέγχει 10 random αριθμούς (9 στην πρώτη περίπτωση με τον πρώτο να είναι το 0 από το reset) για τα 6 rounding modes, και έπειτα αλλάζει αυτόματα είσοδο και προσπελαύνει όλες τις ακραίες περιπτώσεις μέχρι το τέλος. Σε περίπτωση που συναντήσει σφάλμα, σηκώνει ένα σήμα `false` και κάνει `$fatal` την προσομοίωση δίνοντας το expected αποτέλεσμα από την συνάρτηση και το αποτέλεσμα που έλαβε. Μόλις τελειώσει με τα πάντα κάνει αυτόματα `finish`.

Ο τρόπος που γίνεται ο έλεγχος με την τιμή από τη συνάρτηση βασίζεται στο γεγονός ότι το `fp_mult_top` module σερβίρει το αποτέλεσμα δύο κύκλους αργότερα αφού λάβει τις εισόδους `a` και `b` (μπορεί να διαπιστωθεί είτε οπτικά είτε από την ανάλυση της RTL). Επομένως για να γίνει σωστά η σύγκριση, πρέπει το αποτέλεσμα της συνάρτησης που υπολογίζεται στον ίδιο χρόνο να καθυστερήσει 2 κύκλους ρολογιού. Αυτό επιτυγχάνεται μέσω των σημάτων `z_real1` και `z_real2` όπου μέσω `non blocking assignments` λειτουργούν ως `flip flops` ευθυγραμμίζοντας το `z_real2` με το `z` ώστε να μπορεί να γίνει ο έλεγχος.

Η λογική της αυτόματης αλλαγής και της προσπέλασης όλων των αριθμών μονομιάς θα εξηγηθεί παρακάτω, ωστόσο κρίνεται σχετικά πολύπλοκη και ίσως αχρείαστη καθώς θα μπορούσαν να δημιουργηθούν δύο διαφορετικά πολύ απλούστερα testbenches που να επιτυγχάνουν την ίδια δουλειά.

Έτσι λοιπόν, για την υλοποίηση του testbench κατασκευάζουμε ένα `enum` που περιέχει όλες τις ακραίες τιμές (12 στο σύνολο) και μια συνάρτηση που επιστρέφει 32 bit αριθμούς που αντιστοιχούν σε κάθε μια από τις ακραίες τιμές (`pos_inf`, `neg_inf`, `pos_zero`, `neg_zero`, `pos_snan`, `neg_snan`, `pos_qnan`, `neg_qnan`, `pos_norm`, `neg_norm`, `pos_denorm`, `neg_denorm`). Στις περιπτώσεις `norm` δόθηκε ένας σχετικά μεγάλος αριθμός, ωστόσο θα μπορούσε να δοθεί οποιοσδήποτε. Επίσης, χρησιμοποιούμε και το `enum` για τα rounding types, ενώ φτιάχνουμε και έναν `unpacked array` τύπου `string` με τα strings όλων των rounding modes, τα οποία θα δοθούν στη συνάρτηση.



Για τον διαχωρισμό των περιπτώσεων χρησιμοποιούνται 3 σήματα εισόδων για κάθε a και b και ένας πολυπλέκτης που επιλέγει αυτό που θα οδηγήσει το dut. Για την ανάθεση τιμών στα a1 και b1 τα οποία αφορούν τους τυχαίους αριθμούς χρησιμοποιείται η συνάρτηση \$urandom(). Αντίθετα, για την ανάθεση τιμών στα a2 και b2 που περιλαμβάνουν ακραίες τιμές, δημιουργείται ένας πίνακας 12x12 64 bit σε κάθε κελί του και μέσω μιας nested for loop σε ένα initial block εκτελούμε typecast στα iterations της for ώστε να πάρουν μια τιμή από το enum και έπειτα δίνουμε τα enums αυτά στη συνάρτηση για να παράξει 2 32 bit αριθμούς από τα i,j iterations. Έπειτα τα περνάμε με concatenation στην 64 bit θέση του πίνακα, και στο τέλος του block έχουμε έναν πίνακα 12x12 όπου σε κάθε κελί αντιστοιχεί μια 64 bit τιμή με έναν συνδυασμό a,b.

Έπειτα, έχουμε δύο always blocks που ενεργοποιούνται σε κάθε κύκλο ρολογιού με το ένα να είναι υπεύθυνο για την προσπέλαση του πίνακα και το φόρτωμα των τιμών, ενώ το δεύτερο για την οδήγηση των rounding modes, του πολυπλέκτη και των σημάτων ελέγχου του αποτελέσματος.

Στο πρώτο always block φορτώνουμε την 64 bit τιμή στο current value, όπου στο άλλο block θα την αναγνώσουμε κατάλληλα (32 στον a και 32 στον b) ενώ ενημερώνουμε κατάλληλα τους iterators ώστε να προχωράνε σε κάθε στοιχείο του πίνακα ενώ όταν φτάνουν στο τέλος να προχωράνε στην επόμενη γραμμή/στήλη. Το block αυτό είναι σκοπίμως απενεργοποιημένο μέσω ενός counter, ώστε να ξεκινήσει να παράγει τις τιμές τη στιγμή που εμείς θα τις χρειαστούμε, βάζοντας μια κόκκινη γραμμή X στα waves του testbench ώστε να καταλαβαίνουμε ότι βρισκόμαστε σε άλλο mode τη δεδομένη στιγμή.

Στο δεύτερο always block, αρχικοποιούμε με το reset (το οποίο στην περίπτωση αυτή λειτουργεί σύγχρονα) ενώ σε άλλη περίπτωση αυξάνουμε τον counter με τους αριθμούς που έχουμε παράξει, φορτώνουμε τα σήματα a και b μέσω του πολυπλέκτη και εκτελούμε τις χρονικές καθυστερήσεις και τον έλεγχο των αποτελεσμάτων. Για την εναλλαγή των rounding modes είναι υπεύθυνοι οι μετρητές r,k όπου ο r μετράει την περίοδο (σε παραγμένους αριθμούς) που θα σταθούμε σε κάποιο mode, ενώ ο k εφ' όσον είναι πλήρως σύμφωνος με το mapping του enum περνιέται ως είσοδος τόσο στο logic md του module όσο και ως iterator στον πίνακα με τα strings στη συνάρτηση multiplication.

Τέλος το τελευταίο if statement είναι απαραίτητο για την ενημέρωση των r και k καθώς έχουμε διαφορετικές περιόδους στους τυχαίους (10 αριθμοί ανα mode) έναντι στους σταθερούς (144). Για να γίνει ομαλά η μετάβαση, χρησιμοποιείται και ένα έξτρα σήμα control καθώς αν η αλλαγή γίνει μέσα στο πρώτο if, στον επόμενο κύκλο αντί να μηδενίσει θα πάει στο μέρος της αλλαγής και θα συνεχίσει να προσauζάνει χαλώντας την περίοδο (k=5, r=8 αλλάζει το random gen και σηκώνεται το control ώστε στον επόμενο κύκλο να ξαναμπει στο ίδιο και απλά να μηδενίσει τους iterators, αν έμπαινε στο random\_gen==0 οι περίοδοι θα άλλαζαν και δε θα αναγνώριζε τον μηδενισμό). Όταν τελειώσει και η τελευταία περίοδος κάνει display ένα μήνυμα επιτυχίας και finish, ενώ υπάρχει και ένα always block που σε περίπτωση που βρει το σήμα false ψηλά, κάνει άμεσα fatal (τελείωνει η προσομοίωση με μήνυμα λάθους, στο οποίο επιλέγουμε να δούμε το expected με αυτό που βγάλαμε εμείς). Τρέχοντας το testbench λαμβάνουμε πάντα το μήνυμα επιτυχίας, ενώ για λόγους πληρότητας ακολουθεί και μια εικόνα όλων των κυματομορφών και των σημάτων της προσομοίωσης. Ωστόσο, λόγω των display statements δεν είναι απαραίτητη η προβολή κυματομορφών για την αναγνώριση σφάλματος.

Κομμάτι των random αριθμών:



[illegible]

Για την υλοποίηση των immediate assertions

Bit	Flag	Description
0	Zero	$z = \pm Zero$
1	Infinity	$z = \pm Inf$
2	Invalid	$F = NaN$
3	Tiny	$ F  \neq Zero/Inf/NaN$ and $ F  < MinNorm$
4	Huge	$ F  \neq \pm Inf/NaN$ and $ F  > MaxNorm$
5	Inexact	$ F  \neq Zero/Inf/NaN$ and $F \neq z$
6	Unused	Reserved to 0
7	Div by 0	Division by zero, reserved to 0 otherwise

10

assertions όπου εισάγουμε τη γεννήτρια και τον μηχανισμό με τα σταθερά corner cases και χειροκίνητα πλέον επιλέγουμε χωρίς περιορισμό χρόνου πόσο θα τρέξει το simulation. Αντί αυτού μπορούμε να κάνουμε το binding και μέσα στο προηγούμενο testbench εφόσον θέλουμε εκείνο. Αφού γίνει το binding, και με τα δύο testbenches λαμβάνουμε τα εξής αποτελέσματα από το Questa.

Instance	Design unit	Design unit type	Top Category	Visibility	Total coverage	Assertions count	Assertions hit	Assertions missed	Assertion %	Assertion graph
test_bench	test_bench(fast)	Module	DU Instance	+acc=<full>	100.00%	13	13	0	100.00%	
types_t	test_bench(fast)	Module	-	+acc=<full>						
mode_t	test_bench(fast)	Module	-	+acc=<full>						
match	test_bench(fast)	Module	-	+acc=<full>						
#anonblk#174008...	test_bench(fast)	Module	-	+acc=<full>						
multiplier	fp_mult_top(fast)	Module	DU Instance	+acc=<full>	100.00%	13	13	0	100.00%	

## Επίλογος/Συμπεράσματα

Παρατηρείται πως τα assertions αποτελούν έναν εύκολο και γρήγορο τρόπο να δούμε αν όλα πήγαν σωστά χωρίς να μπλέκουμε με ιδιαίτερο προγραμματισμό του εκάστοτε testbench.

Η παρούσα υλοποίηση ακόμα και τα assertions προέκυψαν ύστερα από αναλυτική μελέτη και του IEEE-754 και εκτενή αναζήτηση, καθώς η εκφώνηση της εργασίας δεν έδινε ιδιαίτερα δεδομένα για το πως πρέπει να υλοποιηθούν όλα τα παραπάνω. Αυτό είχε το αρνητικό ότι σε πολλές περιπτώσεις χρειάστηκε το trial and error με την έτοιμη συνάρτηση προκειμένου να βρεθεί ο σωστός τρόπος να υλοποιηθούν ορισμένες μέθοδοι (ιδιαίτερα στο exception και στο rounding module).

Για την εκτέλεση και το compilation χρησιμοποιήθηκε η εξής εντολή από το τερματικό η οποία έσωσε αρκετό χρόνο από τη δημιουργία και τη διαχείριση αρχείων και projects:

```
vlog -sv *.sv
```

```
vsim top_module_name -do "do run.do; view wave; add wave -r /*; run -all"
```

Για οποιαδήποτε διευκρίνιση/ανατροφοδότηση μπορείτε να επικοινωνήσετε στο:  
kvlachak@ece.auth.gr