Εργασία Ψηφιακών Συστημάτων HW σε Χαμηλά Επίπεδα Λογικής ΙΙ

Υλοποίηση Floating Point Multiplier

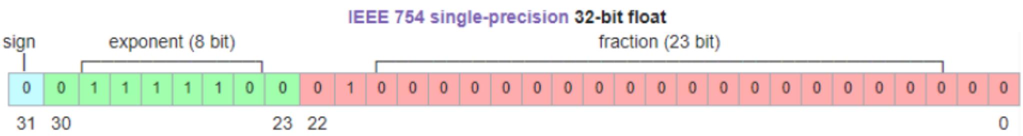
Ελευθερία Βαϊτσοπούλου AEM 10554

Ιούνιος 2024

Εισαγωγή

Σε αυτή την εργασία χρησιμοποιώντας System Verilog στο εργαλείο Questa Intel Started FPGA Edition δημιουργήσαμε έναν floating point multiplier. Ακολουθεί ανάλυση του κώδικα για την υλοποίηση, το testbench και τα αρχεία για τα assertions.

Ως είσοδο έχουμε τους 32 bit αριθμούς a, b και ως έξοδο έχουμε τον αριθμό z και status. Οι αριθμοί a και b χωρίζονται σε 3 μέρη, το bit προσήμου, τα 8 bit εκθέτη και τα υπόλοιπα 23 bit δεκαδικού (mantissa) όπως φαίνεται στο σχήμα 1.

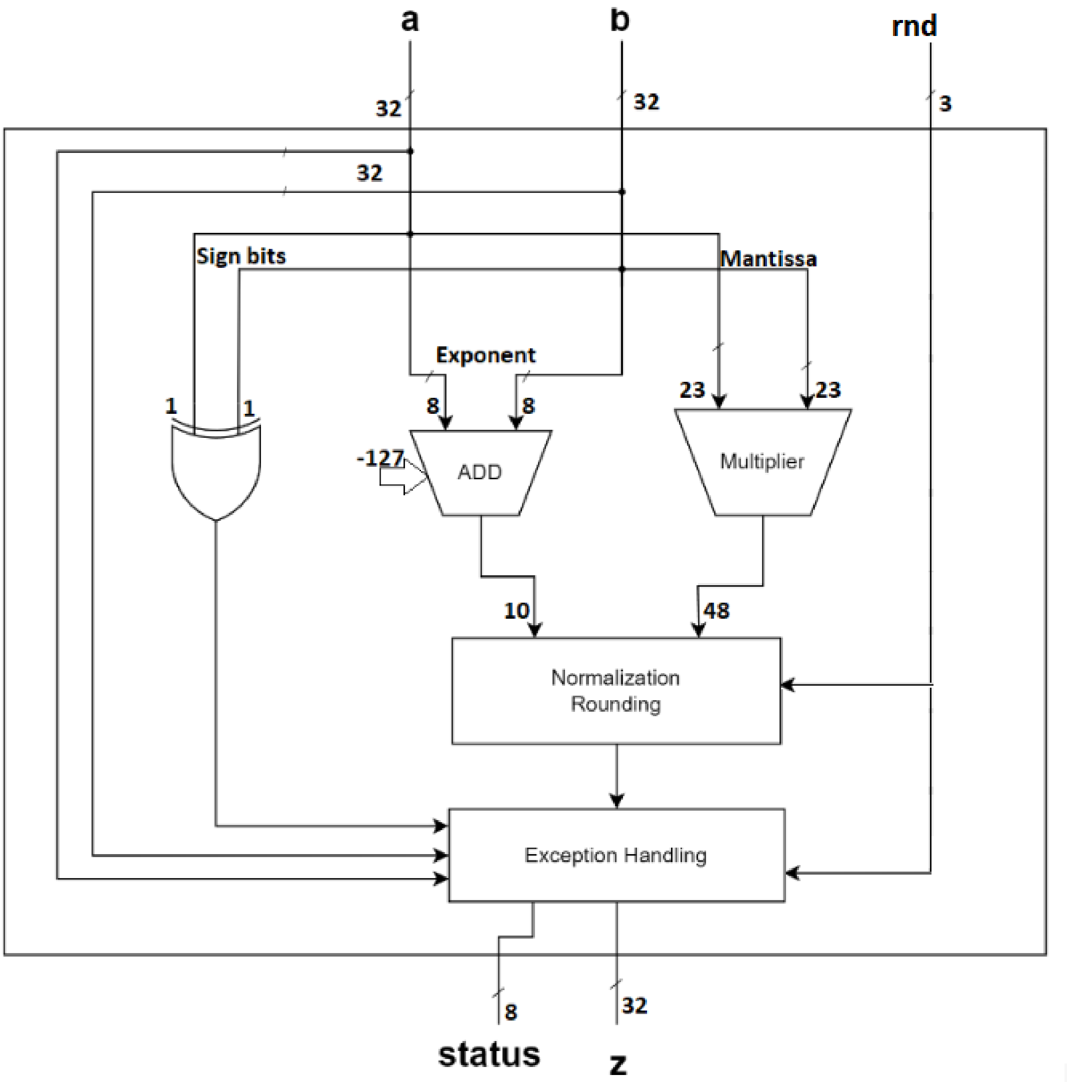


Σχήμα 1: Μονής ακρίβειας floating point σύμφωνα με τα στάνταρ IEEE 754.

Άσκηση 1

Στο πρώτο μέρος της εργασίας υλοποιούμε τα αρχεία για τον πολλαπλασιαστή, fp_mult.sv, normalize_mult.sv, round_mult.sv, exception_mult.sv.

Έχουμε το main module (fp_mult.sv) το οποίο λαμβάνει τους αριθμούς a, b που είναι 32 bits και με την βοήθεια των υπόλοιπων modules μας δίνει το τελικό αποτέλεσμα του πολλαπλασιασμού z σε 32 bits και το status, το οποίο είναι 8 bits και κάθε bit εξαρτάται από τις πράξεις που γίνονται στον πολλαπλασιαστή.



Σχήμα 2: Το block diagram των λειτουργιών του main module.

Από το σχήμα 2 μπορούμε να διακρίνουμε τα 7 στάδια στα οποία έχουμε χωρίσει τον κώδικα του.

1. Floating point number sign calculation
2. Exponent addition
3. Exponent subtraction of bias
4. Mantissa multiplication (including leading ones)
5. Truncation and normalization
6. Rounding

7. Exception handling

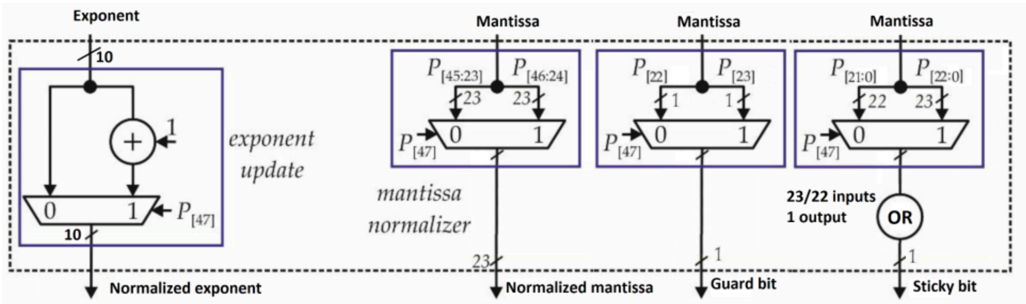
Η σημασία κάθε bit του status βρίσκεται στον παρακάτω πίνακα.

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

Σχήμα 3: Η σημασία κάθε bit του status.

Οι ζητούμενες αναθέσεις έγιναν μέσα σε always_comb blocks ώστε τα σήματα να ενημερώνονται εγκαίρως κάθε φορά που αλλάζει κάποιο από αυτά. Στη συνέχεια καλούνται τα modules για τα βήματα 5 έως 7 μέσω του module instantiation.

Το επόμενο module (normalize_mult.sv) φροντίζει για την κανονικοποίηση του δεκαδικού μέρους του αριθμού (mantissa). Μέσω ενός always_comb βρόχου, ελέγχουμε το ψηφίο πριν το κόμμα και αναλόγως με την τιμή του εκτελούμε τις ζητούμενες πράξεις όπως φαίνονται στο παρακάτω σχήμα 4.



Σχήμα 4: Το block diagram των λειτουργιών του normalization module.

Σημαντικό είναι ότι λαμβάνουμε τα guard and sticky bits τα οποία θα μας βοηθήσουν στην στρογγυλοποίηση του αριθμού στο επόμενο βήμα.

Η υλοποίηση του αρχείου round_mult.sv είναι αυτή που με απασχόλησε περισσότερο. Χρησιμοποιήθηκε στρογγυλοποίηση σύμωνα με τα πρότυπα IEEE 754 και χρειάστηκε με διάφορες δοκιμές να πετύχω το καλύτερο ποσοστό επιτυχημένων πολλαπλασιασμών (περίπου 85%) που θα εξηγηθεί περαιτέρω στην άσκηση 2.

Για την εργασία χρησιμοποιήθηκαν οι εξής στρογγυλοποιήσεις.

Name	Values	Description
round	IEEE_near	IEEE round to nearest, even. Round to the nearest representable value, if both are equally near, output the result with an even significand.
	IEEE_zero	IEEE round towards zero.
	IEEE_pinf	IEEE round to +Infinity.
	IEEE_ninf	IEEE round to -Infinity.
	near_up	Round to the nearest representable value, if both are equally near, output the result closer to +Infinity.
	away_zero	Round away from zero.

Σχήμα 5: Οι στρογγυλοποιήσεις που υλοποιήθηκαν.

Για την υλοποίηση των στρογγυλοποιήσεων χρησιμοποιήθηκε ένα case statement μέσα σε ένα always_comb block. Κάθε περίπτωση εξαρτάται από τα bit: inexact, sign_z (που υπολογίστηκε στο πρώτο βήμα της άσκησης και στο αρχείο αναγράφεται ως calc_sign), guard, sticky and mantissa[0]. Το inexact bit μας δείχνει εάν ο αριθμός που θα προκύψει θα είναι ακριβής αριθμός, κάτι που ισχύει μόνο εάν τα $guard == 0$ και $sticky == 0$.

Αναλόγως με τους συνδυασμούς που χρησιμοποιήθηκαν (σχήμα 6), ενεργοποιείται ή όχι το bit round_up που θα καθορίσει εάν το αποτέλεσμα χρειάζεται να αυξηθεί κατά ένα ή όχι.

```
case (round)
          IEEE_near: begin
                  if (guard == 1 && (sticky == 1 || mantissa[0] == 1))
                          round_up = 1;
                  else
                          round_up = 0;
         end
         IEEE_zero: round_up = 0;
         IEEE_pinf: round_up = (~calc_sign && inexact);
         IEEE_ninf: round_up = (calc_sign && inexact);
         near_up: round_up = guard;
         away_zero: round_up = inexact;
         default: begin
                  if (guard == 1 && (sticky == 1 || mantissa[0] == 1))
                          round_up = 1;
                  else
                          round_up = 0;
         end
endcase
```

Σχήμα 6: Η υλοποίηση του round_up.

Τέλος σύμφωνα με τις οδηγίες, αν το MSB ψηφίο του αποτελέσματος είναι 1, τότε πρέπει το αποτέλεσμα να ολισθήσει δεξιά κατά 1 και ο εκθέτης να αυξηθεί κατά 1, αλλιώς δε χρειάζεται καμία αλλαγή.

Συνεχίζοντας με το τελευταίο ζητούμενο αρχείο, το exception_mult.sv διαχειριζόμαστε τις ακραίες περιπτώσεις πολλαπλασιασμού όπως για παράδειγμα 0 επί άπειρο.

Σε αυτό το αρχείο έχουμε υλοποιήσει ένα typedef enum interp_t που μας βοηθάει στην διαχείριση των περιπτώσεων πολλαπλασιασμού. Περιλαμβάνει τα ZERO, INF, NORM, MIN_NORM, and MAX_NORM. Οι περιπτώσεις NaN θεωρούνται άπειρα και οι περιπτώσεις Denormals θεωρούνται μηδενικά.

Στην συνέχεια υλοποιούμε 2 συναρτήσεις τις num_interp και z_num. Η πρώτη επιστρέφει ένα enum interp_t αναλόγως με τον αριθμό που παίρνει ως όρισμα. Η δεύτερη κάνει το αντίστροφο, από το enum interp_t επιστρέφει τη μη προσημασμένη τιμή του.

Τέλος, μέσα σε ένα always_block χρησιμοποιούμε δύο case statements για τις περιπτώσεις των δύο αριθμών μας. Αναλόγως με τους παράγοντες μας διαχειριζόμαστε τα αποτελέσματα σύμφωνα με το σχήμα 7 και ενεργοποιούνται τα κατάλληλα status bits.

A sA.expA.sigA	B sB.expB.sigB	Infinitely Precision Result (F)
± Zero	± Zero	$(-1)^{sA+sB}$ Zero
± Zero	±Norm	$(-1)^{sA+sB}$ Zero
± Zero	± Inf	+ Inf
± Inf	± Inf	$(-1)^{sA+sB}$ Inf
± Inf	±Norm	$(-1)^{sA+sB}$ Inf
± Inf	± Zero	+ Inf
±Norm	± Zero	$(-1)^{sA+sB}$ Zero
±Norm	± Inf	$(-1)^{sA+sB}$ Inf

Σχήμα 7: Πιθανοί ακραίοι συνδυασμοί παραγόντων.

Οι συνδυασμοί Normal x Normal έχουν την επιπλέον λογική του overflow και underflow, τα οποία υπολογίζονται ως εξής:

```
always_comb begin
        overflow = (signed'(exp_round) > 255);
        underflow = (signed'(exp_norm) < 0);
        z_calc = {sign_z, exp_round[7:0], result[22:0]};
end
exception except_inst(.*);
```

Σχήμα 8: Υπολογισμός του overflow και underflow.

Στη συνέχεια αναλόγως με τη παράμετρο round και αν ο αριθμός είναι θετικός ή αρνητικός, πραγματοποιούνται και οι κατάλληλες στρογγυλοποιήσεις.

Σε αυτή την άσκηση υλοποίησα ένα ακόμη αρχείο, το πακέτο rounding_package.sv το οποίο περιλαμβάνει έναν typedef enum round_t και μία συνάρτηση round_t_to_string. Ο πρώτος έχει τις τιμές που χρησιμοποιήθηκαν για την στρογγυλοποίηση και η δεύτερη παίρνει ως όρισμα την τιμή round_t το επιστρέφει ως string ώστε να μπορεί να χρησιμοποιηθεί ως όρισμα στο αρχείο multiplication.sv που δινόταν και θα μας είναι χρήσιμο αργότερα.

Για να μπορέσουμε να προχωρήσουμε στο επόμενο βήμα της άσκησης ενώνουμε όλα αυτά τα modules στο αρχείο wrapper που μας δίνεται το fp_mult_top.sv.

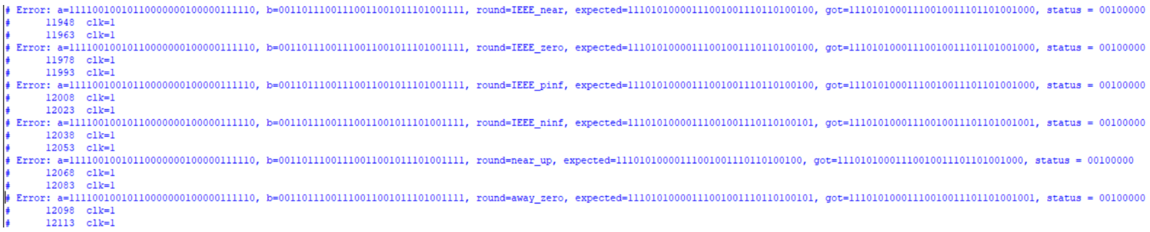
Άσκηση 2

Στο δεύτερο μέρος της εργασίας υλοποιούμε το fp_testbench.sv, για να ελέγξουμε την ορθότητα των αποτελεσμάτων μας. Ορίζουμε ‘$timescale 1ns/1ps$ και περίοδο ρολογιού τα 15ns και ορίζουμε (instantiate) το αρχείο wrapper από την προηγούμενη άσκηση. Ξεκινάμε με το σήμα rst ενεργοποιημένο ώστε να μηδενίσουμε τις αρχικές τιμές των μεταβλητών μας και να συνεχίσουμε σε πράξεις και αποτελέσματα χωρίς προβλήματα.

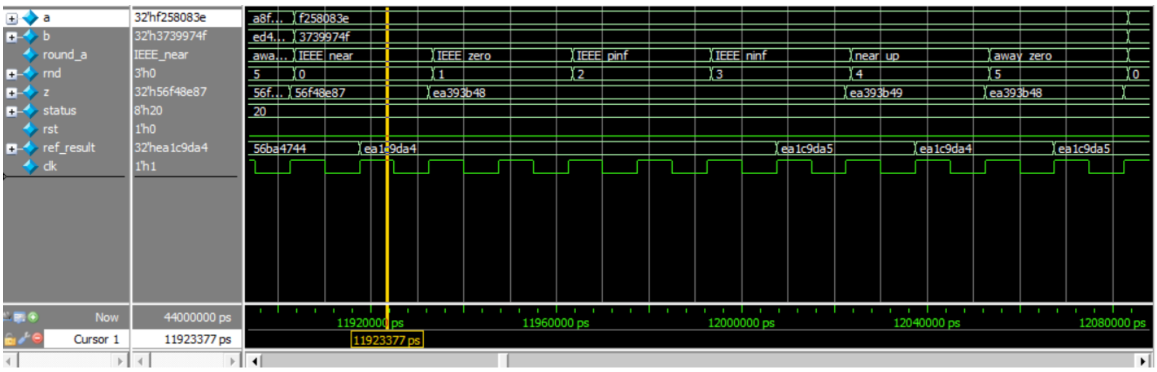
Για την δοκιμή των προηγούμενων modules δοκιμάζουμε αρχικά 100 τυχαίες τιμές για κάθε δυνατή στρογγυλοποίηση. Άρα μαζί με κάθε πιθανή περίπτωση στρογγυλοποίησης πραγματοποιούνται 600 πολλαπλασιασμοί. Χρησιμοποιούμε τη συνάρτηση $urandom για τη παραγωγή τυχαίων αριθμών α και β. Για να ελέγξουμε αν τα αποτελέσματά μας είναι σωστά χρησιμοποιούμε τη συνάρτηση multiplication που μας δίνεται. Έκανα μια μικρή τροποποίηση στο αρχείο της και την όρισα ως πακέτο ώστε να μπορούμε να τη χρησιμοποιήσουμε στο fp_mult_top ώστε να ανανεώνεται η τιμή της μαζί με τη τιμή του z.

Για να συγχρονιστούν τα αποτελέσματα και να συγκριθούν σωστά έχουμε τοποθετήσει καθυστερήσεις 2 κύκλων ρολογιού συν 1ns. Αυτό φαίνεται και στις κυματομορφές στις παρακάτω εικόνες.

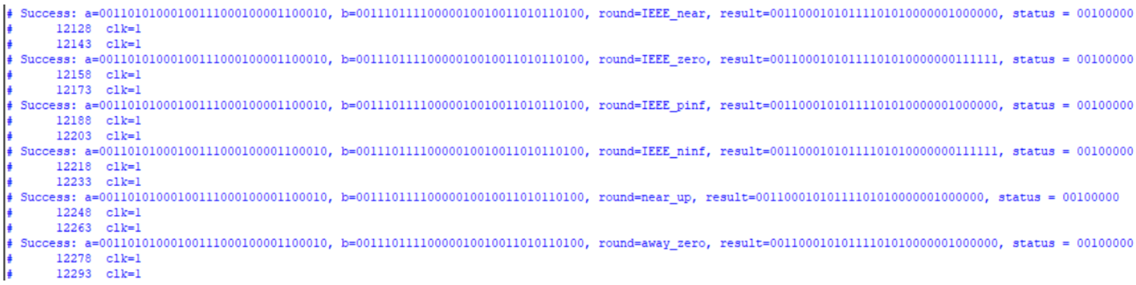
Όπως αναφέραμε και νωρίτερα, οι περισσότεροι πολλαπλασιασμοί γίνονται σωστά. Μετά από αρκετές δοκιμές και συγκρίσεις, παρατηρούμε ότι οι μόνοι πολλαπλασιασμοί που δεν βγαίνουν σωστοί είναι κάποιοι που έχουν ενεργό μόνο το inexact bit, όπως στο σχήμα 9. Είναι λογικό δεδομένου ότι το bit αυτό συμβολίζει ότι το αποτέλεσμα δεν είναι ακριβές. Οι διαφορές στα αποτελέσματα είναι στον υπολογισμό της mantissa. Δεν είναι όλοι οι αριθμοί που έχουν μόνο το bit αυτό ενεργό λάθος όπως φαίνεται στο σχήμα 11 αλλά ένα μικρό μέρος τους μόνο.


```
# Error: a=11110010010110000000100000111110, b=00110111001110011001011101001111, round=IEEE_near, expected=11101010000111001001110110100100, got=11101010001110010011101101001000, status = 00100000
#       11948  clk=1
#       11963  clk=1
# Error: a=11110010010110000000100000111110, b=00110111001110011001011101001111, round=IEEE_zero, expected=11101010000111001001110110100100, got=11101010001110010011101101001000, status = 00100000
#       11978  clk=1
#       11993  clk=1
# Error: a=11110010010110000000100000111110, b=00110111001110011001011101001111, round=IEEE_pinf, expected=11101010000111001001110110100100, got=11101010001110010011101101001000, status = 00100000
#       12008  clk=1
#       12023  clk=1
# Error: a=11110010010110000000100000111110, b=00110111001110011001011101001111, round=IEEE_ninf, expected=11101010000111001001110110100101, got=11101010001110010011101101001001, status = 00100000
#       12038  clk=1
#       12053  clk=1
# Error: a=11110010010110000000100000111110, b=00110111001110011001011101001111, round=near_up, expected=11101010000111001001110110100100, got=11101010001110010011101101001000, status = 00100000
#       12068  clk=1
#       12083  clk=1
# Error: a=11110010010110000000100000111110, b=00110111001110011001011101001111, round=away_zero, expected=11101010000111001001110110100101, got=11101010001110010011101101001001, status = 00100000
#       12098  clk=1
#       12113  clk=1
```

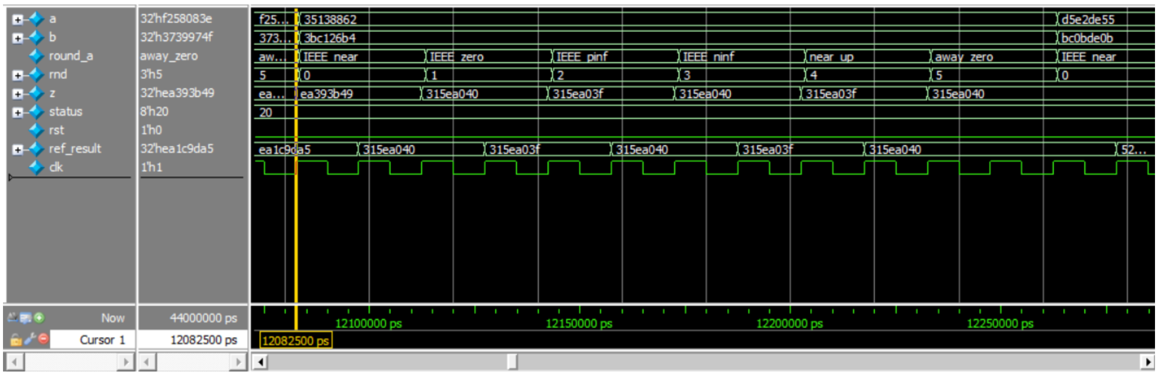
Σχήμα 9: Ανεπιτυχείς πολλαπλασιασμοί με τυχαίους αριθμούς α και β με μόνο το inexact bit ενεργό.



Σχήμα 10: Οι κυματομορφές των σημάτων για τους ανεπιτυχείς πολαπλασιασμούς των α και β με μόνο το inexact bit ενεργό.



Σχήμα 11: Επιτυχείς πολλαπλασιασμοί με τυχαίους αριθμούς α και β με μόνο το inexact bit ενεργό.



Σχήμα 12: Οι κυματομορφές των σημάτων για τους επιτυχείς πολαπλασιασμούς των α και β με μόνο το inexact bit ενεργό.

Στη συνέχεια χρειάζεται να ελέγξουμε τις ακραίες περιπτώσεις. Για αυτό το λόγο δημιουργούμε τον μονοδιάστατο πίνακα corner_cases με 12 διαφορετικές τιμές. Δημιουργούμε και μια συνάρτηση match_case ώστε να ταιριάζουμε κάθε τιμή στην ακραία περίπτωση που αντιπροσωπεύει. Αυτές που χρησιμοποιήσαμε είναι οι ακόλουθες:

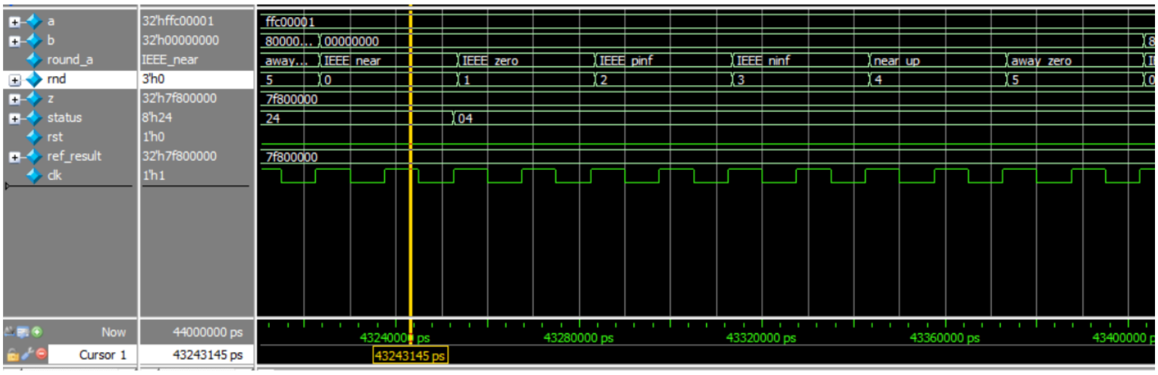
Value (32 bit)	Corner case
01111111100000000000000000000001	Signaling Positive NaN
11111111100000000000000000000001	Signaling Negative
01111111100000000000000000000000	Positive Infinity
11111111100000000000000000000000	Negative Infinity
01000000101000000000000000000000	Positive Normal (e.g., 5.0)
11000000101000000000000000000000	Negative Normal (e.g., -5.0)
00000000000000000000000000000001	Positive Denormal
10000000000000000000000000000001	Negative Denormal
00000000000000000000000000000000	Positive Zero
10000000000000000000000000000000	Negative Zero
01111111110000000000000000000001	Quiet Positive NaN
11111111110000000000000000000001	Quiet Negative NaN

Πίνακας 1: Corner cases που χρησιμοποιήθηκαν για τον έλεγχο της ορθότητας του κώδικα.

Συνολικά για κάθε πιθανή στρογγυλοποίηση και περίπτωση πραγματοποιούνται 864 πολλαπλασιασμοί όπου όλοι βγαίνουν σωστοί. Για αυτό και το συνολικό ποσοστό σωστών αποτελεσμάτων (1250/1464) ανέρχεται στο 85%.

```
# Success: a=quiet_neg_nan, b=pos_zero, round=IEEE_near, result=01111111100000000000000000000000, status = 00000100
#      43268  clk=1
#      43283  clk=1
# Success: a=quiet_neg_nan, b=pos_zero, round=IEEE_zero, result=01111111100000000000000000000000, status = 00000100
#      43298  clk=1
#      43313  clk=1
# Success: a=quiet_neg_nan, b=pos_zero, round=IEEE_pinf, result=01111111100000000000000000000000, status = 00000100
#      43328  clk=1
#      43343  clk=1
# Success: a=quiet_neg_nan, b=pos_zero, round=IEEE_ninf, result=01111111100000000000000000000000, status = 00000100
#      43358  clk=1
#      43373  clk=1
# Success: a=quiet_neg_nan, b=pos_zero, round=near_up, result=01111111100000000000000000000000, status = 00000100
#      43388  clk=1
#      43403  clk=1
# Success: a=quiet_neg_nan, b=pos_zero, round=away_zero, result=01111111100000000000000000000000, status = 00000100
#      43418  clk=1
#      43433  clk=1
```

Σχήμα 13: Παράδειγμα επιτυχούς πολλαπλασιασμού των corner cases sig_neg_nan και pos_zero για κάθε δυνατή στρογγυλοποίηση.



Σχήμα 14: Οι κυματομορφές των σημάτων για τις επιτυχίες των corner cases.

Άσκηση 3

Στο τρίτο μέρος της εργασίας υλοποιούμε τα αρχεία test_status_bits.sv και test_status_z_combinations.sv.

Το πρώτο αρχείο χρησιμοποιώντας immediate assertions τσεκάρει πότε συνδυασμοί bit είναι ενεργοί μαζί. Αυτό μπορούμε να το συμπεράνουμε από το πινακάκι στο σχήμα 3.

Για τα bit huge, tiny και inexact κανονικά συγκρίνεται το αποτέλεσμα από πριν γίνει η στρογγυλοποίηση του αριθμού. Ωστόσο, για να δούμε και μερικές αποτυχίες τα έχω θέσει να αποτυγχάνουν σύμφωνα με το τελικό αποτέλεσμα. Είναι προφανές και απο τη σύνταξη του κώδικα ότι δεν είναι αδύνατο κάποιοι από αυτούς τους συνδυασμούς να συνυπάρχουν. Συγκεκριμένα είναι οι εξής: tiny και zero, tiny και inf, tiny και NaN, huge και zero, huge και inf, huge και NaN, inexact και zero, inexact και inf, inexact και NaN.

```
# Success: a=00100001001010101100011100100101, b=10010110011001001011001000101011, round=IEEE_near, result=10000000000000000000000000000000, status = 00101001
#       9428  clk=1
# FAIL: Zero and Tiny asserted together
# FAIL: Zero and Inexact asserted together
#       9443  clk=1
# FAIL: Zero and Tiny asserted together
# FAIL: Zero and Inexact asserted together
# Success: a=00100001001010101100011100100101, b=10010110011001001011001000101011, round=IEEE_zero, result=10000000000000000000000000000000, status = 00101001
#       9458  clk=1
# FAIL: Zero and Tiny asserted together
# FAIL: Zero and Inexact asserted together
#       9473  clk=1
# FAIL: Zero and Tiny asserted together
# FAIL: Zero and Inexact asserted together
# Success: a=00100001001010101100011100100101, b=10010110011001001011001000101011, round=IEEE_pinf, result=10000000000000000000000000000000, status = 00101001
#       9488  clk=1
# FAIL: Zero and Tiny asserted together
# FAIL: Zero and Inexact asserted together
#       9503  clk=1
```

Σχήμα 15: Παράδειγμα αποτυχίας property για zero - tiny και zero - inexact.

Στον κώδικα δεν έχουμε συμπεριλάβει τα σφάλματα που μπορεί να προέκυπταν από τα τελευταία 2 bit διότι τα έχουμε θέσει πάντα να είναι στο 0 αφού το ένα ονομάζεται unused και το άλλο είναι για τη διαίρεση με το 0 που δε την υλοποιούμε σε αυτή στην εργασία. Όποτε δεν εμφανίζεται μήνυμα αποτυχίας για οποιοδήποτε property τότε σημαίνει ότι έχει περάσει σωστά. Δεν εμφανίζεται μήνυμα επιτυχίας για να μην υπάρχουν υπερβολικά πολλές πληροφορίες στην έξοδο του testbench.

Για το δεύτερο αρχείο έχουμε υλοποιήσει 5 concurrent assertions για να ελέγξουμε τις εξής συνθήκες:

- Όταν ενεργοποιείται το zero bit πρέπει ο εκθέτης του z να είναι ίσος με 0.
- Όταν ενεργοποιείται το inf bit πρέπει ο εκθέτης του z να είναι ίσος με 1.
- Όταν ενεργοποιείται το nan bit πρέπει 2 κύκλους ρολογιού νωρίτερα ο εκθέτης του a να είναι ίσος με 0 και ο εκθέτης του b να είναι ίσος με 1 ή το αντίστροφο.
- Όταν ενεργοποιείται το huge bit πρέπει ο εκθέτης του z να είναι ίσος με 1 ή να ισούται με τη περίπτωση του max normal case.
- Όταν ενεργοποιείται το tiny bit πρέπει ο εκθέτης του z να είναι ίσος με 0 ή να ισούται με τη περίπτωση του min normal case.

Για τη περίπτωση του NaN χρησιμοποιήθηκε η συνάρτηση $past για τους προηγούμενους 4 κύκλους ρολογιού γιατί έχουμε βάλει τα αποτελέσματα να ανανεώνονται κάθε 2 κύκλους ρολογιού.

```
# Success: a=pos_inf, b=pos_denormal, round=IEEE_near, result=01111111100000000000000000000000, status = 00000100
#      23468  clk=1
#   23467500  FAIL: Property for nan's exponent failed.
```

Σχήμα 16: Παράδειγμα αποτυχίας του nan property

Για να μπορούμε να εξετάσουμε τα assertions μας συνδέουμε τα 2 αρχεία αυτά στο testbench που δημιουργήσαμε στην άσκηση 2 με το wrapper (fp_mult_top.sv) μέσω της εντολής bind.