Floating Point Multiplier

Σχεδίαση πολλαπλασιαστή δύο αριθμών κινητής υποδιαστολής σε System Verilog

Μάθημα: Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής ΙΙ

Τσιτσάνου Αννα AEM: 10051

Ιούλιος 2023

Εισαγωγή

Η παρούσα εργασία αφορά τη σχεδίαση και την επαλήθευση ενός πολλαπλασιαστή δύο αριθμών κινητής υποδιαστολής σε System Verilog. Η εργασία χωρίζεται σε τρία τμήματα.
Το πρώτο τμήμα σχετίζεται με την σχεδίαση του πολλαπλασιαστή (design), η οποία επιτυγχάνεται με την υλοποίηση τεσσάρων modules. Τα modules απαρτίζονται από τα main(fp_mult), normalization(normalize_mult), rounding(round_mult) κι exception handling(exception_mult).
Το δεύτερο τμήμα αφορά την υλοποίηση ενός testbench για τον έλεγχο της ορθότητας του κώδικα.
Τέλος, το τρίτο μέρος αποτελεί η δημιουργία ενός SVA για την επαλήθευση της ορθής λειτουργίας του κυκλώματος.

1.Σχεδίαση Πολλαπλασιαστή

Ο πολλαπλασιαστής της συγκεκριμένης εργασίας χρησιμοποιεί δύο floating point αριθμούς single precision, δηλαδή δύο αριθμούς των 32 bit, με 1 bit sign, 8 bit exponent και 23 bit mantissa.

1.1 Main Module

Το main module (fp_mult) έχει δύο εισόδους, δύο αριθμούς των 32 bit, a, b και δύο εξόδους, το 32 bit αποτέλεσμα z του πολλαπλασιασμού των a και b, και το 8 bit status, που δείχνει τι είδος αριθμού είναι το αποτέλεσμα. Ακόμα, έχει και μία παράμετρο round, που καθορίζει με ποιον τρόπο θα γίνει η στρογγυλοποίηση.
Αρχικά, σε ένα always_comb βρίσκουμε το sign, κάνοντας xor τα sign των a και b. Βρίσκουμε το exponent προσθέτοντας τα exponents των a και b και αφαιρώντας το bias(127). Τέλος, βρίσκουμε το mantissa πολλαπλασιάζοντας τα mantissa των a και b (συμπεριλαμβανομένων και των leading ones).
Το exponent και η mantissa που βρέθηκαν παραπάνω δίνονται ως είσοδοι στο normalization module.
Στη συνέχεια παίρνουμε την normalized mantissa, προσθέτουμε το leading one και δίνεται ως είσοδος στο rounding module.

Μετά το rounding, ελέγχεται αν το 25ο bit της mantissa. Αν είναι 1, κάνουμε shift δεξιά κατά ένα το αποτέλεσμα και το αποθηκεύουμε στο post_round_mantissa, ενώ αυξάνουμε το normalised exponent κατά ένα και το αποθηκεύουμε στο post_round_exponent. Αλλιώς, διατηρούμε το round_mantissa και το norm_exponent.
Έπειτα, σε ένα always_comb ελέγχουμε το exponent για overflow και underflow, τα οποία γίνονται 1 όταν η τιμή του post_round_exponent (τα 8 low bits του normalized exponent) είναι ίση με 8'b11111111 και η τιμή του norm_exponent 0 αντίστοιχα.
Σε ένα ακόμα always_comb, γίνεται assigned η τιμή του z_calc, δηλαδή ένα πρώιμο αποτέλεσμα, μετά το rounding και πριν το exception handling. Συγκεκριμένα, στο z_calc[31] τοποθετούμε το sign που υπολογίστηκε νωρίτερα, στο z_calc[30:23] τα 8 low bits της post_round_exponent και στο z_calc[22:0] τη mantissa μετά το rounding, χωρις το leading one bit.
Τέλος, γίνονται assigned τα bit της εξόδου status, χρησιμοποιώντας τις 1 bit εξόδους του exception handling module.

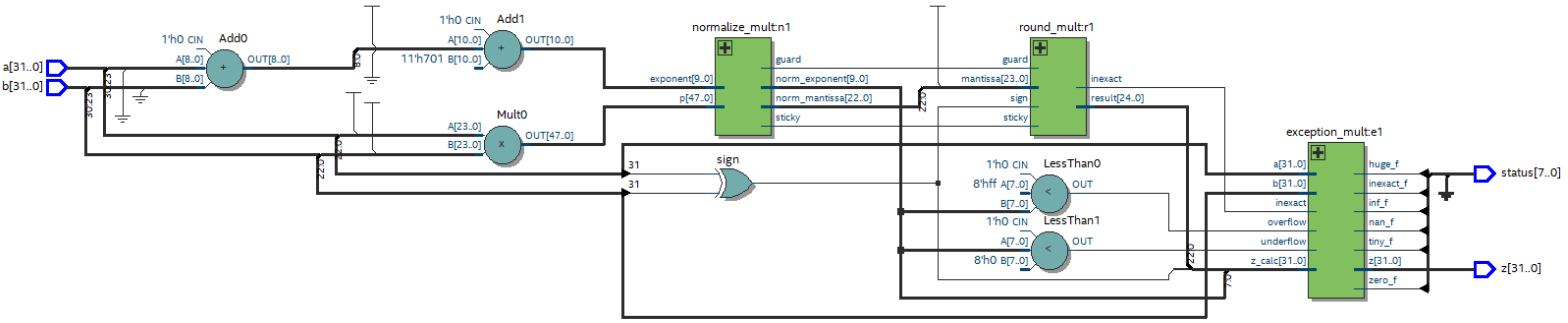
Κώδικας:

```
module fp_mult #(parameter round = "IEEE_near")
                                        (output logic [31:0] z,
                                        output logic [7:0] status,
                                        input logic [31:0] a, b);


        logic sign, sticky, guard, inexact, overflow, underflow, zero_f, inf_f, nan_f, tiny_f, huge_f, inexact_f;
        logic [9:0] exponent, norm_exponent;
        logic [47:0] mantissa;
        logic [22:0] norm_mantissa;
        logic [23:0] mant_to_round;
        logic [24:0] round_mantissa;
        logic [31:0] z_calc;


        normalize_mult
n1(.sticky(sticky),.guard(guard),.norm_exponent(norm_exponent),.norm_mantissa(norm_mantissa),.p(mantiss
a),.exponent(exponent));
        round_mult
r1(.result(round_mantissa),.inexact(inexact),.sign(sign),.sticky(sticky),.guard(guard),.mantissa(mant_to_round))
;
        exception_mult e1(.*);

        always_comb
        begin
        sign <= a[31] ^ b[31];

        exponent <= a[30:23] + b[30:23] - 127;
```

```
	mantissa <= add_leading_one(a[22:0])  * add_leading_one(b[22:0]);
	end



	always_comb
	begin
	mant_to_round = add_leading_one(norm_mantissa);
	end

	always @ (round_mantissa)
	begin
		if(round_mantissa[24])
			begin
				post_round_mantissa <= round_mantissa >> 1;
				post_round_exponent <= norm_exponent + 1;
			end
		else
			begin
				post_round_mantissa <= round_mantissa;
				post_round_exponent <= norm_exponent;
			end
	end



	always_comb begin
		if (norm_exponent[7:0] > 8'b11111111) overflow = 1;
		else overflow = 0;
		if (norm_exponent[7:0] < 0) underflow = 1;
		else underflow = 0;
	end

	always_comb
	begin
		z_calc[31] <= sign;
		z_calc[30:23] <= norm_exponent[7:0];
		z_calc[22:0] <= round_mantissa[22:0];
	end

	always_comb
	begin
		status[0] <= zero_f;
		status[1] <= inf_f;
		status[2] <= nan_f;
		status[3] <= tiny_f;
		status[4] <= huge_f;
		status[5] <= inexact_f;
		status[6] <= 1'b0;
		status[7] <= 1'b0;
	end

endmodule
```

Σημείωση: Για την προσθήκη του leading one στη mantissa δημιουργήθηκε η παρακάτω συνάρτηση.

```
function [23:0] add_leading_one(logic [22:0] variable);

        return variable + 24'b100000000000000000000000;
endfunction
```



Εικόνα 1: main module

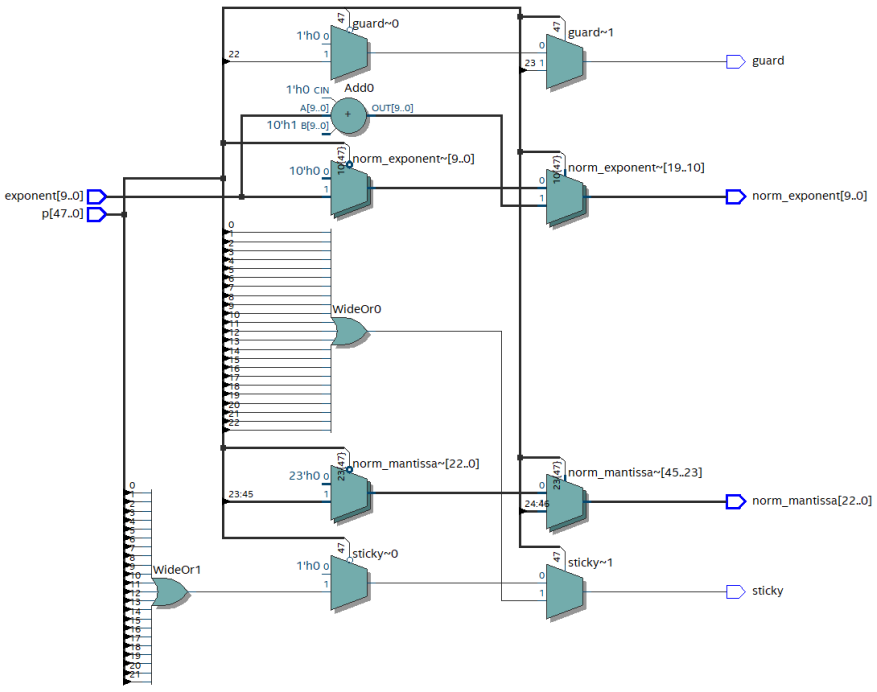
1.2 Normalization Module

Το normalization module(normalize_mult) έχει δύο εισόδους, την 48 bit mantissa(p) και το 10 bit exponent που υπολογίστηκαν στο πρώτο always_comb του main module. Οι έξοδοι του module είναι το 10 bit normalized exponent, η 23 bit normalized mantissa(χωρίς το leading one), το guard bit και το sticky bit.
Το module απαρτίζεται από ένα always_comb στο οποίο ελέγχεται η τιμή του MSB της mantissa και ανάλογα αλλάζουν οι τιμές της εξόδου.
Συγκεκριμένα, αν το MSB είναι 1, σημαίνει πως είναι το leading one, άρα η normalized mantissa θα είναι τα bit p[46:24], το guard bit θα είναι το p[23] και το sticky bit θα είναι το αποτέλεσμα του OR των τελευταίων 23 bit του p, ενώ το exponent θα αυξηθεί κατά ένα.
Αντιθετα, αν το MSB είναι 0, τότε το leading one είναι το p[46], άρα, η normalized mantissa θα είναι τα p[45:23], το guard bi θα είναι το p[22] και το sticky bit το αποτέλεσμα της πράξης OR των τελευταίων 21 bit του p. Το exponent θα παραμείνει ίδιο με της εισόδου. Σε οποιαδήποτε άλλη περίπτωση, όλοι οι έξοδοι μηδενίζονται.

Κώδικας:

```
module normalize_mult(output logic sticky, guard,
                                        output logic [22:0] norm_mantissa,
                                        output logic [9:0] norm_exponent,
                                        input logic [47:0] p,
                                        input logic [9:0] exponent);
always_comb
```

```
begin
if(p[47] == 1)
        begin
        norm_exponent <=  exponent + 1;
        norm_mantissa <= p[46:24];
        guard <= p[23];
        sticky <= |p[22:0];
        end
else if(p[47] == 0)
        begin
        norm_exponent <= exponent;
        norm_mantissa <= p[45:23];
        guard <= p[22];
        sticky <= |p[21:0];
        end
else
        begin
        norm_mantissa <= 0;
        norm_exponent <= 0;
        guard <= 0;
        sticky <= 0;
        end
end
endmodule
```



Εικόνα 2: normalization module

1.3 Rounding module

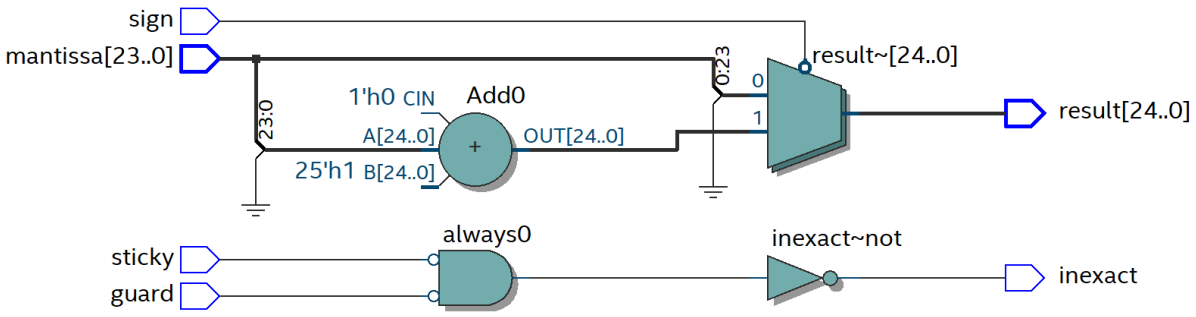
Το rounding module έχει τέσσερις εισόδους, την 24 bit mantissa(normalized mantissa + leading one που υπολογίστηκε από το main module), τα guard και sticky bits από το normalization module και το sign από το main module. Επίσης, χρειάζεται η παράμετρος round.
Διαθέτει 2 εξόδους, το 25 bit result(post rounding mantissa με 1 extra bit για overflow) και το inexact bit.
Αρχικά, υπολογίζεται το inexact bit σε ένα always_comb. Όταν το sticky ΚΑΙ το guard bit είναι μηδέν, το inexact γίνεται ίσο με 1. Σε οποιαδήποτε άλλη περίπτωση το inexact είναι 0.
Στη συνέχεια, μέσα σε ένα always_comb δημιουργούμε ένα case, το οποίο ανάλογα με την τιμή του round δίνει διαφορετική τιμή στη mantissa(δηλαδή τη στρογγυλοποιεί με διαφορετικό τρόπο).

- Αν το round = “IEEE_near”, τότε αν το guard bit είναι μηδέν, το result θα είναι ίσο με τη mantissa, ενώ σε άλλη περίπτωση το result θα είναι ίσο με τη mantissa+1.
- Αν το round = “IEEE_zero”, τότε το result γίνεται ίσο με τη mantissa.
- Αν το round = “IEEE_pinf”, τότε το result γίνεται mantissa + 1 για sign = 0, ενώ γίνεται mantissa για sign = 1.
- Αν το round = “IEEE_ninf”, τότε result γίνεται mantissa για sign = 0, ενώ γίνεται mantissa+1 για sign = 1.
- Αν το round = “near_up”, τότε αν το guard είναι μηδέν, το result θα είναι ίσο με τη mantissa. Σε άλλη περίπτωση το result γίνεται mantissa +1;
- Αν το round = “away_zero”, τοτε το result γίνεται mantissa+1.

Κώδικας:

```
module round_mult#(parameter round = "IEEE_near”)(output logic [24:0] result,
                                                   output logic inexact,
                                                   input logic [23:0] mantissa,
                                                   input logic sign, sticky, guard);


always_comb
begin
if(guard == 0 && sticky == 0)
        begin
        inexact = 0;
```

```
        end
else
        inexact = 1;
end

always_comb
begin
case(round)
        "IEEE_near": begin
                                                if (guard == 0)
                                                        result = mantissa;
                                                else result = mantissa + 1;
                                                end
        "IEEE_zero": result = mantissa;
        "IEEE_pinf": begin
                                                if(sign == 0) result = mantissa + 1;
                                                else result = mantissa;
                                        end
        "IEEE_ninf": begin
                                                if(sign == 1) result = mantissa + 1;
                                                else result = mantissa;
                                        end
         "near_up": begin
                                                if (guard == 0) result = mantissa;
                                                else result = mantissa + 1;
                                        end
        "away_zero": result = mantissa + 1;
                        default: begin
                                                if (guard == 0)
                                                        result = mantissa;
                                                else result = mantissa + 1;
                                        end
        endcase
end


endmodule
```



Εικόνα 3: rounding module

1.4 Exception handling module

Το module exception handling(exception_mult) έχει 6 εισόδους, οι αρχικές 32bit τιμές a, b, το 32 bit αποτέλεσμα z_calc, τα overflow, underflow bits, που υπολογίστηκαν στο main module, καθώς και το inexact bit που υπολογίστηκε στο rounding.
Οι έξοδοι είναι το τελικό 32bit αποτέλεσμα z και 6 bits (zero_f, inf_f, nan_f, tiny_f, huge_f, inexact_f) που αποτελούν μέρος του status.
Αρχικά, δημιουργήθηκε ένα typedef enum interp_t με πιθανές τιμές ZERO, INF, NORM, MIN_NORM, and MAX_NORM.
Έπειτα, υλοποιήθηκε η συνάρτηση num_interp που έχει σαν όρισμα ένα 32bit αριθμό και ανάλογα με το exponent και τη mantissa του, επιστρέφει το είδος του αριθμού, που ανήκει στο enum interp_t(εκτός από τα MIN_NORM,MAX_NORM).
Στη συνέχεια, υλοποιήθηκε μία δεύτερη συνάρτηση που παίρνει ως όρισμα ένα interp_t και επιστρέφει μία 31 bit τιμή που αντιστοιχεί σε αυτό το είδος αριθμού(χωρίς το sign.
Τέλος, σε ένα always_comb αρχικοποιούμε τις τιμές όλων των 1 bit εξόδων στο μηδέν και βρίσκουμε το είδος interp_t που αντιστοιχεί στα a,b, χρησιμοποιώντας τη num_interp. Έπειτα, ανάλογα με το συνδυασμό των

ειδών του a και b τοποθετούμε στο MSB του z το xor των sign των a και b και για τα υπόλοιπα bit του αποτελέσματος καλούμε τη z_num με όρισμα το είδος που προκύπτει από τον παραπάνω συνδυασμό. Ακόμη, ανεβάζουμε στο ένα το 1 bit output που αντιστοιχεί στην κάθε περίπτωση. Ιδιαίτερα για την περίπτωση που τα a, b είναι zero και infinity, ή αντίστροφα, το αποτέλεσμα θεωρείται infinity, ωστόσο θέτω στο 1 τόσο το inf_f και το nan_f. Αν έχω συνδυασμό NORM x NORM, αν έχω overflow προκύπτει MAX_NORM, αν έχω underflow προκύπτει MIN_NORM, αλλιώς το αποτέλεσμα είναι ίσο με το z_calc και το inexact_f γίνεται ίσο με το inexact της εισόδου.
Οι υπόλοιποι συνδυασμοί προκύπτουν από τον παρακάτω πίνακα.

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

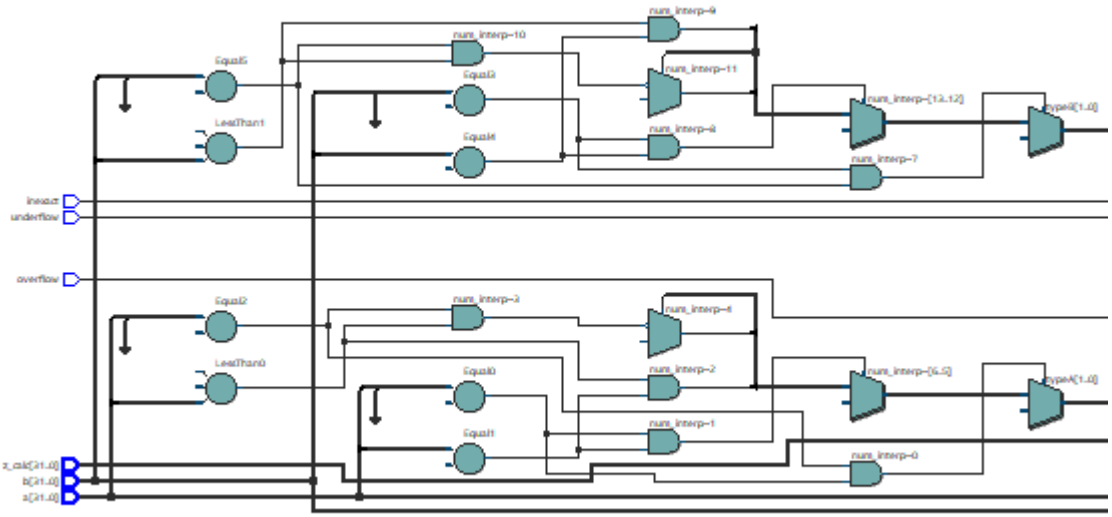
Κώδικας:

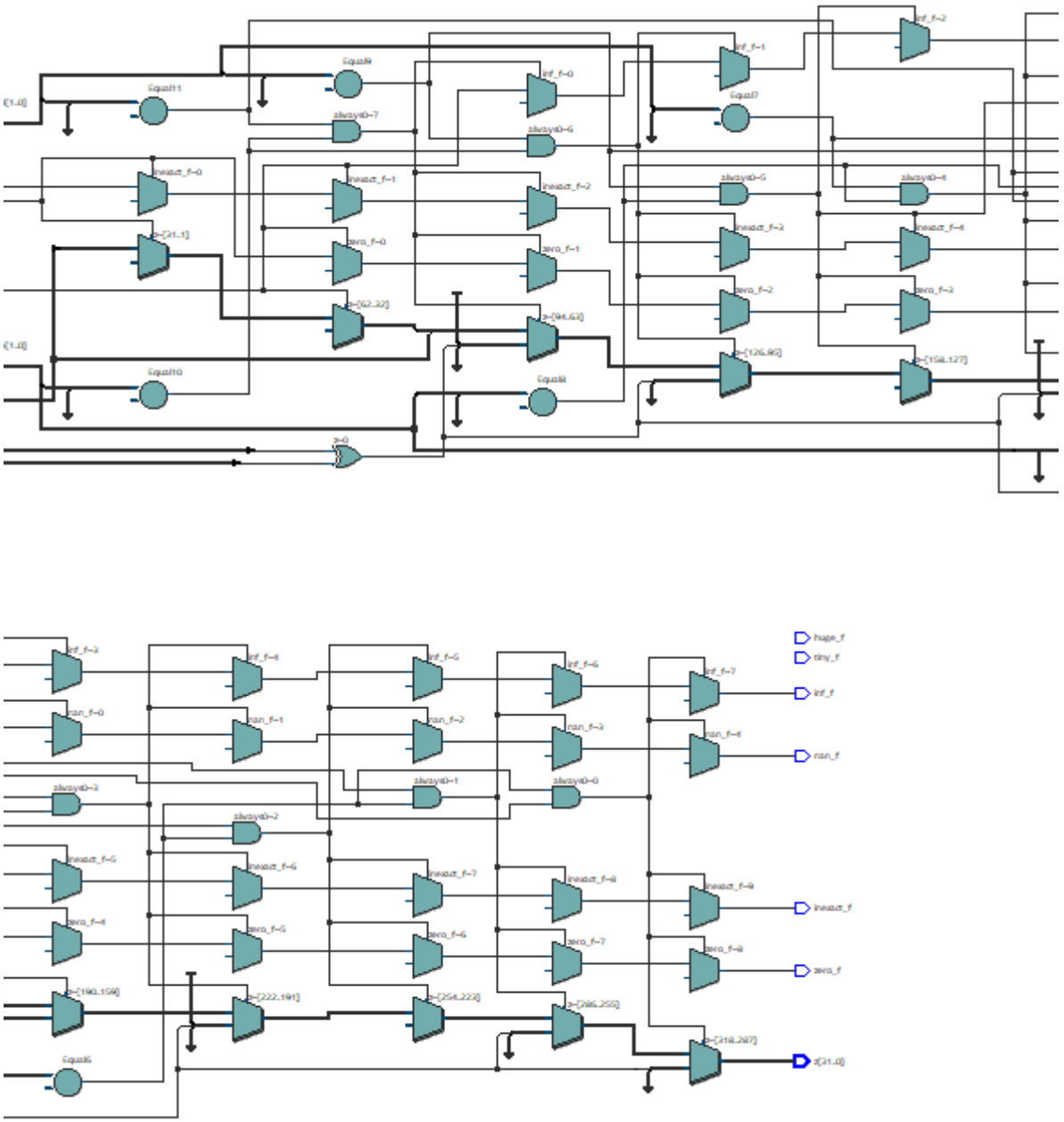
```
module exception_mult#(parameter round = "IEEE_pinf")(output logic [31:0] z,
                                                       output logic zero_f, inf_f, nan_f, tiny_f,
huge_f,inexact_f,
                                                       input logic [31:0] z_calc,a,b,
                                                       input logic overflow, underflow, inexact);

typedef enum {ZERO,INF,NORM,MIN_NORM,MAX_NORM} interp_t;
interp_t typeA, typeB;

function interp_t num_interp(logic [31:0] c);
        if (c[30:23] == 0 && c[22:0] == 0)
                return ZERO;
        else if(c[30:23] == 8'b11111111 && c[22:0] == 0)
                return INF;
        else if(c[30:23] == 8'b11111111 && c[22:0] > 0)
                return INF;
        else if(c[30:23] == 0 && c[22:0] > 0)
                return ZERO;
```

```
        else
                return NORM;
endfunction

function [30:0] z_num(interp_t num_type);
        unique case(num_type)
                ZERO: return 0;
              INF: return 31'b11111111000000000000000000000000;
  MIN_NORM: return 31'b00000001000000000000000000000000;
  MAX_NORM: return 31'b11111110111111111111111111111111;
        endcase
endfunction

always_comb begin
        zero_f = 0;
        inf_f = 0;
        nan_f = 0;
        tiny_f = 0;
        huge_f = 0;
        inexact_f = 0;
        typeA = num_interp(a);
        typeB = num_interp(b);
        if(typeA == ZERO && typeB == ZERO)
                begin
                        z[31] = a[31]^ b[31];
                        z[30:0] = z_num(ZERO);
                        zero_f = 1;
                end
        else if(typeA == ZERO && typeB == NORM)
                begin
                        z[31] = a[31] ^ b[31];
                        z[30:0] = z_num(ZERO);
                        zero_f = 1;
                end
        else if(typeA == ZERO && typeB == INF)
                begin
                        z[31] = 0;
                        z[30:0] = z_num(INF);
                        nan_f = 1;
                        inf_f = 1;
                end
        else if(typeA == INF && typeB == INF)
                begin
                        z[31] = a[31] ^ b[31];
                        z[30:0] = z_num(INF);
                        inf_f = 1;
                end
        else if(typeA == INF && typeB == NORM)
                begin
```

```
                        z[31] = a[31] ^ b[31];
                        z[30:0] = z_num(INF);
                        inf_f = 1;
                end
        else if(typeA == INF && typeB == ZERO)
                begin
                        z[31] = 0;
                        z[30:0] = z_num(INF);
                        nan_f = 1;
                        inf_f = 1;
                end
        else if(typeA == NORM && typeB == ZERO)
                begin
                        z[31] = a[31] ^ b[31];
                        z[30:0] = z_num(ZERO);
                        zero_f = 1;
                end
        else if(typeA == NORM && typeB == INF)
                begin
                        z[31] = a[31] ^ b[31];
                        z[30:0] = z_num(INF);
                        inf_f = 1;
                end
        else
                begin
                        if(overflow)
                                begin
                                        if(round == "IEEE_near" || round == "IEEE_zero" || round ==
"near_up")
                                                begin
                                                        z[31] = z_calc[31];
                                                        z[30:0] = z_num(MAX_NORM);
                                                        huge_f = 1;
                                                end
                                        else
                                                begin
                                                        z[31] = z_calc[31];
                                                        z[30:0] = z_num(INF);
                                                        inf_f = 1;
                                                end
                                end
                        else if(underflow)
                                begin
                                        if(round == "IEEE_near" || round == "IEEE_zero" || round ==
"near_up")
                                                begin
                                                        z[31] = z_calc[31];
                                                        z[30:0] = z_num(MIN_NORM);
                                                        tiny_f = 1;
```

```
                                        end
                                else
                                        begin
                                                z[31] = z_calc[31];
                                                z[30:0] = z_num(ZERO);
                                                zero_f = 1;
                                        end
                        end
                else
                        begin
                                z = z_calc;
                                inexact_f = inexact;
                        end
        end
end
endmodule
```





Εικόνα 4: exception handling module

2.Testbench

Στο αρχείο του testbench αρχικά ορίζουμε την παράμετρο round και αναφέρουμε ένα instance του wrapper module (fp_mult_top). Θέτουμε την περίοδο του clk σε 20ns με ένα always που κάνει toggle την τιμή του κάθε 10ns.
Σε ένα initial block, αρχικοποιούμε τα clk, a και b στο μηδέν, ενώ το rst στο 1.
Έπειτα από 50ns το rst γίνεται 0 και ξεκινάει να λειτουργεί το κύκλωμα δίνοντας τυχαίες τιμές στα a,b. Το τυχαίο assignment πραγματοποιείται άλλες τρεις φορές με delay 100ns μεταξύ τους. Αυτό αποτελεί το πρώτο μέρος του testbench.
Στο 2ο μέρος του testbench, ανά 100ns γίνονται assigned τιμές στα a,b με τέτοιο τρόπο ώστε να καλυφθούν όλοι οι πιθανοί συνδυασμοί θετικών και αρνητικών αριθμών τύπου zero, inf, nan, norm, denorm.
Σε ένα always lock με sensitivity list το z, ελέγχεται αν το z είναι ίσο με το αντίστοιχο αποτέλεσμα του function multiplication και εκτυπώνεται το ανάλογο μήνυμα.
Τέλος, υλοποιήθηκε ένα function που δέχεται ως όρισμα ένα string που περιγράφει ένα είδος αριθμού και επιστρέφει μία 32bit τιμή που αντιστοιχεί σε αυτό το είδος.

Κώδικας:

```
`timescale 1ns/1ps
module fp_tb;

logic [31:0] z;
logic [31:0] a;
logic [31:0] b;
logic [7:0] status;
bit clk, rst;

parameter round = "IEEE_pinf";

fp_mult_top #(round) fp1(.d_z(z),.d_a(a),.d_clk(clk),.d_rst(rst),.d_b(b),.d_status(status));

initial begin
	clk = 0;
	a = 0;
	b = 0;
	rst = 1;

	#50
	rst = 0;
```

```
        //first part
        a = $urandom();
        b = $urandom();
        #100
        a = $urandom();
        b = $urandom();
        #200
        a = $urandom();
        b = $urandom();
        #300
        a = $urandom();
        b = $urandom();

        //second part
        #400
        a = string_to_binary("neg_nan");
        b = string_to_binary("neg_nan");
        #500
        a = string_to_binary("neg_nan");
        b = string_to_binary("pos_nan");
        #600
        a = string_to_binary("neg_nan");
        b = string_to_binary("neg_inf");
        #700
        a = string_to_binary("neg_nan");
        b = string_to_binary("pos_inf");
        #800
        a = string_to_binary("neg_nan");
        b = string_to_binary("neg_norm");
        #900
        a = string_to_binary("neg_nan");
        b = string_to_binary("pos_norm");
        #1000
        a = string_to_binary("neg_nan");
        b = string_to_binary("neg_denorm");
        #1100
        a = string_to_binary("neg_nan");
        b = string_to_binary("pos_denorm");
        #1200
        a = string_to_binary("neg_nan");
        b = string_to_binary("neg_zero");
        #1300
        a = string_to_binary("neg_nan");
        b = string_to_binary("pos_zero");


        #1400
a = string_to_binary("pos_nan");
        b = string_to_binary("neg_nan");
        #1500
        a = string_to_binary("pos_nan");
        b = string_to_binary("pos_nan");
        #1600
        a = string_to_binary("pos_nan");
        b = string_to_binary("neg_inf");
        #1700
        a = string_to_binary("pos_nan");
        b = string_to_binary("pos_inf");
        #1800
        a = string_to_binary("pos_nan");
```

```
b = string_to_binary("neg_norm");
#1900
a = string_to_binary("pos_nan");
b = string_to_binary("pos_norm");
#2000
a = string_to_binary("pos_nan");
b = string_to_binary("neg_denorm");
#2100
a = string_to_binary("pos_nan");
b = string_to_binary("pos_denorm");
#2200
a = string_to_binary("pos_nan");
b = string_to_binary("neg_zero");
#2300
a = string_to_binary("pos_nan");
b = string_to_binary("pos_zero");


#2400
a = string_to_binary("neg_inf");
b = string_to_binary("neg_nan");
#2500
a = string_to_binary("neg_inf");
b = string_to_binary("pos_nan");
#2600
a = string_to_binary("neg_inf");
b = string_to_binary("neg_inf");
#2700
a = string_to_binary("neg_inf");
b = string_to_binary("pos_inf");
#2800
a = string_to_binary("neg_inf");
b = string_to_binary("neg_norm");
#2900
a = string_to_binary("neg_inf");
b = string_to_binary("pos_norm");
#3000
a = string_to_binary("neg_inf");
b = string_to_binary("neg_denorm");
#3100
a = string_to_binary("neg_inf");
b = string_to_binary("pos_denorm");
#3200
a = string_to_binary("neg_inf");
b = string_to_binary("neg_zero");
#3300
a = string_to_binary("neg_inf");
b = string_to_binary("pos_zero");


#3400
a = string_to_binary("pos_inf");
b = string_to_binary("neg_nan");
#3500
a = string_to_binary("pos_inf");
b = string_to_binary("pos_nan");
#3600
a = string_to_binary("pos_inf");
b = string_to_binary("neg_inf");
#3700
a = string_to_binary("pos_inf");
b = string_to_binary("pos_inf");
```

```
#3800
a = string_to_binary("pos_inf");
b = string_to_binary("neg_norm");
#3900
a = string_to_binary("pos_inf");
b = string_to_binary("pos_norm");
#4000
a = string_to_binary("pos_inf");
b = string_to_binary("neg_denorm");
#4100
a = string_to_binary("pos_inf");
b = string_to_binary("pos_denorm");
#4200
a = string_to_binary("pos_inf");
b = string_to_binary("neg_zero");
#4300
a = string_to_binary("pos_inf");
b = string_to_binary("pos_zero");


#4400
a = string_to_binary("neg_norm");
b = string_to_binary("neg_nan");
#4500
a = string_to_binary("neg_norm");
b = string_to_binary("pos_nan");
#4600
a = string_to_binary("neg_norm");
b = string_to_binary("neg_inf");
#4700
a = string_to_binary("neg_norm");
b = string_to_binary("pos_inf");
#4800
a = string_to_binary("neg_norm");
b = string_to_binary("neg_norm");
#4900
a = string_to_binary("neg_norm");
b = string_to_binary("pos_norm");
#5000
a = string_to_binary("neg_norm");
b = string_to_binary("neg_denorm");
#5100
a = string_to_binary("neg_norm");
b = string_to_binary("pos_denorm");
#5200
a = string_to_binary("neg_norm");
b = string_to_binary("neg_zero");
#5300
a = string_to_binary("neg_norm");
b = string_to_binary("pos_zero");


#5400
a = string_to_binary("pos_norm");
b = string_to_binary("neg_nan");
#5500
a = string_to_binary("pos_norm");
b = string_to_binary("pos_nan");
#5600
a = string_to_binary("pos_norm");
b = string_to_binary("neg_inf");
#5700
```

```
a = string_to_binary("pos_norm");
b = string_to_binary("pos_inf");
#5800
a = string_to_binary("pos_norm");
b = string_to_binary("neg_norm");
#5900
a = string_to_binary("pos_norm");
b = string_to_binary("pos_norm");
#6000
a = string_to_binary("pos_norm");
b = string_to_binary("neg_denorm");
#6100
a = string_to_binary("pos_norm");
b = string_to_binary("pos_denorm");
#6200
a = string_to_binary("pos_norm");
b = string_to_binary("neg_zero");
#6300
a = string_to_binary("pos_norm");
b = string_to_binary("pos_zero");


#6400
a = string_to_binary("neg_denorm");
b = string_to_binary("neg_nan");
#6500
a = string_to_binary("neg_denorm");
b = string_to_binary("pos_nan");
#6600
a = string_to_binary("neg_denorm");
b = string_to_binary("neg_inf");
#6700
a = string_to_binary("neg_denorm");
b = string_to_binary("pos_inf");
#6800
a = string_to_binary("neg_denorm");
b = string_to_binary("neg_norm");
#6900
a = string_to_binary("neg_denorm");
b = string_to_binary("pos_norm");
#7000
a = string_to_binary("neg_denorm");
b = string_to_binary("neg_denorm");
#7100
a = string_to_binary("neg_denorm");
b = string_to_binary("pos_denorm");
#7200
a = string_to_binary("neg_denorm");
b = string_to_binary("neg_zero");
#7300
a = string_to_binary("neg_denorm");
b = string_to_binary("pos_zero");


#7400
a = string_to_binary("pos_denorm");
b = string_to_binary("neg_nan");
#7500
a = string_to_binary("pos_denorm");
b = string_to_binary("pos_nan");
#7600
a = string_to_binary("pos_denorm");
```

```
b = string_to_binary("neg_inf");
#7700
a = string_to_binary("pos_denorm");
b = string_to_binary("pos_inf");
#7800
a = string_to_binary("pos_denorm");
b = string_to_binary("neg_norm");
#7900
a = string_to_binary("pos_denorm");
b = string_to_binary("pos_norm");
#8000
a = string_to_binary("pos_denorm");
b = string_to_binary("neg_denorm");
#8100
a = string_to_binary("pos_denorm");
b = string_to_binary("pos_denorm");
#8200
a = string_to_binary("pos_denorm");
b = string_to_binary("neg_zero");
#8300
a = string_to_binary("pos_denorm");
b = string_to_binary("pos_zero");


#8400
a = string_to_binary("neg_zero");
b = string_to_binary("neg_nan");
#8500
a = string_to_binary("neg_zero");
b = string_to_binary("pos_nan");
#8600
a = string_to_binary("neg_zero");
b = string_to_binary("neg_inf");
#8700
a = string_to_binary("neg_zero");
b = string_to_binary("pos_inf");
#8900
a = string_to_binary("neg_zero");
b = string_to_binary("neg_norm");
#9000
a = string_to_binary("neg_zero");
b = string_to_binary("pos_norm");
#9100
a = string_to_binary("neg_zero");
b = string_to_binary("neg_denorm");
#9200
a = string_to_binary("neg_zero");
b = string_to_binary("pos_denorm");
#9300
a = string_to_binary("neg_zero");
b = string_to_binary("neg_zero");
#9400
a = string_to_binary("neg_zero");
b = string_to_binary("pos_zero");


#9500
a = string_to_binary("pos_zero");
b = string_to_binary("neg_nan");
#9600
a = string_to_binary("pos_zero");
```

```
		b = string_to_binary("pos_nan");
		#9700
		a = string_to_binary("pos_zero");
		b = string_to_binary("neg_inf");
		#9800
		a = string_to_binary("pos_zero");
		b = string_to_binary("pos_inf");
		#9900
		a = string_to_binary("pos_zero");
		b = string_to_binary("neg_norm");
		#10000
		a = string_to_binary("pos_zero");
		b = string_to_binary("pos_norm");
		#10100
		a = string_to_binary("pos_zero");
		b = string_to_binary("neg_denorm");
		#10200
		a = string_to_binary("pos_zero");
		b = string_to_binary("pos_denorm");
		#10300
		a = string_to_binary("pos_zero");
		b = string_to_binary("neg_zero");
		#10400
		a = string_to_binary("pos_zero");
		b = string_to_binary("pos_zero");

		#20000 $finish;
end

always #10 clk = !clk;

always @(z)
begin
		if( z != multiplication(round,a,b))
			$display("Error: different results");
		else
			$display("Correct multiplication");
end
endmodule

function [31:0] string_to_binary(string corner_case);
		if(corner_case == "neg_nan")
			return 32'b11111111100000010000000000000000;
		else if(corner_case == "pos_nan")
			return 32'b01111111100000000000001000000000;
		else if(corner_case == "neg_inf")
			return 32'b11111111100000000000000000000000;
		else if(corner_case == "pos_inf")
			return 32'b01111111100000000000000000000000;
		else if(corner_case == "neg_norm")
			return 32'b10001101100000010000101010010100;
		else if(corner_case == "pos_norm")
			return 32'b00110010100001010001010100101010;
		else if(corner_case == "neg_denorm")
			return 32'b10000000000000010000000000000000;
		else if(corner_case == "pos_denorm")
			return 32'b00000000000000000000001000000000;
		else if(corner_case == "neg_zero")
			return 32'b10000000000000000000000000000000;
		else if(corner_case == "pos_zero")
			return 32'b00000000000000000000000000000000;
		else
```

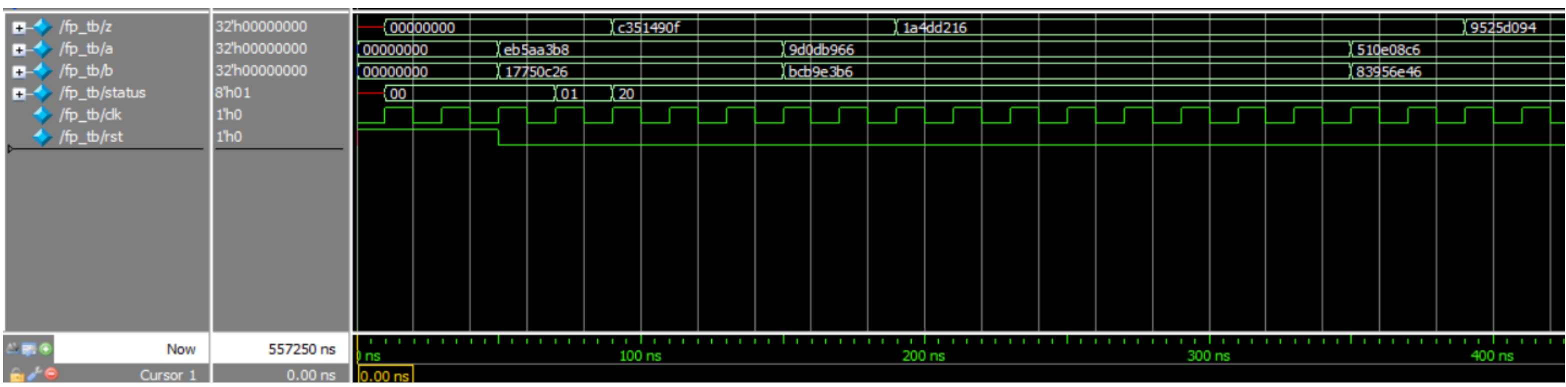
```
                return 0;
endfunction
```

IEEE_near:

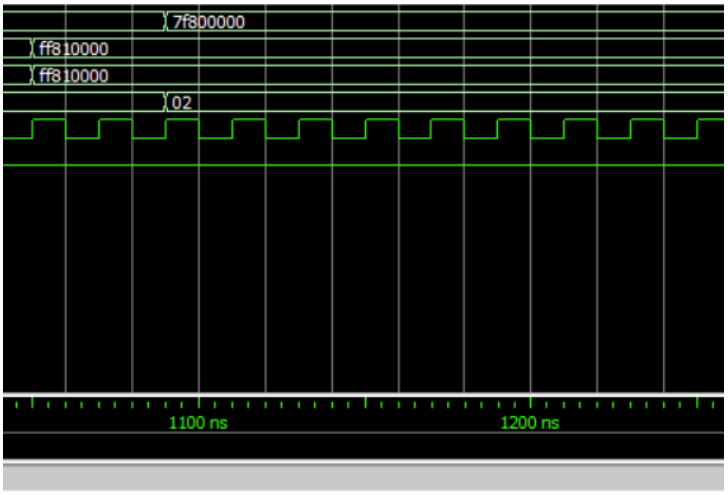
```
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
# Correct multiplication
```

Εικόνα 5: ενδεικτικό τμήμα transcript

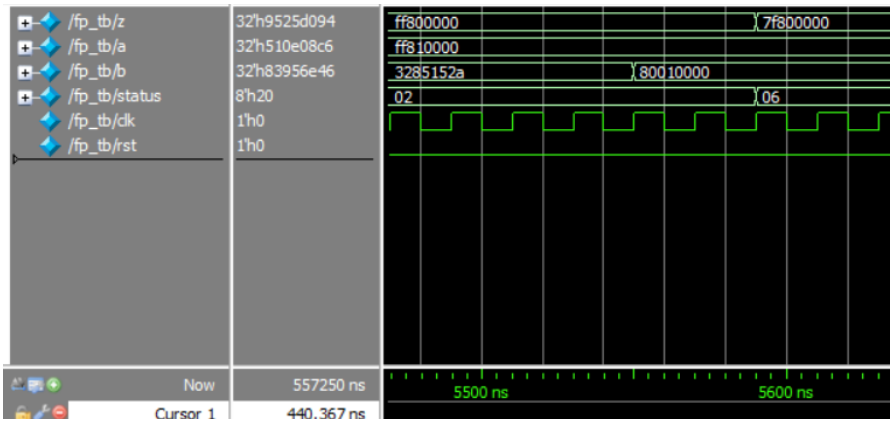
Σημείωση: Ένας από τους ελέγχους των πολλαπλασιασμών εκτύπωνε error, ωστόσο δεν κατάφερα να εντοπίσω το πρόβλημα. Το ίδιο φαινόμενο παρατηρείται σε κάθε περίπτωση round.



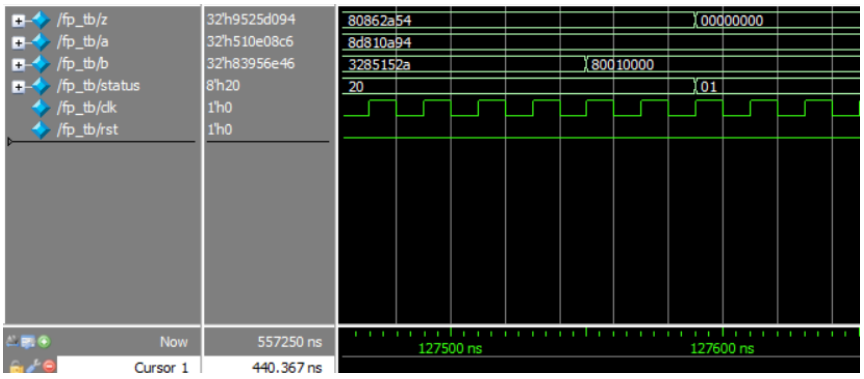
Εικόνα 6: Ενδεικτικά αποτελέσματα inexact



Εικόνα 7: Ενδεικτικό αποτέλεσμα infinity

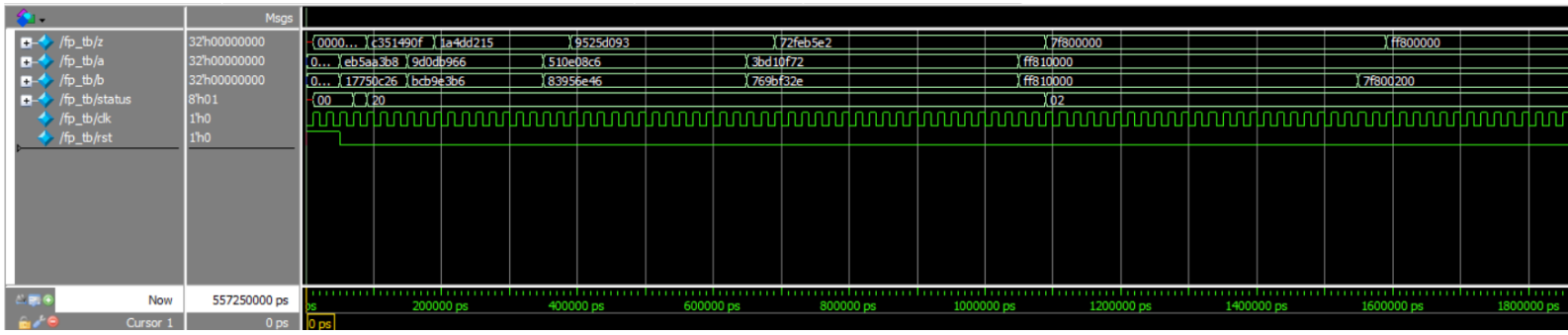


Εικόνα 8: Ενδεικτικό αποτέλεσμα nan



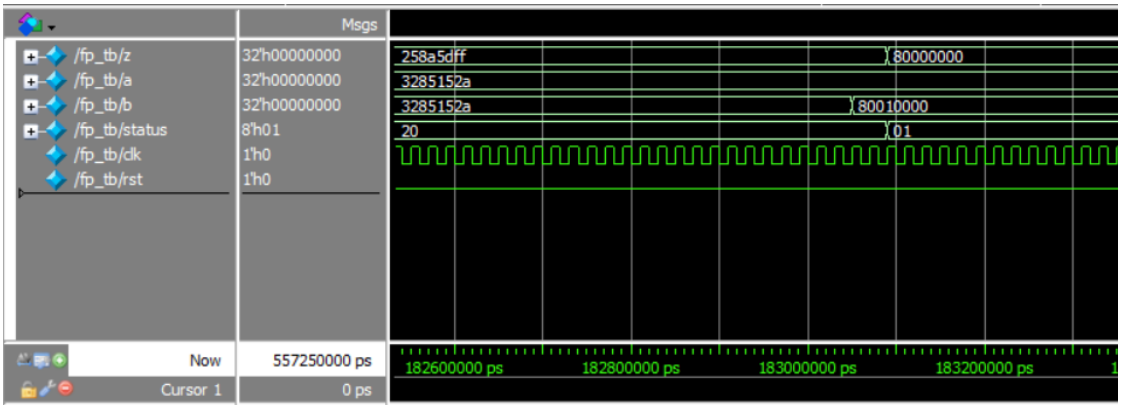
Εικόνα 9: ενδεικτικό αποτέλεσμα zero

IEEE_zero:



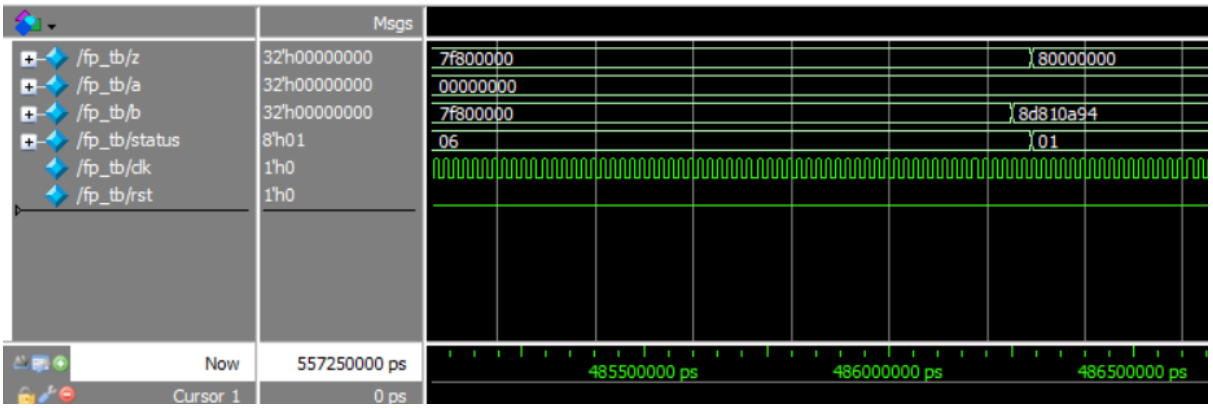
Εικόνα 10: Ενδεικτικό αποτέλεσμα για round = "IEEE_zero"

IEEE_pinf:



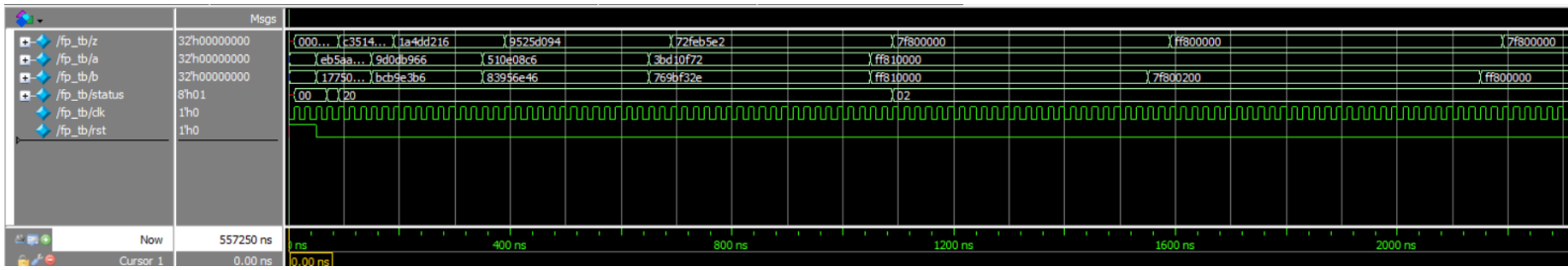
Εικόνα 11: Ενδεικτικό αποτέλεσμα για round = "IEEE_pinf"

IEEE_ninf:



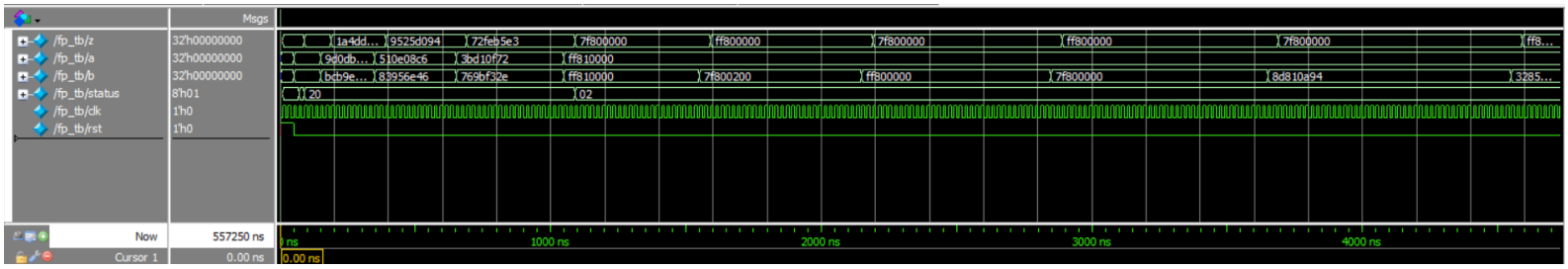
Εικόνα 12: Ενδεικτικό αποτέλεσμα για round = "IEEE_ninf"

near_up:



Εικόνα 12: Ενδεικτικό αποτέλεσμα για round = "near_up"

away_zero:



Εικόνα 13: Ενδεικτικό αποτέλεσμα για round = "away_zero"

3. Assertions

Το τρίτο μέρος χωρίζεται σε 2 επιμέρους τμήματα.
Το πρώτο τμήμα αφορά τα immediate assertions.
Δημιουργήθηκε ένα module test_status_bits το οποίο κάθε posedge clk ελέγχει αν υπάρχει μόνο 1 ή κανένα bit του status ίσο με ένα. Εξαίρεση αποτελεί η περίπτωση του nan, καθώς όπως αναφέρθηκε, όταν το nan_f είναι ίσο με 1, τότε γίνεται και το inf_f = 1.

Κώδικας:

```
module test_status_bits(input clk, rst,
                                        input [7:0] status,
                                        input [31:0] z, a, b);

always @(posedge clk)
        begin
                if(!rst) begin
```

```
                zeroANDinf: assert ((!status[0] && !status[1]) || (status[0]^status[1]))
$display("\n",$stime,,,"%m passed\n");
                else
                        $fatal("\n",$stime,,,"%m assert failed \n");
                zeroANDnan: assert ((!status[0] && !status[2]) || (status[0]^status[2]))
$display("\n",$stime,,,"%m passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
          zeroANDtiny: assert ((!status[0] && !status[3]) || (status[0]^status[3])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
          zeroANDhuge: assert ((!status[0] && !status[4]) || (status[0]^status[4])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
 zeroANDinexact: assert ((!status[0] && !status[5]) || (status[0]^status[5])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
         zeroANDunused: assert ((!status[0] && !status[6]) || (status[0]^status[6])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
zeroANDdivbytwo: assert ((!status[0] && !status[7]) || (status[0]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
           infANDtiny: assert ((!status[1] && !status[3]) || (status[1]^status[3])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
           infANDhuge: assert ((!status[1] && !status[4]) || (status[1]^status[4])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
         infANDinexact: assert ((!status[1] && !status[5]) || (status[1]^status[5])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
          infANDunused: assert ((!status[1] && !status[6]) || (status[1]^status[6])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
 infANDdivbytwo: assert ((!status[1] && !status[7]) || (status[1]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
```

```
            nanANDtiny: assert ((!status[2] && !status[3]) || (status[2]^status[3])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
            nanANDhuge: assert ((!status[2] && !status[4]) || (status[2]^status[4])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
          nanANDinexact: assert ((!status[2] && !status[5]) || (status[2]^status[5])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
           nanANDunused: assert ((!status[2] && !status[6]) || (status[2]^status[6])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
 nanANDdivbytwo: assert ((!status[2] && !status[7]) || (status[2]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
           tinyANDhuge: assert ((!status[3] && !status[4]) || (status[3]^status[4])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
 inexact tiny: assert ((!status[3] && !status[5]) || (status[3]^status[5])) $display("\n",$stime,,,"%m passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
          tinyANDunused: assert ((!status[3] && !status[6]) || (status[3]^status[6])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
tinyANDdivbytwo: assert ((!status[3] && !status[7]) || (status[3]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
 hugeANDinexact: assert ((!status[4] && !status[5]) || (status[4]^status[5])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
          hugeANDunused: assert ((!status[4] && !status[6]) || (status[4]^status[6]))
$display("\n",$stime,,,"%m passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
hugeANDdivbytwo: assert ((!status[4] && !status[7]) || (status[4]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                        else
                                $error("\n",$stime,,,"%m assert failed \n");
unusedANDinexact: assert ((!status[5] && !status[6]) || (status[5]^status[6])) $display("\n",$stime,,,"%m
passed\n");
```

```
                else
                        $error("\n",$stime,,,"%m assert failed \n");
inexactANDdivbytwo: assert ((!status[5] && !status[7]) || (status[5]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
unusedANDdivbytwo: assert ((!status[6] && !status[7]) || (status[6]^status[7])) $display("\n",$stime,,,"%m
passed\n");
                else
                        $error("\n",$stime,,,"%m assert failed \n");
                end
end

endmodule
```

Το δεύτερο τμήμα των assertions αφορά τα concurrent assertions. Υλοποιήθηκε ένα module test_status_z_combinations, που περιέχει 5 properties. Κάθε property ενεργοποιείται με posedge clk και με τη σειρά ελέγχουν αν:

- Όταν το zero είναι true, στον ίδιο κύκλο το exponent είναι 0.
- Όταν το inf είναι true, στον ίδιο κύκλο το exponent έχει όλα τα bit τουίσα με 1.
- Όταν το exponent του a έχει όλα τα bit του ίσα με το μηδέν και το exponent του b είναι 0, μετά από 2 κύκλους το nan είναι tru.
- Όταν το huge είναι true, στον ίδιο κύκλο το exponent του z έχει όλα τα bit ίσα με 1 ή όλα εκτός από το LSB.
- Όταν το tiny είναι true, στον ιδιο κύκλο το exponent του z έχει όλα τα bit ίσα με το 0 ή όλα εκτός από το LSB.

Στη συνέχεια γίνονται assert τα properties και εκτυπώνονται τα ανάλογα αποτελέσματα.

Κώδικας:

```
module test_status_z_combinations(input clk,rst,
                                                input [7:0] status,
                                                input [31:0] z, a, b);

property p1;
@(posedge clk) status[0] |-> (z[30:23] == '0);
endproperty

property p2;
@(posedge clk) status[1] |-> (z[30:23] == '1);
endproperty
```

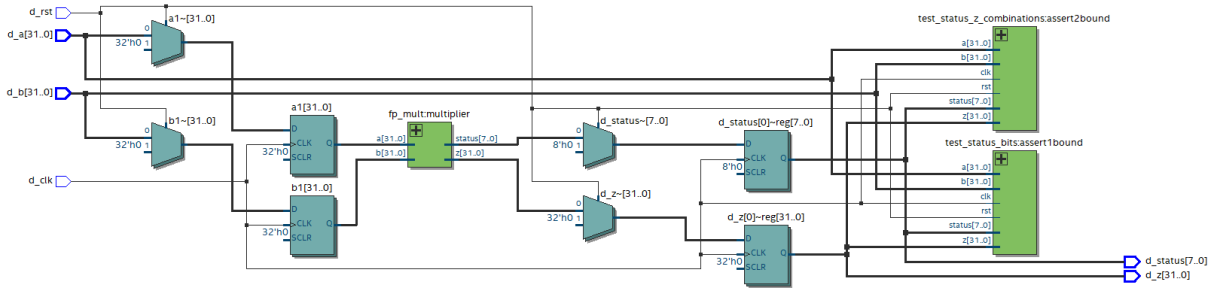
```
property p3;
@(posedge clk) ((a[30:23] == '0 && b[30:23] == '1) || (a[30:23] == '1 && b[30:23] == '0)) |-> ##2 (status[2] == '1);
endproperty

property p4;
@(posedge clk) status[4] |-> (z[30:23] == '1 || z[30:23] == 8'b11111110);
endproperty

property p5;
@(posedge clk) status[3] |-> (z[30:23] == '0 || z[30:23] == 8'b00000001);
endproperty

zero: assert property (p1) $display($stime,,,"PASS");
                           else $display ($stime,,,"FAIL");
 inf: assert property (p2) $display($stime,,,"PASS");
                           else $display ($stime,,,"FAIL");
 nan: assert property (p3) $display($stime,,,"PASS");
                           else $display ($stime,,,"FAIL");
huge: assert property (p4) $display($stime,,,"PASS");
                           else $display ($stime,,,"FAIL");
tiny: assert property (p5) $display($stime,,,"PASS");
                           else $display ($stime,,,"FAIL");

endmodule
```

Σημείωση: Δεν κατάφερα να κάνω το bind μεταξύ αυτών των module και του wrapper διότι το quartus εμφάνιζε error κατα το compilation. Εν τέλει τα πρόσθεσα ως instances στο wapper(fp_mult_top) και λειτουργουν ως κανονικά modules. Ωστόσο, τα αποτελέσματα εκτυπώνονται στο transcript.

Κώδικας fp_mult_top:

```
//This module is given for the exercises
module fp_mult_top #(parameter round = "near_up")(
    d_clk, d_rst, d_a, d_b, d_z, d_status
);

   input logic [31:0] d_a, d_b;  // Floating-Point numbers
   output logic [31:0] d_z;    // a ± b
   output logic [7:0] d_status;  // Status Flags
   input logic d_clk, d_rst;

   logic [31:0] a1, b1;  // Floating-Point numbers
   logic [31:0] z1;    // a ± b
   logic [7:0] status1;  // Status Flags

   fp_mult #(round) multiplier(z1,status1,a1,b1);

            test_status_bits assert1bound(d_clk,d_rst,d_status,d_z,d_a,d_b);
            test_status_z_combinations assert2bound(d_clk,d_rst,d_status,d_z,d_a,d_b);
```

```
  always @(posedge d_clk)
    if (d_rst == 1)
      begin
        a1 <= '0;
        b1 <= '0;
        d_z <= '0;
        d_status <= '0;
      end
    else
      begin
        a1 <= d_a;
        b1 <= d_b;
        d_z <= z1;
        d_status <= status1;
      end

endmodule
```



Εικόνα 14: wrapper module

```
#  556010000  fp_tb.fp1.assert1bound.infANDinexact passed
#
#
#  556010000  fp_tb.fp1.assert1bound.infANDunused passed
#
#
#  556010000  fp_tb.fp1.assert1bound.infANDdivbytwo passed
#
#
#  556010000  fp_tb.fp1.assert1bound.nanANDtiny passed
#
#
#  556010000  fp_tb.fp1.assert1bound.nanANDhuge passed
#
#
#  556010000  fp_tb.fp1.assert1bound.nanANDinexact passed
#
#
#  556010000  fp_tb.fp1.assert1bound.nanANDunused passed
#
#
#  556010000  fp_tb.fp1.assert1bound.nanANDdivbytwo passed
#
#
#  556010000  fp_tb.fp1.assert1bound.tinyANDhuge passed
#
#
#  556010000  fp_tb.fp1.assert1bound.inexactANDtiny passed
#
#
#  556010000  fp_tb.fp1.assert1bound.tinyANDunused passed
#
```

Εικόνα 15: Ενδεικτικό τμήμα του transcript για το verification του 1ου τμήματος των assertions

```
#  471890000  p2 PASS
#  471910000  p3 PASS
#  471910000  p2 PASS
#  471930000  p3 PASS
#  471930000  p2 PASS
#  471950000  p3 PASS
#  471950000  p2 PASS
#  471970000  p3 PASS
#  471970000  p2 PASS
```

Εικόνα 16: Ενδεικτικό τμήμα του transcript για το verification του 2ου τμήματος των assertions, properties p2,p3.

```
#  497610000   p1 PASS
#  497630000   p1 PASS
#  497650000   p1 PASS
#  497670000   p1 PASS
#  497690000   p1 PASS
#  497710000   p1 PASS
#  497730000   p1 PASS
#  497750000   p1 PASS
#  497770000   p1 PASS
#  497790000   p1 PASS
#  497810000   p1 PASS
```

Εικόνα 16: Ενδεικτικό τμήμα του transcript για το verification του 2ου τμήματος των assertions, property p1.