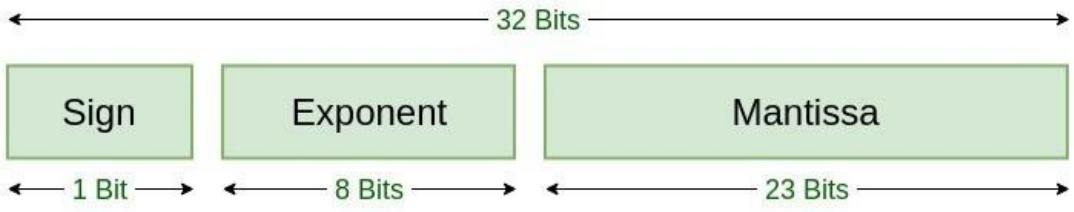
Εισαγωγή:

Floating Point είναι οι αριθμοί με δεκαδικά ψηφία, στους οποίους δεν υπάρχει σταθερό πλήθος ψηφίων πριν ή μετά την υποδιαστολή. Η χρήση τέτοιου είδους αριθμών είναι αρκετά συχνή στους υπολογιστές και για αυτό το λόγο δημιουργήθηκε το πρότυπο IEEE 754, έτσι ώστε να υπάρχει αρμονία στην αναπαράσταση αυτών των αριθμών μεταξύ των διάφορων υλοποιήσεων από κατασκευαστή σε κατασκευαστή.

Η αναπαράσταση των floating point αριθμών στο πρότυπο αυτό γίνεται με βάση την επιστημονική σημειογραφία. Δηλαδή να υπάρχει μόνο ένα ψηφίο αριστερά της υποδιαστολής και αυτό να μην είναι μηδέν. Τον κάθε αριθμό τον χαρακτηρίζουν το πρόσημο, η μάντισσα (fraction), η βάση και ο εκθέτης. Χειριζόμενοι δυαδικούς αριθμούς έχουμε σαν βάση το 2.

Στη Single-Precision αναπαράσταση χρησιμοποιούμε 32-bit για να περιγράψουμε τον floating point αριθμό. Από αυτά το 31ο (MSB) είναι το πρόσημο (1 = - και 0 = +), τα επόμενα 8-bit αναπαριστούν τον εκθέτη (30ο έως 23ο) ο οποίος έχει bias και παίρνει τιμές από -126 έως 127 για αριθμούς single-precision, και τα τελευταία 23-bit αποτελούν τη mantissa του αριθμού.



Εικόνα 1: Αναπαράσταση Single-Precision Floating Point αριθμού

Είναι σημαντικό να σημειώσουμε ότι το πρότυπο IEEE 754, αντιμετωπίζει τους παραπάνω vectors με τρόπο ώστε να μην χρειάζεται να επισημαίνεται κάθε φορά πληροφορία που εννοείται, και προσπαθεί να απλοποιεί τη διαδικασία για τυχόν συγκρίσεις μεταξύ αριθμών. Πιο συγκεκριμένα, η mantissa περιέχει ένα «κρυφό» bit που εννοείται ότι έχει την τιμή '1' για τους normal αριθμούς (για τους οποίους ανταποκρίνεται και η εργασία μας) και '0' όταν πρόκειται για subnormal αριθμούς και επίσης ο εκθέτης δεν αναπαρίσταται απευθείας, αλλά προστίθεται μια μεροληψία (bias) έτσι ώστε ο μικρότερος αναπαραστάσιμος εκθέτης να παριστάνεται ως 1. Έτσι η γενική μορφή ενός floating point single precision αριθμού σύμφωνα με το IEEE 754 πρότυπο είναι η παρακάτω.

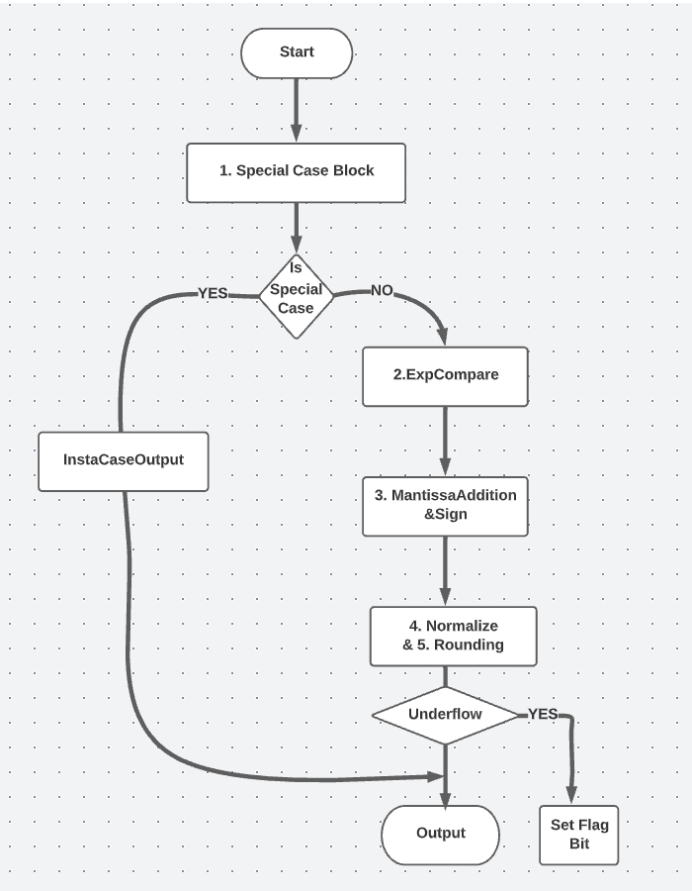
$$(-1)^{Sign} \times (1 + Fraction) \times 2^{(Exponent-127)}$$

Αλγόριθμός:

Τώρα όσον αφορά τη διαδικασία πρόσθεσης floating point αριθμών, το πρότυπο ακολουθεί ένα συγκεκριμένο αλγόριθμό και υλοποιεί ορισμένα βήματα ώστε να φτάσει στο αποτέλεσμα. Αρχικά να σημειώσουμε ότι για την κατανόηση της λειτουργίας των βημάτων και γενικά του αλγορίθμου, συμβουλευτήκαμε το βιβλίο Computer Organization and Design των John L. Hennessy, David A. Patterson όπως επίσης και το πρότυπο IEEE Standard για Floating-Point Arithmetic.

Με την κατανόηση αυτών, αλλάξαμε/τροποποιήσαμε ορισμένα σημεία στον αλγόριθμο και στα βήματα ώστε να αντιστοιχούν καλύτερα στην υλοποίηση και στην προσέγγιση που ακολουθήσαμε.

Παρακάτω φαίνεται ένα flow diagram που περιέχει τα βήματα ροής του αλγορίθμου που υλοποιεί το σύστημά μας, τα οποία θα αναλυθούν παρακάτω. Διαφέρει ελάχιστα από το προτεινόμενο στο σύγγραμμα που συμβουλευτήκαμε καθώς είναι λίγο διαφορετική η υλοποίηση που ακολουθήσαμε και κάθε αλλαγή θα τεκμηριωθεί παρακάτω στις επιμέρους υπο-μονάδες που συντελούν στο συνολικό κύκλωμα.



Εικόνα 2: Flow Diagram αλγορίθμου υλοποίησης

Το κανονικό μονοπάτι εκτελεί όλα τα βήματα από το 1 έως το 5, σε τυχόν ειδική περίπτωση όμως, μετά το πρώτο βήμα υπολογίζεται κατευθείαν η έξοδος και παραλείπονται τα ενδιάμεσα βήματα. Τα βήματα εξηγούνται παρακάτω:

- **1ο βήμα: Έλεγχος εισόδων και αναγνώριση ειδικών περιπτώσεων**

Σε αυτό το βήμα ελέγχονται οι δυο αριθμοί που εισέρχονται στη μονάδα για τυχόν ειδικές περιπτώσεις, στις οποίες το αποτέλεσμα μπορεί να υπολογιστεί κατευθείαν χωρίς να χρειαστεί να περάσουν από όλο το data path τα δεδομένα. Με αυτό το βήμα εξοικονομούμε πόρους και ενέργεια.

- **2ο βήμα**: **Σύγκριση εκθετών για την ευθυγράμμιση των αριθμών πριν την πρόσθεση**

Αυτό το βήμα συγκρίνει τους εκθέτες των δυο αριθμών που εισάγονται, ώστε να καθορίσει ποιος είναι μικρότερος, αφαιρεί τον μικρότερο από τον μεγαλύτερο και υπολογίζει έτσι πόσα bits πρέπει να μετατεθεί η mantissa του αριθμού με τον μικρότερο εκθέτη ώστε να είναι στην ίδια τάξη μεγέθους και να μπορούν να προστεθούν.

- **3ο βήμα: Άθροιση των δυο significands**

Εδώ αθροίζονται οι significands των δυο αριθμών και υπολογίζεται επίσης το πρόσημο που θα έχει ο αριθμός στη τελική έξοδο από το κύκλωμα.

- **4ο βήμα: Κανονικοποίηση του αθροίσματος**

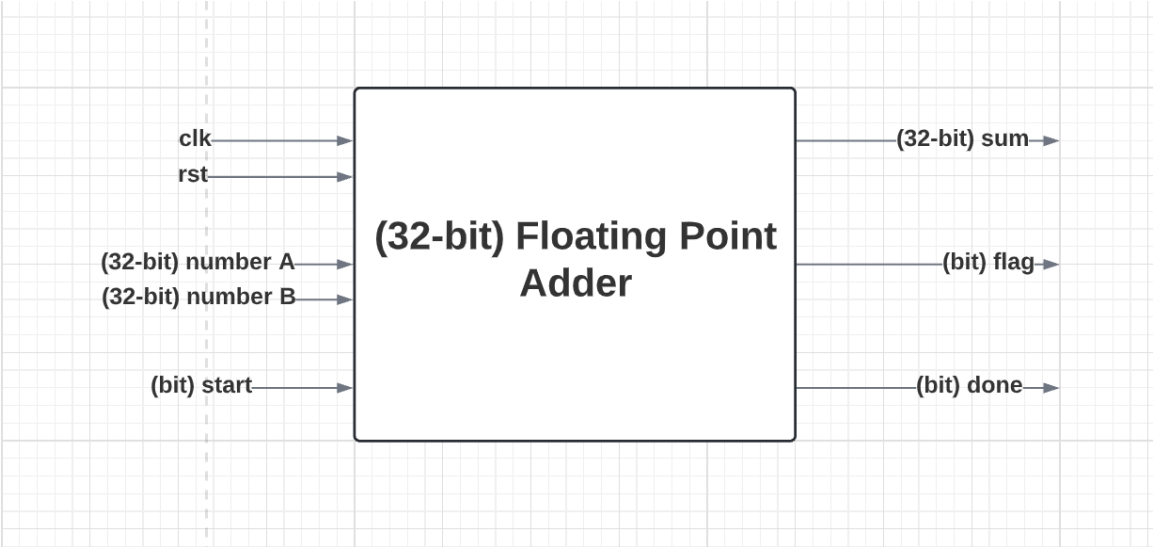
Σε αυτό το βήμα πραγματοποιείται η κανονικοποίηση του παραπάνω αθροίσματος και του μεγαλύτερου εκθέτη ώστε να αναπαρίστανται βάσει της σημειογραφίας του προτύπου που αναφέραμε παραπάνω.

- **5ο βήμα: Στρογγυλοποίηση του significand**

Εδώ γίνεται στρογγυλοποίηση στη mantissa που προκύπτει ως άθροισμα των δυο, ώστε να κρατείται μια ακρίβεια στον αριθμό των bits.

<u>Υλοποίηση</u>:

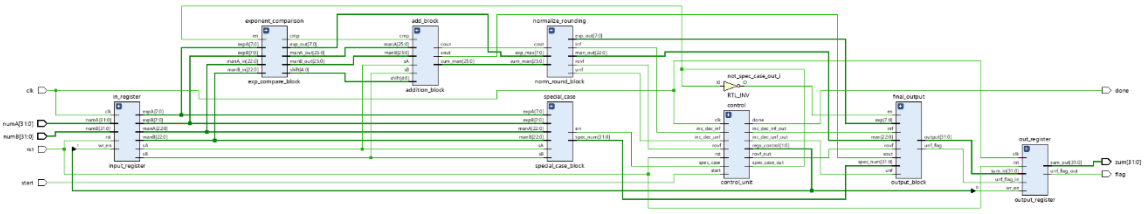
Το συνολικό μπλοκ της υλοποίησής μας φαίνεται παρακάτω και το εσωτερικό του θα αναλυθεί στις επόμενες σελίδες. Εν συντομία αποτελείται από επιμέρους υπο-μονάδες, που εκτελούν συγκεκριμένες λειτουργίες βάσει του αλγορίθμου, οι οποίες βρίσκονται ανάμεσα σε δυο καταχωρητές (εισόδου και εξόδου) που αποθηκεύουν τα αντίστοιχα δεδομένα. Το κύκλωμά μας παίρνει σαν εισόδους τους δυο 32-bit αριθμούς **numA**, **numB** που θέλουμε να προσθέσουμε, ένα σήμα ρολογιού **clk** και ένα σήμα **reset** (ασύγχρονο), που χρονίζουν και αρχικοποιούν το σύστημά μας αντίστοιχα, και ένα σήμα **start** το οποίο καθορίζει την αρχή της λειτουργίας του. Σαν έξοδοι βγαίνουν το 32-bit άθροισμα **sum** των δυο αριθμών, μετά την όποια επεξεργασία έχει δεχθεί, μια σημαία **flag** ενός bit που δηλώνει την ύπαρξη τυχόν *underflow* (περίπτωση όπου ένας αρνητικός εκθέτης γίνεται πολύ μεγάλος και δε χωράει πλέον στον 8-bit vector του εκθέτη) , που μπορεί να προκύψει στις εσωτερικές υπο-μονάδες του υπο-κυκλώματος, και ένα σήμα **done** το οποίο δηλώνει το τέλος της επεξεργασίας δεδομένων στη μονάδα και μπορεί να ενημερώνει μια τυχόν επόμενη κυκλωματική μονάδα για το ότι η έξοδος είναι έτοιμη και μπορεί να αναγνωστεί από τον καταχωρητή που βρίσκεται στην έξοδο του κυκλώματος.



Εικόνα 3: (32-bit) Floating Point Adder Block

Παρακάτω φαίνεται το εσωτερικό του παραπάνω μπλοκ με όλες τις υπο-μονάδες και τους καταχωρητές εισόδου και εξόδου. Το σχήμα αυτό παράχθηκε από το πρόγραμμα Vivado κατά την RTL ανάλυση του συστήματος και φαίνεται πώς διασυνδέονται όλες οι επιμέρους υπο-μονάδες μεταξύ τους. Επίσης μπορεί να διακριθεί το data path του συστήματος, δηλαδή ότι τα δεδομένα εισέρχονται στον καταχωρητή εισόδου, από εκεί περνούν στο **Special Case block**, στη συνέχεια (όχι ειδική περίπτωση) στο **Exponent Compare block**, από εκεί στο **Addition block**, μετά στο **Normalization and Rounding**

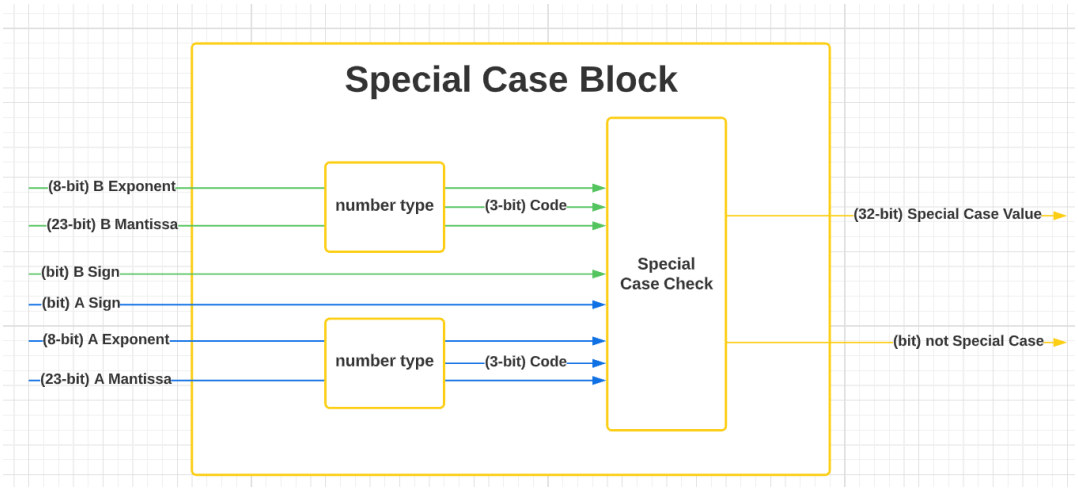
block, ύστερα στο **Output block**, και τέλος εισέρχονται στο καταχωρητή εξόδου από όπου διαβάζονται.



Εικόνα 4: (32-bit) Floating Point Adder Schematic

Οι υπο-μονάδες που αποτελούν το κύκλωμα είναι έξι συν τους δυο καταχωρητές στην είσοδο και στην έξοδο, αντιστοιχούν στις κύριες διεργασίες που εκτελεί το σύστημα βάσει και των βημάτων του αλγορίθμου που ακολουθήσαμε, και είναι οι ακόλουθες. Το Special Case block, το Exponent Compare block, το Addition block, το Normalization and Rounding block, το Output block και το Control Unit. Οι υπο-μονάδες αναλύονται παρακάτω.

Ακολουθούμε τη φυσιολογική ροή του data path και ξεκινάμε από το Special Case block.



Εικόνα 5: Special Case Block

Το παραπάνω μπλοκ πραγματοποιεί τις ακόλουθες λειτουργίες. Πρώτον παρατηρούμε ότι περιέχει δυο υπο-μονάδες **Number Type**, η κάθε μια από αυτές δέχεται σαν είσοδο τα επιμέρους στοιχεία (εκθέτης, mantissa) των δυο αριθμών εισόδου και ανάλογα με αυτά καθορίζει το τύπο του κάθε αριθμού και παράγει ένα κωδικό για τον κάθε τύπο (με χρήση πολυπλεκτών και συγκριτών). Παρακάτω φαίνεται ένας πίνακας που εξηγεί

πώς προκύπτουν οι διάφοροι τύποι και το κομμάτι κώδικα VHDL που περιγράφει τη παραπάνω υπο-μονάδα.

Exponent	Mantissa	Object
0	0	Zero
0	Nonzero	Denormalized number*
1-254	Anything	+/- FP number
255	0	+ / - infinity
255	Nonzero	NaN

Εικόνα 6: IEEE encoding of floating-point numbers

```
--unit that checks input number's type
--(normal, subnormal, zero, infinity, Nan)

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity num_type is
    port(exp: in std_logic_vector(7 downto 0);       --exponent of input
         man: in std_logic_vector(22 downto 0);      --mantissa of input
         in_type: out std_logic_vector(2 downto 0)   --type of input
    );
end num_type;

architecture num_type_arch of num_type is
begin
    in_type <= "000" when exp="00000000" and man="00000000000000000000000" else   --zero
               "001" when exp="00000000" and man/="00000000000000000000000" else  --subnormal
               "010" when exp>"00000000" and exp<"11111111" else                  --normal
               "101" when exp="11111111" and man="00000000000000000000000" else   --infinity
               "100" when exp="11111111" and man/="00000000000000000000000" else  --not a number
               "100";                                                             --every other type (X,Z,-,...) is considered a NaN
end num_type_arch;
```

Εικόνα 7: VHDL κώδικας της υπο-μονάδας που βρίσκει το τύπο του αριθμού που εισάγεται

Στη συνέχεια το επόμενο υπο-μπλοκ, **Special Case Check** λαμβάνει σαν εισόδους τους κωδικούς των δυο αριθμών εισόδου, που παράγονται από τα **Number Type** μπλοκ, τα πρόσημα τους, τους εκθέτες και τις mantissas και ελέγχει για τυχόν ειδικές περιπτώσεις. Ανάλογα έχει σαν έξοδο μια τιμή *Special Case Value (32-bit)* η οποία παράγεται σε τυχόν ειδική περίπτωση και αποτελεί την τιμή του αποτελέσματος. Σε περίπτωση που δεν έχουμε ειδική περίπτωση (κανονική λειτουργία) ο αριθμός αυτός είναι 'X'. Επίσης παράγεται το σήμα εξόδου *not Special Case* το οποίο είναι '0' σε ειδική περίπτωση και '1' στη κανονική λειτουργία. Το σήμα αυτό περνάει στο επόμενο μπλοκ στο data path σαν σήμα enable έτσι ώστε να μην λειτουργούν οι επόμενες μονάδες όταν προκύπτει ειδική περίπτωση, που ξέρουμε απευθείας το αποτέλεσμα, ώστε να εξοικονομείται ενέργεια και χρόνος στο σύστημά μας.

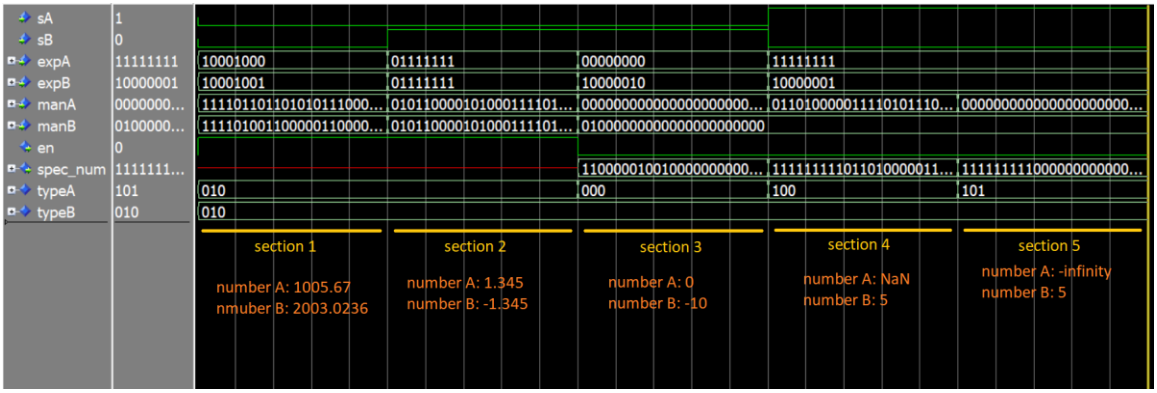
Εδώ πρέπει να σημειώσουμε πως στην υλοποίησή μας χειριζόμαστε μόνο normal (όχι και subnormal) floating-point αριθμούς και τις υπόλοιπες περιπτώσεις τις χειριζόμαστε σαν ειδικές. Παρακάτω φαίνονται ενδεικτικά σημεία από το κώδικα VHDL που περιγράφει το **Special Case Check** μπλοκ και ένας πίνακας που απεικονίζει τις διάφορες περιπτώσεις.

```
--both A and B are normals--
if ((typeA and typeB)="010") then  --normal case, normal operation
        enable := '1';
        sO := 'X';
        expO := "XXXXXXXX";
        manO := "XXXXXXXXXXXXXXXXXXXXXXX";
else
    enable := '0';
    --one is subnormal--
    if (typeA="001") then
        sO := sB;
        expO := expB;
        manO := manB;
    elsif (typeB="001") then
        sO := sA;
        expO := expA;
        manO := manA;

    --one is NaN--
    elsif (typeA="100") then
        sO :=sA;
        expO := expA;
        manO := manA;
    elsif (typeB="100") then
        sO := sB;
        expO := expB;
        manO := manB;

    --one is infinity--
    elsif (typeA="101" and typeB(2)='0') then
        sO := sA;
        expO := expA;
        manO := manA;
    elsif (typeA(2)='0' and typeB="101") then
        sO := sB;
        expO := expB;
        manO := manB;

    --both are infity
    elsif ((typeA and typeB)="101") then
        if(sA/=sB) then --different signs
            sO := '0';
            expO := "11111111";
            manO := "00000000000000000000001";
        else --same signs
            sO := sA;
            expO := expA;
            manO := manA;
        end if;
```

Εικόνα 8: Ενδεικτικά σημεία από τον VHDL κώδικα που περιγράφει το Special Case Check μπλοκ

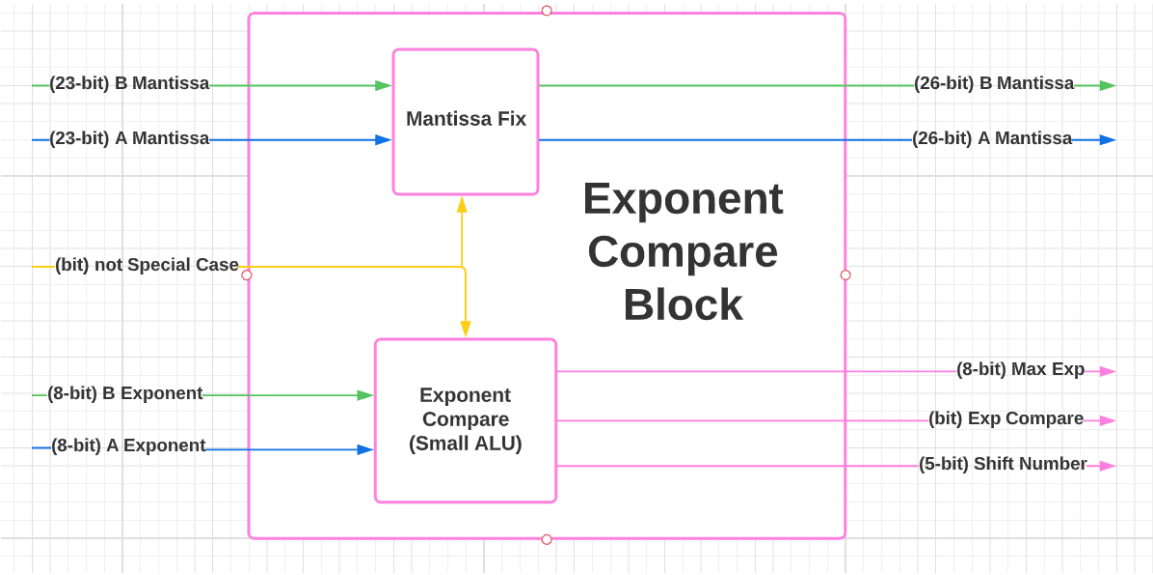
Case	Sign	A	B	Spec Num	En
Normal+Normal	X	Normal	Normal	A+B	1
Normal+SubNormal	X	Normal	Subnormal	A	0
Subnormal+Subnormal	X	Subnormal	Normal	B	0
Normal/Sub/Zero/Infinite+Infinite	SA=SB	x	Infinite	Infinite	0
Infinite+Normal/Sub/zero/Infinite	SA=SB	Infinite	x	Infinite	0
Infinite + Infinite	SA!=SB	Infinite	Infinite	NaN	0
NaN+x	x	NaN	x	NaN	0
Normal/SubNormal+0	x	Normal	0	A	0
0 + Normal/Subnormal	x	0	Normal	B	0

Παρακάτω φαίνεται μια ενδεικτική προσομοίωση του Special Case Block.



Εικόνα 9: Προσομοίωση του Special Case Block

Η παραπάνω προσομοίωση όπως και όσες ακολουθήσουν είναι χωρισμένες σε section για ευκολία στη κατανόησή τους. Ξεκινώντας από το section 1, βάζουμε σαν είσοδο στο μπλοκ το πρόσημο, τον εκθέτη και τη mantissa των δυο αριθμών που αναγράφονται κάτω από το section, όπως παρατηρούμε δεν πρόκειται για ειδική περίπτωση καθώς και οι δύο είναι κανονικοί αριθμοί (κωδικό "010") οπότε δεν παράγεται ειδικός αριθμός και το σήμα **en** που δηλώνει την τυχόν μη-ειδική περίπτωση είναι high. Στο section 2 είναι αντίστοιχη περίπτωση. Στο section 3, παρατηρούμε ότι ο ένας αριθμός είναι μηδέν (κωδικός "000") και ο άλλος -10, πρόκειται για ειδική περίπτωση καθώς το άθροισμά τους βγαίνει κατευθείαν και έτσι παράγεται ο ειδικός αριθμός αποτελέσματος, με τιμή -10 προφανώς και το σήμα **en** πέφτει. Τα section 4 και 5 δείχνει επίσης ειδικές περιπτώσεις, μία με είσοδο NaN (not a number) και μία με είσοδο μείον άπειρο αντίστοιχα. Οι έξοδοι είναι ανάλογες και διαπιστώνεται η ορθή λειτουργία του μπλοκ.



Εικόνα 10: Exponent Compare Block

Επόμενο μπλοκ που αναλύουμε είναι το **Exponent Compare Block** που φαίνεται παραπάνω. Πρόκειται για το δεύτερο μπλοκ στη σειρά του data path των ενδιάμεσων υπο-μπλοκ και εκτελεί τις εξής λειτουργίες. Αρχικά παρατηρούμε ότι πρόκειται για δυο υπο-μονάδες που λειτουργούν παράλληλα και έχουν και οι δυο ένα κοινό σήμα εισόδου *not Special Case* το οποίο το λαμβάνουν από το προηγούμενο μπλοκ και λειτουργεί σαν σήμα enable όπως προαναφέραμε. Η **Mantissa Fix** υπο-μονάδα απλά συνενώνει κάποια bits στους vectors των δυο mantissas. Τοποθετεί σαν MSB(most significant bit) το implicit bit, που εννοείται ότι είναι '1' για όλους τους normal αριθμούς και χρειάζεται στη συνέχεια στο στάδιο της άθροισης. Επίσης τοποθετεί δυο bits (guard και round bit = "00") στις LSB θέσεις τα οποία χρειάζονται το στάδιο της στρογγυλοποίησης. Η **Small ALU** υπο-μονάδα εκτελεί αρχικά σύγκριση των δυο εκθετών και μετά, αφαίρεση του μικρότερου εκθέτη από το μεγαλύτερο ώστε να υπολογιστεί ο αριθμός των θέσεων που πρέπει να μετατοπιστεί η mantissa του αριθμού με το μικρότερο εκθέτη, ώστε να είναι ευθυγραμμισμένες. Επίσης, βγάζει σαν έξοδο τον μεγαλύτερο από τους δυο εκθέτες και ένα σήμα *Exponent Compare* το οποίο χρειάζεται σε επόμενο στάδιο και δηλώνει ποιος εκθέτης είναι μεγαλύτερος.

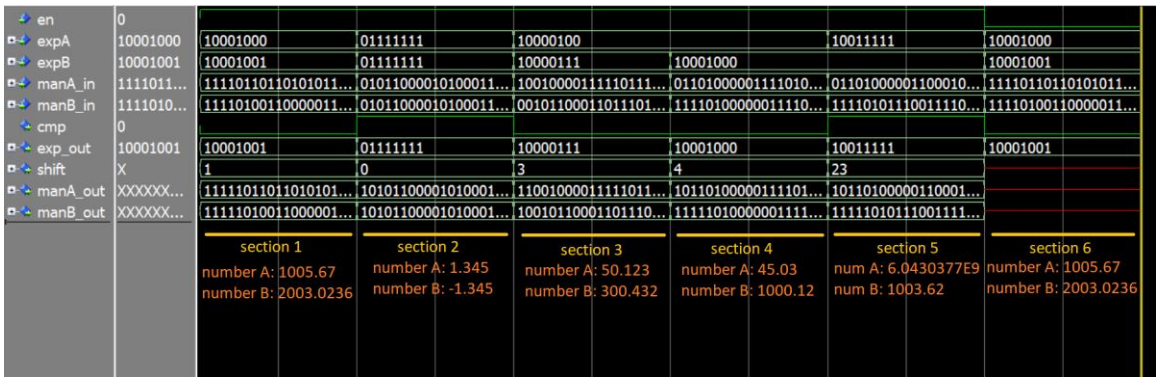
Παρακάτω φαίνονται κομμάτια από τον VHDL κώδικα των δυο υπο-μονάδων και μια ενδεικτική προσομοίωση ολόκληρου του **Exponent Compare Block**.

```vhdl
--adds implicit mantissa bit
--and also adds round and guard bits needed in later stages

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity man_fix is
    generic(n: integer:= 23);
    port(en: in std_logic;                                          --enable signal for unit
         manA_in,manB_in: in std_logic_vector(n-1 downto 0);        --input mantissas of A,B
         manA_out,manB_out: out std_logic_vector(n+2 downto 0) --output mantissas of A,B
        );
end man_fix;
architecture man_fix_arch of man_fix is
begin
    manA_out <= '1' & manA_in & "00" when en='1' else
             (others => 'X');
    manB_out <= '1' & manB_in & "00" when en='1' else
             (others => 'X');
end man_fix_arch;
```

Εικόνα 11: VHDL κώδικας που περιγράφει τη Man Fix υπο-μονάδα

```vhdl
architecture small_ALU_arch of small_ALU is
    signal c: std_logic;
begin
    process(en,expA,expB)
    variable dif: signed(7 downto 0);
    begin
        if(en='1') then
            if(unsigned(expA) > unsigned(expB)) then ----------------------B needs shifting
                dif := signed(expA) - signed(expB);
                if (unsigned(dif)<="11010") then
                    shift <= std_logic_vector(dif(4 downto 0));
                elsif (unsigned(dif)>"11010") then
                    shift <= "11010"; --the result becomes subnormal
                else
                    shift <= "XXXXX";
                end if;
            elsif(unsigned(expA) < unsigned(expB)) then -------------------A needs shifting
                dif := signed(expB) - signed(expA);
                if (unsigned(dif)<="11010") then
                    shift <= std_logic_vector(dif(4 downto 0));
                elsif (unsigned(dif)>"11010") then
                    shift <= "11010"; --the result becomes subnormal
                else
                    shift <= "XXXXX";
                end if;
            else ----------------------------------------------expA=expB no shifting needed
                shift <= "00000";
            end if;
        else
          shift <= "XXXXX";
        end if;
    end process;
    c <= '0' when unsigned(expB) > unsigned(expA) else
         '1';
    exp_out <= expB when c='0' else
               expA;
    cmp <= c;
end small_ALU_arch;
```

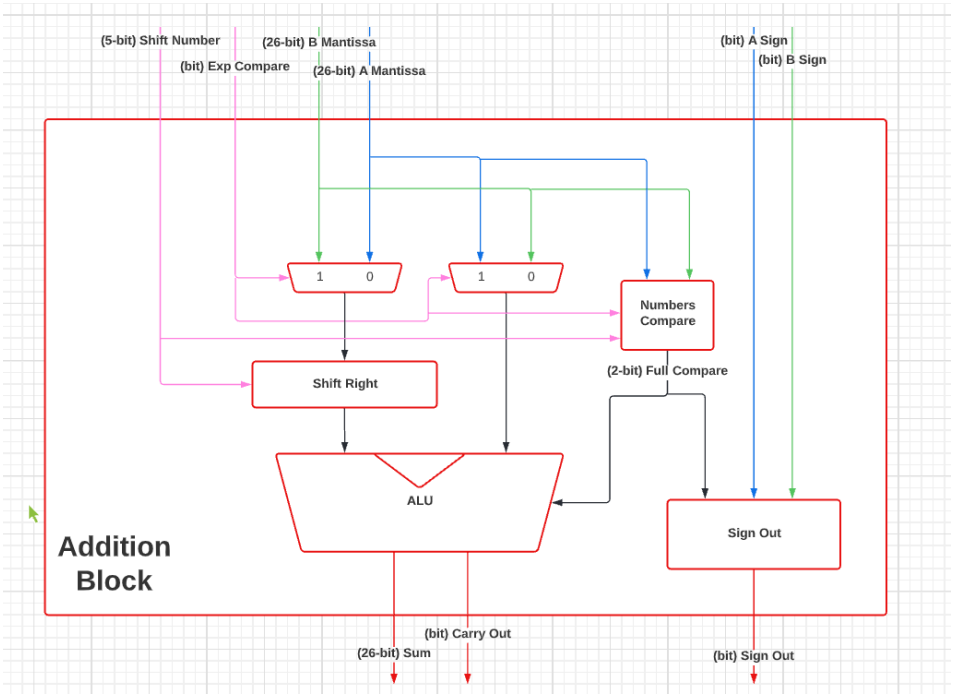
Εικόνα 12: VHDL κώδικας που περιγράφει τη Small ALU υπο-μονάδα



Εικόνα 13: Προσομοίωση του Exponent Compare Block

Στα πρώτα 5 section της παραπάνω προσομοίωσης, φαίνονται διάφορες περιπτώσεις αριθμών που εισάγονται και η υπο-μονάδα υπολογίζει το μεγαλύτερο εκθέτη από τους δύο, τη διαφορά των δυο εκθετών και συνενώνει το επιπλέον απαραίτητα bits στις δυο mantissas. Στο section 6 ρίχνουμε το σήμα *en* και παρατηρούμε ότι δεν υπολογίζονται οι έξοδοι καθώς λειτουργεί σαν enable στο μπλοκ.

Επόμενο υπο-μπλοκ προς ανάλυση σύμφωνα με το data path είναι το **Addition Block** του οποίου το κυκλωματικό διάγραμμα φαίνεται παρακάτω.



Εικόνα 14: Addition Block

Αρχικά δυο πολυπλέκτες βάσει του σήματος *Exp Compare* που προέρχεται από το προηγούμενο μπλοκ κατευθύνουν τις δυο mantissas ώστε αυτή που χρειάζεται να μετατοπιστεί περνάει από τη **Shift Right** υπο-μονάδα.

```vhdl
entity shift_R is
        generic(n: integer := 26; m: integer:=5);
        port(man_in: in std_logic_vector(n-1 downto 0);      --input mantissa that possibly needs to be shifted
             shift: in std_logic_vector(m-1 downto 0);       --amount that the mantissa of the number with the smaller exponent needs to be shifted (based on their diff)
             man_out: out std_logic_vector(n-1 downto 0)     --shifted output mantissa
        );
end shift_R;

architecture shift_R_arch of shift_R is
        --mid supp signals
        signal bit_man_in: bit_vector(n-1 downto 0);
        signal int_shift: integer range 0 to 26;
begin
        bit_man_in <= to_bitvector(man_in);
        int_shift <= to_integer(unsigned(shift));
        man_out <= to_stdlogicvector(bit_man_in srl int_shift);
end shift_R_arch;
```

Εικόνα 15: VHDL κώδικας που περιγράφει την Shift Right υπο-μονάδα

Αυτή έχοντας σαν είσοδο το σήμα *shift* εκτελεί τον απαραίτητο αριθμό μετατοπίσεων και τη μετατοπισμένη mantissa, τη προωθεί στη **Big ALU** υπο-μονάδα.

Η **Number Compare** υπο-μονάδα, είναι ένας σύνθετος πολυπλέκτης ο οποίος μας δηλώνει ποια από τις δύο εισόδους μας είναι ο μεγαλύτερος αριθμός κατά απόλυτη τιμή. Ειδικότερα, χρησιμοποιώντας τα σήματα *shift* και *cmp* αποφαίνεται αν ο αριθμός Α η Β είναι μεγαλύτερος και δηλώνει με τις τιμές "01" και "00" ότι ο Β είναι μεγαλύτερος. Η τιμή "10" δηλώνει ότι ο Α είναι μεγαλύτερος και η τιμή "11" ότι οι δύο αριθμοί είναι ίσοι. Το αποτέλεσμα από την **Number Compare** υπο-μονάδα στην συνέχεια χρησιμοποιείται για να καθοριστεί το πρόσημο (sign) του αποτελέσματος μέσω της **Sign Out** υπο-μονάδας και να γίνει σωστά η πράξη της πρόσθεσης από τη **Big ALU**. Ο παρακάτω πίνακας περιγράφει λεπτομερώς τα αποτελέσματα του compare ανάλογα τις εισόδους του.

Case(Full_CMP)	CMP	Shift	Mantissa	Output
B>A(EXP)	0	X	X	00(B>0)
A>=B(EXP)	1	!=00000	X	10(A>0)
A>=B(EXP)	1	00000	A_Man>B_Man	10(A>B)
A>=B(EXP)	1	00000	A_Man<B_Man	01(B>0)
A>=B(EXP)	1	00000	A_Man=B_Man	11(Equal)

```vhdl
entity compare is
        generic(n: integer:= 26);
        port(manA,manB: in std_logic_vector(n-1 downto 0);    --mantissas of A,B
             cmp: in std_logic;                               --compare bit which says which exponent was larger
             shift: in std_logic_vector(4 downto 0);          --amount that the mantissa of the number with the smaller exponent needs to be shifted (based on their diff)
             full_cmp: out std_logic_vector(1 downto 0)       --full compare vector needed for later stages
    );
end compare;
architecture compare_arch of compare is
begin
    full_cmp <= "00" when cmp='0' else
                "10" when shift /= "00000" else
                "01" when manB>manA else
                "10" when manA>manB else
                "11";
end compare_arch;
```

Εικόνα 16: VHDL κώδικας που περιγράφει την Numbers Compare υπο-μονάδα

Η **Sign Out** υπο-μονάδα είναι υπεύθυνη για να εξάγει το πρόσημο του αθροίσματός που σε μία πρόσθεση ομόσημων είτε ετερόσημων αριθμών είναι το πρόσημο του μεγαλύτερου αριθμού. Οπότε πρόκειται για έναν απλό πολυπλέκτη ο οποίος ανάλογα την τιμή της **Number Compare** υπο-μονάδας, καθορίζει το πρόσημο της εξόδου να

είναι αυτό του μεγαλύτερου αριθμού. Στην περίπτωση που οι αριθμοί είναι ίσοι (compare block output “11”) το πρόσημο επιλέγουμε να καθορίζεται από τον αριθμό A.

```
entity signout is
    port(sA,sB: in std_logic;                          --signs of A,B
         full_cmp: in std_logic_vector(1 downto 0);    --full compare vector
         sout: out std_logic                           --signout of the addition
        );
end signout;

architecture signout_arch of signout is
begin
    sout <= sA when full_cmp = "11" else
            sA when full_cmp = "10" else
            sB when full_cmp(1)='0' else
            'X';
end signout_arch;
```

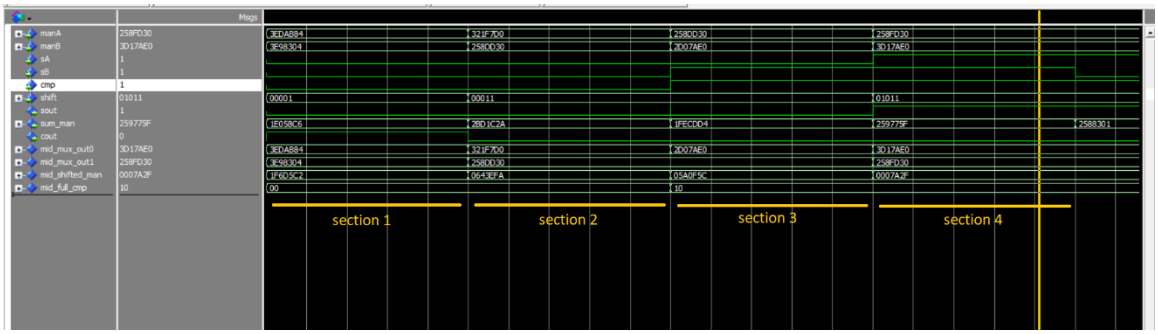
Εικόνα 17: VHDL κώδικας που περιγράφει την Sign-out υπο-μονάδα

Μεγάλη σε μέγεθος (area) υπο-μονάδα στο Addition υπο-μπλοκ είναι η **Big ALU** υπο-μονάδα. Τη έχουμε περιγράψει στη VHDL με behavioral κώδικα. Ανάλογα τα πρόσημα των δυο αριθμών και την έξοδο της **Number Compare** υπο-μονάδας κάνει πρόσθεση ή αφαίρεση των δυο mantissas. Παρακάτω φαίνεται ο VHDL κώδικας που την περιγράφει.

```
architecture big_ALU_arch of big_ALU is
        --supp signals_extra bit for carry
        signal mid_manA: unsigned(n downto 0);
        signal mid_manB: unsigned(n downto 0);
        signal mid_sum_man: unsigned(n downto 0);
begin
        mid_manA <= unsigned('0'& manA);
        mid_manB <= unsigned('0'& manB);

        process(mid_manA,mid_manB,full_cmp,sA,sB)
        begin
        --same sign just add them
            if(sA = sB) then
            mid_sum_man <= mid_manA + mid_manB;
            --diff sign & manB > manA
            elsif((sA/=sB) and full_cmp="01") then
                    mid_sum_man <= mid_manA - mid_manB;
            --diff sign & manA > manB
            else
                    mid_sum_man <= mid_manB - mid_manA;
            end if;
       end process;

        --output logic
        sum_man <= std_logic_vector(mid_sum_man(n-1 downto 0));
        cout <= std_logic(mid_sum_man(n));

end big_ALU_arch;
```

Εικόνα 18: VHDL κώδικας που περιγράφει την Big ALU υπο-μονάδα

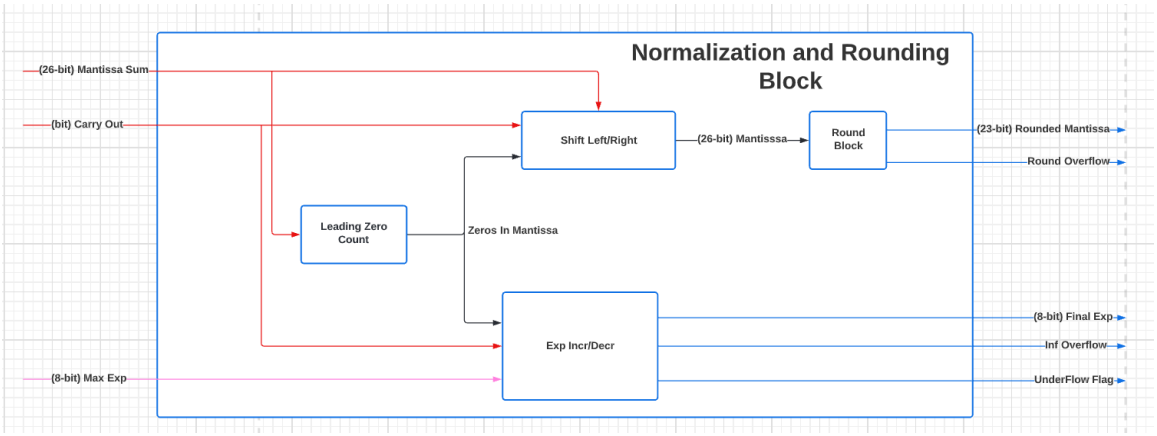
Παρακάτω φαίνεται μια ενδεικτική προσομοίωση του συνολικού **Addition Block**.



Εικόνα 19: Ενδεικτική προσομοίωση του Addition Block

Στη παραπάνω προσομοίωση εισάγονται διάφορες τιμές mantissas, πρόσημων, *shift* και *cmp* σημάτων. Οι mantissas εμφανίζονται σε δεκαεξαδική μορφή.

Επόμενο υπο-μπλοκ στο data path είναι το **Normalization and Rounding Block** το οποίο, όπως διαπιστώνεται από το όνομα, εκτελεί τις διεργασίες της κανονικοποίησης και στρογγυλοποίησης. Στη διεργασία κανονικοποίησης του αθροίσματος των δυο mantissas, όπου για να αντιστοιχεί στην επιστημονική σημειογραφία που εξηγήσαμε στην εισαγωγή, είτε μετατοπίζεται δεξιά και αυξάνεται ο εκθέτης (max exponent), είτε μετατοπίζεται αριστερά και μειώνεται ο εκθέτης. Στη διεργασία της στρογγυλοποίησης, στρογγυλοποιούμε τη mantissa, αφότου έχει κανονικοποιηθεί, χρησιμοποιώντας και τα επιπλέον bits που προσθέσαμε προηγουμένως. Παρακάτω φαίνεται το κυκλωματικό διάγραμμα αυτού το μπλοκ.



Εικόνα 20: Normalization and Rounding Block

Το άθροισμα των δυο mantissas που υπολογίστηκε στο προηγούμενο μπλοκ, εισέρχεται στην **Leading Zero Count** υπο-μονάδα η οποία μετρά το πλήθος συνεχόμενων μηδενικών που εμφανίζονται σε αυτό και το περνά σαν έξοδο, στις **Shift Left/Right** και **Exponent Increment/Decrement** υπο-μονάδες.

```vhdl
architecture zero_count_arch of zero_count is
begin
    process(man_in)
    variable count: integer range 0 to 32;
    begin
        count:=0;
        for i in man_in'range loop
            case man_in(i) is
                when '0' => count:=count+1;
                when others => exit;
            end case;
        end loop;
    zeros <= std_logic_vector(to_unsigned(count,zeros'length));
    end process;
end zero_count_arch;
```

Εικόνα 21: VHDL κώδικας που περιγράφει την Leading Zero Count υπο-μονάδα

Αφού έχουμε υπολογίσει με την **Leading Zero Count** υπο-μονάδα τον αριθμό των leading zeros, στην συνέχεια κανονικοποιούμε την μάντισσα του αθροίσματος. Ουσιαστικά για να γίνει αυτό θα πρέπει να ολισθαίνουμε αριστερά ή δεξιά την μάντισσα για να μας δώσει την κανονικοποιημένη μορφή του αθροίσματος. Στην συνέχεια θα πρέπει να αυξήσουμε τον exponent ανάλογα αν ολισθαίναμε την μάντισσα αριστερά ή δεξιά. Ειδικότερα η **Shift_LR** υπο-μονάδα, αν το *cout* = '1' (carry-out 1 από την άθροιση) θα ολισθήσει την μάντισσα δεξιά με την εντολή srl κατά μία θέση ώστε στο MSB της μάντισσας να έχουμε 1 και να είναι σε κανονικοποιημένη μορφή. Σε διαφορετική περίπτωση η μάντισσα θα ολισθήσει αριστερά όσο δείχνει η τιμή του zero counter (sll int_zero_count) έτσι ώστε να εμφανιστεί στο MSB ο άσσος.

```vhdl
architecture shift_LR_arch of shift_LR is
    --mid supp signals
    signal to_shift: bit_vector(n downto 0);              --vector that will be shifted
    signal int_zero_count: integer range 0 to 26;         --integer number of zero count
    signal shifted: bit_vector(n downto 0);               --shifted vector
begin

        int_zero_count <= to_integer(unsigned(zero_count));
        to_shift <= to_bitvector(cout & man_in);
        shifted <= to_shift srl 1 when cout = '1' else --we dont need bit in position 26-is the carry out bit
                   to_shift sll int_zero_count;
        man_out <= to_stdlogicvector(shifted(n-1 downto 0));

end shift_LR_arch;
```

Εικόνα 22: VHDL κώδικας που περιγράφει την Shift_LR υπο-μονάδα

Η **Exponent Inc/Dec** υπο-μονάδα, ουσιαστικά αν υπάρχει *carry out* θα αυξήσει τον εκθέτη κατά 1. Σε διαφορετική περίπτωση, ο εκθέτης θα ελαττωθεί κατά το ποσό που δείχνει ο zero counter. Η μονάδα αυτή παράγει και δύο άλλες εξόδους το underflow όταν ο εκθέτης είναι μικρότερος του zero counter και δεν θα μπορέσουμε να αναπαραστήσουμε αυτόν τον αριθμό επειδή είναι πολύ μικρός (subnormal). Η δεύτερη έξοδος δηλώνει το overflow (exp="11111111") το οποίο στην περίπτωσή μας αναπαρίσταται με το άπειρο. Τέλος, έχει σημασία να πούμε ότι αν έχουμε carry out μηδέν και zero count = 26 (mantissa όλα μηδέν) σημαίνει ότι έγινε πρόσθεση αντίθετων

αριθμών και ο εκθέτης θα είναι μηδέν. Παρακάτω φαίνεται απόσπασμα του VHDL κώδικα που περιγράφει την υπο-μονάδα.

```
architecture inc_dec_exp_arch of inc_dec_exp is

    signal mid_exp_in: unsigned(n downto 0);
    signal mid_exp_out: unsigned(n downto 0);
        signal mid_int: integer range 0 to 26;

begin

        mid_exp_in <= unsigned('0' & exp_in);
        mid_int <= to_integer(unsigned(zero_count));

        process(mid_exp_in,mid_int,cout)
        begin
            if (cout = '1') then
            mid_exp_out <= mid_exp_in + 1;
            elsif(mid_int > 0) then
                    mid_exp_out <= mid_exp_in - mid_int;
            else
                    mid_exp_out <= mid_exp_in;
        end if;
        end process;

        unf <= '1' when mid_exp_out(n) = '1' else
               '1' when mid_exp_out(n-1 downto 0) = "00000000" else
               '0';
        inf <= '1' when mid_exp_out(n-1 downto 0) = "11111111" else
               '0';
        exp_out <= "00000000" when zero_count = "11010" and cout = '0' else
                   std_logic_vector(mid_exp_out(n-1 downto 0)) when mid_exp_out(n) = '0' else
                   "00000000";

 end inc_dec_exp_arch;
```

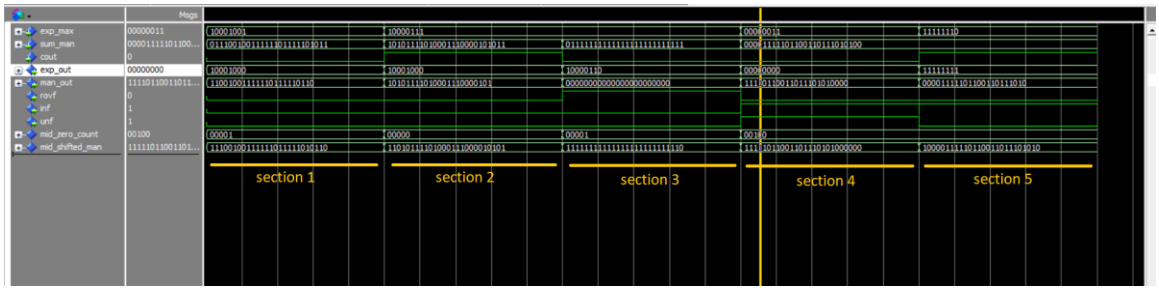
Εικόνα 23: VHDL κώδικας που περιγράφει την Exponent Inc/Dec υπο-μονάδα

Η μετατοπισμένη mantissa εισέρχεται μετά στην **Rounding** υπο-μονάδα. Εκεί αφού τα δεδομένα μας έχουν περάσει από την normalization υπο-μονάδα, στην συνέχεια στρογγυλοποιούμε το τελικό αποτέλεσμα. Για την στρογγυλοποίηση χρησιμοποιούμε την μέθοδο στρογγυλοποίησης στον κοντινότερο αριθμό και σε περίπτωση που απέχει το ίδιο εκτελούμαι με τέτοιο τρόπο την στρογγυλοποίηση ώστε να έχουμε άρτιο(LSB = 0) τελευταίο bit (round to nearest, tie to even). Συγκεκριμένα, ανάλογα τις τιμές του round και guard bit (2 LSBs) θα αυξήσουμε κατά ένα το αποτέλεσμα είτε θα το αφήσουμε ως έχει. Οπότε για guard και round bits “00”, “01” αφήνουμε την μάντισσα ως έχει, για τιμή “11” αυξάνουμε την μάντισσα και αποφασίζουμε αναλόγως για την μέση περίπτωση, δηλαδή για τιμή “10”. Αυτή η υπο-μονάδα βγάζει επίσης σαν έξοδο και ένα σήμα *round overflow* το οποίο γίνεται high στη περίπτωση υπερχείλισης δηλαδή όταν η διαδικασία της στρογγυλοποίησης αυξήσει τον αριθμό των ψηφίων της mantissa. Παρακάτω φαίνεται ο VHDL κώδικας που περιγράφει την υπο-μονάδα.

```vhdl
architecture round_arch of round is
    signal mid_man: unsigned(m downto 0);
    signal mid_Rman: unsigned(m downto 0);
    signal LGR: std_logic_vector(2 downto 0);  --this mid signal stores the 3 last bits of input mantissa
                                               --then uses them for round to nearest, tie to even
begin
        LGR <= man_in(2 downto 0);
        mid_man <= unsigned('0' & man_in(n-2 downto 2));
        mid_Rman <= mid_man when LGR(1 downto 0) < "10" else
                    mid_man + 1 when LGR(1 downto 0) > "10" else
                    mid_man when LGR = "010" else
                    mid_man + 1 when LGR = "110" else
                    "XXXXXXXXXXXXXXXXXXXXXXXX";
        man_out <= std_logic_vector(mid_Rman(m-1 downto 0));
        rovf <= std_logic(mid_Rman(m));

end round_arch;
```

Εικόνα 24: VHDL κώδικας της Rounding υπο-μονάδας

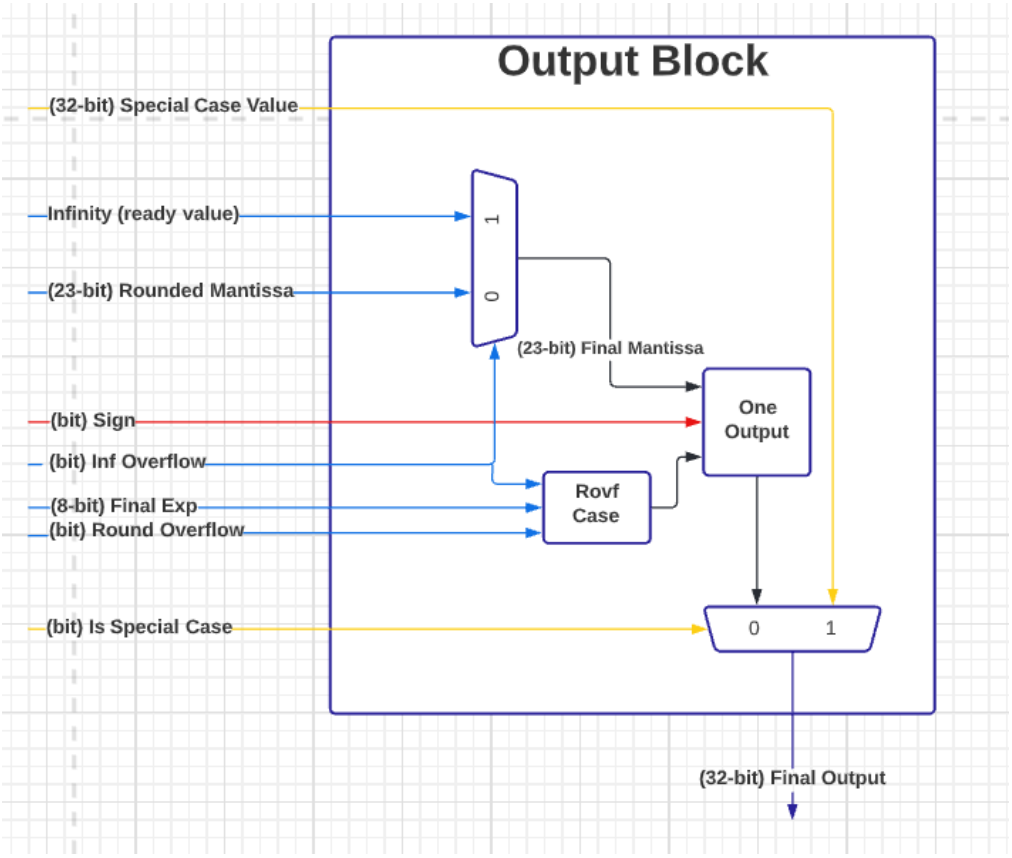
Παρακάτω φαίνεται μια ενδεικτική προσομοίωση του **Normalization and Rounding Block**.



Εικόνα 25: Ενδεικτική προσομοίωση του Normalization and Rounding Block

Τα δυο πρώτα section δεν περιέχουν αξιοσημείωτα αποτελέσματα καθώς δε προκύπτουν ειδικές περιπτώσεις. Στο section 3 όμως προκύπτει overflow από τη **Rounding** υπο-μονάδα. Στο section 4, προκύπτει underflow από την **Exp Inc/Dec** υπο-μονάδα. Στο section 5, προκύπτει overflow από την **Exp Inc/Dec** υπο-μονάδα.

Τελευταία υπο-μονάδα στο data path πριν τον καταχωρητή εξόδου είναι το **Output Block** και το κυκλωματικό του διάγραμμα φαίνεται παρακάτω.



Εικόνα 26: Output Block

Αρχικά μέσω ενός πολυπλέκτη και με σήμα ελέγχου το σήμα overflow/υπερχείλισης (infinity), που προκύπτει από την **Exponent Inc/Dec** υπο-μονάδα, επιλέγεται είτε η στρογγυλοποιημένη mantissa από το προηγούμενο μπλοκ, είτε τιμή που αντιστοιχεί στο άπειρο σε περίπτωση που προκύπτει υπερχείλιση.

Επειδή λοιπόν από την διαδικασία της στρογγυλοποίησης υπάρχει η περίπτωση ύπαρξης overflow (κατά την αύξηση κατά ένα της μάντισσας) ή υπο-μονάδα **Round Overflow Case** διαχειρίζεται αυτή την περίπτωση και ανάλογα αν έχει προκύψει overflow από το round block, το παραπάνω block θα αυξήσει κατά ένα τον exponent ώστε να αναπαρασταθεί σωστά το αποτέλεσμα της πρόσθεσης. Σημαντικό είναι, να αναφέρουμε, ότι σε περίπτωση που ο εκθέτης έχει την τιμή “FF”hex (εκθέτης για την αναπαράσταση του απείρου) η αύξηση αυτή δεν θα γίνει επειδή το αποτέλεσμα έχει φτάσει είδη στην μέγιστη τιμή του. Παρακάτω φαίνεται απόσπασμα του κώδικα που περιγράφει την υπο-μονάδα.

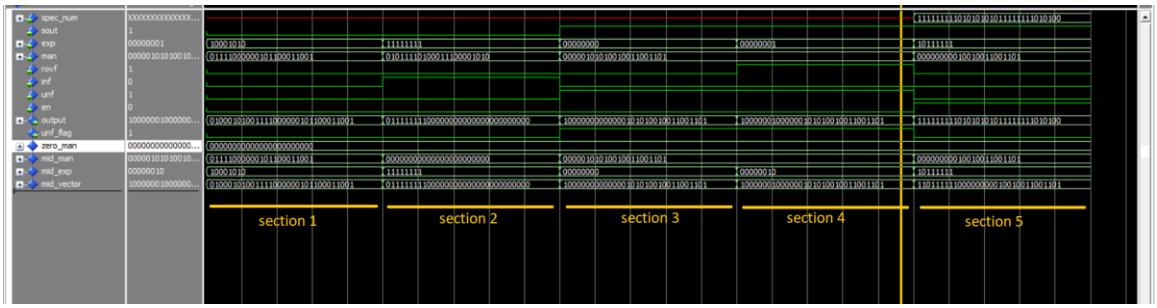
```vhdl
architecture round_ovf_arch of round_ovf is
    signal mid_exp_out: unsigned(n-1 downto 0);
    signal sel: std_logic;
begin

    mid_exp_out <= unsigned(exp_in) + 1;

    sel <= '0' when inf = '1' else
           '1' when rovf= '1' else
           '0';

    with sel select
        exp_out <= std_logic_vector(mid_exp_out) when '1',
                   exp_in when '0',
                   "XXXXXXXX" when others;
end round_ovf_arch;
```

Εικόνα 27: VHDL κώδικας που περιγράφει τη Round Overflow υπο-μονάδα

Ύστερα, η έξοδοι του πολυπλέκτη, της **Round Overflow Case** υπο-μονάδας και σήμα *sign-out* που προκύπτει από το **Addition Block** εισέρχονται στην υπο-μονάδα **One Output**, η οποία απλώς συνενώνει τις τρεις αυτές συνιστώσες στον τελικό 32-bit αριθμό που παράγεται σαν αποτέλεσμα σε κανονική λειτουργία του συστήματος.

Για τυχόν ειδική περίπτωση, όπου η ειδική τιμή που βγαίνει σαν αποτέλεσμα υπολογίζεται στο **Special Case Block**, ένας πολυπλέκτης τοποθετείται ώστε τότε, με το σήμα *is special case,* που δηλώνει αυτό το συμβάν, ως σήμα επιλογής να διαλέγουμε την κατάλληλη από τις δυο εισόδους του.

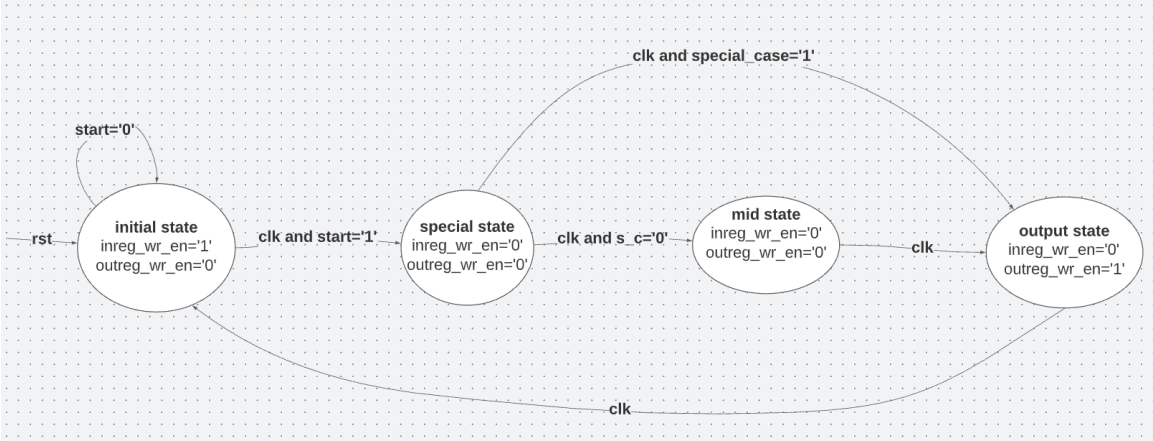
Παρακάτω φαίνεται μια ενδεικτική προσομοίωση του **Output Block**.



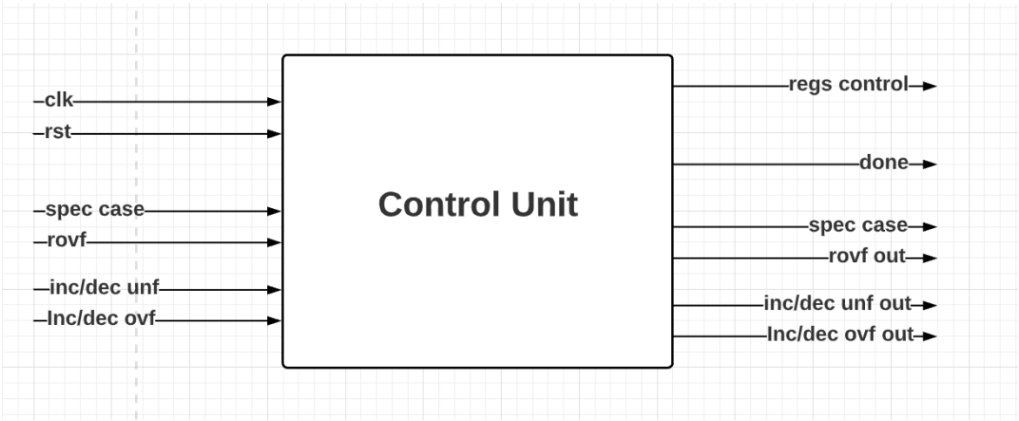
Εικόνα 28: Ενδεικτική προσομοίωση του Output Block

Το section 1 δεν παρουσιάζει κάποια ειδική περίπτωση πρόκειται για απλή περίπτωση λειτουργίας του μπλοκ όπου απλά συνενώνει τα επιμέρους στοιχεία του αποτελέσματος. Στο section 2, προκύπτει υπερχείλιση (infinity) καθώς σηκώνεται το σήμα *inf* και βλέπουμε και στην έξοδο τη τιμή του συν άπειρο. Στο section 3, έχουμε περίπτωση underflow, οπότε βλέπουμε μηδενικό εκθέτη στην έξοδο και σηκωμένη την αντίστοιχη flag. Στο section 4, προκύπτει underflow, σε αυτή τη περίπτωση το αποτέλεσμα είναι λανθασμένο. Στο section 5, πρόκειται για ειδική περίπτωση που αναγνωρίζεται στο **Special Case Block**. Το αποτέλεσμα παράγεται στο μπλοκ αυτό, προωθείτε στο **Output Block** και επιλέγεται από τον πολυπλέκτη σαν έξοδος.

Πιο σημαντική υπο-μονάδα στο σύστημά μας είναι το **Control Unit**. Αυτό ελέγχει όλη τη λειτουργία στο σύστημα. Τρέχει παράλληλα με το data path και ανάλογα με ορισμένα control σήματα που προκύπτουν από τις διάφορες υπο-μονάδες, τα οποία τα δέχεται σαν εισόδους, και τα προωθεί στη επόμενα μπλοκ βάση του data path, καθορίζει τη κατάσταση που βρίσκεται το σύστημα μας ανάλογα το κομμάτι διεργασίας που εκτελεί. Το **Control Unit** αποτελείται από μια FSM (finite state machine) που υλοποιεί τη λειτουργία του. Παρακάτω φαίνεται το διάγραμμα που περιγράφει τη λειτουργία αυτή.



Εικόνα 29: Διάγραμμα λειτουργίας FSM του Control Unit



Εικόνα 30: Κυκλωματικό διάγραμμα του Control Unit

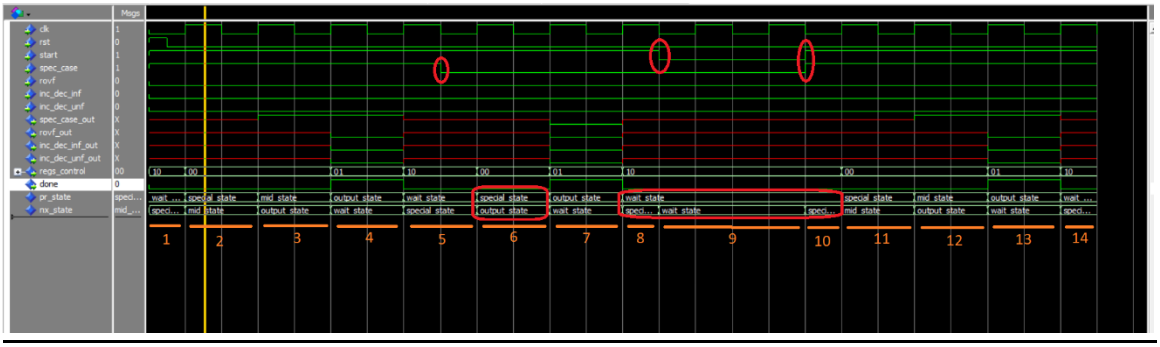
Η FSM είναι τύπου Moore και αποτελείται από τέσσερις καταστάσεις, την αρχική **initial state** όπου το σύστημα ξεκινά τη λειτουργία του και επιστρέφει με το ασύγχρονο σήμα *reset* ή με το πέρας της τελευταίας κατάστασης. Επόμενη κατάσταση, εφόσον έχουμε ενεργό σήμα *start,* είναι η **special state** όπου το σύστημα πραγματοποιεί τη διεργασία του **Special Case Block** και παράγεται το σήμα *not_special_case* βάση του οποίου αποφασίζεται η επόμενη κατάσταση στην μηχανή. Αν έχουμε ειδική περίπτωση, επόμενη κατάσταση είναι η κατάσταση εξόδου όπου παράγεται το τελικό αποτέλεσμα.

Αν όχι, επόμενη είναι η **mid-state** στην οποία πραγματοποιούνται οι διεργασίες των **Exponent Compare Block**, **Addition Block** και **Normalization and Rounding Block**. Τέλος φτάνουμε στην **output state**, όπου παράγεται η έξοδος του κυκλώματος, και με το επόμενο ρολόι επιστρέφουμε πάλι στην αρχική όπου αναμένουμε βάση του σήματος start τις επόμενες τιμές. Το **Control Unit**, εκτός ότι προωθεί σημαντικά σήματα από υπο-μπλοκ σε υπο-μπλοκ, ανάλογα τη κατάσταση, ελέγχει τους καταχωρητές εισόδου και εξόδου. Από το διάγραμμα της μηχανής παρατηρούμε ότι στην αρχική κατάσταση επιτρέπουμε εγγραφή μόνο στο καταχωρητή εισόδου καθώς από εκεί διαβάζουμε τα δεδομένα στο επόμενο μπλοκ του data path, στις δυο ενδιάμεσες καταστάσεις δεν επιτρέπουμε εγγραφή καθώς το σύστημα δεν έχει υπολογίσει ακόμα το αποτέλεσμα και δεν είμαστε έτοιμοι να δεχθούμε τις επόμενες τιμές προς επεξεργασία. Παρακάτω φαίνεται το κομμάτι κώδικα που περιγράφει τη λογική εναλλαγής καταστάσεων ανάλογα και του σήματος *special case*, όπου παραλείπονται τα ενδιάμεσα μπλοκ επεξεργασίας και η αντίστοιχη κατάσταση της μηχανής.

```
51      --next state and output process
52      P2: process(start, pr_state, spec_case)
53      begin
54          case pr_state is
55              when wait_state =>
56                  if(start='1') then
57                      nx_state <= special_state;
58                  else
59                      nx_state <= wait_state;
60                  end if;
61              when special_state =>
62
63                  if(spec_case='1') then
64                      nx_state <= mid_state;
65                  else
66                      nx_state <= output_state;
67                  end if;
68              when mid_state =>
69                  nx_state <= output_state;
70              when output_state =>
71                  nx_state <= wait_state;
72          end case;
73      end process P2;
```

Εικόνα 31: VHDL κώδικας που περιγράφει την εναλλαγή καταστάσεων στο Control Unit

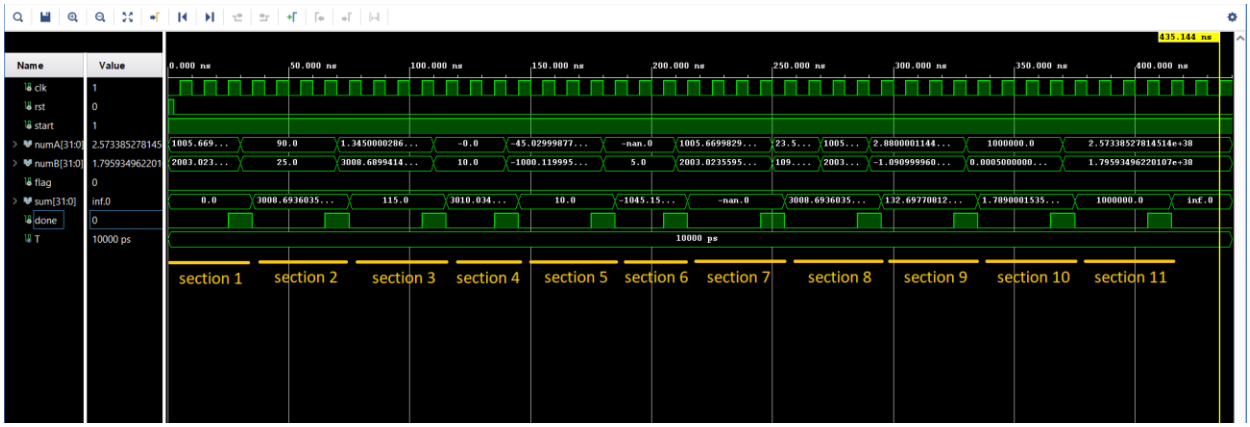
Παρακάτω υπάρχει ενδεικτική προσομοίωση που επιβεβαιώνει την ορθή λειτουργία του Control Unit.



Εικόνα 32: Ενδεικτική προσομοίωση του Control Unit

Στο section 1, αρχικοποιούμε με το σήμα reset τη μηχανή που ξεκινά από την αρχική κατάσταση όπου βλέπουμε ότι επιτρέπεται η εγγραφή μόνο στο καταχωρητή εσόδου, βάση του σήματος *regs_control*. Το σήμα *reset* πέφτει μετά από T/4 και παρατηρούμε ότι μετά την πρώτη ανερχόμενη παρυφή του ρολογιού, στην αρχή του section 2 δηλαδή, η μηχανή περνά στην special state. Από εκεί καθώς δεν πρόκειται για ειδική περίπτωση επόμενη κατάσταση είναι η mid-state. Και η αμέσως επόμενη η output state. Στη κατάσταση αυτή βλέπουμε ότι επιτρέπεται εγγραφή στο καταχωρητή εξόδου και η επόμενη κατάσταση στην οποία περνάει το σύστημα στο section 5, είναι η wait state. Από εκεί περνάει στη special state και βλέποντας ότι έχουμε ειδική περίπτωση η επόμενη κατάσταση στην οποία θα περάσει με την ανοδική ακμή του ρολογιού είναι η output state. Στο section 8, έχει επιστρέψει στην αρχική κατάσταση στην οποία μένει μέχρι το section 10 καθώς έχουμε απενεργοποιήσει το σήμα start. Στο επόμενα sections μέχρι το section 14, πραγματοποιεί ακόμα ένα κύκλο των καταστάσεων σε κανονική λειτουργία δηλαδή χωρίς ειδικές περιπτώσεις.

Έχοντας πλέον ολοκληρώσει την ανάλυση και προσομοίωση λειτουργίας των επιμέρους μονάδων που συντελούν το συνολικό σύστημα, μπορούμε να ελέγξουμε την ορθή λειτουργία του μέσω προσομοίωσης. Παρακάτω φαίνεται μια ενδεικτική προσομοίωση του συστήματος.



Εικόνα 33: Ενδεικτική προσομοίωση του 32-bit Floating Point Adder

Στο section 1, γίνεται αρχικοποίηση του συστήματος μέσω του ασύγχρονου σήματος reset, εισέρχονται δυο τιμές και με το σήμα start να είναι ενεργό, ξεκινά η διαδικασία του συστήματος. Σε τρεις κύκλους βάσει και της μηχανής του Control Unit, αφού δε πρόκειται για ειδική περίπτωση, είναι έτοιμο το αποτέλεσμα οπότε σηκώνεται το σήμα done για να ειδοποιήσει ότι στην επόμενη ανερχόμενη ακμή του ρολογιού μπορούμε να διαβάσουμε το αποτέλεσμα από τον καταχωρητή εξόδου. Μέχρι και το section 3 πρόκειται για κανονική λειτουργία τους συστήματος και τα σωστά αποτελέσματα που περιμένουμε φαίνονται στο σήμα εξόδου sum. Στο section 4, έχουμε εισάγει τιμές που αντιστοιχούν σε ειδική περίπτωση (-0 και 10) όπου το αποτέλεσμα μπορεί να υπολογιστεί στο special case block και δεν χρειάζεται να περάσει από την ενδιάμεση

κατάσταση το σύστημα. Σε αυτή τη περίπτωση χρειάζεται ένας λιγότερος κύκλος για να βγει η έξοδος του κυκλώματος. Όντως παρατηρούμε ότι σε ειδική περίπτωση το αποτέλεσμα χρειάζεται ένα λιγότερο κύκλο. Ειδική περίπτωση έχουμε και στο section 6, όπου ο ένας από τους δυο αριθμούς εισόδου είναι NaN (not a number) και βλέπουμε ότι βάση και του αντίστοιχου πίνακα στο Special Case Block περνιέται σαν έξοδος. Ιδιαίτερο ενδιαφέρον έχει και το section 11, όπου το αποτέλεσμα είναι αρκετά μεγάλο και δεν μπορεί να αναπαρασταθεί στη μορφή που χρησιμοποιούμε (single-precision) και έτσι έχουμε σαν αποτέλεσμα άπειρο.

Vivado:

Αφότου ολοκληρώσαμε το σχεδιασμό και την διαπίστωση ορθής λειτουργίας του συστήματός μας, προχωρήσαμε στο περιβάλλον του Vivado ώστε να πραγματοποιήσουμε Synthesis και Implementation αυτού σε ένα επιλεγμένο part κατασκευαστή. Από αυτές τις λειτουργίες εξαγάγαμε σημαντικές πληροφορίες για το σύστημά μας, όπως η συχνότητα στην οποία μπορεί να λειτουργήσει το συστήματός μας, οι πόροι που καταλαμβάνει στο FPGA που επιλέξαμε και η ενέργεια που ξοδεύει κατά τη λειτουργία του. Αυτές οι πληροφορίες εμφανίζονται παρακάτω σε εικόνες που πήραμε από την εφαρμογή.

Είναι σημαντικό να πούμε ότι καθώς δεν έχουμε ξεκαθαρίσει ορισμένα constraints στο device όσων αφορά τα pins I/O και τη τοποθέτηση στο συνολικό χώρο, πολλά από τα παρακάτω δεδομένα δεν είναι τα βέλτιστα δυνατά.

Ξεκινώντας από το στάδιο του Synthesis, όπως ακολουθείται και από την εφαρμογή, αναφέρουμε τα διάφορα αποτελέσματα στους διάφορους διαγνωστικούς τομείς του συστήματος.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.327 ns	Worst Hold Slack (WHS):	0.066 ns	Worst Pulse Width Slack (WPWS):	4.650 ns
Total Negative Slack (TNS):	-6.250 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	30	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	132	Total Number of Endpoints:	132	Total Number of Endpoints:	100

Timing constraints are not met.

Εικόνα 34: Timing Summary από το Synthesis στάδιο

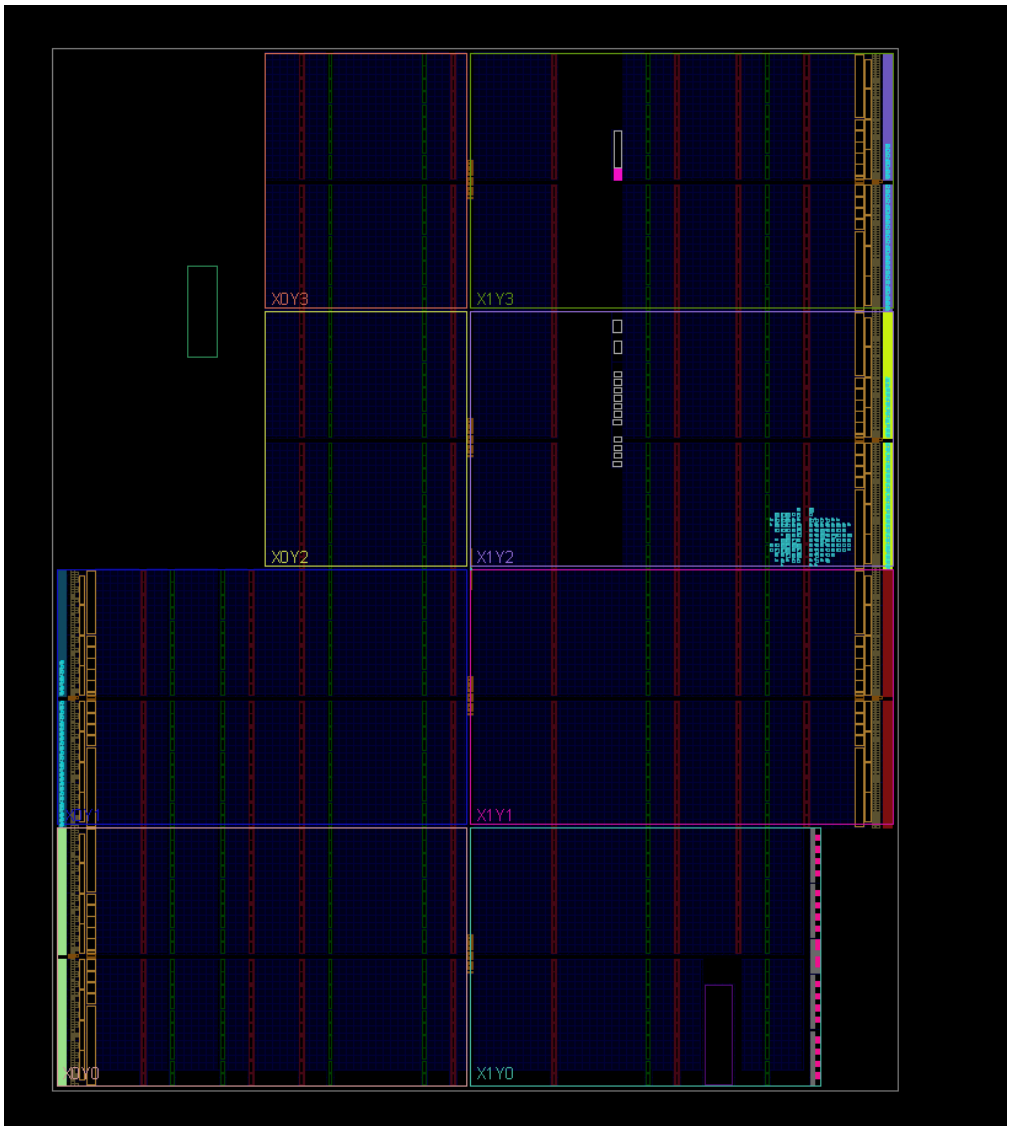
Βλέπουμε ότι υπάρχει setup violation στο synthesis στάδιο. Αυτό μπορεί να επιλυθεί αυξάνοντας το χρόνο περιόδου ώστε να προλαβαίνει το σύστημα να κρατάει τις σωστές τιμές κατά την εγγραφή σε FFs.

Παραπάνω φαίνονται οι πόροι που λαμβάνει το σύστημά μας από το FPGA ώστε να λειτουργήσει.

Hierarchy

Name	Slice LUTs (78600)	Slice Registers (157200)	F7 Muxes (39300)	Bonded IOB (150)	BUFGCTRL (32)
fp_adder	381	99	1	101	1
add_block (addition_block)	0	0	0	0	0
control (control_unit)	2	2	0	0	0
exponent_comparison (exp_compare_block)	78	0	0	0	0
in_register (input_register)	246	64	1	0	0
normalize_rounding (norm_round_block)	51	0	0	0	0
out_register (output_register)	4	33	0	0	0

Εικόνα 35: Utilization report στο Synthesis στάδιο



Εικόνα 36: Εικόνα συσκευής και περιοχής που καταναλώνεται από το σύστημά μας κατά το στάδιο του Implementation

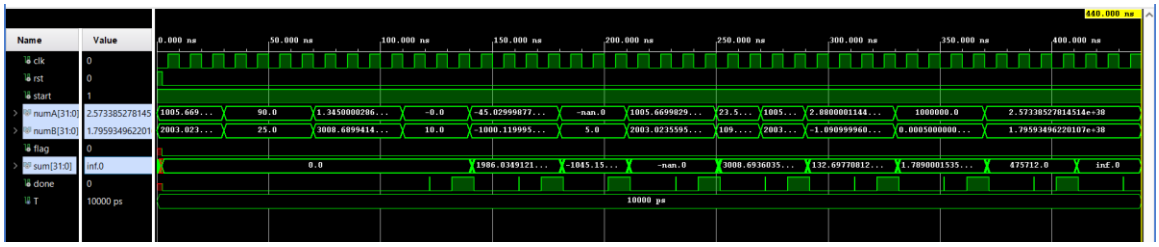
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.135 ns	Worst Hold Slack (WHS):	0.205 ns	Worst Pulse Width Slack (WPWS):	4.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	132	Total Number of Endpoints:	132	Total Number of Endpoints:	100

All user specified timing constraints are met.

Εικόνα 37: Timing Summary από το Implementation στάδιο

Στο στάδιο Implementation το σύστημά μας δεν παρουσιάζει setup ή hold violations. Βέβαια δεν λειτουργεί απολύτως σωστά όπως φαίνεται και από την παρακάτω χρονική προσομοίωση του.



Εικόνα 38: Χρονική προσομοίωση στο Implementation στάδιο

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
synth_1 (active)	**constrs_1**	**synth_design Complete!**								**381**	**99**	**0.0**	**0**	**0**
impl_1	constrs_1	route_design Complete!	0.135	0.000	0.205	0.000	0.000	0.136	0	402	99	0.0	0	0

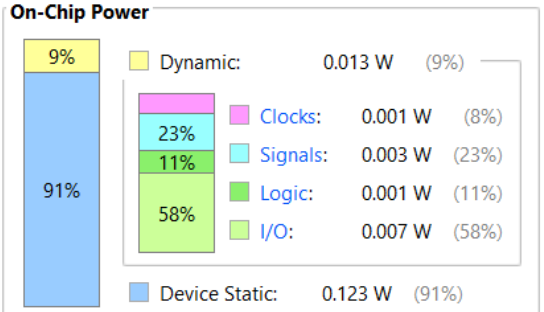
Εικόνα 39: Χρήση LUTs (look-up tables) και FFs (flip-flops) από το σύστημά μας

Summary

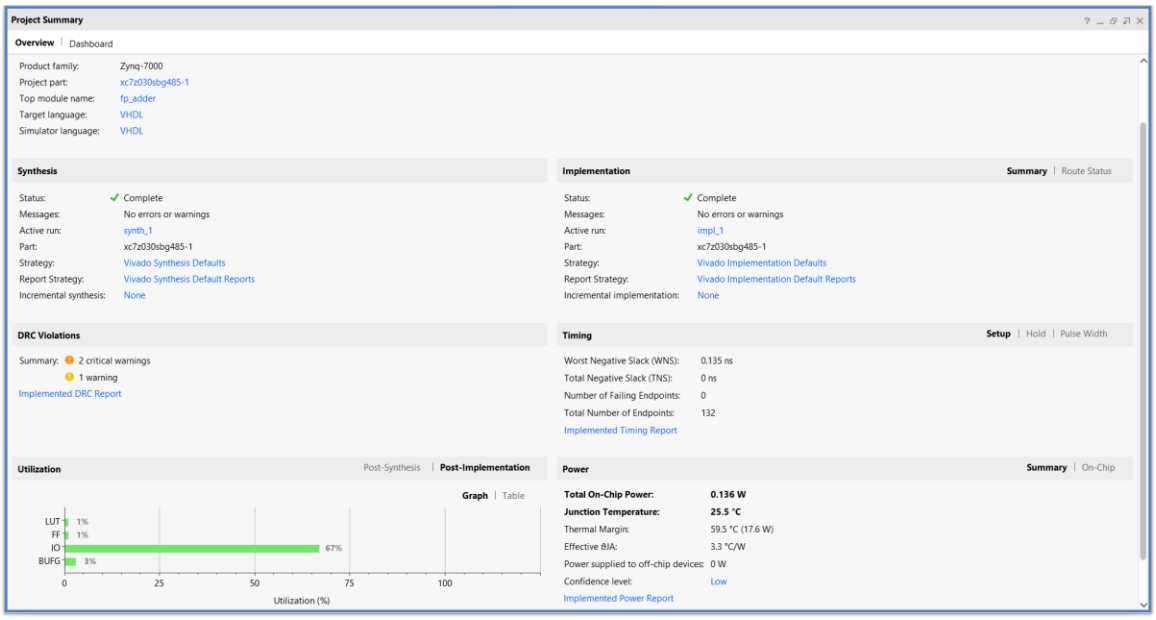
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.136 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (17.6 W)
Effective ϑJA:	3.3°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

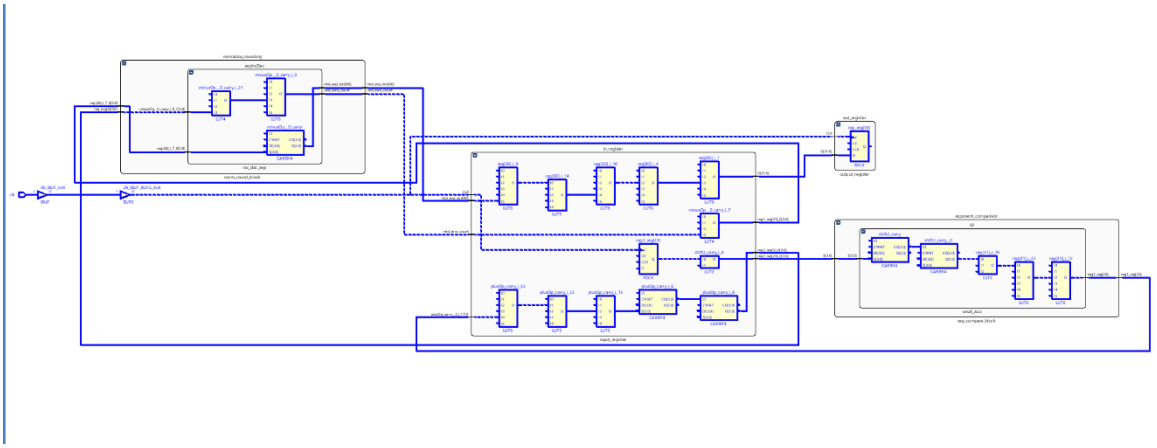
Launch Power Constraint Advisor to find and fix



Εικόνα 40: Power Summary



Εικόνα 41: Project Summary του συστήματός μας μετά το Synthesis και Implementation



Εικόνα 42: Critical Path του συστήματος μας