ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμημα ΗΜΜΥ. Τομεας Ηλεκτρονικης

Ψηφιακα Συστηματα HW σε Χαμηλα Επιπεδα Λογικης II

Εργασία

Πολλαπλασιασμός αριθμών κινητής υποδιαστολής

Συντάκτης:
Χρήστος Χουτουρίδης [8997]
cchoutou@ece.auth.gr

Διδάσκοντες:
Βασίλης Παυλίδης

Ευάγγελος Τζουβάρας

15 Ιουνίου 2025

1. Εισαγωγη

Η παρούσα εργασία αφορά στην υλοποίηση ενός πολλαπλασιαστή αριθμών κινητής υποδιαστολής μονής ακρίβειας (IEEE-754 single-precision) σε γλώσσα περιγραφής υλικού SystemVerilog. Η σχεδίαση του κυκλώματος ακολουθεί μια αρθρωτή και κλιμακούμενη προσέγγιση, βασισμένη σε δομή pipeline τριών σταδίων. Το κύριο ζητούμενο ήταν η διαχωρισμένη υλοποίηση της κανονικοποίησης, της στρογγυλοποίησης και της διαχείρισης εξαιρέσεων, με δυνατότητα ελέγχου διαφορετικών τρόπων στρογγυλοποίησης.

Στόχος της εργασίας είναι η κατανόηση της εσωτερικής λειτουργίας των αριθμητικών μονάδων κινητής υποδιαστολής, η υλοποίηση τους σε επίπεδο RTL και η επαλήθευση της σωστής λειτουργίας τους μέσω εκτενούς ελέγχου με test benches. Η εργασία απαιτεί κατανόηση τόσο της γλώσσας SystemVerilog όσο και του προτύπου IEEE-754, και συνδυάζει αρχιτεκτονική σχεδίαση, ψηφιακή λογική και επαλήθευση μέσω προσομοίωσης.

1.1. Παραδοτέα

Τα παραδοτέα της εργασίας αποτελούνται από:

- Την παρούσα αναφορά.
- Τον κατάλογο **src**/, που περιέχει τον κώδικα της SystemVerilog με τα ζητηθέντα modules.
- Τον κατάλογο **sim**/, που περιέχει το ζητηθέν test bench αλλά και επιμέρους tests για τα υπόλοιπα modules (πλην του exception handler).
- Το σύνδεσμο με το αποθετήριο που περιέχει όλο το project στο vsim, τους κώδικες της SystemVerilog και της αναφοράς καθώς και τα παραδοτέα.

Στην εργασία **δεν** υλοποιήθηκαν τα εξής:

- *Όλοι οι 144* συνδυασμοί των corner cases για το test bench με βάση τους τυχαίους αριθμούς και τον έλεγχο μέσω multiplication.sv. Αντ' αυτού δημιουργήθηκαν test cases που ελέγχουν τόσο την "κανονική" λειτουργία όσο και αρκετά corner cases αλλά οι τιμές τους εισήχθηκαν *με το χέρι*.
- Το μέρος με τα *immediate assertions*.

2. Προσεγγιση Υλοποιησης

Για την εργασία χρησιμοποιήσαμε μια **αρθρωτή** προσέγγιση. Καθώς η πολυπλοκότητα έμοιαζε αρκετά μεγάλη, σε συνδυασμό με τη μικρή προσωπική εμπειρία σε ψηφιακή σχεδίαση με verilog, προτιμήσαμε να ακολουθήσουμε μια **test driven** προσέγγιση ανά module. Έτσι χωρίσαμε την ανάπτυξη στα modules που πρότεινε η εκφώνηση και για αυτά δημιουργήσαμε test benches. Τα βασικά tests που θα χρειαζόταν να υλοποιηθούν πρώτα ήταν για τα *normalize* και *round*. Το *exception handler* θα μπορούσε να εκμαιευτεί με "αφαίρεση" των παραπάνω από το συνολικό test bench του *fp_mult*. Όλα αυτά τα αρχεία μπορούν να βρεθούν στον κατάλογο *sim/*.

Τα test αυτά αν και κάπως τετριμμένα, έπαιξαν πολύ σημαντικό ρόλο στην υλοποίηση καθώς μας δώσανε τη δυνατότητα να χωριστούν τα interfaces των modules και να πιστοποιήσουν ότι η αναγκαία λειτουργικότητα είναι παρούσα. Κάτι που φυσικά μας βοήθησε στην απομόνωση των σφαλμάτων κλπ...

Έχοντας υλοποιήσει λοιπόν τα test προβήκαμε στην υλοποίηση των ίδιων των modules. Αρχικά υλοποιήσαμε τα *normalize* και *round* και έπειτα το *fp_mult*. Με αυτά τα τρία να περνούν τα επιμέρους tests προχωρήσαμε στην υλοποίηση του *exception handler*. Για αυτό το module δεν υλοποιήσαμε πρώτα tests καθώς υπήρχαν τα συνολικά tests και το μόνο modul-άκι που "άλλαζε" ήταν αυτό.

Τέλος με όλη τη λειτουργικότητα έτοιμη, προβήκαμε στην τελευταία αλλαγή, που ήταν η μετατροπή της λογικής για χρήση **pipeline τριών σταδίων**. Μιας και τα αρχεία για τα tests για το pipeline είναι διαφορετικά από τα αρχικά, ο αναγνώστης μπορεί εδώ να βρει αυτή την ενδιάμεση έκδοση.

2.1. Pipeline

Η αρχική υλοποίηση όπως είδαμε είχε combinational logic. Επομένως το αποτέλεσμα του πολλαπλασιασμού θα ήταν διαθέσιμο σε κάθε clock του ρολογιού του *fp_mult_top*. Παρόλα αυτά ο διαχωρισμός σε 3 στάδια μειώνει αρκετά το βάθος, χωρίς να μεγαλώνει την ανάγκη για extra clock ticks. Αντ' αυτού μειώνει το συνολικό latency. Τα στάδια που χρησιμοποιήθηκαν είναι με τη σειρά τα εξής:

1. **Normalize**: Όπου καλείται το normalize_mult.
2. **Round**: Όπου καλείται το round_mult και το
3. **Exception handler**: Όπου καλείται το exception_mult

Για το pipeline εργαστήκαμε με γνώμονα τη διατήρηση όλης της απαραίτητης πληροφορίας στα στάδια, ούτως ώστε ο χρήστης να μπορεί σε κάθε clock να τροφοδοτεί το module με διαφορετικούς αριθμούς για πολλαπλασιασμό. Όλα τα σήματα από την είσοδο τροφοδοτούνται από το ένα στάδιο στο επόμενο. Η μετάδοση αυτή σταματάει μόνο αν τα σήματα δεν είναι απαραίτητα για τη λειτουργία του επόμενου module. Με αυτό τον τρόπο για παράδειγμα όλη η απαραίτητη πληροφορία για το κάθε module, περνάει από το ένα στάδιο στο επόμενο σε κάθε clock. Για την ακρίβεια, μετά τον αρχικό υπολογισμό του πρόσημου καθώς και του αποτελέσματος του πολλαπλασιασμού για τον εκθέτη και τη mantissa:

- Οι **αριθμοί** προς πολλαπλασιασμό φτάνουνε μέχρι τον exception handler.
- Ο **εκθέτης** και η **mantissa** φτάνει μέχρι τον exception handler, όπου και συνθέτει τον τελικό αριθμό πριν την είσοδο.
- Το υπολογισμένο **πρόσημο** του τελικού αριθμού φτάνει μέχρι το round module.
- Τα **guard** και **sticky** bits φτάνουν από το normalize μέχρι το round module.
- Το **inexact** bit από το round μέχρι το exception handler.

Ενώ επιλέξαμε να υποστηρίξουμε με pipeline όλη σχεδόν την πληροφορία, ένα σήμα δεν ακολουθεί την οδό των σταδίων. Πρόκειται για το **round**. Ενώ για τα υπόλοιπα θεωρήσαμε πως ο χρήστης πρέπει να μπορεί να τα αλλάζει σε κάθε clock του ρολογιού, αυτό υποθέσαμε ότι θα επιλέγεται κατά την έναρξη και θα **παραμένει αμετάβλητο κάθ' όλη τη διάρκεια λειτουργίας**. Για το λόγο αυτό, δεν χρειάστηκε να το "περάσουμε" από τα στάδια.

3. Λεπτομερειες Υλοποιησης

3.1. `fp_mult.sv` — Floating Point Multiplier (Top Module)

Το module `fp_mult` υλοποιεί έναν πλήρως καταχωρημένο πολλαπλασιαστή αριθμών κινητής υποδιαστολής μονής ακρίβειας (IEEE-754). Αποτελείται από ένα pipeline 3 σταδίων:

- **Stage 0 (Decode)**: Εξάγει πρόσημο, εκθέτες και mantissas από τις εισόδους `a`, `b`. Υπολογίζει το αρχικό γινόμενο mantissas και το bias-adjusted εκθέτη. Η λειτουρία αυτή βρίσκεται σε ένα section με combinational logic.

```
sign  = a[31] ^ b[31];
exp_a = a[30:23];
exp_b = b[30:23];
s0_exponent = exp_a + exp_b - 127;

mant_a = (exp_a == 0) ? {1'b0, a[22:0]} : {1'b1, a[22:0]};
mant_b = (exp_b == 0) ? {1'b0, b[22:0]} : {1'b1, b[22:0]};
s0_mantissa = mant_a * mant_b;
```

- **Stage 1 (Normalize)**: Κανονικοποιεί το αποτέλεσμα, προσαρμόζοντας την mantissa και τον εκθέτη. Παράγει επίσης guard και sticky bits για στρογγυλοποίηση.
- **Stage 2 (Round)**: Εφαρμόζει τον επιλεγμένο τρόπο στρογγυλοποίησης και υπολογίζει την τελική

mantissa και εκθέτη καθώς και το σήμα inexact.

- **Stage 3 (Exception Handling)**: Εντοπίζει καταστάσεις όπως NaN, infinities, overflow/underflow, μηδενικά και υπολογίζει την τελική έξοδο `z` και το flag `status`.

Υποστηρίζει ασύγχρονη επαναφορά (`rst`) και συγχρονισμό μέσω ρολογιού (`clk`).

3.2. `normalize_mult.sv` — Normalize Stage

Το module αυτό δέχεται ένα 48-bit mantissa και έναν 10-bit εκθέτη. Ελέγχει εάν χρειάζεται shift για κανονικοποίηση της mantissa (ώστε να ξεκινά από MSB=1) και αντίστοιχα προσαρμόζει τον εκθέτη.

- `mantissa_out`: Κανονικοποιημένη mantissa (23 bits)
- `guard_bit, sticky_bit`: Χρήσιμα για την ακρίβεια στη στρογγυλοποίηση
- `exponent_out`: Νέος εκθέτης μετά την κανονικοποίηση

3.3. `round_mult.sv` — Round Stage

Το module εφαρμόζει έναν από τους υποστηριζόμενους τρόπους στρογγυλοποίησης σύμφωνα με το IEEE-754 standard:

- Round to Nearest Even
- Round Toward Zero
- Round Toward inf
- Round Toward $-$ inf
- Near-Up
- Away from Zero

Για κάθε διαφορετικό mode με βάση το round υλοποιεί διαφορετική προσέγγιση. Για παράδειγμα για IEEE nearest event έχουμε:

```
if (guard_bit && (sticky_bit || mantissa_extended[0])) begin
    mantissa_rounded = mantissa_extended + 1;
end
```

Ενώ για IEEE pinf:

```
if (!sign_in && (guard_bit | sticky_bit)) begin
    mantissa_rounded = mantissa_extended + 1;
end
```

Η υλοποίηση των modes έγινε με διαφορετικές συναρτήσεις, μέσα στο module. Οι είσοδοι περιλαμβάνουν την mantissa, τον εκθέτη, το πρόσημο και τα bits `guard` και `sticky`. Οι έξοδοι είναι η νέα mantissa και ο εκθέτης μετά τη στρογγυλοποίηση, καθώς και το σήμα `inexact`.

3.4. `round_modes.sv` — Round Mode Helpers

Ορίζει ένα enumeration τύπο `round_mode_t` για τους 6 υποστηριζόμενους τρόπους στρογγυλοποίησης, που χρησιμοποιούνται κυρίως κατά τον χειρισμό overflow και underflow στην εξαίρεση.

```
typedef enum logic [2:0] {
    RND_IEEE_NEAREST_EVEN = 3'd0,
    RND_IEEE_ZERO         = 3'd1,
    RND_IEEE_PINF         = 3'd2,
    RND_IEEE_NINF         = 3'd3,
```

```
    RND_NEAR_UP          = 3'd4,
    RND_AWAY_ZERO        = 3'd5
} round_mode_t;
```

3.5. `exception_mult.sv` — Exception Handling

Αυτό το module χειρίζεται corner cases και ειδικές τιμές:

- Ανίχνευση και χειρισμός NaNs, infinities, zero
- Overflow/underflow μετά τη στρογγυλοποίηση
- Ορισμός της τελικής τιμής `z`
- Ορισμός των status flags: `zero_f`, `inf_f`, `nan_f`, `tiny_f`, `huge_f`, `inexact_f`

Χρησιμοποιεί ένα `enum interp_t` για την κατηγοριοποίηση αριθμών (ZERO, INF, NORM, MIN_NORM, MAX_NORM) και δύο συναρτήσεις:

- `num_interp()`: Επιστρέφει το είδος του αριθμού (π.χ., αν είναι infinity ή zero)
- `z_num()`: Επιστρέφει την bitwise αναπαράσταση ενός αριθμού βάσει του είδους του

Η λογική ελέγχει τους συνδυασμούς εισόδων και εφαρμόζει τις κατάλληλες ενέργειες βάσει πίνακα (table 4) και του rounding mode. Ο βασικός κορμός είναι μια switch case όπου γίνεται το dispatch όλων των corner cases:

```
a_class = num_interp(a);
b_class = num_interp(b);

case ({a_class, b_class})
    {ZERO, ZERO}, {ZERO, NORM}, {NORM, ZERO}: // ...
    {ZERO, INF}, {INF, ZERO}: // ...
    {INF, INF}, {NORM, INF}, {INF, NORM}: // ...
    default: // {NORM, NORM} ...
endcase
```

Ειδικά στο default βρίσκεται η λογική του extra rounding όταν έχουμε NaN, Inf, κλπ.

4. Επαληθευση και Επικυρωση

Όπως αναφέραμε παραπάνω οι επαλήθευση έγινε σταδιακά. Σε κάθε αρχείο στον κατάλογο *sim/* υπάρχουν tests που επιβεβαιώνουν την ορθή λειτουργία του κάθε module. Για να τρέξουμε τα test χρησιμοποιήσαμε το vsim. Για παράδειγμα για τα test του round_mult:

```
vlog sim/round_mult_tb
vsim -voptargs=+acc work.round_tb
run 200ns
```

Όλες οι εκτελέσεις εμφανίζουν μηνύματα στην κονσόλα με τα αποτελέσματα.

Τα τελικά tests που ανήκουν στο fp_mult_top_tb έχουν ελέγχους που ακολουθούν κάποια test vectors όπως παρακάτω:

```
initial begin
    $display("Starting fp_mult test...\n");

    // Test cases
    tests[0] = '{32'h3f800000, 32'h40000000, 32'h40000000, "+1.0 * +2.0 = +2.0"};
    tests[1] = '{32'h40400000, 32'h40400000, 32'h41100000, "+3.0 * +3.0 = +9.0"};
```

```
    tests[2] = '{32'hc0400000, 32'h40400000, 32'hc1100000, "-3.0 * +3.0 = -9.0"};
    tests[3] = '{32'hbf800000, 32'h40000000, 32'hc0000000, "-1.0 * +2.0 = -2.0"};
    // ...
end
```

Με βάση αυτά τα vectors, παράγονται μηνύματα που έχουν τη μορφή:

```
# [8] +12.34 * -0.0001 = -0.001234
#     A=414570a4 B=b8d1b717 => Z=baa1be2b (expected baa1be2b) PASS
```

Ένα στιγμιότυπο από την έξοδο φαίνεται στην εικόνα 1.

```
# Loading work.round_mult(fast)
# Loading work.exception_mult_sv_unit(fast)
# Loading work.exception_mult(fast)
VSIM 8> run 400ns
# Starting fp_mult test...
#
# [1] +1.0 * +2.0 = +2.0
#     A=3f800000 B=40000000 => Z=40000000 (expected 40000000) PASS
#
# [2] +3.0 * +3.0 = +9.0
#     A=40400000 B=40400000 => Z=41100000 (expected 41100000) PASS
#
# [3] -3.0 * +3.0 = -9.0
#     A=c0400000 B=40400000 => Z=c1100000 (expected c1100000) PASS
#
# [4] -1.0 * +2.0 = -2.0
#     A=bf800000 B=40000000 => Z=c0000000 (expected c0000000) PASS
#
# [5] +0.5 * +0.5 = +0.25
#     A=3f000000 B=3f000000 => Z=3e800000 (expected 3e800000) PASS
#
# [6] 1.0 * +0.0 = +0.0
#     A=3f800000 B=00000000 => Z=00000000 (expected 00000000) PASS
#
# [7] +42.0 * -7.0 = -294.0
#     A=42280000 B=c0e00000 => Z=c3930000 (expected c3930000) PASS
#
# [8] +12.34 * -0.0001 = -0.001234
#     A=414570a4 B=b8d1b717 => Z=baa1be2b (expected baa1be2b) PASS
#
# [9] 0.0 * 0.0 = 0.0
#     A=00000000 B=00000000 => Z=00000000 (expected 00000000) PASS
#
```

Σχήμα 1: *Στιγμιότυπο εξόδου.*

5. Συμπερασματα

Μέσα από την παρούσα εργασία υλοποιήθηκε με επιτυχία μια πλήρως καταχωρημένη αριθμητική μονάδα πολλαπλασιασμού αριθμών κινητής υποδιαστολής, σύμφωνα με το πρότυπο IEEE-754. Η υλοποίηση με χρήση pipeline τριών σταδίων προσέφερε βελτιωμένη απόδοση και σαφή διαχωρισμό λειτουργιών, καθιστώντας τη σχεδίαση πιο ευέλικτη και επεκτάσιμη. Ο διαχωρισμός σε modules με αυστηρά ορισμένα interfaces διευκόλυνε την ανάπτυξη και τον έλεγχο των επιμέρους λειτουργιών, ενώ η προσέγγιση βασισμένη σε επαλήθευση (test-driven development) διασφάλισε την ορθότητα κάθε σταδίου.

Αν και δεν καλύφθηκαν όλοι οι δυνατοί συνδυασμοί corner cases, η λογική του κυκλώματος και οι ενδείξεις των αποτελεσμάτων επαληθεύουν ότι η λειτουργία του multiplier είναι σε μεγάλο βαθμό ορθή. Η εργασία παρείχε μια ουσιαστική εμπειρία στην πρακτική υλοποίηση σύνθετων αριθμητικών μονάδων σε χαμηλό επίπεδο, αναδεικνύοντας τις προκλήσεις αλλά και τις δυνατότητες της σύγχρονης ψηφιακής σχεδίασης.