Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Energy-Efficient Design and Implementation of Approximate Floating-Point Multiplier

Διπλωματικη Εργασια

της

ΘΕΟΔΩΡΑΣ ΠΑΠΑΡΟΥΝΗ

Επιβλέπων: Κιαμάλ Πεκμεστζή
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Αύγουστος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Energy-Efficient Design and Implementation of Approximate Floating-Point Multiplier

Διπλωματικη Εργασια

της

ΘΕΟΔΩΡΑΣ ΠΑΠΑΡΟΥΝΗ

Επιβλέπων: Κιαμάλ Πεκμεστζή
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Αυγούστου 2019.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Κιαμάλ Πεκμεστζή	Δημήτριος Σούντρης	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π

Αθήνα, Αύγουστος 2019

(Υπογραφή)

.....

ΘΕΟΔΩΡΑ ΠΑΠΑΡΟΥΝΗ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στον τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών στη διάρκεια του ακαδημαϊκού έτους 2018-2019 και επισφραγίζει τις σπουδές μου στο Εθνικό Μετσόβιο Πολυτεχνείο. Θα ήθελα ιδιαιτέρως να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ.Πεκμεστζή Κιαμάλ για την καθοδήγησή του και για την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον και επίκαιρο αντικείμενο. Επίσης, θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα Λέοντα Βασίλειο για την εξαιρετική συνεργασία και για την πολύτιμη βοήθεια που μου παρείχε όποτε τη χρειάστηκα.
Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου για την άπλετη και πολύπλευρη στήριξη και κατανόηση καθόλη τη διάρκεια των σπουδών μου, καθώς και κάποιους πολύ δικούς μου και σημαντικούς για εμένα ανθρώπους για την υπομονή τους και τη σημαντική υποστήριξη όλο αυτό το διάστημα και ιδιαίτερα τους τελευταίους μήνες.

Περίληψη

Το Πρότυπο IEEE 754 κινητής υποδιαστολής είναι η πιο κοινή αναπαράσταση για πραγματικούς αριθμούς σε υπολογιστές. Οι αριθμητικές μονάδες κινητής υποδιαστολής χρησιμοποιούνται σε πολλές εφαρμογές, όπως η ψηφιακή επεξεργασία σήματος και η μηχανική μάθηση. Οι πολλαπλασιαστές είναι από τις σημαντικότερες αριθμητικές μονάδες και χρησιμοποιούνται σε ένα ευρύ φάσμα ψηφιακών συστημάτων. Ωστόσο, μια μονάδα πολλαπλασιασμού απαιτεί υψηλό υπολογιστικό φορτίο, το οποίο οδηγεί σε σημαντική κατανάλωση ενέργειας. Ο υπολογισμός κατά προσέγγιση είναι ένας τρόπος για να σχεδιάσουμε γρήγορα και ενεργειακά αποδοτικά συστήματα.

Αυτή η διπλωματική εργασία παρουσιάζει μια προσέγγιση του πολλαπλασιαστή αριθμών κινητής-υποδιαστολής. Η προσέγγιση εφαρμόζεται στον πολύπλοκο πολλαπλασιασμό των αριθμών mantissa. Η εφαρμογή του προτεινόμενου κατά προσέγγιση πολλαπλασιαστή στα 16 ψηφία μειώνει τα delay, area, energy έως και 32%, 54%, 53% αντίστοιχα σε σχέση με τον ακριβή πολλαπλασιαστή, ενώ παρουσιάζει σφάλμα μικρότερο από 3,4%. Επίσης, η εφαρμογή του προτεινόμενου κατά προσέγγιση πολλαπλασιαστή στα 32 ψηφία μειώνει τα delay, area, energy έως και 46%, 83%, 82% αντίστοιχα σε σύγκριση με τον ακριβή πολλαπλασιαστή, ενώ παρουσιάζει σφάλμα μικρότερο από 2,2%. Ο προτεινόμενος κατά προσέγγιση πολλαπλασιαστής αριθμών κινητής υποδιαστολής αξιολογήθηκε και σε επίπεδο εφαρμογής.

Λέξεις Κλειδιά

Προσεγγιστικός Υπολογισμός, Αριθμητικά Κυκλώματα, Αριθμοί Κινητής Υποδιαστολής, Πολλαπλασιασμός, Πρότυπο IEEE 754, Σχεδίαση ASIC, Ανοχή στα Σφάλματα, Κατανάλωση Ενέργειας

Abstract

IEEE Standard 754 floating-point is the most common representation today for real numbers on computers. Floating-point (FP) arithmetic units are used in many applications, such as digital signal processing and machine learning. Multipliers are the most important operational units and are used in a wide range of digital systems. However, a multiplication unit suffer from high computational load, which leads to considerable power consumption. Approximate computing is a way to design fast and energy efficient systems.

This diploma thesis describes an approximation of floating-point multiplier. The approximation is applied to the costly mantissa multiplication. The implementation of proposed approximate FP multiplier in 16 bit reduces delay, area and energy up to 32%, 54%, and 53% respectively compared with the exact multiplier, while incurring in an error less than 3.4%. Also, the implementation of proposed approximate FP multiplier in 32 bit reduces delay, area and energy up to 46%, 83%, and 82% respectively compared with the exact multiplier, while incurring in an error less than 2.2%. The proposed approximate FP multiplier was also evaluated at application level.

Keywords

Approximate Computing, Arithmetic Circuits, Floating-Point Numbers, Multiplication, IEEE Standard 754, ASIC Design, Error Tolerance, Energy Efficiency

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Εισαγωγή

Η "floating-point" αριθμητική χρησιμοποιείται ευρέως σε πολλούς τομείς, και ειδικά για επιστημονικούς υπολογισμούς καθώς και για επεξεργασία σημάτων. Με το πέρασμα των χρόνων έχουν αναπτυχθεί πολλές προσεγγίσεις σε αριθμούς. Η "floating-point" αριθμητική είναι η πιο αποτελεσματική αναπαράσταση πραγματικών αριθμών, και ειδικά στον τομέα των υπολογιστών, γιατί δίνει τη δυνατότητα να αναπαρασταθούν πολύ μικροί και πολλοί μεγάλοι αριθμοί.

Ο όρος "floating-point" αναφέρεται στο γεγονός ότι η υποδιαστολή ενός αριθμού μπορεί να τοποθετηθεί σε οποιαδήποτε θέση του αριθμού. Η μετατόπιση της υποδιαστολής μεταβάλλει και τον εκθέτη που περιλαμβάνεται στον αριθμό.

Από την εμφάνιση του πρώτου υπολογιστή έχουν χρησιμοποιηθεί διάφορες "floating-point" αναπαραστάσεις. Το 1985, όμως, καθιερώθηκε το πρότυπο IEEE 754 για την αναπαράσταση "floating-point".

Η υπολογιστική ταχύτητα των πράξεων με αριθμούς "floating-point" μετράτε σε "FLOPS", και είναι ένα πολύ σημαντικό μέγεθος στα υπολογιστικά συστήματα. Τα συστήματα ηλεκτρονικών υπολογιστών διαθέτουν μια ειδικά σχεδιασμένη "floating-point" μονάδα η οποία εκτελεί τις πράξεις μεταξύ "floating-point" αριθμών.

Σύντομο Θεωρητικό Υπόβαθρο

Δυαδικό Σύστημα Αριθμών

Το δυαδικό σύστημα είναι το βασικό αριθμητικό σύστημα των ηλεκτρονικών συστημάτων. Κάθε δυαδικός αριθμών εκφράζεται με δυνάμεις του 2, και αποτελείται αποκλειστικά από τα ψηφία "0" και "1". Κάθε ψηφίο ονομάζεται bit.

Κάθε αριθμός αναπαρίσταται στο δυαδικό σύστημα ως εξής:

$$a = \sum_{i=0}^{n-1} 2^i * a_i \tag{0.1}$$

όπου, a : είναι η τιμή του αριθμού,

Πίνακας 1: Πίνακας κωδικοποίησης Modified Booth

Δυαδικά ψηφία			Ψηφίο Modified Booth	Κωδικοποιημένο ψηφίο y_i		
y_{2j+1}	y_{2j}	y_{2j-1}		$sign = s_j$	$x1 = one_j$	$x2 = two_j$
0	0	0	0	0	0	0
0	0	1	+1	0	1	0
0	1	0	+1	0	1	0
0	1	1	+2	0	0	1
1	0	0	-2	1	0	1
1	0	1	-1	1	1	0
1	1	0	-1	1	1	0
1	1	1	0	1or0	0	0

b : είναι ο αριθμός των ψηφίων του αριθμού,

a_i : είναι η τιμή κάθε ψηφίου.

Για παράδειγμα, ο αριθμός 17 αναπαρίσταται στο δυαδικό σύστημα ως 10001. Χρησιμοποιώντας την εξίσωση (0.1) ο δυαδικός αριθμός 10001 αντιστοιχεί στον:

$1 * 2^0 + 0 * 2^1 + 0 * 2^2 + 0 * 2^3 + 1 * 2^4 = 17_{10}$.

Η μέγιστη τιμή ενός αριθμού που αποτελείται από n ψηφία είναι $2^n - 1$ και η ελάχιστη είναι 0. Το πρώτο ψηφίο ενός αριθμού ονομάζεται Most Significant Bit (MSB) και το τελευταίο ψηφίο του αριθμού ονομάζεται Least Significant Bit (LSB). Στο παραπάνω παράδειγμα με τον αριθμό 10001 το ψηφίο που πολλαπλασιάζεται με το 2^4 είναι το MSB και το ψηφίο που πολλαπλασιάζεται με το 2^0 είναι το LSB. Σε κάθε δυαδικό αριθμό το ψηφίο που πολλαπλασιάζεται με το 2^0 είναι το LSB.

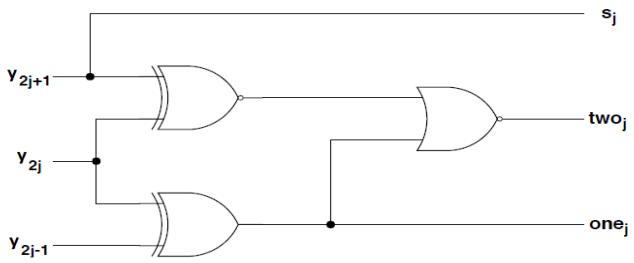
Η αριθμητική στο δυαδικό σύστημα είναι παρόμοια με την αριθμητική των άλλων αριθμητικών συστημάτων. Στους δυαδικούς αριθμούς εκτελούνται όλες οι πράξεις, όπως η πρόσθεση, η αφαίρεση, ο πολλαπλασιασμός και η διαίρεση. Η πρόσθεση είναι παρόμοια με αυτή των δεκαδικών αριθμών. Στην παρούσα διπλωματική εργασία θα ασχοληθούμε με τον πολλαπλασιασμό.

Τροποποιημένος αλγόριθμος του Booth (Modified Booth algorithm)

Έστω δύο αριθμοί X, Υ σε συμπλήρωμα ως προς 2. Ο Υ δίνεται από τη σχέση:

$$Y = \sum_{j=0}^{n/2-1} {y_j}^{MB} * 4^j \tag{0.2}$$

Στον επόμενο πίνακα 1 φαίνεται πώς διαμορφώνονται τα σήματα κατά τον τροποποιημένο αλγόριθμο του Booth (radix-4 encoding)



Σχήμα 1: One, Two και S σήματα

Οι λογικές εξισώσεις που περιγράφουν τον πίνακα 1 είναι:

$$one_j = y_{2j-1} \oplus y_{2j} \tag{0.3}$$

$$two_j = (y_{2j+1} \oplus y_{2j}) * \overline{one}_j \tag{0.4}$$

$$s_j = y_{2j+1} \tag{0.5}$$

Στο σχήμα 1 φαίνεται το κύκλωμα που υλοποιεί τις προαναφερθείσες εξισώσεις.

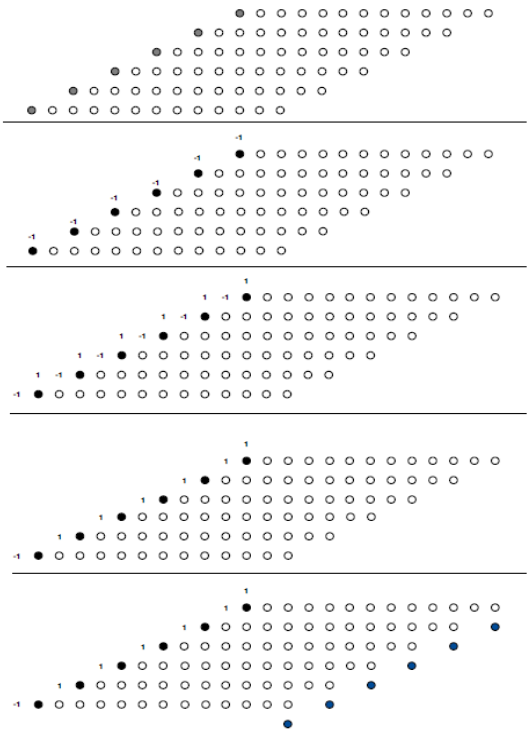
Στη συνέχεια παρουσιάζουμε τον τρόπο με τον οποίο παράγονται τα μερικά γινόμενα. Η εξίσωση (0.6) περιγράφει το γινόμενο P των δύο αριθμών X, Y, με τον αριθμό Y να είναι κωδικοποιημένος κατά τον τροποποιημένο αλγόριθμο του Booth.

$$P = X * Y = \sum_{j=0}^{n/2-1} X * {y_j}^{MB} * 2^{2j} = ct + \sum_{j=0}^{n/2-1} PP_j * 2^{2j} \tag{0.6}$$

όπου ct είναι ο διορθωτικός όρος και PP_j τα μερικά γινόμενα.

Ο διορθωτικός όρος ct είναι αναγκαίος για την εξαγωγή σωστού αποτελέσματος και στην περίπτωση που κάποια μερικά γινόμενα είναι αρνητικοί αριθμοί. Στο σχήμα 2 φαίνονται τα στάδια που ακολουθούνται ώστε να προκύψει ο όρος ct σε έναν πολλαπλασιαστή 12 ψηφίων.

Όπως φαίνεται στο σχήμα 2 τα γκρι κυκλάκια αντιπροσωπεύουν την επέκταση προσήμου όταν ο X πολλαπλασιάζεται με το 2 ($two_j = 1$). Το MSB κάθε μερικού γινομένου έχει αρνητικό βάρος οπότε τα γκρι κυκλάκια αντικαθίσταται από τα μαύρα ($\overline{p}$) και προστίθεται επιπλέον ο παράγοντας -1 ($-p = \overline{p} - 1$). Στη συνέχεια χρησιμοποιώντας το μαθηματικό



Σχήμα 2: Μερικά γινόμενα και διορθωτικός όρος

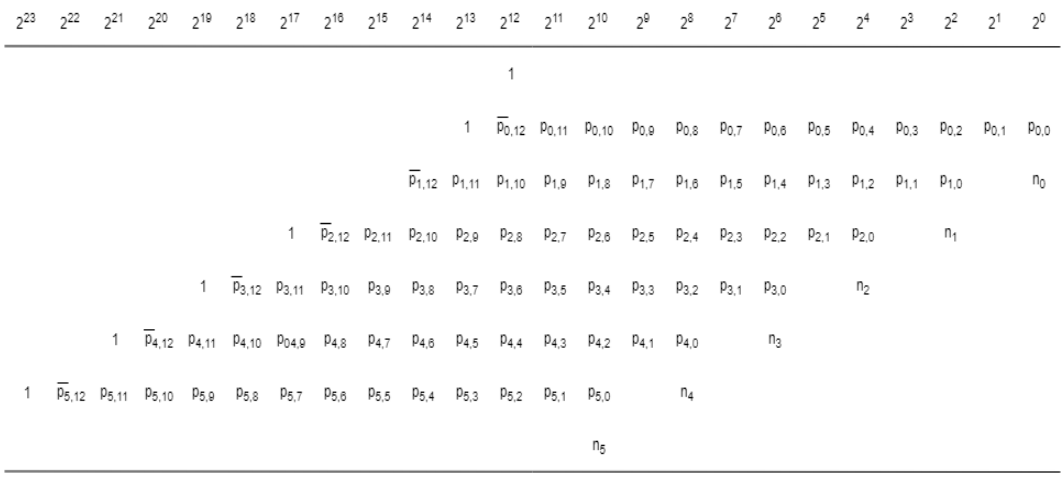
κόλπο (2 - 1 = 1) το -1 αντικαθίσταται από το (-2 + 1). Έπειτα πραγματοποιούνται οι αφαιρέσεις που προέκυψαν από το προηγούμενο στάδιο. Στο τελευταίο στάδιο προστίθενται και τα μπλε κυκλάκια που αντιπροσωπεύουν τα διορθωτικά ψηφία (n_j). Αν το κωδικοποιημένο κατά Modified Booth ψηφίο έχει αρνητικό βάρος τότε το διορθωτικό ψηφίο παίρνει την τιμή 1, ώστε να γίνει η μετάβαση από το συμπλήρωμα ως προς 2 στο συμπλήρωμα ως προς ένα. Αν όμως το βάρος είναι θετικό, τότε το διορθωτικό ψηφίο παίρνει την τιμή 0.

Ο διορθωτικός όρος ct υπολογίζεται από την παρακάτω σχέση:

$$ct = \sum_{j=0}^{n/2-1} \left(n_j * 2^{2j}\right) + 2^n * \left(1 + \sum_{j=0}^{n/2-2} \left(2^{2j+1}\right) - 2^{n-1}\right) \tag{0.7}$$

Τα μερικά γινόμενα υπολογίζονται με τη χρήση των σημάτων one_j, two_j και s_j μέσω των επόμενων εξισώσεων:

$$PP_j = \overline{p}_{j,n} * 2^{n+2j} + \sum_{i=0}^{n-1} \left(p_{j,i} * 2^{i+2j}\right) \tag{0.8}$$



Σχήμα 3: Πίνακα μερικών γινομένων μαζί με τον διορθωτικό όρο για έναν πολλαπλασιαστή 12x12

$$p_{j,i} = ((x_i \oplus s_j) * one_j) + ((x_{i-1} \oplus s_j) * two_j) \tag{0.9}$$

$$p_{j,0} = ((x_0 \oplus s_j) * one_j) + (s_j * two_j) \tag{0.10}$$

$$\overline{p}_{j,n} =! (((x_{n-1} \oplus s_j) * one_j) + ((x_{n-1} \oplus s_j) * two_j)) \tag{0.11}$$

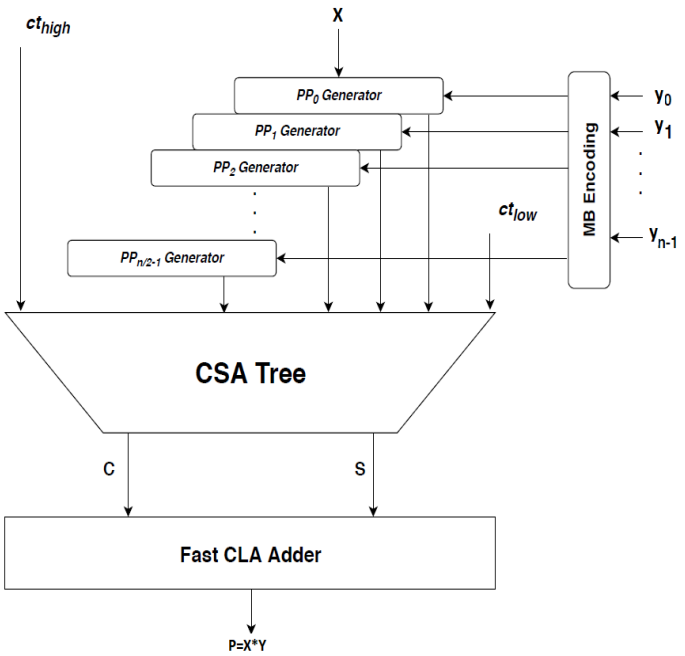
$$n_j = s_j * (one_j + two_j) \tag{0.12}$$

Χρησιμοποιώντας όλα τα παραπάνω το σχήμα 3 έχουμε τον πίνακα των μερικών γινομένων μαζί με τον διορθωτικό όρο για έναν πολλαπλασιαστή 12x12.

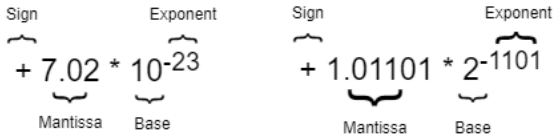
Ο πολλαπλασιαστής που αποτελείται από τις γεννήτριες παραγωγής μερικών γινομένων, το MB κωδικοποιητή, το CSA Wallace δέντρο και το γρήγορο CLA αθροιστή φαίνεται στο σχήμα 4.

Floating-point αριθμοί

Οι αριθμοί "floating-point" (κινητής υποδιαστολής) είναι ένας πιθανός τρόπος αναπαράστασης των πραγματικών αριθμών. Το πρότυπο IEEE 754 [1] παρουσιάζει δύο διαφορετι-



Σχήμα 4: Πολλαπλασιαστής τύπου Modified Booth



Σχήμα 5: Αναπαράσταση κινητής υποδιαστολής σε δεκαδική και σε δυαδική μορφή

κές μορφές κινητής υποδιαστολής, τη δυαδική μορφή και τη δεκαδική μορφή (σχήμα 5). Κάθε αριθμός σε μορφή κινητής υποδιαστολής αποτελείται από τρία μέρη: το πρόσημο, τη "mantissa" και τον εκθέτη. Η βάση (base or radix) είναι το 2 και το 10 για τους δυαδικούς τους δεκαδικούς αριθμούς αντίστοιχα.

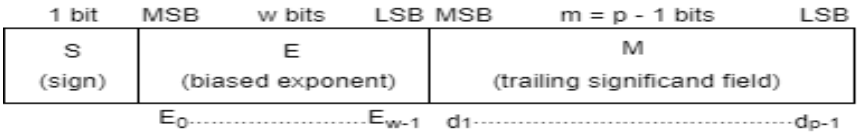
Στην παρούσα διπλωματική εργασία θα ασχοληθούμε με την κωδικοποίηση κινητής υποδιαστολής σε δυαδική μορφή και συγκεκριμένα με τη "half-precision" (που είναι κωδικοποίηση στα 16 ψηφία) και τη "single-precision" (που είναι κωδικοποίηση στα 32 ψηφία)

Κωδικοποίηση Κινητής Υποδιαστολής σε Δυαδική Μορφή

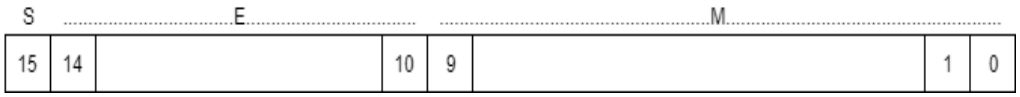
Κάθε αριθμός κινητής υποδιαστολής έχει μόνο μία κωδικοποίηση σε δυαδική μορφή. Ένας αριθμός κινητής υποδιαστολής που κωδικοποιείται σε δυαδική μορφή με χρήση k ψηφίων φαίνεται στο σχήμα 6.

Οι δυαδικοί αριθμοί με τη μορφή του σχήματος 6 αποτελούνται από το πρόσημο (το MSB), τον εκθέτη E που είναι τα επόμενα w ψηφία (E = e + bias), και τη "mantissa" που είναι τα τελευταία m ψηφία. Ανάλογα με την τιμή του E και του M οι αριθμοί κατηγοριοποιούνται ως εξής:

- "Normal" αριθμοί είναι αυτοί που το E παίρνει ακέραιες τιμές μεγαλύτερες από το 1 και



Σχήμα 6: Floating-point μορφή [1]



Σχήμα 7: IEEE half precision αναπαράσταση ενός αριθμού κινητής υποδιαστολής

μικρότερες από το $2^w - 2$, ανεξάρτητα από την τιμή του M.

- "Zero" αριθμοί είναι οι αριθμοί που έχουν E=0 και M=0.

- "Subnormal" αριθμοί είναι οι αριθμοί που έχουν E=0 και $M \neq 0$.

- "Infinite" αριθμοί είναι οι αριθμοί που έχουν E=2^w-1 και M=0.

- "NaN" αριθμοί είναι οι αριθμοί που έχουν E=2^w-1 και $M \neq 0$.

Παρακάτω παρουσιάζονται οι δύο κωδικοποιήσεις κινητής υποδιαστολής σε δυαδική μορφή με τις οποίες θα ασχοληθούμε στην παρούσα διπλωματική εργασία.

- **Half precision**

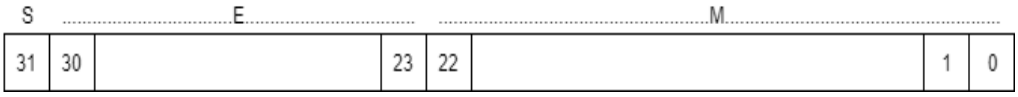
  Το σχήμα 7 δείχνει την αναπαράσταση "half precision" ενός αριθμού κινητής υποδιαστολής. Ο αριθμός αυτός αποτελείται από 16 ψηφία. Το πρώτο ψηφίο είναι το ψηφίο προσήμου (S), τα επόμενα 5 ψηφία αναπαριστούν τον εκθέτη (E), και τα 10 τελευταία ψηφία τη "mantissa" (M). Προσθέτοντας ένα επιπλέον ψηφίο μπροστά από τη "mantissa" έχουμε μια επιπλέον μεταβλητή που ονομάζεται "significand". Αν ο εκθέτης είναι μεγαλύτερος από το 0 και μικρότερος από το 31, και υπάρχει 1 στο (MSB) του "significand" τότε ο αριθμός είναι "normal". Η εξίσωση (0.13) περιγράφει έναν "normal" αριθμό.

$$Z = (-1)^S * 2^{(E-Bias)} * (1.M) \tag{0.13}$$

  Όπου: $M = m_9 * 2^{-1} + m_8 * 2^{-2} + m_7 * 2^{-3} + ... + m_1 * 2^{-9} + m_0 * 2^{-10}$,

  ανδ Βιας $= 15$.

  Η μικρότερη τιμή του εκθέτη είναι το $E_{min} = 00001_2 - 01111_2 = -14$, και η μεγαλύτερη είναι το $E_{max} = 11110_2 - 01111_2 = 15$. Επίσης ο μικρότερος "normal" αριθμός είναι ο $2^{-14} \approx 6.10 * 10^{-5}$ και ο μεγαλύτερος "normal" αριθμός είναι ο $(2 - 2^{-10}) * 2^{15} \approx 65504$.



Σχήμα 8: IEEE single precision αναπαράσταση ενός αριθμού κινητής υποδιαστολής

- **Single precision**

  Το σχήμα 8 δείχνει την αναπαράσταση "single precision" ενός αριθμού κινητής υποδιαστολής. Ο αριθμός αυτός αποτελείται από 32 ψηφία. Το πρώτο ψηφίο είναι το ψηφίο προσήμου (S), τα επόμενα 8 ψηφία αναπαριστούν τον εκθέτη (E), και τα 23 τελευταία ψηφία τη "mantissa" (M). Προσθέτοντας ένα επιπλέον ψηφίο μπροστά από τη "mantissa" έχουμε μια επιπλέον μεταβλητή που ονομάζεται "significand". Αν ο εκθέτης είναι μεγαλύτερος από το 0 και μικρότερος από το 255, και υπάρχει 1 στο (MSB) του "significand" τότε ο αριθμός είναι "normal". Όπως αναφέραμε η εξίσωση (0.13) περιγράφει έναν "normal" αριθμό.

  Όπου: $M = m_{22} * 2^{-1} + m_{21} * 2^{-2} + m_{20} * 2^{-3} + ... + m_1 * 2^{-22} + m_0 * 2^{-23}$,

  ανδ Βιας $= 127$.

  Η μικρότερη τιμή του εκθέτη είναι το $E_{min} = 01_H - 7F_H = -126$, και η μεγαλύτερη είναι το $E_{max} = FE_H - 7F_H = 127$. Επίσης ο μικρότερος "normal" αριθμός είναι ο $2^{-126} \approx 1.18 * 10^{-38}$ και ο μεγαλύτερος "normal" αριθμός είναι ο $(2 - 2^{-23}) * 2^{127} \approx 3.40 * 10^{38}$.

Προτεινόμενος προσεγγιστικός πολλαπλασιαστής αριθμών κινητής υποδιαστολής

Στη διπλωματική αυτή εργασία αρχικά υλοποιήσαμε τον ακριβή πολλαπλασιαστή δύο αριθμών κινητής υποδιαστολής.

Ακριβής πολλαπλασιαστής αριθμών κινητής υποδιαστολής

Έστω A, B δύο "normal" αριθμοί κινητής υποδιαστολής. Σύμφωνα με την εξίσωση (0.13) έχουμε:

$$A = (-1)^{S_A} * 2^{(E_A - Bias)} * (1.M_A) \tag{0.14}$$

ανδ

$$B = (-1)^{S_B} * 2^{(E_B - Bias)} * (1.M_B) \tag{0.15}$$

Πίνακας 2: Κατηγοριοποίηση σε Overflow,Underflow,Normal ανάλογα με την τιμή του E (binary16)

Κατηγορία	E	Σχόλια
Underflow	E<15	Δεν μπορεί να αλλάξει κατά την κανονικοποίηση
	E=15	Μπορεί να γίνει normal αριθμός κατά την κανονικοποίηση (αν προστεθεί 1 στον εκθέτη)
Normal αριθμοί	$16 \leq E \leq 45$	Μπορεί να μετατραπεί σε overflow κατά την κανονικοποίηση
Overflow	E>45	Δεν μπορεί να αλλάξει κατά την κανονικοποίηση

Χρησιμοποιώντας τις παραπάνω εξισώσεις το γινόμενο των δύο αριθμών δίνεται από την ακόλουθη εξίσωση:

$$P = A * B = (-1)^{(S_A+S_B)} * 2^{(E_A+E_B-2*Bias)} * (1.M_A * 1.M_B) \tag{0.16}$$

Για τον πολλαπλασιασμό των δύο αριθμών ακολουθούνται τα επόμενα 7 στάδια.

1. Το πρόσημο S_P του γινομένου προκύπτει αν εφαρμόσουμε τη λογική πύλη XOR στα πρόσημα S_A και S_B. Αν οι αριθμοί A και B έχουν το ίδιο πρόσημο τότε το P έχει πρόσημο 0 αλλιώς έχει πρόσημο 1.

2. Προσθέτουμε τους εκθέτες: $E = E_A + E_B$

3. Εξετάζουμε τις περιπτώσεις να έχουμε underflow ή overflow.

   Ανάλογα με την τιμή του E οι αριθμοί κατηγοριοποιούνται όπως φαίνεται στους πίνακες 2 (για 16x16) και 3 (για 32x32).

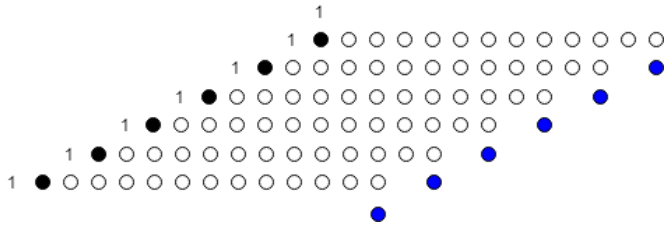
   Οι πίνακες 2 και 3 δείχνουν ότι η περίπτωση uderflow μπορεί να οδηγήσει σε "normal" αριθμό, καθώς και ότι ένας "normal" αριθμός μπορεί να μεταβεί σε overflow. Αυτές οι δύο περιπτώσεις θα εξεταστούν στο στάδιο 7.

4. Για να πάρουμε την τιμή του εκθέτη του τελικού γινομένου πρέπει να προσθέσουμε στο $(E_A + E_B - 2Bias)$ μια φορά το Bias: $E_P = E_A + E_B - 2Bias + Bias = E_A + E_B - Bias = E - Bias$

5. Για τον πολλαπλασιασμό των significands $(1.M_A, 1.M_B)$ θα χρησιμοποιήσουμε τον τροποποιημένο αλγόριθμο του Booth και το δέντρο Wallace που έχουμε περιγράψει παραπάνω.

Πίνακας 3: Κατηγοριοποίηση σε Overflow,Underflow,Normal ανάλογα με την τιμή του E (binary32)

Κατηγορία	E	Σχόλια
Underflow	E<127	Δεν μπορεί να αλλάξει κατά την κανονικοποίηση
	E=127	Μπορεί να γίνει normal αριθμός κατά την κανονικοποίηση (αν προστεθεί 1 στον εκθέτη)
Normal αριθμοί	$128 \leq E \leq 381$	Μπορεί να μετατραπεί σε overflow κατά την κανονικοποίηση
Overflow	E>381	Δεν μπορεί να αλλάξει κατά την κανονικοποίηση



Σχήμα 9: Το δέντρο των μερικών γινομένων του ακριβή πολλαπλασιαστή 12x12

– Στην περίπτωση του πολλαπλασιαστή 16x16 η mantissa έχει 10 ψηφία. Προσθέτοντας ένα επιπλέον ψηφίο παίρνουμε το significand. Έτσι το 1.M αποτελείται από 11 ψηφία. Επειδή όμως το 1.M δεν είναι αρνητικός αριθμός πρέπει να προσθέσουμε ένα επιπλέον 0 στην αρχή. Έτσι τελικά ο πολλαπλασιαστής μας θα είναι ένας 12x12 πολλαπλασιαστής. Οι δύο αριθμοί που πολλαπλασιάζονται είναι:

$X_{12} = 01.M_A$,

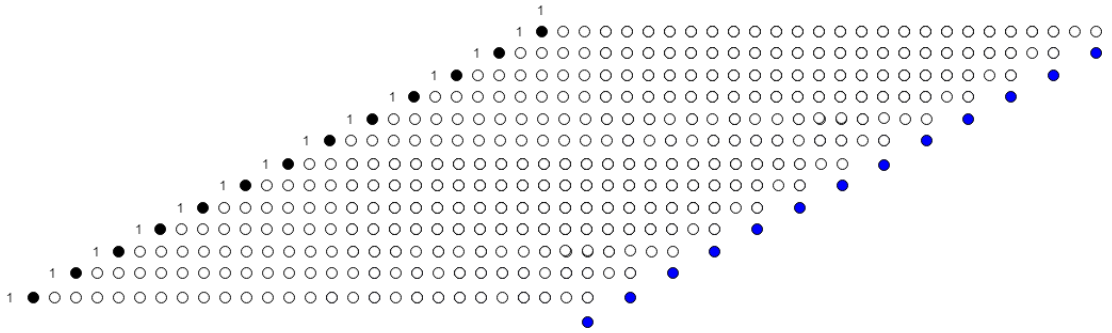
$Y_{12} = 01.M_B$.

Το σχήμα 9 δείχνει το δέντρο των μερικών γινομένων του ακριβή πολλαπλασιαστή 12x12 που χρησιμοποιεί τον τροποποιημένο αλγόριθμο του Booth.

– Στην περίπτωση του πολλαπλασιαστή 32x32 η mantissa έχει 23 ψηφία. Προσθέτοντας ένα επιπλέον ψηφίο παίρνουμε το significand. Έτσι το 1.M αποτελείται από 24 ψηφία. Επειδή όμως το 1.M δεν είναι αρνητικός αριθμός πρέπει να προσθέσουμε ένα επιπλέον 0 στην αρχή. Επειδή όμως ο αριθμός Υ κωδικοποιείται κατά τον τροποποιημένο αλγόριθμο του Booth χρειαζόμαστε και άλλο ένα 0 μπροστά από το 01.M. Έτσι τελικά ο πολλαπλασιαστής μας θα είναι ένας 26x26 πολλαπλασιαστής. Οι δύο αριθμοί που πολλαπλασιάζονται είναι:

$X_{26} = 001.M_A$,

$Y_{26} = 001.M_B$.



Σχήμα 10: Το δέντρο των μερικών γινομένων του ακριβή πολλαπλασιαστή 26x26

Σχήμα 11: Το γινόμενο (P) του πολλαπλασιαστή 12x12

Το σχήμα 10 δείχνει το δέντρο των μερικών γινομένων του ακριβή πολλαπλασιαστή 26x26 που χρησιμοποιεί τον τροποποιημένο αλγόριθμο του Booth.

6. Κανονικοποιούμε το αποτέλεσμα ώστε να υπάρχει μόνο ένα 1 μπροστά από την υποδιαστολή.

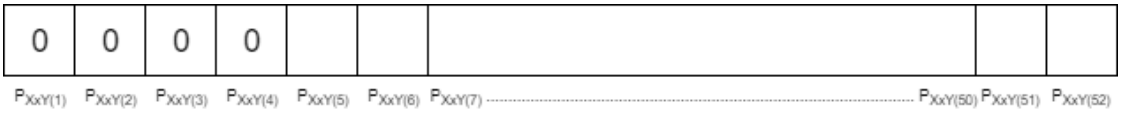
   – Στην περίπτωση του πολλαπλασιαστή 12x12 ψηφίων το αποτέλεσμα $P_{X\times Y}$ αποτελείται από 24 ψηφία. Τα πρώτα 2 ψηφία είναι 0. Το σχήμα 11 δείχνει το $P_{X\times Y} = 00xx.xxxx\ldots xxxx$.

   Αν $P_{X\times Y}(3) = 1$, η υποδιαστολή πρέπει να μετακινηθεί μία θέση αριστερά και ο εκθέτης E_P πρέπει να αυξηθεί κατά 1. Επειδή τα διαθέσιμα ψηφία της mantissa είναι 10, εφαρμόζουμε τη μέθοδο της περικοπής για τα υπόλοιπα. Η τιμή της mantissa είναι τα ψηφία από το $P_{X\times Y}(4)$ έως το $P_{X\times Y}(13)$ ($M_P = P_{X\times Y}(4)\ldots P_{X\times Y}(13)$).

   Αλλιώς, η τιμή του εκθέτη E_P δεν αλλάζει και η τιμή της mantissa είναι τα ψηφία από το $P_{X\times Y}(5)$ έως το $P_{X\times Y}(14)$ ($M_P = P_{X\times Y}(5)\ldots P_{X\times Y}(14)$).

   – Στην περίπτωση του πολλαπλασιαστή 26x26 ψηφίων το αποτέλεσμα $P_{X\times Y}$ αποτελείται από 52 ψηφία. Τα πρώτα 4 ψηφία είναι 0. Το σχήμα 12 δείχνει το $P_{X\times Y} = 000xx.xxxxx\ldots xxxxx$.

   Αν $P_{X\times Y}(5) = 1$, η υποδιαστολή πρέπει να μετακινηθεί μία θέση αριστερά και ο εκθέτης E_P πρέπει να αυξηθεί κατά 1. Επειδή τα διαθέσιμα ψηφία της mantissa είναι



Σχήμα 12: Το γινόμενο (P) του πολλαπλασιαστή 26x26

23, εφαρμόζουμε τη μέθοδο της περικοπής για τα υπόλοιπα. Η τιμή της mantissa είναι τα ψηφία από το $P_{X\times Y}(6)$ έως το $P_{X\times Y}(28)$ ($M_P = P_{X\times Y}(6)\dots P_{X\times Y}(28)$). Αλλιώς, η τιμή του εκθέτη E_P δεν αλλάζει και η τιμή της mantissa είναι τα ψηφία από το $P_{X\times Y}(7)$ έως το $P_{X\times Y}(29)$ ($M_P = P_{X\times Y}(7)\dots P_{X\times Y}(29)$).

7. Δύο επιπλέον έλεγχοι. .

   – Αν $E = 15$ για binary16 (η $E = 127$ για binary32) (από το στάδιο 2) και ο εκθέτης E_P είχε αυξηθεί κατά 1 από το στάδιο 6, τότε το αποτέλεσμα από underflow γίνεται normal.

   – Αν $E = 45$ για binary16 (η $E = 381$ για binary32) (από το στάδιο 2) και ο εκθέτης E_P είχε αυξηθεί κατά 1 από το στάδιο 6, τότε το αποτέλεσμα από normal γίνεται overflow.

Προσεγγιστικός πολλαπλασιαστής αριθμών κινητής υποδιαστολής

Για να εφαρμόσουμε κατά προσέγγιση πολλαπλασιασμό, δύο αριθμών κινητής υποδιαστολής, θα εφαρμόσουμε την προσέγγιση στο στάδιο 6 όπου πολλαπλασιάζεται οι αριθμοί X και Υ. Η τεχνική που χρησιμοποιήθηκε για την προσέγγιση είναι αυτή που αναπτύχθηκε από τους συγγραφείς του [9] και ονομάζεται Hybrid Partial Product Perforation-Rounding.

Έστω X και Υ οι δύο αριθμοί n ψηφίων. Για το γινόμενο τους ισχύει:

$$X \times Y = \sum_{j=0}^{n/2-1} X * {y_j}^{MB} * 4^j \tag{0.17}$$

όπου, ${y_j}^{MB} \in \{0, \pm 1, \pm 2\}$.

Partial Product Perforation

Σύμφωνα με τη μέθοδο αυτή αγνοούμε τα k πρώτα συνεχόμενα μερικά γινόμενα ξεκινώντας από το λιγότερο σημαντικό. Δηλαδή τα k λιγότερο σημαντικά ψηφία του Υ κωδικοποιημένα κατά τον τροποποιημένο αλγόριθμο του Booth, δεν παράγονται. Άρα τα 2k LSBs του Υ (συμπεριλαμβανομένου του b_{-1}) απαλείφονται. Τελικά το γινόμενο των X και Υ υπολογίζεται ως εξής:

$$X \times Y|_k = \sum_{j=k}^{n/2-1} X * {y_j}^{MB} * 4^j \tag{0.18}$$

Partial Product Rounding

Σύμφωνα με τη μέθοδο αυτή τα m-1 LSBs του X απαλείφονται, και προσθέτουμε το ψηφίο x_{m-1} στο υπόλοιπο πιο σημαντικό κομμάτι X_m, όπως φαίνεται παρακάτω:

$$X_m + x_{m-1} = \langle x_{n-1}, x_{n-2}...x_m \rangle_{2,s} + x_{m-1} \tag{0.19}$$

Η περικοπή των m-1 LSBs θα οδηγούσε σε σημαντικά σφάλματα. Προκειμένου να αποφευχθεί αυτό, το τελευταίο LSB (x_{m-1}) προστίθεται στο X_m ώστε να έχουμε μικρότερο σφάλμα. Στη συνέχεια παρουσιάζονται οι δύο περιπτώσεις για τις διαφορετικές τιμές του x_{m-1}.

Αν $x_{m-1} = 0$, τότε τα κατά προσέγγιση μερικά γινόμενα $\left(\tilde{P}_j\right)$ υπολογίζονται από τη σχέση: $\tilde{P}_j = \left(X_m + 0\right) * {y_j}^{MB} = X_m * {y_j}^{MB}\big)$.

Εάν $x_{m-1} = 1$, και χρησιμοποιώντας τη σχέση $X_m + 1 = -\overline{X}_m$, τα κατά προσέγγιση μερικά γινόμενα $\left(\tilde{P}_j\right)$ υπολογίζονται από τη σχέση: $\tilde{P}_j = (X_m + 1) * {y_j}^{MB} = \left(-\overline{X}_m\right) * {y_j}^{MB} = \overline{X}_m * \left(-{y_j}^{MB}\right)$, όπου $\left(-{y_j}^{MB}\right) = (-1)^{\overline{s}_j} * (2 * two_j + one_j)$.

Χρησιμοποιώντας τη σχέση ${X_m}^* = X_m \oplus x_{m-1}$ προκειμένου να σχηματιστεί το X_m ή το $\overline{X}_m$ οι δύο περιπτώσεις μπορούν να συγχωνευτούν. Επιπλέον χρησιμοποιείται η σχέση ${s_j}^* = s_j \oplus x_{m-1}$ για να σχηματιστεί το ${y_j}^{MB}$ ή το $-{y_j}^{MB}$. Έτσι η σχέση που υπολογίζει τα μερικά γινόμενα διαμορφώνεται ως εξής: $\tilde{P}_j = {X_m}^* * {y_j}^{MB^*}$, όπου ${y_j}^{MB^*} = (-1)^{{s_j}^*} * (2 * two_j + one_j)$.

Hybrid Partial Product Perforation-Rounding

Οι δύο παραπάνω τεχνικές συνδυάζονται και έτσι προκύπτει η τεχνική Hybrid Partial Product Perforation-Rounding. Αυτή η τεχνική χαρακτηρίζεται από τις παραμέτρους k και m. Η παράμετρος k αναφέρεται στον αριθμό των μερικών γινομένων που απαλείφονται, ενώ η παράμετρος m αναφέρεται στον αριθμό των ψηφίων των μερικών γινομένων που απαλείφονται. Η εξίσωση που περιγράφει την τεχνική Hybrid Partial Product Perforation-Rounding είναι:

$$X \times Y|_{k,m} = \sum_{j=k}^{n/2-1} \tilde{P}_j * 4^j = \sum_{j=k}^{n/2-1} {X_m}^* * {y_j}^{MB^*} * 4^j \tag{0.20}$$

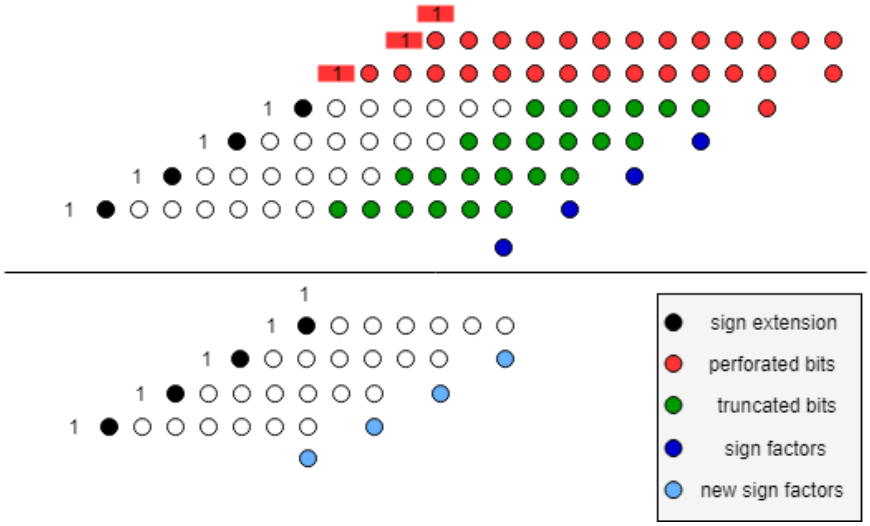
όπου k ∈ [0, n/2-1) και m ∈ [0, n-1).

Ο διορθωτικός όρος στην περίπτωση αυτή δίνεται από τη σχέση: ${c_j}^* = {s_j}^* * (one_j + two_j)$.

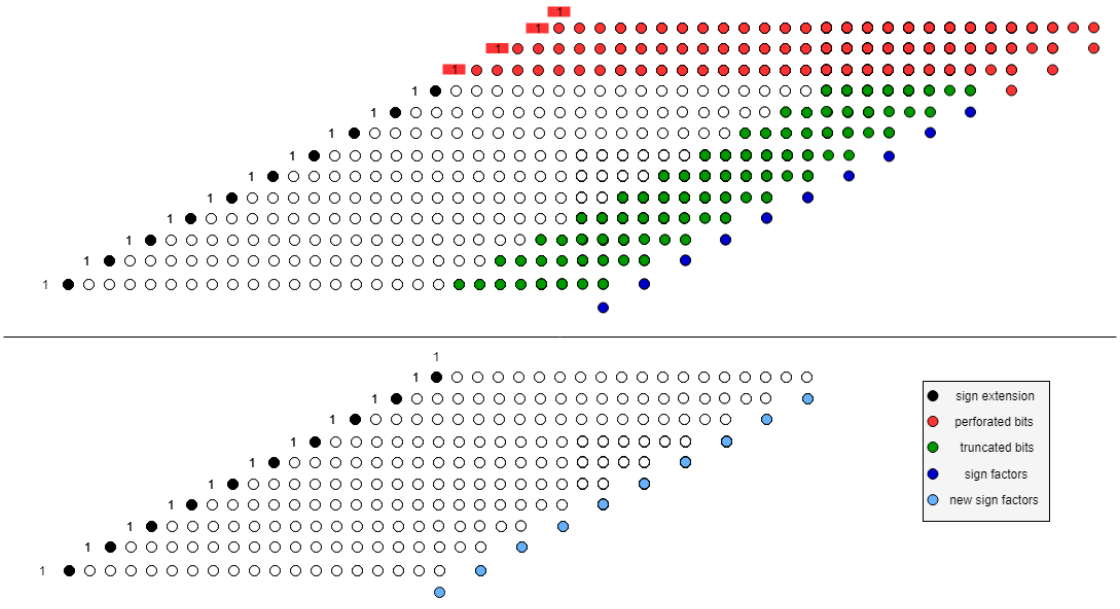
Το σχήμα 13 απεικονίζει έναν πολλαπλασιαστή 12x12 ψηφίων με k=2 και m=6.

Το σχήμα 14 απεικονίζει έναν πολλαπλασιαστή 26x26 ψηφίων με k=3 και m=8.

Ο κατά προσέγγιση σχεδιασμός έχει έναν επιπλέον έλεγχο. Η έξοδος του πολλαπλασιασμού της μαντισσα του A με αυτή του B (1.xx ... xx * 1.xx ... xx) έχει τη μορφή xx.xxxx



Σχήμα 13: Πολλαπλασιαστής 12x12 ψηφίων με k=2 και m=6



Σχήμα 14: Πολλαπλασιαστής 26x26 ψηφίων με k=3 και m=8

..... xxxx. Τα ψηφία αριστερά από την υποδιαστολή μπορεί να είναι 01 ή 10 ή 11 στον ακριβή πολλαπλασιαστή. Στον κατά προσέγγιση σχεδιασμό, αυτά τα δυο δυαδικά ψηφία μπορούν επίσης να είναι μηδενικά και τα δύο. Επομένως, η περίπτωση αυτή ελέγχεται και αν ισχύει αυτό η έξοδος χαρακτηρίζεται ως overflow.

Πειραματικά Αποτελέσματα

Όλοι οι πολλαπλασιαστές υλοποιήθηκαν σε Verilog HDL. Για τη σύνθεσή τους χρησιμοποιήθηκε το Synopsys Design Compiler και η προσομοίωσή τους έγινε με το Mentor Graphics ModelSim.

Ανάλυση σφαλμάτων

Για τη μέτρηση των σφαλμάτων χρησιμοποιούμε το μέσο σφάλμα (MRED) λόγω της ιδιότητάς του να επηρεάζεται λιγότερο από την κατανομή των εισόδων. Το (MRED) δίνεται από τη σχέση:

$$MRED = \frac{\sum RED_{AB}}{M} \tag{0.21}$$

όπου:

- Το RED ορίζεται ως η αριθμητική διαφορά μεταξύ του ακριβούς αποτελέσματος και του κατά προσέγγιση αποτελέσματος διαιρουμένου με το ακριβές αποτέλεσμα.

$$RED_{AB} = \frac{|P - \tilde{P}|}{|P|} \tag{0.22}$$

- $M = N - sum\_wrong\_inf - sum\_NaN\_red$. (N είναι ο αριθμός των εισόδων. Οι μεταβλητές sum_wrong_inf και sum_NaN_red θα αναλυθούν στη συνέχεια.)

Η πιθανότητα να έχουμε σφάλμα RED μικρότερο του 2% δίνεται από την παρακάτω σχέση:

$$PRED = \frac{pos}{N} \tag{0.23}$$

όπου, pos είναι ο συνολικός αριθμός των RED που είναι μικρότερα του 2%.
Στην υλοποίησή μας το αποτέλεσμα μπορεί να είναι:

- Κανονικός αριθμός που περιγράφεται από την εξίσωση (0.13).
- Overflow που αναπαρίσταται ως άπειρο.
- Underflow που αναπαρίσταται ως μηδέν.

Η μεταβλητή sum_wrong_inf αυξάνεται κατά 1 αν το P είναι άπειρο και το $\tilde{P}$ όχι, και αντίστροφα. Η μεταβλητή sum_NaN_red αυξάνεται κατά 1 αν το P είναι μηδέν και το $\tilde{P}$ όχι, και αντίστροφα. Έτσι προκύπτουν οι παρακάτω σχέσεις:

$$WRONG\_INF = \frac{sum\_wrong\_inf}{N} \tag{0.24}$$

$$NaN\_red = \frac{sum\_NaN\_red}{N} \tag{0.25}$$

Αξιολόγηση σε Επίπεδο Κυκλώματος

Στο κεφάλαιο 6 παρουσιάζονται αναλυτικά τα αποτελέσματα για τους πολλαπλασιαστές 16x16 και 32x32 αντίστοιχα.

- 16x16

  Όπως φαίνεται στον πίνακα 6.2 το σφάλμα κυμαίνεται από 0.05% έως 3.33%. Το κέρδος σε delay, area, energy φτάνει το 32%, 54%, 53% αντίστοιχα. Οι μεταβλητές NaN_red και WRONG_INF είναι σχεδόν μηδενικές.

- 32x32

  Όπως φαίνεται στον πίνακα 6.3 το σφάλμα κυμαίνεται από 0.00% έως 2.2%. Το κέρδος σε delay, area, energy φτάνει το 46%, 83%, 82% αντίστοιχα. Οι μεταβλητές NaN_red και WRONG_INF είναι σχεδόν μηδενικές.

Τα σχήματα 6.1 και 6.2 παρουσιάζουν την αξιολόγηση κάθε τεχνικής σε διαγράμματα Pareto για 16 και 32 bit αντίστοιχα.

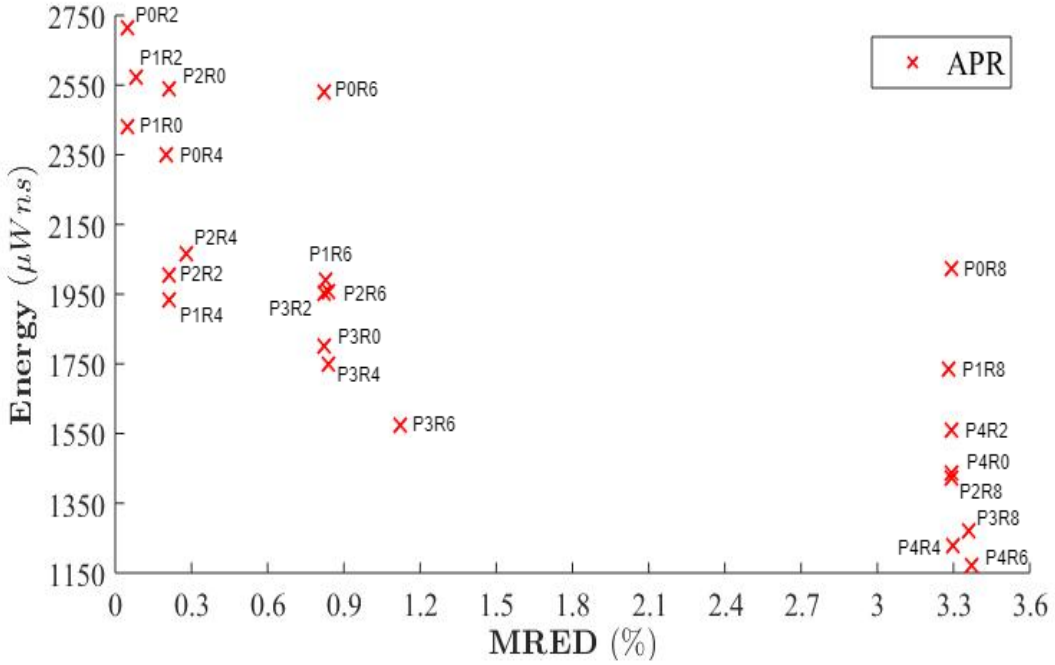
Για ένα δεδομένο σύστημα το μπροστινό μέρος του pareto δείχνει τις πιο αποτελεσματικές υλοποιήσεις. Χρησιμοποιώντας τα σχήματα 15 και 16 βρίσκουμε τις προσεγγίσεις που βρίσκονται πάνω στα σύνορα. Οι προσεγγίσεις αυτές είναι:

- $PR|_{0,2}$, $PR|_{1,0}$, $PR|_{1,4}$, $PR|_{3,4}$, $PR|_{3,6}$, και $PR|_{4,6}$, για binary16.
- $PR|_{4,12}$, $PR|_{6,12}$, $PR|_{6,14}$, $PR|_{8,16}$, $PR|_{10,18}$, και $PR|_{10,20}$, για binary32.

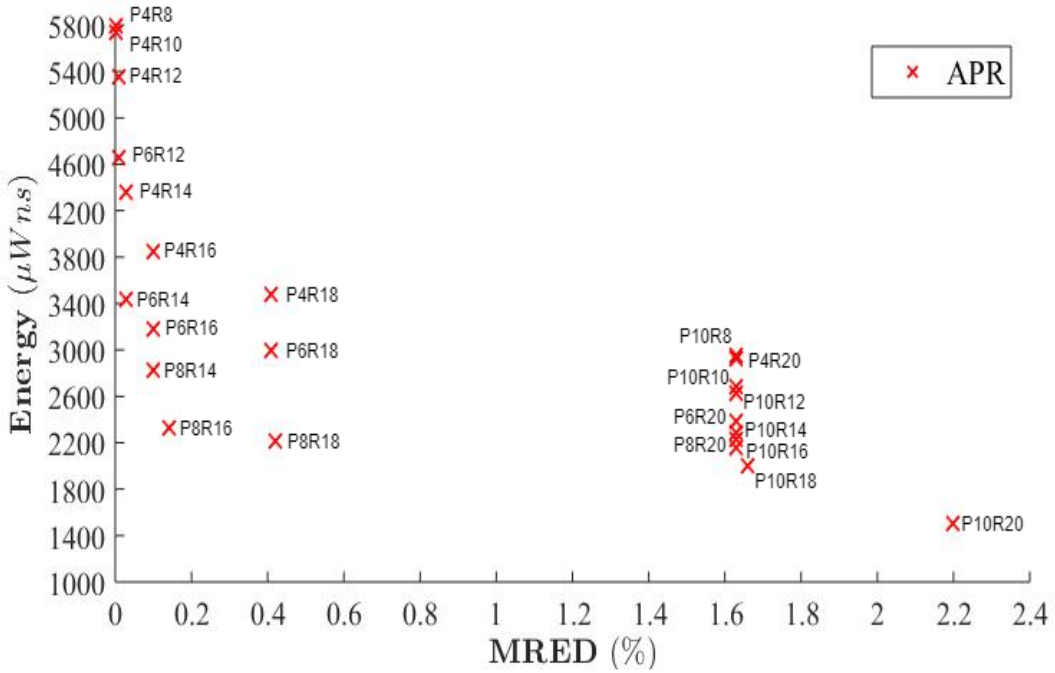
Στο σχήμα 15 παρατηρούμε ότι όλες οι υλοποιήσεις με rounding=8 καθώς και αυτές με perforation=4 εισάγουν σφάλμα γύρω στο 3.4%. Αυτή η απότομη αύξηση του σφάλματος πιθανόν συμβαίνει γιατί το rounding=8 απαλείφει από το Υ αρκετά σημαντικά ψηφία της τάξης του 2^{-4} και μικρότερης αξίας. Ομοίως το perforation=4 απαλείφει από το X τα αντίστοιχα ψηφία.

Στο σχήμα 16 παρατηρούμε ότι σχεδόν όλες οι υλοποιήσεις με rounding=20 καθώς και αυτές με perforation=10 εισάγουν σφάλμα γύρω στο 1.6%. Αυτή η απότομη αύξηση του σφάλματος πιθανόν συμβαίνει γιατί το rounding=20 απαλείφει από το Υ αρκετά σημαντικά ψηφία της τάξης του 2^{-5} και μικρότερης αξίας. Ομοίως το perforation=10 απαλείφει από το X τα αντίστοιχα ψηφία.

Το σχήμα 6.3 παρουσιάζει τον τρόπο με τον οποίο επηρεάζεται το MRED από τις παραμέτρους k και m, για πολλαπλασιαστή μεγέθους n = 16, 32 bits. Όπως φαίνεται το perforation



Σχήμα 15: Pareto Energy-Error 16bit



Σχήμα 16: Pareto Energy-Error 32bit

εισάγει μεγαλύτερη τιμή σφάλματος από το rounding. Επιπλέον καθώς αυξάνονται τα bits του αριθμού το σφάλμα επηρεάζεται λιγότερο για ίδιες τιμές k και m. Για παράδειγμα, για n=16, k=4 και m=6 το MRED είναι 3.33%. Ενώ για n=32, k=4 και m=8 το MRED είναι 0.0%.

Τα κέρδη σε area και energy για τις καλύτερες υλοποιήσεις που επιλέχθηκαν φαίνονται στο σχήμα 6.4.

Για τα 16bits το MRED κυμαίνεται μεταξύ 0.05% και 3.33%, το κέρδος σε area μεταξύ 7.29% και 54.65% και το κέρδος σε energy μεταξύ 3.57% και 53.55%.

Για τα 32bits το MRED κυμαίνεται μεταξύ 0.01% και 2.20%, το κέρδος σε area μεταξύ 46.03% και 83.45% και το κέρδος σε energy μεταξύ 37.25% και 82.43%.

Αξιολόγηση σε Επίπεδο Εφαρμογής

Η εφαρμογή φίλτρων σε εικόνες έχει ως στόχο τη βελτίωση της ποιότητας ή τον τονισμό των γνωρισμάτων τους (αφαίρεση αμυχών, ανίχνευση ακμών και τονισμός των περιγραμμάτων, εξάλειψη του φαινομένου κόκκινων ματιών από εικόνες προσώπων, θόλωση της εικόνας κτλ.)

Η αποτελεσματικότητα των προτεινόμενων κατά προσέγγιση πολλαπλασιαστών αξιολογήθηκε χρησιμοποιώντας μια εφαρμογή για θόλωμα (Gaussian blur) της εικόνας. Το Gaussian blur είναι ένας τύπος φίλτρου που χρησιμοποιεί μια Gaussian συνάρτηση για τον υπολογισμό του μετασχηματισμού που εφαρμόζεται σε κάθε pixel της εικόνας. Το θόλωμα της εικόνας, που συνήθως χρησιμοποιείται για τη μείωση του θορύβου και των λεπτομερειών της εικόνας, επιτυγχάνεται από τη συνέλιξη μεταξύ ενός πυρήνα και της εικόνας:

$$g(x, y) = w * f(x, y) \tag{0.26}$$

όπου $g(x, y)$ είναι η φιλτραρισμένη εικόνα, $f(x, y)$ είναι η πρωτότυπη εικόνα, w είναι το φίλτρο.

Η έξοδος της εικόνας προσδιορίζεται χρησιμοποιώντας την ακόλουθη εξίσωση:

$$Y(i+1, j+1) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} X(i+1+m, j+1+n) * G_{blur}(m+2, n+2) \tag{0.27}$$

G_{blur} δίνεται από τη σχέση:

$$G_{blur} = \frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{0.28}$$

Για την εφαρμογή αυτή χρησιμοποιήσαμε δύο διαφορετικές εικόνες, τη Lena και τον cameraman.

- πολλαπλασιαστής 16x16 ψηφίων

  Το σχήμα 6.5 δείχνει την πρωτότυπη εικόνα Lena (Σχήμα 6.5(a)), και τις εικόνες που έχουν προκύψει από την εφαρμογή του φίλτρου για θόλωμα χρησιμοποιώντας τον ακριβή πολλαπλασιαστή (Σχήμα 6.5(b)) και τους προτεινόμενους κατά προσέγγιση πολλαπλασιαστές (Σχήμα 6.5(c)-(h)).

- πολλαπλασιαστής 32x32 ψηφίων

  Το σχήμα 6.6 δείχνει την πρωτότυπη εικόνα cameraman (Σχήμα 6.6(a)), και τις εικόνες που έχουν προκύψει από την εφαρμογή του φίλτρου για θόλωμα χρησιμοποιώντας τον ακριβή πολλαπλασιαστή (Σχήμα 6.6(b)) και τους προτεινόμενους κατά προσέγγιση πολλαπλασιαστές (Σχήμα 6.6(c)-(h)).

Το αποτέλεσμα της διαδικασίας θολώματος δεν είναι εύκολο να αξιολογηθεί οπτικά, για αυτό χρησιμοποιούμε δύο δείκτες: το δείκτη PSNR (Peak Signal-to-Noise Ratio) καθώς και το δείκτη SSIM (Structural Similarity Index). Οι τιμές των δεικτών αυτών φαίνονται στους πίνακες 6.4-6.7. Το PSNR δίνεται από τη σχέση (0.29).

$$PSNR = 10 * log_{10}\left(\frac{{MAX_I}^2}{MSE}\right) \tag{0.29}$$

Όπου το MAX_I δείχνει τη μέγιστη δυνατή τιμή pixel της εικόνας (στην περίπτωσή μας είναι το 255). Το μέσω τετραγωνικό σφάλμα MSE δίνεται από τη σχέση:

$$MSE = \frac{1}{m * n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I(i,j) - K(i,j)) \tag{0.30}$$

Το SSIM χρησιμοποιείται για τη μέτρηση της ομοιότητας μεταξύ δύο εικόνων. Οι τιμές του SSIM που είναι 1 δείχνουν την καλύτερη απόδοση, δηλαδή, όμοια με αυτή του ακριβούς πολλαπλασιαστή. Οι υπόλοιπες τιμές που είναι μεγαλύτερες από το 0.80 αντιστοιχούν σε επίσης καλές προσεγγίσεις.

Chapter 1

Introduction-Motivation

1.1 Floating-point arithmetic

Floating-point (FP) arithmetic is widely used in many areas, especially scientific computation and signal processing. Over the years, a variety of approximating real numbers have been introduced. One of them, the floating-point arithmetic, is clearly the most efficient way of representing real numbers in computers. Floating-point numbers have two advantages over integers. First, they can represent values between integers. Second, because of the scaling factor, they can represent a much greater range of values.

This FP arithmetic uses formulaic representation of real numbers as an approximation so as to support a trade-off between range and precision. For this reason, it is often found in systems which include very small and very large numbers, because, they require fast processing times.

The term floating-point refers to the fact that a number's radix point (decimal or binary point) can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent component, and thus the floating-point representation can be thought as a kind of scientific notation.

Since the first computer appeared, a variety of floating-point representations have been used. In 1985, the IEEE 754 [1] Standard for Floating-Point Arithmetic was established, and since the 1990s, the most commonly encountered representations are those defined by the IEEE.

The speed of floating-point operations measured in terms of FLOPS. This measurement is an important characteristic of a computer system, especially for applications that involve intensive mathematical calculations.

A floating-point unit is a part of a computer system specially designed to carry out operations on floating-point numbers.

Chapter 2

Theoretical Background

2.1 Introduction

In this chapter we analyze all the algorithms and techniques were used in this diploma thesis. Specifically the binary numbers, the floating point numbers, the encoding in a binary interchange format of a floating-point number, multiplication of two floating-point numbers etc will be explained.

2.2 Binary numeral system

The binary system is the basic numeral system in digital electronics. Each binary number expressed in the base-2 numeral system, which uses exclusively the digits "0" (zero) and "1" (one). Specifically each digit is referred as bit (a_i).

Every number is represented in the binary system according to the equation (2.1)

$$a = \sum_{i=0}^{n-1} 2^i * a_i \tag{2.1}$$

where,

a : is the value of the number,

b : is the number of bits of the binary number,

a_i : is the value of each bit.

For example, the decimal number 17 is represented in the binary system as 10001. Using the equation (2.1) the binary number 10001 corresponds to:

$1 * 2^0 + 0 * 2^1 + 0 * 2^2 + 0 * 2^3 + 1 * 2^4 = 17_{10}$.

The maximum value of a number which consists of n bits is $2^n - 1$ and the minimum is 0. The left-most bit in a string called Most Significant Bit (MSB) and the right-most

bit called Least Significant Bit (LSB). In the above example the bit multiplied with 2^4 is the MSB and the bit multiplied with 2^0 is the LSB. In all cases the LSB multiplied with 2^0.

2.2.1 Binary arithmetic

Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction, multiplication, and division can be performed on binary numerals. The addition of two positive binary numbers is similar to the addition of decimal numbers. In this diploma thesis we will analyze the multiplication. The equation (2.1) can represent only positive numbers. Two's complement is a mathematical operation on binary numbers, and it is an example of a radix complement. It is used in computing as a method of signed number representation.

2.2.1.1 Two's complement

A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two.

The value a of a n-bit number $(a_{n-1}a_{n-2}...a_2a_1a_0)$ is given by modifying equation (2.1) as follows:

$$a = -a_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} 2^i * a_i \tag{2.2}$$

The most significant bit determines the sign of the number and it is sometimes called the sign bit. If the MSB is "1" the number is negative. The range of numbers representation is not the same as the unsigned binary numbers. Using n bits, all integers from -(2^{n-1}) to $(2^{n-1}-1)$ can be represented. For example, a 4-bit unsigned number can represent the values 0 (0000) to 15 (1111). However a two's complement 4-bit number can only represent positive integers from 0 to 7 (0111), because the rest of the bit combinations with the most significant bit as "1" represent the negative integers from -1 (1111) to -8 (1000).

To get the two's complement of a binary number, the bits are inverted by using the bitwise NOT operation (one's complement). Then the value 1 is added to the one's complement, ignoring the overflow which occurs when talking the two's complement of 0. For example, the decimal number 7 is represented by $A = 0111_{(2)}$ (using 4 bits). The MSB is 0, so it represent a non-negative value. To get the one's complement required to invert the bits of the number, $\mathbb{A} \rightarrow 1000_{(2)}$. Then add 1 to the number 1000 and have the binary number 1001. Using the equation (2.2) the binary number 1001 corresponds to: $-1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = -7_{(10)}$

2.3 Booth's Algorithms

This is a kind of algorithm which uses a more straightforward approach. This algorithm also has the benefit of the speeding up the multiplication process [10] and it is very efficient too.

2.3.1 Booth's Multiplication Algorithm

As we have explained a number Y can be written according to the equation (2.2) as:

$$Y = -y_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} 2^i * y_i \tag{2.3}$$

Moreover using a mathematical trick Y can be written as: Y=2Y-Y=Z

$$\begin{array}{lcccccccc} 2Y & = & -y_{n-1} & y_{n-2} & y_{n-3} & . & . & y_0 & 0 \\ -Y & = & 0 & y_{n-1} & -y_{n-2} & . & . & -y_1 & -y_0 \\ Z & = & & z_{n-1} & z_{n-2} & . & . & z_1 & z_0 \end{array}$$

where: $z_0 = 0 - y_0$,

$z_1 = y_0 - y_1$,

$z_2 = y_1 - y_2$,

.

.

.

$z_{n-2} = y_{n-3} - y_{n-2}$

The digit z_{n-1} is computed as follows: $z_{n-1} = -2y_{n-1} + y_{n-2} + y_{n-1} = y_{n-2} - y_{n-1}$

Thus Y can be computed as:

$$Y = \sum_{i=0}^{n-1} 2^i * z_i \tag{2.4}$$

The product P of two n-bit numbers X, Y is : P=X*Y (2n-bits). According the equation (2.2) the two's complement X, Y and P can be expressed as:

$$X = -x_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} 2^i * x_i \tag{2.5}$$

$$Y = -y_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} 2^i * y_i \tag{2.6}$$

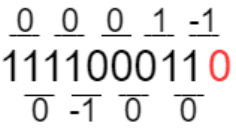


Figure 2.1: 2 bit pairing as per Booth encoding

Table 2.1: Radix-2 encoding

y_i	y_{i-1}	Encoded digits $(y_{i-1} - y_i)$
0	0	0
0	1	1
1	0	-1
1	1	0

$$P = -p_{2n-1} * 2^{2n-1} + \sum_{i=0}^{2n-2} 2^i * p_i \tag{2.7}$$

According to Booth's multiplication algorithm and using the equation (2.4) the product P is formed:

$$P = X * Y = \sum_{i=0}^{n-1} (y_{i-1} - y_i) * X * 2^i \tag{2.8}$$

Booth's algorithm compare 2 bits at a time with overlapping technique. Grouping starts from the LSB, and the first block only uses one bit of the multiplier and assumes a zero for the second bit as shown in Figure 2.1.

According the equation (2.8) for each bit y_i, for i running from 0 to n-1, the bits y_i and y_{i-1} are considered. Where these two bits are equal, the partial product (PP) is left unchanged. Where $y_i = 0$ and $y_{i-1} = 1$, X is added to PP, and where $y_i = 1$ and $y_{i-1} = 0$, X is subtracted from PP . In all cases the partial product shift left 1 bit. The final value of PP is the signed product.

In the table 2.1 the radix-2 encoding is displayed. As it shown in Table (2.1) each encoded digit can be 0 or 1 or -1.

For example we have to multiply two signed numbers : X=10110 and Y=00101 (Figure 2.2). According to radix-2 encoding, Y is $01\bar{1}1\bar{1}$. The arithmetic shift to the left is used to avoid overflow and it is equivalent to sign extension. The digits in the boxes represent

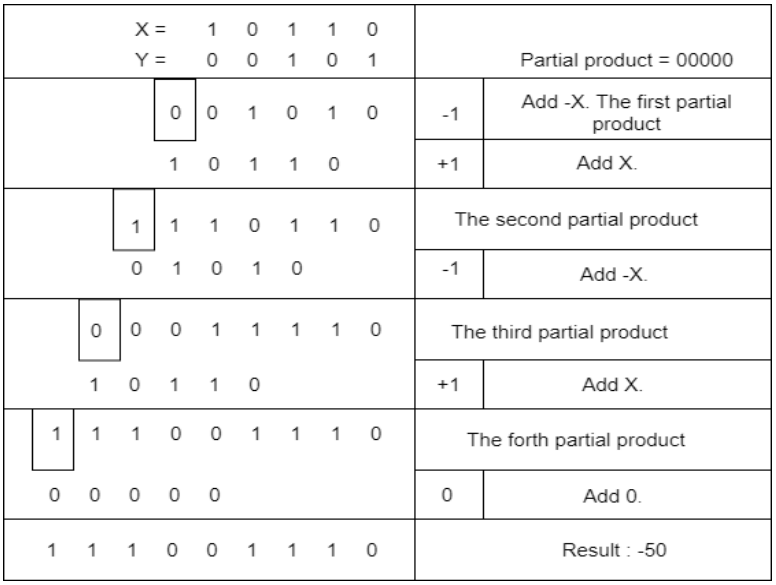


Figure 2.2: Multiplication of two numbers with Booth Algorithm

this extension.

The basic advantage of Booth's multiplication algorithm is the fact that handles both positive and negative numbers uniformly. Also achieves efficiency in the number of additions required when the multiplier has a few large blocks of 1's. One of the solutions of realizing high speed multipliers is to enhance parallelism which helps to decrease the number of subsequent calculation stages. The Booth algorithm (Radix-2) has two disadvantages.

1) The number of add / subtract operations and the number of shift operations becomes variable and becomes inconvenient in designing parallel multipliers.

2) The algorithm becomes inefficient when there are isolated 1's. These problems are overcome by using Modified Booth Algorithm (Radix-4).

2.3.2 Modified Booth's Multiplication Algorithm

The basic idea of modified booth algorithm is that, instead of shifting and adding for every column of the multiplier term and multiplying by 0 or 1, we only take every second column and multiplying by $+1$, -1, $+2$, -2 or 0, to obtain the same result. Thus, half of the partial products can be reduced using this algorithm. We modified the equation (2.4) as follows:

$$Y = \sum_{i=0}^{n-1} 2^i * z_i = \sum_{j=0}^{n/2-1} \left(2^{2j} * z_{2j} + 2^{2j+1} * z_{2j+1}\right) = \sum_{j=0}^{n/2-1} (z_{2j} + 2 * z_{2j+1}) * 2^{2j} = \sum_{j=0}^{n/2-1} {y_j}^{MB} * 4^j \tag{2.9}$$

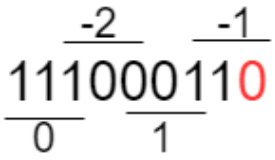


Figure 2.3: 3 bit pairing as per Modified Booth encoding

Table 2.2: Booth and Modified Booth encoding

			Booth encoding		Modified Booth encoding
y_{i+1}	y_i	y_{i-1}	z_{i+1}	z_i	${y_j}^{MB} = z_i + 2 * z_{i+1} = y_i - 2 * y_{i+1} + y_{i-1}$
0	0	0	0	0	0
0	0	1	0	+1	+1
0	1	0	+1	-1	+1
0	1	1	+1	0	+2
1	0	0	-1	0	-2
1	0	1	-1	+1	-1
1	1	0	0	-1	-1
1	1	1	0	0	0

where,

$$ {y_j}^{MB} = z_{2j} + 2 * z_{2j+1} = y_{2j-1} - y_{2j} + 2 * (y_{2j} - y_{2j+1}) = -2 * y_{2j+1} + y_{2j} + y_{2j-1} \quad (2.10) $$

Modified booth algorithm compare 3 bits at a time with overlapping technique. Grouping starts from the LSB, and the first block only uses two bits of the multiplier and assumes a zero for the third bit as shown in Figure 2.3.

In table 2.2 the booth and the modified booth encoding is displayed. The bit y_{-1} is 0.

Radix-4 Booth algorithm is given below:

- Extend the sign bit by one position if necessary to ensure that the number is even.
- Append a 0 to the right of the LSB of the multiplier.
- According to the value of each vector, each Partial Product will be 0, +y, –y, +2y or –2y.

For example we have to multiply two signed numbers : A= 10110101(-75) and B=01110010 (114) (Figure 2.4). According to the radix-4 encoding, B is $2\bar{1}1\bar{2}$. In the case of the modified booth algorithm, an arithmetic shift to the left by two places is required. The

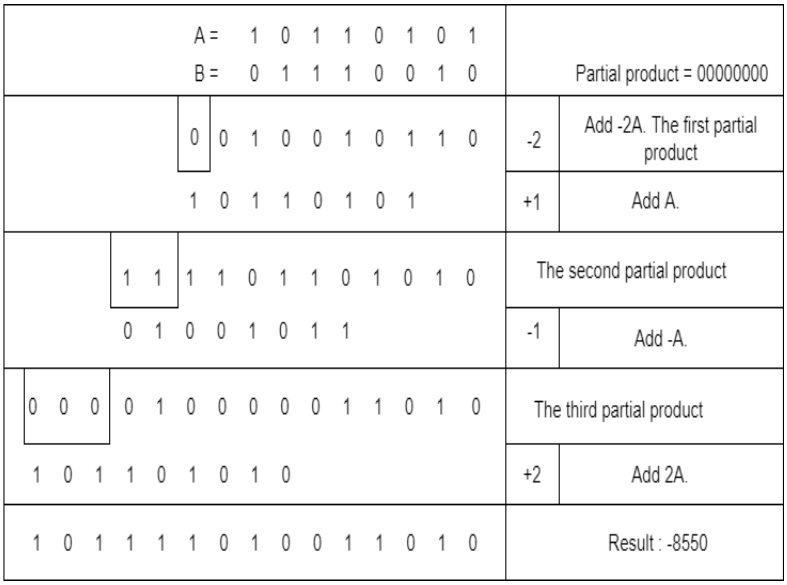


Figure 2.4: Multiplication of two numbers with Modified Booth Algorithm

arithmetic shift to the left is used to avoid overflow and it is equivalent to sign extension. The digits in the boxes represent this extension.

The advantage of this method is the halving of the number of partial products. Thus, the propagation delay to run circuit, the complexity of the circuit, and power consumption can be reduced. However, the computation of -2A and +2A increases the complexity of the circuit. In most applications the Modified Booth Algorithm is used for the exact multiplication of two numbers.

2.4 Binary multiplication

Similar to the multiplication of decimal numbers, binary multiplication follows the same process for producing a product result of two binary numbers. The binary multiplication is much easier as it contains only 0 and 1. The four fundamental rules for binary multiplication are:

$0 \times 0 = 0$

$0 \times 1 = 0$

$1 \times 0 = 0$

$1 \times 1 = 1$

Let a m-bit unsigned number A and a n-bit unsigned number B. The multiplication of these two is given in figure 2.5.

For the unsigned numbers the result (P) is expressed as follows:

$$P = A * B = \sum_{i=0}^{m-1} 2^i * a_i * \sum_{j=0}^{n-1} 2^j * b_j = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (a_i * b_j) * 2^{i+j} = \sum_{k=0}^{m+n-1} p_k * 2^k \quad (2.11)$$

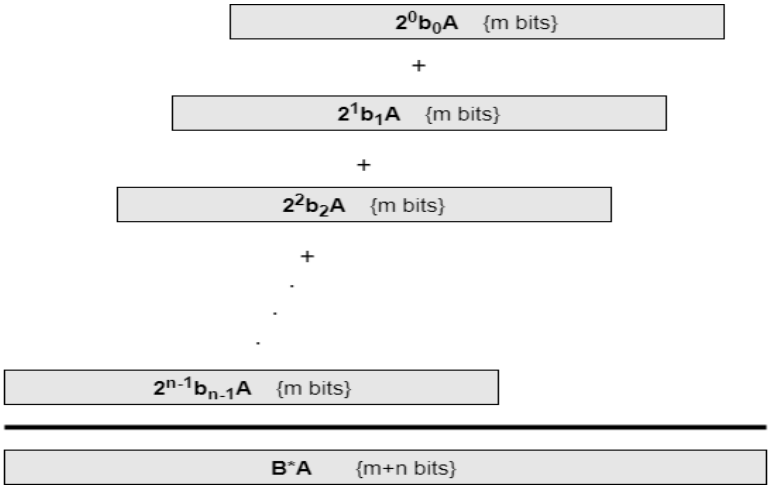


Figure 2.5: Multiplication of two unsigned numbers

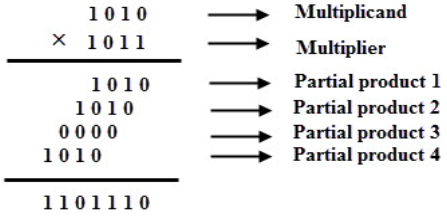


Figure 2.6: Multiplication of numbers 1010 and 1011

The figure 2.6 describes a multiplication of two unsigned numbers 1010 and 1011. From this multiplication, partial products are generated for each digit of the multiplier. Then all these partial products are added to produce the final product value. In the partial product multiplication, when the multiplier bit is zero, the partial product is zero, and when the multiplier bit is 1, the resulted partial product is the multiplicand.

2.4.1 Wallace tree multiplication

The modified booth multiplication algorithm reduces the number of partial products to 50%. However for large multipliers, 32-bit and over, the number of partial products is over 16-bit. In this case, the performance of modified booth algorithm is limited. Wallace tree multiplier can achieve a better result than the previous multiplier. This algorithm uses full adders, because, they add three bits and give as output two bits. Figure 2.7 shows the partial products, which are formed, when two 5-bit numbers (A, B) are multiplied. Figure 2.8 describe a 5x5 Wallace tree algorithm. This implementation uses 12 full adders for the first 4 stages. Moreover, 3 full adders and 5 half adders are used from the Carry Propagation Adder in the last stage.

According this algorithm the rows of partial products are divided into groups of three.

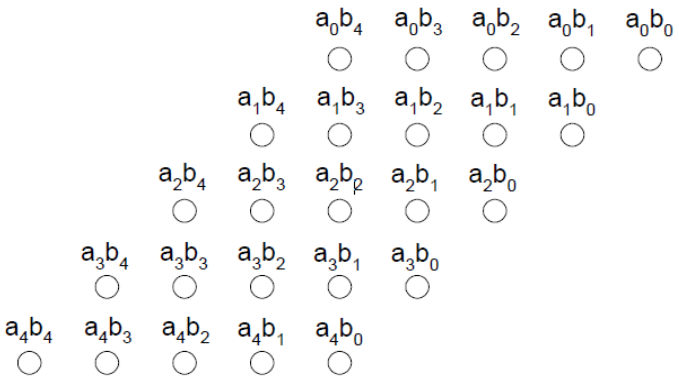


Figure 2.7: Partial products of two 5-bit numbers

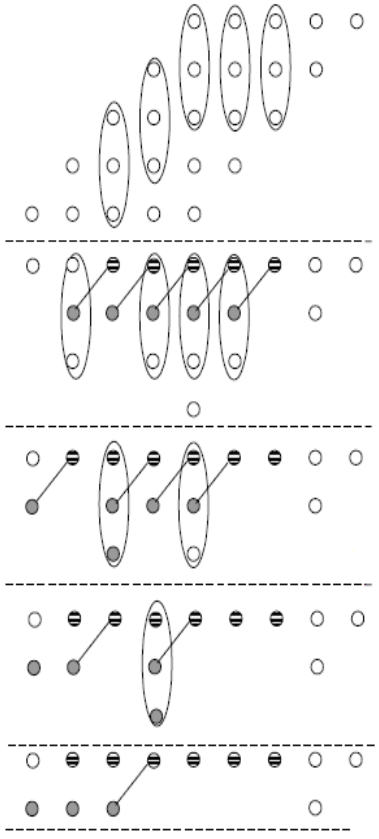


Figure 2.8: Wallace tree algorithm

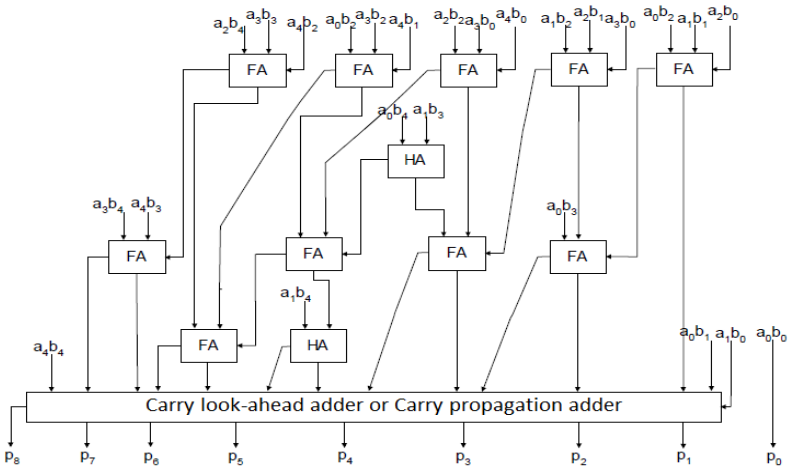


Figure 2.9: A 5x5 Wallace tree multiplier

Table 2.3: Modified Booth Encoding Table

Binary bits			Modified Booth's Digits	Encoded digit y_i		
y_{2j+1}	y_{2j}	y_{2j-1}		$sign = s_j$	$x1 = one_j$	$x2 = two_j$
0	0	0	0	0	0	0
0	0	1	+1	0	1	0
0	1	0	+1	0	1	0
0	1	1	+2	0	0	1
1	0	0	-2	1	0	1
1	0	1	-1	1	1	0
1	1	0	-1	1	1	0
1	1	1	0	1or0	0	0

Each set of three rows give a set of two rows, which consists of the sum bit and the carry bit. The bits, which cannot form of a group of three, are left alone. The steps in figure 2.8 can be reduced if both full and half adders are used. The bits, which cannot form of a group of three, form of a group of two. The rest of the algorithm is the same as the previous algorithm. Figure 2.9 shows a 5x5 Wallace tree multiplier with the addition of half adders.

2.4.2 Modified Booth Algorithm using Wallace tree

Let X and Y two unsigned n-bit numbers in two's complement. Table 2.3 shows how modified booth's encoding digits of Y are implemented into circuits.

The equations which describing the table 2.3 are:

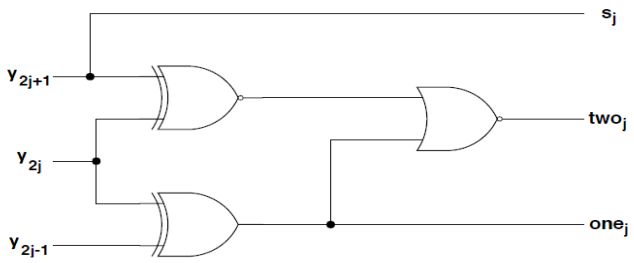


Figure 2.10: Modified Booth Encoding

$$one_j = y_{2j-1} \oplus y_{2j} \tag{2.12}$$

$$two_j = (y_{2j+1} \oplus y_{2j}) * \overline{one}_j \tag{2.13}$$

$$s_j = y_{2j+1} \tag{2.14}$$

The circuit that implements these logic equations is shown in figure 2.10.

The equation (2.15) describes the product P by using a correction term (ct) and the partial products (PP_j):

$$P = X * Y = \sum_{j=0}^{n/2-1} X * {y_j}^{MB} * 2^{2j} = ct + \sum_{j=0}^{n/2-1} PP_j * 2^{2j} \tag{2.15}$$

The figure 2.11 shows how the ct of a 12-bit multiplier is formed.

As shown in figure 2.11, the grey circles are the extra bit. This extra bit is used when X is multiplied by 2 ($two_j = 1$). The MSB of each partial product has a negative weight, so the grey circles are replaced with the black circles ($\overline{p}$) and a factor -1 is added ($-p = \overline{p} - 1$). Subsequently, the -1 is replaced by -2+1 (-1=-2+1). In the fourth step we have to replace 2-1 by 1. Lastly, the blue circles that present the correction bits (n_j) are added. If the modified booth's digit has a negative weight (${y_j}^{MB} = -1, -2, i.e. s_j = 1$), the correction bit is one in order to get the 2's complement from the 1's complement. In case of positive weight, the correction bit is zero.

The correction term is computed through the equation:

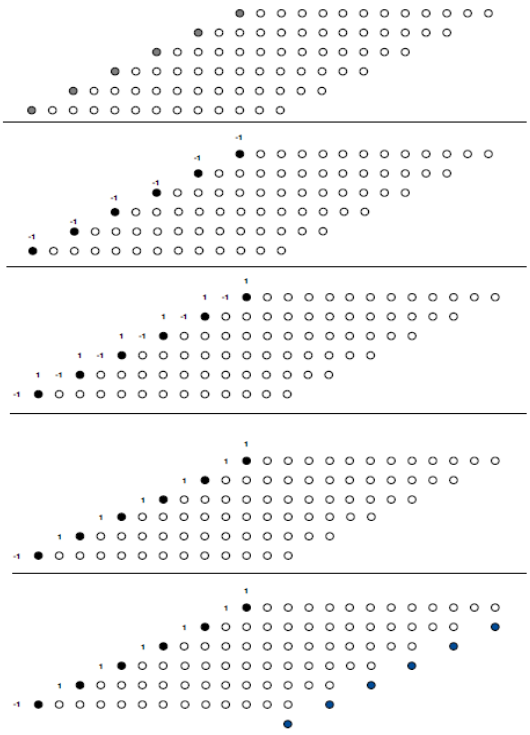


Figure 2.11: Partial Products and Correction Term

$$ct = \sum_{j=0}^{n/2-1} \left(n_j * 2^{2j}\right) + 2^n * \left(1 + \sum_{j=0}^{n/2-2} \left(2^{2j+1}\right) - 2^{n-1}\right) \tag{2.16}$$

Using the equation (2.16), the equation (2.15) is computed as follows:

$$P = \sum_{j=0}^{n/2-1} \left(n_j * 2^{2j}\right) + 2^n * \left(1 + \sum_{j=0}^{n/2-2} \left(2^{2j+1}\right) - 2^{n-1}\right) + \sum_{j=0}^{n/2-1} PP_j * 2^{2j} \tag{2.17}$$

Where:

$$PP_j = \overline{p}_{j,n} * 2^{n+2j} + \sum_{i=0}^{n-1} \left(p_{j,i} * 2^{i+2j}\right) \tag{2.18}$$

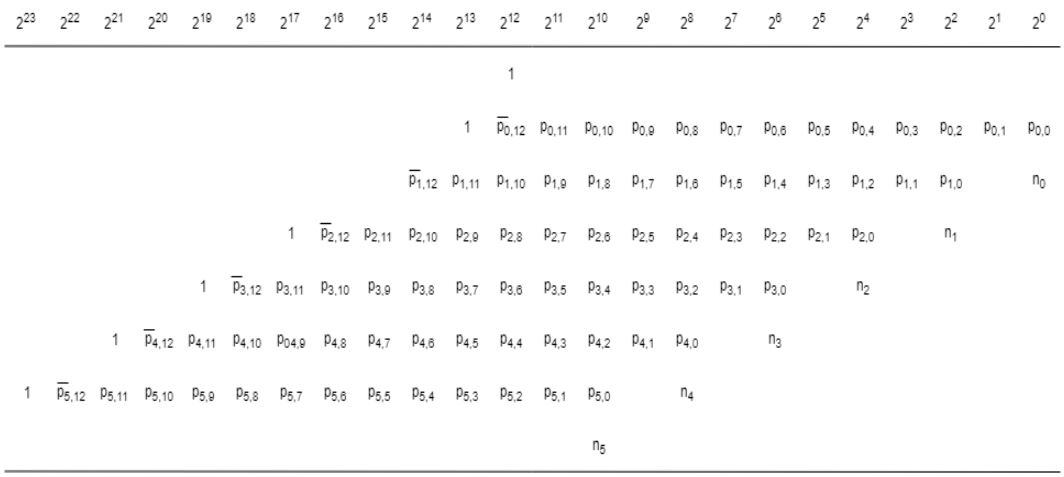


Figure 2.12: Partial products matrix with the correction term of a 12x12 multiplier

$$p_{j,i} = ((x_i \oplus s_j) * one_j) + ((x_{i-1} \oplus s_j) * two_j) \tag{2.19}$$

$$p_{j,0} = ((x_0 \oplus s_j) * one_j) + (s_j * two_j) \tag{2.20}$$

$$\overline{p}_{j,n} =! (((x_{n-1} \oplus s_j) * one_j) + ((x_{n-1} \oplus s_j) * two_j)) \tag{2.21}$$

$$n_j = s_j * (one_j + two_j) \tag{2.22}$$

Using all the above, Figure 2.12 shows a partial product matrix with the correction term of a 12x12 multiplier.

The multiplier, which consists of the partial products generators, the MB encoder, the CSA Wallace tree and the fast carry look-ahead adder (CLA) adder [11], is shown in figure 2.13.

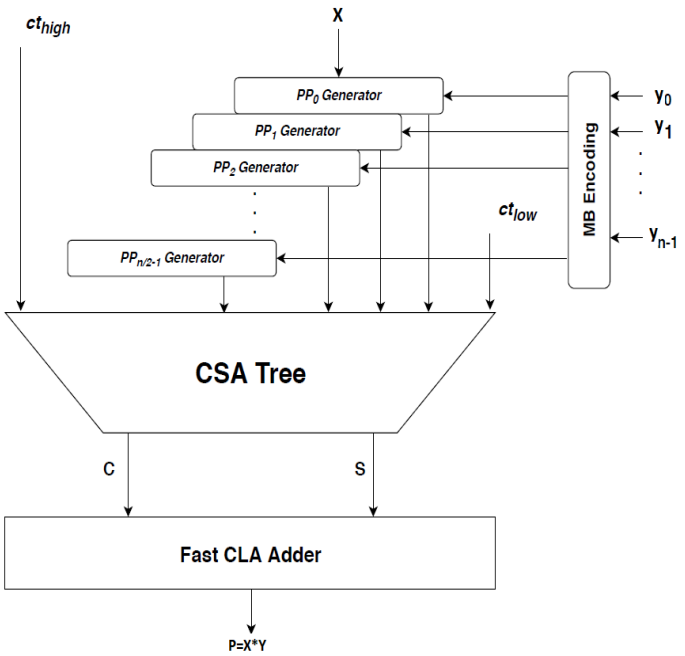


Figure 2.13: Modified Booth Multiplier

2.5 Floating-point numbers

A number representation specifies some way of encoding a number, usually as a string of digits. There are several mechanisms by which strings of digits can represent numbers.

Floating point numbers are one possible way of representing real numbers. The IEEE 754 [1] standard presents two different floating point formats (Figure 2.14):

- Binary format
- Decimal format

All the floating point numbers are composed by three components:

- Sign: it indicates the sign of the number (0 positive and 1 negative).
- Mantissa: it sets the value of the number.
- Exponent: it contains the value of the base power (biased).

The base (or radix) is implied and it is common to all the numbers (2 for binary numbers and 10 for decimal numbers)

2.5.1 The Standard IEEE 754

The standard IEEE 754 provides for many closely related formats, differing in only a few details. Five of these formats are called basic formats and the others are recommended for extending these basic formats.

Basic formats:

Figure 2.14: A decimal and a binary floating point representation

- Three binary formats, with encoding in lengths of 32, 64 and 128.
- Two decimal formats, with encoding in lengths of 64 and 128.

Not basic formats:

- Two binary formats, with encoding in lengths of 16 and 256.
- A decimal format, with encoding in length of 32.

A floating-point datum, which can be a signed zero, finite non-zero number, signed infinity, or a NaN (not-a-number) [12], can be mapped to one or more representations of floating-point data in a format.

The representations of floating-point data in a format consist of:

- triples (sign, exponent, significand): in radix b, the floating-point number, which represented by a triple, is $(-1)^{sign}$ x $b^{exponent}$ x significand
- +infinite,-infinite
- NaN

The set of finite floating-point numbers representable within a particular format is determined by the following integer parameters:

- b = the radix, 2 or 10
- p = the number of digits in the significand (precision)
- E_{max} = the maximum exponent E
- E_{min} = the minimum exponent E

The different formats are summarized in Table 2.4.

In this diploma thesis we will analyze the binary interchange format encoding and specifically the binary16, the binary32 and the binary64.

Table 2.4: The floating-point formats

Name	Common name	Base (the radix d)	Number of digits in the significand (p)	E_{min}	E_{max}	Notes
binary16	Half precision	2	11	−14	+15	Not basic
binary32	Single precision	2	24	−126	+127	
binary64	Double precision	2	53	−1022	+1023	
binary128	Quadruple precision	2	113	−16382	+16383	
binary256	Octuple precision	2	237	−262142	+262143	Not basic
decimal32		10	7	−95	+96	Not basic
decimal64		10	16	−383	+384	
decimal128		10	34	−6143	+6144	

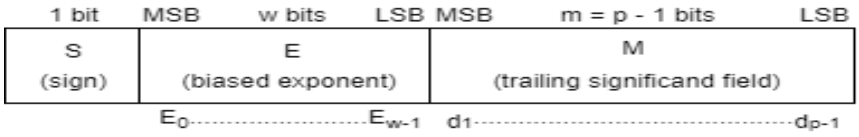


Figure 2.15: Floating-point format [1]

2.5.2 Binary Interchange Format Encoding

Each floating-point number has only one encoding in a binary interchange format. Representations of floating point data in the binary interchange formats are encoded, using k bits, in three fields as shown in figure 2.15.

The binary number in figure 2.15 consists of:

a) 1-bit sign S.

b) w-bit biased exponent E = e + bias.

c) $(m = p - 1)$-bit trailing significant field digit string $M = d_1 d_2 ... d_{p-1}$; the leading bit of the significand, d_0, is implicitly encoded in the biased exponent E.

The range of the encoding's biased exponent (E) shall include:

- every integer between 1 and $2^w - 2$, to encode normal numbers.

- the reserved value 0 to encode ± 0 and subnormal numbers.

- the reserved value 2^w-1 to encode $\pm infinite$ and NaNs.

The value v of the floating-point datum representation, is inferred from the constituent fields as follows:

a) If E = $2^w - 1$ and $M \neq 0$, then v is NaN regardless of S.

b) If E = $2^w - 1$ and M = 0 , then v = $-1^S \times (+\infty)$.

c) If $1 \leq E \leq 2^w - 2$, then the value of the corresponding floating-point number is v = $(-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times \text{M})$; thus normal numbers have an implicit leading significant bit of 1.

d) If E = 0 and $M \neq 0$, then the value of the corresponding floating-point number is v = $(-1)^S \times 2^{E_{min}} \times (0 + 2^{1-p} \times \text{M})$; thus subnormal numbers have an implicit leading significant bit of 0.

e) If E = 0 and M = 0 , then v = $(-1)^S \times (+0)$ (signed zero).

2.5.2.1 Half precision binary floating-point format

Figure 2.16 shows the IEEE 754 half precision binary format representation. It consists of a one bit sign (S), a five bit exponent (E), and a ten bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand. If the exponent is greater than 0 and smaller than 31, and there is 1 in the MSB of the significand then the number is said to be a normalized number. In this case the real number is represented by equation (2.23).

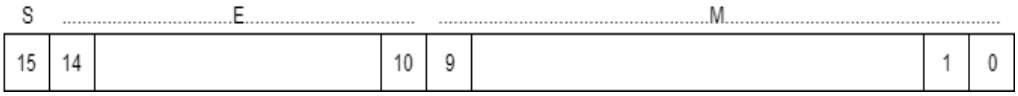


Figure 2.16: IEEE half precision floating-point respesentation

Table 2.5: The representations of floating-point data in half precision format

Exponent	Significant zero	Significant non-zero	Equation
00000_2	zero,-0	subnormal numbers	$(-1)^S * 2^{-14} * 0.M$
$00001_2, ..., 11110_2$	normalized value		$(-1)^S * 2^{(E-15)} * 1.M$
11111_2	+infinity,-infinity	Nan (quiet, signalling)	

$$Z = (-1)^S * 2^{(E-Bias)} * (1.M) \tag{2.23}$$

Where: $M = m_9 * 2^{-1} + m_8 * 2^{-2} + m_7 * 2^{-3} + ... + m_1 * 2^{-9} + m_0 * 2^{-10}$,
and Bias = 15.

The half-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 15; also known as exponent bias in the IEEE 754 standard.

$E_{min} = 00001_2 - 01111_2 = -14$,
$E_{max} = 11110_2 - 01111_2 = 15$,
Exponent bias = $01111_2 = 15$

Thus, in order to get the true exponent as defined by the offset-binary representation, the offset of 15 has to be subtracted from the stored exponent E. The stored exponents 00000_2 and 11111_2 are interpreted specially as shown in Table 2.5.

The minimum positive normal value is $2^{-14} \approx 6.10 * 10^{-5}$ and the minimum positive subnormal value is $2^{-24} \approx 5.96 * 10^{-8}$.

2.5.2.2 Single precision binary floating-point format

Figure 2.17 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number. In this case the real number is represented by equation (2.23).

Where $M = m_{22} * 2^{-1} + m_{21} * 2^{-2} + m_{20} * 2^{-3} + ... + m_1 * 2^{-22} + m_0 * 2^{-23}$,

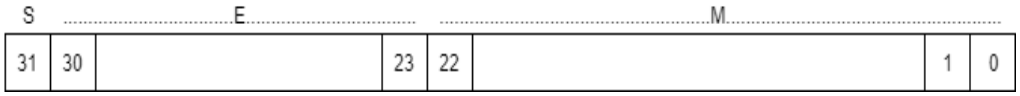


Figure 2.17: IEEE single precision floating-point representation

Table 2.6: The representations of floating-point data in single precision format

Exponent	Significant zero	Significant non-zero	Equation
00_H	zero,-0	subnormal numbers	$(-1)^S * 2^{-126} * 0.M$
$01_H, ..., FE_H$	normalized value		$(-1)^S * 2^{(E-127)} * 1.M$
FF_H	+infinity,-infinity	Nan (quiet, signalling)	

and Bias = 127.

The single-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 127; also known as exponent bias in the IEEE 754 standard.

$E_{min} = 01_H - 7F_H = -126$,

$E_{max} = FE_H - 7F_H = 127$,

Exponent bias = 7FH = 127

Thus, in order to get the true exponent as defined by the offset-binary representation, the offset of 127 has to be subtracted from the stored exponent E. The stored exponents 00_H and FF_H are interpreted specially as shown in Table 2.6.

The minimum positive normal value is $2^{-126} \approx 1.18 * 10^{-38}$ and the minimum positive subnormal value is $2^{-149} \approx 1.4 * 10^{-45}$.

2.5.2.3 Double precision binary floating-point format

Figure 2.18 shows the IEEE 754 double precision binary format representation; it consists of a one bit sign (S), an eleven bit exponent (E), and a fifth two bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand. If the exponent is greater than 0 and smaller than 2047, and there is 1 in the MSB of the significand then the number is said to be a normalized number. In this case the real number is represented by equation (2.23).

Where $M = m_{51} * 2^{-1} + m_{50} * 2^{-2} + m_{49} * 2^{-3} + ... + m_1 * 2^{-51} + m_0 * 2^{-52}$,

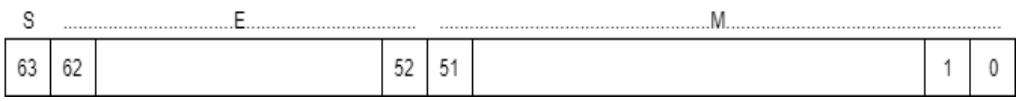


Figure 2.18: IEEE double precision floating-point representation

Table 2.7: The representations of floating-point data in double precision format

Exponent	Significant zero	Significant non-zero	Equation
00000000000_2	zero,-0	subnormal numbers	$(-1)^S * 2^{-1022} * 0.M$
00000000001_2,..., 11111111110_2	normalized value		$(-1)^S * 2^{(E-1023)} * 1.M$
11111111111_2	+infinity,-infinity	Nan (quiet, signalling)	

and Bias = 1023.

The double-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 1023; also known as exponent bias in the IEEE 754 standard.

$E_{min} = 00000000001_2 - 01111111111_2 = -1022$,

$E_{max} = 11111111110_2 - 01111111111_2 = 1023$,

Exponent bias = $01111111111_2 = 1023$

Thus, in order to get the true exponent as defined by the offset-binary representation, the offset of 1023 has to be subtracted from the stored exponent E. The stored exponents 00000000000_2 and 01111111111_2 are interpreted specially as shown in Table 2.7.

The minimum positive normal value is 2^{-1022} and the minimum positive subnormal value is 2^{-1074}.

Chapter 3

Related Work in the Field of Floating-Point Operations

3.1 Floating-point adder

Floating-point addition is the most common floating-point operation and accounts for almost half of the scientific operation. As a consequence, it is a fundamental component of math co-processor, DSP processors, embedded arithmetic processors, and data processing units. These components demand high numerical stability and accuracy and hence are floating-point based. Floating-point adder is the most complex operation in a floating-point unit. Moreover, it is a costly operation in terms of hardware and timing, as it needs different types of building blocks with variable latency. In floating-point addition implementations, latency is the overall performance bottleneck. Various algorithms and design approaches have been developed by the VLSI community [13, 14, 15].

3.1.1 Standard floating-point adder algorithm

In this section we analyze the standard floating point algorithm architecture, and the hardware modules designed as part of this algorithm. The standard architecture is the prototype algorithm for floating-point addition in any kind of hardware and software design [2].

Let N1 and N2 be two floating-point numbers. Each number consists of sign (s1, s2), exponent (e1, e2), and significand (f1, f2). Given these two numbers, Figure 3.1 shows the flowchart of the standard floating-point adder algorithm. A description of the algorithm is as follows:

- The two operands, N1 and N2 are read in.

- Then N1 and N2 compared for denormalization and infinity. If numbers are denormalized, set the implicit bit to 0 otherwise it is set to 1. At this point, the fraction part is extended to 24 bits.

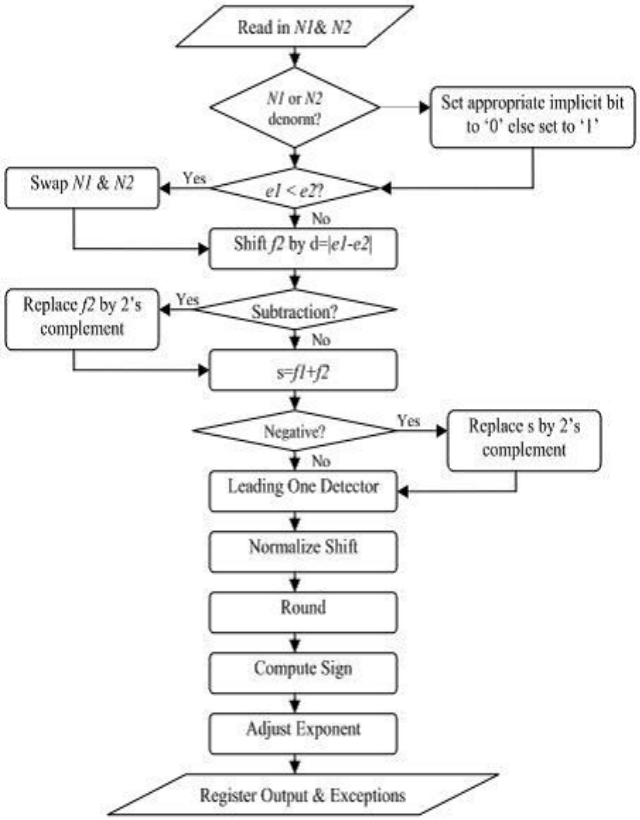


Figure 3.1: Flowchart for standard floating-point adder [2]

- Subsequently, the two exponents, e1 and e2 are compared. If e1 is less than e2, N1 and N2 are swapped, and previous f2 will now be referred to as f1 and vice versa.

- The smaller fraction, f2, is shifted right by the absolute difference result of the two exponents' subtraction. Now both the numbers have the same exponent.

- The two signs are used to see whether the operation is a subtraction or an addition. If the operation is a subtraction, f2 is replaced by 2's complement.

- Now the two fractions are added using a 2's complement adder.

- If the result sum is negative, it has to be inverted and a 1 has to be added to the result.

- The result is then passed through a leading one detector or leading zero counter. This is the first step in the normalization.

- Then, the result is shifted left to be normalized. In some cases, 1-bit right shift is needed.

- Subsequently, the result is rounded towards nearest even, the default rounding mode.

- If the carry out from the rounding adder is 1, the result is left shifted by one.

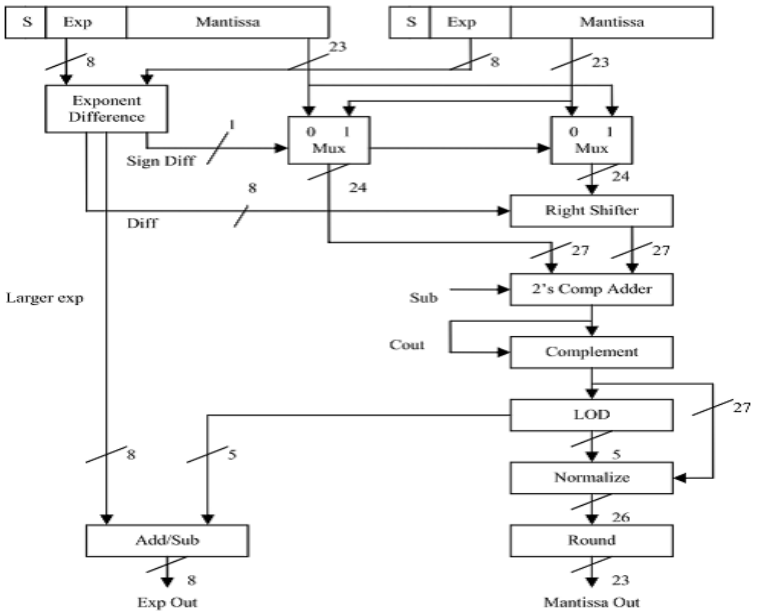


Figure 3.2: Micro-architecture of standard floating-point adder [3]

Table 3.1: Adder implementation analysis

Adder Type	Combinational Delay (ns)	Slices
Ripple-Carry	15.91	18
Carry-Save	11.951	41
Carry-Look Ahead	9.720	39
VHDL	6.018	8

- Using the results from the leading one detector, the exponent is adjusted. The sign is computed and after overflow and underflow check, the result is registered.

Using the above algorithm, the standard floating point adder was designed. Figure 3.2 shows the detailed micro-architecture of the design and the main hardware modules which are necessary for floating-point addition [2].

All these modules add a significant delay to the overall latency of the floating-point adder.

Malik et al. [3] presented a survey, which was directed towards designing different available implementations of all these components separately onto a Virtex 2p FPGA device with a speed grade of 7.

First of all, the authors compared each one of three different adders with the VHDL inbuilt adder function, "+" . 16 bit carry look-ahead adder, carry save adder, and ripple carry adder were designed and synthesized for Virtex2p FPGA. Combinational delay and slices obtained using Xilinx ISE and shown in table 3.1.

Table 3.2: Right shift shifter implementation analysis

Shifter Type	Combinational Delay (ns)	Slices
Align	10.482	71
Barrel	9.857	71
Behavioral	9.357	201

Table 3.3: LOD implementation analysis

LOD Type	Combinational Delay (ns)	Slices
Behavioral	9.05	20
Oklobazija	8.32	18

In order to pre-normalize the mantissa of the number with the smaller exponent, a right shift shifter is used to right shift the mantissa by the absolute exponent difference. Three custom shifters were designed for this purpose. Table 3.2 shows the synthesis results obtained by using Xilinx ISE.

After the addition, the next step is to normalize the result. The first step is to identify the leading or first one in the result. This result is used to shift left the adder's result by the number of zeros in front of the leading one. In order to perform this operation, special hardware, called Leading One Detector (LOD) is implemented. Behavioral and Oklobazija type LOD were implemented and Table 3.3 shows the synthesis results.

Using the results from the LOD, the result from the adder is left shifted to normalize the result. Table 3.4 gives the synthesis results obtained from Xilinx ISE implemented for Virtex2p device.

The design was implemented for only one pipeline stage. The minimum clock period reported by the synthesis tool after placing and routing was 27.059ns. The levels of logic reported were 46. That means the maximum clock speed that can be achieved for this implementation is 36.95 MHz. The number of slices reported by the synthesis tool was 541.

Table 3.4: Left shift shifter implementation analysis

LOD Type	Combinational Delay (ns)	Slices
Behavioral	8.467	80
VHDL	8.565	90

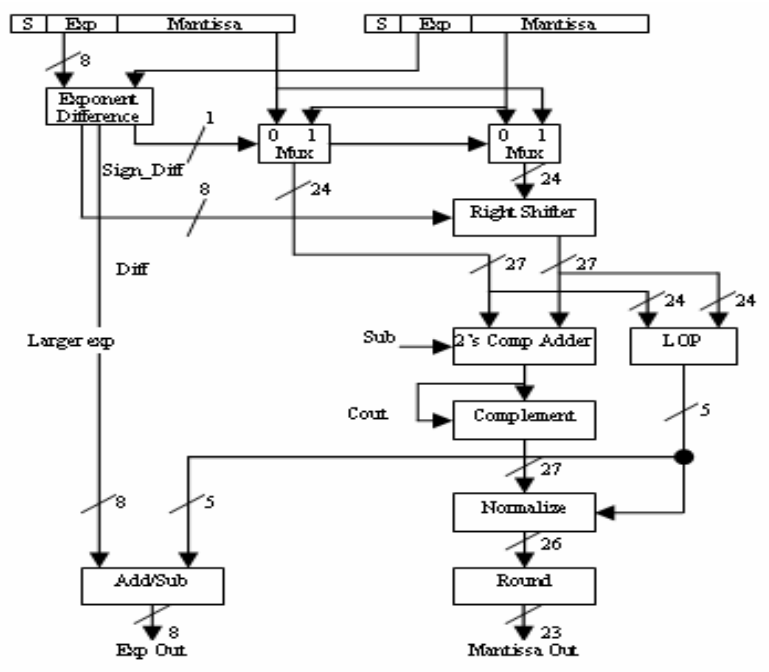


Figure 3.3: Micro-architecture of LOP algorithm [3]

3.1.2 LOP Algorithm

The goal of a floating-point adder is to obtain better overall latency. This improvement, compared with the above adder, is achieved by the Leading One Predictor (LOP) algorithm. Figure 3.3 shows the micro-architecture of the LOP algorithm.

This algorithm was first introduced by Flynn [16] in 1991. Since then there have been number of improvements [14, 17]. The most feasible design was given by J.D. Bruguera and T. Lang [14], and it detects the error concurrently with the leading one detection. Malik et al. [3] show that the main improvement seen in LOP design is the level of logic reduced by 23% with an added expense of increasing the area by 38%. The improvement of the minimum clock period is small. Therefore, it is not a feasible design option for FPGA. However, LOP algorithm is a good option in VLSI design, because, the levels of logic affect the latency.

Another design was given by Malik et al. [4]. In order to achieve five levels of pipeline and gain maximum clock frequency, the LOP has been further pipelined in three stages as shown in figure 3.4.

The tree like structure requires identical modules working in parallel which are exploited to divide the overall latency of the LOP into three different pipeline stages. They used ModelSim [18] to simulate all the modules in a hierarchy design process. After the design was tested for functional correctness, the design was synthesized using Xilinx ISE 6.3i [19] synthesis tool provided by Xilinx. This tool synthesizes HDL for different Xilinx FPGAs and devices. They concluded that by using properly placed pipeline stages, LOP algorithm can be used to significantly improve both latency and area compared to the

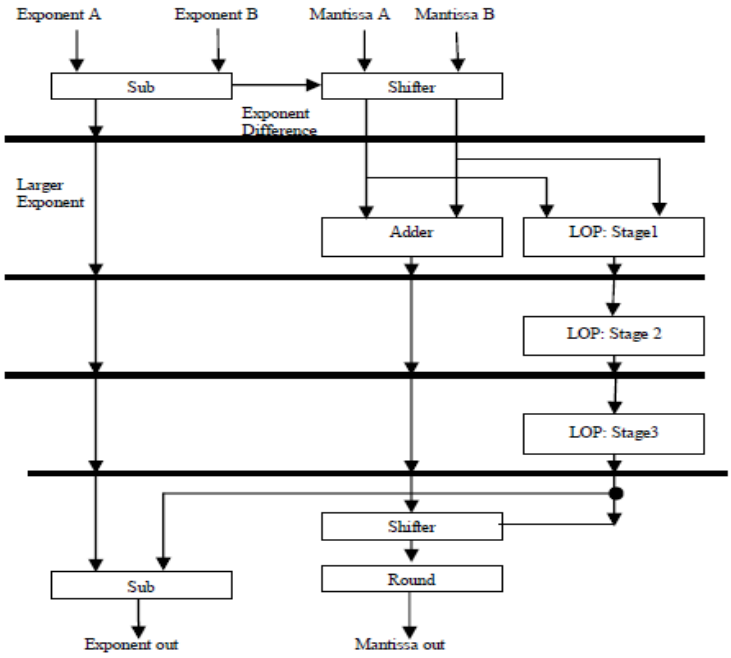


Figure 3.4: The LOP algorithm in three stages [4]

Xilinx Intellectual Property available for FP adder.

3.1.3 Far and Close Data-path Algorithm

According to the studies, 43% of floating-point instructions [20] have an exponent difference of zero or one. In order to manipulate this fact and to improve performance in terms of latency, the data path is divided into far and close path. The close path is computed for all the operations in case the exponent difference is 0 or 1, while the far path computes for the rest of the computations. The algorithm is potentially larger than the previously implemented algorithms but has shown significant improvement in latency and thus used in almost all the present commercial microprocessors. The micro-architecture of far and close data-path algorithm is shown in Figure 3.5.

Malik et al. [3] compared the far and close data-path algorithm to the standard algorithm. They ascertain that one shifter delay and rounding delay has been removed for critical paths of data-paths with help of almost double the hardware and compound adder implementation. Table 3.5 shows a comparison between the standard algorithm and far and close data-path algorithm implementation on a Virtex 2p FPGA device.

The minimum clock period reported by the synthesis tool after placing and routing was 19% better than that of standard floating-point adder implementation. The levels of logic reported were showing 34% improvement. Both these improvements were evident because in both the critical paths one shifter and one adder has been removed. The number of slices reported by the synthesis tool was 1018 which is a significant increase compared to standard algorithm.

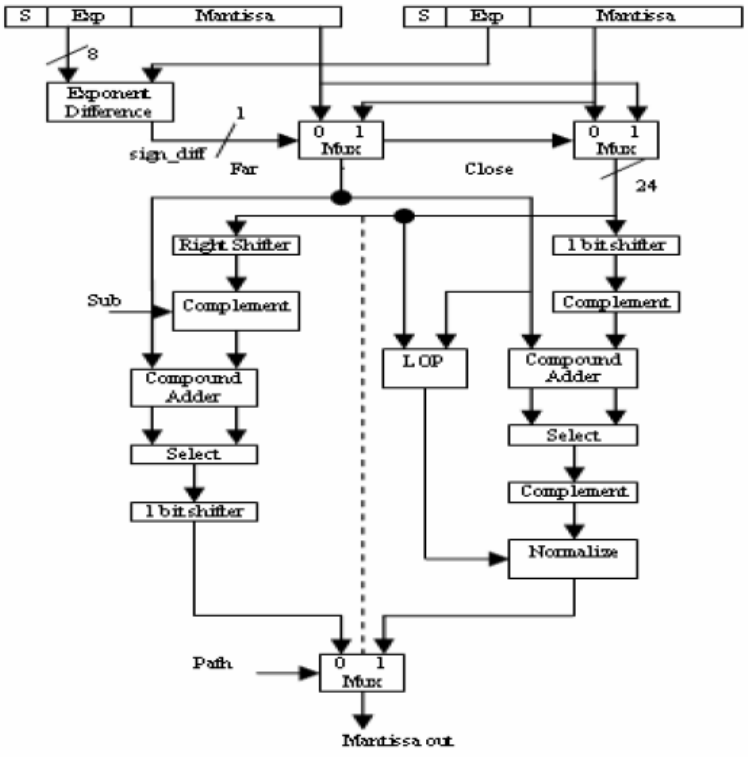


Figure 3.5: Micro-architecture of far and close data-path algorithm [3]

Table 3.5: Standard and F&C algorithm analysis

	Clock Period (ns)	Clock Speed (MHz)	Area (Slices)	Levels of Logic
Standard	27.059	36.95	541	46
F& C	21.821	45.82	1018	30
%of imp.	+19%	+19%	-88%	+34%

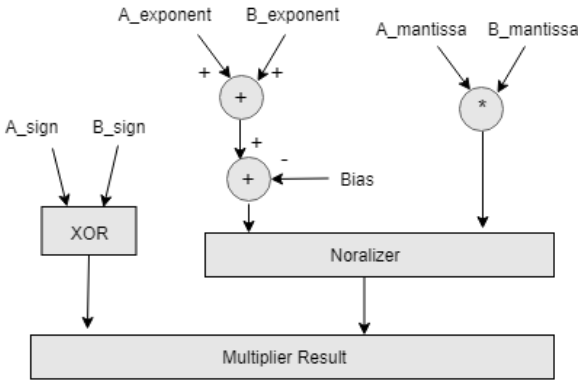


Figure 3.6: Block diagram of floating-point multiplier

3.2 Floating-point multiplier

Floating-point multipliers are widely used in digital signal processing and multimedia. Thus, floating-point multiplication plays a major role in the design and implementation aspects of floating-point processors [21].

3.2.1 Floating-point multiplication algorithm

Floating-point numbers which are normalized have the form which described with the equation (2.23). Significand is the mantissa with an extra MSB bit. The following steps are carried out to multiply two floating-point numbers :

1. Multiplying the significand: i.e. $(1.M_1 * 1.M_2)$.
2. Placing the decimal point in the result.
3. Adding the exponents: i.e. $(E_1 + E_2 - Bias)$.
4. Obtaining the sign by XOR operation of S_1 and S_2.
5. Normalizing the result: i.e. obtaining 1 at the MSB of the results "significand".
6. Rounding operation is performed on the result to fit in the available bits.
7. Checking for underflow/overflow occurrence.

The figure 3.6 shows the block diagram of a floating-point multiplier.

The below example shows a multiplication of two floating-point numbers. For each number, consider a floating-point representation similar to the IEEE 754 single precision floating-point format, but with a reduced number of mantissa bits (only 3). Let A (0 10000100 010 = 40), and B (1 10000001 101 = -6.5) these two numbers.

To multiply A and B the bellow 6 steps are followed:

1. Multiply significand:

   1.010

   x1.101

   1010

   0000

   1010

   1010

   10000010

2. Place the decimal point to the result:10.000010

3. Add exponents:

   10000100

   +10000001

   100000101

   The exponent representing the two numbers is already shifted/biased by the bias value (127) and it is not the true exponent: $E_A = E_{A-true} + bias$, $E_B = E_{B-true} + bias$ and $E_A + E_B = E_{A-true} + E_{B-true} + 2 * bias$. So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

   100000101

   -01111111

   10000110

4. Obtain the sign bit and put the result together: 1 10000110 10.000010

5. Normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1.

   1 10000110 10.000010 (before normalizing)

   1 10000111 1.0000010 (normalized)

   The result is: 1 10000111 000001 (without the hidden bit)

6. The mantissa bits are more than 3 bits (mantissa available bits), and rounding is needed. If we applied the truncation rounding mode then the stored value is: 1 10000111 000

K.Deergha Rao et al. [5] proposed the architecture of the 24x24 bit Vedic real multiplier based on four 12x12 bit Vedic real multipliers. A Vedic multiplier (VM) is faster than the Array multiplier and it has been designed in [22]. The 24 x24 bit VM architecture using 12 x12 bit VMs and RCAs is shown in Figure 3.7.

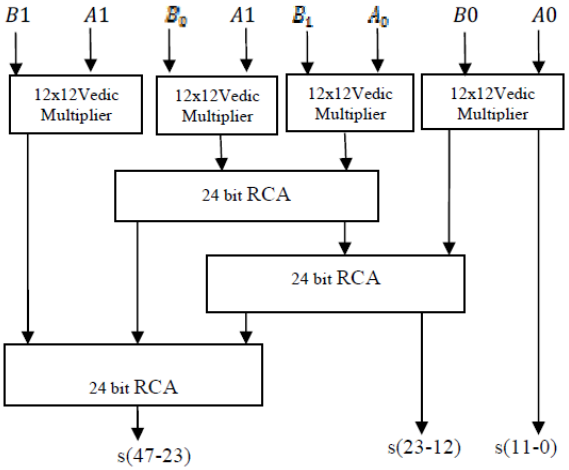


Figure 3.7: Architecture of the conventional 24x24 bit Vedic Multiplier using 12x12 Vedic multipliers [5]

The proposed 32x32 bit complex floating point Vedic multiplier using four floating point real Vedic multipliers solution is coded in VHDL and implemented on .FPGA Device: xc7k480t,ffg1156,C,-3.

The performance of the proposed Vedic, Array, and Booth, 32-bit complex floating point multipliers are compared in terms of delay, power and device Utilization. According to their results, the proposed complex floating point Vedic multiplier is much faster as compared to Booth and Array multipliers but with little increase in power consumption.

Kodali et al.[23] compered three floating-point multiplication algorithms, namely Booth, normal Karatsuba, and recursive Karatsuba. They have implemented these algorithms for both single and double precision using VHDL. It has been synthesized and routed on Virtex-6 FPGA target using Xilinx ISE. Simulation results have been analyzed in ModelSim-SE. They observed that the recursive Karatsuba algorithm performs better than normal Karatsuba and Booth algorithm. Moreover, the recursive Karatsuba algorithm has the least FPGA resources utilised and the speed is relatively high. So, the authors came to the conclusion that recursive Karatsuba is the best algorithm among the three algorithms.

In [6] compared performance of multiplexer based on single precision floating point multipliers like Array, Wallace Tree and Vedic multipliers are done. The above multipliers shows in figures 3.8, 3.9 and 3.7 respectively. The single-precision floating point multiplier, which is presented in this paper, is designed using Verilog. The simulation and synthesis of the different multipliers are performed by using Xilinx ISE and performance parameters are summarized. Functional verification of Array, Wallace tree and Vedic are performed and results are summarized.

In this paper different single precision floating-point multipliers are analyzed using 4x1 and 2x1 multiplexers and regular full adder based single precision floating-point multipliers with respect to area and delay. According the results 4x1 multiplexer based Vedic

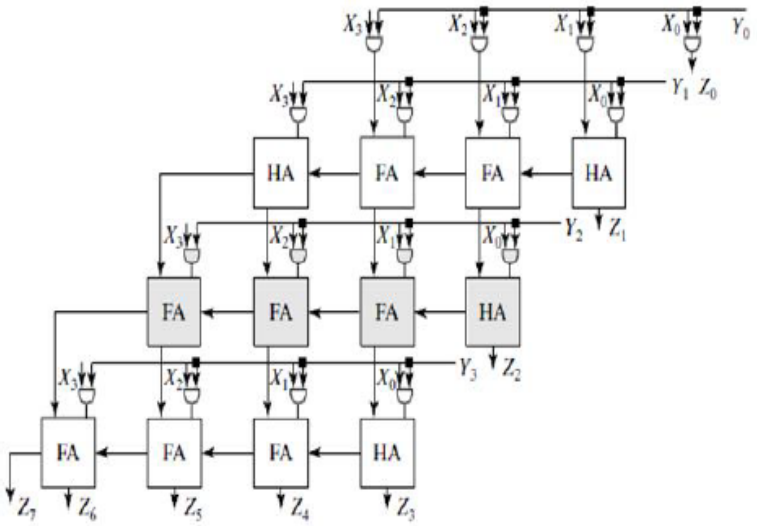


Figure 3.8: 4x4 array multiplier [6]

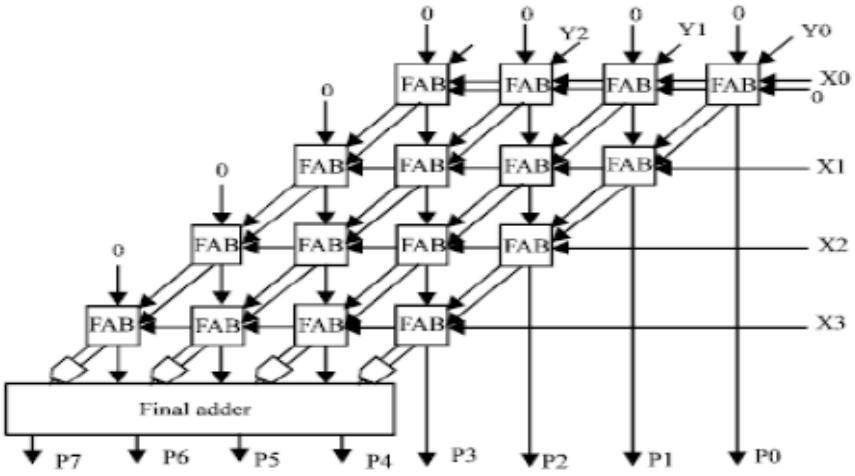


Figure 3.9: 4x4 Wallace Tree multiplier [6]

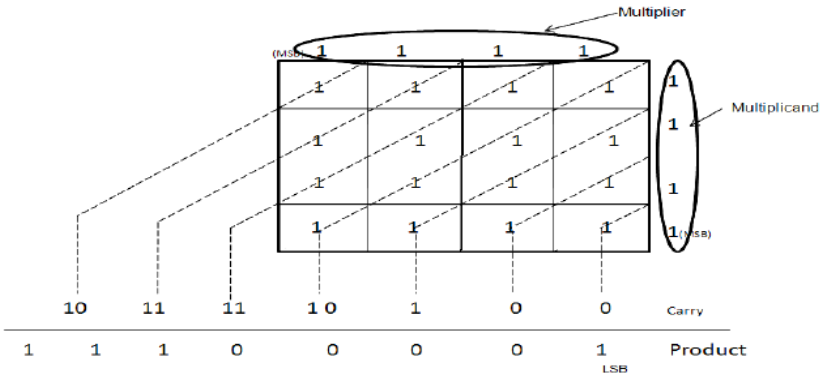


Figure 3.10: Steps involved for 4-bit binary numbers multiplication using Urdhva Tiryagbhyam Technique [7]

multiplier has the lowest delay compared to that of other multipliers. Further, the analysis is extended to Vedic with 2x1 multiplexer to improve the delay. By using 2x1 multiplexer, floating-point multiplication requires 1003 LUTs and 548 slices and delay is 43.61 ns. By reducing complexity the overall performance can be improved. From the results it is concluded that Single Precision Vedic multiplication using 2x1 multiplexer gives better performance in terms of area and delay.

Y. Srinivasa Rao et al.[7] compered two implementation of double-precision floating-point multipliers. One of them using Urdhva Tiryagbhyam technique and the other using Karatsuba technique. The Urdhva Tiryagbhyam technique [24] can take (2n-1) steps for designing the n-bit multiplier. In figure 3.10 a 4-bit multiplier can take 7-steps for designing of the multiplier, using Urdhva Tiryagbhyam technique.

The 4-bit multiplier has 4-bit multiplier as one of the operand and 4-bit multiplicand is the second operand. The carry resents the previous state generated carry and it can be added to the current to get the final product.

According to [7] the delay of the Karatsuba multiplier is 18.139ns and the delay of the Urdhva Tiryagbhyam multiplier is 15.034ns. As a consequence Urdhva Tiryagbhyam multiplier is faster in comparison with the Karatsuba multiplier. Also the Urdhva Tiryagbhyam multiplier consumes less power in comparison with the Karatsuba multiplier.

Chapter 4

Related Work in the Field of Approximate Computing

4.1 Approximate computing

Approximate Computing (AC) is a wide spectrum of techniques that reduce the accuracy of computation in order to improve energy, performance, and other metrics. Approximate computing has attracted significant traction for both academia and industry. AC exploits the fact that several applications, such as machine learning and multimedia processing, do not necessarily need to produce precise results to be useful. For instance, we can use a lower resolution image encoder in applications where high-quality images are not necessary. By relaxing the numerical equivalence between the specification and implementation of error-tolerant applications, approximate computing deliberately introduces "acceptable errors" into the computing process and promises significant energy-efficiency gains.

Various approximate computing techniques at almost all computing layers have been presented in the literature, over the year. There are different techniques for the implementation of AC. A classifications of different AC techniques is illustrated in Figure 4.1. As it shown those techniques are classified into hardware-based and software-based. Techniques at the hardware level [25], [26], [27], [28], [29] have less accurate and high efficient energy components. Software-level techniques [30], [31], [32] reduce calculations or accesses to memory to improve performance at the expense of precision in the results [33].

4.2 Approximate software

The most essential techniques for AC based on software are: code perforation, bit width reduction, float point to fixed point conversation, synchronization elision.

Code perforation

This technique is based on identifying parts of code that can be discarded without exceeding an error threshold. This technique includes loop perforation. Loop perforation

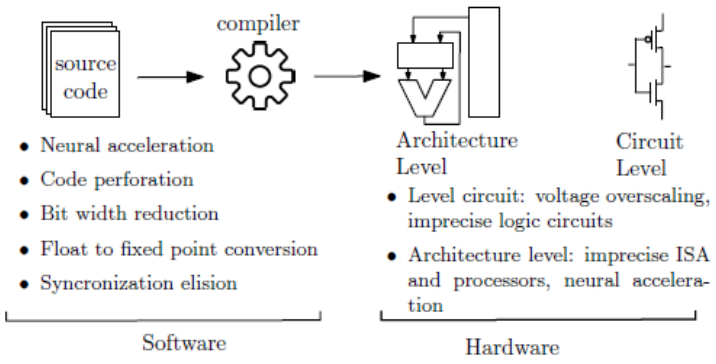


Figure 4.1: Classification of different approximate computing techniques [8]

transforms loops to execute a subset of their iterations. The goal is to reduce the amount of computational work that is required to produce its result.

The survey which conducted by Sidiroglou-Douskos et al. [34] shows that loop perforation can effectively increase a range of applications with the ability to operate at various attractive points in the tradeoff space. The applications, which have been perforated, are often able to deliver significant performance improvements, at the cost of a small (typically 5% or less) decrease in the accuracy. The results from their implemented loop perforation system show that loop perforation can dramatically increase the ability of applications to achieve decrease of the energy consumption and/or increase of the performance in exchange for the decrease of the accuracy.

Bit width reduction

The improvement of perforation and power consumption can take place by reducing the accuracy, which is basically the number of bits of a floating point number. In [35] the authors present a method based on the approximate computing, in order to hide information in internet of things devices. This can be achieved by using some less significant bits of number mantissa. They introduce the threats to IoT devices, the available mitigation methods, and propose an approximate computing based security primitive that enables us to embed various information during the process of approximate computing. The authors, also, demonstrate how information can be generated and hidden for several security applications.

Float point to fixed point conversation

In order to reduce energy cost, an approximation of a floating point program can be used. According this technique Aaomodt et al. [36] noted some impressive speedups. In their paper is shown that an SQNR improvement equivalent to carrying up to 2.0 extra bits of precision throughout the computation is achievable using IRP-SA in conjunction with the FMLS operation. Furthermore, they emphasize in the possibility of an achievable 13% speedup, by simply adding a FMLS operation with a few output shift distances. In addition, a complementary-scaling technique (IDS) was proposed. This technique enables the parameterization of the fixed-point scaling of a variable depending upon the context of

its definitions and uses. An implementation of IDS specialized to single-nested loops was found to improve accuracy of a lattice filter benchmark by the equivalent of more than 16 bits of precision.

Synchronization elision

In parallel applications, synchronization overhead is a major performance-limiting factor. Synchronization relaxing increases the possibility of performance and efficiency improvement. Renganarayana et al. [37] have shown that significant performance speedups can be obtained by relaxing the widespread assumption that all programmer specified synchronizations are essential to produce good quality results. Their experiments with a variety of benchmarks show that relaxing synchronization using this methodology can achieve significant speedups. For example, up to 15x for the K-means benchmark and up to 3x on top of the expert optimized BFS kernel in Graph500, with no degradation in the quality of the results. The positive results presented on relaxed synchronization are a strong indication of the potential of the general area of approximate computing, where they sacrifice the determinism and preciseness of the general computing paradigm as practiced today for improved latency, reduced energy consumption, and lower system cost, while providing results acceptably close to what would otherwise be possible.

4.3 Approximate hardware

4.3.1 Approximate architecture

Hardware consist of processor, memory, and storage. It is difficult to improve the performance, energy efficiency, and density of these component in parallel, because improving one often sacrifices the other. Approximate computing can improve performance and energy efficiency, in case of processors, as well as density for memories and storage.

For processor architectures many approaches have been proposed to run energy-efficient code segments or single instructions. Chippa et al. [38] present an integrated and systematic approach to approximate computing in hardware. They designed and fabricated a Recogniton and Mining (RM) processor and demonstrated 2-20X savings in energy over a variety of applications. Furthermore, Esmaeilzabeth et al. [39] proposed an ISA (Instruction Set Architecture) that simplifies the hardware by relying on the compiler to provide certain invariants statically, eliminating the need for checking or recovery at run time. The authors describe a high-level microarchitecture that supports interleaved high and low-voltage operations and a detailed design for a dual-voltage SRAM array that implements approximation-aware caches and registers. They model the power of their proposed dual-voltage microarchitecture and evaluate its energy consumption in the context of a variety of error-tolerant benchmark applications. Experimental results show energy savings up to 43%. In the same way, Sampson et al. [40] proposed to use a type system based on information-flow tracking ideas: variables and objects can be declared as approximate or precise; approximate data can be processed more cheaply and less reliably; and they can

statically prove that approximate data does not unexpectedly affect the precise state of a program. Their type system provides a general way of using approximation. They implement their type system on top of Java and experiment with several applications, from scientific computing to image processing to games. Their results show that annotations are easy to insert: only a fraction of declarations must be annotated and endorsements are rare. Once a program is annotated for approximation, the runtime system or architecture can choose several approximate execution techniques. The hardware-based model, that they proposed, shows potential energy savings between 7% and 38%.

Alternatively, performance can be improved by transforming segments of code into a neural-inspired algorithm running on hardware accelerators. A proposal to accelerate code based on Parrot transformation is shown in [25].

4.3.2 Approximate circuit

For a given circuit, we could achieve reduced energy consumption by lowering its supply voltage without reducing the corresponding operational frequency. Over the years, there are several proposals in the literature for approximate arithmetic units. One of those proposals is adders. A few representative approximate adder designs are described in the following.

In [41] a novel approximate adder design to considerably reduce energy consumption with a very moderate error rate, has been presented for energy efficient neuromorphic VLSI systems. The results show that the proposed adder is 2.4x faster and 43% energy efficient over traditional adders. Accordingly, the proposed design approach is applicable to energy efficient neuromorphic VLSI system designs. Ye et al. [42] proposed a reconfiguration-orient approximate adder design. In particular, experimental results demonstrate that they simultaneously achieve much better throughput and image quality when applying their adder to a discrete cosine transform (DCT) application.

Another important arithmetic unit used in computing is the multiplier. Kulkarni et al. [43] proposed a power-efficient multiplier contribution with 2x2 approximate multiplier blocks. With a mean error of 1.39% - 3.35% and power savings between 30% - 50%, the underdesigned multiplier architecture presented allows for trading of accuracy for power. The results suggest that design-for-error based techniques have significant potential for power savings, and can be easily integrated into today's automated ASIC design flow.

Chapter 5

Proposed Floating-Point Multiplier

5.1 Introduction

In this chapter, we will analyze all the work that has been constructed in this diploma thesis. First of all, two different floating-point multipliers will be described, in section 5.2. Then our proposed approximate multipliers are explained in section 5.3.

5.2 Accurate floating-point multipliers

Two floating-point multipliers will be analyzed, in this diploma thesis. The one multiplier has as input two floating-point numbers which are encoded in the binary16 interchange format. The other multiplier has as input two floating-point numbers which are encoded in the binary32 interchange format. As referred in subsection 2.5.2.1 a normalized number in represented by equation (2.23).

Let A and B be two n-bit floating-point numbers. According to the equation (2.23):

$$A = (-1)^{S_A} * 2^{(E_A - Bias)} * (1.M_A) \tag{5.1}$$

and

$$B = (-1)^{S_B} * 2^{(E_B - Bias)} * (1.M_B) \tag{5.2}$$

Using the above equations, the product (P) of A and B is:

$$P = A * B = (-1)^{(S_A + S_B)} * 2^{(E_A + E_B - 2*Bias)} * (1.M_A * 1.M_B) \tag{5.3}$$

Table 5.1: XOR accuracy table

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

To multiply A and B the bellow 7 steps are followed:

1. Obtaining the sign S_P by XOR operation of S_A and S_B. If A and B have the same sign-bit, the sign-bit of P is 0, else it is 1. The truth table of Exclusive OR Gate is shown in Table 5.1.

2. Adding the exponents: $E = E_A + E_B$

3. Checking for underflow/overflow.

   – For 16x16 bit multiplier, the stored exponent ($E_P = E_A + E_B - Bias$) is 5-bit. For real numbers E_P is greater than 0 and less than 31. So, $0 < E_P < 31 \Rightarrow 0 < E_A + E_B - 15 < 31 \Rightarrow 15 < E_A + E_B < 46 \Rightarrow 15 < E < 46$.

   According to the exponent's and mantissa's values a floating-point number (binary16) referred as zero, subnormal, normal, infinite or NaN. These five cases are summarized as follow:

   * If $E < 15$, the result of multiplier is underflow.
   * If $E = 15$, then $E = 15 \Rightarrow E_A + E_B = 15 \Rightarrow E_A + E_B - 15 = 0 \Rightarrow E_P = 0$. The product P which has zero exponent ($E_P = 0$), represent either zero number (if mantissa of P (M_P) is zero), or subnormal number (if mantissa of P (M_P) is not zero). Zero and subnormal numbers are considered as underflow. These numbers may turn to normalized numbers during normalization.
   * If $16 \leq E \leq 45$, the result is normalized number (real number). A normalized number may turn in overflow during normalization.
   * If $E > 45$, the result is overflow. In the specific case where $E = 46$: $E = 46 \Rightarrow E_A + E_B = 46 \Rightarrow E_A + E_B - 15 = 46 - 15 \Rightarrow E_P = 31$.
     The product P which has $E_P = 31$, represent either infinite number (if mantissa of P (M_P) is zero), or NaN number (if mantissa of P (M_P) is not zero). Infinite and NaN numbers are considered as overflow.

Table 5.2: Normalization Effect on Result's E and Overflow/Underflow Detection (binary16)

Category	E	Comments
Underflow	E<15	Can't be compensated during normalization
	E=15	May turn to normalized number during normalization (dy adding 1 to it)
Normalized number	$16 \leq E \leq 45$	May result in overflow during normalization
Overflow	E>45	Can't be compensated

According to the value of E, the number is categorized as it shown in table 5.2. If E is greater than 45, the result is overflow. If E is less than 16, the result is underflow.

- For 32x32 bit multiplier, the stored exponent ($E_P = E_A + E_B - Bias$) is 8-bit. For real numbers E_P is greater than 0 and less than 255. So, $0 < E_P < 255 \Rightarrow 0 < E_A + E_B - 127 < 255 \Rightarrow 127 < E_A + E_B < 382 \Rightarrow 127 < E < 382$.

  According to the exponent's and mantissa's values a floating-point number (binary32) referred as zero, subnormal, normal, infinite or NaN. These fivecases are summarized as follow:

  * If $E < 127$, the result of multiplier is underflow.
  * If $E = 127$, then $E = 127 \Rightarrow E_A + E_B = 127 \Rightarrow E_A + E_B - 127 = 0 \Rightarrow E_P = 0$.

    The product P which has zero exponent ($E_P = 0$), represent either zero number (if mantissa of P (M_P) is zero), or subnormal number (if mantissa of P (M_P) is not zero). Zero and subnormal numbers are considered as underflow. These numbers may turn to normalized numbers during normalization.
  * If $128 \leq E \leq 381$, the result is normalized number (real number). A normalized number may turn in overflow during normalization.
  * If $E > 381$, the result is overflow. In the specific case where $E = 382$: $E = 382 \Rightarrow E_A + E_B = 382 \Rightarrow E_A + E_B - 127 = 382 - 127 \Rightarrow E_P = 255$.
    The product P which has $E_P = 255$, represent either infinite number (if mantissa of P (M_P) is zero), or NaN number (if mantissa of P (M_P) is not zero). Infinite and NaN numbers are considered as overflow.

  According to the value of E, the number is categorized as it shown in table 5.3. If E is greater than 381, the result is overflow. If E is less than 128, the result is underflow.

Tables 5.2 and 5.3 show that an underflow result may turn to normalized number,

Table 5.3: Normalization Effect on Result's E and Overflow/Underflow Detection (binary32)

Category	E	Comments
Underflow	E<127	Can't be compensated during normalization
	E=127	May turn to normalized number during normalization (dy adding 1 to it)
Normalized number	$128 \leq E \leq 381$	May result in overflow during normalization
Overflow	E>381	Can't be compensated

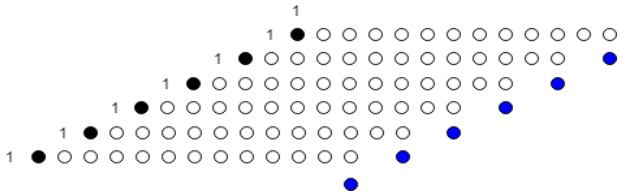


Figure 5.1: The partial product tree of a 12x12 accurate modified booth multiplier

and a normalized number may result in overflow during normalization. These two cases will be analyzed in stage 7.

4. We should subtract the bias from the E: $E_P = E - Bias$ (E_P is the stored exponent of product P)

5. The significands $(1.M_A, 1.M_B)$ will be multiplied by using a modified booth multiplier (subsection 2.4.2).

   – In case of 16x16 bit multiplier, the number of mantissa's bits is 10. Significand is the mantissa with an extra MSB bit. So 1.M consists of 11 bits. The significand is not negative number. Thus, we should add an extra zero bit in front of 1.M. The modified booth multiplier will be a 12x12 bit multiplier. The two numbers are:

   $X_{12} = 01.M_A$,

   $Y_{12} = 01.M_B$.

   Figure 5.1 shows the partial product tree of a 12x12 accurate modified booth multiplier.

   – In case of 32x32 bit multiplier, the number of mantissa's bits is 23. Significand is the mantissa with an extra MSB bit. So 1.M consists of 24 bits. The significand is not negative number. Thus, we should add an extra zero bit in front of 1.M. Now, each number consists of 25 bits. Because, Y must be encoded in modified booth algorithm, it needs an extra zero bit in front of 01.M, this time. The modified booth multiplier will be a 26x26 bit multiplier. The two numbers are:

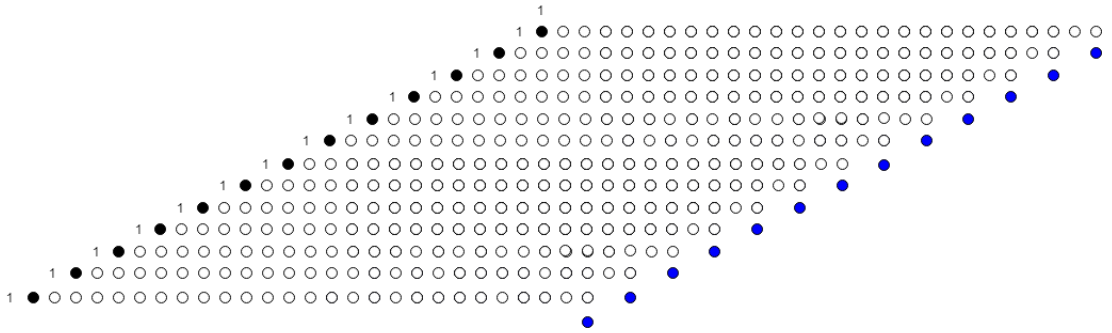


Figure 5.2: The partial product tree of a 26x26 accurate modified booth multiplier

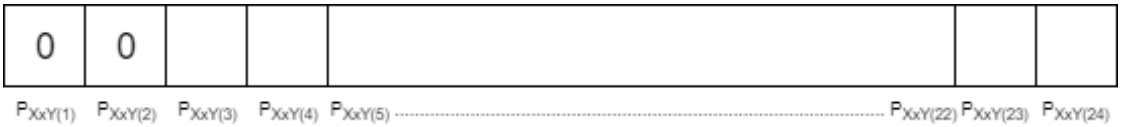


Figure 5.3: The product (P) of the 12x12 multiplier

$X_{26} = 001.M_A$,
$Y_{26} = 001.M_B$.
Figure 5.2 shows the partial product tree of a 26x26 accurate modified booth multiplier.

6. Normalize the result so that there is a 1 just before the radix point (decimal point).

   – In case of 12x12 bit multiplier the result $P_{X\times Y}$ consists of 24 bits. The first 2 bits are 0. Figure 5.3 shows the result $P_{X\times Y} = 00xx.xxxx\ldots xxxx$.
   If $P_{X\times Y}(3) = 1$, the radix point must be moved one place to the left and the exponent E_P must be increased by 1. The mantissa's bits are more than 10 bits (mantissa available bits), and we applied the truncation rounding mode. The stored value of mantissa is from $P_{X\times Y}(4)$ to $P_{X\times Y}(13)$ ($M_P = P_{X\times Y}(4)\ldots P_{X\times Y}(13)$).
   Else, the exponent E_P left unchanged and the stored value of mantissa is from $P_{X\times Y}(5)$ to $P_{X\times Y}(14)$ ($M_P = P_{X\times Y}(5)\ldots P_{X\times Y}(14)$).
   – In case of 26x26 bit multiplier the result $P_{X\times Y}$ consists of 52 bits. The first 4 bits are 0. Figure 5.4 shows the result $P_{X\times Y} = 0000xx.xxxxx\ldots xxxxx$.
   If $P_{XxY}(5) = 1$, the radix point must be moved one place to the left and the exponent E_P must be increased by 1. The mantissa's bits are more than 23 bits (mantissa available bits), and we applied the truncation rounding mode. The stored value of mantissa is from $P_{X\times Y}(6)$ to $P_{X\times Y}(28)$ ($M_P = P_{X\times Y}(6)\ldots P_{X\times Y}(28)$).

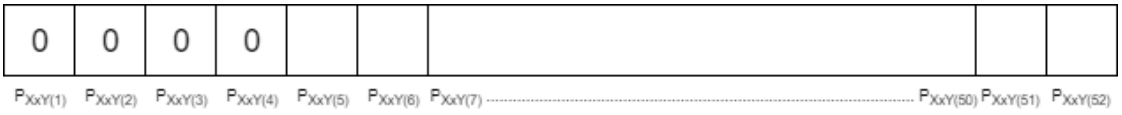


Figure 5.4: The product (P) of the 26x26 multiplier

Table 5.4: The multiplier's output according to the categorization of the number (binary16)

Category	Product P = AxB (sign)	Product P = AxB (exponent)	Product P = AxB (mantissa)
Underflow	$S_P = S_A \oplus S_B$	00000	0000000000
Normalized number	$S_P = S_A \oplus S_B$	E_P (stage 4, 6, 7)	M_P (stage 6)
Overflow	$S_P = S_A \oplus S_B$	11111	0000000000

Table 5.5: The multiplier's output according to the categorization of the number (binary32)

Category	Product P = AxB (sign)	Product P = AxB (exponent)	Product P = AxB (mantissa)
Underflow	$S_P = S_A \oplus S_B$	00000000	00000000000000000000000
Normalized number	$S_P = S_A \oplus S_B$	E_P (stage 4, 6, 7)	M_P (stage 6)
Overflow	$S_P = S_A \oplus S_B$	11111111	00000000000000000000000

Else, the exponent E_P left unchanged and the stored value of mantissa is from $P_{X\times Y}(7)$ to $P_{X\times Y}(29)$ ($M_P = P_{X\times Y}(7) \dots P_{X\times Y}(29)$).

7. Two addition checks.
   - If $E = 15$ for binary16 (or $E = 127$ for binary32) (stage 2) and the exponent E_P was increased by 1 from stage 6, the number referred as normalized.
   - If $E = 45$ for binary16 (or $E = 381$ for binary32) (stage 2) and E_P was increased by 1 from stage 6, the result is overflow.

Tables 5.4, 5.5 show the multiplier's output according to the categorization of the number.

The flowchart of our accurate floating-point multiplier shows in figure 5.5.

5.3 Proposed approximate multiplier

In order to improve energy, performance, and other metrics we use an approximation in our design.

5.3.1 Hybrid partial product perforation-rounding

The authors in [9] proposed a hybrid technique combining the partial product perforation of [44] with a truncation/rounding method. This approximation will be used in our design. Specifically when we multiply the numbers X and Y (stage 5).

Let X and Y be two n-bit 2's complement binary numbers. Using the equation (2.9) their product $X \times Y$ is calculated as follows:

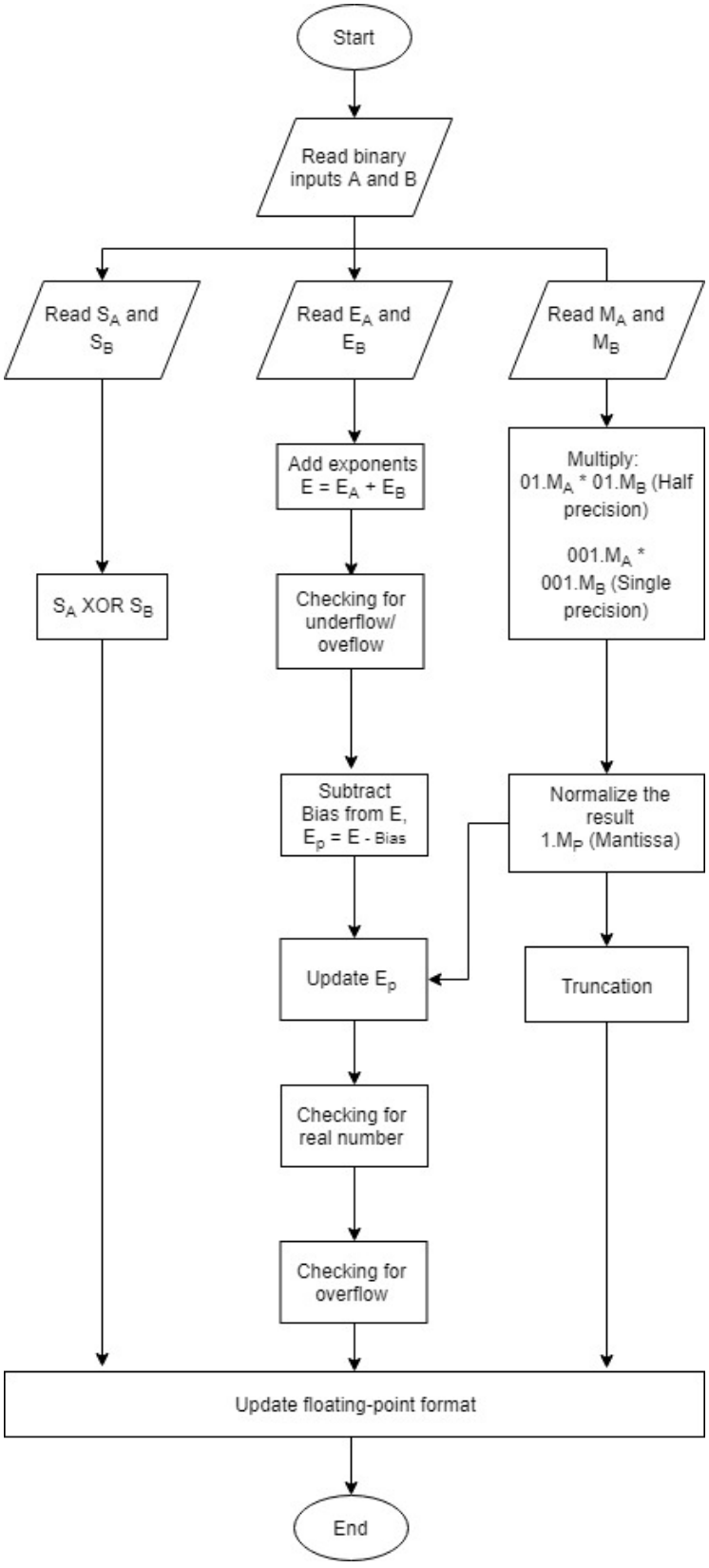


Figure 5.5: Flowchart of our accurate floating-point multiplier

$$X \times Y = \sum_{j=0}^{n/2-1} X * {y_j}^{MB} * 4^j \tag{5.4}$$

where ${y_j}^{MB} \in \{0, \pm 1, \pm 2\}$.

Partial Product Perforation

The partial product perforation technique, which was presented in [9], dismisses the generation of k successive partial products starting from the least significant ones. Therefore the k least significant modified Booth digits are not generated; namely, the 2k LSBs of Y (including y_{-1}) are discarded. Thus, the product $X \times Y$ is calculated approximately by the next equation:

$$X \times Y|_k = \sum_{j=k}^{n/2-1} X * {y_j}^{MB} * 4^j \tag{5.5}$$

Partial Product Rounding

In the partial product rounding technique the m-1 LSBs of X are discarded, and x_{m-1} is added with the most significant remaining part (X_m), as follows:

$$X_m + x_{m-1} = \langle x_{n-1}, x_{n-2}...x_m \rangle_{2,s} + x_{m-1} \tag{5.6}$$

The truncation of the m-1 LSBs would lead to significant errors in the calculations. The last remaining LSB (x_{m-1}) is added to X_m to lead to smaller errors. The partial products with modified booth encoding are produced combining two cases.

In case of $x_{m-1} = 0$, the inexact partial products $\left(\tilde{P}_j\right)$ are calculated by $\tilde{P}_j = \left(X_m + 0\right) * {y_j}^{MB} = X_m * {y_j}^{MB}\right)$

In case of $x_{m-1} = 1$, and using the relation $X_m + 1 = -\overline{X}_m$, the inexact partial products $\left(\tilde{P}_j\right)$ are calculated by $\tilde{P}_j = (X_m + 1) * {y_j}^{MB} = \left(-\overline{X}_m\right) * {y_j}^{MB} = \overline{X}_m * \left(-{y_j}^{MB}\right)$, where $\left(-{y_j}^{MB}\right) = (-1)^{\overline{s}_j} * (2 * two_j + one_j)$. Using the relation ${X_m}^* = X_m \oplus x_{m-1}$ to form X_m or $\overline{X}_m$ the two cases are combined. Similarly ${s_j}^* = s_j \oplus x_{m-1}$ is used to form either ${y_j}^{MB}$ or $-{y_j}^{MB}$. Therefore, the partial products are computed by $\tilde{P}_j = {X_m}^* * {y_j}^{MB^*}$, where ${y_j}^{MB^*} = (-1)^{{s_j}^*} * (2 * two_j + one_j)$.

Hybrid Partial Product Perforation-Rounding

Partial product perforation and partial product rounding are combined to form the proposed technique called hybrid partial product perforation-rounding. The proposed

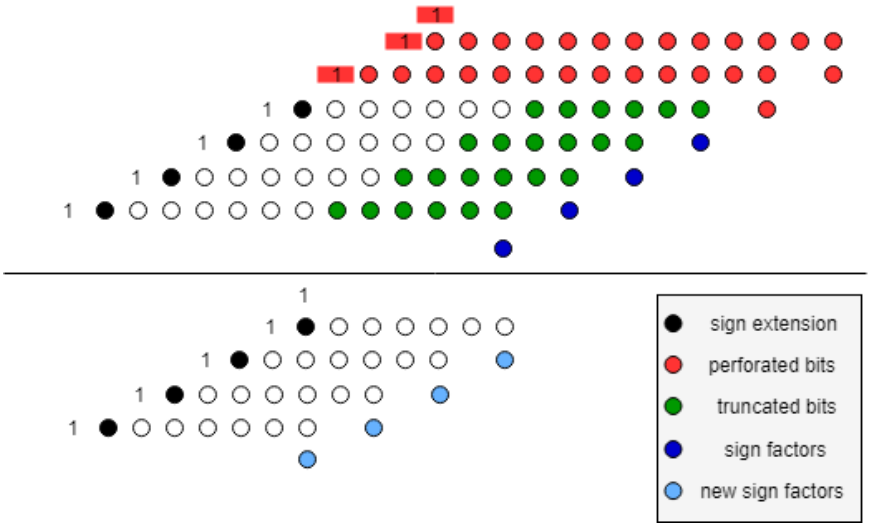


Figure 5.6: An approximate 12x12 bit multiplier with k=2 and m=6

approximate multiplier is characterized by two configuration parameters: k and m. The parameter k refers to the number of the perforated partial products starting from the least significant ones, and m labels the partial product bit that rounding is applied to. The notation $PR|_{k,m}$ is used to label the selected configuration. Note that k ∈ [0, n/2-1) and m ∈ [0, n-1). Hence, in the proposed hybrid technique, the multiplication is performed approximately as follows:

$$X \times Y|_{k,m} = \sum_{j=k}^{n/2-1} \tilde{P}_j * 4^j = \sum_{j=k}^{n/2-1} {X_m}^* * {y_j}^{MB^*} * 4^j \tag{5.7}$$

The implementation of the partial product accumulation is performed by an accurate Wallace tree, whereas its carry-save output is added by a prefix (fast) adder. The accumulation tree, except for the properly weighted partial products, includes the correction term ("1" and sign factors). The n/2-k sign factors (${c_j}^*$) are defined as ${c_j}^* = {s_j}^* * (one_j + two_j)$.

Figure 5.6 shows an approximate 12x12 bit multiplier with k=2 and m=6.

Figure 5.7 shows an approximate 26x26 bit multiplier with k=3 and m=8.

The flowchart of proposed approximate floating-point multiplier shows in figure 5.8. The approximate design has an extra check. The output of mantissa's (1.xx...xx * 1.xx...xx) multiplication has the form xx.xxxx.....xxxx. The bits left to the radix point can be 01 or 10 or 11 in the accurate multiplier. In the approximate design, these two bits can also be zero. So this case is checked and the output is turned to overflow.

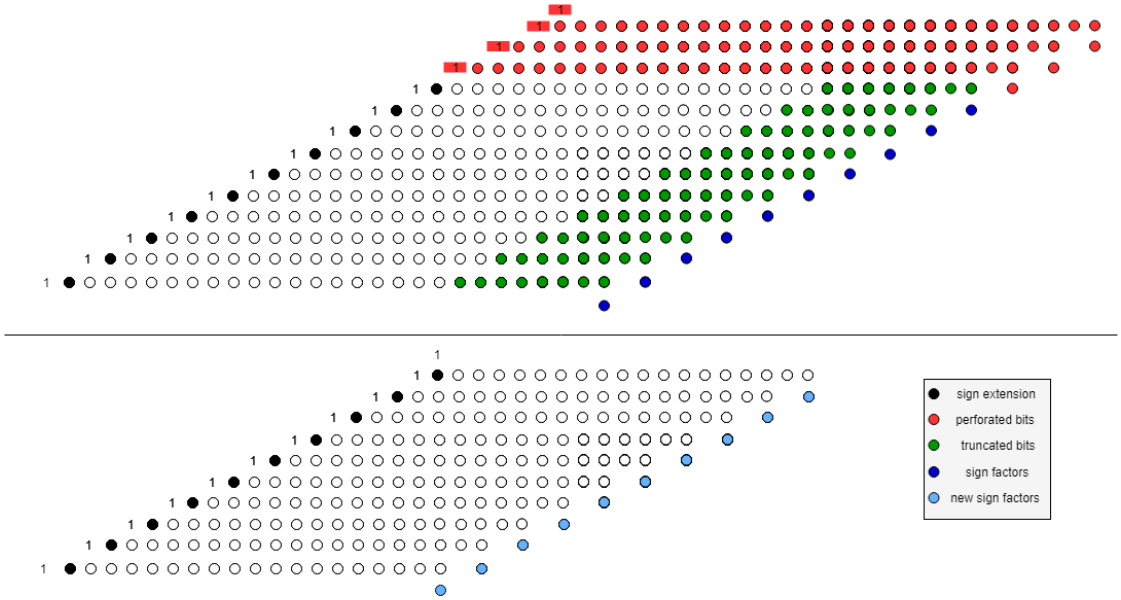


Figure 5.7: An approximate 26x26 bit multiplier with k=3 and m=8

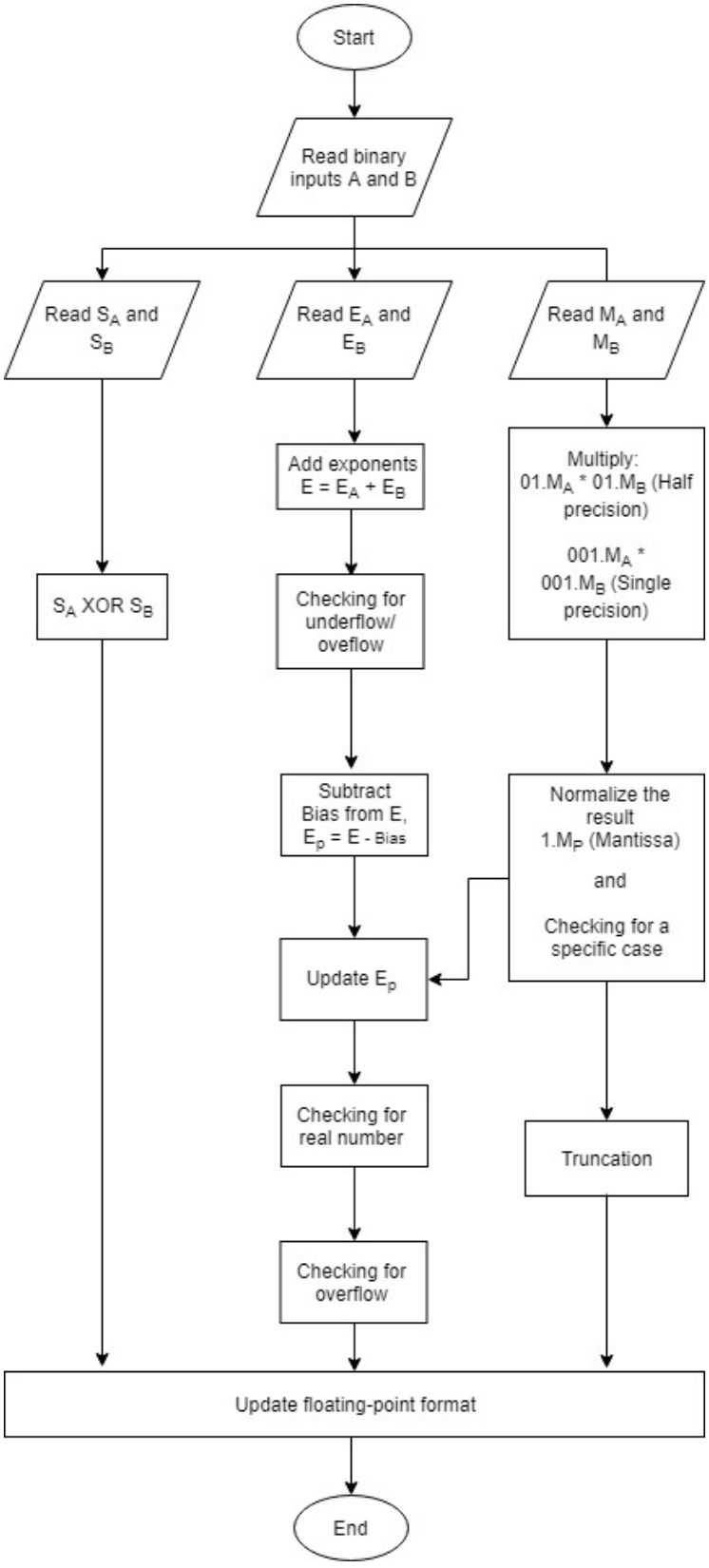


Figure 5.8: Flowchart of proposed approximate floating-point multiplier

Chapter 6

Experiment

6.1 Tools and Experimental Setup

All the multipliers to be compared, are implemented in Verilog HDL, synthesized using Synopsys Design Compiler and the TSMC 65-nm standard cell library, and simulated with Mentor Graphics ModelSim. The critical path delay and the area are reported by Synopsys Design Compiler, while the power consumption is measured with Synopsys PrimeTime-PX tool using all the possible input combinations. All the designs are synthesized and simulated at 1V, i.e., the nominal supply voltage. The procedure, that was followed, consists of the next steps:

- The project is created via the command **$create_project**.
- The Verilog code and the testbench are prepared, and they are checked via the commands **$make check_vlog** and **$make check_tb**, respectively.
- The rtl simulation is occurred via the command **$make rtl_sim**. Mentor Graphic ModelSim opens and the simulation begins. After that, the functional correctness of our design is checked. So, with the help of matlab, we implement and simulate our design in the software level. Matlab and rtl simulation generate an output file of the results of 100000 multiplications. We compare the two output files and if they are identical, the design is correct.
- The command **$make env** set all the parameters of the synthesis (clock period, library etc.). Next, via the command **$make dcsyn** the design synthesis happens. This step is repeated until the critical path (smallest delay) is found. The command **$make dcsyn** uses the tool Synopsys Design Compiler and the TSMC 65-nm standard cell library in order to find out if the time constrains are violated and also computes the total area needed for the given parameters.
- The command **$make sta** defines the clock period of the testbench to be the same with the clock period, which was used in the design synthesis. Furthermore, the

Static Time Analysis (STA) provides a more accurate answer of whether the time constrains are met. Next, the command **$make gate_sim** is performed and a gate level simulation is occurred. Lastly, the command **$make power** computes the power consumption with the help of the Synopsys PrimeTime-PX tool.

- The final step is to compute the error of the approximate multiplier. To achieve this, matlab is used, where the comparison of the accurate and the approximate result leads to the computation of the average error.

6.2 Error Analysis

In this section, we analyze the accuracy of the proposed multiplier adopting the mean relative error distance (MRED) metric [45] due to its advantage of being less affected by the input's distributions. RED is defined as the arithmetic difference between the accurate and the approximate product divided by the accurate product, while MRED is the average of REDs for a set of given inputs. The possibility of having RED smaller than 2% (PRED) is another important metric used in [46] and [47] for evaluating approximate radix multipliers. Considering the multiplication of two n-bit numbers, A and B, with the $\tilde{P}$ being the approximate product and P the accurate, the RED is calculated by:

$$RED_{AB} = \frac{|P - \tilde{P}|}{|P|} \tag{6.1}$$

MRED is calculated by:

$$MRED = \frac{\sum RED_{AB}}{M} \tag{6.2}$$

where $M = N - sum\_wrong\_inf - sum\_NaN\_red$. (N is the number of inputs. The metrics sum_wrong_inf and sum_NaN_red are gone to be explained in table 6.1.)

PRED is given by:

$$PRED = \frac{pos}{N} \tag{6.3}$$

where pos the total sum of REDs being smaller than 2%.

In our design the output can be:

- a real number, or
- overflow (which is represented as infinite), or
- overflow (which is represented as zero).

Table 6.1 shows 5 combinations of P, $\tilde{P}$ when RED is not computed by equation (6.1), and 2 extra metrics for some of these cases:

Table 6.1: 5 specific combinations of P, $\tilde{P}$

P	$\tilde{P}$	RED	sum_wrong_inf	sum_NaN_red
infinite	infinite	0	-	-
infinite	Not infinite (real number)	-	sum_wrong_inf + 1	-
Not infinite (real number)	infinite	-	sum_wrong_inf + 1	-
zero	zero	0	-	-
zero	Not zero (real number)	-	-	sum_NaN_red + 1
Not zero (real number)	zero	-	-	sum_NaN_red + 1

As it shown in table 6.1, the value of sum_wrong_inf is increased by 1 when P is infinite and $\tilde{P}$ is not infinite, and vice versa. The value of sum_NaN_red is increased by 1 when P is zero and $\tilde{P}$ is not zero, and vice versa. So we compute two addition metrics as follows:

$$WRONG\_INF = \frac{sum\_wrong\_inf}{N} \tag{6.4}$$

$$NaN\_red = \frac{sum\_NaN\_red}{N} \tag{6.5}$$

6.3 Evaluation at Circuit Level

This section includes the evaluation of the proposed design in terms of accuracy (error) and hardware (delay, area, power, and energy). Firstly, all the simulations were made for a 16x16 floating-point multiplier ($n = 16$). In Table 6.2 all results of the half precision floating-point multiplier, which use the Hybrid Partial Product Perforation-Rounding technique, in critical path delay are displayed. The error ranges from 0.05% to 3.33%. The gain in delay, area, energy is up to 32%, 54%, 53% respectively. The NaN_red and WRONG_INF variables are almost 0%.

In addition, all the simulations were made for a 32x32 floating-point multiplier ($n = 32$). In Table 6.3 all results of the single precision floating-point multiplier, which use the Hybrid Partial Product Perforation-Rounding technique, in critical path delay are displayed. The error ranges from 0.00% to 2.2%. The gain in delay, area, energy is up to 46%, 83%, 82% respectively. The NaN_red and WRONG_INF variables are almost 0%.

Figures 6.1 and 6.2 show the Pareto diagrams of approximation half and single precision floating-point multiplier, respectively.

As it shown in figure 6.1, all the implementations with $rounding = 8$, as well as those with $perforation = 4$ have error about 3.33 %. In $rounding = 8$, this increase in error is probably due to removing from Y very significant digits (2^{-4}). Similarly $perforation = 4$

Table 6.2: Total Results of $PR|_{k,m}$ in Critical Path Delay (binary16)

Multiplier	Delay (ns)	Power (μW)	Area (μm^2)	Energy ($\mu W * ns$)	MRED (%)	PRED (%)	NaN_red (%)	WRONG_INF (%)
CMB	0.76	3314	2388	2518.64	-	-	-	-
$PR\|_{0,2}$	0.74	3668	2341	2714.32	0.05	100.00	0.00	0.00
$PR\|_{0,4}$	0.75	3134	1875	2350.50	0.20	99.98	0.00	0.01
$PR\|_{0,6}$	0.68	3720	2044	2529.60	0.81	92.27	0.03	0.03
$PR\|_{0,8}$	0.64	3163	1607	2024.32	3.24	42.84	0.09	0.11
$PR\|_{1,0}$	0.73	3327	2214	2428.71	0.05	100.00	0.00	0.00
$PR\|_{1,2}$	0.71	3624	2166	2573.04	0.08	100.00	0.00	0.00
$PR\|_{1,4}$	0.71	2725	1792	1934.75	0.21	99.98	0.00	0.01
$PR\|_{1,6}$	0.67	2971	1740	1990.57	0.81	92.12	0.03	0.04
$PR\|_{1,8}$	0.62	2801	1492	1736.62	3.24	42.80	0.09	0.12
$PR\|_{2,0}$	0.69	3682	2205	2540.58	0.20	99.98	0.01	0.01
$PR\|_{2,2}$	0.69	2904	1779	2003.76	0.21	99.99	0.01	0.01
$PR\|_{2,4}$	0.66	3132	1792	2067.12	0.28	99.98	0.01	0.01
$PR\|_{2,6}$	0.64	3059	1661	1957.76	0.83	90.84	0.03	0.05
$PR\|_{2,8}$	0.60	2369	1301	1421.40	3.24	42.91	0.09	0.17
$PR\|_{3,0}$	0.64	2811	1583	1799.04	0.81	92.31	0.02	0.03
$PR\|_{3,2}$	0.64	3048	1583	1950.72	0.81	92.11	0.02	0.03
$PR\|_{3,4}$	0.62	2822	1481	1749.64	0.83	90.61	0.03	0.05
$PR\|_{3,6}$	0.58	2714	1415	1574.12	1.10	78.07	0.04	0.12
$PR\|_{3,8}$	0.55	2312	1177	1271.60	3.31	42.92	0.09	0.39
$PR\|_{4,0}$	0.54	2658	1303	1435.32	3.25	42.78	0.09	0.14
$PR\|_{4,2}$	0.56	2786	1390	1560.16	3.25	42.76	0.09	0.16
$PR\|_{4,4}$	0.54	2275	1167	1228.50	3.26	42.75	0.09	0.21
$PR\|_{4,6}$	0.51	2294	1083	1169.94	3.33	42.81	0.10	0.43

Table 6.3: Total Results of $PR|_{k,m}$ in Critical Path Delay (binary32)

Multiplier	Delay (ns)	Power (μW)	Area (μm^2)	Energy ($\mu W * ns$)	MRED (%)	PRED (%)	NaN_red (%)	WRONG_INF (%)
CMB	1.00	8527	8690	8527.00	-	-	-	-
$PR\|_{4,8}$	0.90	6442	5261	5797.80	0.00	100.00	0.00	0.00
$PR\|_{4,10}$	0.87	6593	5216	5735.91	0.00	100.00	0.00	0.00
$PR\|_{4,12}$	0.85	6295	4690	5350.75	0.01	100.00	0.00	0.00
$PR\|_{4,14}$	0.83	5248	3773	4355.84	0.03	100.00	0.00	0.00
$PR\|_{4,16}$	0.79	4863	3391	3841.77	0.10	100.00	0.00	0.00
$PR\|_{4,18}$	0.75	4641	3240	3480.75	0.41	100.00	0.00	0.00
$PR\|_{4,20}$	0.72	4070	2692	2930.40	1.63	60.75	0.01	0.01
$PR\|_{6,12}$	0.80	5820	3842	4656.00	0.01	100.00	0.00	0.00
$PR\|_{6,14}$	0.80	4286	3043	3428.80	0.03	100.00	0.00	0.00
$PR\|_{6,16}$	0.76	4190	2928	3184.40	0.10	100.00	0.00	0.00
$PR\|_{6,18}$	0.73	4110	2648	3000.30	0.41	100.00	0.00	0.00
$PR\|_{6,20}$	0.70	3397	2243	2377.90	1.63	60.76	0.01	0.01
$PR\|_{8,14}$	0.73	3864	2543	2820.72	0.10	100.00	0.00	0.00
$PR\|_{8,16}$	0.73	3181	2113	2322.13	0.14	100.00	0.00	0.00
$PR\|_{8,18}$	0.69	3208	2012	2213.52	0.42	99.99	0.00	0.00
$PR\|_{8,20}$	0.64	3477	2006	2225.28	1.63	60.70	0.01	0.03
$PR\|_{10,8}$	0.67	4416	2457	2958.72	1.63	61.03	0.01	0.00
$PR\|_{10,10}$	0.68	3942	2179	2680.56	1.63	61.04	0.01	0.00
$PR\|_{10,12}$	0.67	3925	2176	2629.75	1.63	61.05	0.01	0.00
$PR\|_{10,14}$	0.64	3561	1957	2279.04	1.63	61.01	0.01	0.01
$PR\|_{10,16}$	0.61	3546	1869	2163.06	1.63	61.04	0.01	0.02
$PR\|_{10,18}$	0.59	3381	1690	1994.79	1.66	60.97	0.01	0.07
$PR\|_{10,20}$	0.54	2774	1438	1497.96	2.20	55.14	0.01	0.26

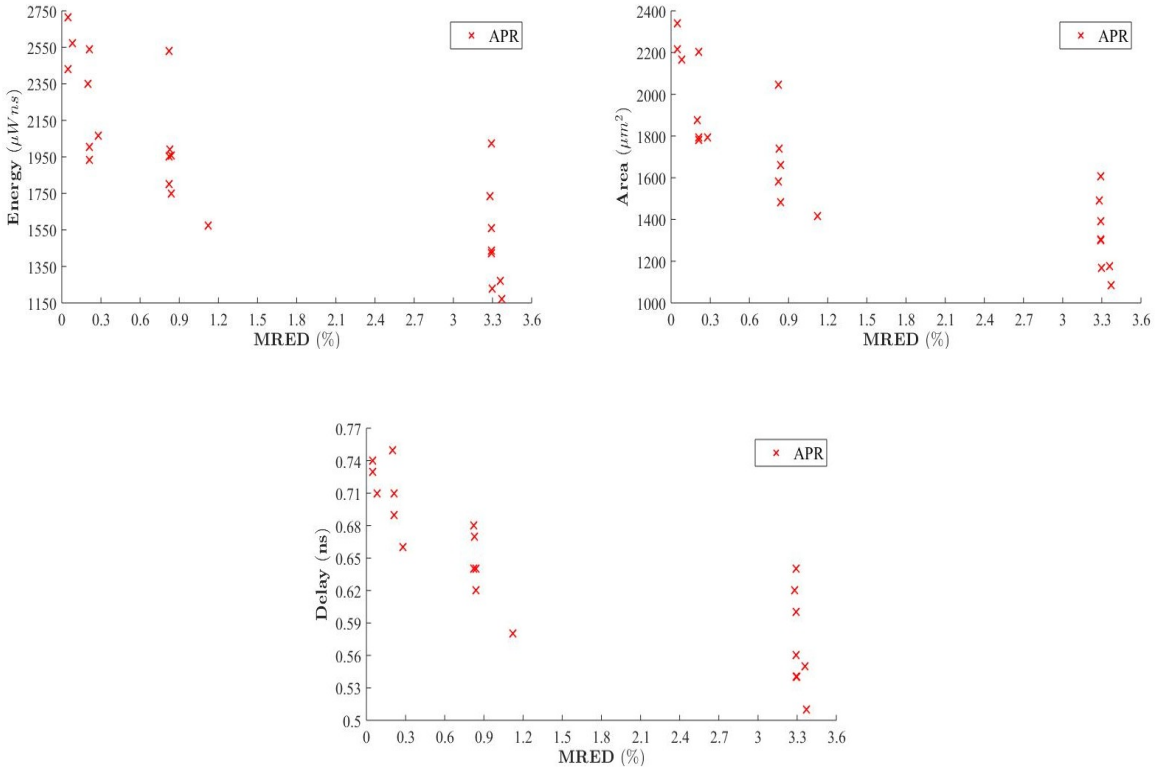


Figure 6.1: Evaluation of the proposed approximate half precision floating-point multipliers in Pareto diagrams, when synthesized and operating at their critical path delay

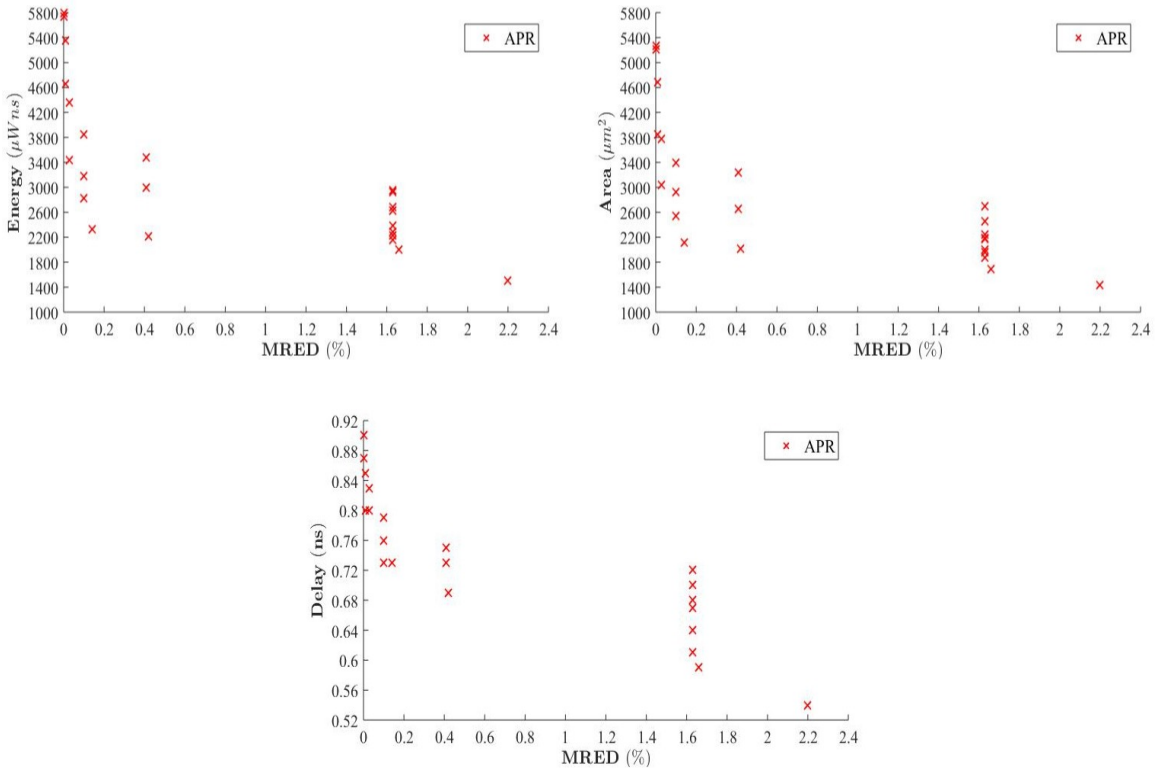


Figure 6.2: Evaluation of the proposed approximate single precision floating-point multipliers in Pareto diagrams, when synthesized and operating at their critical path delay

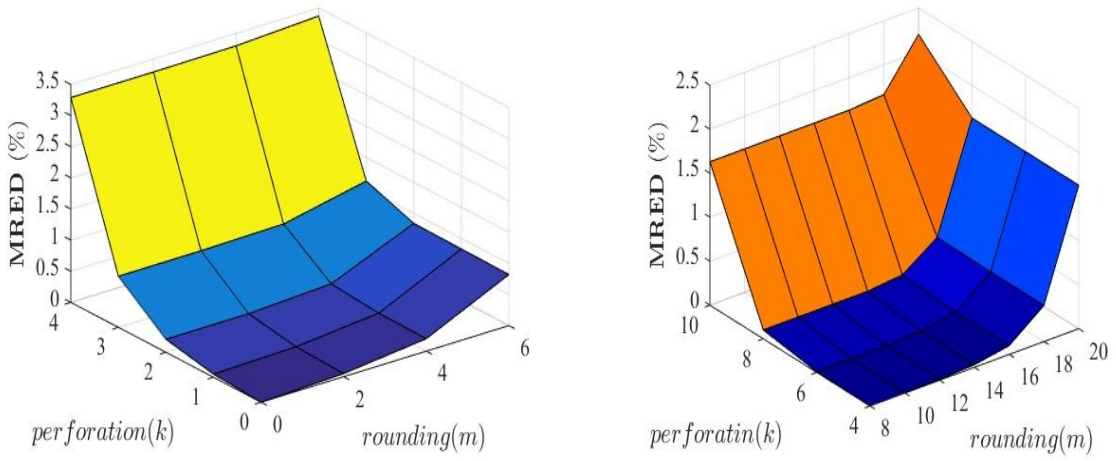


Figure 6.3: MRED variation of $PR|_{k,m}$ multiplier with respect to configuration parameters for floating-point multiplier size (left) 16 bits, and (right) 32 bits.

removes from X the corresponding digits.

As it shown in figure 6.2, all the implementations with $rounding = 20$, as well as those with $perforation = 10$ have error about 1.6 %. In $rounding = 20$, this increase in error is probably due to removing from Y very significant digits (2^{-5}). Similarly $perforation = 10$ removes from X the corresponding digits.

Figure 6.3 presents how MRED is affected by the configuration parameters k and m for floating-point multiplier size n = 16, 32. As shown, perforation introduces higher error than rounding, due to the significance of the bits that are pruned. Moreover, as the bit-width increases, MRED is less affected by the approximations. As it shown for n = 16 MRED is up to 3.33 percent, when k=4 and m=6. For n = 32 MRED is 0.0 percent, when k=4 and m=8.

Pareto frontier

For a given system, the Pareto frontier or Pareto set or Pareto front is the set of parameterizations (allocations) that are all Pareto efficient. Finding Pareto frontiers is particularly useful in engineering. By yielding all of the potentially optimal solutions, a designer can make focused tradeoffs within this constrained set of parameters, rather than needing to consider the full ranges of parameters.

Using the Pareto diagram (Energy-MRED), the approximations which lie on the frontier are chosen. These approximations are:

- $PR|_{0,2}$, $PR|_{1,0}$, $PR|_{1,4}$, $PR|_{3,4}$, $PR|_{3,6}$, and $PR|_{4,6}$, for binary16.
- $PR|_{4,12}$, $PR|_{6,12}$, $PR|_{6,14}$, $PR|_{8,16}$, $PR|_{10,18}$, and $PR|_{10,20}$, for binary32.

The gain of area and energy, of each of the above multipliers are computed and are shown in figure 6.4.

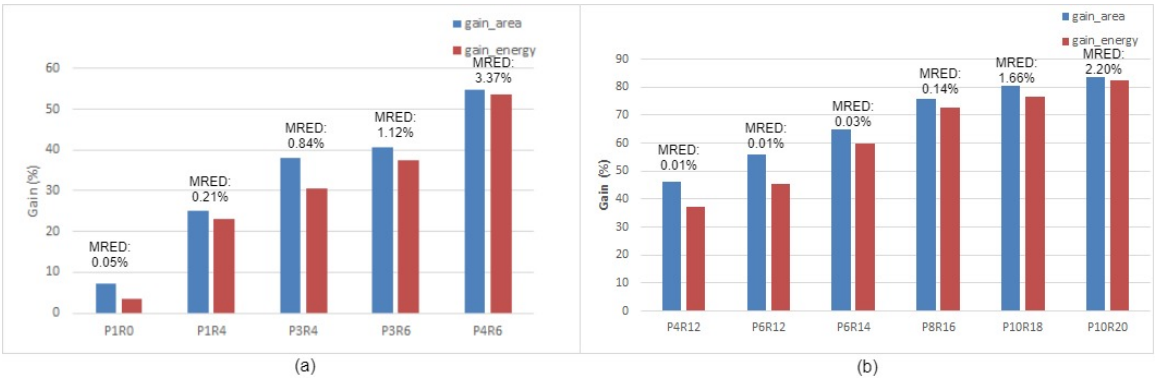


Figure 6.4: Hardware gains and accuracy results using (a) the proposed 16x16 approximate multipliers, and (b) the proposed 32x32 approximate multipliers

For 16bit multiplier MRED ranges from 0.05% to 3.37%, the gain of area ranges from 7.29% to 54.65%, and the gain of energy ranges from 3.57% to 53.55%.

For 32bit multiplier MRED ranges from 0.01% to 2.20%, the gain of area ranges from 46.03% to 83.45%, and the gain of energy ranges from 37.25% to 82.43%.

6.4 Evaluation at Application Level

In image processing, a kernel, convolution matrix, or mask is a small matrix. It is used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image. The general expression of a convolution is:

$$g(x, y) = w * f(x, y) \tag{6.6}$$

where $g(x, y)$ is the filtered image, $f(x, y)$ is the original image, w is the filter kernel.

The effectiveness of the proposed approximate floating-point multipliers was assessed by using them in an image blurring (Gaussian blur) application. The image blurring output is determined using the following equation:

$$Y(i+1, j+1) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} X(i+1+m, j+1+n) * G_{blur}(m+2, n+2) \tag{6.7}$$

where $X(i, j)$ denote the pixel of the i-th row and the j-th column of input, and $Y(i+1, j+1)$ denote the pixel of the $(i+1)$-th row and the $(j+1)$-th column of output.

G_{blur} is given by:

Figure 6.5: Image blurring using the proposed approximate multipliers. (a) Original image (Lena). Image blurring using (b) the accurate multiplier, (c) the $PR|_{0,2}$ design, (d) the $PR|_{1,0}$ design, (e) the $PR|_{1,4}$ design, (f) the $PR|_{3,4}$ design, (g) the $PR|_{3,6}$ design, and (h) the $PR|_{4,6}$ design

$$G_{blur} = \frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{6.8}$$

Two different grayscale test images, Lena and cameraman, where used for the image blurring application.

- 16x16 bit floating-point multiplier

  Figure 6.5 presents the original image of Lena (Figure 6.5(a)), and the output images produced by image blurring using an accurate 16x16 bit floating-point multiplier (Figure 6.5(b)) and the proposed approximate multipliers (Figure 6.5(c)–(h)).

  Figure 6.6 presents the original image of cameraman (Figure 6.6(a)), and the output images produced by image blurring using an accurate 16x16 bit floating-point multiplier (Figure 6.6(b)) and the proposed approximate multipliers (Figure 6.6(c)–(h)).

- 32x32 bit floating-point multiplier

  Figure 6.7 presents the original image of Lena (Figure 6.7(a)), and the output images produced by image blurring using an accurate 32x32 bit floating-point multiplier (Figure 6.7(b)) and the proposed approximate multipliers (Figure 6.7(c)–(h)).

  Figure 6.8 presents the original image of cameraman (Figure 6.8(a)), and the output images produced by image blurring using an accurate 32x32 bit floating-point multiplier (Figure 6.8(b)) and the proposed approximate multipliers (Figure 6.8(c)–(h)).

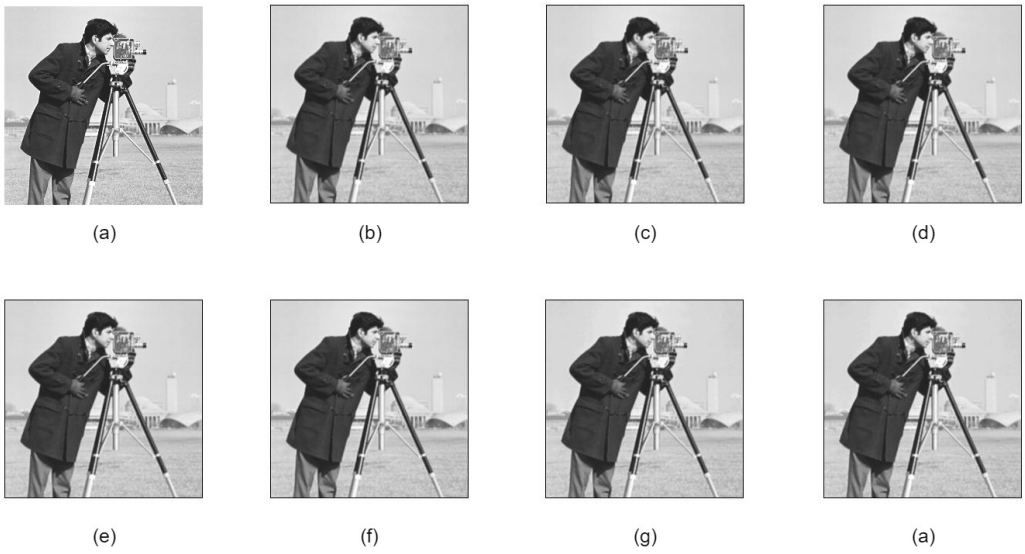


Figure 6.6: Image blurring using the proposed approximate multipliers. (a) Original image (cameraman). Image blurring using (b) the accurate multiplier, (c) the $PR|_{0,2}$ design, (d) the $PR|_{1,0}$ design, (e) the $PR|_{1,4}$ design, (f) the $PR|_{3,4}$ design, (g) the $PR|_{3,6}$ design, and (h) the $PR|_{4,6}$ design

Figure 6.7: Image blurring using the proposed approximate multipliers. (a) Original image (Lena). Image blurring using (b) the accurate multiplier, (c) the $PR|_{4,12}$ design, (d) the $PR|_{6,12}$ design, (e) the $PR|_{6,14}$ design, (f) the $PR|_{8,16}$ design, (g) the $PR|_{10,18}$ design, and (h) the $PR|_{10,20}$ design

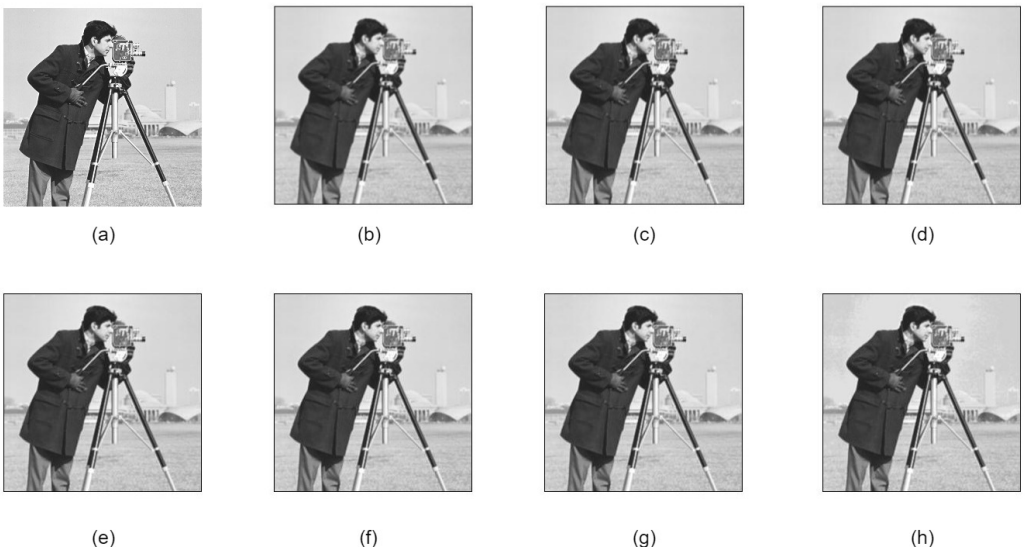


Figure 6.8: Image blurring using the proposed approximate multipliers. (a) Original image (cameraman). Image blurring using (b) the accurate multiplier, (c) the $PR|_{4,12}$ design, (d) the $PR|_{6,12}$ design, (e) the $PR|_{6,14}$ design, (f) the $PR|_{8,16}$ design, (g) the $PR|_{10,18}$ design, and (h) the $PR|_{10,20}$ design

The quality of the blurring process may not be easily assessed by human eyes. To measure the quality of the approximate output images, two metrics, the Peak Signal-to-Noise Ratio (PSNR) and the Structural Similarity Index (SSIM), were used [48]. PSNR is calculated as follows:

$$PSNR = 10 * log_{10}\left(\frac{{MAX_I}^2}{MSE}\right) \tag{6.9}$$

PSNR is most easily defined via the mean squared error (MSE). Given a noise-free m×n monochrome image I and its noisy approximation K, MSE is defined as:

$$MSE = \frac{1}{m * n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I(i,j) - K(i,j))^2 \tag{6.10}$$

Tables 6.4 and 6.5 display the PSNR (in dB) and SSIM results of different approximate multipliers 16x16 bit utilized for image blurring on Lena and cameraman, respectively.

Tables 6.6 and 6.7 display the PSNR (in dB) and SSIM results of different approximate multipliers 32x33 bit utilized for image blurring on Lena and cameraman, respectively.

The approximations which have $SSIM = 1$ showed the best performance as that of exact multiplication. The values of SSIM which are over to 0.8 but they are not 1 correspond to good approximations as well.

Table 6.4: PSNR and SSIM values of the outputs of image blurring (Lena), using different approximate multiplier designs (16x16 bit)

Lena		
Multiplier	PSNR (dB)	SSIM
$PR\|_{0,2}$	∞	1
$PR\|_{1,0}$	∞	1
$PR\|_{1,4}$	59.22	0.99
$PR\|_{3,4}$	59.22	0.99
$PR\|_{3,6}$	50.86	0.98
$PR\|_{4,6}$	50.86	0.98

Table 6.5: PSNR and SSIM values of the outputs of image blurring (cameraman), using different approximate multiplier designs (16x16 bit)

Cameraman		
Multiplier	PSNR (dB)	SSIM
$PR\|_{0,2}$	∞	1
$PR\|_{1,0}$	∞	1
$PR\|_{1,4}$	55.17	0.99
$PR\|_{3,4}$	55.17	0.99
$PR\|_{3,6}$	48.20	0.87
$PR\|_{4,6}$	48.20	0.87

Table 6.6: PSNR and SSIM values of the outputs of image blurring (Lena), using different approximate multiplier designs (32x32 bit)

Lena		
Multiplier	PSNR (dB)	SSIM
$PR\|_{4,12}$	∞	1
$PR\|_{6,12}$	∞	1
$PR\|_{6,14}$	∞	1
$PR\|_{8,16}$	∞	1
$PR\|_{10,18}$	54.46	0.99
$PR\|_{10,20}$	44.46	0.95

Table 6.7: PSNR and SSIM values of the outputs of image blurring (cameraman), using different approximate multiplier designs (32x32 bit)

Cameraman		
Multiplier	PSNR (dB)	SSIM
$PR\|_{4,12}$	∞	1
$PR\|_{6,12}$	∞	1
$PR\|_{6,14}$	∞	1
$PR\|_{8,16}$	∞	1
$PR\|_{10,18}$	52.99	0.95
$PR\|_{10,20}$	40.66	0.80

Chapter 7

Conclusion

In this diploma thesis, inexact half and single precision FP multiplier designs have been proposed. All the multipliers are implemented in Verilog HDL, synthesized using Synopsys Design Compiler and the TSMC 65-nm standard cell library, and simulated with Mentor Graphics ModelSim. The results from evaluation at circuit level shown that the proposed approximate FP multiplier reduces delay, area and energy up to 32%, 54%, and 53% respectively compared with the exact multiplier for 16bit, while incurring in an error less than 3.4%. Also, the proposed approximate FP multiplier reduces delay, area and energy up to 46%, 83%, and 82% respectively for 32bit, while incurring in an error less than 2.2%. Furthermore, the effectiveness of the approximate multipliers was assessed in an image blurring application.

Bibliography

[1] IEEE 754-2008, ieee standard for floating-point arithmetic. 2008.

[2] Ali Farmani. High performance hardware design of ieee floating point adder in fpga with vhdl.

[3] Ali Malik and Seok-Bum Ko. A study on the floating-point adder in fpgas. In *2006 Canadian Conference on Electrical and Computer Engineering*, pages 86–89. IEEE, 2006.

[4] Ali Malik and Seok-Bum Ko. Effective implementation of floating-point adder using pipelined lop in fpgas. In *Canadian Conference on Electrical and Computer Engineering, 2005.*, pages 706–709. IEEE, 2005.

[5] K Deergha Rao, PV Muralikrishna, and Ch Gangadhar. Fpga implementation of 32 bit complex floating point multiplier using vedic real multipliers with minimum path delay. In *2018 5th IEEE Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)*, pages 1–6. IEEE, 2018.

[6] KV Gowreesrinivas and P Samundiswary. Comparative performance analysis of multiplexer based single precision floating point multipliers. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 430–435. IEEE, 2017.

[7] Y Srinivasa Rao, M Kamaraju, and DVS Ramanjaneyulu. An fpga implementation of high speed and area efficient double-precision floating point multiplier using urdhva tiryagbhyam technique. In *2015 Conference on Power, Control, Communication and Computational Technologies for Sustainable Growth (PCCCTSG)*, pages 271–276. IEEE, 2015.

[8] Alexander Aponte-Moreno, Alejandro Moncada, Felipe Restrepo-Calle, and Cesar Pedraza. A review of approximate computing techniques towards fault mitigation in hw/sw systems. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6. IEEE, 2018.

[9] V. Leon, G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi. Walking through the energy-error pareto frontier of approximate multipliers. *IEEE Micro*, 38(4):40–49, Jul-Aug 2018.

[10] V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi. Energy-efficient VLSI implementation of multipliers with double LSB operands. *IET Circuits Devices Syst.*, to be published, doi: 10.1049/iet-cds.2018.5039, 2019.

[11] Yuke Wang, C Pai, and Xiaoyu Song. The design of hybrid carry-lookahead/carry-select adders. *IEEE Transactions on circuits and systems II: Analog and Digital Signal Processing*, 49(1):16–24, 2002.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[13] P-M Seidel and Guy Even. Delay-optimized implementation of ieee floating-point addition. *IEEE Transactions on computers*, 53(2):97–113, 2004.

[14] Javier D Bruguera and Tomas Lang. Leading-one prediction with concurrent position correction. *IEEE Transactions on Computers*, 48(10):1083–1097, 1999.

[15] Stuart F Oberman, Hesham Al-Twaijry, and Michael J Flynn. The snap project: Design of floating point arithmetic units. In *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*, pages 156–165. IEEE, 1997.

[16] Nhon Quach and Michael J Flynn. *Leading one prediction-implementation, generalization, and application.* Computer Systems Laboratory, Stanford University, 1991.

[17] Erdem Hokenek and Robert K. Montoye. Leading-zero anticipator (lza) in the ibm risc system/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):71–77, 1990.

[18] Model technology, available at:. `http://www.model.com/products/60/se.asp`.

[19] Xilinx design tools center: Free ise webpack 6.3i, available at:. `http://www.xilinx.com/`.

[20] Stuart F Oberman and MJ Flynn. *Advanced computer arithmetic design.* J. Wiley, 2001.

[21] Taciano A Rodolfo, Ney LV Calazans, and Fernando G Moraes. Floating point hardware for embedded processors in fpgas: Design space exploration for performance and area. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 24–29. IEEE, 2009.

[22] Honey Durga Tiwari, Ganzorig Gankhuyag, Chan Mo Kim, and Yong Beom Cho. Multiplier design based on ancient indian vedic mathematics. In *2008 International SoC Design Conference*, volume 2, pages II–65. IEEE, 2008.

[23] Ravi Kishore Kodali, Satya Kesav Gundabathula, and Lakshmi Boppana. Fpga implementation of ieee-754 floating point karatsuba multiplier. In *2014 International*

Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), pages 300–304. IEEE, 2014.

[24] Sandesh S Saokar, RM Banakar, and Saroja Siddamal. High speed signed multiplier for digital signal processing applications. In *2012 IEEE International Conference on Signal Processing, Computing and Control*, pages 1–6. IEEE, 2012.

[25] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[26] Armin Alaghi and John P Hayes. Strauss: Spectral transform use in stochastic circuit synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1770–1783, 2015.

[27] Martin Van Leussen, Jos Huisken, Lei Wang, Hailong Jiao, and José Pineda de Gyvez. Reconfigurable support vector machine classifier with approximate computing. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 13–18. IEEE, 2017.

[28] V. Leon, G. Zervakis, D. Soudris, and K. Pekmestzi. Approximate hybrid high radix encoding for energy-efficient inexact multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):421–430, March 2018.

[29] Vasileios Leon, Konstantinos Asimakopoulos, Sotirios Xydis, Dimitrios Soudris, and Kiamal Pekmestzi. Cooperative arithmetic-aware approximation techniques for energy-efficient multipliers. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 160:1–160:6, New York, NY, USA, 2019. ACM.

[30] Xin He, Guihai Yan, Yinhe Han, and Xiaowei Li. Acr: Enabling computation reuse for approximate computing. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 643–648. IEEE, 2016.

[31] Lawrence McAfee and Kunle Olukotun. Emeuro: A framework for generating multi-purpose accelerators via deep learning. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 125–135. IEEE, 2015.

[32] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 63–68. IEEE, 2017.

[33] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test*, 33(1):8–22, 2015.

[34] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[35] Mingze Gao and Gang Qu. A novel approximate computing based security primitive for the internet of things. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[36] Tor M Aamodt and Paul Chow. Compile-time and instruction-set methods for improving floating-to fixed-point conversion accuracy. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):26, 2008.

[37] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with relaxed synchronization. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 41–50. ACM, 2012.

[38] Vinay K Chippa, Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing: An integrated hardware approach. In *2013 Asilomar conference on signals, systems and computers*, pages 111–117. IEEE, 2013.

[39] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, volume 47, pages 301–312. ACM, 2012.

[40] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[41] Yongtae Kim, Yong Zhang, and Peng Li. An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–137. IEEE Press, 2013.

[42] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. On reconfiguration-oriented approximate adder design and its application. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 48–54. IEEE, 2013.

[43] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *2011 24th Internatioal Conference on VLSI Design*, pages 346–351. IEEE, 2011.

[44] Georgios Zervakis, Kostas Tsoumanis, Sotirios Xydis, Dimitrios Soudris, and Kiamal Pekmestzi. Design-efficient approximate multiplication circuits through partial product perforation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(10):3105–3117, 2016.

[45] Jinghang Liang, Jie Han, and Fabrizio Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on computers*, 62(9):1760–1771, 2012.

[46] Honglan Jiang, Jie Han, Fei Qiao, and Fabrizio Lombardi. Approximate radix-8 booth multipliers for low-power and high-performance operation. *IEEE Transactions on Computers*, 65(8):2638–2644, 2015.

[47] Weiqiang Liu, Liangyu Qian, Chenghua Wang, Honglan Jiang, Jie Han, and Fabrizio Lombardi. Design of approximate radix-4 booth multipliers for error-tolerant computing. *IEEE Transactions on Computers*, 66(8):1435–1441, 2017.

[48] Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.