Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής και Συστημάτων Πληροφορικής
Εργαστήριο Σχεδίασης Ψηφιακών Συστημάτων

Harnessing CIM techniques for accelerating sum operations in FPGA-DRAM architectures.

Διπλωματική Εργασία

του

Διαμαντίδη Θεοχάρη

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής και Συστημάτων Πληροφορικής
Εργαστήριο Σχεδίασης Ψηφιακών Συστημάτων

Harnessing CIM techniques for accelerating sum operations in FPGA-DRAM architectures.

Διπλωματική Εργασία

του

Διαμαντίδη Θεοχάρη

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Μαρτίου 2025:

.....	.....	.....
Δημήτριος	Σωτήριος	Γεώργιος
Σούντρης	Ξύδης	Λεντάρης
Καθηγητής	Καθηγητής	Καθηγητής
Ε.Μ.Π.	Ε.Μ.Π.	ΠΑ.Δ.Α.

Αθήνα, Μάρτιος 2025

.....

Διαμαντίδης Θεοχάρης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών, Ε.Μ.Π.

Περίληψη

Στα πλαίσια της παρούσας εργασίας παρουσιάζεται ο τρόπος με το οποίο μπορούμε να αξιοποιήσουμε το ήδη υπάρχον (hardware) το οποίο διαθέτουν οι μνήμες τυχαίας προσέλασης (DRAM) με σκοπό να εκτελέσουμε πράξεις μέσα στην ίδια την μνήμη. Η μεταφορά των δεδομένων από τον επεξεργαστή στην κύρια μνήμη και τανάπαλιν αποτελεί ένα από τα σημαντικότερα εμπόδια που αντιμετωπίζουν τα σύγχρονα υπολογιστικά συστήματα. Σε αυτή την διπλωματική εργασία εξετάζουμε τρόπους με τους οποίους μπορούμε να εκτελέσουμε πράξει σε επίπεδο (bit) χρησιμοποιώντας αποκλειστικά τις αναλογικές ιδότητες που έχουν οι πίνακες μνήμης. Συγκεκριμένα εξετάζουμε τις πράξεις δυαδικής λογικής που μπορούν να πραγματοποιηθούν ανάμεσα σε περισσότερα του ενός (bit) όπως για παράδειγμα είναι οι πύλες (AND, OR, NOT). Έπειτα συνδυάζοντας αυτές τις πύλες κατασκευάζουμε μια λειτουργική μονάδα πλήρους αθροιστή (Full Adder) η οποία πραγαμτοποιεί την δυαδική πρόσθεση αποκλειστικά μέσα στην μνήμη. Δείχνουμε ότι μπορούμε να εκτελέσουμε τις απλές δυαδικές εντολές με ποσοστό επιιχίας μέχρι και 90% ενώ τα αποτελέσματα της δυαδκής πρόσθεσης λαμβάνονται με επιτυχία μέχρι και 80%. Παρουσιάζεται η κυκλωματική υλοποίηση ενισχυτή ανίχνευσης σήματος που τεκμηριώνει τον τρόπο λειτουργίας της παραπάνω διαδικάσιας που υλοποιείται χρησιμοποιώντας το Cadence IC Suite σε τεχνολογία TSMC 90nm CMOS process.

Λέξεις Κλειδιά: Δυναμική Μνήμη Τυχαίας Προσπέλασης, Πλήρης Αθροιστής, Ενισχυτής ανίχνευσης σήματος, Πράξεις με μνήμη, Λογικές Πύλες

Abstract

In the context of this work, we present a method for leveraging the existing hardware capabilities of random access memory DRAM to perform operations directly within the memory itself. The transfer of data between the processor and main memory, and vice versa, constitutes one of the most significant bottlenecks in modern computing systems. In this thesis, we explore methods for performing bit-level operations exclusively using the analog properties inherent in memory arrays. Specifically, we examine binary logic operations that can be performed between multiple bits, such as the AND, OR, NOT gates. Subsequently, by combining these gates, we construct a functional full adder unit (Full Adder) that performs binary addition entirely within the memory. We demonstrate that simple binary operations can be executed with a success rate of up to 90%, while the results of binary addition achieve success rates of up to 80%. Additionally, we present the circuit implementation of a signal detection amplifier that validates the operation of this process. The implementation is carried out using the Cadence IC Suite on TSMC 90nm CMOS process technology.

Keywords: Dynamic Random Access Memory (DRAM), Full Adder, Signal Detection Amplifier, In-Memory Computing, Logic Gates.

Ευχαριστίες

Με την παράδοση της παρούσας διπλωματικής εργασίας ολοκληρώνεται ο κύκλος των προπτυχιακών μου σπουδών. Στο σημείο αυτό θα ήθελα να ευχαριστήσω αρχικά τον επιβλέποντα καθηγητή μου κ. Δημήτριο Σούντρη για την καθοδήγηση που μου παρείχε καθ΄ όλη την διάρκεια εκπόνησης της παρούσας εργασίας. Το θέμα αυτό επιλέχθηκε με δική του πρωτοβουλία και ήταν αυτό που ταίριαζε με τις γνώσεις και τα ενδιαφέροντά μου. Θα ήθελα επίσης να ευχαριστήσω τον κ. Γεώργιο Λεντάρη (Καθηγητή ΠΑΔΑ) για την πολύτιμη βοήθειά του, καθώς και για τις λύσεις που πρότεινε στα προβλήματα που εμφανίστηκαν κατά την διάρκεια της τεχνικής υλοποίησης της διπλωματικής.

Επίσης θα ήθελα να ευχαριστήσω θερμά την ομάδα του κ.Onur Mutlu (ETH) και ειδικότερα τους Olgun Ataberk και Ismail Emir Yüksel για την πολύ σημαντική συνεισφορά τους και τον χρόνο που αφιέρωσαν καθώς χωρίς την αρωγή τους δεν θα είχα την υποδομή για τον έλεγχο των αποτελεσμάτων που προέκυψαν από τις προσομοιώσεις.

Τέλος θα ήθελα να ευχαριστήσω ιδιαιτέρως την οικογένειά μου και τα αδέρφια μου για την υποστήριξή τους, όλους εκείνους που με βοήθησαν κατά την διάρκεια της φοίτησής μου, καθώς επίσης και τους φίλους μου για τις όμορφες στιγμές που περάσαμε αυτά τα 5 χρόνια σπουδών.

Διαμαντίδης Θεοχάρης
Μάρτιος 2025

Acknowledgments

With the submission of this thesis, the cycle of my undergraduate studies comes to an end. At this point, I would like to express my gratitude. First and foremost, I would like to thank my supervisor, Professor Dimitrios Soudris, for his guidance throughout the entire duration of this thesis. The topic was chosen on his initiative and was perfectly aligned with my knowledge and interests. I would also like to thank Professor Georgios Lentaris (University of West Athens) for his valuable assistance and for the solutions he proposed to the challenges that arose during the technical implementation of the thesis.

Furthermore, I would like to express my sincere gratitude to the team of Professor Onur Mutlu (ETH), and in particular to Olgun Ataberk and Ismail Emir Yüksel, for their significant contributions and the time they dedicated. Without their support, I would not have had the infrastructure necessary for verifying the results obtained from the simulations.

Finally, I would like to extend my heartfelt thanks to my family and siblings for their unwavering support, as well as to everyone who helped me during my studies. I am also grateful to my friends for the wonderful moments we shared throughout these five years of study.

Diamantidis Theocharis
March 2025

Contents

List of Figures

List of Tables

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Ένα σύστημα υπολογιστή απαιτεί μνήμη για την αποθήκευση των δεδομένων του και των οδηγιών των προγραμμάτων που εκτελεί. Μέσα σε ένα σύστημα υπολογιστή υπάρχουν διάφοροι τύποι μνήμης που χρησιμοποιούν διάφορες τεχνολογίες και έχουν διαφορετικούς χρόνους προσπέλασης. Η μνήμη που χρησιμοποιούν οι υπολογιστές μπορεί να ταξινομηθεί σε δύο τύπους, την κύρια μνήμη και αποθηκευτική μνήμη. Η κύρια μνήμη είναι αυτή που έχει ταχύτερους χρόνους προσπέλασης και εκτελούνται οι περισσότερες εντολές των προγραμμάτων. Η κύρια μνήμη αυτή αποκαλείται μνήμη τυχαίας προσπέλασης (Random Access Memory, RAM). Οι χρόνοι που απιτούνται στις μήμες αυτού του είδους για την αποθήκευση και την ανάκληση των πληροφοριών είναι ανεξάρτητοι από την φυσική θέση στην οποία είναι αποθηκευμένη η πληροφορία.

Αντίθετα με τις μνήμες τυχαίας προσπέλασης, στις σειριακές ή ακολουθιακές μνήμες, όπως για παράδειγμα είναι οι δίσκοι, τα δεδομένα είναι διαθέσιμα μόνο κατά την ίδια ακολουθία με την οποία αποθηκεύτηκαν αρχικά. Συνεπώς, ο χρόνος που απαιτείται για την προσπέλαση μιας συγκεκριμένης πληροφορίας εξαρτάται από την θέση αποθήκευσης της στην μνήμη και έχει μεγαλύτερο χρόνο από αυτήν της μνήμης τυχαίας προσπέλασης. Η κανονικότητα που επιδικνύει η δομή των κυκλωμάτων μνήμης τα καθιστά ιδανική εφαρμογή για τις τεχνολογίες VLSI[3]. Συχνά οι διατάξεις μνήμης καταλαμβάνουν την πλειονότητα των τρανζίστορ σε ένα σύστημα σε ψηφίδα (system-on-chip τεχνολογίας CMOS).

1.2 Θεωρητικό Υπόβαθρο

Σε ένα ολοκληρωμένο κύκλωμα μνήμης τα bits είναι προσπελάσιμα είτε ατομικά, είτε ανά ομάδες των 4 έως 16. Στην παρούσα υλοποίηση η μνήμη ήταν προσπελάσιμη κατά ομάδες των 8 bits. Μια διάταξη μνήμης περιλαμβάνει 2^n λέξεις (words των 2^m βιτς έκαστη). Κάθε bit αποθηκεύεται σε ένα κύτταρο μνήμης, όπου στην περίπτωση των δυναμικών μνημών τυχαίας προσπέλασης είναι ένας πυκνωτής. Κατά την διάρκεια της λειτουργίας εγγραφής ή ανάγνωσης ενεργοποιούνται τα κύτταρα σε αυτήν την γραμμή λέξη και μέσα απο απομωνωτές ή κάποιους στοιχειώδεις ενισχυτές, διβάζονται ή εγγράφονται τα δεδομένα. Μια τυπική διάταξη μνήμης μπορεί να έχει χιλιάδες γραμμές εκ των οποίων κάθε γραμμή περιλαμβάνει χιλιάδες στήλες. Στα πλαίσια των πειραματικών μετρήσεων αξιολογήθηκαν μνήμες, οι οποίες περιλάμβαναν 65×1024 γραμμές εκ των οποίων κάθε γραμμή περιλάμβανε 8×1024 λέξεις των 8 bits. Σε απλούστευση, η οργάνωση της μνήμης παρουσιάζεται στο παρακάτω σχήμα και στην επόμενη παράγραφο αναλύονται εκτενέστερα τα περιφεριακά κυκλώματα που χρησιμοποιούνται.[4]

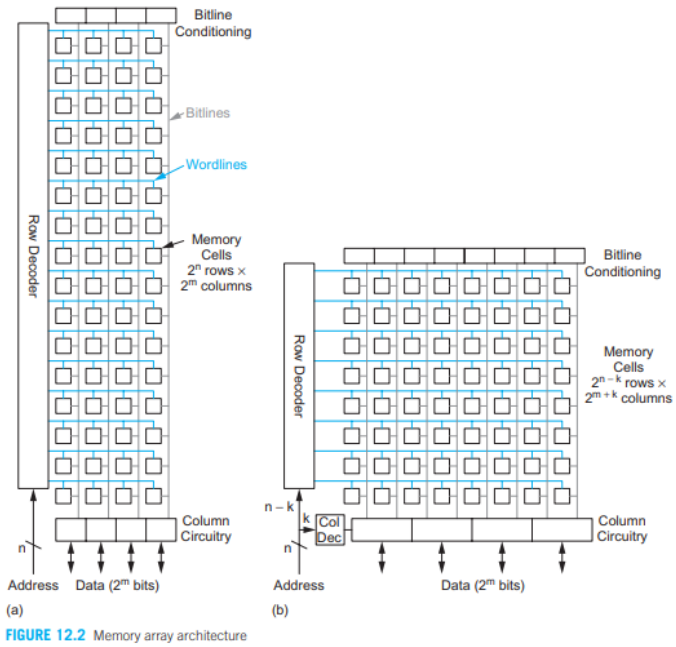
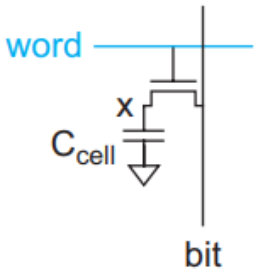


FIGURE 12.2 Memory array architecture

Σχήμα 1.1: Αρχιτεκτονική Διάταξη Μνήμης.

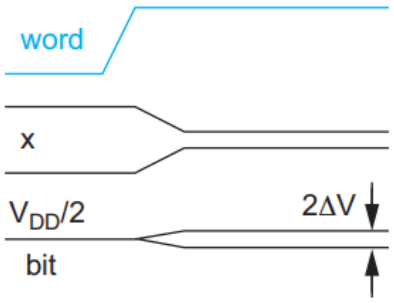
1.2.1 Η οργάνωση ενός ολοκληρωμένου κυκλώματος μνήμης DRAM

Οι δυναμικές μνήμες αποθηκεύουν τα περιεχόμενα τους με την μορφή φορτίου σε έναν πυκνωτή. Αυτό σημαίνει ότι το βασικό κύτταρο είναι πολύ μικρό και έτσι μπορούμε να κατασκευάσουμε πολύ πυκνές διάταξεις μνημών δυναμικής προσπέλασης. Το μειονέκτημα είναι ότι το περιεχόμενο των πυκνωτών θα πρέπει να διαβάζεται συχνά και να ανανεώνεται, επειδή υπάρχουν διαρροές. Ένα κύτταρο δυναμικής μνήμης αποτελείται από ένα τρανζίστορ και από εναν πυκνωτή όπως φαίνεται στο παρακάτω σχήμα.



Σχήμα 1.2: 1T κύτταρο μνήμης DRAM.

Το κύτταρο προσπελάζεται θέτοντας την γραμμή λέξης ίση με την τροφοδοσία έτσι ώστε το device να αρχίσει να άγει εισάγοντας μια μικρή αντίσταση (το τρανζίστορ θα βρίσκεται στην περιοχή τριόδου). Με αυτόν τον τρόπο ο πυκνωτής θα συνδεθεί στην γραμμή bit η οποία είναι αρχικά φορτισμένη στο μέσο της τροφοδοσίας Vdd/2. Στην συνέχεια ο πυκνωτής μοιράζεται το φορτίο με την γραμμή, προκαλώντας μια μεταβολή τάσης ΔV με αποτέλεσμα στον κόμβο της γραμμής βιτ η τάση να μεταβάλλεται όπως εικονίζεται παρακάτω ανάλογα με το αν ο πυκνωτής είναι αρχικά φορτισμένος ή όχι. Πιο συγκεκριμένα αν είναι αρχικά φορτισμένος, τότε επειδή η τάση στον κόμβο θα αρχίσει να αυξάνεται καθως τα φορτία θα "ρέουν" από τον πυκνωτή που αποθηκεύει την πληροφορία προς την χωρητικότητα που παρουσιάζει η γραμμή μεταφοράς. Αντίθετα, αν ο πυκνωτής είναι αρχικά αφόρτιστος, τότε φορτία από την παρασιτική χωρητικότητα της γραμμής μεταφοράς θα "ρέουν" προς τον πυκνωτή με αποτέλεσμα η τάση στον κόμβο να αρχίσει να μειώνεται[4].

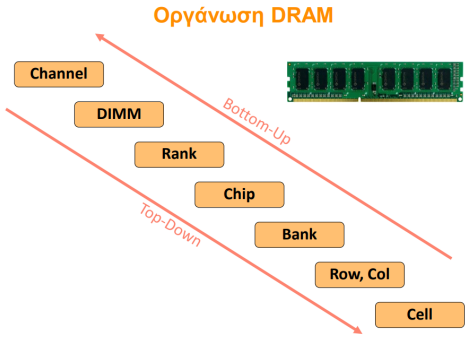


Σχήμα 1.3: Τάση στον κόμβο της στήλης bit.

Με αυτόν τον τρόπο μπορούμε να διαβάσουμε το περιεχόμενο του πυκνωτή διαταρράσοντας όμως την τάση στον κόμβο x, όπως φαίνεται παραπάνω. Αυτό σημαίνει ότι το κύτταρο πρέπει να επανεγγραφεί μετά από κάθε ανάγνωση.

1.2.2 Η αρχιτεκτονική της μνήμης σε υψηλότερο επίπεδο

Η οργάνωση της μνήμης DRAM σε αφηρημένο επίπεδο φαίνεται στην παρακάτω εικόνα.

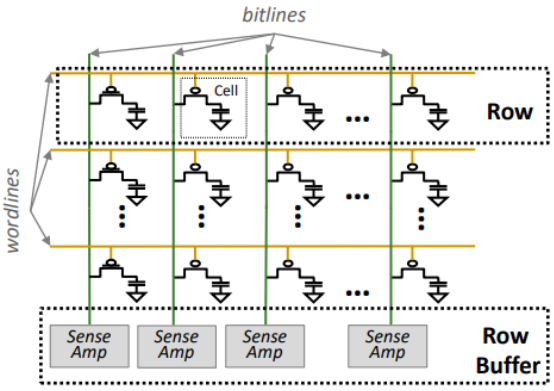


Σχήμα 1.4: Οργάνωση της μνήμης DRAM.

Στο χαμηλότερο επίπεδο (όπως εξηγήθηκε στην προηγούμενη παράγραφο) βρίσκεται η κυψελίδα με τον πυκνωτή, ο οποίος αποθηκεύει την εκάστοτε λογική τιμή με την μορφή φορτίου. Επίσης έχουμε το τρανζίστορ προσπέλασης που συνδέει και αποσυνδέει τον πυκνωτή από την γραμμή μεταφοράς.

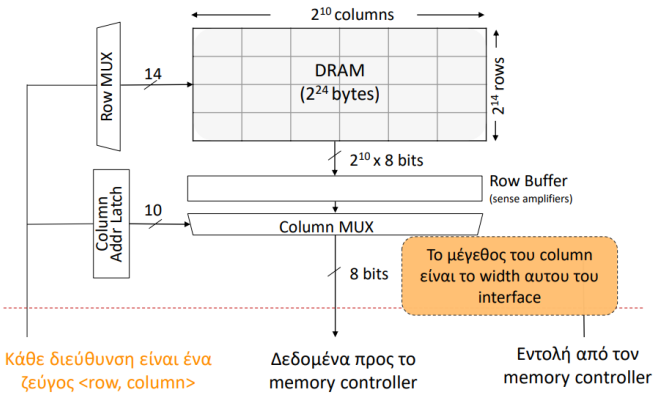
Συνδυάζοντας έναν πίνακα κυψελίδων σχηματίζεται η δομή που φαίνεται στο επόμενο σχήμα. Κάθε γραμμή έχει ένα κοινό καλώδιο wordline που ενεργοποιεί τα τρανζίστορ προσπέλασης. Το αποτέλεσμα περνάει υπό μορφή τάσης στις γραμμές μεταφοράς bitlines και κάθε βιτλινε συνδέεται με εναν ενισχυτή, ο οποίος δύναται να ενισχύει και τις μικρές μεταβολές τάσεις που παρατηρεί στην

είσοδό του. Η παρακάτω δομή διαμορφώνει το λεγόμενο subbaray και σε κάθε subarray μόνο μια γραμμή δύναται να είναι ανοιχτή κάθε φορά.



Σχήμα 1.5: DRAM subarray.

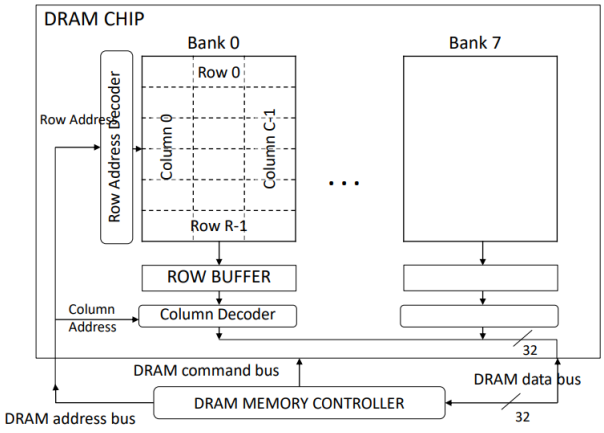
Αν οργανώσουμε τις στήλες σε ομάδες των 8bits, τότε σχηματίζεται μια δομή όπου έχουμε γραμμές και στήλες. Η ενεργοποίηση των γραμμών γίνεται από τον αποκωδικοποιητή διεύθυνσης γραμμής (row decoder), το οποίο επιλέγει αυξάνοντας την τάση κάθε γραμμής. Έτσι στην έξοδο του subarray επιλέγονται όλες οι λέξεις της κάθε γραμμής και στην συνέχεια με έναν αποκωδικοποιητή διεύθυνσης στήλης, προκύπτει η λέξη που θέλουμε πραγματικά να διαβάσουμε.



Σχήμα 1.6: Rows and Columns of a Subarray.

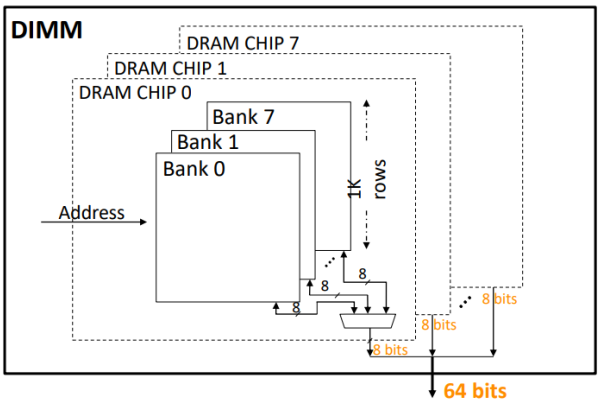
Οι στήλες και οι γραμμές αυτές των subarrays αν επαναληφθούν σε μεγάλο βαθμό θα μπορούσαν να σχηματίσουν την μνήμη. Ωστόσο, έτσι θα δημιουργούταν ένας μεγάλος μονολοθικός πίνακας, ο οποίος θα είχε μεγάλο

χρόνο απόκρισης και δεν θα επέτρεπε παράλληλες προσπελάσεις στην μνήμη. Για αυτόν τον λόγο σχηματίζουμε τα λεγόμενα banks της μνήμης, τα οποία είναι N φορές μικρότερα σε μέγεθος από την συνολική μνήμη που θέλουμε. Προσπελάσεις σε διαφορετικά banks μπορούν να επικαλύπτονται και bits από την διεύθυνση καθορίζουν ποιο bank θα ενεργοποιηθεί. Συνδυάζοντας τα πολλαπλά banks προκύπτει το DRAM Chip το οποίο αποτελείται συνήθως από 8banks. Στο κάθε chip τα banks διαμοιράζονται κοινούς διαύλους δεδομένων, εντολών, διευθύνσεων. Δηλαδή γραμμές διαφορετικών banks ενεργοποιούνται ταυτόχρονα μέσα στο chip.



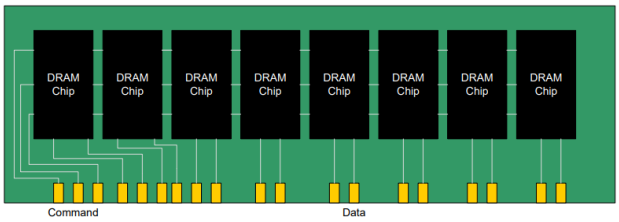
Σχήμα 1.7: DRAM chip.

Ωστόσο για την κατασκευή ενός chip με μεγάλο interface όπως φαίνεται με 32 bits απαιτείται κοστοβόρο υλικό, για αυτό χρησιμοποιούμε πολλά chips με μικρότερο interface για την δημιουργία της διεπαφής. Έτσι σχηματίζεται το rank δηλαδή ένα σύνολο από chips που αποκρίνονται στην ίδια εντολή και στην ίδια διεύθυνση ταυτόχρονα, αποθηκεύοντας όμως διαφορετικό μέρος των ζητούμενων δεδομένων. Αυτό που τελικά συνδέεται στην μητρική μας κάρτα είναι το αποκαλούμενο DIMM το οποίο περιλαμβάνει 1 ή περισσότερα ranks και φαίνεται στην ακόλουθη εικόνα.



Σχήμα 1.8: 64-bit Wide DIMM (one rank).

Έτσι λοιπόν σχηματίζεται η συνολική μνήμη, η οποία χρησιμοποιείται στους σημερινούς υπολογιστές.



Σχήμα 1.9: DRAM chip.

1.2.3 Λειτουργία του κυττάρου μνήμης

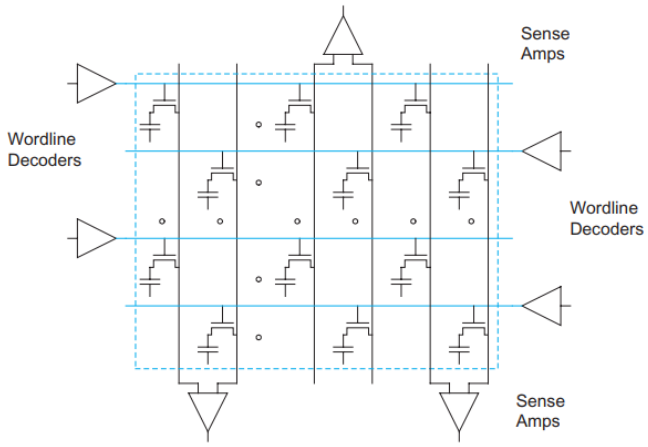
Σε αυτήν την παράγραφο αναλύεται ο τρόπος με τον οποίο λειτουργεί η εγγραφή και η ανάγνωση στην μνήμη. Θεωρούμε μια λειτουργία εγγραφής και υποθέτουμε ότι η γραμμή λέξης και η γραμμή bit βρίσκονται στην τάση τροφοδοσίας Vdd. Έτσι το τρανζίστορ θα άγει φορτίζοντας τον πυκνωτή αποθήκευσης δεδομένων. Καθώς λοιπόν φορτίζεται ο πυκνωτής, η τάση στα άκρα του θα φτάσει κοντά στην τροφοδσία μετά από ένα μικρό χρονικό διάστημα. Ωστόσο, λόγω φαινομένων διαρροής το φορτίο του πυκνωτή μειώνεται και για αυτό το κύτταρο πρέπει να ανανεώνεται περιοδικά προκειμένου να διατηρεί ορθή τιμή. Κατά την διάρκεια της ανανέωσης διαβάζεται το περιεχόμενο του κυττάρου και γράφεται εκ νέου η ψηφιακή πληροφορία σε αυτό, αποκαθιστώντας την τάση του πυκνωτή στην σωστή τιμή. Η ανανέωση αυτή γίνεται περίπου κάθε 5ms έως 10ms [1]. Αφού ο

αποκωδικοποιητής γραμμής επιλέγει μια συγκεκριμένη γραμμή αυξάνοντας την τάση κατά μήκος της γραμμής λέξης, τα τρανζίστορ προσπέλασης υποχρεώνονται σε κατάσταση αγωγής, με αποτέλεσμα να συνδέουν τους πυκνωτές αποθήκευσης των κυττάρων με τις γραμμές bit. Στην περίπτωση της λειτουργίας ανάγνωσης η γραμμή bit έχει προφορτιστεί σε Vdd/2. Σημειώνουμε ότι η χωρητικότητα της γραμμής bit είναι 10 φορές μεγαλύτερη από την χωρητικότητα του κυττάρου. Αν ο πυκνωτής είναι αρχικά αφόρτιστος, τότε η τάση στα άκρα της γραμμής bit μειώνεται ενώ αν είναι αρχικά φορτισμένος, η τάση στα άκρα της γραμμής bit αυξάνεται σε μέγεθος μέχρι 90mV. Η διδαδικασία ανάγνωσης είναι καταστροφική, αφού αλλοιώνει το περιεχόμενο του πυκνωτή του κυττάρου μνήμης. Η μικρή αυτή μεταβολή ανιχνεύεται στην συνέχεια απο τον ενισχυτή, ο οποίος ανάλογα με την αύξηση ή την μείωση οδηγεί την γραμμή και κατ' επέκταση τον πυκνωτή στο Vdd ή στην γη αποθηκεύοντας ορθά την τιμή. Έτσι ανανεώνονται τα κύτταρα της επιλεγμένης γραμμής. Ταυτόχρονα το σήμα στην έξοδο του ενισχυτή ανίχνευσης της επιλεγμένης στήλης τροφοδοτείται στην γραμμή εξόδου δεδομένων μέσω του αποκωδικοποιητή διεύθυνσης στήλης.

Η λειτουργία εγγραφής διεξάγεται με παρόμοιο τρόπο εκτός από ότι το προς εγγραφή bit, το οποίο υπάρχει στην γραμμή εισόδου δεδομένων και εφαρμόζεται από τον αποκωδικοποιητή διεύθυνσης στήλης στην επιλεγμένη γραμμή bit. Ανάλογα με το φορτίο αυτό μεταβάλλεται και η τάση στον πυνωτή του κυττάρου, ενώ ταυτόχρονα ανενώνονται και όλα τα υπόλοιπα κύτταρα της επιλεγμένης γραμμής.

Αν και οι λειτουργίες εγγραφής και ανάγνωσης οδηγούν σε αυτόματη ανανέωση του περιεχομένου όλων των κυττάρων της εκάστοτε γραμμής ο memory controller που διαθέτει η μνήμη λαμβάνει μέριμνα για περιοδική ανανένωση του συνόλου της μνήμης ανά κάποιο χρονικό διάστημα από 5 έως 10ms ανάλογα με τις προδιαγραφές του εκάστοτε κατασκευαστή. Κατά την διάρκεια της ανανέωσης το ολοκληρωμένο δεν είναι διαθέσιμο για λειτουργίες εγγραφής και ανάγνωσης. Λόγω του μικρού εύρους μεταβολής οι γραμμές bit είναι πολύ ευαίσθητες στον θόρυβο και για αυτό χρησιμοποιούνται διάφορες τεχνικές με σκοπό την αντιμετώπιση αυτου του προβλήματος. Μία από αυτές είναι η αρχιτεκτονική διπλωμένων γραμμών που παρουσιάζεται στο σχήμα 1.10. Σε αυτήν την αρχιτεκτονική κάθε γραμμή bit συνδέεται μόνο με τα μισά κύτταρα. Οι γειτονικές γαρμμές bit οργανώνονται σε ζεύγη και χρησιμοποιούνται ως είσοδοι στους ενισχυτές αίσθησης. Όταν τίθεται μια γραμμή λέξης η μία γραμμή bit θα αλλάξει κατάσταση, ενώ η γειτονική της θα λειτουργήσει ως αναφορά. Επειδή πολλές πηγές θορύβου επηρεάζουν εξίσου τις δύο γειτονικές γραμμές bit, εμφανίζονται ως θόρυβος λειτουργίας κοινού σήματος ο οποίος απορρίπτεται από τον ενισχυτή. Το πλεονέκτημα αυτό

έρχεται όμως με το κόστος ότι χρειαζόμαστε περισσότερες γραμμές και άρα περισσότερες παρασιτικές χωρητικότητες και μεγαλύτερη επιφάνεια στο φυσικό σχέδιο.



Σχήμα 1.10: Διπλωμένες γραμμές bit.

1.3 Χρονισμοί Μνήμης

Για την ορθή λειτουργία της μνήμης είναι απαραίτητο να πληρούνται κάποιες απαιτήσεις στους χρονισμούς μεταξύ των οποίων στέλνονται οι δάφορες εντολές στην μνήμη. Οι βασικότερες εντολές παρουσιάζονται στον παρακάτω πίνακα.

Εντολές	**Λειτουργία**
ACT(Activate)	Με αυτήν την εντολή ενεργοποιείται η γραμμή που συνδέει τα κύτταρα μνήμης με τις γραμμές bit.
PRE(Precharge)	Κλείνει την ήδη υπάρχουσα γραμμή που είναι ανοιχτή και προφορτίζει τις γραμμές bit στο Vdd/2.
RE(Read)	Με αυτήν την εντολή διαβάζουμε τα δεδομένα τις διεύθυνσης που έχουμε αποστείλει.
WR(Write)	Με αυτήν την εντολή γράφουμε τα δεδομένα που στέλνουμε σε μια συγκεκριμένη διεύθυνση.

Πίνακας 1.1: Βασικές Εντολές Μνήμης

Οι εντολές αυτές όμως δεν μπορούν να χρησιμοποιούνται με οποιοδήποτε

τρόπο, αλλά θα πρέπει να ακολουθούν μια διαδοχική σειρά προκειμένου να γράφουμε και να διαβάζουμε στην μνήμη. Για παράδειγμα δεν μπορούν να δοθούν ταυτόχρονα δύο διαδοχικές εντολές ACT, καθώς ο memory controller θα απορρίψει την δεύτερη διεύθυνση που θα δοθεί για ενεργοποίηση. Επίσης δεν μπορούμε να εκτελέσουμε εντολές για διάβασμα και γράψιμο χωρίς να έχει προηγηθεί κάποια εντολή ACT με σκοπό νε ενεργοποιηθεί κάποια γραμμή. Αλλά ακόμα και εντολές που μπορούν δρομολογηθούν διαδοχικά θα πρέπει να πληρούν κάποιους χρονικούς περιορισμούς, ώστε η εντολή να οδηγήσει στο επιθυμητό αποτέλεσμα. Μερικοί χρονικοί περιορισμοί φαίνονται στον παρακάτω πίνακα. [2]

Χρονισμοί	**Περιγραφή**
t_{RAS}	Row Access Strobe. Η χρονική διάρκεια που απαιτείται από την ενεργοποίηση της γραμμής με εντολή ACT μέχρι να επέλθει μια εντολή PRE.
t_{RP}	Row Precharge. Η χρονική διάρκεια που απαιτείται για την ενεργοποίηση της γραμμής με εντολή ACT από μια εντολή PRE.
t_{RC}	Row Cycle. Η χρονική διάρκεια που απαιτείται για την ενεργοποίηση δύο διαφορετικών γραμμών.
t_{WR}	Write Recovery Time. Η χρονική διάρκεια που απαιτείται να στείλουμε PRE αφού έχουμε γράψει δεδομένα στη μνήμη.
t_{RTP}	Read to Precharge. Η χρονική διάρκεια που απαιτείται να στείλουμε PRE, αφού έχουμε διαβάσει δεδομένα από τη μνήμη.

Πίνακας 1.2: Βασικοί Χρονισμοί Μνήμης

Οι χρονισμοί αυτοί είναι απαραίτητο να τηρούνται με σκοπό η μνήμη να επιτελεί την λειτουργία της. Υπέυθυνος για την τήρηση των παραπάνω χρονισμών είναι ο Memory Controller στον οποίο ο χρήστης μεταβιβάζει τις εντολές και ο MC τηρώντας τους χρονισμούς στέλνει τα κατάλληλα σήματα. Οι χρονισμοί αυτοί είναι άμεσα συσχετιζόμενοι με τις χωρητικότητες που έχουν προκύψει κατά τον φυσικό σχεδιασμό της μνήμης και δίνονται από τον εκάτσοτε κατασκευαστή. Στην παρούσα εργασία παραβίασαμε τους περιορισμούς αυτούς με σκοπό να λειτουργήσουμε εκτός των ορίων που δίνονται και να αξιοποιήσουμε το υπάρχον υλικό με στόχο να εκτέλεουμε

πράξεις μέσα στην ίδα την μνήμη. Διαφορετικές ακολουθίες προσπελάσεων με διαφορετικές τοπικότητες έχουν διαφορετικούς χρόνους απόκρισης. Για αυτό οι κατασκευαστές δίνουν χρονικούς περιορισμούς μεγαλύτερους από ό,τι συνήθως προκύπτουν για να καλύψουν τυχόν αστοχίες που ενδέχεται να προκύψουν. Η βασική λετουργία του Memory Controller δηλαδή είναι να κάνει το refresh στα διάφορα κύττρα, ώστε να μην αλλοιώνεται το περιεχόμενο τους και να εξυπηρετεί τις αιτήσεις μνήμης προς την DRAM σεβόμενος τους χρονικούς περιορισμούς. Για να το κάνει αυτό, αποθηκεύει τις αιτήσεις σε έναν ενδιάμεσο buffer και στην συνέχεια τις χρονοδρομολογεί με στόχο την υψηλή επίδοση.

1.4 Δομικά Στοιχεία για την εκτέλεση πράξεων

Η εκτέλεση υπολογιστικών πράξεων εντός μνήμης (in-memory computing) έχει εδώ και καιρό προταθεί ως λύση στο πρόβλημα του "Memory Wall". Το πρόβλημα αυτό ουσιαστικα αναφέρει ότι η εξέλιξη στους επεξεργαστές δεν συμβαδίζει με την εξέλιξη που έχει η μνήμη και για αυτόν τον λόγο δεν έχουμε το κατά μέγιστο κέρδος στην ταχύτητα. Ο αυξανόμενος αριθμός των πυρήνων στους επεξεργαστές έχει εξελιχθεί σε πολύ σημαντικότερο βαθμό από από το memory bandwidth, με αποτέλεσμα το latency ανάμεσα στους υπολογιστικούς πόρους και την off-chip μνήμη να παραμένει ένα εμπόδιο για την ταχύτερη επεξεργασία. Στην παρούσα εργασία μελετούμε τον υπολογισμό εντός μνήμης, με καθόλου τροποποιήσεις στον σχεδιασμό της DRAM χρησιμοποιώντας έτοιμες, εμπορικά διαθέσιμες και μη τροποποιημένες DRAM. Αυτό επιτυγχάνεται παραβιάζοντας τις ονομαστικές χρονικές προδιαγραφές (timing specifications) και ενεργοποιώντας διαδοχικά πολλές γραμμές σε πολύ μικρό χρονικό διάστημα, γεγονός που οδηγεί στο να παραμένουν ταυτόχρονα πολλαπλές γραμμές ανοικτές. Αυτό με τη σειρά του, επιτρέπει τον διαμοιρασμό φορτίου (charge sharing) στις γραμμές bit. Χρησιμοποιώντας σειρά εντολών που παραβιάζει τις χρονικές προδιαγραφές, μπορούμε να εκτελέσουμε πράξεις όπως ειναι η αντιγραφή δεδομένων και λογικές πράξεις μέσα στην μνήμη, όπως OR, AND. Στην συνέχεια με τις παραπάνω αυτες πράξεις φτιάχνουμε μια λογική μονάδα που μπορεί να προσθέτει αριθμούς (Full Adder). Με τη χρήση προσαρμοσμένου ελεγκτή DRAM (DRAM controller) σε FPGA και με εμπορικά διαθέσιμες DRAM μονάδες, εξετάζουμε το ποσοστό επιτυχίας που μπορεί να προσφέρει η εκτέλεση πράξεων μέσα στην μνήμη.

1.4.1 Προσαρμοσμένος Memory Controller

Για τον έλεγχο των χρονισμών των εντολών μέσα στην μνήμη είναι απαραίτητη η χρήση ενός custom hardware, το οποίο θα αναλαμβάνει τον έλεγχο της μνήμης. Για αυτόν τον λόγο θα χρησιμοποιήσουμε το "DRAM BENDER", το οποίο είναι μια δομή που έχει υλοποιηθεί σε FPGA και επιτρέπει τον πειραματισμό σε εμπορικές μνήμες[5]. Η δομή αυτή είναι ιδιαίτερα εύχρηστη και μπορεί να χρησιμοποιηθεί και σε μνήμες DDR4. Συγκεκριμένα, εκμεταλλεύεται πλήρως το interface των μνημών που υπόκεινται σε έλεγχο και μπορούμε με γλώσσες υψηλού επιπέδου να προσαρμόσουμε τον χρονισμό των εντολών χωρίς περιορισμούς. Το block διάγραμμα φαίνεται στην παρακάτω εικόνα.

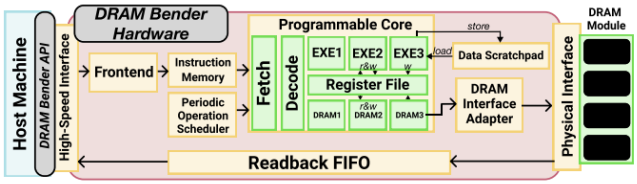


Figure 2: Block diagram of DRAM Bender

Σχήμα 1.11: Διάγραμμα του Dram Bender.

Η δομή αυτή υποστηρίζει ένα custom instruction set που είναι κατάλληλο για τον έλεγχο των μνημών DRAM όπως οι ACT, PRE, WRITE, READ και μπορεί να εκτελεί οποιαδήποτε σειρά εντολών χωρίς να είναι απαραίτητοι οι χρονικοί περιορισμοί που έχουν οι μνήμες. Η ακρίβεια στους χρονισμούς ορίζεται με βάση την ταχύτητα του FPGA και βρίσκεται σε επίπεδο nanosecond. Το host machine μπορεί να επικοινωνήσει μέσω PCIe και να στείλει προγράμματα που είναι γραμμένα σε γλώσσες υψηλού επιπέδου, όπως C++ και Python. Στην συνέχεια τα δεδομένα αποθηκεύονται σε έναν buffer, ο οποίος έχει εύρος 512 bits και επιστρέφονται πίσω στο host machine. Για τον προγραμματισμό και την αποστολή εντολών προς την μνήμη, ο χρήστης μπορεί μέσα απο αντικείμενα να κατασκευάσει μια λίστα από εντολές. Μπορεί να χρησιμοποιήσει labels, ώστε να εκτελεί branches καθώς η χρήση επαναληπτικών δομών καταλαμβάνουν μεγαλύτερο μέγεθος στην μνήμη εντολών. Στην παρούσα εργασία το FPGA που χρησιμοποιήθηκε είναι το ALVEO U200. Για την αποστολή των εντολών αρχικά τοποθετούνται σε μια λίστα οι λειτουργίες της μνήμης που θέλουμε να εκτελέσουμε. Μία εντολή FPGA περιλαμβάνει 4 εντολές για την μνήμη καθώς η ταχύτητα με την οποία μπορούν να εκτελεστούν είναι περίπου τετραπλάσια. Ανάμεσα σε διαδοχικές εντολές μπορούμε να τοποθετήσουμε εντολές που είναι idle και λειτουργούν

σαν καθυστερήσεις. Έτσι λοιπόν μπορούμε να έχουμε έλεγχο του χρόνου που μεσολαβεί ανάμεσα σε διαδοχικές εντολές. Η εγγραφή και η ανάγνωση γίνονται με την χρήση buffers και έτσι μπορούμε να έχουμε πλήρη έλεγχο των μνημών προς εξέταση.

1.4.2 Πράξεις Στην Μνήμη

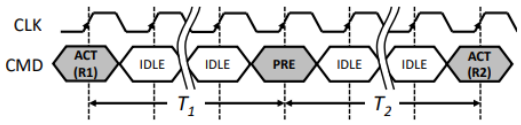
Για την εκτέλεση των πράξεων εντός της μνήμης θα χρησιμοποιήσουμε τις βασικές εντολές των μνημών με σκοπό να εκμεταλλευτούμε τις χωρητικότητες.

Precharge: Η εντολή της προφόρτισης χρησιμοποιείται με σκοπό να κλείσει την γραμμή η οποία έχει ήδη ανοιξει οδηγώνας όλα τα word-lines στην γη και προφορτίζει τα bit lines σε Vdd/2. Εφαρμόζεται σε όλο το Bank, το οποίο δίνεται σαν διεύθυνση.

Activate: Η εντολή της ενεργοποίησης στοχεύει μια συγκεκριμένη γραμμή και θέτει το word line ίσο με Vdd. Πριν έρθει η εντολή αυτή, πρέπει να έχει προηγηθεί μια εντολή προφόρτισης, ώστε τα bit lines να βρίσκονται σε δυναμικό Vdd/2. Όταν ενεργοποιηθεί το τρανζίστορ, τότε ο πυκνωτής που διατηρεί την πληροφορία συνδέεται με το bit -line και συμβαίνει διαμοιρασμός φορτίου. Επειδή η χωριτικότητα του κελιού είναι σχετικά μικρή, επηρεάζει κατά ένα πολύ μικρό ποσοστό την τάση στο bit-line. Ο ενισχυτής αντιλαμβάνεται αυτήν την μικρή διαφορά και με θετική ανάδραση οδηγεί την έξοδο είτε στην τροφοδοσία είτε στην γη.

Read/Write: Οι εντολές αυτές αφορούν 8 διαδοχικά συνεχόμενες στήλες και ξεκινούν από την αφετηρία που την δίνουμε σαν διεύθυνση. Τα δεδομένα που διαβάζονται από τους ενισχυτές μεταφέρονται σε global buffers και στην συνέχεια στα I/O pins.

Αν σε αυτές τις εντολές μειώσουμε τους χρόνους μεταξύ των οποίων στέλνονται διαδοχικά, τότε μπορούμε να εκτελέσουμε απλές λογικές πράξεις μέσα στην μνήμη, όπως απλές βασικές πύλες AND, OR. Η βασική αρχή στην οποία στηρίζονται οι πράξεις μέσα στην μνήμη είναι η διαδοχική εκτέλεση των εντολών ACT(R1)-PRE-ACT(R2). Με αυτήν την ακολουθία εντολών αν παραβιάσουμε τους περιορισμούς στα διαστήματα T1 και T2, τότε μπορούμε να εκτελέσουμε διάφορες πράξεις, όπως φαίνεται στο παρακάτω σχήμα.

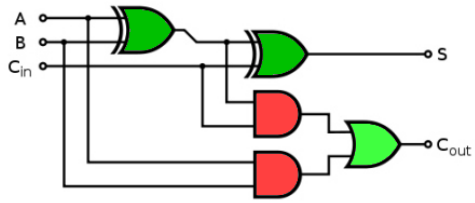


Σχήμα 1.12: Διάγραμμα ACT-PRE-ACT.

Η πιο απλή πράξη που μπορούμε να εκτελέσουμε μέσα στην μνήμη είναι η αντιγραφή μιας ολόκληρης γραμμής σε μια άλλη. Για να το πετύχουμε αυτό ενεργοποιούμε την γραμμή R1 και περιμένουμε κάποιο χρονικό διάστημα T1 μέχρι να φορτιστούν οι πυκνωτές και η γραμμή bitline. Τότε εκτελούμε την εντολή Precharge η οποία αφού κλείσει την γραμμή, τείνει να επαναφέρει την γραμμή Bitline σε δυναμικό Vdd/2. Αν παραβιάσουμε το χρονικό διάστημα t_{RP}, τότε η γραμμή bitline έχει δυναμικό μεγαλύτερο από Vdd/2. Επομένως, αν ανοίξει μια νέα γραμμή, το φορτίο της στήλης θα μείνει σχεδόν αναλλοίωτο, θα ενισχυθεί και θα περάσει και στους πυκνωτές της νές γραμμής που άνοιξε. Ωστόσο, το χρονικό διάστημα T2 δεν πρέπει να γίνει πολύ μικρό, γιατί αυτό θα ανοίξει και νέες γραμμές, οδηγώντας σε διαφορετικό αποτέλεσμα. Αν μειώσουμε και τους δύο χρονισμούς T1 και T2, τότε εκμεταλλευόμενοι αυτό το γεγονός, ενεργοποιούμε ταυτόχρονα περισσότερες από μια γραμμές. Τότε με τον διαμοιραασμό φορτίου που παρατηρείται πάνω στην γραμμή bitline, καταφέρνουμε να εκτελέσουμε λογικές πράξεις, οι οποιές στην συνέχεια ως καθολικές πύλες μπορούν να υλοποιήσουν οποιαδήποτε συνάρτηση, ακόμα και τον λογικό αθροιστή. [1]

1.5 Αποτελέσματα και Συμπεράσματα

Στην παρούσα εργασία για να βρεθούν με σχετική ακρίβεια οι καθυστερήσεις σχεδιάσαμε τον ενισχυτή στην τεχνολογία 90νμ στο ἀδενςε. Μέσα από τις προσομοιώσεις βρίσκουμε ότι οι καθυστερήσεις είναι μερικά nanoseconds. Έτσι, έχουμε μια εικόνα της τάξης των οποίων πρέπει να είναι οι καθυστερήσεις, με σκοπό να ρυθμίσουμε τους χρόνους idle ανάμεσα στις εντολές. Έτσι θα πετύχουμε τον διαμοιρασμό φορτίου πάνω στην γραμμή bitline και θα πετύχουμε την υλοποίηση των απλών λογικών πράξεων. Ωστόσο, μέσα στην μνήμη επειδή δεν μπορούμε να εφαρμόσουμε την λογική πράξη της αντιστροφής, αποθηκεύουμε τα δεδομένα και στις δύο τους καταστάσεις. Αυτό βέβαια έχει το μειονέκτημα ότι καταναλώνουμε περισσότερες γραμμές στην μνήμη, το οποίο όμως θεωρούμε αμελητέο καθώς κάθε subarray έχει στο σύνολο του 512 γραμμές. Ουσιαστικά λοιπόν όποια πράξη εφαρμόζεται στα δεδομένα που θέλουμε, η συμπληρωματικη της πράξη πρέπει να εφαρμόζεται και στα αντίστροφα δεδομένα της. Έχοντας αυτό ως βάση, υλοποιούμε τον αθροιστή που με απλές λογικές πύλες παρουσιάζεται παρακάτω.



Σχήμα 1.13: Πλήρης Αθροιστής.

Η υλοποίηση αυτή μπορεί να μην επιτυγχάνει καλύτερη ταχύτητα από έναν απλό επεξεργαστή, αλλά αν στόχος είναι η εκτέλεση διανυσματικών πράξεων, τότε επειδή στην γραμμή υπάρχουν 65×2^{10} στήλες μπορούμε να εφαρμόσουμε ταυτόχρονα την πράξη της άθροισης σε όλες αυτές και άρα να έχουμε πολύ μεγαλύτερο παραλληλισμό. Ωστόσο, το αποτέλεσμα είναι ότι δεν εμφανίζεται σωστά το αποτέλεσμα σε όλες τις γραμμές, αλλά περίπου στο 80% αυτών. Επιπλέον, εκτελόντας διαδοχικά στο ίδιο κύτταρο την πράξη της πρόσθεσης, βλέπουμε ότι το αποτέλεσμα δεν είναι πάντα ορθό και σε κάποιες περιπτώσεις αλλάζει ανάλογα με τις εισόδους που δέχεται το κάθε κύτταρο. Εκτελόντας πρόσθεση 2 έως 16 bit βρίσκουμε ότι το ποσοστό κυμαίνεται από 70% μέχρι και 30% για τους αριθμούς 16 bit.

Chapter 2

Introduction

As the demand for faster and more energy-efficient computing grows, traditional computing architectures face significant challenges. The von Neumann bottleneck, a limitation in conventional systems where data constantly moves between memory and processors, has become a major obstacle in achieving high-speed and low-power computing. To address this issue, Compute-in-Memory is addressed as an innovative approach that integrates computation directly into memory hardware [1] [6] [7].

Compute-in-Memory eliminates the need for excessive data transfer by performing computations where the data is stored. Instead of constantly shuttling information between memory and processors, CIM architectures allow operations such as logical operations, and even deep learning computations to occur within the memory itself. This significantly reduces power consumption and improves processing speed, making it an ideal solution for applications such as artificial intelligence, edge computing, and big data processing.

Prior research has demonstrated that logical operations can be performed directly in memory without requiring hardware modifications. For instance, RowClone, an in-memory technique, enables efficient data copying between rows. Additionally, DRAM hardware can be leveraged to execute bitwise logical operations, such as logical AND and OR. Other studies suggest that storing fractional voltages in DRAM cells allows for multi-input logical operations [1] [8] [7] [9]. Furthermore, it has been shown that activating multiple rows within a DRAM subarray—a fundamental mechanism behind these operations—can further enhance computational capabilities.

In this work, we build on previous research to implement an in-memory

Full Adder and evaluate its success rate and performance. To achieve this, we utilize fundamental logical operations and apply them to a DDR4 module. Additionally, we analyze the underlying DRAM hardware architecture and demonstrate how it can be leveraged to execute these operations efficiently.

Chapter 3

Theoretical background

3.1 DRAM's chip

Dynamic memories (DRAM) store their data in the form of electric charge within a capacitor. This design allows for extremely compact memory layouts, making DRAM highly dense and efficient. However, a key drawback is that capacitors leak charge over time, requiring frequent refresh operations to maintain data integrity. A single DRAM cell consists of one transistor and one capacitor, as illustrated in the figure below.

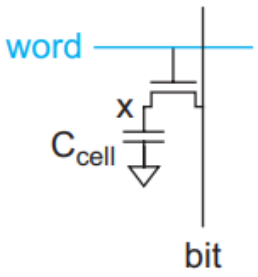


Figure 3.1: Storage Cell.

DRAM Cell Operation The DRAM cell is accessed by activating the wordline, which raises its voltage to Vdd, causing the access transistor to conduct (operating in the triode region) and introducing a low resistance path. This connects the capacitor to the bitline, which is initially precharged to Vdd/2.

Once connected, the capacitor shares its charge with the bitline, inducing a voltage change (ΔV) at the bitline node. The behavior of this voltage

change depends on whether the capacitor was initially charged or discharged.

If the capacitor was initially charged, charge flows from the capacitor into the bitline's parasitic capacitance, causing the bitline voltage to increase. If the capacitor was initially discharged, charge flows from the bitline to the capacitor, causing the bitline voltage to decrease. This voltage fluctuation, illustrated in the figure below, represents the stored binary value (0 or 1).

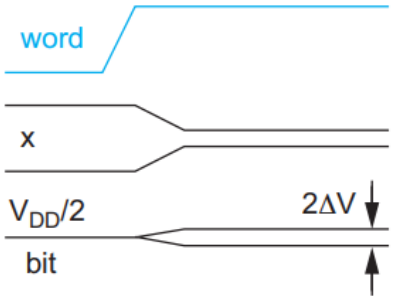


Figure 3.2: Cell Voltage.

Since the read process disturbs the stored charge, the cell's voltage level is altered after each read. As a result, every read operation requires a subsequent write-back (refresh) to restore the original data.

To fully understand how the capacitor in a DRAM cell can be charged to the maximum supply voltage V_{DD}, consider a write-1 operation. During this process, both the word line and the bit line are set to V_{DD}, allowing the access transistor to conduct and charge the storage capacitor C_S. However, conduction stops once the capacitor voltage reaches $(V_{DD} - V_t)$, a limitation similar to that in pass-transistor logic (PTL). To address this, DRAM designs incorporate a technique where the word line voltage is boosted to $V_{DD} + V_t$, ensuring the capacitor reaches the full V_{DD}. Despite this, charge leakage over time necessitates periodic refreshing. During a refresh cycle, the cell's data is read and rewritten to restore the capacitor voltage. Typically, this refresh occurs every **5 to 10 milliseconds**.

The row decoder selects a specific row by increasing the word line voltage, activating all access transistors within that row. As a result, the storage capacitors in the selected row become connected to their corresponding bit lines. This means that each cell capacitor C_S is electrically linked in parallel with the bit-line capacitance C_B, as depicted in next figure.

Figure 3.3: When the voltage of the selected word line is raised, the transistor conducts, thus connecting the storage capacitor with the bitline capacitor.

In typical DRAM configurations, C_S ranges between **20 fF and 30 fF**, whereas C_B is approximately **ten times larger**. When performing a read operation, the bit line is precharged to $\frac{V_{DD}}{2}$. To determine how much the bit-line voltage changes when a cell capacitor C_S is connected, we assume the initial capacitor voltage is V_{CS}, where $V_{CS} = V_{DD}$ for a stored '1' and $V_{CS} = 0V$ for a stored '0'.

Applying charge conservation, we obtain:

$$C_S V_{CS} + C_B \frac{V_{DD}}{2} = (C_B + C_S)\left(\frac{V_{DD}}{2} + \Delta V\right)$$

Solving for ΔV, we get:

$$\Delta V = \frac{C_S}{C_B + C_S}\left(V_{CS} - \frac{V_{DD}}{2}\right)$$

Since C_B is significantly larger than C_S, we approximate:

$$\Delta V \approx \frac{C_S}{C_B}\left(V_{CS} - \frac{V_{DD}}{2}\right)$$

For a stored '1' ($V_{CS} = V_{DD}$), the voltage change is:

$$\Delta V(1) \approx \frac{C_S}{C_B}\frac{V_{DD}}{2}$$

For a stored '0' ($V_{CS} = 0$), the voltage shift becomes:

$$\Delta V(0) \approx -\frac{C_S}{C_B}\frac{V_{DD}}{2}$$

However this voltage need some time to charge the capacitances (some nanoseconds) and this is not happening immediately. This is one of the parameters we will exploit to achieve compute in memory

3.2 DRAM Architecture

The Dram's architecture appears below

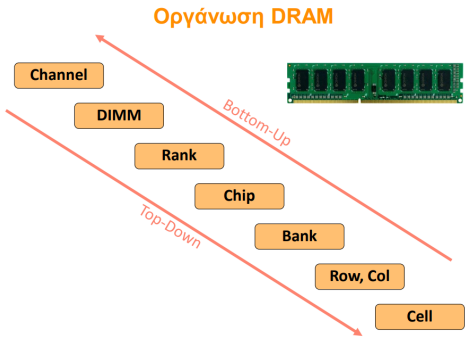


Figure 3.4: DRAM architecture.

At the lowest level, as described in the previous section, each DRAM cell consists of a capacitor that stores the logical value in the form of electric charge. Additionally, an access transistor connects and disconnects the capacitor from the bitline, enabling data access.

By arranging multiple DRAM cells in a matrix, we obtain the structure shown in the next figure. Each row shares a common wordline, which activates the access transistors of that row. The stored charge is then transferred as a voltage signal to the bitlines. Each bitline is connected to a sense amplifier, which detects and amplifies small voltage changes.

This structure forms what is known as a subarray, where only one row can be activated at a time.

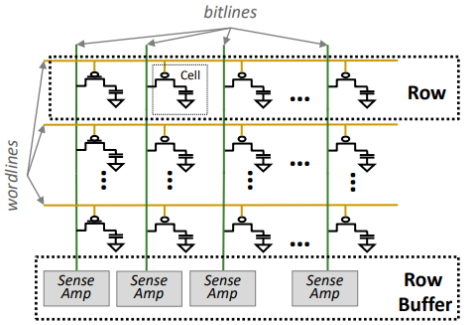


Figure 3.5: DRAM subarray level.

DRAM Subarrays and Memory Organization When organizing columns into groups of 8 bits, we obtain a grid-like structure of rows and columns. Row access is managed by a row address decoder, which selects a specific row by increasing its voltage. Consequently, all words in the selected row are accessed simultaneously. A column address decoder then determines the specific word that will be read.

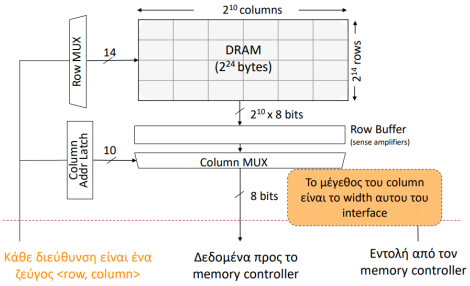


Figure 3.6: The circuit of the subarray.

If multiple subarrays are replicated extensively, they form the overall DRAM memory. However, a monolithic memory array would suffer from high access latency and lack of parallelism. To address this issue, DRAM is divided into banks, which are N times smaller than the total memory capacity. Accesses to different banks can overlap, improving memory efficiency. The bank selection is determined by specific bits in the memory address.

By combining multiple banks, we obtain a DRAM chip, typically consisting of 8 banks. Within each chip, banks share common data, command, and address buses, allowing simultaneous row activations across different banks.

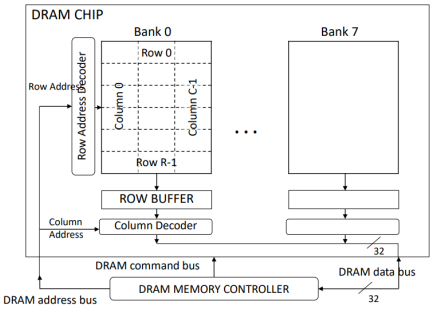


Figure 3.7: DRAM chip architecture.

A DRAM chip with a large data interface (e.g., 32-bit) requires expensive hardware. Instead of using a single wide interface chip, multiple smaller-interface chips are combined to achieve the required width. This results in the formation of a rank, a set of chips that respond to the same command and address simultaneously while storing different portions of the requested data.

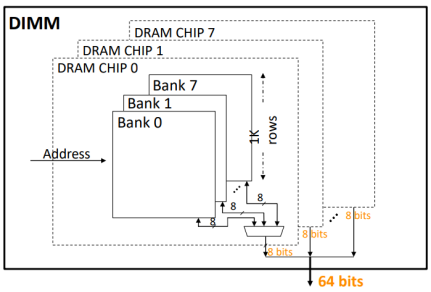


Figure 3.8: DRAM DIMM level.

The component that is ultimately installed on the motherboard is the DIMM (Dual In-line Memory Module), which contains one or more ranks. The final organization of DRAM memory, as used in modern computer systems, is shown in the following figure.

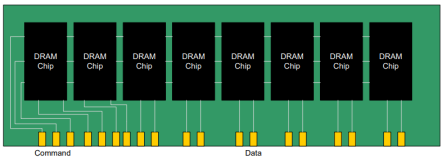


Figure 3.9: DRAM top level.

3.3 The Subarray Full Circuit

Consider a write operation where both the word line and the bit line are set to the supply voltage V_{DD}. As a result, the transistor conducts, charging the storage capacitor. Once the capacitor is charged, its voltage approaches the supply voltage after a short period. However, due to leakage effects, the charge in the capacitor decreases over time, which is why the cell must be periodically refreshed to maintain the correct value.

During the refresh cycle, the cell's content is read, and the digital information is rewritten, restoring the capacitor voltage to its proper level.

This refresh process occurs approximately every 5ms to 10ms. When the row decoder selects a specific row by increasing the voltage along the word line, the access transistors switch to the conducting state, connecting the storage capacitors of the cells to the bit lines.

In a read operation, the bit line is precharged to $V_{DD}/2$. It is important to note that the bit-line capacitance is ten times larger than the cell capacitance. If the capacitor is initially uncharged, the voltage at the bit line decreases, whereas if it is charged, the bit-line voltage increases by up to 90mV. The read process is destructive, as it disturbs the charge stored in the memory cell capacitor. The small voltage variation is then detected by the sense amplifier, which determines whether the voltage has increased or decreased and subsequently drives the bit line (and the capacitor) either to V_{DD} or ground, correctly restoring the stored value. This process ensures that the cells in the selected row are refreshed. Simultaneously, the signal at the output of the sense amplifier corresponding to the selected column is sent to the data output line through the column address decoder.

The write operation follows a similar process, except that the bit to be written, available at the data input line, is applied to the selected bit line via the column address decoder. Depending on the charge, the voltage on the cell capacitor changes accordingly, while all other cells in the selected row are also refreshed.

Although both read and write operations automatically refresh the contents of all cells in a given row, the memory controller is responsible for periodically refreshing the entire memory at intervals ranging from 5 to 10ms, depending on the manufacturer's specifications. During the refresh cycle, the memory module is unavailable for read or write operations.

Due to the small voltage variations, bit lines are highly susceptible to noise, necessitating various techniques to mitigate this issue. One such technique is the folded bit-line architecture, illustrated in the next figure. In this architecture, each bit line is connected to only half of the cells, while neighboring bit lines are grouped into pairs and serve as inputs to the sense amplifiers. When a word line is activated, one bit line changes state, while its neighboring bit line functions as a reference. Since many noise sources affect both adjacent bit lines equally, they appear as common-mode noise, which is rejected by the sense amplifier. However, this advantage comes at a cost: more bit lines are required, leading to increased parasitic capacitance and a larger physical layout in the memory design.

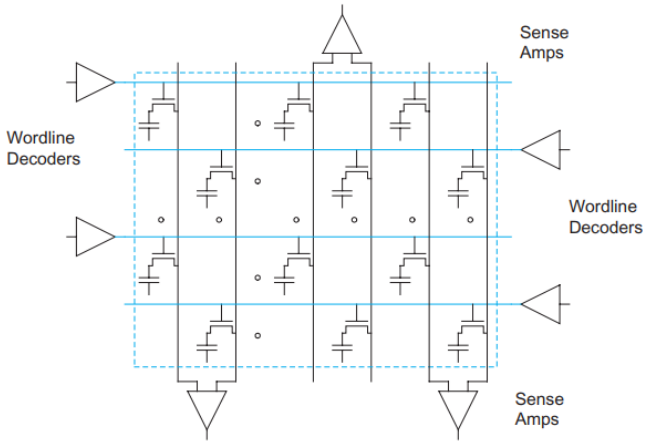


Figure 3.10: Folded Bitline Architecture.

3.4 Timing and Basic Commands

For the correct operation of memory, it is necessary to satisfy certain timing requirements that define the order in which various commands are sent to the memory. The fundamental commands are presented in the table below.

Instructions	**Operation**
ACT(Activate)	This command activates a row, allowing access to the bit lines.
PRE(Precharge)	Closes a previously activated row, resetting the bit lines to $V_{dd}/2$.
RE(Read)	Reads the data stored in a specific address within the activated row.
WR(Write)	Reads the data stored in a specific address within the activated row.

Table 3.1: Basic Memory Commands

However, these commands cannot be executed in arbitrary order; they must adhere to strict timing constraints that govern the interactions between different operations in the memory. These constraints may include minimum time intervals required between consecutive commands, such as an appropriate delay between an ACT command and a subsequent read or write operation. Additionally, other commands may need to be executed in sequence to achieve the desired operation efficiently. Various timing

constraints are illustrated in subsequent tables and figures.

These timing constraints are crucial to ensure the proper operation of memory. The Memory Controller is responsible for enforcing these timing requirements by managing the user's access requests and issuing appropriate control signals. The timing parameters are closely related to the physical characteristics of the memory and are defined by the manufacturers based on experimental measurements. Efficient management of these timing constraints is essential to optimize hardware utilization while maintaining performance.

Timings	**Details**
t_{RAS}	Row Access Strobe.The time required from activating a row using the ACT command until issuing a PRE command.
t_{RP}	Row Precharge. The time required for the activation of a row with an ACT command until a PRE command is issued.
t_{RC}	Row Cycle. The total time required to complete an entire row access cycle, including precharge time.
t_{WR}	Write Recovery Time. The minimum delay required after a write operation before issuing a PRE command.
t_{RTP}	Read to Precharge. The minimum delay required after a read operation before issuing a PRE command.

Table 3.2: Basic Timing Parameters

Proper handling of memory operations relies on these timing parameters. Various techniques, such as pipelining and out-of-order execution, are employed to optimize memory access performance. In modern systems, memory timing plays a critical role in determining overall system efficiency, as incorrect handling can lead to performance degradation or even data corruption.

Chapter 4

Experiments and Results

4.1 Framework

DRAM Bender [5] implements a custom Instruction Set Architecture (ISA) optimized for DRAM control. The instruction set includes key DRAM operations such as:

- `ACT (Activate)` – Opens a specific DRAM row.
- `PRE (Precharge)` – Closes an open row to allow access to a new row.
- `WRITE` – Writes data to DRAM.
- `READ` – Reads data from DRAM.

DRAM Bender is an FPGA-based infrastructure designed to enable fine-grained control over DRAM commands. It provides a programmable interface that allows users to issue DRAM commands in arbitrary order while adhering to or intentionally violating standard timing constraints. The framework is particularly useful for DRAM characterization, security research, and performance optimization. This instruction set allows executing arbitrary sequences of DRAM commands while circumventing the default memory controller timing constraints. The accuracy of command scheduling is determined by the operating frequency of the FPGA, reaching the nanosecond scale.

The host system communicates with DRAM Bender via a PCIe interface, enabling efficient transfer of control instructions and data. DRAM Bender allows users to define DRAM test programs using high-level programming languages, such as C++ and Python. These programs are transmitted to

the FPGA, where they are executed, and the resulting data is stored in a 512-bit buffer before being sent back to the host for further analysis.Each DRAM instruction can contain up to four DRAM commands, allowing for efficient scheduling and execution of operations.

To program the FPGA and issue commands to DRAM, users construct a sequence of instructions organized in a command list. The execution model supports:

- **Branching Mechanisms:** Users can define branch labels to implement conditional execution.
- **Loop Constructs:** While loops and iterative structures are supported, though they may consume more instruction memory.

4.2 Sense Amplifier Circuit

Apart from the storage cells, the sense amplifier is one of the most crucial elements in a memory chip. It plays a vital role in the functionality of DRAMs, while its implementation in DRAMs contributes to improvements in both speed and area efficiency. There exist multiple designs for sense amplifiers, some of which bear a strong resemblance to the active-load MOS differential amplifier. This section introduces a differential sense amplifier that leverages positive feedback. However, the one-transistor DRAM cell described operates as a single-ended circuit, utilizing only a single bit line. To mimic a differential signal source in DRAMs, a technique known as the "dummy-cell" method is employed, which we will examine shortly. For analysis purposes, we assume that the memory cell being read generates a differential voltage between the B and $\overline{B}$ lines. This voltage, which can range from 20 mV to 500 mV depending on the specific memory type and cell design, is fed into the sense amplifier's input terminals.

The sense amplifier then amplifies this small voltage to produce a full-swing output signal ranging from 0 to V_{DD}. An interesting characteristic of the amplifier circuit under discussion is that its input and output terminals are the same. The next figure illustrates the sense amplifier along with a portion of the column circuitry in a RAM chip. The sense amplifier itself is essentially a standard latch, constructed by cross-coupling two CMOS inverters. The first inverter consists of transistors Q_1 and Q_2, while the second inverter is formed by Q_3 and Q_4. Additionally, transistors Q_5 and Q_6 function as switches, enabling the sense amplifier to connect to ground and V_{DD} only when data sensing is required. When ϕ_s is low, the amplifier

is deactivated, thereby conserving power. This power efficiency is crucial, as each column in the memory array has its own sense amplifier, resulting in thousands of such amplifiers per chip.

A notable characteristic of this circuit is that the terminals x and y serve as both input and output nodes. These terminals are linked to the complementary bit lines B and $\overline{B}$. The sense amplifier's role is to detect the small voltage differential between these bit lines and amplify it to produce a full-swing signal. For example, during a read operation, if the stored bit is a '1', a slight voltage difference develops where v_B is higher than $v_{\overline{B}}$. The amplifier then drives v_B up to V_{DD} and $v_{\overline{B}}$ down to 0V. This amplified output is forwarded to the chip's I/O pin via the column decoder (not shown in the figure) while simultaneously restoring the value '1' in the DRAM cell, ensuring data integrity despite the inherently destructive nature of DRAM reads.

The figure also depicts the precharge and equalization circuit, which ensures proper initialization of the bit lines before a read operation. When ϕ_p is high (equal to V_{DD}), all three transistors in the circuit become conductive. Transistors Q_8 and Q_9 work to precharge the bit lines B and $\overline{B}$ to $V_{DD}/2$, while Q_7 accelerates this process by equalizing the initial voltages of both lines. This equalization step is critical, as any pre-existing voltage difference between the bit lines before a read operation could lead to incorrect signal amplification by the sense amplifier. The process of reading data from a DRAM cell involves several key steps. The sequence of events during a read operation is outlined below:

1. **Precharge and Equalization:** The precharge and equalization circuit is activated by asserting the control signal ϕ_p. This ensures that the bit lines B and $\overline{B}$ are initially set to the same voltage, typically $V_{DD}/2$. Following this, ϕ_p is deactivated, leaving the bit lines floating momentarily before the next step.

2. **Word Line Activation:** The word line is raised, linking the memory cell to the bit lines B and $\overline{B}$. At this point, a small voltage difference emerges between the two lines. If the memory cell stores a logical '1', the voltage on B (v_B) becomes slightly higher than that on $\overline{B}$ ($v_{\overline{B}}$). Conversely, if the cell contains a logical '0', v_B is slightly lower than $v_{\overline{B}}$. To optimize performance at high speeds, this voltage difference is kept relatively small, typically in the range of 20–500 mV.

3. **Sense Amplification:** Once a sufficient voltage difference has developed between B and $\overline{B}$, the sense amplifier is activated. This is

achieved by enabling transistors Q_6 and Q_2, which connect the amplifier to both ground and V_{DD} via the sense control signal ϕ_s. Initially, the input nodes of the inverters within the amplifier are at $V_{DD}/2$, causing the circuit to be in a transitional, unstable equilibrium state. Due to this condition, the regenerative effect of the latch circuit takes over, amplifying the small voltage difference in a positive-feedback manner.

The next figure illustrates this process, showing the evolution of the signal on the bit line during both a read-1 and read-0 operation. Once triggered, the sense amplifier amplifies the slight initial difference ($\Delta V(1)$ or $\Delta V(0)$) to full voltage levels. The bit line corresponding to a read-1 is driven to V_{DD}, while for a read-0, it is pulled down to 0V. The complementary waveforms are developed on the $\overline{B}$ line.

The behavior of the sense amplifier ensures that the stored data is reliably read out while also initiating the necessary refresh operation to restore the original cell content. In the following sections, we will further analyze the temporal characteristics of this amplification process. Deriving a precise mathematical expression for the output signal of the sense amplifier, as shown in the figure, is a complex task. This requires employing large-signal, nonlinear models of the inverter voltage-transfer characteristics while also considering the impact of positive feedback. Instead of using this detailed approach, we will analyze the operation in a semi-quantitative manner.

At the moment when the sense amplifier is activated, both inverters within the circuit are operating in the transition region, around $V_{DD}/2$. In small-signal analysis, each inverter can be modeled using its transconductance parameters, g_{m1} and g_{m2}, which correspond to transistors Q_1 and Q_2, respectively. These values are evaluated under the condition of an input bias of $V_{DD}/2$. If a small signal component v_s is superimposed on this bias at the input of an inverter, it results in an incremental output current given by:

$$I_{\text{out}} = g_m v_s$$

This current is directed to either of the capacitors, denoted as C_g or C_j, within the circuit. The voltage developed across the capacitor is then fed back into the other inverter, where it is further amplified by the transconductance parameter G_m.

This feedback mechanism creates a regenerative process in which the current contributes to an output current feeding the opposite capacitor. Due to the positive feedback, the signal in the loop continuously amplifies, leading to an exponential rise or decay in the voltages v_B and $v_{\overline{B}}$. The rate of this exponential change is determined by the time constant (C_g/G_m) or (C_j/G_m), under the assumption that $C_g = C_j$.

For a read-1 operation, the voltage at the bit line v_B is given by:

$$v_B = \frac{V_{DD}}{2} + \Delta V(1) e^{G_m t/C_g}, \quad v_B \leq V_{DD}$$

Similarly, in a read-0 operation, the bit line voltage follows the equation:

$$v_B = \frac{V_{DD}}{2} - \Delta V(0) e^{G_m t/C_g}, \quad v_B \geq 0$$

These expressions, derived under the assumption of small-signal behavior, accurately describe the initial exponential evolution of v_B. However, they become less precise when the signal approaches extreme values near 0 or V_{DD}. Nevertheless, these approximations are useful in estimating the time required for the sense amplifier to drive the voltage to a sufficient level for reliable data retrieval.

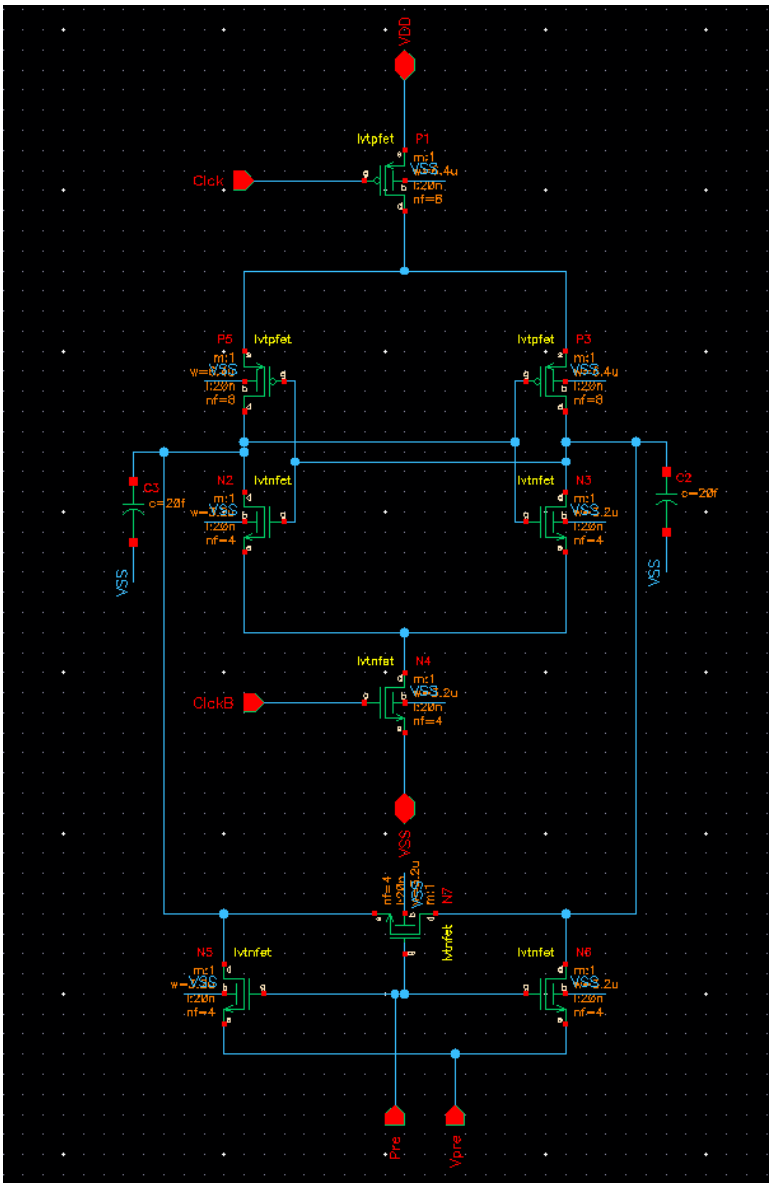


Figure 4.1: Sense Amplifier Circuit.

The results of the of the simulations are presented below.

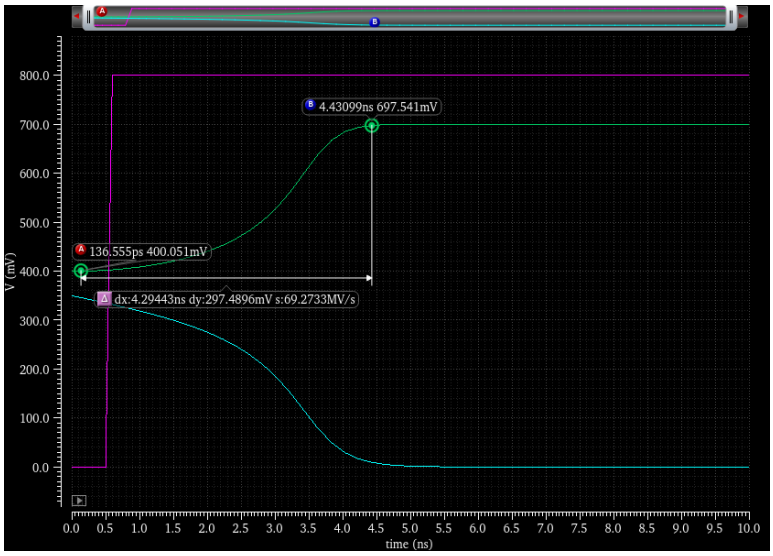


Figure 4.2: Timing that the sense amplifier needs to charge the bitlines.

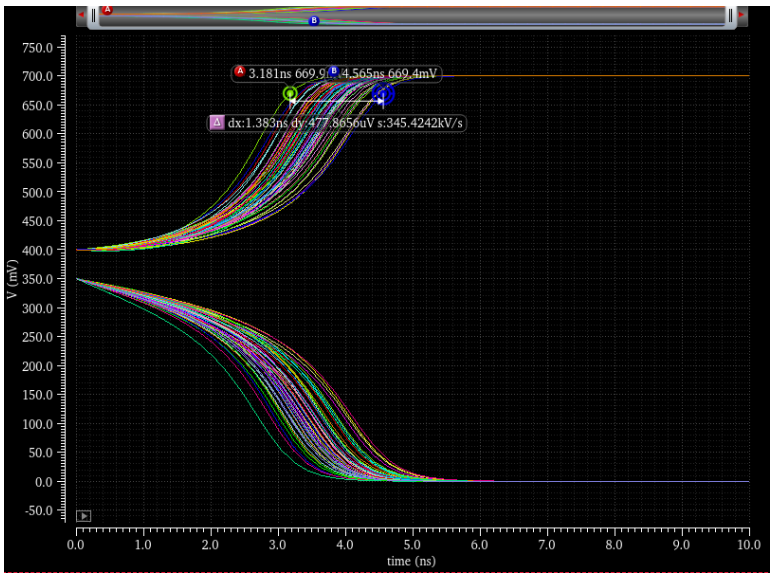


Figure 4.3: Monte Carlo Analysis of the timings that the Sense Amplifier needs to charge the bitlines.

First, we observe that the bitlines require a certain amount of time to recharge, typically around 4 to 5. During this period, the charge on the bitline remains between $\frac{V_{dd}}{2}$ and either ground or V_{dd}. A *Monte Carlo*

analysis reveals that these recharge times exhibit variations of up to 1, implying that each sense amplifier may require a slightly different amount of time to fully restore the bitline charge.

Furthermore, the charge stored in the capacitor may also vary due to *process variations* and *mismatch effects*, leading to deviations in the sensing process. The memory controller is typically programmed to fetch instructions based on standardized timing constraints to ensure that all variations are accommodated. However, by leveraging the analog behavior of the sense amplifiers and intentionally violating these timing constraints, we can exploit in-memory operations without requiring any modifications to the hardware. By leveraging the **DRAM Bender** framework, we can deliberately violate timing constraints, reducing the number of nanoseconds required for instruction fetching. By modifying these parameters, we enable a range of in-memory operations, including **Row Copy**, **logical AND**, **logical OR**, **logical NOT**, and **Multiple Row Activation**, without requiring any changes to the underlying hardware architecture.

4.3 Row Copy

Our analysis begins with the command sequence depicted in next figure, which consists of three DRAM commands: `ACTIVATE(R1)`, `PRECHARGE`, and `ACTIVATE(R2)`. These commands target two distinct rows, R_1 and R_2, within the same DRAM bank. The timing intervals between consecutive commands are denoted as T_1 and T_2, which are regulated by the number of idle cycles introduced between them.

Under standard operating conditions, the timing constraints dictate that T_1 must be greater than t_{RAS}, and T_2 must be greater than t_{RP}. Following these constraints ensures that row R_1 is properly opened, precharged, and subsequently replaced by row R_2, without altering the stored data. However, by strategically reducing the timing intervals T_1 and T_2 beyond specification limits, we can induce a non-nominal operational state in the DRAM, enabling the realization of alternative in-memory operations [8] [6] [1].

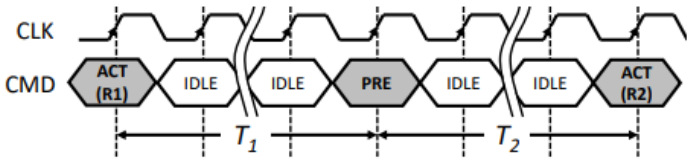


Figure 4.4: Command Sequence for in memory operations. [1]

The **row copy** operation is the simplest among the three studied in-memory operations, enabling the duplication of data from one row, R_1, to another, R_2. This operation is performed by first loading the contents of R_1 into the bitline and subsequently overwriting R_2 using the sense amplifier.

To achieve row copy, we reduce the timing interval T_2, as shown in previous figure, to a value significantly shorter than t_{RP}, thereby causing the second `ACTIVATE` command to prematurely interrupt the ongoing `PRECHARGE` command. The only constraint on T_1 is that it must be sufficiently long to allow the sense amplifier to fully drive the bitline with the data stored in R_1.

It is important to note that T_2 must not be reduced excessively, as doing so could inadvertently allow another row to be activated. While this effect can be exploited to implement more complex in-memory operations, such as **logical AND** and **logical OR**, it is destructive in the context of row copy.

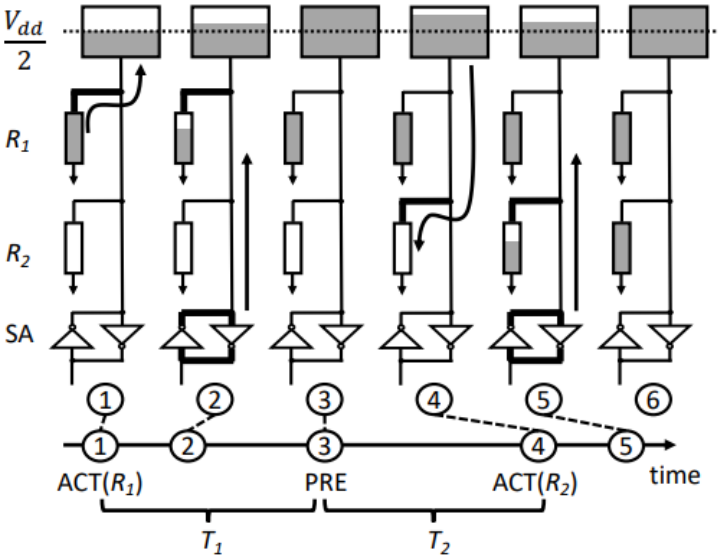


Figure 4.5: Timeline for a single bit of a column in a row copy operation.The data is loaded to the bitline R1 and over-writes R2. [1]

In this study, we analyzed multiple subarrays and demonstrated that the **row copy** operation can achieve a success rate of up to 98. The observed errors are more prevalent when copying rows that are located further apart. This phenomenon arises due to the increased resistance and capacitance of the transmission lines, which attenuate the charge available to replenish the cell capacitors, thereby reducing the effectiveness of the operation. The result of the row copy is shown in the next figure. Each subarray consists of a total of 512 rows. The results indicate the presence of three distinct groups exhibiting similar success rates.

- **Group 1:** Rows separated by at most 50 positions achieve a success rate of up to 98%.
- **Group 2:** Rows with distances between 50 and 230 exhibit a success rate of approximately 92%.
- **Group 3:** For rows spaced beyond 230, the success rate decreases to around 89%.

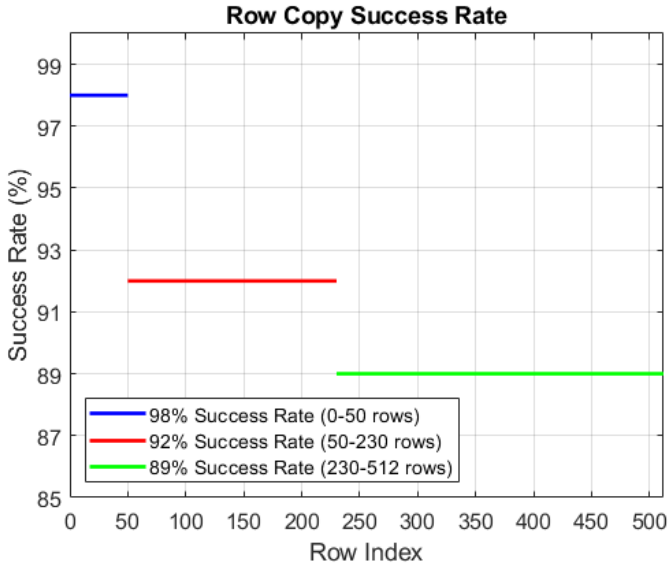


Figure 4.6: Row Copy Success the regarding the distance of the activated Row.

4.4 Logical AND/OR

By further decreasing the timing intervals in the command sequence we successfully activated three different rows simultaneously. This

phenomenon enables the exploitation of charge sharing to perform in-place logical operations. To achieve this, one of the three activated rows is preloaded with either all-zeros or all-ones. Upon executing the command sequence, the logical **AND** or **OR** operation between the other two rows naturally emerges across all three rows.

Specifically, to implement the logical **AND/OR** operations, both T_1 and T_2 are minimized, ensuring that the commands `ACTIVATE(R1)`, `PRECHARGE`, and `ACTIVATE(R2)` are executed in immediate succession without idle cycles. When the addresses of the two source rows, R_1 and R_2, are chosen appropriately, the command sequence implicitly activates a third row, R_3. The timing diagrams for the **AND** and **OR** operations are provided in the next figures.

In these figures, the rows are depicted according to their physical arrangement, with addresses 0, 1, and 2 from top to bottom. Notably, both operations utilize identical command sequences and timing constraints. The only distinguishing factors are:

- The specific rows designated for storing the operand values, highlighted with rectangles in the figures.
- The operation-selecting constant, which is stored in the remaining row.

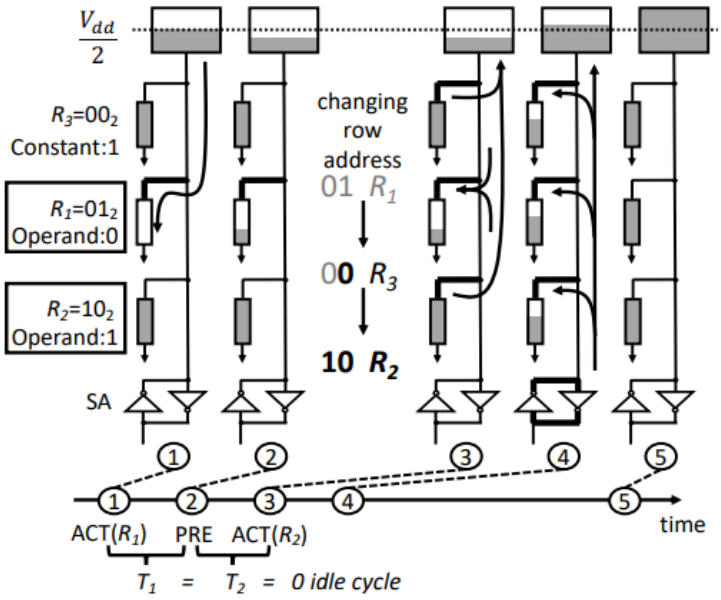


Figure 4.7: Logical OR. [1]

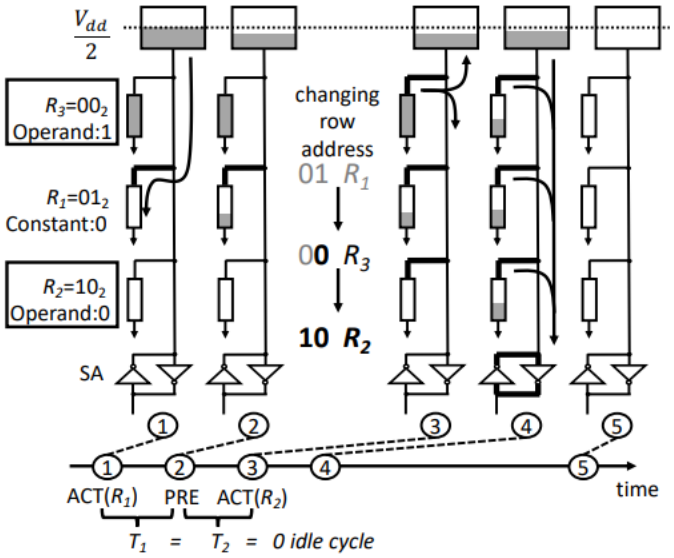


Figure 4.8: Logical AND. [1]

As in the previous case, **Step 1** in previous figures begins with the first `ACTIVATE` command, which opens row R_1. However, unlike the row copy procedure, this activation is immediately followed by a `PRECHARGE` command in **Step 2**, effectively interrupting the activation process. The short timing interval T_1 ensures that the sense amplifier remains disabled. This is a crucial condition, as enabling the sense amplifier prematurely would restore the charge of R_1 to the bitline, potentially overwriting data in other rows.

In **Step 3**, the second `ACTIVATE` command is issued, interrupting the ongoing `PRECHARGE` operation. As outlined in **Section 2**, the `PRECHARGE` command typically serves two purposes: it deactivates the wordline to close the currently open row and drives the bitline to $V_{\mathrm{dd}}/2$. However, by minimizing the timing interval T_2, both of these actions can be prevented. Consequently, the initial row R_1 remains open while the second `ACTIVATE` command updates the row address from R_1 to R_2. During this transition, an intermediate row R_3 momentarily appears on the row address bus. Since the `PRECHARGE` command was interrupted early, the bank remains in a state where the wordline is still being set. This state causes the wordline to follow the intermediate row address, effectively opening R_3. As a result, by the end of **Step 3**, both R_2 and R_3 are activated, while R_1 remains open from the beginning.

Following this, in **Step 4**, charge sharing occurs, equalizing the voltage

across all three activated rows and the bitline. The resulting voltage level depends on the majority value stored in R_1, R_2, and R_3. The process of configuring logical **AND** and **OR** operations by preloading a designated operation-selecting constant row is discussed in subsequent sections.

Finally, in **Step 5**, the computed result is stored across all three rows. Similar to the row copy operation, logical **AND** and **OR** operations are restricted to rows within the same subarray, ensuring consistent charge sharing behavior.

Theoretically, if all three rows have identical cell capacitance and are activated simultaneously, the resulting operation would follow the majority function. Specifically, if at least two of the three rows contain a logic value of one, the output will also be one; otherwise, it will be zero. However, in practical scenarios, row R_1 is activated first, granting it a longer duration to influence the bitline. As a result, the three rows do not contribute equally to the final outcome.

In the next figure presents the experimentally derived truth table for all possible value combinations in rows R_1, R_2, and R_3. With the exception of the case where $R_1 = 1$, $R_2 = 0$, and $R_3 = 0$ (marked with an 'X' to denote an unpredictable result), all other input combinations yield the expected logical outcome. Based on these observations, we selectively use the robust combinations from next figure to define operand rows while designating the remaining row as a predefined constant for implementing logical **AND** and **OR** operations.

R_3 \ R_1R_2	00	01	10	11
0	0	0	X	1
1	0	1	1	1

Figure 4.9: Truth Table from Charge Sharing in 3 rows.[1]

By setting R_1 to a fixed logic zero, the truth table reduces to the region enclosed by the dotted circle, effectively performing a logical **AND** operation on the values stored in R_2 and R_3. Conversely, setting R_3 to a constant logic one reduces the truth table to the solid circle, thereby implementing a logical **OR** operation on R_1 and R_2.

Thus, in addition to precise control of timing intervals, an essential requirement for executing logical **AND** and **OR** operations is the presence

of a predefined constant zero or one. Before initiating an **AND** operation, row R_1 must be preloaded with all-zeros. Similarly, prior to executing an **OR** operation, row R_3 must be initialized with all-ones. During the DRAM setup phase, these constant values are stored in two dedicated rows within each subarray, ensuring efficient execution of in-memory logic operations.

Having established the row copy and logical AND/OR operations, a crucial missing element for arbitrary computations is the NOT operation. To address this limitation, prior research has proposed the *in situ* implementation of the NOT operation in DRAM [6] [7] [8].

We incorporate both the regular and negated versions of a value, enabling more complex computations. In this framework, every variable consists of two parts: one representing the original value and the other its negation. The simultaneous availability of both forms facilitates the construction of universal logic functions such as NAND, which can be leveraged to execute arbitrary computations. Equations (1) to (5) illustrate various possible logical operations, where the right-hand side of each equation is formulated exclusively using AND and OR gates.

To ensure arbitrary computations at any execution point, additional steps are required to generate the negated pair. For instance, computing the XOR operation between two values, A and B, using only AND/OR gates, necessitates three fundamental operations: $A \wedge \bar{B}$, $A \vee B$, and the logical OR of the previous two results. Similarly, for computing XNOR, three additional operations—$\bar{A} \vee B$, $A \vee \bar{B}$, and the logical AND of the prior two—are required. This ensures that computations maintain the pairwise value structure, extending the framework to more complex operations.

Adopting this pairwise approach increases memory usage and effectively doubles the number of operations. However, the benefits in computational capabilities outweigh these costs, as the inherent massive parallelism of row-wise DRAM execution mitigates performance overheads. Moreover, not all negated intermediate values need explicit computation. Through logical minimization, redundant steps can be eliminated. The resulting logical functions are expressed as follows:

$$\text{NOT: } \neg(A, \bar{A}) = (\bar{A}, A) \tag{1}$$
$$\text{AND: } (A, \bar{A}) \wedge (B, \bar{B}) = (A \wedge B, A \vee B) \tag{2}$$
$$\text{OR: } (A, \bar{A}) \vee (B, \bar{B}) = (A \vee B, A \wedge B) \tag{3}$$
$$\text{NAND: } \neg((A, \bar{A}) \wedge (B, \bar{B})) = (\bar{A} \vee \bar{B}, A \wedge B) \tag{4}$$
$$\text{XOR: } (A, \bar{A}) \oplus (B, \bar{B}) = ((A \wedge \bar{B}) \vee (\bar{A} \wedge B), (A \vee B) \wedge (\bar{A} \vee \bar{B})) \tag{5}$$

This formulation ensures that logical operations remain efficient while maintaining compatibility with the DRAM-based computation framework.

4.5 Fraction Operation

However, a limitation in DDR4 memory arises when attempting to activate three rows simultaneously using the ACT-PRE-ACT command sequence. Our investigation of DRAM behavior revealed that instead of three, the system inherently enables the activation of 4, 8, or 16 rows at the same time [9].

To implement the previously discussed operations, it is essential to neutralize the effect of one of the activated rows during the charge-sharing process. This is accomplished by employing a specialized operation that adjusts the charge stored in the cell capacitor to approximately $V_{dd}/2$, effectively negating its influence on subsequent computations.

The Frac operation [2] enables the storage of fractional values across an entire row. To implement Frac, we leverage two fundamental DRAM commands: ACTIVATE and PRECHARGE. These commands must be executed consecutively, without any idle cycles in between. At a high level, the objective is to utilize the PRECHARGE command to disrupt the row activation process, thereby preventing the sense amplifier from being enabled.

The next figure illustrates the voltage behavior of both the bit-line and the DRAM cell during the Frac operation. Initially, in step 1, a PRECHARGE command is issued to set the bit-line voltage to $V_{dd}/2$. This command first ensures that any previously activated row is closed. Consequently, the bit-line voltage remains unchanged for at least one memory cycle. The initial voltage of the cell capacitor can be either V_{dd} or 0; for this example, we assume it starts at V_{dd}.

In step 2, the Frac operation begins by issuing an ACTIVATE command to the target row, which raises the word-line and connects the cell capacitor

to the bit-line. As charge sharing occurs, both the cell and bit-line reach an equilibrium voltage slightly above $V_{dd}/2$. Since the bit-line capacitance is significantly larger than that of the cell, the final voltage is closer to the initial bit-line voltage [**?**].

Next, in step 3, a PRECHARGE command is issued to interrupt the activation process before the sense amplifier engages. As a result, the cell capacitor is disconnected from the bit-line while holding a fractional voltage—neither V_{dd} nor 0, but slightly above $V_{dd}/2$. Given the necessity of waiting for the PRECHARGE command to complete, the total latency of a single Frac operation amounts to 7 memory cycles (comprising two command cycles and five idle cycles).

With just one Frac operation, fractional values can be stored in DRAM cells within the same row. However, this stored voltage depends on the initial state of the cell. For columns where the initial voltage is 0, the resulting voltage will settle between $V_{dd}/2$ and 0. To improve accuracy—bringing the stored voltage closer to $V_{dd}/2$ and reducing dependence on the initial value—multiple Frac operations can be executed sequentially.

At step 4, after the completion of the previous PRECHARGE command, an additional Frac operation can be issued. Our findings indicate that the more Frac operations performed, the closer the resultant voltage approaches $V_{dd}/2$, ensuring greater consistency across the row, regardless of the initial voltage state. We analyze the impact of initial cell voltage and the number of Frac operations on the final voltage in Section V.

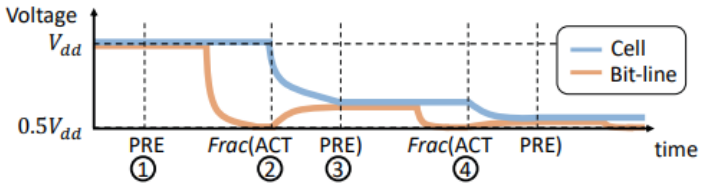


Figure 4.10: Voltage Level of the Capacitor and the bit line during Frac operation. [2]

To evaluate the success rate of logical AND/OR operations, we followed the sequence of steps below:

- **Identifying Simultaneously Activated Rows**
  We begin by identifying four rows that can be activated simultaneously. This is achieved using the command sequence `ACT-PRE-ACT`. We then write a predefined value to memory.

⇓

- **Storing a Fractional Voltage ($V_{dd}/2$)**
  In one of these four rows, we store a voltage level close to $V_{dd}/2$ using the Frac operation. This is done by issuing an `ACT` command, immediately followed by a `PRECHARGE` command, and then waiting for 5 cycles. This process is repeated three additional times, accumulating a total delay of 21 cycles.

  ⇓

- **Executing the Logical AND/OR Operation**
  Based on whether a logical AND or OR operation is desired, we store either '1' or '0' in the third row. The command sequence `ACT-PRE-ACT` is applied, followed by a 4-cycle wait for charge sharing. A final `PRECHARGE` command is issued before reading back the data to verify correctness.

  ⇓

- **Repeating the Experiment and Collecting Statistics**
  The experiment is repeated approximately 1000 times to gather statistically meaningful results.

4.6 Experimental Results

The results indicate the following success rates across the total of 65×2^{10} columns:

- **11% of the columns** achieved a success rate of **85%**.
- **66% of the columns** achieved a success rate of **62%**.
- **23% of the columns** achieved a success rate of **47%**.

These findings highlight a significant variation in the performance of different columns.

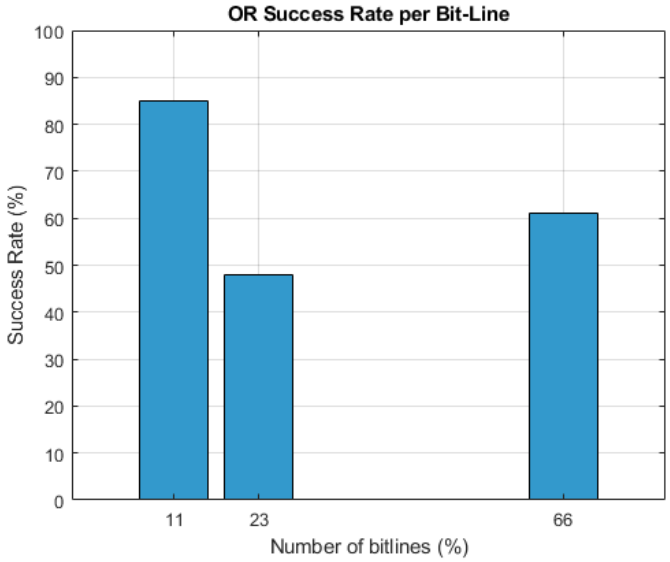


Figure 4.11: OR Success Rate.

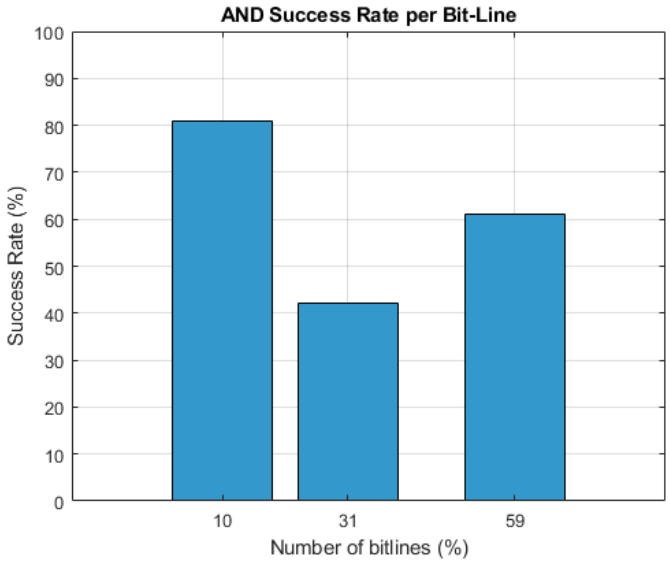


Figure 4.12: AND Success Rate.

The graphs above show that we can have 3 groups of bit lines. The first group is about the 10% of the columns (i.e 6500) have a success rate over 80% both in AND and OR logical gates. This means that they are quite

reliable and can produce the correct result in most cases. The experiment showed that the logical AND is less reliable than the logical OR as fewer columns can achieve rates up until 60%.

4.7 Full Adder

The circuit of the Full Adder is appeared below

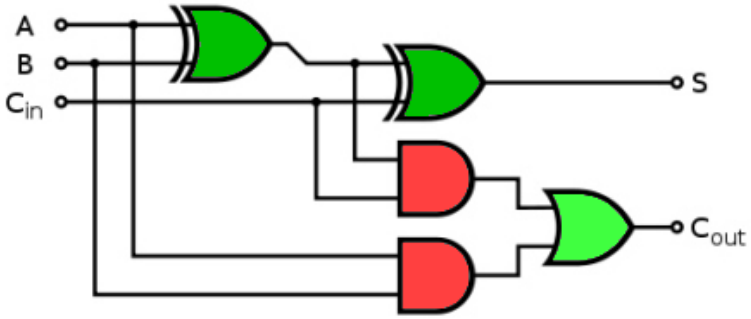


Figure 4.13: Full Adder.

To achieve the full addition of two single-bit numbers, we select the subarray that provides the highest success rate for logical AND and OR operations. The following figures illustrate the process of performing addition within the DRAM subarray. There are several constraints that impact our approach:

1. **Limited Subarray Size:** The selected subarray consists of 512 rows, and we cannot utilize rows from other subarrays.

2. **Negation Handling:** If an operation requires negation, we must execute the entire logical negated operation, as direct bit inversion is not feasible within the subarray.

3. **Bit-Serial Arithmetic:** The subarray does not support shifting operations for multi-bit numbers. Therefore, we implement a bit-serial arithmetic approach, where numbers are stored along the same bit-line but across different rows.

To enhance the efficiency of row copy operations, we strategically store the numbers in the middle region of the subarray. As demonstrated in previous figures, this placement leads to improved row copy performance

and overall computation reliability. Let us assume that the data is stored within the DRAM. Since all operations are performed between two bits, we need to activate four rows simultaneously. Depending on the required operation, we store a predefined value in an assisting cell: a logical **1** for OR operations and a logical **0** for AND operations.

We describe the execution of one such operation, as the remaining operations follow a similar approach. Given that there are a total of **nine operations** to be performed (with XOR consisting of two AND operations and one OR operation), we organize the computations into nine distinct groups. Each group consists of four rows, which are activated simultaneously using the `ACT-PRE-ACT` command sequence.

Upon closer examination, we observe that the **carry bit** from one operation will serve as the **carry-in** for the next stage of the full adder. Consequently, we also need to compute the negated version of the carry-out, which is given by the following Boolean expression:

$$(\neg(A \oplus B) + \neg C_{\text{in}}) \cdot (\neg A + \neg B)$$

To execute the required operations efficiently, we organize the computation into **15 groups**, each consisting of **4 rows**. We describe the execution of a single logical **AND** operation, noting that the procedure for logical **OR** follows the same structure.

To preserve the data, we first issue a **row copy** command within one of the groups. This process is repeated three times—twice for the operand rows and once for the row that determines the operation (which, in the case of AND, is initialized to **0**). Since a row copy operation requires **9 memory cycles**, the total cost for a full row clone operation amounts to **27 cycles**.

Next, we execute a **fractional operation** on the last row of the group, which has a latency of **7 cycles**. Following this, we apply the `ACT-PRE-ACT` sequence to perform the AND operation, requiring an additional **8 memory cycles**.

Once the AND operation is completed, the result is copied to the destination row of another group with a latency of **9 memory cycles**. Similarly, the result of the **SUM** is transferred to the designated row for final storage, also requiring **9 cycles**.

In total, each group executes one operation with a latency of:

$$27 + 7 + 8 + 9 = 51 \text{ memory cycles}$$

To perform these operations across all **15 groups**, the total latency required is:

$$15 \times 51 = 765 \text{ memory cycles}$$

However, since the memory speed is **four times faster** than the FPGA used in our setup, the effective latency in FPGA cycles is:

$$\frac{744}{4} = 192 \text{ FPGA cycles}$$

Thus, executing the full addition of single-bit data requires approximately **186 FPGA cycles**.

We evaluate the success rate of addition for different bit-widths, including **1-bit**, **4-bit**, **8-bit**, and **16-bit** operations. The experimental results indicate the following success rates:

- **1-bit addition:** 75% success rate.
- **4-bit addition:** 67% success rate.
- **8-bit addition:** 47% success rate.
- **16-bit addition:** 28% success rate.

These results highlight a decreasing trend in success rate as the bit-width increases, suggesting that larger bit-width operations introduce additional challenges in DRAM-based computation.

Although the reported success rates may initially seem low, our simulations indicate that when performing addition over multiple numbers, the occurrence of random bit errors within the operands tends to cancel out. As a result, the overall cumulative error remains below 10%

This phenomenon is illustrated in the following figure.

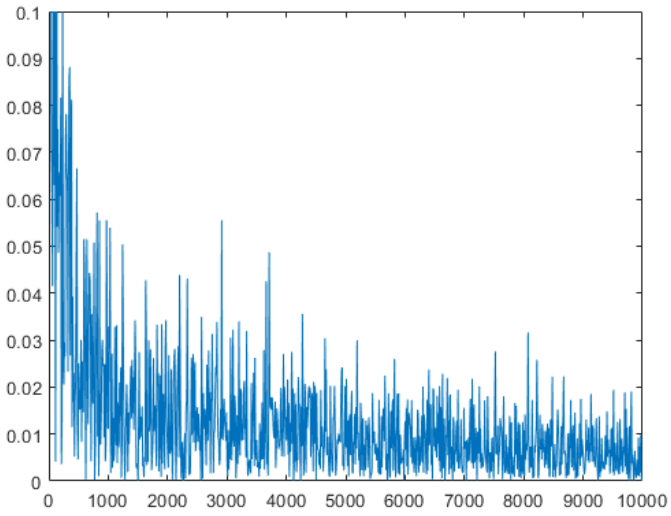


Figure 4.14: As the number of additions increases, the absolute error of erroneous numbers exhibits a decreasing trend. This suggests that random bit errors tend to partially cancel out, leading to a more stable overall result. Consequently, the cumulative error remains within acceptable limits, demonstrating the inherent error-mitigation effect in large-scale additions. .

The number of rows required for computation is a function of the bit length of the operands, as illustrated in the following figure.

For a full adder, we require **15 groups of 4 rows** dedicated solely to computation. Additionally, the number of rows must accommodate the bit length of both operands, the sum, and the carry-out.

Thus, the total number of rows follows a linear function given by:

$$\text{Total Rows} = 61 + 3 \times \text{Bit Length}$$

This relationship highlights the scalability of row usage as the bit length increases.

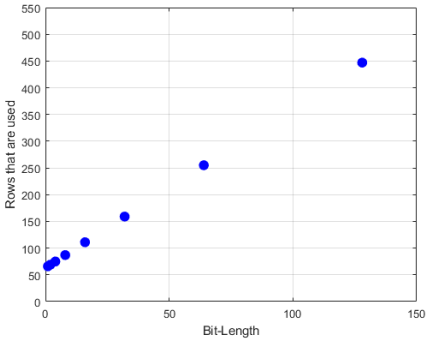


Figure 4.15: Rows occupied depending the bit length of the operators.

The number of operators that can fit in the subarray regarding their bit length is appeared below

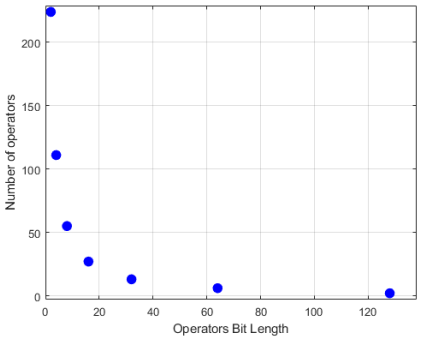


Figure 4.16: Number of operators in one subarray depending the bit length of the operators.

To evaluate the speedup achieved through in-memory addition, we consider the summation of **1024**, **10,240**,**102,400** numbers, assuming that each number is represented with an **8-bit** length. The evaluation is conducted under two different computational scenarios.

In the first scenario, all additions are performed within the **FPGA**, where each addition requires exactly one cycle. Consequently, the total number of cycles required is equal to the number of operands being summed, namely **1024** and **10,240** cycles, respectively.

In the second scenario, computation is carried out within **DRAM** using an in-memory addition approach. The numbers are divided into groups that fit within a single subarray, enabling parallel execution. This approach significantly reduces the overall cycle count compared to

FPGA-based computation. The spped up is given from the following function

$$\frac{1024}{N} + 192 \times 8 \times N$$

The results of these two scenarios are presented in the following section.

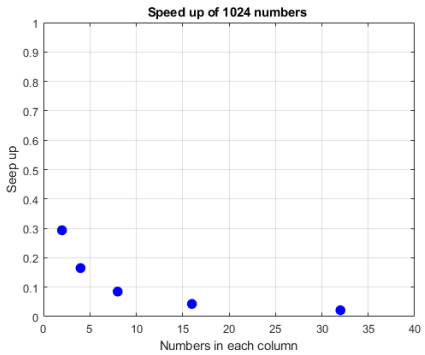


Figure 4.17: Speed up of the addition of 1024 numbers.

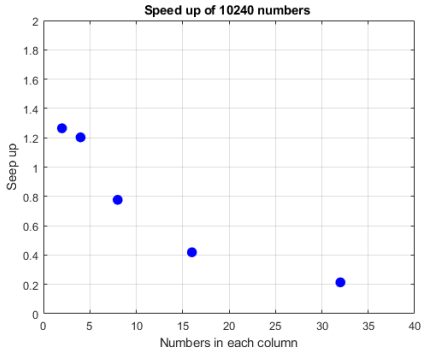


Figure 4.18: Speed up of the addition of 10240 numbers.

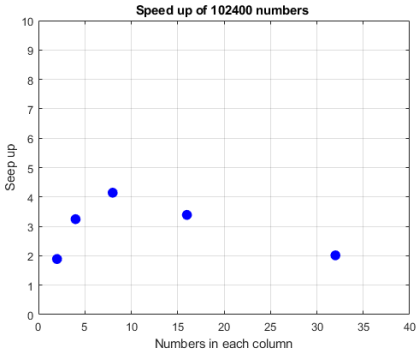


Figure 4.19: Speed up of the addition of 102400 numbers.

Chapter 5

Comments and future work

Our analysis indicates that for a relatively small number of operands, such as **1024**, no significant speedup is observed. However, as the number of operands increases, we begin to observe measurable improvements in performance.

When utilizing only **two operands per bitline**, a moderate speedup is achieved. For larger datasets, the speedup exceeds **4×**, reaching its peak efficiency when **eight operands per bitline** are utilized. This demonstrates the scalability of in-memory addition, where higher operand density per bitline leads to greater computational efficiency.

Compute-in-Memory (CIM) is an emerging paradigm designed to mitigate the memory bottleneck in conventional von Neumann architectures. Instead of continuously transferring data between memory and processing units, CIM enables computations to be performed directly within memory cells, thereby reducing latency and power consumption. This approach has broad applications across multiple domains.

One of the most prominent applications of CIM is in artificial intelligence and machine learning, where matrix-vector multiplications are fundamental to deep learning workloads. By executing these operations directly within memory, CIM accelerates neural network inference and reduces energy consumption, making it particularly useful for edge computing and IoT devices. Additionally, CIM is well-suited for cryptographic computations, including hashing and encryption, where high-speed bitwise operations are required.

Bibliography

[1] F. Gao, G. Tziantzioulis, and D. Wentzlaf, "Computedram: In-memory compute using off-the-shelf drams," *IEEE/ACM International Symposium on Microarchitecture*, 2019.

[2] F. Gao, G. Tziantzioulis, and D. Wentzlaf, "Fracdram: Fractional values in off-the-shelf dram," *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[3] S. A. S and S. K. C, *Microelectronic Circuits*. Oxford university press, 2004.

[4] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. 2011.

[5] A. Olgun, H. H. A., G. Yağlıkçı, and O. Mutlu, "Dram bender: An extensible and versatile fpga-based infrastructure to easily test state-of-the-art dram chips," *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[6] A. Olgun, J. G. Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu, "Pidram: A holistic end-to-end fpga-based framework for processing-in-dram," *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[7] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, J. Gómez-Luna, M. Alser, O. Mutlu, and S. Ghose, "SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM," 2024.

[8] I. E. Yuksel, Y. C. Tugrul, A. Olgun, F. N. Bostanci, A. G. Yaglikci, G. F. Oliveira, H. Luo, J. Gómez-Luna, M. Sadrosadati, and O. Mutlu, "Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis," 2024.

[9] I. E. Yuksel, Y. C. Tugrul, F. N. Bostanci, A. G. Yaglikci, A. Olgun, G. F. Oliveira, M. Soysal, H. Luo, J. G. Luna, M. Sadrosadati, and O. Mutlu, "PULSAR: Simultaneous Many-Row Activation for Reliable and High-Performance Computing in Off-the-Shelf DRAM Chips," 2024.