Τ.Ε.Ι. ΚΡΗΤΗΣ

ΤΜΗΜΑ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΠΟΛΥΜΕΣΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη συστήματος επαύξησης αξιοπιστίας για πολυπύρηνα ενσωματωμένα συστήματα σε πραγματικό χρόνο

Χριστοφοράκης Ιωάννης
Α.Μ. 2463

Επιβλέπων Καθηγητής:
Κορνάρος Γεώργιος

24 ΙΟΥΛΙΟΥ 2012

Ευχαριστίες

Ιδιαίτερα θερμές ευχαριστίες προς τον επιβλέποντα καθηγητή μου κ. Γεώργιο Κορνάρο, που χωρίς την ολόπλευρη στήριξη του, καθ' όλη τη διάρκεια εκπόνησης της εργασίας, δεν θα είχε καταστεί δυνατή η ολοκλήρωση της.

Abstract

Complex multi-core embedded systems are ever more prevalent in modern microelectronic products. The development of such systems creates new possibilities for evolution in aerospace, medicine, communications and consumer eras. Improving reliable performance of embedded systems is therefore increasingly important and challenging as more complex applications are developed on these systems.

This work focuses on increasing the ability of an embedded system to self-identify malfunctioning in the running applications, rapid repair and re-execute in real time. A light-weight methodology is proposed to ensure reliability through monitoring, debugging and real-time replacement of the failing sections of software algorithms in an embedded multi-core system.

The principles of this work are demonstrated in a prototyped platform and measurements on an embedded image processing application validate.

Σύνοψη

Θέμα της πτυχιακής εργασίας αυτής είναι η ανάπτυξη υποστηρικτικού λογισμικού και ειδικού υποσυστήματος hardware για ενσωματωμένα συστήματα ώστε να ενισχυθεί η δυνατότητα ανεκτικότητας σε σφάλματα καθώς και η αξιοπιστία τους συστήματος και ειδικά όταν εκτελούνται εφαρμογές πραγματικού χρόνου

Πολύπλοκα πολυπύρηνα ενσωματωμένα συστήματα είναι όλο και πιο διαδομένα σε σύγχρονα προϊόντα μικροηλεκτρονικής. Η ανάπτυξη αυτών των συστημάτων δημιουργεί νέες δυνατότητες για την εξέλιξη σε πόλους τομείς της επιστήμης και της τεχνολογίας. Η βελτίωση της αξιόπιστης απόδοσης των ενσωματωμένων συστημάτων είναι συνεπώς, πολύ σημαντική για τις όλο και περισσότερο σύνθετες εφαρμογές που αναπτύσσονται σε αυτά.

Η εφαρμογή της εργασίας μας πραγματοποιείται σε μια πρωτότυπη πλατφόρμα και επικυρώνεται με μετρήσεις και σχετικά συμπεράσματα σε μια ενσωματωμένη εφαρμογή επεξεργασίας εικόνας .

Πίνακας Περιεχομένων

Κατάλογος Σχημάτων

Κατάλογος των Γραφημάτων

1. Εισαγωγή

Σε αυτήν την εργασία αναπτύξαμε ένα σύστημα παρακολούθησης και επιδιόρθωσης πραγματικού χρόνου(real time monitoring) σε πολυπύρηνο ενσωματωμένο σύστημα με FPGA με στόχο την επαύξηση της αξιοπιστίας του συστήματος. Στην μεθοδολογία που επιλέξαμε γι αυτό το σκοπό χρειάστηκε να αναπτυχτεί το απαραίτητο λογισμικό που υποστήριζε την παρακολούθηση της εφαρμογής του ενός πυρήνα από τον άλλο(σύστημα δύο πυρήνων), καθώς και ο σχεδιασμός ενός block υλικού που είναι υπεύθυνο για την επιδιόρθωση και αντικατάσταση, μέρους με αναγνωρισμένο fault, της ανωτέρω εφαρμογής, όπως επίσης και της επανεκκίνησης του πυρήνα που βρίσκεται εγκατεστημένη η εφαρμογή. Ο σχεδιασμός υλικού έγινε με χρήση της γλώσσας περιγραφής υλικού VHDL, ενώ το απαραίτητο λογισμικό καθώς και η εφαρμογή έγιναν με χρήση της γλώσσας προγραμματισμού C και βιβλιοθηκών της Xilinx.

Το σύστημα μας όπως ήδη σημειώσαμε έχει εγκατεστημένους δύο πυρήνες. Ο ένας είναι ο πυρήνας πάνω στον οποίο εκτελείται ένας επιλεγμένος αλγόριθμος και ο δεύτερος είναι εκείνος που επιφορτίζεται με τον ρόλο του παρακολουθητή(monitor core). Ο παρακολουθητής ελέγχει το κατά πόσο η λειτουργία του αλγορίθμου βρίσκεται εντός του συνολικού Μέσου Χρόνου εκτέλεσης και αν όχι ποιο είναι το στάδιο(stage) ή τα στάδια(stages) που βρίσκονται εκτός προβλεπόμενου χρόνου εκτέλεσης. Ο έλεγχος των σταδίων γίνεται από κοινόχρηστες σημαίες(flags) –σημεία ελέγχου έναρξης/λήξης σταδίων εντός κώδικα του αλγορίθμου-. Η εφαρμογή που επιλέχτηκε για έλεγχο και επιδιόρθωση είναι ο δυναμικός αλγόριθμος canny. Ο αλγόριθμος αυτός είναι ανιχνευτής ακμών μιας εικόνας και χρησιμοποιείται ευρύτατα σε εφαρμογές επεξεργασίας εικόνων. Οι επεμβάσεις μας στον κώδικα του αλγορίθμου είχαν να κάνουν κυρίως με την τοποθέτηση των σημαιών ελέγχου και την αποθήκευση των δεδομένων του(εικόνα επεξεργασίας) στην τοπική μνήμη του επεξεργαστή μετά από διαμερισμό σε μικρότερες εικόνες.

Η λειτουργικότητα του συστήματος μας επιτρέπει στον χρήστη τόσο την σε πραγματικό χρόνο παρακολούθηση και επιδιόρθωση του αλγορίθμου, όσο και την αντικατάσταση του σταδίου ή των σταδίων που βρέθηκαν εκτός προσδοκώμενου χρόνου εκτέλεσης. Ενώ έχει τη δυνατότητα τροποποιήσεις των δεδομένων εισαγωγής καθώς ακόμη και την σε πραγματικό χρόνο δοκιμή διαφόρων παραλλαγών της ίδιας εφαρμογής. Τέλος του παρέχεται η δυνατότητα επανεκκίνησης του συστήματος και ολικής αντικατάστασης του κώδικα της εφαρμογής.

Η αναπτυξιακή πλακέτα που χρησιμοποιήθηκε για την ανάπτυξη της εργασίας είναι η ML405 της Xilinx, που υποστηρίζει Virtex 4 FPGA. Το σύστημα αυτό σχεδιάστηκε και προγραμματίστηκε στο περιβάλλον της Xilinx, ISE Design Suite 12.1 – EDK 12.1. Για προσημείωση της λειτουργίας του block υλικού χρησιμοποιήσαμε τον Modelsim 6.5b από την Mentor Graphics.

1.1 Κίνητρο για τη διεξαγωγή της εργασίας

Σήμερα, τα ενσωματωμένα συστήματα με real-time εφαρμογές έχουν καταστεί κεντρικό κομμάτι της ζωής μας. Σε αντίθεση από τα γενικά συστήματα υπολογιστή, σε πραγματικό χρόνο το σύστημα θεωρείται ότι λειτουργεί σωστά μόνο αν επιστρέψει το σωστό αποτέλεσμα εντός των χρονικών περιορισμών που έχουν οριστεί [1][2]. Η αξιόπιστη λειτουργία των συστημάτων πραγματικού χρόνου είναι πρωταρχικής σημασίας για τα εκατομμύρια των χρηστών που εξαρτώνται από τα συστήματα αυτά καθημερινά. Για να εξασφαλιστεί η λειτουργία ενός αξιόπιστου συστήματος έχουν αναπτυχθεί αρκετά έργα που βασίζονται σε τεχνικές υλικού ή λογισμικού. θα αναλυθούν στην επόμενη ενότητα, μαζί με τις διαφορές, τα πλεονεκτήματα τους και τα μειονεκτήματα των δύο αυτών κατηγοριών. Συγχρόνως, τα ενσωματωμένα συστήματα με multi-core εφαρμογές είναι οι πρώτες επιλογές για πολύπλοκο

και ταυτόχρονα αξιόπιστο σχεδιασμό. Η Multi-core επεξεργασία αναγνωρίζεται ως βασική συνιστώσα για συνεχή βελτίωση των επιδόσεων και για τα ενσωματωμένα συστήματα είναι μεγάλης οικονομικής σημασίας η συγκεκριμένη επιλογή. Να σημειώσουμε πως ενσωματωμένες εφαρμογές συναντάμε σε καταναλωτικές ηλεκτρονικές συσκευές, σε επεξεργασία σήματος, στον έλεγχο του αυτοκινήτου, στον αυτόματο πιλότο αεροσκαφών, και ούτω καθεξής.

Η κεντρική ιδέα της δουλειάς μας είναι η επίτευξη αυξημένης αξιοπιστίας για εφαρμογές ενσωματωμένων συστημάτων πολλαπλών πυρήνων που η επιτυχής λειτουργία τους εξαρτάται από τον χρόνο. Παρόμοιες προσπάθειες έχουν γίνει από διάφορες ερευνητικές ομάδες, ορισμένες με επιτυχία, όπως η AUTOSAR αρχιτεκτονική για την ασφάλεια και την προστασία εφαρμογών [6], ή ένα αυτοελεγχόμενο Hardware Journal για την ανεκτικότητα σε σφάλματα(fault-tolerant) της αρχιτεκτονικής του επεξεργαστή [4], και πολλές άλλες. Υπάρχουν νέα προβλήματα που έρχονται να αντιμετωπίσουν οι σύγχρονες τεχνικές για την παρακολούθηση σε πραγματικό χρόνο ή οι γενικές τεχνικές με στόχο την επαύξηση της αξιοπιστίας των ενσωματωμένων συστημάτων, όπως η μεταβλητότητα της διαδικασίας ή παροδικά σφάλματα που περιγράφεται στο [3].

Εμείς εστιάζουμε τις προσπάθειές μας ώστε να επιτραπεί ο σχεδιασμός για την υποστήριξη των γενικών εφαρμογών, αλγορίθμων, η οποία σε κάθε περίπτωση να συνοδεύεται από κατάλληλο σχεδιασμό του υλικού. Εμείς δεν θέλουμε το σύστημα μας να υποστηρίζει μόνο fault-tolerant ή self-control, θέλουμε ένα σύστημα που να έχει την ευελιξία να αναγνωρίσει το σφάλμα, το soft-error και θα μπορέσει να το προσπεράσει ή να την αντικαταστήσει με αξιοπιστία και ευκολία σε πραγματικό χρόνο.

1.2 Σκοπός και στόχοι Εργασίας

Οι στόχοι αυτής της εργασίας είναι οι εξής:

- Ανάπτυξη ενός συστήματος συνδυασμένης μεθοδολογίας για την επίτευξη αυξημένης αξιοπιστίας(reliability) σε ένα πολυπύρηνο ενσωματωμένο σύστημα, σε πραγματικό χρόνο.
- Δημιουργία κατάλληλου software για την υποστήριξη του real-time monitoring και checking
- Σχεδιασμός κατάλληλου hardware για την υποστήριξη και υλοποίηση της fault-tolerance διαδικασίας.
- Έλεγχος λειτουργικότητας δυναμικού αλγόριθμου στο ανεπτυγμένο πολυπύρηνο σύστημα.

1.3 Δομή Εργασίας

Το υπόλοιπο αυτής της εργασίας είναι οργανωμένο ως εξής:

- Στο κεφάλαιο 2 δίνουμε μια εισαγωγή για το τι είναι τα ενσωματωμένα συστήματα, στα οποία εφαρμόζεται η εργασία μας, εξηγούμε την έννοια της αξιοπιστίας(reliability) ενσωματωμένων συστημάτων. Ακόμη κάνουμε εισαγωγή στην fault-tolerance ιδιότητα που παρέχουμε στο σύστημα μας. Τέλος δίνουμε μια περιγραφή της διαδικασίας του monitoring γενικά, ενώ εξηγούμε και τις βασικές μεθοδολογίες που χρησιμοποιούνται.
- Στο κεφάλαιο 3 κάνουμε αναφορά σε εργασίες που έχουν ήδη γίνει σχετικά με το αντικείμενο που εξετάζουμε.

- Στο κεφάλαιο 4 γίνεται μια γενική περιγραφή της μεθοδολογίας που χρησιμοποιούμε, αλλά και για ποιες άλλες μεθοδολογίες θα μπορούσαν να χρησιμοποιηθούν.

- Στο κεφάλαιο 5 γίνεται αναλυτική περιγραφή του HW/Αρχιτεκτονικής του συστήματος μας.

- Στο κεφάλαιο 6 παρουσιάζουμε αναλυτικά το SW/Λογισμικό που δημιουργήσαμε για να υποστηριχτεί η επιλεγμένη μεθοδολογία μας.

- Στο κεφάλαιο 7 παρουσιάζουμε τα αποτελέσματα που εξήχθησαν από τα πειράματα που διενεργήσαμε καθώς και τη γενικότερη αξιολόγηση του συστήματος μας.

- Τέλος η παρουσίαση της εργασίας ολοκληρώνεται στο κεφάλαιο 8 με τα τελικά συμπεράσματα και τις κατευθύνσεις για μελλοντική ανάπτυξη.

2. Εισαγωγή στα Ενσωματωμένα Συστήματα, στην ιδιότητα του fault-tolerance και στις μεθοδολογίες monitoring.

2.1 Τι είναι ενσωματωμένα συστήματα

Ένα ενσωματωμένο σύστημα είναι ένα υπολογιστικό σύστημα ειδικού σκοπού, σχεδιασμένο έτσι ώστε να εκτελεί μια ή και περισσότερες συναρτήσεις, συνήθως σε σταθερές πραγματικού χρόνου. Είναι συνήθως ενσωματωμένο σαν ένα μέρος μιας ολοκληρωμένης συσκευής, περιλαμβάνοντας hardware καθώς και μηχανικά μέρη. Σε αντίθεση, ένα υπολογιστικό σύστημα γενικού σκοπού, όπως ένας προσωπικός υπολογιστής, ανάλογα με τον προγραμματισμό που του έχει γίνει, μπορεί να εκτελέσει πολλά διαφορετικά καθήκοντα. Τα ενσωματωμένα συστήματα ελέγχουν πολλές από τις κοινές καθημερινές συσκευές που χρησιμοποιούμε σήμερα. Από τη στιγμή που ένα σύστημα είναι προσανατολισμένο σε συγκεκριμένα καθήκοντα, οι μηχανικοί σχεδίασης μπορούν να το βελτιστοποιήσουν, μειώνοντας το μέγεθος και το κόστος του.

2.1.1 Βασικά χαρακτηριστικά ενσωματωμένων συστημάτων

Υπάρχουν κάποια βασικά χαρακτηριστικά των ενσωματωμένων συστημάτων:

- Εκτελούν ειδικευμένες εφαρμογές
- Περιλαμβάνουν τουλάχιστον ένα προγραμματιζόμενο στοιχείο
- Αλληλεπιδρούν συνεχώς με το περιβάλλον τους
- Λειτουργούν σε πραγματικό χρόνο
- Πρέπει να ικανοποιούν πρόσθετους περιορισμούς που αφορούν κατανάλωση ενέργειας. Βάρος, μέγεθος, κόστος, αξιοπιστία, ασφάλεια.

2.1.2 Εφαρμογές και χρήση των ενσωματωμένων συστημάτων

• Μηχανήματα αναλήψεως (ATMs)

• Ναυσιπλοΐα, hardware/software συστημάτων ελέγχου τάσεων, συστήματα αεροναυπηγικής, πυραύλους, GPS

• Ελεγκτές κινητήρων και ABS αυτοκινήτων, συστήματα ελέγχου απόδοσης μηχανής (καθορισμό μίγματος, χρονισμό, κλπ.).

• Προϊόντα αυτοματισμού σπιτιού, όπως θερμοστάτες, κλιματιστικά, συστήματα συναγερμού

• Αριθμομηχανές

• Οικιακές συσκευές, όπως φούρνοι μικροκυμάτων, συσκευές DVD ,μαγνητόφωνα

• Ιατρικός εξοπλισμός

• Παιχνιδομηχανές

• Περιφερειακά υπολογιστών, όπως εκτυπωτές και routers

• Σε βιομηχανικά συστήματα, κυρίως σε ελεγκτές για χειρισμού των μηχανημάτων.

• Δημόσια Διοίκηση, όπως σε αυτόματα συστήματα στάθμευσης, συστήματα για κλήσεις σε παραβάτες κυκλοφορίας, στάθμευσης, κλπ

2.1.3 Παράγοντες για τη σχεδίαση Ενσωματωμένων Συστημάτων

Παραθέτουμε μερικές από τις σημαντικότερες παραμέτρους που έχουν σημασία στην σχεδίαση ενσωματωμένων συστημάτων. Οι παράμετροι είναι:

• Κόστος κατασκευής: αφορά την μεγαλύτερη κατηγορία των ενσωματωμένων συστημάτων σε όγκο πωλήσεων. Με δεδομένο ότι υπάρχει επεξεργαστής κάποιας μορφής (ακόμη και ειδικής κατασκευής VLSI σε κάποιες περιπτώσεις) σε συσκευές από ρολόγια χειρός, αριθμομηχανές, κινητά τηλέφωνα, κλιματιστικά, φούρνους μικροκυμάτων, τηλεοράσεις, ψυγεία, και κάθε είδους μαζικά παραγόμενες συσκευές χαμηλού κόστους απόκτησης, το κόστος του ηλεκτρονικού μέρους αυτών είναι σημαντικός παράγοντας για την επιτυχία ενός προϊόντος.

• Κατανάλωση ενέργειας: Όλες περίπου οι φορητές ενσωματωμένες μικροηλεκτρονικές συσκευές είναι ευαίσθητες στο ζήτημα της κατανάλωσης ενέργειας (π.χ. προσωπικός ψηφιακός βοηθός – PDA, Personal Digital Assistant).

• Υπολογιστική Ισχύς: Κατά περίπτωση απαιτείται και αντίστοιχη υπολογιστική ισχύς. Εντυπωσιακό παράδειγμα είναι το ότι ενσωματωμένα συστήματα παιχνιδιών όπως το Sony Playstation® και το Nintendo® έχουν ιδιαίτερα ισχυρούς επεξεργαστές με ειδικές δυνατότητες για γρήγορα γραφικά.

• Αξιοπιστία: Κάποιες εφαρμογές έχουν ιδιαίτερα υψηλές απαιτήσεις αξιοπιστίας λόγω είτε της αδυναμίας πρόσβασης στην συσκευή κατά την λειτουργία (π.χ. κάψουλες διαστημικής εξερεύνησης) είτε λόγω κινδύνου απώλειας ανθρώπινης ζωής σε περίπτωση αστοχίας (π.χ. ηλεκτρονικά αεροπλάνων).

• Ευελιξία κατά την Χρήση: Όταν μία εταιρία κατασκευής μεταγωγέων παραδίδει ένα προϊόν (π.χ. ένα δρομολογητή – router), αυτός έχει κάποιες ενσωματωμένες δυνατότητες, π.χ. εξέταση πακέτων προερχομένων από προγραμματιζόμενες διευθύνσεις που καταδεικνύουν ανεπιθύμητα πακέτα όπως spam. Κατά την λειτουργία ενδέχεται να αλλάξουν τα χαρακτηριστικά τέτοιων πακέτων, είναι επομένως απαραίτητη η αναβάθμιση στο πεδίο τόσο του λογισμικού του συστήματος, όσο και κάποιων χαρακτηριστικών του υλικού (hardware) αυτού και του ενδιάμεσου επιπέδου firmware.

• Λειτουργία σε Πραγματικό Χρόνο: Είναι προφανές πως αν τα ηλεκτρονικά ενός αεροπλάνου δεν ελέγξουν τις επιφάνειες αυτού ορισμένες φορές το δευτερόλεπτο, αυτό θα πέσει, ενώ ένα κινητό τηλέφωνο από την στιγμή της εκκίνησής του μπορεί να χρειαστεί μερικά δευτερόλεπτα μέχρις ότου αρχικοποιηθεί, βρει το δίκτυο και είναι έτοιμο να δεχθεί

και να στείλει τηλεφωνήματα, ή μία ψηφιακή φωτογραφική μηχανή μπορεί να χρειαστεί αρκετά δευτερόλεπτα για να αποθηκεύσει μία φωτογραφία.

2.1.4 Επεξεργαστές στα ενσωματωμένα συστήματα

Στα ενσωματωμένα συστήματα απαιτείται αποτελεσματικότητα όσον αφορά την κατανάλωση ενέργειας, την απόδοση και το μέγεθος του κώδικα. Γενικά ο επεξεργαστής ενός ενσωματωμένου συστήματος εκτελεί ένα σύνολο αλγορίθμων και περιλαμβάνει μονάδες επεξεργασίας δεδομένων και ελέγχου.

2.1.4.1 Είδη επεξεργαστών :

- Επεξεργαστές γενικού σκοπού
- Application-specific processors(ASIPs)
- Application-specific circuits(ASICs)

2.1.4.2 Επεξεργαστές γενικού σκοπού

Προγραμματιζόμενα στοιχεία υλικού που έχουν τη δυνατότητα εκτέλεσης μεγάλης κλίμακας εφαρμογών. Περιλαμβάνουν μνήμες προγράμματος και δεδομένων, μονάδα επεξεργασίας δεδομένων με ΑΛΜ γενικού σκοπού και επαρκή αριθμό επεξεργαστών και μονάδα ελέγχου. Οι επεξεργαστές γενικού σκοπού γενικά προσφέρει χαμηλό κόστος, υψηλή ευελιξία, χαμηλή απόδοση και κατανάλωση ενέργειας.

2.1.4.3 Επεξεργαστές ειδικού σκοπού

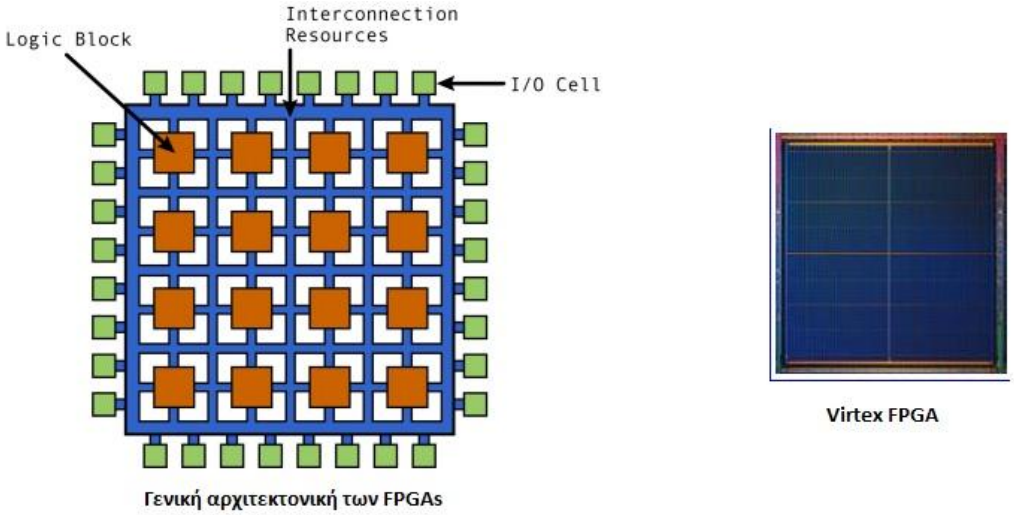
Προγραμματιζόμενα στοιχεία για εκτέλεση συγκεκριμένης εφαρμογής ή οικογένειας εφαρμογών με κοινά χαρακτηριστικά. Περιλαμβάνει μνήμες προγράμματος και δεδομένων με λειτουργικές μονάδες ειδικού σκοπού. Προσφέρουν υψηλή απόδοση, μικρό μέγεθος, χαμηλή κατανάλωση ενέργειας, μικρή ευελιξία

2.1.4.4 Κυκλώματα ειδικού σκοπού(ASIC)

Ψηφιακά κυκλώματα για την εκτέλεση ενός και μόνο αλγορίθμου(εφαρμογή ή μέρος εφαρμογής). Τα κυκλώματα αυτά είναι απαραίτητα στις περιπτώσεις όπου υπάρχει απόλυτος περιορισμός ταχύτητας ή κατανάλωσης ενέργειας. Περιλαμβάνουν μόνο τα κυκλώματα που απαιτούνται για την εκτέλεση του αλγορίθμου για τον οποίο έχουν σχεδιαστεί. Προσφέρουν υψηλή απόδοση, χαμηλή κατανάλωση ενέργειας, μικρό μέγεθος, υψηλό κόστος.

2.1.5 Προγραμματιζόμενα στοιχεία υλικού

Προκατασκευασμένα ψηφιακά κυκλώματα που μπορούν να προγραμματιστούν από το σχεδιαστή/χρήστη. Είναι σε διατάξεις πίνακα προγραμματιζόμενης λογικής που προγραμματίζονται από το χρήστη και όχι από τον κατασκευαστή. Αποτελούν εναλλακτική λύση των ASIC με μικρότερο κόστος ανάπτυξης και πιο γρήγορη διαθεσιμότητα. Επίσης οδηγούν σε υλοποιήσεις μεγαλύτερης επιφάνειας, χαμηλότερης απόδοσης και υψηλότερης κατανάλωσης. Τέτοια παραδείγματα είναι τα PLD και FPGA. Η δική μας εργασία αναπτύσσεται πάνω σε FPGA, και συγκεκριμένα στο Virtex 4 της Xilinx.



Σχήμα 2.1: Γενική αρχιτεκτονική των FPGAs και Virtex FPGA

2.2 Αξιοπιστία στα Ενσωματωμένα Συστήματα – Reliability

Τα ενσωματωμένα συστήματα έχουν διεισδύσει στην καθημερινή μας ζωή. Από αυτοκίνητα μέχρι ανελκυστήρες, συσκευές, κινητά τηλέφωνα και πολλά ακόμη. Όλο και περισσότερο ενσωματωμένες εφαρμογές γίνονται κρίσιμης σημασίας για την καθημερινότητα, όπως π.χ. στην αεροπορία και σε οπλικά συστήματα. Αυτό σημαίνει πως από τη στιγμή που αυξάνετε η χρησιμότητα τέτοιων εφαρμογών για σημαντικές εξυπηρετήσεις της καθημερινότητας μας, είναι απαραίτητο να βελτιωθεί και η αξιοπιστία, η ασφάλεια τέτοιων εφαρμογών.

Κάποια ζητήματα που επηρεάζουν την αξιοπιστία των ενσωματωμένων συστημάτων είναι η ακεραιότητα του σήματος, ζητήματα που προκύπτουν από εσωτερικές και εξωτερικές πηγές θορύβου καθώς και η επιτάχυνση της γήρανσης των συσκευών. Στα ενσωματωμένα συστήματα δεν μπορούν να χρησιμοποιηθούν οι τεχνικές βελτίωσης της αξιοπιστίας τους, όπως για τους ηλεκτρονικούς υπολογιστές. Αυτό διότι αυτά τα συστήματα είναι σε λειτουργία πάρα πολύ συχνά, αλλά με περιορισμένου πόρους, όπως το μικρότερο μέγεθος μνήμης ή χωρίς σκληρό δίσκο κ.α.. Ακόμη στην προσπάθεια να παρέχεις αξιόπιστη λειτουργία στα ενσωματωμένα συστήματα, είναι άκρως απαραίτητο να ικανοποιήσεις ταυτόχρονα αυστηρούς περιορισμούς, όπως η κατανάλωση ενέργειας και την σε πραγματικό χρόνο απόδοση. Αυτό θα έχει ως συνέπεια, αξιοπιστία βασισμένη επιθετικό πλεονασμό(redundancy) όπως η Triple-Module Redundancy(TMR), να μην είναι βιώσιμη. Όμοιες προσεγγίσεις, όπως οι τεχνικές checkpoint και rollback που χρησιμοποιούνται πολύ συχνά ενδέχεται να μην έχουν αποτέλεσμα. Επίσης τα ενσωματωμένα συστήματα συχνά έχουν μειωμένο θόρυβο χάρη στις βελτιστοποιήσεις στην ενέργεια ή την εγκατάσταση τους σε αντίξοα περιβάλλοντα. Εκτός από τα προβλήματα αξιοπιστίας του υλικού, η αυξανόμενη ποσότητα του περιεχόμενου του λογισμικού σε ενσωματωμένα συστήματα αποτελεί επίσης μια σημαντική πρόκληση. Πολλές περιπτώσεις ενσωματωμένων εφαρμογών, έχει αποδειχτεί, πως οφείλουν την αναξιοπιστία τους σε δυσλειτουργία του λογισμικού τους. Όπως πρόσφατα, το περιστατικό με το υβριδικό αυτοκίνητο όπου ακινητοποιήθηκε ξαφνικά όταν έπρεπε να αναπτύξει ταχύτητες αυτοκινητοδρόμου.

Για την αξιοπιστία λοιπόν των ενσωματωμένων συστημάτων υπάρχουν κάποιοι παράγοντες που πρέπει να λάβουμε υπόψη. Ένας από αυτούς είναι η μεταβλητότητα που

παρατηρείτε στη διαδικασία. Οι προκλήσεις για όσο γίνεται μικρότερο μέγεθος τρανζίστορ έχουν οδηγήσει σε σημαντική μεταβολή στις παραμέτρους του τρανζίστορ, όπως το μήκος του καναλιού επί παραδείγματι. Οι μεταβολές διάφορων παραμέτρων, προέρχονται από αρκετά φαινόμενα, όπως η μετατόπιση πλακιδίων(wafer misalignment), οι τυχαίες διακυμάνσεις προσμίξεων(random dopant fluctuations) κ.α.. Η απόκλιση μεταξύ σχεδιασμένων και κατασκευασμένων παραμέτρων των τρανζίστορ δημιουργεί σημαντικά λειτουργικά προβλήματα, όπως προβλήματα σταθερότητας σε κελιά(cells) μνήμης, τη δημιουργία νέων διαδρομών στον σχεδιασμό και αυξημένη διαρροή ρεύματος.

Ακόμη ένας παράγοντας που επηρεάζει την αξιοπιστία των ενσωματωμένων συστημάτων είναι τα περιοδικά σφάλματα(transient faults). Πρόκειται για σφάλματα που προκαλούνται από προσωρινές συνθήκες για το chip(όπως όταν η ισχύς του θορύβου υπερβαίνει ένα ορισμένο όριο) ή από εξωτερικούς θορύβους(όπως είναι τα soft-errors). Το κύκλωμα παραμένει αναλλοίωτο χωρίς να έχει υποστεί ζημιές, ακόμη και αν εισάγονται υπολογιστικά λάθη. Όπως οι τάσεις τροφοδοσίας μειώνονται καθώς και τα μεγέθη γίνονται όλο και μικρότερα στις τεχνολογίες του μέλλοντος, έτσι και η ανοχή σε soft-errors(soft-error tolerance) θα προαχθεί σε σημαντική πρόκληση για τον σχεδιασμό των μελλοντικών ενσωματωμένων συστημάτων.

Ένα ακόμη ζητούμενο για την αξιοπιστία των σύγχρονων ενσωματωμένων συστημάτων είναι το φαινόμενο της επίδρασης γήρανσης(aging effect). Στα σύγχρονα ενσωματωμένα συστήματα, τα επίπεδα ανώτατης θερμοκρασίας έχουν αυξηθεί. Αυτός είναι ο λόγος που διάφοροι μηχανισμοί αποτυχίας έχουν επιταχυνθεί, όπως το electro-migration, το thermal-cycling, με αποτέλεσμα την ολοκληρωτική μείωση της αξιοπιστίας. Επομένως δημιουργείται σοβαρότατος κίνδυνος η επιταχυνόμενη γήρανση να προκαλέσει προβλήματα στην συσκευή μέσα στο διάστημα εγγύησης της. Εδώ λοιπόν έχουμε ακόμη μια πρόκληση, για το πώς ένα ενσωματωμένο σύστημα θα ανταπεξέλθει στις σύγχρονες απαιτήσεις με αξιοπιστία και σταθερότητα.

Τέλος την αξιοπιστία επηρεάζει και το λογισμικό που καλείται να υποστηρίξει το ενσωματωμένο σύστημα. Πιθανά λάθη στο ενσωματωμένο σύστημα που προέρχονται από μέτριο compilation ή από ατυχείς ρυθμίσεις του source κώδικα, έχουν ως συνέπεια καίρια πλήγματα στην αξιοπιστία των συστημάτων. Ειδικότερα σήμερα που οι απαιτήσεις σε μεγάλη ποσότητα και ταυτόχρονα καλής ποιότητας κώδικα είναι κατά πολύ αυξημένες. Για την ανοχή τέτοιων σφαλμάτων αναπτύσσονται το τελευταίο διάστημα αρκετοί εναλλακτικοί μηχανισμοί.

Η δική μας δουλεία δεν επικεντρώνεται τόσο στον με ποιόν τρόπο παράγεται ένα fault στο σύστημα μας, δηλαδή δεν εξετάζουμε ποιος από τους παραπάνω παράγοντες δημιουργεί το πρόβλημα, αλλά πως αυτό θα αντιμετωπιστεί. Μας απασχόλησε λοιπόν με ποια διαδικασία θα επιτύχουμε fault detection, και γενικότερα η δυνατότητα ισχυρού fault tolerance μηχανισμού, ανεξάρτητα από την πηγή παραγωγής σφαλμάτων.

2.3 Fault Tolerance ιδιότητα

Fault-Tolerance είναι η ιδιότητα που επιτρέπει ένα σύστημα (συχνά υπολογιστικό) να συνεχίσει σωστά την λειτουργία του σε περίπτωση αποτυχίας (ή περισσότερων αποτυχιών παράλληλα) σε μερικές από τις λειτουργίες του. Εάν η λειτουργική ποιότητα του συστήματος μειώνεται, η μείωση είναι ανάλογη με την σοβαρότητα της αποτυχίας, όπως σε ένα αφελώς-σχεδιασμένο σύστημα στο οποίο ακόμη και μια μικρή αποτυχία μπορεί να προκαλέσει τη συνολική διακοπή. Η αποκατάσταση λαθών σε τέτοιου είδους συστήματα (fault tolerant systems) μπορεί να χαρακτηριστεί είτε ως **κίνηση-προς τα εμπρός (roll-forward)** είτε ως **κίνηση-προς τα πίσω (roll-back).**

Όταν το σύστημα ανιχνεύσει ότι έχει γίνει ένα λάθος, η αποκατάσταση λαθών ‘κίνηση-προς τα μπροστά’ αναλαμβάνει την κατάσταση του συστήματος εκείνο το χρονικό διάστημα

και το διορθώνει, ώστε να είναι σε θέση να συνεχίσει. Η αποκατάσταση λαθών 'κίνηση-προς τα πίσω' επαναφέρει την κατάσταση του συστήματος σε κάποια προηγούμενη-σωστή κατάσταση, χρησιμοποιώντας σημεία ελέγχου (checkpoints), και συνεχίζει την λειτουργία από εκείνο το σημείο. Η αποκατάσταση λαθών 'κίνηση-προς τα πίσω' απαιτεί οι διαδικασίες μεταξύ του σημείου ελέγχου και της λανθάνουσας κατάστασης να είναι μοναδικές. Μερικά συστήματα χρησιμοποιούν και τους δύο τρόπους αποκατάστασης λαθών για τα διαφορετικά λάθη ή για διαφορετικά μέρη ενός λάθους. Στο πλαίσιο ενός ιδιαίτερου συστήματος, το ελάττωμα-ανοχή μπορεί να επιτευχθεί με την πρόγνωση των εξαιρετικών καταστάσεων και την δημιουργία του συστήματος με την ικανότητα να τις αντιμετωπίζει, γενικά, να στοχεύει να είναι από μόνο του σταθεροποιήσιμο (self-stabilization) έτσι ώστε το σύστημα λειτουργεί χωρίς λάθη. Ωστόσο αν οι συνέπειες μιας διακοπής του συστήματος είναι καταστροφικές, ή το κόστος αξιοπιστίας του συστήματος είναι πολύ υψηλό, μια καλύτερη λύση μπορεί να είναι η χρήση κάποιας μορφής διπλασιασμού. Εν πάση περιπτώσει, αν η συνέπεια μιας διακοπής του συστήματος είναι καταστροφική, το σύστημα πρέπει να είναι σε θέση να επαναφέρεται σε κάποια ασφαλή κατάσταση. Αυτό είναι παρόμοιο με την αποκατάσταση λαθών 'κίνηση-προς τα πίσω' αλλά μπορεί να επιτευχθεί και από άνθρωπο αν είναι παρόν.

Στη δική μας περίπτωση το fault tolerance επιτυγχάνεται μέσα από τη διαδικασία του real time monitoring, η οποία πραγματοποιείται με βάση τους χρόνους εκτέλεσης που έχουμε υπολογίσει, για τον αλγόριθμο που χρησιμοποιούμε πειραματικά.

2.3.1 Monitoring

Οι ερευνητικές προσπάθειες για real-time monitoring έχει αφιερωθεί σε μεγάλο βαθμό στην αντιμετώπιση των επιπτώσεων πολλαπλών επιδράσεων(effects) και τις χρονικές παρεμβάσεις σε πλήθος εφαρμογών παρακολούθησης(monitoring). Αυτό έχει ως αποτέλεσμα να υπάρχουν δεκάδες τεχνικές που είναι καθαρά, τεχνικές λογισμικού για την υποστήριξη ειδικού υλικού. Συστήματα software monitoring προσφέρουν τη φθηνότερη και πιο ευέλικτη λύση όπου η πλέον διαδεδομένη τεχνική είναι να εισάγεται κώδικας - ως όργανο μέτρησης - σε καίρια σημεία του κώδικα που θέλουμε να παρακολουθήσουμε. Όταν λοιπόν το 'όργανο μέτρησης' ενεργοποιείται ξεκινά η διαδικασία του monitoring και η πληροφορία που μας ενδιαφέρει αποθηκεύεται στην μνήμη του κώδικα που παρακολουθούμε. Τα μειονεκτήματα της συγκεκριμένης τεχνικής είναι η αξιοποίηση των πόρων του συστήματος που εκτελείται η παρακολουθούμενη εφαρμογή, όπως η χώρος μνήμης και ο χρόνος εκτέλεσης του επεξεργαστή. Επιπλέον, για την αποφυγή probes effects ο κώδικας μέτρησης πρέπει να φυλάσσεται στο λογισμικό που αναπτύσσεται ή να αντισταθμίζεται από την σε πραγματικό χρόνο schedulability ανάλυση. Πάντως και με τις δύο εναλλακτικές το αποτέλεσμα θα είναι να υπάρχουν κυρώσεις στην απόδοση του συστήματος.

Από την άλλη πλευρά τα συστήματα παρακολούθησης υλικού(hardware monitoring), χρησιμοποιούν ειδικό υλικό ώστε να εξετάσουν παθητικά τα φυσικά λεωφορεία του παρακολουθούντος συστήματος, όπως ο επεξεργαστής τα busses του συστήματος, έτσι συλλέγουν πληροφορίες που αφορούν το monitoring χωρίς όμως να γίνεται παρέμβαση στην εκτέλεση της εφαρμογής. Με αυτήν την τακτική έχουμε ένα μεγάλο πλεονέκτημα, καθώς μπορεί να αποφευχθούν ολοκληρωτικά τα probe effects. Μειονεκτήματα είναι η εξάρτηση(dependency) από την αρχιτεκτονική και το σχετιζόμενο κόστος. Η υβριδική παρακολούθηση(hybrid monitoring) χρησιμοποιείται συνδυάζοντας τις δύο παραπάνω μεθόδους, μειώνοντας έτσι τις επιπτώσεις του software που χρησιμοποιείται ως όργανο μέτρησης.

2.3.1.1 System-On-Chip Monitoring

Με τη σημερινή, σε πολύ μεγάλο βαθμό ενσωμάτωση του υλικού, βάζοντας ολόκληρα συστήματα σε ένα chip (SoC), οι παραδοσιακοί hardware monitors, αντιμετωπίζουν σοβαρές δυσκολίες. Πολλές φορές εκτός από τον επεξεργαστή, τα I/O, τις μνήμες, ακόμη και η

τοπική-σταθερή μνήμη εντάσσονται στο ίδιο chip. Δεδομένου επίσης ότι τα chip packages μπορεί να είναι αποφρακτικά έχοντας περιορισμένες καρφίτσες(pins), έχει καταστεί σχεδόν αδύνατο για το εξωτερικό hardware να εξετάσει εσωτερικά σήματα. Για συστήματα πραγματικού χρόνου που στηρίζονται σε αυτές τις αρχιτεκτονικές εγκαταστάσεις είναι ανάγκη να υπάρχει η δυνατότητα πρόσβασης σε πληροφορίες εκτέλεσης που διαμένουν on-chip, καθώς και για την αποφυγή παρεμβολών στην εκτέλεση του συστήματος.

Μέχρι σήμερα οι ομάδες σχεδιασμού, σχεδιάζουν μεγάλης κλίμακας ολοκληρωμένα κυκλώματα(ICs), τα οποία στηρίζονται κυρίως στις υπάρχουσες τεχνολογίες επαναχρησιμοποίησης[1], τα λεγόμενα IPs(intellectual property). Τα υπάρχοντα σχέδια SoC ενσωματώνουν όλο και πιο ολοκληρωμένα και με αυξανομένη λειτουργικότητα κυρίως για πολυπύρηνα συστήματα, Σαν SoCs συνήθως απευθύνονται σε ένα πεδίο συγγραφής, που συνήθως περιλαμβάνει εξειδικευμένες διευθύνσεις IP με τη μορφή συν-επεξεργαστών(co-processors), η οποία μπορεί να λύσει επαρκώς να επιλύσει tasks μέσα στο συγκεκριμένο πεδίο εφαρμογής, καθώς και προγραμματιζόμενοι πυρήνες κάνοντας τα πολυπύρηνα συστήματα ετερογενή. Με την προσθήκη προγραμματιζόμενων πυρήνων(ARM, MIPS, Trimedia), τα πολυπύρηνα SoC συστήματα γίνονται όλο και πιο επαναπρογραμματιζόμενα.

Σύγχρονες εφαρμογές πολυμέσων έχουν εμφανιστεί. Οι εφαρμογές αυτές γίνονται πολύπλοκες όσο αφορά το μέγεθος με αποτέλεσμα το πεδία τέτοιων εφαρμογών να έχουν αυξηθεί ραγδαία. Επίσης ακόμη ένα παράγοντας για την αύξηση του μεγέθους είναι η όλο και αυξανόμενη πολυπλοκότητα από τα συνεχώς προστιθέμενα χαρακτηριστικά που υποστηρίζονται, όπως για παράδειγμα τα πολύ-παράθυρα(multi-window) στην τηλεόραση. Το πεδίο τέτοιων εφαρμογών είναι αρκετά διευρυμένο σε σχέση με το γνήσιες εφαρμογές audio-video για απλές τηλεοράσεις. Ακόμη οι χρήστες έχουν υψηλές προσδοκίες από την ποιότητα εικόνας και ήχου από τέτοιες προχωρημένες εφαρμογές.

Η ανάγκη λοιπόν που δημιουργείται για επέκταση των εφαρμογών, οδηγείται από τον δυναμισμό και τη σύγκλιση σε τέτοιες εφαρμογές. Οι αιτήσεις(apps) πρέπει να είναι να είναι σε θέση να αντιμετωπίζουν ξαφνικές αλλαγές στις απαιτήσεις της πηγή και στη διαθεσιμότητα των πόρων. Οι επεκτατικές εφαρμογές, που περιγράψαμε, μπορούν να χειριστούν τις αλλαγές όσο αφορά τη διαθεσιμότητα των πόρων κατά το χρόνο εκτέλεσης, επιτρέποντας έτσι, για παράδειγμα, κομψές υποβαθμίσεις ή αναβαθμίσεις στην ποιότητα που γίνεται αντιληπτή από τον χρήστη.

Αντιλαμβανόμαστε λοιπόν, πως μελλοντικές αρχιτεκτονικές SoC γίνονται τόσο περίπλοκες που θα πρέπει οι επιδόσεις τους να παρακολουθούνται κατά το χρόνο εκτέλεσης τους.

On-chip monitors επίδοσης είναι η μέθοδος RST(Reservoir Saturation Tool), που διερευνάτε η εφαρμογή της. Ακόμη έχει προταθεί από το [2] η χρήση μιας μονάδας παρακολούθησης των επιδόσεων(PMU) για System-On-Chip(SoC) πλατφόρμα για το AMBA AXI bus, ώστε να δοθεί μια εικόνα για τους σχεδιαστές του συστήματος με σκοπό την ανάλυση των προβλημάτων απόδοσης του συστήματος. Ακόμη μια σημαντική εφαρμογή παρακολούθησης είναι μια από τις πιο μεγάλες σχεδιαστικές επιτυχίες της intel, ο επεξεργαστής Pentium 4, ο οποίος διαθέτει επίσης ενσωματωμένους monitors για την επίβλεψη της απόδοσης του συστήματος[3]. Υποστηρίζει 48 ανιχνευτές γεγονότος(event detector) και 18 μετρητές γεγονότος(counter event) με δυνατότητα μέτρησης για 18 events ταυτόχρονα. Επειδή ο επεξεργαστής υποστηρίζει multi-threaded εκτελέσεις περιλαμβάνει qualication από την ανίχνευση γεγονότων σύμφωνα με το thread id και qualication από το μέτρημα γεγονότων από το thread mode. Οι monitors αυτοί είναι συνήθως ενσωματωμένοι στον ίδιο τον επεξεργαστή και χρησιμοποιούνται κυρίως για τη δυναμική ρύθμιση των εφαρμογών.

Άλλη μία μεθοδολογία στους on-chip monitors είναι οι θερμικοί(thermal) monitors. Οι θερμικοί υποστηρίζουν την σε πραγματικό χρόνο, θερμική διαχείριση με τη μορφή της θερμικής-επίγνωσης με λειτουργικό σύστημα, το οποίο χρησιμοποιεί την μετανάστευση εργασίας(task migration) ως τον τρόπο για τον έλεγχο και την μείωση των θερμών σημείων.

Ακόμη οι λεγόμενοι ελεγκτές ισχυρισμού(assertion-checkers) έχουν χρησιμοποιηθεί επιτυχώς για την επαλήθευση μεγάλων ICs, όπως στην [4]. Οι συγγραφείς πιστεύουν ότι οι σχεδιαστές του υλικού θα πρέπει να συμφιλιωθούν με το γεγονός ότι το υλικό, όπως και το λογισμικό, δεν θα μεταφέρεται χωρίς να έχει σφάλματα. Ωστόσο θεωρούν πως η κατάσταση αυτή μπορεί να γίνει αποδεκτή έχοντας τους κατάλληλους μηχανισμούς για την επικύρωση σε πραγματικό χρόνο, όπου θα ανιχνεύει τα σφάλματα και θα τα διορθώσει εφόσον κριθεί απαραίτητο. Προτείνεται μηχανισμός on-chip με ελεγκτές ισχυρισμού(assertion-checkers) πραγματικού χρόνου, που θα παρακολουθεί το σύστημα συνεχώς κατά το χρόνο εκτέλεσης με σκοπό τον εντοπισμό σφαλμάτων. Ένας ισχυρισμός είναι μια δήλωση που αποτελεί μέρος των προδιαγραφών σχεδιασμού στο σύστημα.

Πυρήνες(IP) - monitors έχουν επίσης χρησιμοποιηθεί, ως μέθοδος, σε σύνθετα SoCs για εντοπισμό σφαλμάτων. Κάθε υπολογιστικός πυρήνας του συστήματος είναι εξοπλισμένος με έναν monitor, η οποία τοποθετείται μεταξύ πυρήνα και των αντίστοιχων διασυνδέσεων του δικτύου τους. Κάθε πυρήνας μπορεί να επιδιορθωθεί μεμονωμένα. Είναι μια λύση με κόστος αλλά αρκετά βήματα μπροστά από τις συνηθισμένες πρακτικές, καθώς με την εισαγωγή πολλαπλών monitors επιτυγχάνετε σοβαρή πρόοδος στην εξέλιξη της εργασίας που παρακολουθείται.

3. Συναφείς Εργασίες

Η έννοια της αξιοπιστίας είναι συνεχώς ένα θέμα έρευνας στον τομέα της πληροφορικής και ειδικότερα στα σε πραγματικό χρόνο ενσωματωμένα συστήματα(real-time embedded systems). Αρχικά έχουμε κατηγοριοποιήσει τις υπάρχουσες τεχνικές, σε τεχνικές υλικού και λογισμικού για την αύξηση της αξιοπιστίας ενός συστήματος ενσωματωμένων υπολογιστών. Σχολιάζουμε επίσης εργαλεία που προκαλούν σφάλματα(injection tools) και τεχνικές διαχείρισης ενέργειας.

3.1 Τεχνικές Υλικού

Στην [7] οι συγγραφείς εισάγουν την αρ'όρου ζωής αξιοπιστία και σχεδιάζουν ένα σύστημα το οποίο υποστηρίζει την αυτόματη ανοχή σε βλάβες υλικού. Η προτεινόμενη αρχιτεκτονική υποστηρίζει online εφαρμογή, με τον κατάλληλο online αλγόριθμο. Αυτός ο αλγόριθμος βασίζεται σε πληροφορίες από τον παρατηρητή(observer) και μπορεί να ανιχνεύσει ελαττώματα απ'όλους τους πόρους του συτήματος.

Μια τεχνική ανάλυσης αξιοπιστίας προτείνεται στο [8], η οποία βασίζεται στο λεγόμενο Δυαδικό Διάγραμμα Απόφασης(Binary Decision Diagram). Οι συγγραφείς υποστηρίζουν ότι η τεχνική τους αποφεύγει δαπανηρές επαναλήψεις των πόρων και προσφέρει λύσεις υψηλής ποιότητας με επαυξημένη την αξιοπιστία του συστήματος. Στην ίδια λογική και οι συγγραφείς στην [9] παρουσιάζουν μια γενική εικόνα σε σχεδιαστικές προσεγγίσεις αυτοελέγχου(self-checking), που όλες βασίζονται σε κάποιο είδος πλεονασμού. Χρησιμοποιούν κυρίως τον χρονικό πλεονασμό(time redundancy). Στην δική μας εργασία ακολουθούμε την Τρίτη περίπτωση πλεονασμού. Γενικά η αυτό-διάγνωση(self-detection) είναι υψίστης σημασίας για ένα ενσωματωμένο σύστημα και ένας από τους κύριους στόχους της εργασία μας.

Μία ακόμη σημαντική προσπάθεια παρουσιάζεται στο [10] όπου οι συγγραφείς αντιμετωπίζουν τις βλάβες σε συσκευές FPGAs και ASICs, τα λεγόμενα Single Event Uset(SEU) ή Single Event Transient(SET). Συνδυάζουν τις τεχνικές Triple Modular Redundancy(TMR) με Partial Dynamic Reconfigurations (Μερική Δυναμική Επανεκτέλεσης), ώστε να αποφευχθούν οι αρνητικές πλευρές της λύσης του TMR. Τα αποτελέσματα της μελέτης περιπτώσεων δείχνουν μεγάλες δυνατότητες για την αντιμετώπιση soft-errors. Η επόμενη δουλειά σχετίζεται με ανίχνευση σφαλμάτων και τεχνικές ανοχής σφαλμάτων, υλικού[15]. Παρουσιάζουν ένα εξειδικευμένο σύστημα υλικού αυτοελέγχου, που χρησιμοποιείται ως κεντρικό θέμα στην στρατηγική σχεδίασης για τη δημιουργία επεξεργαστή που θα ανέχεται σφάλματα(fault-tolerant processor). Η αποτελεσματική ανάκτηση με επαναφορά(rollback recovery) είναι δυνατή χάρη στην συγκεκριμένη αρχιτεκτονική η οποία δεν στηρίζεται στις σύνηθες αρχιτεκτονικές RISC ή CISC, αλλα σε μια μορφή αρχιτεκτονικής στοίβας. Ο κύριος στόχος της δουλειάς αυτής και του αυτοελέγχου είναι, τα δεδομένα που δεν έχουν ακόμη επικυρωθεί να αποστέλλονται στην κύρια μνήμη ώστε να επιτραπεί μια γρήγορη εκτέλεση επαναφοράς στις καταστάσεις που είναι εσφαλμένες. Χρησιμοποιούνται επίσης τεχνικές κωδικοποίησης σφάλματος, στον πυρήνα του επεξεργαστή για τον εντοπισμό λαθών και στο HW που έχουν σχεδιάσει για την προστασία προσωρινά αποθηκευμένων δεδομένων, από τις πιθανές αλλαγές που μπορεί να προκαλέσουν παροδικά σφάλματα.

3.2 Τεχνικές λογισμικού

Όσον αφορά τις τεχνικές λογισμικού στο [12] οι συγγραφείς δημιουργουν ένα πλαίσιο ανοχής σφαλμάτων που βασίζεται στο μοντέλο ανοχής σφαλμάτων υποστηριζόμενο από threads. Αυτό το μοντέλο αξιολογεί την ανάπτυξη ενός συστήματος φιλτραρίσματος ενός ραντάρ με αποτέλεσμα να έχουμε πλεονεκτήματα όπως την ευκολία διαμόρφωσης και υψηλής ευελιξίας. Στην ίδια λογική και το [13], παρουσιάζεται η σύνδεση αποτυχημένου λογισμικού με τον πραγματικό κόσμο. Πρακτικά περιγράφουν γεγονότα που βασίζονται στην ταυτοποίηση σφαλμάτων λογισμικού που προκαλούνται κυρίως από εφαρμογές που δυσλειτουργούν σε πραγματικές καταστάσεις. Η συγκεκριμένη ανάλυση παρέχει καλύτερη κατανόηση το πώς αποτυχίες στο λογισμικό επηρεάζουν τη ζωή μας και πως μπορούμε να δημιουργήσουμε εργαλεία μοντελοποίησης για να αποφευχθούν τέτοιες αποτυχίες.

Σε τεχνικές λογισμικού για επαύξηση της αξιοπιστίας εστιάζει η [14]. Το έργο περιγράφει μεθόδους για την ανάληψη δράσης από τις αποτυχίες του λογισμικού και μελετά διάφορους τύπους από βλάβες που προκαλούνται από σφάλματα λογισμικού. Όπως τονίζουν, αναγνωρίζονται τρεις βασικές κατηγορίες τεχνικών που μπορούν να αναπτυχθούν για τη διαχείριση: α) αποφυγή βλάβης, β) την ανίχνευση σφαλμάτων, γ) ανοχή σε σφάλματα. Η εργασία μας επικεντρώνεται κυρίως στην τρίτη περίπτωση.

Στο [16] οι συγγραφείς προτείνουν μια προσέγγιση για την ανάλυση των αρχιτεκτονικών λογισμικού όσον αφορά την αξιοπιστία. Τα σενάρια σφαλμάτων στηρίζονται σε καθιερωμένα μοντέλα λαθών και μεθόδους ανάλυσης επιδράσεων. Παρουσιάζουν την τεχνική SARA, μία αρχιτεκτονική λογισμικού, μεθόδου ανάλυσης της αξιοπιστίας για ενσωματωμένα συστήματα. Έχουν εφαρμόσει μια μέθοδο για την ανάλυση της αρχιτεκτονικής λογισμικού της ψηφιακής τηλεόρασης, όπου ο ορισμός του μοντέλου σφαλμάτων, τα σενάρια αποτυχίας και οι υπολογισμοί έχουν σοβαρή επιρροή από τον τομέας της μηχανικής αξιοπιστίας[17].

3.3 Εργαλεία εισφοράς σφαλμάτων(injection tools)

Στην κατεύθυνση για μετριασμό των SEUs με τον πλέον βέλτιστο τρόπο ερευνητές στην [11] δημιούργησαν ένα σύστημα για ένεση σφαλμάτων(fault injection)- δηλαδή εισαγωγή σφαλμάτων στο σύστημα- πάνω στην Virtex FPGA πλατφόρμα, χρησιμοποιώντας εργαλείο υλικού από VHDL. Το συγκεκριμένο εργαλείο έχει τη δυνατότητα να εισφέρει σφάλματα στον μηχανισμό διαμόρφωσης της συσκευής. Η δημιουργία του εργαλείου δεν επηρεάζεται από τον συνολικό σχεδιασμό του συστήματος κι έτσι είναι ικανό για ταχεία παροχή σφάλματος. Γενικότερα η τεχνική του injection είναι ο πιο συνηθισμένος τρόπος για την επαλήθευση τεχνικών fault tolerant.

3.4 Θέματα διαχείρισης ενέργειας

Η διαχείριση και κατανάλωση ενέργειας έχουν αναδειχθεί ως τα κορυφαία θέματα στα σύγχρονα υπολογιστικά συστήματα. Στο [18] οι συγγραφείς ασχολούνται με ιδιότητες fault tolerant σε σύστημα διαχείρισης ενέργειας. Η μείωση της τάσης τροφοδοσίας για την εξοικονόμηση ενέργειας αυξάνει τα ποσοστά εμφάνισης των soft-errors που προκαλούνται από SEU για δύο λόγους: i) η μείωση της τάσης κάνει τα ψηφιακά κυκλώματα πιο επιρρεπή σε λάθη ii) ο περιορισμός της τάσης τροφοδοσίας αυξάνει τη διάρκεια της διαδικασίας η οποία αυξάνει τις πιθανότητες εμφάνισης SEU. Ως εκ τούτου είναι πολύ σημαντικό να δημιουργηθεί μια βέλτιστη μεθοδολογία που θα εξετάζει τον δυναμικό Προγραμματισμό της Τάσης(Dynamic Voltage Scheduling). Για την ανάπτυξη αυτής της μεθοδολογίας οι συγγραφείς μελετούν μοντέλα για ενεργειακή καθυστέρηση και soft errors που προέρχονται από SEU, όπως το μοντέλο α) Ισχύς και Ενέργειας, β) Μοντέλο καθυστέρησης γ) Μοντέλο εργασίας και τέλος το δ) μοντέλο διάρκειας ζωής και MTTF.

4. Γενική Περιγραφή Μεθοδολογίας

Σε αυτήν την ενότητα θα δώσουμε μια γενική περιγραφή της μεθοδολογίας μας, σε σχέση με άλλες πιθανές μεθόδους που θα μπορούσαμε ενδεχομένως να χρησιμοποιήσουμε.

4.1 Μεθοδολογίες που θα μπορούσαν να χρησιμοποιηθούν

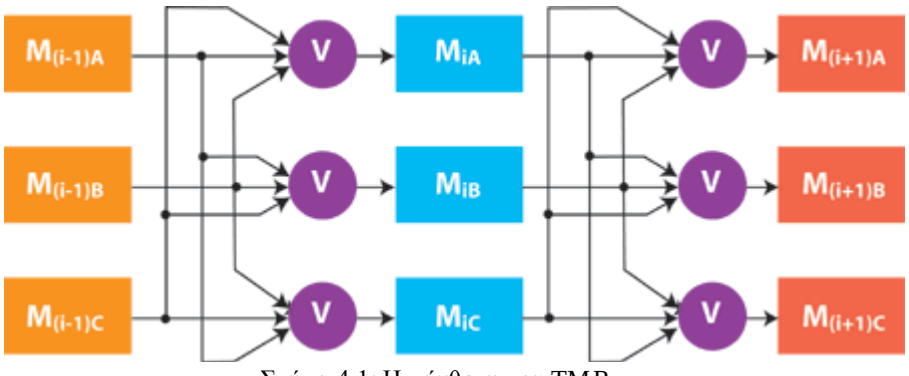
Πέρα από την μεθοδολογία που τελικώς χρησιμοποιήσαμε υπάρχουν δοκιμασμένες μεθοδολογίες που έχουν ήδη χρησιμοποιηθεί είτε ως επιλογές υλικού, είτε ως επιλογές λογισμικού.

4.1.1 TMR(Triple Modular Redundancy)

Μία από αυτές του υλικού είναι η TMR. Είναι η πλέον διαδεδομένη τακτική υλικού, παθητικού πλεονασμού που επιτυγχάνει την απόκρυψη βλάβης. Χρησιμοποιεί τρία αντίγραφα του συνόλου του συστήματος και προσθέτει τον εκλογέα(voter), ο οποίος αναλαμβάνει να αναγνωρίσει το σωστό αποτέλεσμα μεταξύ των τριών αντιγράφων με βάση τι υπερισχύει πλειοψηφικά**(Σχήμα).** Η τεχνική μπορεί να εφαρμοστεί σε διαφορετικά επίπεδα άντλησης, από το σύνολο του συστήματος μέχρι σε ένα στοιχείο του, σε αυτή τη δεύτερη περίπτωση έχουμε μειωμένη ανίχνευση latency μαζί με υψηλότερο κόστος σε σχέση με ένα RST. Επιπλέον λόγω του γεγονότος ότι ο voter μπορεί να επηρεαστεί από ένα fault, θα πρέπει να είναι συνολικά αυτοελεγχόμενος(Self-Checking), δηλαδή να είναι σε θέση να εντοπίσει τα λάθη του, αν και θα εξακολουθεί να είναι μοναδικό σημείο αποτυχίας.

Η TMR εφαρμόζεται για τον περιορισμό των soft-errors στα στοιχεία μνήμης που αποθηκεύονται προσωρινά δεδομένα που χρησιμοποιούνται κατά τη διάρκεια των υπολογισμών, καθώς και στην αποθήκευση των ρυθμίσεων για bitstream. Στην κατάσταση RST, η κατάσταση απόκρυψης του εκλογικού συστήματος παρέχει την δυνατότητα αυτόνομης ανάκαμψης από το λάθος, ενώ στη δεύτερη περίπτωση μια επαναρίθμηση της συσκευής είναι εφικτή.

Δεδομένου ότι η TMR μπορεί να εφαρμοστεί σε διαφορετικά διακριτά επίπεδα και σε διαφορετικές περιοχές γενικά, η πιθανή υποβάθμιση των επιδόσεων και οι φορές επαναρίθμησης χαρακτηρίζουν τις διάφορες λύσεις.



Σχήμα 4.1: Η σύνθεση του TMR

4.1.2 ODC(Orthogonal Defect Classification)

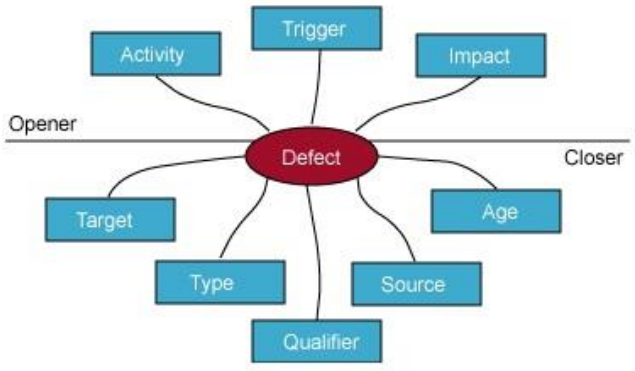
Μια ενδιαφέρουσα πρόταση για επαύξηση της αξιοπιστίας λογισμικού, είναι η τεχνική της IBM.

Το ODC είναι ένα σύστημα ταξινόμησης των faults και βοηθά στην καταγραφή των πληροφοριών για τα faults που εμφανίζονται. Επίσης είναι ένα σύστημα μέτρησης για τις διαδικασίες λογισμικού και βασίζετε στις σημασιολογικές πληροφορίες που περιέχονται στα εμφανιζόμενα faults. Ακόμη μπορεί να βοηθήσει στην αξιολόγηση της αποτελεσματικότητας και αποδοτικότητας των δοκιμών για εντοπισμό σφαλμάτων. Με απλά λόγια το ODC ορίζει ένα σύνολο χαρακτηριστικών για δοκιμαστές σφαλμάτων ώστε να καταγράψει τα πιθανά σφάλματα. Παρέχει επίσης μια σειρά πειραματικών κανόνων για την ανάλυση αυτών των σφαλμάτων. Έπειτα, με βάση αυτή την ανάλυση μπορούν να ληφθούν ακριβή μέτρα για τη βελτίωση της ποιότητας του κώδικα.

Κατά την εφαρμογή της τεχνικής, ο ελεγκτής(tester) έχει τον ρόλο του «αποστολέα», και ο προγραμματιστής τον ρόλο του «ανταποκριτή». Τα οκτώ βασικά χαρακτηριστικά **(Σχήμα 4.1)** της εφαρμογής εξηγούνται παρακάτω:

- **Activity(δραστηριότητα),** αναφέρεται στο πραγματικό στάδιο της διαδικασίας(έλεγχος κώδικα, δοκιμή λειτουργίας, κλπ) όταν διαπιστώνονται βλάβες.
- **Trigger(ενεργοποίηση),** περιγράφει το περιβάλλον ή την κατάσταση που θα πρέπει να υπάρχει για να εκτεθούν τα σφάλματα.
- **Impact(επίπτωση),** αναφέρεται είτε στη φαινομενική, είτε στην πραγματική επίπτωση προς τους χρήστες
- **Target(στόχος),** αποτελεί υψηλού επιπέδου ταυτότητα(τη σχεδίαση, τον κώδικα, κ.α.), της οντότητας που ορίστηκε.
- **Type(τύπος),** αντιπροσωπεύει τη φύση της πραγματικής διόρθωσης που έγινε
- **Qualifier(χαρακτηριστής),** διευκρινίζει αν η λύση που δόθηκε, ήταν λόγω έλλειψη, ή ανακριβή , ή εξωτερικού κώδικα ή πληροφορίας.
- **Source(πηγή),** δείχνει αν το fault που βρέθηκε στον κώδικα, επαναχρησιμοποιείται από μια βιβλιοθήκη, μεταφέρεται από τη μια πλατφόρμα στην άλλη, είτε ανατίθεται σε έναν άλλο προμηθευτή(vendor)
- **Age(ηλικία),** προσδιορίζει την ιστορία του στόχου(τη σχεδίαση, τον κώδικα, κλπ) που είχε το σφάλμα.

Σήμερα, το ODC υποστηρίζεται από την IBM – RCQ και την Jmystiq.



Σχήμα 4.2: Τα χαρακτηριστικά του ODC

4.2 Γενική περιγραφή της υλοποιούμενης μεθοδολογίας

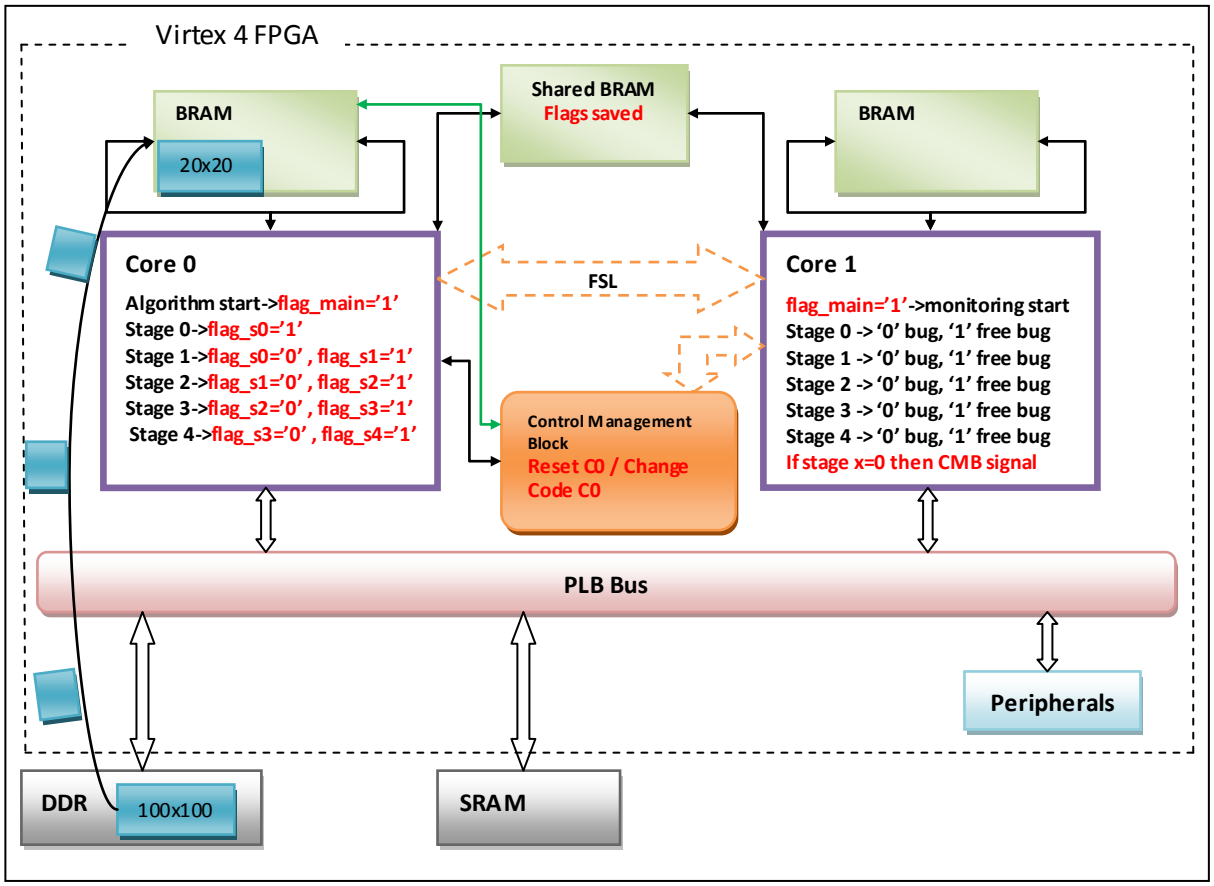
Επιλέξαμε μια συνδυαστική μέθοδος (Υλικού και Λογισμικού), που μας παρέχει λιγότερο κόστος και μεγαλύτερη ευελιξία. Η μεθοδολογία μας στηρίχτηκε τόσο στο σύστημα που μας παρέχει η Xilinx, καθώς αναπτύχτηκε στο Virtex 4 FPGA, όσο και στις βιβλιοθήκες της εταιρίας που είναι απαραίτητες για την επικοινωνία των επεξεργαστών με τα περιφερειακά και τις μνήμες καθότι δεν υπάρχει λειτουργικό σύστημα να υποστηρίξει αυτήν την σύνδεση, παρά μόνο οι controllers των περιφερειακών και των μνημών. Η γενική αρχιτεκτονική καθώς και οι βασικές λειτουργίες του λογισμικού περιγράφονται σε αυτήν την ενότητα**(Σχήμα 4.2.1).** Στην επόμενες δύο ενότητες ακολουθεί αναλυτική περιγραφή τόσο της αρχιτεκτονικής, όσο και του Λογισμικού που αναπτύξαμε.

Η μεθοδολογία μας έχει λοιπόν συνδυάσει τόσο την ανάπτυξη ειδικού λογισμικού που θα υποστηρίξει τη λειτουργία του monitoring – fault detection, όσο και τη δημιουργία εξειδικευμένου υλικού που αναλαμβάνει, με βάση τα αποτελέσματα του monitoring, της λειτουργίες επανεκκίνησης, διόρθωσης, αντιγραφής κώδικα.

Αρχικά για να δοκιμάσουμε την μεθοδολογίας μας, επιλέξαμε ο έλεγχος να διαμορφωθεί με βάση τον δυναμικό αλγόριθμο Canny, τον οποίο διαχωρίσαμε σε 5 Βασικά stages σε σχέση με την λειτουργία του. Ο canny τρέχει στον επεξεργαστή 1(core 1), την διαδικασία του fault detection θα υποστηρίξει ο επεξεργαστής 2(core 2). Ο κώδικας του monitoring γράφεται στον δεύτερο επεξεργαστή. Η ανάγκη άμεσης επικοινωνίας των δύο επεξεργαστών λύνετε με την σύνδεση τους μέσω FSL(Fast Symplex Link), ενώ για τον πλήρη έλεγχο των σταδίων του αλγορίθμου, δημιουργήσαμε κοινόχρηστες σημαίες(flags) ή κοινόχρηστα σημεία ελέγχου. Οι σημαίες αυτές αλλάζουν τιμές σύμφωνα με την έναρξη και λήξη του κάθε σταδίου και φυσικά με την έναρξη και λήξη του αλγορίθμου συνολικά. Οι σημαίες αποθηκεύονται σε μια επιπλέον μνήμη που προσθέσαμε η οποία είναι προσβάσιμη και από τους δύο επεξεργαστές(shared BRAM), ώστε να είναι εφικτή η άμεση ενημέρωση του core 1 για οποιαδήποτε αλλαγή. Ακόμη τα δεδομένα που εισάγονται για την επεξεργασία εικόνας που υποστηρίζει ο canny, εισάγονται με την μέθοδο της διαμέρισης, από την εξωτερική μνήμη DDR στην τοπική μνήμη BRAM, του core 0 κι από εκεί επεξεργάζονται με βάση τη τεχνική ανίχνευσης ακμών του canny. Για την τεχνική της διαμέρισης μιλάμε αναλυτικά σε επόμενο κεφάλαιο. Πρόσθέτουμε την ανάγκη μεγάλων μεγεθών μνήμης για την αντιμετώπιση των δυναμικών απαιτήσεων του αλγορίθμου. Γι αυτόν το λόγο δημιουργήσαμε ένα section μνήμης μέσα στην εξωτερική DDR, που αποθηκεύονται τα δυναμικά μεγέθη του αλγορίθμου.

Από κει και πέρα ο έλεγχος που πραγματοποιείτε στηρίζεται στις μετρήσεις που έχουν γίνει, τόσο για την συνολική χρονική έκταση του αλγορίθμου(έναρξη - λήξη) καθώς και την χρονική έκταση του κάθε stage. Εάν το core 1 ελέγξει κάποιο stage εκτός Μέσου Χρόνου Εκτέλεσης(ΜΧΕ), τότε επαναλαμβάνει τη διαδικασία του monitoring, έως συνολικά 3 φορές. Οι τιμές των Μέσων Χρόνων Εκτέλεσης για κάθε stage και συνολικά για τον αλγόριθμο βρίσκονται συγκεντρωμένες στον **Πίνακα 1.**

Επιπλέον δημιουργήσαμε με VHDL ένα Block υλικού(Control Management Block - CMB), το οποίο συνδέσαμε μέσω FSL με το core 1, ώστε όταν προκύπτει εύρεση σφάλματος, ο core 1 να δίνει σήμα στο Reset Block και εκείνο να επανεκκινεί τον core 0 ή να αντιγράφει ή να τροποποιεί τον κώδικα του επεξεργαστή αναλόγως τις ανάγκες του χρήστη, στο stage που έχει αναγνωρισθεί το σφάλμα. Το R.B. είναι συνδεδεμένο μέσω πολυπλεκτών τόσο με την μνήμη του core 0 όσο και με τον ίδιο τον επεξεργαστή. Το CMB έχει τη πρόσβαση να επηρεάσει και την τοπική μνήμη του core 0 προκειμένου να γράψει τα νέα δεδομένα, μόλις κριθεί απαραίτητο. Επομένως το CMB συνδέεται από την μία με τον επεξεργαστή που είναι επιφορτισμένος με τη διαδικασία του monitoring και του fault detection, μέσω FSL, κι από την άλλη με τον επεξεργαστή που υποστηρίζει την εφαρμογή που έχουμε για δοκιμή, επίσης με την τοπική μνήμη του core 0 καθώς και με τον controller της μνήμης. Οι τελευταίες συνδέσεις γίνονται μέσα από τα σύρματα του FPGA.



Σχήμα 4.3: Γενική αρχιτεκτονική και λειτουργίες του συστήματος

5. Περιγραφή Αρχιτεκτονικής/HW

Η αρχιτεκτονική μας στηρίζεται στην βασική αρχιτεκτονική που έχει διαμορφώσει η Xilinx στα συστήματα της. Βασιζόμαστε δηλαδή τους **επεξεργαστές microblaze 7.30 a**, οι οποίοι συνδέονται με το PLB Bus, ώστε να έχουν πρόσβαση με τα περιφερειακά και τις εξωτερικές μνήμες. Η ευχρηστία των FPGAs και η προχωρημένη αναπτυξιακή πλακέτα της Xilinx(ml 405) μας επιτρέπουν την προσθήκη αρκετών επεξεργαστών καθώς και τον παράλληλο προγραμματισμό τους. Η συχνότητα του ρολογιού μας βρίσκεται στα 100MHZ, ένας κύκλος ρολογιού μετριέται στα 10ns.

5.1 Μνήμες

Ο κάθε microblaze επεξεργαστής συνδέεται με την **τοπική του μνήμη BRAM.** Η χωρητικότητα των τοπικών μνήμων βρίσκεται στα **64k για τον microblaze_0** και στα **32k για τον microblaze_1.** Να σημειώσουμε πως οι ανάγκες μας σε χωρητικότητα δεν εξαντλούνται μόνο στη χρήση των δύο τοπικών μνημών. Για την εξυπηρέτηση της μεθοδολογία μας δημιουργήσαμε ακόμη την κοινόχρηστη, **shared BRAM με χωρητικότητα 8k.** Επιπλέον για την αποθήκευση των δεδομένων του αλγορίθμου, που είναι προς επεξεργασία, χρησιμοποιούμε την **εξωτερική DDR χωρητικότητας 32M.** Από την εξωτερική μνήμη, διαμερισμένα τα δεδομένα μας φτάνουν στην τοπική μνήμη BRAM. Επίσης γίνεται και η χρήση της **SRAM με χωρητικότητα 1M,** στην οποία αποθηκεύονται κομμάτια κώδικα που μπορούν να χρησιμοποιηθούν για την αντικατάσταση μέρους κώδικα που έχει ανιχνευτεί σφάλμα.

5.2 Περιφερειακά και buses

Η Xilinx για την επικοινωνία επεξεργαστών και περιφερειακών χρησιμοποιεί το **PLB Bus,** στο οποίο συνδέονται, επεξεργαστές, εξωτερικές ή κοινόχρηστες – εσωτερικές μνήμες και τα περιφερειακά. Από τα περιφερειακά που συνδέονται στο PLB Bus εμείς χρησιμοποιούμε τον **timer/counter** της Xilinx και για μετρήσεις του χρόνου εκτέλεσης των stages και του αλγορίθμου που τρέχουμε για τις δοκιμές. Η επικοινωνία των επεξεργαστών και των περιφερειακών συγχρονίζεται από τοποθετημένους arbiters(διαιτητές) που είναι υπεύθυνοι για το ποιος θα «μιλήσει» πρώτος σύμφωνα με το που θέλει να «μιλήσει».

5.3 Επεξεργαστές και επικοινωνία

Για τις ανάγκες της εργασίας μας, τοποθετούμε στο σύστημα μας **δύο microblaze.** Τον monitor microblaze(core 1) και τον under test microblaze(core 0) και οι δύο προγραμματισμένοι στην ίδια συχνότητα. Εξυπηρετούνται από τις μνήμες που αναφέραμε παραπάνω, αλλά προκύπτει και η ανάγκη άμεσης επικοινωνίας μεταξύ τους. Αυτό επιτυγχάνεται με τη χρήση του FSL bus που μας παρέχει η Xilinx. Το FSL δίνει τη δυνατότητα με τη χρήση κατάλληλων εντελών να στέλνουν μηνύματα οι επεξεργαστές μεταξύ τους. Κάθε επεξεργαστής προγραμματίζεται κατάλληλα αναλόγως τις απατήσεις που έγκειται από την χρησιμότητα του.

5.4 Control Management Block

Η μεθοδολογία μας στηρίζεται στην δυνατότητα της, σε πραγματικού χρόνου, παρακολούθησης άρα απαιτείται και σε πραγματικό χρόνο η αντιμετώπιση σφαλμάτων που πιθανά ανιχνεύονται. Τη λειτουργία λοιπόν της άμεσης διόρθωσης ή πιθανής αναχαίτισης σφαλμάτων έρχεται να εξυπηρετήσει ένα block υλικού που δημιουργήσαμε, **το Control Management(CMB).**

5.4.1 Η λειτουργία του CMB

Η λειτουργία του block εξαρτάται από τα αποτελέσματα του monitoring που πραγματοποιεί το core 1. Επομένως απαιτείτε σύνδεση του block με τον core 1, την οποία και επιτυγχάνουμε με FSL bus.

5.4.1.1 Η λειτουργία του CMB σε σχέση με το core1

Το CMB δέχεται από το FSL 3 λέξεις των 32 bit. Για εμάς η **πρώτη λέξη είναι η εντολή(command)** για το πώς θα λειτουργήσει το block, **η δεύτερη δίνει την επόμενη διεύθυνση γραφής** που πρέπει να ξεκινήσει και να τελειώσει η αντικατάσταση του κώδικα και **η τρίτη λέξη μας δίνει τα δεδομένα** που έρχονται από το επεξεργαστή.

Υπόδειγμα VHDL κώδικα για τη λειτουργία του CMB

```
1  if (nr_of_reads = 2) then
2                                        -- command
3          if (unsigned(FSL_S_Data) = X"00000001") then
4                                              reset_enable <= '1' ;
5                                        end if ;
6          if (unsigned(FSL_S_Data) = X"00000002") then
7                                              reset_enable <= '0' ;
8                                        end if ;
9          if (unsigned(FSL_S_Data) = X"00000003") then
10                                             move_enable <= '1' ;
11                                       end if ;
```

Εφόσον το CMB δεχτεί την ***πρώτη λέξη(nr_of_reads=2)*:**

- Εάν λοιπόν το block δεχτεί από το FSL την τιμή λέξης ‘1’, τότε ενεργοποιείται το σήμα *reset_enable* ώστε να γίνει επανεκκίνηση στην μνήμη του core 0.
- Εάν η λέξη έρθει ‘2’ τότε απενεργοποιείτε το *reset_enable*
- Εάν η λέξη έρθει ‘3’ τότε ξεκινά η μεταφορά του νέου κώδικα με την ενεργοποίηση του σήματος *move_enable*.
- Στην τιμή ‘4’ το *move_enable* απενεργοποιείται.
- Στην τιμή ‘5’ παίρνει την διεύθυνση από την οποία θα ξεκινήσει η αντικατάσταση του fault stage, *muxAddrStart*
- Τέλος στην τιμή ‘6’ θα πάρει την διεύθυνση στην οποία θα ολοκληρώσει την αντικατάσταση.

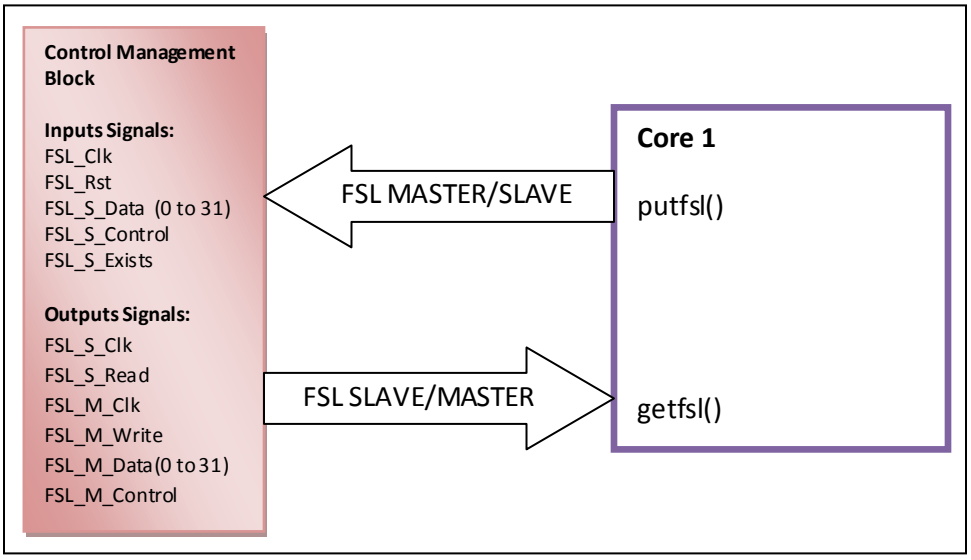
Στην δεύτερη λέξη *(nr_of_reads=1)*:

Αυξάνουμε κατά μια τη θέση μνήμης στην οποία γράφουμε τα νέα δεδομένα κατά την αντικατάσταση: *mux_addr <= mux_addr + X"00000004";*

Τέλος στην Τρίτη λέξη*(nr_of_reads=3)*:

Το CMB δέχεται τα δεδομένα που πρόκειται να τοποθετήσει στην θέση των παλιότερων *input_data <= FSL_S_Data ;*

Από τον core 1 δίνονται οι τιμές με την εντολή putfsl. Παρακάτω υπάρχει ένα σχετικό σχήμα για την επικοινωνία των δύο IPs,με τα σήματα εισόδου – εξόδου του RB, που εξυπηρετούν τις παραπάνω λειτουργικές ανάγκες.



Σχήμα 5.1: Επικοινωνία CMB με core1, μέσω FSL. Inputs /Outputs σήματα του CMB

5.4.1.2 Η λειτουργία του σε σχέση με τον core 0

Η Παραπάνω λειτουργίες του έχουν να κάνουν αποκλειστικά με τα μηνύματα που λαμβάνει το RB από το core 1 . Από την άλλη πλευρά όμως το CMB πρέπει να γράψει δεδομένα στην μνήμη του core 0 από πού θα τα επεξεργαστεί στη συνέχεια ο επεξεργαστής. Επίσης είναι σημαντικό να προχωρήσει και σε επανεκκίνηση της λειτουργίας του core 0. Για να συμβούν όλα αυτά έπρεπε να προσθέσουμε υλικό και να τροποποιήσουμε κατάλληλα την υπάρχουσα συνδεσιμότητα.

Για την διαδικασία αυτή προσθέσαμε 2 πολυπλέκτες στο σύστημα μας. Έναν ανάμεσα στον RB, τα σήματα reset του συστήματος και τον επεξεργαστή με τον controller της μνήμης και έναν ανάμεσα στον controller της μνήμης και την μνήμη.

Επομένως από το RB για το core 0 και την μνήμη του φεύγουν συνολικά 6 σήματα του core 0:

1. Το **MB0Reset** που είναι υπεύθυνο ενεργοποιείται όταν πρέπει το CMB να γράψει στην μνήμη του core 0. Διότι για να γράψει κάποιος άλλος στην μνήμη του επεξεργαστή πρέπει αρχικά να σταματήσει ο ίδιος ο επεξεργαστής να χρησιμοποιεί την μνήμη του, αλλά και ο controller της μνήμης. Το συγκεκριμένο σήμα οδηγείτε σε πολυπλέκτη ο οποίος θα το απελευθερώσει εφόσον το MuxSel ενεργοποιηθεί

2. Το **MuxSel** ενεργοποιείτε στην περίπτωση που το CMB πρέπει να μεταφέρει δεδομένα στην μνήμη του επεξεργαστή core 0. Μόλις ολοκληρωθεί δηλαδή η διαδικασία του monitoring, ανιχνευτεί fault stage και δοθεί η διεύθυνση από την οποία θα ξεκινήσει η τροποποίηση του κώδικα, ενεργοποιείται και το συγκεκριμένο σήμα, ώστε από τους τοποθετημένους πολυπλέκετες να περάσουν στις εξόδους τους τα σήματα που έρχονται από το CMB. Συνδέεται στις εισόδους sel και selmux των δύο πολυπλεκτών(στους επιλογείς).

Ενεργοποίηση MuxSel στο CMB

```
1  if (reset_enable = '1') then
2    mux_sel          <= '1';
3    mb0_reset        <= '1';
4    mux_enable       <= '1';
5        else
6    mux_sel          <= '0';
7    mb0_reset        <= '0';
8   mux_enable        <= '0';
9        end if;
```

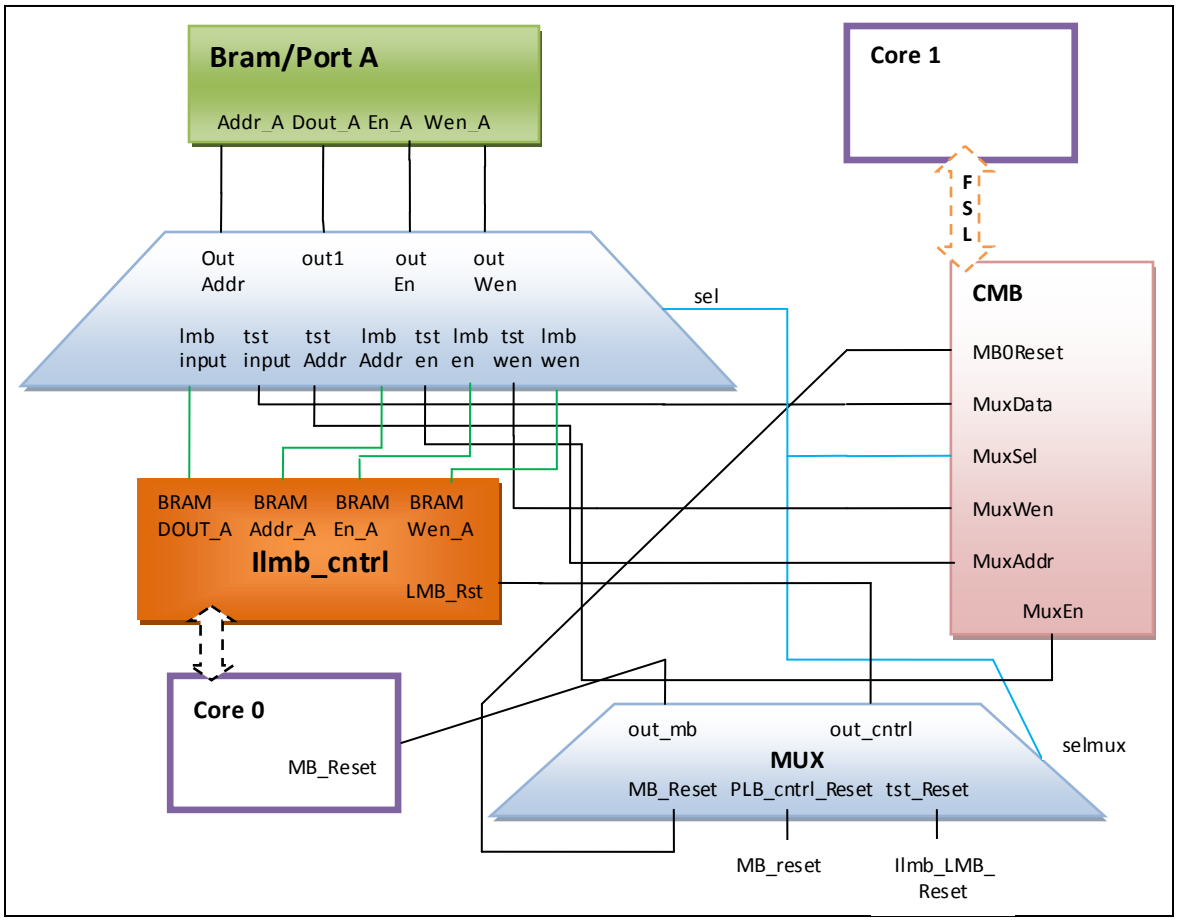
3. **MuxData,** 32bit σήμα που μεταφέρει τα δεδομένα αντικατάστασης από το CMB στην είσοδο **tst_input** του πολυπλέκτη ανάμεσα στον controller και την τοπική BRAM. Όταν ενεργοποιείται το sel περνάει από την έξοδο **out1** του πολυπλέκτη για να φτάσει στην είσοδο **Dout_A** της PORT A, της μνήμης BRAM.

4. **MuxWen,** 4bit σήμα που συνδέετε με το **tst_wen** του πολυπλέκτη πριν την μνήμη**.** Ενεργοποιείται όταν είναι ανάγκη να γράψει στην μνήμη το CMB, περνάει από την έξοδο **out_wen** του πολυπλέκτη και παέι στην είσοδο **Wen_A** της BRAM.

5. **MuxAddr,** 32bit σήμα που δίνει μέσω του ίδιου πολυπλέκτη τη διεύθυνση που θα γράφουν τα νέα δεδομένα, συνδέεται με το **tst_Addr** του πολυπλέκτη και από την έξοδο του **out_addr** φτάνει στο **Addr_A** της BRAM.

6. Τέλος το **MuxEn** ενεργοποιείται όταν ενεργοποιείται το MuxSel δίνοντας το αντίστοιχο enable σήμα και στην μνήμη του core 0.

Αναλυτική σχηματική προσέγγιση γίνεται στο παρακάτω **σχήμα 5.2** καθώς και η συνδεσμολογία μεταξύ των IPs.



Σχήμα 5.2: Συνδεσμολογία του Control Management Block με τα IPs

6. Περιγραφή Λογισμικού/SW

Η μεθοδολογία μας όπως εξηγήσαμε, και ποιο πάνω στηρίζεται και στην ανάπτυξη ειδικού λογισμικού. Από την μία ο ένας επεξεργαστής(core 0) υποστηρίζει τις λειτουργίες της εύρεσης ακμών του canny. Από την άλλη ο επεξεργαστής(core 1) υποστηρίζει την λειτουργία της παρακολούθησης και της ανίχνευσης σφαλμάτων, για τον πρώτο. Σε αυτό το κεφάλαιο θα μιλήσουμε αναλυτικά και για τους δύο αλγορίθμους.

6.1 Canny αλγόριθμος

Πρόκειται για ανιχνευτή ακμών, ο canny είναι ένας διαχειριστής ανίχνευσης ακμών που χρησιμοποίει αλγόριθμο πολλαπλών σταδίων για να ανιχνεύσει μεγάλο εύρος των ακμών σε εικόνες. Αναπτύχτηκε από τον John Canny, από κει και το όνομα του. Για τις δικές μας ανάγκες διαχωρίσαμε τον αλγόριθμο σε 5 διαφορετικά στάδια, σύμφωνα λοιπόν με αυτόν τον διαχωρισμό θα προχωρήσουμε στην ανάλυση του αλγορίθμου.

6.1.1 Stage 0 – Μεταφορά δεδομένων – Διαμέριση

Στο στάδιο αυτό ουσιαστικά πραγματοποιείται η διαμέριση των δεδομένων που έρχονται σε επεξεργασία από τον αλγόριθμο. Με λίγα λόγια η εικόνα μας, που το μέγεθος της βρίσκεται στο 100x100 και είναι αποθηκευμένη στην εξωτερική DDR, διαμερίζεται σε εικόνες των 10x10, τις οποίες περνάμε στην τοπική BRAM του microblaze. Ο κώδικας της συγκεκριμένης διαδικασίας φαίνετε παρακάτω:

Κώδικας διαμέρισης - μεταφοράς δεδομένων από την DDR στην BRAM.

```
1  for(i=0; i<ROWSMAX; i++){
2          imgdata[i] = rand() % 255 + 0 ;
3              }
.                                                    .....
.
5
6  for (x=0; x<ROWS; x++){
7          for (y=0; y<COLUMNS; y++){
8          for(i=0; i<r_c_max; i++){
9                  imgdata2[x][y] = imgdata[i] ;
10                 im->data[x][y]=imgdata2[x][y] ;      }
11         }
```

Να σημειώσουμε πως η αρχικοποίηση της εικόνας γίνεται στην DDR, *unsigned char *imgdata=0x86000004 ;* Αντιθέτως η διαμερισμένη εικόνα βρίσκεται στην τοπική μνήμη του επεξεργαστή, *unsigned char imgdata2[10][10] ;*

Ακόμη να προσθέσουμε πως κατά τη διάρκεια του συγκεκριμένου σταδίου ο αλγόριθμος για πρώτη φορά κάνει χρήση της βασικής δομής, για την λειτουργία του, της *newimage, im = (IMAGE)newimage (ROWS, COLUMNS);*

Η *newimage* είναι η δομή με την οποία ο αλγόριθμος δεσμεύει χώρο στην μνήμη ανάλογα με το μέγεθος της εικόνας, γι αυτό και ως ορίσματα δέχεται τις μεταβλητές που υποδηλώνουν τα μεγέθη της εικόνας. Για την εξυπηρέτηση του ιδιαίτερα απαιτητικού αυτού δυναμικού τρόπου δέσμευσης χώρου στην μνήμη, μετά από αρκετές δοκιμαστικές τεχνικές, καταλήξαμε στην δημιουργία ειδικού χώρου μνήμης, μέσα στην DDR.

Κάλεσμα newimage για δέσμευση χώρου

```
1  im = (IMAGE)newimage (ROWS, COLUMNS);
2  im->info->oi = PBM_SE_ORIGIN_ROW;
3  im->info->oj = PBM_SE_ORIGIN_COL;
4  PBM_SE_ORIGIN_ROW = 0;
5  PBM_SE_ORIGIN_COL = 0;
```

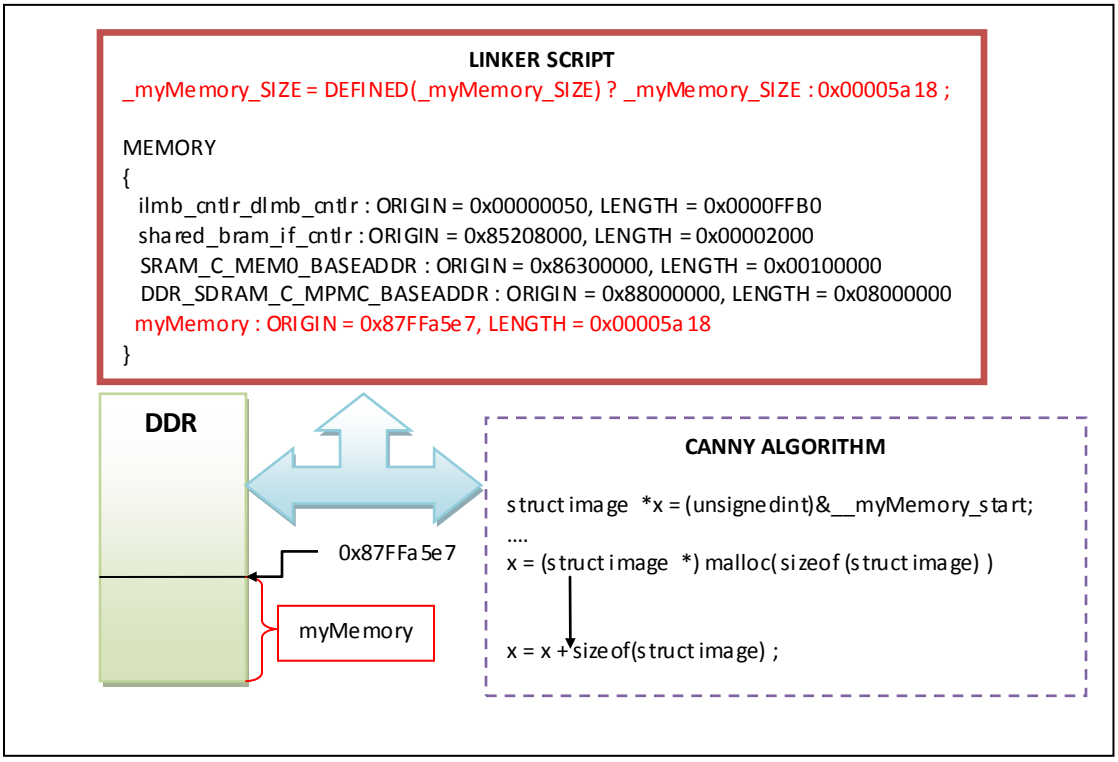
Η newimage – δημιουργία νέας εικόνας στην μνήμη

```
1  /*     The IMAGE data structure        */
2   struct image {
3                          struct header *info;          //Pointer to header
4                          unsigned char **data;          //Pixel values
5  };
6  typedef struct image * IMAGE
7
8  struct image  *newimage (int nr, int nc)
9  {
10              struct image  *x = (unsigned int)&__myMemory_start;      // New image
11             x->info = (unsigned int)&__myMemory_start + 1351;
12             x->data = (unsigned int)&__myMemory_start + 2703;
13       unsigned char *ptr;            // new pixel array
14       int i;
15       x->data[i] = (unsigned int)&__myMemory_start + 4054;
16       if (nr < 0 || nc < 0) {
17             xil_printf ("Error: Bad image size (%d,%d)\n", nr, nc);
18             return 0;
19       }
20 /*     Allocate the image structure    */
21 x = (struct image  *) malloc( sizeof (struct image) );
22
23
24
25
```

6.1.1.1 Δημιουργία μνήμης εξυπηρέτησης εξυπηρέτησης δυναμικών αναγκών

Να σημειώσουμε πως πρόκειται για μια δυναμική εφαρμογή. Δηλαδή ο χώρος της μνήμης που απαιτείται καταλαμβάνεται κατά τη διάρκεια εκτέλεσης του αλγορίθμου. Αυτό δημιουργεί υψηλές απαιτήσεις στη διάθεση μνήμης και καταμερισμού των sections εντός της μνήμης. Αντιμετωπίσαμε πολλά προβλήματα για να καταφέρουμε να ελέγξουμε τις ανάγκες της δυναμικής επέκτασης του αλγορίθμου στην μνήμη. Μετά από αρκετές δοκιμές καταλήξαμε στο να δημιουργήσουμε μια ειδική μνήμη μέσα στην εξωτερική μνήμη DDR. Καταλάβαμε δηλαδή χώρο της DDR για να εξυπηρετήσουμε τις ανάγκες του αλγορίθμου με τέτοιο τρόπο ώστε να μπορεί να εκτελεστεί πολλαπλά χωρίς περιορισμούς στο heap κομμάτι της μνήμης.

Αυτό που κάναμε ήταν ότι δέσμευε από τον αλγόριθμο, χώρο στην μνήμη δυναμικά(malloc) να δεσμεύει χώρο πλέον στο section μνήμης που σχεδιάσαμε στην DDR. Ο χώρος που δεσμεύσαμε ήταν όσος απαιτείται για την κάλυψη των αναγκών μίας εκτέλεσης του canny, συνολικά 23064bytes. Κάθε φορα που ο canny καλεί την δομή newimage ανακαταλαμβάνει το ίδιο section μνήμης κι έτσι δεν απαιτείτε επιπλέον χώρος, όπως στην περίπτωση που δεσμευόταν χώρος στην μνήμη με την malloc **(Σχήμα 6.1).** Για την δημιούργια της ειδικής μνήμης χρειάστηκε να επέμβουμε στο linker script του microblaze.



Σχήμα 6.1: Δημιουργία μνήμης για την εξυπηρέτηση δυναμικών αναγκών

6.1.2 Stage 1/2 – Έλεγχος διαθεσιμότητας μνήμης

Στην επόμενη φάση ελέγχουμε – ο ίδιος αλγόριθμος κάνει τον έλεγχο – εάν ο χώρος που ζητάμε για τη δημιουργία δύο εικόνων στην BRAM επαρκή, εάν δεν επαρκή τυπώνεται σχετικό μήνυμα. Ο χώρος μνήμης που απαιτείτε για τις εικόνες δεσμεύεται επίσης δυναμικά, καλώντας την *newimage*. Αυτές οι δύο εικόνες είναι οι βασικές εικόνες πάνω στις οποίες γίνεται η επεξεργασία για την εύρεση ακμών και το μέγεθος τους εξαρτάται από τις απαιτήσεις του χρήστη, είναι οι *magim*, και *oriim*.

Δημιουργία τοπικών εικόνων στην BRAM

```
1  magim = newimag(im->info->nr, im->info->nc);
2    if (magim == NULL)
3              {
4                xil_printf ("Out of storage: Magnitude\r\n");
5                exit (1);
6              }
7
8              oriim = newimag(im->info->nr, im->info->nc);
9    if (oriim == NULL)
10             {
11               xil_printf ("Out of storage: Orientation\r\n");
12               exit (1);
13             }
```

6.1.3 Stage 3/4 – Εκτέλεση των βασικών συναρτήσεων Canny και Hysteresis

Αφού περάσουμε από τον έλεγχο επάρκειας της τοπικής μνήμης, συνεχίζουμε στο βασικό κομμάτι του κώδικα, κατά το οποίο καλούνται οι βασικές συναρτήσεις του αλγορίθμου, *canny* και *hysteresis*, οι οποίες κάνουν τον έλεγχο για την εύρεση ακμών στην εικόνα που εισάγαμε παραπάνω.

Εκτέλεση canny και hysteresis αλγορίθμων

```
1  canny (s, im, magim, oriim);
2     for(q=0;q<800000;q++);
3  hysteresis (high, low, im, magim, oriim);
```

Ο canny με τη χρήση του φίλτρου του gauss εξομαλύνει τα χαρακτηριστικά της εικόνας, *gau[i] = meanGauss ((float)i, s);*, και στη συνέχεια κάνει τη συνέλιξη της γνήσιας εικόνας με την εικόνα που έχει υποστεί το φιλτράρισμα, *seperable_convolution (im, gau, width, smx, smy);*, και τέλος δημιουργεί μια νόρμα με βάση τις διαστάσεις τις εικόνας, *z = norm (dx[i][j], dy[i][j]);*. Παρακάτω ακολουθεί το μεγαλύτερο κομμάτι του κώδικας της συνάρτησης.

Η συνάρτηση canny

```
1  void canny (float s, IMAGE im, IMAGE mag, IMAGE ori)
2  {
3  ……
4  /* Create a Gaussian and a derivative of Gaussian filter mask */
5            for(i=0; i<MAX_MASK_SIZE; i++)
6            {
7             gau[i] = meanGauss ((float)i, s);
8             if (gau[i] < 0.005)
9             {
10                     width = i;
11                     break;
12            }
13            dgau[i] = dGauss ((float)i, s);
14           }
15 ……
16 /* Convolution of source image with a Gaussian in X and Y directions  */
17           seperable_convolution (im, gau, width, smx, smy);
18
19 /* Now convolve smoothed data with a derivative */
20           dx = f2d (im->info->nr, im->info->nc);
21           dxy_seperable_convolution (smx, im->info->nr, im->info->nc,
22                    dgau, width, dx, 1);
23           free(smx[0]); free(smx);
24
25           dy = f2d (im->info->nr, im->info->nc);
26           dxy_seperable_convolution (smy, im->info->nr, im->info->nc,
27                    dgau, width, dy, 0);
28           free(smy[0]); free(smy);
29
30 /* Create an image of the norm of dx,dy */
31           for (i=0; i<im->info->nr; i++)
32            for (j=0; j<im->info->nc; j++)
33            {
34               z = norm (dx[i][j], dy[i][j]);
35               mag->data[i][j] = (unsigned char)(z*MAG_SCALE);
36            }
37 ……
38 }
```

Στη συνέχεια έρχεται να εκτελεστεί η hysteresis που αναλαμβάνει να βρει τις ακμές τις εικόνας . Αρχικά μηδενίζει τα pixels της εικόνας *im->data[i][j] = 0;*, συνεχίζει καλώντας την συνάρτηση trace, *trace (i, j, low, im, mag, oriim);*, η οποία σύμφωνα με το αυτό-οριζόμενο threshold θα βρει τα pixels εκείνα που συντελούν τις ακμές τις εικόνας και θα τα τους δώσει την τιμή 255 *if (im->data[i][j] == 0) im->data[i][j] = 255*

Η συνάρτηση Hysteresis

```
1   void hysteresis (int high, int low, IMAGE im, IMAGE mag, IMAGE oriim)
2   {
3               for (i=0; i<im->info->nr; i++)
4                 for (j=0; j<im->info->nc; j++)
5                   im->data[i][j] = 0;
6   if (high<low)
7               {
8                 estimate_thresh (mag, &high, &low);
9                 xil_printf ("Hysteresis thresholds (from image): HI %d LOW %D\r\n",
10  high, low);
11              }
12              for (i=0; i<im->info->nr; i++)
13                for (j=0; j<im->info->nc; j++)
14                  if (mag->data[i][j] >= high)
15                    trace (i, j, low, im, mag, oriim);
16
17  /* Make the edge black (to be the same as the other methods) */
18              for (i=0; i<im->info->nr; i++)
19                for (j=0; j<im->info->nc; j++)
20                  if (im->data[i][j] == 0) im->data[i][j] = 255;
21                  else im->data[i][j] = 0;
22  }
```

6.2 Μεθοδολογία monitoring – fault detection

Προχωράμε στην ανάλυση της μεθοδολογίας μας σε σχέση με τον έλεγχο για εμφάνιση λαθών στον παραπάνω αλγόριθμο. Υπενθυμίζουμε πως ο επεξεργαστής που υποστηρίζει τον έλεγχο είναι ο core 1, επομένως εκεί γράφουμε και τον σχετικό κώδικα.

6.2.1 Περιγραφή Μεθοδολογίας - Κοινόχρηστες Σημαίες(flags)

Για κάθε ένα από τα παραπάνω stages προχωράμε στον χρονικό τους έλεγχο. Σύμφωνα δηλαδή με τον Μέσο Χρόνο Εκτέλεσης που μετρήσαμε(λεπτομέρειες στην επόμενη ενότητα). Το ερώτημα είναι πως ελέγχουμε αυτή τη διαδικασία. Ελέγχουμε τη διαδικασία με τη χρήση κοινόχρηστων σημαιών ελέγχου(flags). Δημιουργούμε δείκτες οι οποίοι «δείχνουν» σε διευθύνσεις της κοινόχρηστης BRAM(shared Bram). Ο κάθε ένας από αυτούς τους δείκτες εξυπηρετεί και ένα από τα στάδια που αναλύσαμε προηγουμένως.

Οι δείκτες – σημαίες στην κοινόχρηστη BRAM.

```
1  u32 *flag_magnitude     = (u32 *) 0x85208000     + 4 ;
2  u32 *flag_orientation   = (u32 *) 0x85208000     + 5 ;
3  u32 *flag_main          = (u32 *) 0x85208000     + 6 ;
4  u32 *flag_canny         = (u32 *) 0x85208000     + 7 ;
5  u32 *flag_hyster        = (u32 *) 0x85208000     + 9 ;
```

Χρησιμότητα βασικών κοινόχρηστων δεικτών:
flag_magnitude: Έλεγχος επάρκειας τοπικής μνήμης, για την εικόνα maimag

flag_orientation: Έλεγχος επάρκειας τοπικής μνήμης, για την εικόνα oriimag

flag_main: Ελέγχει αν έχει ξεκινήσει η λειτουργία του αλγορίθμου.

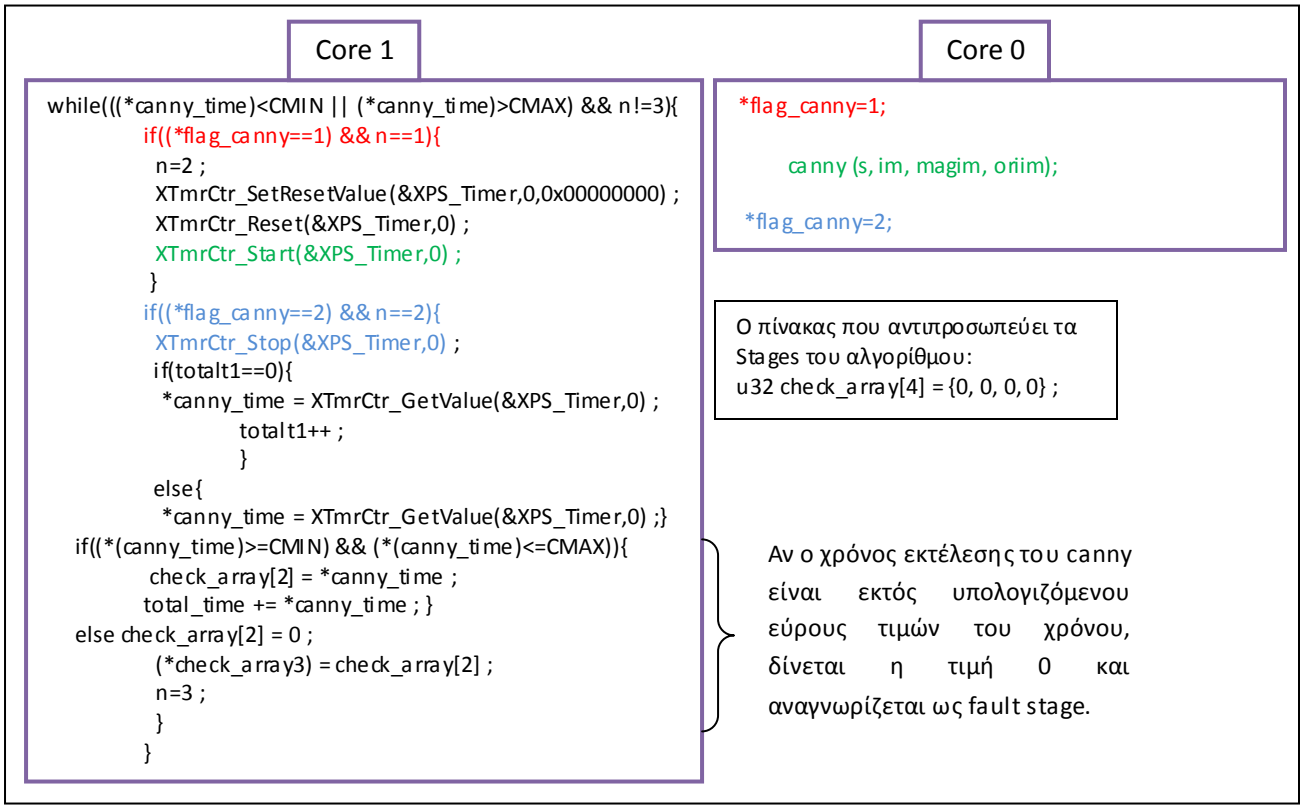
flag_canny: Έλεγχος έναρξης – λήξης της συνάρτησης canny.

flag_hyster: Έλεγχος έναρξης – λήξης της συνάρτησης hysteresis.

Παίρνοντας ως παράδειγμα τη σημαία *flag_canny* ας δούμε πως δουλεύει η μεθοδολογία μας και για τους δύο επεξεργαστές.

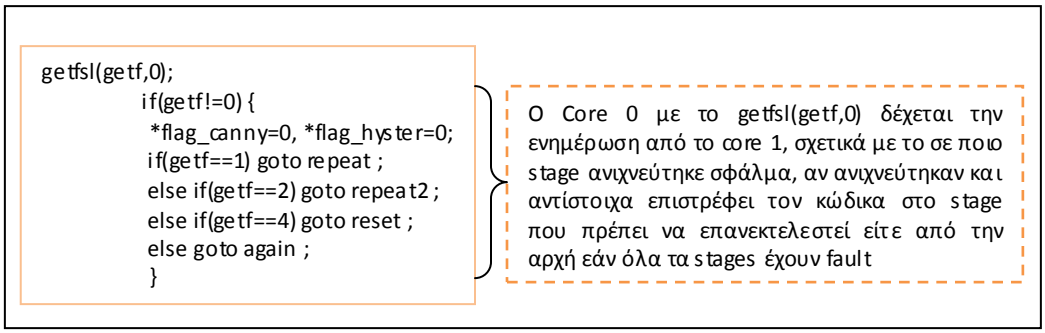
Προτού καλεστεί η συνάρτηση από το πρόγραμμα τοποθετείτε η σημαία που σχετίζεται με το συγκεκριμένο stage(stage 3), *flag_canny*, αλλάζοντας του την τιμή, *flag_canny=1;*. Την ίδια στιγμή ο core 1 ελέγχει αν αλλάξει η τιμή της σημαίας. Αν αλλάξει η τιμή της σημαίας και γίνει ίση με ‘1’, ενεργοποιείτε ο μετρητής, *XTmrCtr_Start(&XPS_Timer,0) ;*, και ξεκινά η μέτρηση για τον χρόνο εκτέλεσης του canny. Η μέτρηση ολοκληρώνεται εφόσον η τιμή της σημαίας γίνει ίση με ‘2’, *XTmrCtr_Stop(&XPS_Timer,0) ;* δηλαδή αφόσον ολοκληρωθεί ο canny. Σε κάθε περίπτωση διασφαλίζεται πως η μέτρηση θα εκκινήσει μόνο μια φορά, χωρίς να μηδενιστεί στο ενδιάμεσο και να επανεκκινήσει ο μετρητής, με τη χρήση τοπικών μεταβλητών σε ρόλο σημαιών.

Αφού ολοκληρωθεί η μέτρηση κρατάμε την τιμή που προκύπτει, **canny_time = XTmrCtr_GetValue(&XPS_Timer,0) ;*, σε μία επίσης κοινόχρηστη μεταβλητή, της οποίας η τιμή αποθηκεύετε σε ένα πίνακα, οπού κάθε του θέση αντιπροσωπεύει και ένα stage του αλγορίθμου. Εφόσον λοιπόν – για το stage που μελετάμε – ο χρόνος που υπολογίστηκε είναι μικρότερος ή μεγαλύτερος από την μέγιστη ή ελάχιστη τιμή, **σε σχέση με τον μέσο χρόνο κατά +/-15%,** τότε η τιμή του stage στον πίνακα θα γίνει ‘0’, ώστε να αναγνωριστεί ως fault stage. Αν όμως ο χρόνος βρίσκεται μέσα στο εύρος των τιμών η τιμή γίνεται ίση με τον χρόνο εκτέλεσης που υπολογίστηκε.

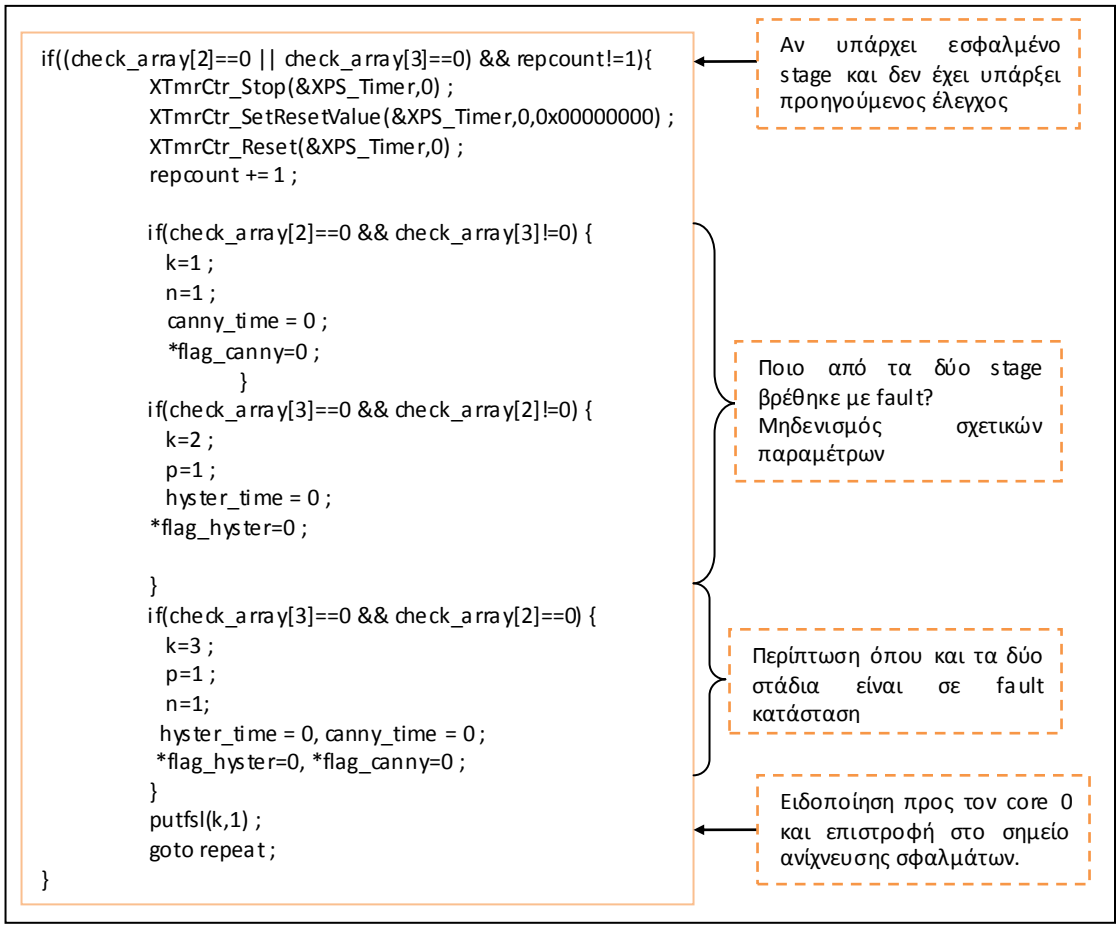


Σχήμα 6.2: Ανίχνευση Σφαλμάτων στο stage 3 – Σχετικός κώδικας στους core1 και core 0

Αφού ολοκληρωθεί η ανίχνευση πιθανών σφαλμάτων, ελέγχουμε αν και ποιο στάδιο του αλγορίθμου παρουσίασε προβληματική συμπεριφορά. Στην περίπτωση εκείνη που όλα τα στάδια έτρεξαν ορθά, ο αλγόριθμο μας τυπώνει σχετικό μήνυμα. Στην περίπτωση που κάποιο stage του αλγορίθμου έχει σημειωθεί ίσο με '0', τότε αφού μηδενιστούν όλες οι μεταβλητές που σχετίζονται με το stage, επαναλαμβάνετε η παραπάνω διαδικασία, έως και για 2 φορές επαναληπτικά, και στέλνετε μέσω fsl σχετική ενημέρωση στον core 0, ώστε και εκείνος να επαναλάβει το stage που βρέθηκε εκτός χρόνου. Να σημειωθεί πως εφόσον όλα τα stages βρεθούν εκτός χρόνου, εκτελείτε ξανά ολόκληρος ο αλγόριθμος(reset). Ακολουθούν οι σχετικοί κώδικες **στα σχήματα 6.3 και 6.4 ,** με κατατοπιστικές επεξηγήσεις.



Σχήμα 6.3: Αποδοχή κατάστασης σταδίων



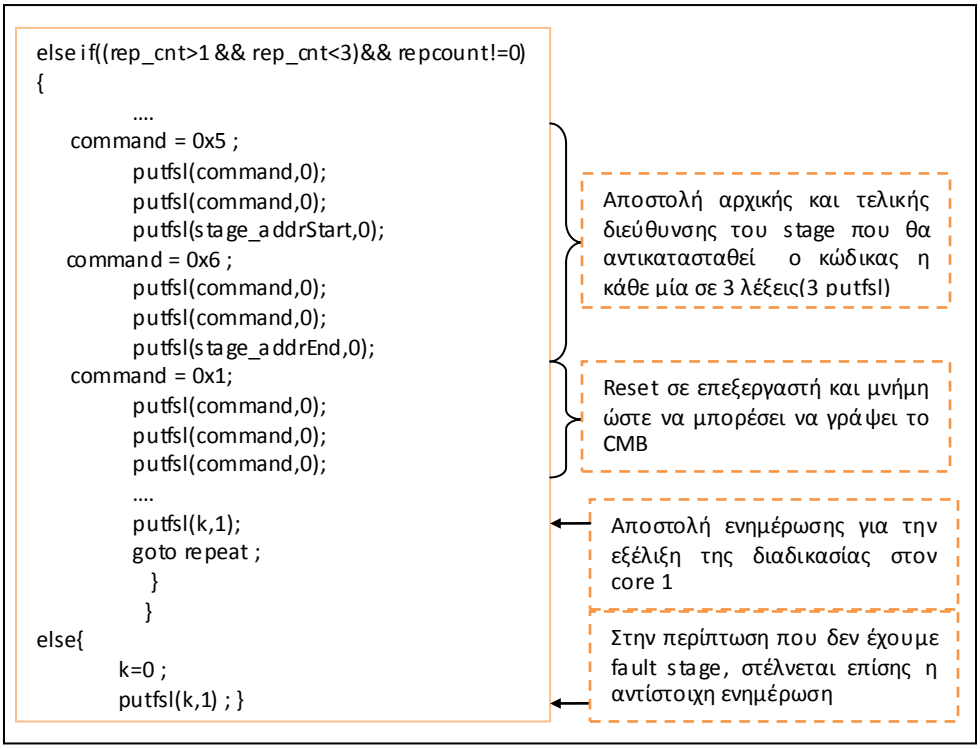
Σχήμα 6.4: Έλεγχος για το ποιο stage είναι εσφαλμένο

6.2.2 Περιγραφή Μεθοδολογίας – Επικοινωνία core 1 με CMB

Τώρα περνάμε στην επόμενη φάση της μεθοδολογίας. Αφού ολοκληρωθεί ο έλεγχος και εφόσον διαγνωσθεί πρόβλημα στη λειτουργία του αλγορίθμου, ο επεξεργαστής στέλνει μέσω fsl, σχετικά μηνύματα στο CMB ώστε να ξεκινήσει η διόρθωση του κώδικα.

Με προϋπόθεση ότι έχει γίνει τουλάχιστον μία φορά όλη η προηγούμενη διαδικασία το core 1, έχοντας το εύρος διευθύνσεων του stage που θα αντικαταστήσει τον κώδικα, ξεκινά να στέλνει σήματα στο CMB. Αρχικά θα του στείλει τις διευθύνσεις*(command=0x5)*, έπειτα θα ζητήσει να γίνει reset στον επεξεργαστή – core 1 - και την μνήμη*(command=0x1)* του ώστε να μπορέσει να γράψει τα δεδομένα, στη συνέχεια*(command=0x3)* θα ξεκινήσει η μεταφορά των νέων δεδομένων, τα οποία βρίσκονται στην SRAM *(sramData = *(dstPoint2 + g) ;)* και αφού ολοκληρωθεί η μεταφορά των δεδομένων θα απενεργοποιηθούν τα σήματα του reset και της μεταφοράς δεδομένων(*command = 0x4; command = 0x2;)*. **Σε όλες τις περιπτώσεις το core 1 στέλνει 3 λέξεις των 32 bit μέσω του fsl.**

Ο κώδικας του παρακάτω **σχήματος 6.5** έρχεται σε απόλυτη συμφωνία με τον κώδικα που αναλύεται στην ενότητα **5.4.1.1** που αφορά τη λειτουργία του CMB. Έπειτα ο αλγόριθμος μας αναλαμβάνει να ενημερώσει τον χρήστη για τα αποτελέσματα της όλης διαδικασίας με μηνύματα στην οθόνη.



Σχήμα 6.5: Αποστολή σημάτων από τον core 1 στο CMB ανάλογα με τα αποτελέσματα του monitoring

7.Αξιολόγηση - Πειράματα - Αποτελέσματα

Η αξιολόγηση της εργασίας μας χωρίζεται σε τρία διακριτά μέρη. Το ένα έχει να κάνει με την αξιολόγηση και τον πειραματισμό της λειτουργίας του canny αλγορίθμου που επιλέξαμε για monitoring, το δεύτερο έχει να κάνει με τα πειράματα που έγιναν συνολικά για το σύστημα που δημιουργήσαμε και τέλος το τρίτο παρουσιάζει τα αποτελέσματα των πειραμάτων.

7.1 Αξιολόγηση Canny algorithm

Αρχικά διαχωρίσαμε τον αλγόριθμο σε 4 βασικά στάδια εκτέλεσης, σύμφωνα με τη συνολική του λειτουργία. Τα στάδια είναι τα εξής:

Stage 0 : Πρόκειται για το 'κατέβασμα' της εικόνας από την εξωτερική μνήμη DDR στην τοπική μνήμη του επεξεργαστή, BRAM. Η διαδικασία αυτή συμβαίνει προκειμένου να διαμερίσουμε την εικόνα μας σε μικρότερες εικόνες. Να σημειωθεί πως αν δεν γίνει η διαμέριση και τα δεδομένα μας φορτωθούν απευθείας από την DDR θα υπάρχει τρομερή καθυστέρηση και ο Μέσος χρόνος εκτέλεσης συνολικά του αλγορίθμου θα αυξηθεί σημαντικά. Ο Μέσος Χρόνος Εκτέλεσης της διαδικασίας υπολογίστηκε στα **0,87msec**.

Stage 1 : Πρόκειται για έλεγχο της μνήμης μας. Ο αλγόριθμος δημιουργεί μια εικόνα σύμφωνα με τις διαστάσεις που εμείς δίνουμε. Η εικόνα αυτή κρατάει τον αντίστοιχο χώρο στην μνήμη μέσα από τη δομή *newimage*. Επομένως χρειάζεται να ελεγχτεί αν η τοπική μνήμη ή ενδεχομένως η εξωτερική μνήμη του επεξεργαστή διαθέτουν την απαιτούμενη χωρητικότητα για την εικόνα. Αν η μνήμη δεν μπορεί να ανταπεξέλθει, τότε αφού αναγνωριστεί σαν *stage fault* το stage 1, προκαλείται *reset* από τον MB1 στον MB0 και η διαδικασία του monitoring επανεκκινεί από την αρχή, ελέγχοντας νέα δεδομένα(διαφορετική εικόνα) αυτή τη φορά. Εάν η διαδικασία επαναληφθεί τρεις συνεχόμενες φορές με το το stage 1 να αναγνωρίζεται σαν *stage fault το* σύστημα σταματά κάθε λειτουργία.

Stage 2 : Ακόμη ο αλγόριθμος απαιτεί τη δημιουργία με τον ίδιο τρόπο μιας όμοιας εικόνας ως *orientation image*. Η ίδια ακριβώς διαδικασία συμβαίνει και στο stage 2 για αυτήν την εικόνα.

Τα δύο παραπάνω stages δεν επιβαρύνουν την εκτέλεση του αλγορίθμου στο χρόνο. Είναι όμως καθοριστικά για τη συνέχεια τόσο του αλγορίθμου όσο και του monitoring.

Stage 3 : Έλεγχος ορθής λειτουργίας της συνάρτησης canny. Ουσιαστικά οι βασικές λειτουργίες του αλγορίθμου, είναι το κάλεσμα της συνάρτησης *canny* καθώς και το κάλεσμα της συνάρτησης *hysteresis*. Από τις δύο αυτές συναρτήσεις εξαρτάται η εκτελεσιμότητα και η ανάπτυξη του αλγορίθμου. Στο stage 3 ελέγχεται το κατά πόσο η συνάρτηση *canny* εκτελείται εντός του προβλεπόμενου Μέσου Χρόνου Εκτέλεσης, ο οποίος υπολογίζεται στα **4,83 msec.** Σε περίπτωση που το συγκεκριμένο stage παρατηρηθεί ως *fault stage* επανεκτελείτε, εφόσον όμως ολοκληρωθεί το monitoring για όλα τα stages. Η διαδικασία αυτή μπορεί να επαναληφτεί έως τρεις φορές, στην τρίτη ίδια περίπτωση τυπώνεται σχετικό μήνυμα που ζητά από τον σχεδιαστή/χρήστη να παρέμβει ο ίδιος για την διόρθωση του συστήματος.

Stage 4 : Έλεγχος ορθής λειτουργίας της συνάρτησης hysteresis. Σε αυτό το stage γίνεται ακριβώς η ίδια διαδικασία με το προηγούμενο, αλλά για την συνάρτηση hysteresis. Ο υπολογιζόμενος Μέσος Χρόνος της συνάρτησης είναι **0,205 msec.**

Ο Συνολικός Μέσος Χρόνος Εκτέλεσης του αλγορίθμου υπολογίζεται στα 0,87(S0) + 4,83(S3) + 0,205(S4)=**5,905msec.** Να σημειωθεί πως οι μετρήσεις αυτές γίνονται για εικόνα μεγέθους 20x20. Παρακάτω στον **Πίνακα 1** βλέπουμε τις μετρήσεις που έχουν πραγματοποιηθεί και για μικρότερη εικόνα(10x10), αλλά και για μεγαλύτερη(30χ30). Είναι διακριτό πως όσο αυξάνονται τα δεδομένα που δέχεται ως είσοδο το σύστημα μας και ο αλγόριθμος, αυξάνετε αντίστοιχα και ο συνολικός χρόνος εκτέλεσης του αλγορίθμου. Πράγμα που σημαίνει πως θα έχουμε μεγαλύτερη καθυστέρηση σε περίπτωση κατά την οποία κάποιο από τα stages αποδειχθεί εσφαλμένο(εκτός μέσου χρόνου)

	T_Avg_10x10 (msec)	T_Avg_20x20 (msec)	T_Avg_30x30 (msec)
workload	100	400	900
Constraints check(S1)	0	0	0
Block transfer (S2)	0,21315	0,87	2,0142675
Canny (S3)	1,195425	4,83	11,2967663
Hysterisis (S4)	0,050225	0,205	0,47462625

Πίνακας 1: Χρόνος εκτέλεσης των stages ανάλογα με το μέγεθος των δεδομένων

Ακόμη ένα συμπέρασμα που προκύπτει από τον παραπάνω πίνακα είναι πως η αύξηση του *T_Avg,* είναι σχετικώς ανάλογη με την αύξηση των δεδομένων που εισάγονται. Παράδειγμα για το stage 4 και μια εικόνα 10x10 το T_Avg = 0,0500225, ενώ για εικόνα 4 φορές μεγαλύτερη από την πρώτη(4*100=400) το T_Avg = 0,205*(4*0,0500225=0,20009).* Επίσης για εικόνα 2,25 φορές μεγαλύτερη από τη δεύτερη(2,25*400=900) το T_Avg = 0,47462625*(2,25*0,205=0,46125).* Ειδικότερα παρατηρούμε **απόκλιση κατά 1,05%** στις μετρήσεις της 30x30 εικόνας.

7.1.1 Ανάλυση αξιολόγησης

Ο αλγόριθμος μετρήθηκε σύμφωνα με την παραπάνω διαμέριση, η οποία έγινε από τους σχεδιαστές, αλλά ανταποκρίνεται στον τρόπο λειτουργίας του αλγορίθμου. Παρακάτω θα εξετάσουμε τον τρόπο υπολογισμού του Μέσου Χρόνου Εκτέλεσης για το κάθε stage του αλγορίθμου.

7.1.1.1 Stage 0

Για τον χρόνο που διαρκεί η μεταφορά της εικόνας από την εξωτερική DDR στην τοπική BRAM ενεργοποιούμε κατάλληλα τον hw timer του συστηματός όπως φαίνετε παρακατω.

Κώδικας για την μέτρηση μεταφοράς της εικόνας από την DDR στην BRAM

```
1  XTmrCtr_SetResetValue(&XPS_Timer,0,0x00000000) ;
2  XTmrCtr_Reset(&XPS_Timer,0) ;
3  XTmrCtr_Start(&XPS_Timer,0) ;
4  for (x=0; x<ROWS; x++){
5      for (y=0; y<COLUMNS; y++){
6          for(i=0; i<r_c_max; i++){
7                          imgdata2[x][y] = imgdata[i] ;
8                          im->data[x][y]= imgdata2[x][y] ;
9                                  }
10                        }
11             }
12 XTmrCtr_Stop(&XPS_Timer,0) ;
```

Η ενεργοποίηση του timer γίνεται την χρήση της συνάρτησης *XTmrCtr_Start()* και με τη χρήση της *XTmrCtr_Stop()* επιτυγχάνουμε τον τερματισμό του. Οι συναρτήσεις αυτές χρησιμοποιούνται διότι η εφαρμογή μας δεν τρέχει πάνω από ένα λειτουργικό σύστημα(OS), επομένως είναι ανάγκη να επικοινωνήσουν οι επεξεργαστές με τους controllers κάθε IP block. Η διαδικασία της μέτρησης γίνεται αποκλειστικά από τον MB0.

7.1.1.2 Stage 1 και Stage 2

Όπως εξηγήσαμε παραπάνω(7.1) τόσο το stage 1 όσο και το επόμενο stage 2 δεν έχουν χρονικές επιπτώσεις στην εκτέλεση του αλγορίθμου μας. Επομένως για τον έλεγχο τους χρειάζεται μόνο η σωστή χρησιμοποίηση των κατάλληλων κοινόχρηστων flags, των δυο επεξεργαστών. Εξάλλου η μεθοδολογία μας είναι βασισμένη στην χρήση κοινόχρηστων μεταβλητών ως flags ελέγχου. Για τα stage 1, 2 χρησιμοποιούμε τα *flag_magnitude* και *flag_orientation*. Τα δύο flags τοποθετούνται στην κοινόχρηστη μνήμη που δημιουργήσαμε. Ο κώδικας και των δύο επεξεργαστών, που σχετίζεται με την συγκεκριμένη διαδικασία παρατίθεται παρακάτω.

Δήλωση των δύο flags ως δείκτες στην κοινόχρηστη μνήμη

```
1  u32 *flag_magnitude      = (u32 *) 0x85208000      + 4 ;
2  u32 *flag_orientation    = (u32 *) 0x85208000      + 5 ;
```

Διαδικασία ελέγχου με τη χρήση των κοινόχρηστων flags(MB0)

```
1  if (magim == NULL)
2              {
3               *flag_magnitude=1;
4               xil_printf ("Out of storage: Magnitude\r\n");
5               exit (1);
6              }
7              *flag_magnitude=2;
8              oriim = newimage(10, 10);
9              if (oriim == NULL)
10             {
11              *flag_orientation=1;
12              xil_printf ("Out of storage: Orientation\r\n");
13              exit (1);
14             }
15             *flag_orientation=2;
```

Διαδικασία ελέγχου με τη χρήση των κοινόχρηστων flags(MB1)

```
1  while(tst!=1){
2   if(*flag_magnitude==1){
3              print("Out of storage: Magnitude\r\n");
4              tst=1 ;
5             }
6   else if(*flag_magnitude==2){
7             check_array[0] = 1;
8             tst=1;
9     }
10  }
11  tst=0;
12  while(tst!=1){
13   if(*flag_orientation==1){
14    print("Out of storage: Orientation\r\n");
15             tst=1 ;
16   }
17   else if(*flag_orientation==2){
18    check_array[1] = 1;
19             tst=1;
20    }
21  }
```

Αναλόγως λοιπόν με την αλλαγή τιμής του flag ελέγχου από τον MB0 ενημερώνεται και στον MB1 με συνέπεια είτε την συνέχεια της διαδικασίας του monitoring είτε το άμεσο τερματισμό της και την εμφάνιση μηνύματος προς τον χρήστη.

7.1.1.3 Stage 3 και Stage 4

Τα επόμενα δύο stages είναι τα βασικότερα για την ορθή εκτέλεση του αλγορίθμου, αλλά και για τη διαδικασία του monitoring. Σε αυτήν τη φάση το monitoring εξαρτάται από τον χρόνο εκτέλεσης του stage. Επομένως η μέτρηση του χρόνου εκτέλεσης των δυο συναρτήσεων που ελέγχουμε αντιστοίχως στα δύο stages έχουν καθοριστική σημασία.

Στο πρώτο stage ο timer γίνεται enable ακριβώς πριν κληθεί η συνάρτηση του canny να εκτελεστεί στον επεξεργαστή MB0. Ο timer ενεργοποιείται όμως από τον MB1 μόλις αλλάξει το σχετικό με τον έλεγχο flag ελέγχου. Η ίδια ακριβώς διαδικασία πραγματοποιείται και για το stage 4 με τον timer να ενεργοποιείται με τον ίδιο τρόπο όπως και για το stage 3.

Για τον έλεγχο κλήσης και τέλους των δύο συναρτήσεων χρησιμοποιούμε δυο κοινόχρηστες μεταβλητές ελέγχου, τις *flag_hyster* και *flag_canny* για τον έλεγχο της hysteresis και της canny αντίστοιχα. Ο έλεγχος αυτός πραγματοποιείται στον MB0 ώστε να ενημερώνεται ο MB1 για να εκκινήσει των μετρητή. Η συνάρτηση *XTmrCtr_SetResetValue()* μηδενίζει τον μετρητή , η συνάρτηση *XTmrCtr_Reset()* τον επανεκκινεί και τέλος με τη χρήση της *XTmrCtr_Start()* ξεκινά η χρονομέτρηση. Με τη χρήση της *XTmrCtr_Stop()* σταματά η χρονομέτρηση και με την *XTmrCtr_GetValue()* παίρνουμε την τιμή(τον χρόνο) της μέτρησης. Παρακάτω ακολουθεί ο σχετικός κώδικας των δύο επεξεργαστών.

Κώδικας ελέγχου των δύο συναρτήσεων(MB0)

```
*flag_canny=1;
            canny (s,im, magim, oriim);
*flag_canny=2;
   for(q=0;q<800000;q++);
            repeat2:
*flag_hyster=1;
            hysteresis (high, low, im, magim, oriim);
 *flag_hyster=2;
```

Κώδικας μέτρησης του χρόνου εκτέλεσης των δύο συναρτήσεων(MB1)

```
while(n!=3){
 if((*flag_canny==1) && n==1){
              n=2 ;
              XTmrCtr_SetResetValue(&XPS_Timer,0,0x00000000) ;
              XTmrCtr_Reset(&XPS_Timer,0) ;
              XTmrCtr_Start(&XPS_Timer,0) ;
             }
            if((*flag_canny==2) && n==2){
            n=3;
             XTmrCtr_Stop(&XPS_Timer,0) ;
             if(totalt1==0){
              *canny_time = XTmrCtr_GetValue(&XPS_Timer,0) ;
                      xil_printf("The execution time of canny algorithm was: %d ms \r\n", *canny_time);
                      totalt1++ ;
                      }
             }
            }
            while(p!=3){
```

```
20  if((*flag_hyster==1) && p==1){
21                p=2;
22                XTmrCtr_SetResetValue(&XPS_Timer,0,0x00000000) ;
23                XTmrCtr_Reset(&XPS_Timer,0) ;
24                XTmrCtr_Start(&XPS_Timer,0) ;
25               }
26              if((*flag_hyster==2) && p==2){
27               p=3;
28                XTmrCtr_Stop(&XPS_Timer,0) ;
29              if(totalt2==0){
30                 *hyster_time = XTmrCtr_GetValue(&XPS_Timer,0) ;
31                          xil_printf("The execution time of hysteresis algorithm was: %d ms\r\n",
32  *hyster_time);
33                          totalt2++ ;
34                         }
35              }
36  }
37
```

Με βάση τους χρόνους που μετρήσαμε παραπάνω διενεργείτε η διαδικασία του monitoring στη συνέχεια και στηρίζονται τα πειράματα που θα αναλύσουμε στην επόμενη ενότητα.

7.2 Πειράματα - Μετρήσεις

Είναι σημαντικό να αξιολογήσουμε το σύστημα που δημιουργήσαμε καθώς και τον τρόπο που μπορεί να καθοριστεί η λειτουργία ενός αλγορίθμου σε σχέση με τα πιθανά faults που μπορεί να εμφανιστούν είτε με ευθύνη του υλικού, είτε με ευθύνη του λογισμικού μας. Επίσης με τα πειράματα που κάναμε και τα αποτελέσματα που παρουσιάζουμε πιο κάτω, είναι δυνατόν να εξάγουμε συμπεράσματα τόσο για τις αδυναμίες - άρα και για μελλοντική δουλειά που απορρέει από αυτές - όσο και για τα πλεονεκτήματα του συστήματος μας.

Τα πειράματα και οι μετρήσεις που ακολουθούν, βασιζόμενα στους Μέσους χρόνους εκτέλεσης των παραπάνω stages, έχουν ως κύριο στόχο τον υπολογισμό του κόστους ενός λάθους ή πολλαπλών λαθών που στοιχίζουν στο σύστημα, σε σχέση με το πότε αυτά συμβαίνουν. Το πότε συμβαίνουν έχει να κάνει, με το αν τα λάθη σημειώνονται κατανεμημένα ομοιόμορφα στον χρόνο ή όλα μαζί.

Επιπλέον τα πειράματα μας - όπως σημειώσαμε στην προηγούμενη παράγραφο - επικεντρώνονται στην παρουσία λαθών σε σχέση με τον χρόνο, επομένως έχουμε ως συνέπεια, το πόσο αποδοτική ή μη αποδοτική μπορεί να είναι η διαδικασία του ελέγχου που πραγματοποιούμε, κάτω από απαιτητικές(συσσωρευμένα λάθη χρονικά) ή μη απαιτητικές(μη συσσωρευμένα λάθη) συνθήκες. Από την άλλη πλευρά μελετάμε και την εκτέλεση ενός δυναμικού αλγορίθμου στις αντίστοιχες συνθήκες. Κάτω από το συγκεκριμένο σκεπτικό εξετάζουμε ακόμη και την επίδραση στον αλγόριθμο που έχουν τα διαφορετικά μεγέθη δεδομένων(μέγεθος εικόνας στη συγκεκριμένη περίπτωση) τα οποία είναι πιθανόν να κληθεί να εξυπηρετήσει.

Βεβαίως υπάρχουν και άλλες περιπτώσεις πειραματικών μεθόδων που καταγράψαμε και θα βοηθούσαν στη μελέτη και εξαγωγή αντίστοιχων πορισμάτων, αλλά δεν επικεντρωθήκαμε στην παρούσα εργασία. Μία τέτοια περίπτωση θα ήταν η εύρεση των παραμέτρων Mean Time Between Failures(MTBF) και Mean Time To Repair(MTTR) ακόμη και του Availability, παράμετρος που προκύπτει από την εύρεση των προηγούμενων. Σε αυτήν τη μελέτη σκοπός μας θα ήταν ο υπολογισμός των χρόνων διάγνωσης και διόρθωσης των λαθών σε σχέση με τη διάρκεια της εφαρμογής και ακόμη ο υπολογισμός του ελεύθερου διαστήματος που επανεκτελείται η εφαρμογή έπειτα από τη διάγνωση και τη διόρθωση. Επομένως θα μελετούσαμε την αξιοπιστία του συστήματος μας συνολικά(reliability). Επίσης

μια περίπτωση που θα μπορούσαμε να μελετήσουμε θα ήταν το είδος των λαθών που προκαλούνται κ πως αυτά επηρεάζουν το σύστημα μας καθώς και την λειτουργία του αλγορίθμου. Υπάρχουν λάθη που γεννά το hardware τα οποία είναι είτε μόνιμα(permanent faults), είτε παροδικά(transient faults), είτε διακοπτόμενα(intermitted faults). Θα μπορούσαμε λοιπόν να ελέγξουμε το είδος των λαθών και πως αυτά μπορούμε να τα αντιμετωπίσουμε και φυσικά να μελετήσουμε πως επηρεάζεται σε κάθε περίπτωση το σύστημα μας. Σίγουρα θα μπορούσαμε να μελετήσουμε και πολλές ακόμη περιπτώσεις ώστε να επεξεργαστούμε τόσο τις δυνατότητες του συστήματος και της μεθοδολογίας μας όσο και για να βγάλουμε γενικότερα συμπεράσματα για την ανίχνευση λαθών και τις τεχνικές αντιμετώπισης των λαθών αυτών.

7.2.1 Αποτελέσματα

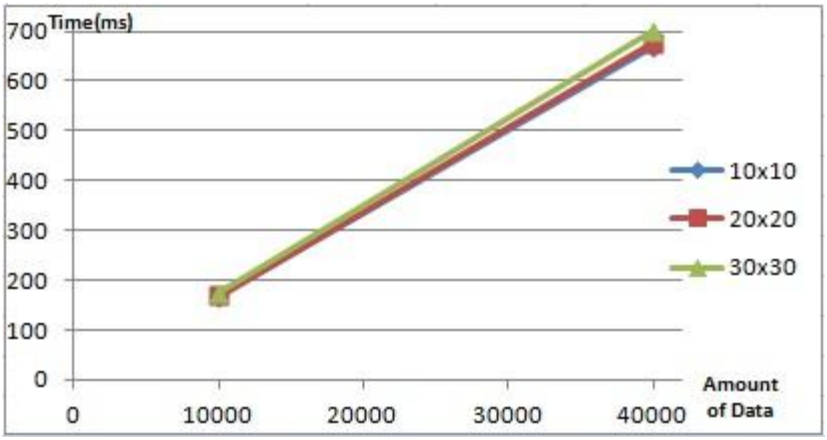
Σε αυτήν την ενότητα θα εξηγήσουμε τα αποτελέσματα που προκύπτουν από τα πειράματα που διεξαγάγαμε και τις μετρήσεις που πήραμε.

7.2.1.1 Μελέτη σε σχέση με το μέγεθος δεδομένων(Amount of Data)

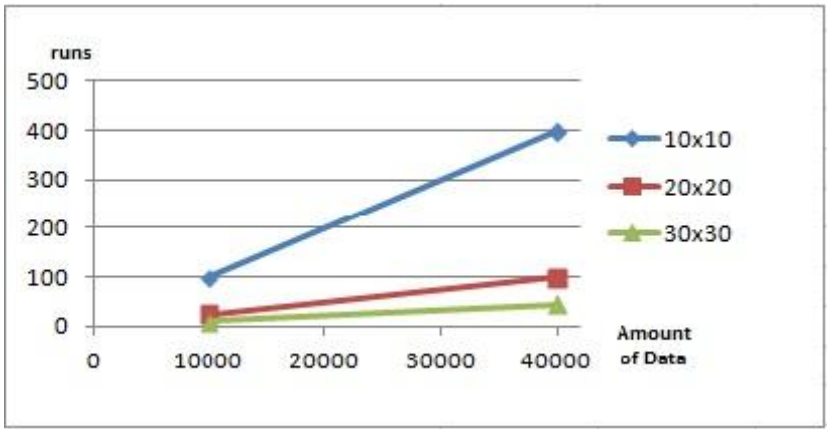
Οι πρώτες μετρήσεις έχουν να κάνουν με τα δεδομένα που είναι επιφορτισμένο το σύστημα μας και πως σε σχέση με αυτά, εκτελείται ο αλγόριθμος χρονικά, αλλά και ανάλογα με τα δεδομένα πόσες φορές χρειάζεται να επαναληφθεί ο αλγόριθμος.

Έχουμε υπολογίσει όπως εξηγήσαμε προηγουμένως τον Μέσο Χρόνο Εκτέλεσης(T_Avg) των stages και συνολικά του αλγορίθμου. Επίσης εξηγήσαμε σε προηγούμενη ενότητα πως διαμερίζουμε την εικόνα μας σε μικρότερες και αυτές τις εικόνες δέχεται ως είσοδο ο αλγόριθμος. Παρακάτω παρουσιάζομαι δύο γραφήματα χρησιμοποιώντας τα στοιχεία αυτά. Στο **Γράφημα 1** παρατηρούμε την διάρκεια εκτέλεσης του αλγορίθμου σε σχέση με το μέγεθος των δεδομένων που καλείται να εξυπηρετήσει το σύστημα. Έχουμε τρεις περιπτώσεις διαμέρισης της αρχικής εικόνας: Εικόνες 10x10, εικόνες 20x20 και εικόνες 30x30. Για τις αρχικές μας εικόνες ελέγχουμε δύο περιπτώσεις: Εικόνα 100x100(10000 μέγεθος δεδομένων) και εικόνα 200x200(40000 μέγεθος δεδομένων). Γίνεται εύκολα αντιληπτό πως για μεγαλύτερο μέγεθος δεδομένων απαιτείτε περισσότερος χρόνος για την εκτέλεση του αλγορίθμου. Μικρή είναι η διαφοροποίηση στους χρόνους μεταξύ των διαφορετικών εικόνων(10x10, 20x20 και 30x30).

Στο **Γράφημα 2** μελετάμε την επίδραση του μεγέθους των δεδομένων στην εκτέλεση του αλγορίθμου. Είναι προφανές από το γράφημα πως όσο μεγαλύτερη η εικόνα που δέχεται ως είσοδο ο αλγόριθμος τόσο περισσότερες εκτελέσεις απαιτούνται για την επεξεργασία ολόκληρης της εικόνας. Επομένως ένα βασικό συμπέρασμα που προκύπτει από το πρώτο δείγμα μετρήσεων είναι πως μπορεί το μέγεθος των διαμερισμένων εικόνων να μην επηρεάζει ιδιαίτερα την απόδοση του συστήματος μας, άρα και τον χρόνο εκτέλεσης του αλγορίθμου, αλλά όσο μικρότερη είναι η διαμερισμένη εικόνα τόσες περισσότερες οι απαιτούμενες επανεκτελέσεις του αλγορίθμου. Αυτό μας οδηγεί να καταλήξουμε στην απόρριψη πολύ μικρών διαμερισμών καθότι, οι πολλαπλές επανεκτελέσεις εκτός από το ότι αυξάνουν την καθυστέρηση(delay), αυξάνουν και την πιθανότητα εμφάνισης σφάλματος(fault) καθώς απαιτούνται πολλαπλάσιοι κύκλοι ρολογιού.



Γράφημα 7.1: Χρονική εξέλιξη του αλγορίθμου σε σχέση με το μέγεθος των δεδομένων



Γράφημα 7.2: Εκτελέσεις του αλγορίθμου σε σχέση με το μέγεθος των δεδομένων

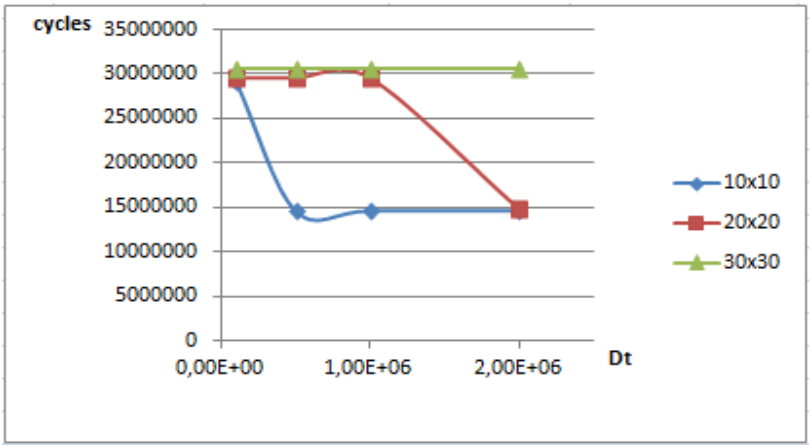
7.2.1.2 Μελέτη σύμφωνα με την συχνότητα εμφάνισης των λαθών

Στο δεύτερο δείγμα μετρήσεων μελετάμε την απόδοση του συστήματος μας σχετικά με την εμφάνιση λαθών στον χρόνο. Προσπαθούμε να δούμε πως μπορεί να επηρεάσει την εκτέλεση των stages και συνολικά του αλγορίθμου η εμφάνιση λαθών σε διαφορετικές συχνότητες.

Εξετάζουμε τέσσερα διαφορετικά χρονικά διαστήματα εμφάνισης λαθών για τις τρεις διαμερισμένες εικόνες(10x10, 20x20, 30x30). Έχουμε αρχικά υπολογίσει τους κύκλους που απαιτούνται για μία εκτέλεση του αλγορίθμου και για τις τρεις εικόνες. Για την **10x10 εικόνα χρειάζονται 145880 κύκλοι,** για την **20x20 εικόνα 590500 κύκλοι** και για την **30x30 εικόνα 1378566 κύκλοι.** Επίσης γνωρίζουμε σε περίπτωση εμφάνισης λάθους σε κάποιο από τα stages πως η διαδικασία του monitoring επαναλαμβάνεται έως και τρεις φορές βάση της μεθοδολογίας μας. Ακόμη, βάση της μεθοδολογίας μας, ανεξάρτητα σε ποιο σημείο του αλγορίθμου διαγνωστεί σφάλμα, ολοκληρώνεται η εφαρμογή και έπειτα επανεκκινεί.

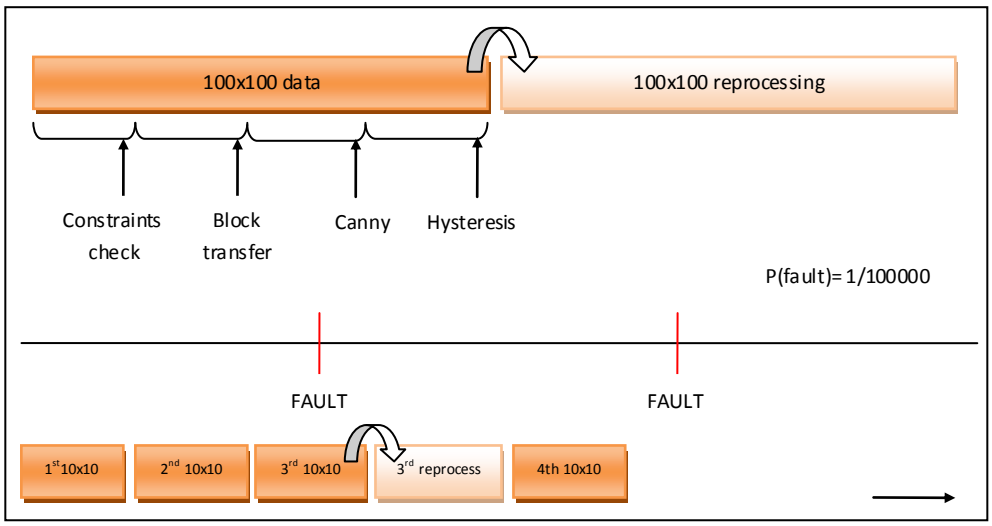
Έχοντας υπόψη τα παραπάνω προκύπτουν τα αποτελέσματα που βλέπουμε στο **Γράφημα 3.** Παρατηρούμε πως όσο μεγαλύτερο το διάστημα μεταξύ των λαθών που εμφανίζονται, τόσο πιθανότερο να έχουμε λιγότερες επανεκτελέσεις για τον αλγόριθμο. Βέβαια εξαρτάται και από το μέγεθος των δεδομένων σε κάθε περίπτωση. Επί παραδείγματι, παρακολουθούμε στο σημείο όπου **Dt=500000,** για workload=100, με single run cycles=145880 και για workload=25 με single run cycles=590500, πως έχουμε μείωση στις εκτελέσεις της 10x10 εικόνας, ενώ μένουν σταθερές στην εικόνα 20x20. Το συμπέρασμα που προκύπτει είναι πως όσο μεγαλύτερο το workload και όσο μικρότερο Dt τόσους

περισσότερους κύκλους εκτέλεσης και επανεκτέλεσης έχουμε. Είναι χαρακτηριστικό στο γράφημα η απεικόνιση της 30x30 εικόνας. Το υψηλό workload απαιτεί τόσους κύκλους εκτέλεσης που ακόμη και στην περίπτωση όπου έχουμε εμφάνιση λάθους μόλις 1/2*10^6, και πάλι βρίσκεται εντός του διαστήματος του αρχικού χρόνου εκτέλεσης επομένως απαιτείται επανάληψη της εκτέλεσης.



Γράφημα 7.3: Εξέλιξη του monitoring σε σχέση με την συχνότητα εμφάνισης λαθών στο χρόνο

Ένα ακόμη ζητούμενο που προκύπτει από τη συγκεκριμένη μελέτη είναι τι τελικά συμφέρει, να επεξεργαζόμαστε τα δεδομένα διαμερισμένα σε μικρότερα κομμάτια ή ως ενιαία πληροφορία, αλλάζει ο χρόνος εκτέλεσης στις δύο περιπτώσεις? Στην περίπτωση που έχουμε διαμερίσει την εικόνα μας σε μικρότερες εικόνες, τότε η εμφάνιση λάθους θα προκαλέσει την επανάληψη του monitoring μόνο για την διαμερισμένη εικόνα και όχι για το σύνολο των δεδομένων. Αντιθέτως σε περίπτωση που το monitoring πραγματοποιείται στο σύνολο των δεδομένων πιθανή εμφάνιση λάθους θα προκαλέσει την ολική επανεξέταση των δεδομένων**(Σχήμα 7.1).**



Σχήμα 7.1: Fault detection για ολόκληρη την εικόνα και για διαμερισμένη εικόνα

Είναι σαφές πως η μεθοδολογία της διαμέρισης έχει το πλεονέκτημα ότι σε περίπτωση λάθους, οι υπολογισμοί επαναλαμβάνονται μόνο για το κομμάτι της εικόνας που διαγνώστηκε, αποφεύγοντας έτσι την επανάληψη των υπολογισμών για όλη την εικόνα. Γι αυτό το λόγο επιλέχθηκε και από εμάς.

8. Συμπεράσματα – Μελλοντικές Επεκτάσεις

Από την μεθοδολογία που αναπτύξαμε προκύπτουν κάποια βασικά συμπεράσματα, τόσο για τη συγκεκριμένη μεθοδολογία όσο και γενικότερα για την επιστημονική προσπάθεια γύρω από το ζήτημα της αξιοπιστίας των ενσωματωμένων συστημάτων.

Αρχικά είναι βέβαιο πως με την ανάπτυξη σύγχρονων μεθόδων είναι ικανή η όλο και καλύτερη αντιμετώπιση των εμφανιζόμενων σφαλμάτων, από ένα ενσωματωμένο σύστημα. Βέβαια είναι εξαιρετικά φιλόδοξο να πιστέψουμε πως το ζήτημα της αξιοπιστίας και της αντιμετώπισης σφαλμάτων θα επιλυθεί εξολοκλήρου, καθώς οι αυξανόμενες απαιτήσεις σε ταχύτατα και χωρητικότητα δημιουργούν τις προϋποθέσεις για εμφάνιση σφαλμάτων(soft errors).

Όσο αφορά τη δική μας εργασία, προκύπτουν **πέντε κύρια συμπεράσματα:**

1. Η αύξηση των δεδομένων που εισάγονται για επεξεργασία, επηρεάζει τη διάρκεια εκτέλεσης του αλγορίθμου, αλλά δεν υπάρχουν μεγάλες διαφοροποιήσεις ανάλογα με τις διαφοροποιήσεις στο μέγεθος των εισαγόμενων εικόνων.

2. Από την άλλη η αύξηση των δεδομένων, δηλαδή η επιπλέον επιφόρτιση του συστήματος μας με δεδομένα, επιβάλει περισσότερους κύκλους για την επεξεργασία των δεδομένων μας, άρα και επιπλέον καθυστέρηση κατά τη διαδικασία της αντιγραφής δεδομένων από το CMB.

3. Αυτό μας οδηγεί να καταλήξουμε στην απόρριψη πολύ μικρών διαμερισμών καθότι, οι πολλαπλές επανεκτελέσεις εκτός από το ότι αυξάνουν την καθυστέρηση(delay), αυξάνουν και την πιθανότητα εμφάνισης σφάλματος(fault) καθώς απαιτούνται πολλαπλάσιοι κύκλοι ρολογιού.

4. Το σύστημα μας μπορεί να ανταπεξέλθει στην εμφάνιση πολλαπλών σφαλμάτων, στην ίδια εφαρμογή και φυσικά μπορεί να αναγνωρίσει πολλαπλά σφάλματα κατά τη διάρκεια του ελέγχου(monitoring).

5. Η διαδικασία μας εν τέλει παρέχει αυξημένη αξιοπιστία, κατά τον έλεγχο μίας εφαρμογής με ορισμένα στάδια από τον χρήστη και μετρήσιμο τον χρόνο εκτέλεσης τους.

8.1 Μελλοντικές Επεκτάσεις

Πλήθος επεκτάσεων μπορούν να ελεγχτούν με βάση την μεθοδολογία που αναπτύξαμε παραπάνω:

- Αυτόματος ορισμός stages από τον επεξεργαστή-παρακολουθητή με την εισαγωγή threads
- Αυτόματη χρονομέτρηση της εκτέλεσης των stages και του αλγορίθμου με την εισαγωγή threads
- Παράλληλη επιδιόρθωση κώδικα από το CMB για εσφαλμένα stages.
- Διασύνδεση δύο ίδιων συστημάτων, που θα εκτελούν παράλληλα την πιο πάνω διαδικασία και θα έχουν τη δυνατότητα της σύγκρισης και ενοποίησης των αποτελεσμάτων που θα προκύπτουν.

Βιβλιογραφία

[1] W. Savage, J. Chilton, and R. Camposano. Ip reuse in the system on a chipera. In Proceeding of ISSS, pages 2{7, 2000.

[2] H. min Kyung, G. ho Park, J. W. Kwak, W. Jeong, T.-J. Kim, and S.-B. Park. Performance monitor unit design for an axi-based multi-core soc platform. In SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, pages 1565{1572, New York, NY, USA, 2007. ACM.

[3] B. Sprunt. Pentium 4 performance-monitoring features. IEEE Micro, 22:72{82, 2002.

[4] A. A. Bayazit and S. Malik. *Complementary use of runtime validation and model checking*. In ICCAD '05: Proceedings of the 2005 IEEE/ACM Inter- national conference on Computer-aided design, pages 1052{1059, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Calin Ciordas. *Monitoring-Aware Network-on-Chip Design*, 2009

[6] Mohammed El Shobaki. *On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems, 2002*

[7] Michael Glaß, Martin Lukasiewycz, Christian Haubelt, and Jurgen Teich. *Lifetime Reliability Optimization for Embedded Systems: A System-Level Approach*, 2010

[8] R. E. Bryant, *Graph-based algorithms for boolean function manipulation*, Trans. on Comp., vol. 35, no., 1986.

[9] Christiana Bolchini, and Fabio Salice. *The Design of Self-Checking Systems*, 1999.

[10] Cristiana Bolchini, Antonio Miele and Marco D. Santambrogio. *TMR and Partial Dynamic Recon_guration to mitigate SEU faults in FPGAs*, 2008.

[11] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, A. Marmo, S. Pastore, G.R. Sechi. *A Tool for Injecting SEU-like Faults into the Configuration Control Mechanism of Xilinx Virtex FPGAs*, 2003.

[12] Francisco Afonso, Carlos Silva, Adriano Tavares. *Application-Level Fault Tolerance in Real-Time Embedded Systems*, 2007.

[13] Walter W. Schilling, Jr., Dr. Mansoor Alam, *Embedded Systems Software Reliability*, 2006.

[14] Eduardo Valido-Cabrera, *Software Reliability Methods*, 2006.

[15] Mohsin Amin, Abbas Ramazani, FabriceMonteiro, Camille Diou, and Abbas Dandache, *A Self-Checking Hardware Journal for a Fault-Tolerant Processor Architecture*, 2011.

[16] Bedir Tekinerdogan, Hasan Sözer, Mehmet Aksit, *Software Architecture Reliability Analysis using Failure Scenarios*, 2006.

[17] J. B. Dugan. Handbook of Software Reliability 12 Engineering, *Software System Analysis Using Fault Trees, pages 615-659*, 1996.

[18] Foad Dabiri, Naivd Amini, Mahsan Rofouei and Majid Sarrafzadeh, *Reliability-Aware Optimization for DVS-Enabled Real-Time Embedded Systems*, 2007.

[19] Baoxian Zhao, Hakan Aydin, Dakai Zhu, *On Maximizing Reliability of Real-Time Embedded Applications Under Hard Energy Constraint*, 2010.

Παράρτημα

Α. Παρουσίαση Πτυχιακής

Επαύξηση Αξιοπιστίας σε Πολυπύρηνα Ενσωματωμένα Συστήματα

Χριστοφοράκης Ιωάννης ΑΜ 2463

Επιβλέπων Καθηγητής: Κορνάρος Γεώργιος

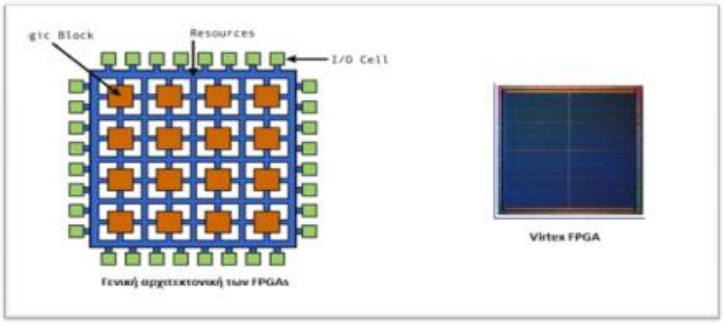
Τμήμα Εφαρμοσμένης Πληροφορικής και Πολυμέσων

Εισαγωγή

- Τι είναι Ενσωματωμένα Συστήματα
- Ένα ενσωματωμένο σύστημα είναι ένα υπολογιστικό σύστημα ειδικού σκοπού, σχεδιασμένο έτσι ώστε να εκτελεί μια ή και περισσότερες συναρτήσεις, συνήθως σε σταθερές πραγματικού χρόνου.
- Αυτοκίνητο, PS, ATM, GPS, κλιματιστικά κ.α.

Εισαγωγή

- Προγραμματιζόμενα στοιχεία υλικού
- Προκατασκευασμένα ψηφιακά κυκλώματα που μπορούν να προγραμματιστούν από το σχεδιαστή/χρήστη
  - PLDs
  - FPGAs



Αξιοπιστία στα Ενσωματωμένα - Reliability

- **Παράγοντες που επηρεάζουν την Αξιοπιστία**
- Οι προκλήσεις για όσο γίνεται μικρότερο μέγεθος τρανζίστορ έχουν οδηγήσει σε σημαντική μεταβολή στις παραμέτρους του τρανζίστορ, όπως το μήκος του καναλιού
- Περιοδικά σφάλματα(**transient faults**). Πρόκειται για σφάλματα που προκαλούνται από προσωρινές συνθήκες για το **chip**
- Φαινόμενο της επίδρασης γήρανσης(**aging effect**).
- Λάθη στο λογισμικό

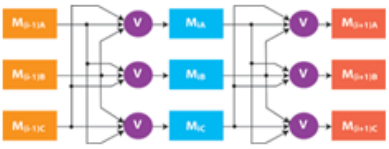
Αξιοπιστία στα Ενσωματωμένα - Reliability

- **Fault Tolerance**
  - Είναι η ιδιότητα που επιτρέπει ένα σύστημα να συνεχίσει σωστά την λειτουργία του σε περίπτωση αποτυχίας (ή περισσότερων αποτυχιών παράλληλα) σε μερικές από τις λειτουργίες του.
- **Η αποκατάσταση λαθών**
  - **κίνηση-προς τα πίσω** επαναφέρει την κατάσταση του συστήματος σε κάποια προηγούμενη-σωστή κατάσταση, χρησιμοποιώντας σημεία ελέγχου (checkpoints)
  - **κίνηση-προς τα εμπρός** (roll-forward)

Αξιοπιστία στα Ενσωματωμένα - Reliability

- Monitoring
  - Τεχνικές Λογισμικού(ODC)
  - Κώδικας ως «όργανο μέτρησης»

  - Τεχνικές Υλικού(TMR)
  - Hardware Monitoring



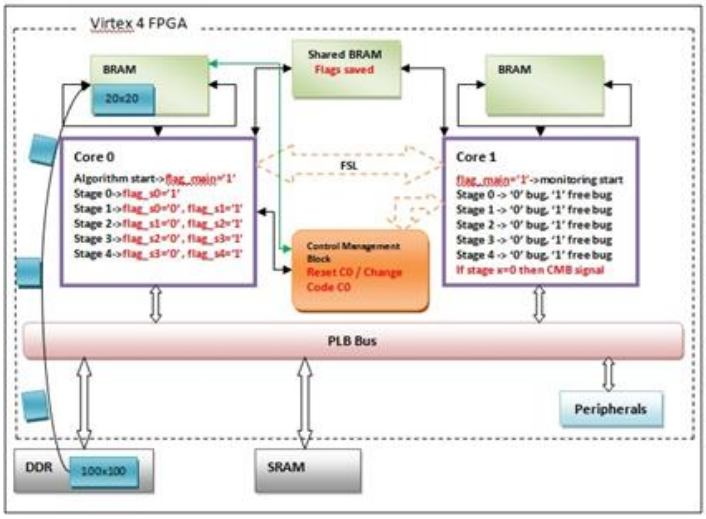
  - Hybrid Monitoring
  - Συνδυασμός τον παραπάνω

Υλοποιούμενη Μεθοδολογία

- Συνδυασμένη Μεθοδολογία sw/hw
  - Μεγαλύτερη ευελιξία
  - Μικρότερο κόστος
- Ανάπτυξη στο αναπτυξιακό περιβάλλον της Xilinx(EDK 12.1)
- Εισαγωγή Control Management Block για αντικατάσταση /επιδιόρθωση /επανεκκίνηση δεδομένων
- Ανάπτυξη ειδικού λογισμικού για real-time monitoring/ fault detection

Υλοποιούμενη Μεθοδολογία

Γενική Αρχιτεκτονική και λειτουργίες

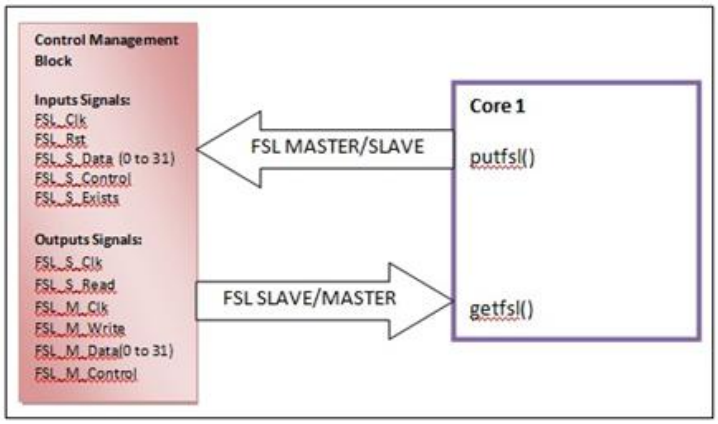


Υλοποιούμενη Μεθοδολογία

- Αρχιτεκτονική
- Βασική αρχιτεκτονική συστήματος της Xilinx
  - 2 cores
    - 1 monitoring
    - 1 canny algorithm
  - 5 μνήμες
    - 2 LBRAM
    - 1 κοινόχρηστη
    - 2 εξωτερικές
- Control Management Block
  - Σύνδεση μέσω FSL με core 1
  - Σύνδεση μέσω nets με core 0

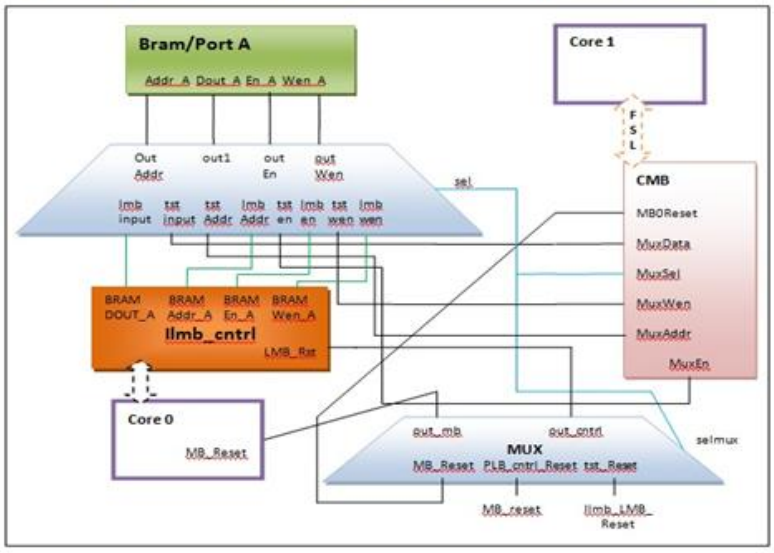
Υλοποιούμενη Μεθοδολογία

- Το CMB δέχεται από το FSL 3 λέξεις των 32 bit.
  - **Πρώτη λέξη** είναι η εντολή(command),
  - **Δεύτερη λέξη** δίνει την επόμενη διεύθυνση γραφής
  - **Τρίτη λέξη** δίνει τα δεδομένα που έρχονται από το επεξεργαστή.



Υλοποιούμενη Μεθοδολογία

- Συνδεσμολογία CMB με το core 0 και την μνήμη του.



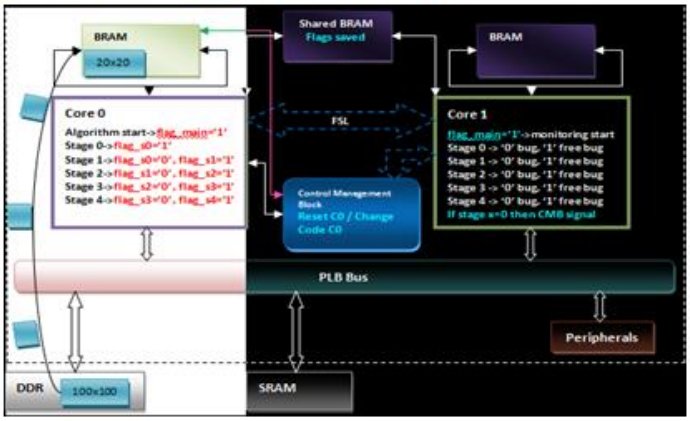
Υλοποιούμενη Μεθοδολογία

- Περιγραφή Λογισμικού
- Canny Αλγόριθμος
  - Stages αλγορίθμου
- Μεθοδολογία monitoring – fault detection
  - Κοινόχρηστες σημαίες
  - Επικοινωνία core 1 με CMB

Υλοποιούμενη Μεθοδολογία

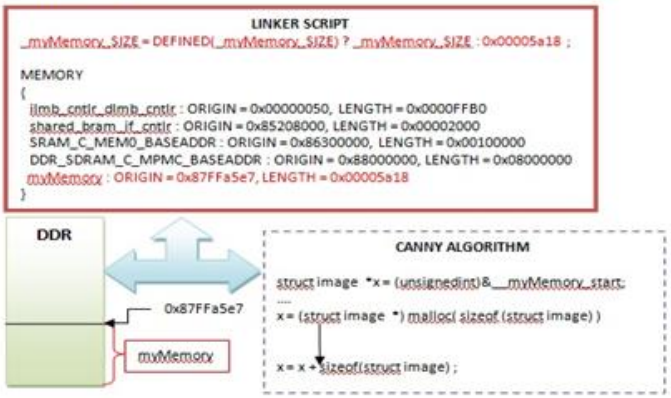
- Stage 0
  - Μεταφορά Δεδομένων / Διαμέριση

Δεδομένα από την DDR στην BRAM



Υλοποιούμενη Μεθοδολογία

- Δημιουργία Ειδικής Μνήμης/Section
  - Καταλάβαμε χώρο της DDR για να εξυπηρετήσουμε τις ανάγκες του αλγορίθμου με τέτοιο τρόπο ώστε να μπορεί να εκτελεστεί πολλαπλά χωρίς περιορισμούς



Υλοποιούμενη Μεθοδολογία

- Stage 1-2
  - Στην επόμενη φάση ελέγχουμε εάν ο χώρος που ζητάμε για τη δημιουργία δύο εικόνων στην BRAM επαρκή.
  - Ο χώρος μνήμης που απαιτείτε για τις εικόνες δεσμεύεται επίσης δυναμικά, καλώντας την *newimage*.
  - Αυτές οι δύο εικόνες είναι οι βασικές εικόνες πάνω στις οποίες γίνεται η επεξεργασία για την εύρεση ακμών και το μέγεθος τους εξαρτάται από τις απαιτήσεις του χρήστη, είναι οι *magim*, και *oriim*

Υλοποιούμενη Μεθοδολογία

- Stage 3-4
  - Εκτέλεση των βασικών συναρτήσεων Canny και Hysteresis
- Canny
  - φίλτρο gauss, *gau[i] = meanGauss ((float)i, s);*
  - Συνέλιξη *seperable_convolution (im, gau, width, smx, smy);*,
  - τέλος δημιουργεί μια νόρμα με βάση τις διαστάσεις τις εικόνας, *z = norm (dx[i][j], dy[i][j]);*
- Hysteresis
  - μηδενίζει τα pixels της εικόνας *im->data[i][j] = 0;*,
  - Καλεί την συνάρτηση trace, *trace (i, j, low, im, mag, oriim);*, η οποία σύμφωνα με το αυτό-οριζόμενο threshold θα βρει τα pixels εκείνα που συντελούν τις ακμές τις εικόνας και θα τα τους δώσει την τιμή 255 *if (im->data[i][j] == 0) im->data[i][j] = 255*

Υλοποιούμενη Μεθοδολογία

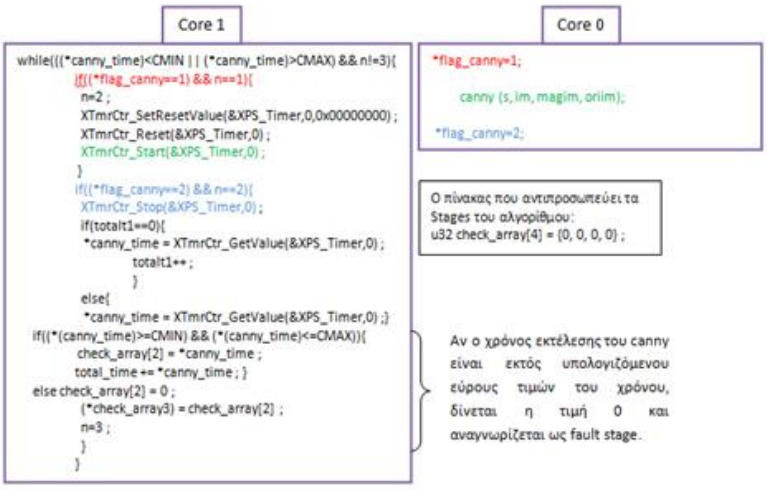
- Μεθοδολογία monitoring – fault detection

Κοινόχρηστες Σημαίες(flags):

- **flag_magnitude:** Έλεγχος επάρκειας τοπικής μνήμης, για την εικόνα maimag
- **flag_orientation:** Έλεγχος επάρκειας τοπικής μνήμης, για την εικόνα oriimag
- **flag_main:** Ελέγχει αν έχει ξεκινήσει η λειτουργία του αλγορίθμου.
- **flag_canny:** Έλεγχος έναρξης – λήξης της συνάρτησης **canny**.
- **flag_hyster:** Έλεγχος έναρξης – λήξης της συνάρτησης **hysteresis**.

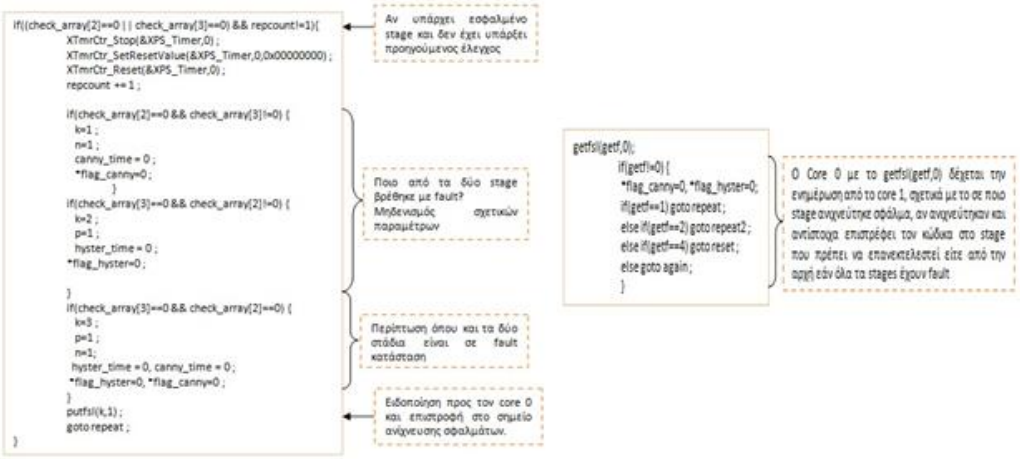
Υλοποιούμενη Μεθοδολογία

- Ανίχνευση Σφαλμάτων στο stage 3 – Σχετικός κώδικας στους core1 και core 0



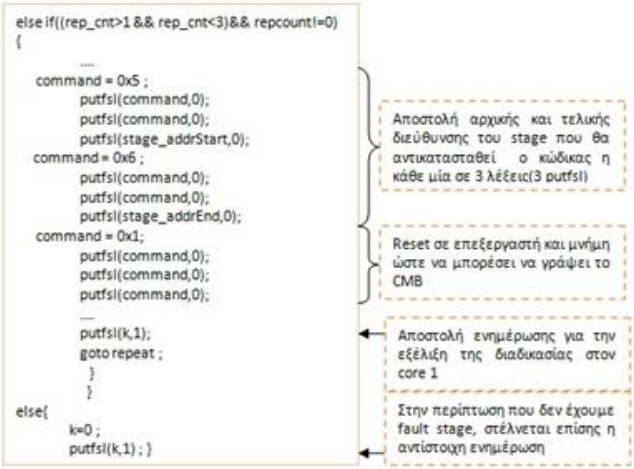
Υλοποιούμενη Μεθοδολογία

- Έλεγχος για το ποιο stage είναι εσφαλμένο



Υλοποιούμενη Μεθοδολογία

- Αποστολή σημάτων από τον core 1 στο CMB ανάλογα με τα αποτελέσματα του monitoring



Πειράματα - Αποτελέσματα

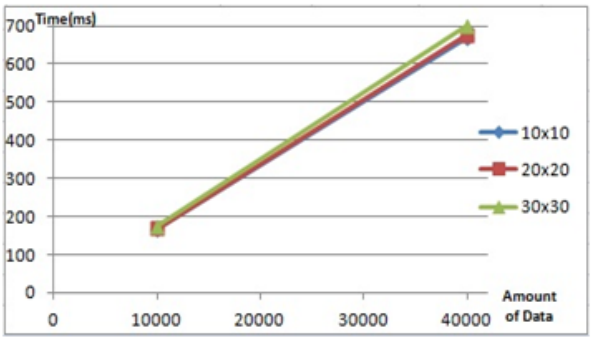
- Αξιολόγηση Canny Αλγορίθμου
  - Χρόνος εκτέλεσης των stages ανάλογα με το μέγεθος των δεδομένων

	T_Avg_10x10 (msec)	T_Avg_20x20 (msec)	T_Avg_30x30 (msec)
workload	100	400	900
Constraints check(S1)	0	0	0
Block transfer (S2)	0,21315	0,87	2,0142675
Canny (S3)	1,195425	4,83	11,2967663
Hysterisis (S4)	0,050225	0,205	0,47462625

  - Ειδικότερα παρατηρούμε **απόκλιση κατά 1,05%** στις μετρήσεις της 30x30 εικόνας.
  - Για τον χρονομέτρηση κάθε stage ενεργοποιούμε κατάλληλα τον hw timer του συστήματος.

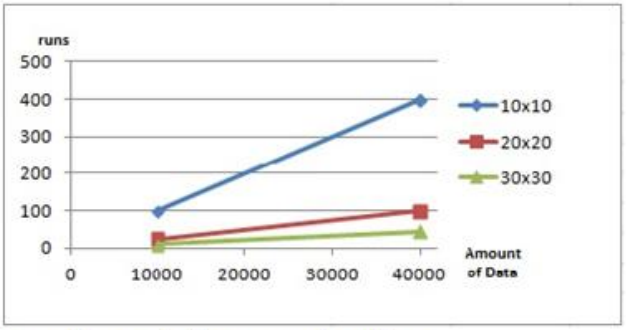
Πειράματα - Αποτελέσματα

- Μετρήσεις
  - Μελέτη σε σχέση με το μέγεθος δεδομένων(Amount of Data)
  - Τρεις εικόνες 10x10, 20x20, 30x30



Πειράματα - Αποτελέσματα

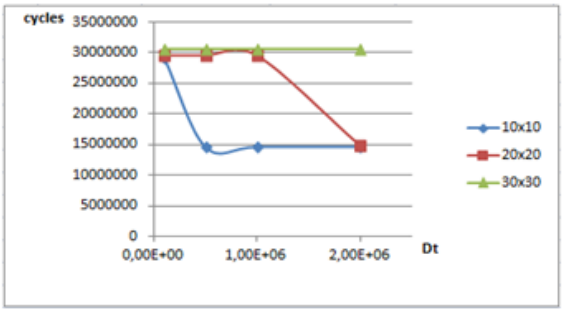
- Εκτελέσεις του αλγορίθμου σε σχέση με το μέγεθος των δεδομένων
  - Για τις ίδιες εικόνες 10x10, 20x20, 30x30



  - Απορρίπτουμε πολύ μικρούς διαμερισμούς καθότι,
  - πολλαπλές επανεκτελέσεις
  - αυξάνετε η καθυστέρηση(delay),
  - αυξάνετε η πιθανότητα εμφάνισης σφάλματος(fault) καθώς απαιτούνται πολλαπλάσιοι κύκλοι ρολογιού.

Πειράματα - Αποτελέσματα

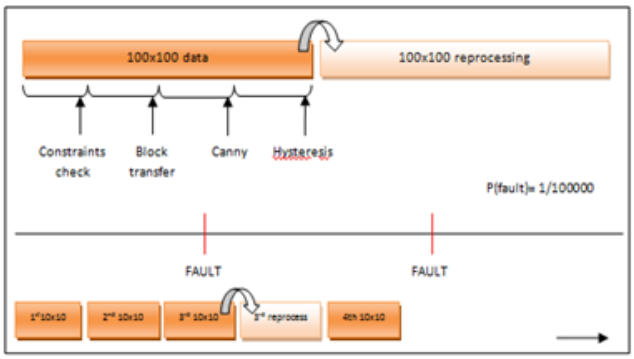
- **Μελέτη σύμφωνα με την συχνότητα εμφάνισης των λαθών**
  - Εξέλιξη του **monitoring** σε σχέση με την συχνότητα εμφάνισης λαθών στο χρόνο



  - Το υψηλό **workload** απαιτεί τόσους κύκλους εκτέλεσης που ακόμη και στην περίπτωση όπου έχουμε εμφάνιση λάθους μόλις 1/2*10^6, πάλι βρίσκεται εντός του διαστήματος του αρχικού χρόνου εκτέλεσης επομένως απαιτείται επανάληψη της εκτέλεσης

Πειράματα - Αποτελέσματα

- **Επιλογή της Διαμέρισης**



  - Η μεθοδολογία της διαμέρισης έχει το πλεονέκτημα ότι σε περίπτωση λάθους, οι υπολογισμοί επαναλαμβάνονται μόνο για το κομμάτι της εικόνας που διαγνώστηκε, αποφεύγοντας έτσι την επανάληψη των υπολογισμών για όλη την εικόνα.

Συμπεράσματα

- Η αύξηση των δεδομένων που εισάγονται για επεξεργασία, επηρεάζει τη διάρκεια εκτέλεσης του αλγορίθμου, αλλά δεν υπάρχουν μεγάλες διαφοροποιήσεις ανάλογα με τις διαφοροποιήσεις στο μέγεθος των εισαγόμενων εικόνων.
- Η αύξηση των δεδομένων επιβάλει περισσότερους κύκλους για την επεξεργασία των δεδομένων μας, άρα και επιπλέον καθυστέρηση κατά τη διαδικασία της αντιγραφής δεδομένων από το **CMB**.
- Απορρίπτουμε πολύ μικρές διαμερίσεις καθότι, οι πολλαπλές επανεκτελέσεις εκτός από το ότι αυξάνουν την καθυστέρηση(**delay**), αυξάνουν και την πιθανότητα εμφάνισης σφάλματος(**fault**).
- Το σύστημα μας μπορεί να ανταπεξέλθει στην εμφάνιση πολλαπλών σφαλμάτων, στην ίδια εφαρμογή και φυσικά μπορεί να αναγνωρίσει πολλαπλά σφάλματα κατά τη διάρκεια του ελέγχου(**monitoring**).
- Η διαδικασία μας εν τέλει παρέχει αυξημένη αξιοπιστία, κατά τον έλεγχο μίας εφαρμογής με ορισμένα στάδια από τον χρήστη και μετρήσιμο τον χρόνο εκτέλεσης τους.

Μελλοντικές Επεκτάσεις

- Αυτόματος ορισμός **stages** από τον επεξεργαστή-παρακολουθητή με την εισαγωγή **threads**
- Αυτόματη χρονομέτρηση της εκτέλεσης των **stages** και του αλγορίθμου με την εισαγωγή **threads**
- Παράλληλη επιδιόρθωση κώδικα από το **CMB** για εσφαλμένα **stages**.
- Διασύνδεση δύο ίδιων συστημάτων, που θα εκτελούν παράλληλα την πιο πάνω διαδικασία και θα έχουν τη δυνατότητα της σύγκρισης και ενοποίησης των αποτελεσμάτων που θα προκύπτουν.

Ευχαριστούμε για την προσοχή σας!