ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Ενσωματωμένα Συστήματα και σήματα διακοπής. Η περίπτωση του ARM

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθανάσιος Γιαννέτσος

Βόλος, Δεκέμβριος 2006

Αφιερώνω την εργασία αυτή
στους γονείς μου

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή του τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων και βασικό επιβλέποντα της πτυχιακής αυτής εργασίας κ. Γ. Σταμούλη που μου έδωσε την ευκαιρία να πραγματοποιήσω αυτή την μελέτη. Η υποστήριξή του, η αμέριστη συμπαράστασή του, αλλά και οι διαρκείς και εύστοχες υποδείξεις του βοήθησαν στην έγκαιρη ολοκλήρωση αυτής της μελέτης.

Επιπρόσθετα, θα ήθελα να ευχαριστήσω τον έτερο επιβλέποντα καθηγητή και επισκέπτη καθηγητή στο τμήμα, κ. Ν. Ευμορφόπουλο. Οι γνώσεις του και η βοήθειά του αποτέλεσαν τα θεμέλια της προσπάθειας για μια όσο το δυνατόν καλύτερη και πιο εμπεριστατωμένη παρουσίαση του θέματος. Δεν θα μπορούσα να ξεχάσω τον προπτυχιακό φοιτητή του τμήματος Μ. Μπασούκο για την βοήθεια που μου προσέφερε.

Τέλος, θα ήθελα να ευχαριστήσω τον αδερφό μου Τόλη για την υποστήριξη που μου επέδειξε. Επίσης ευχαριστώ όλους τους φίλους μου οι οποίοι μου συμπαραστάθηκαν αμέριστα καθ' όλη τη διάρκεια εκπόνησης της διπλωματικής αυτής εργασίας.

Περιεχόμενα

Πρόλογος

Στην δεκαετία του 1960, οι υπολογιστές είχαν την δυνατότητα να αποκτούν, να επεξεργάζονται και να κατέχουν δεδομένα ή ακόμα και να παίρνουν αποφάσεις σε πολύ μεγάλες ταχύτητες. Παρόλα αυτά, υπήρχαν κάποια μειονεκτήματα όσον αφορά τον έλεγχο τέτοιων υπολογιστικών μηχανημάτων. Ήταν ακριβά, περιελάμβαναν πολύπλοκα προγράμματα, ενώ και το προσωπικό των διάφορων εταιρειών ήταν δύσπιστο απέναντί τους (ως προς την εκμάθησή τους). Ωστόσο, αναπτύχθηκε η νέα ιδέα των ηλεκτρονικών συσκευών. Ονομαζόντουσαν "Programmable Controllers" και στην συνέχεια αποτέλεσαν μέρος των ενσωματωμένων συστημάτων (embedded systems). Η ιδέα αυτή στηρίχθηκε στον συνδυασμό της τεχνολογίας που υπήρχε (για τους υπολογιστές) και σε παραδοσιακές ηλεκτρομηχανικές ακολουθίες (electromechanical sequences)[1]. Ο πρώτος Programmable Controller δημιουργήθηκε το έτος 1969.

Το 1978, ο οργανισμός National Engineering Manufacturers, έδωσε προς χρήση το πρότυπο ενός προγραμματίσιμου μικροεπεξεργαστή (programmable microcontroller). Ο ορισμός του ήταν σχεδόν ίδιος με αυτόν ενός υπολογιστικού μηχανήματος. Περιείχε ελεγκτές αριθμητικών πράξεων (numerical controllers) και ελεγκτές ακολουθιών (sequential controllers) για να εκτελεί εντολές βασισμένες σε γεγονότα (η εκτέλεσή τους βασιζόταν στην εμφάνιση μιας σειράς γεγονότων).

Το πρώτο αναγνωρισμένο ενσωματομένο σύστημα ήταν το "Apollo Guidance Computer" (σύστημα πλοήγησης) που δημιουργήθηκε από τον Charles Stark Draper στο

MIT Instrumentation Laboratory. Κάθε πτήση προς το φεγγάρι είχε δυο τέτοια μηχανήματα. Θεωρήθηκε πολύ ριψοκίνδυνο εγχείρημα, μιας και χρησιμοποιούσε την νέα τεχνολογία μονολιθικών ολοκληρωμένων κυκλωμάτων προκειμένου να μειώσει το μέγεθος και το βάρος.

[1] Στην εφαρμοσμένη μηχανική, η ηλεκτρομηχανική συνδυάζει τις επιστήμες του ηλεκτρομαγνητισμού, της ηλεκτρολογίας και της μηχανικής.

Το πρώτο μαζικής παραγωγής ενσωματωμένο σύστημα, ήταν το σύστημα πλοήγησης Autonetics D-17 που χρησιμοποιήθηκε στον πύραυλο Minuteman το 1961. Αποτελούνταν από διακεκριμένα τρανζίστορ και έναν σκληρό δίσκο για κύρια μνήμη.

Στην πορεία αυτής της ανάπτυξης των ενσωματωμένων συστημάτων, φτάνουμε στα μέσα της δεκαετίας του 1980, όπου πολλά εξωτερικά μέρη συστημάτων όπως η μνήμη ή η μονάδα αριθμητικών πράξεων, συγχωνεύτηκαν σε ένα μόνο chip όπως είναι ο επεξεργαστής έχοντας ως αποτέλεσμα την δημιουργία νέων ολοκληρωμένων συστημάτων που ονομαζόντουσαν microcontrollers. Επομένως, μια εκτεταμένη χρήση των ενσωματωμένων συστημάτων ήταν εφικτή.

Ένα από τα σημαντικότερα χαρακτηριστικά τέτοιων συστημάτων, είναι η υποστήριξη της λειτουργίας με την χρήση σημάτων διακοπής (interrupts)[2]. Αυτό σημαίνει ότι οι διάφορες λειτουργίες που έχουμε αναθέσει σε έναν υπολογιστή "πυροδοτούνται" από συγκεκριμένα γεγονότα. Ένα interrupt μπορεί να παραχθεί, για παράδειγμα, από έναν χρονομετρητή (timer) με μια προκαθορισμένη συχνότητα. Τέτοιων ειδών σήματα, επεξεργάζονται από μικρούς και απλούς handlers και πρέπει να έχουν μικρή περίοδο λανθάνουσας κατάστασης (latency time). Συνήθως, τα συστήματα αυτά "τρέχουν" και ένα task σε έναν κύριο βρόγχο, άλλα το task αυτό δεν είναι ευαίσθητο σε απροσδόκητες καθυστερήσεις. Οι λειτουργίες που γίνονται μέσα στους διάφορους handlers θα πρέπει να είναι μικρές και γρήγορες έτσι ώστε ο χρόνος που το σύστημα θα είναι σε μια λανθάνουσα κατάσταση (interrupt) να είναι ελάχιστος.

Παρακάτω, θα αναλύσουμε τις διάφορες μορφές που μπορούν να έχουν τα interrupts καθώς και τις ενέργειες που κάνει ένα σύστημα για να μπορέσει να εξυπηρετήσει ένα τέτοιο σήμα διακοπής.

---

[2] Επειδή ο όρος σήματα διακοπής στην ελληνική γλώσσα δεν είναι πολύ δόκιμος, στην συνέχεια του κειμένου θα χρησιμοποιούμε τον αγγλικό όρο interrupts.

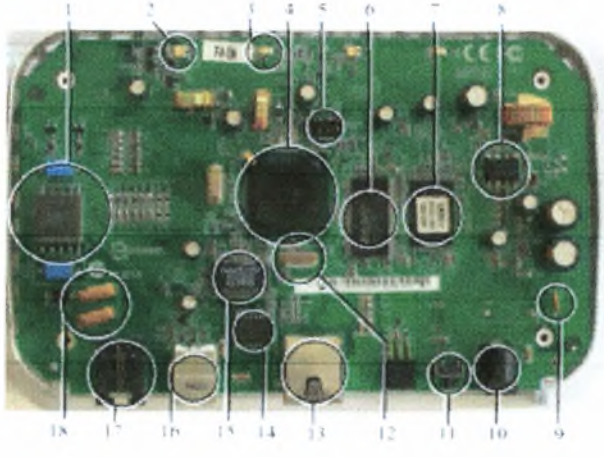
Εισαγωγή

Τι είναι ένα ενσωματωμένο σύστημα; Ουσιαστικά, είναι ένα ειδικού σκοπού υπολογιστικό σύστημα το οποίο ελέγχει διάφορες λειτουργίες (οι οποίες είναι προκαθορισμένες) με συγκεκριμένες απαιτήσεις, σε αντίθεση με τα γενικού σκοπού συστήματα. Κάποια από αυτά, έχουν και μερικούς πραγματικού-χρόνου περιορισμούς τους οποίους πρέπει να ικανοποιήσουν για λόγους ασφάλειας και χρησιμότητας. Αφού τέτοια συστήματα δημιουργούνται για την διεκπεραίωση συγκεκριμένων λειτουργιών, μπορούμε να τα βελτιστοποιήσουμε μειώνοντας το μέγεθός τους και το κόστος τους.

Υπολογιστές τσέπης ή PDAs, μπορούν να θεωρηθούν ενσωματωμένα συστήματα λόγω του σχεδιασμού του φυσικού εξοπλισμού τους (hardware), παρόλο που είναι αρκετά επεκτάσιμα όσον αφορά το λογισμικό τους.

Γενικά, η κατηγορία των ενσωματωμένων επεκτείνεται από μικρές φορητές συσκευές, όπως MP3 players, PSP consoles, iPods, εκτυπωτές, κινητά τηλέφωνα, μέχρι και μεγάλες σταθερές συσκευές, όπως σηματοδότες.

Στην διπλανή φωτογραφία, βλέπουμε έναν δρομολογητή (router), ο οποίος αποτελεί ένα ενσωματωμένο σύστημα. Τα αριθμημένα μέρη περιλαμβάνουν έναν μικροεπεξεργαστή (4), μνήμη RAM (6) και flash μνήμη (7).



Τα ενσωματωμένα συστήματα διαδίδονται ολοένα και περισσότερο. Πολλές συσκευές περιέχουν τέτοια συστήματα για την διεκπεραίωση συγκεκριμένων προκαθορισμένων λειτουργιών. Για παράδειγμα, το 2000, η αγορά των υπολογιστών ήταν περίπου στα 8 δισεκατομμύρια δολάρια. Από αυτά, περίπου τα 7.8 δισεκατομμύρια δαπανήθηκαν για την αγορά ενσωματωμένων συστημάτων.

Περιγραφή και περιορισμοί

Όπως αναφέραμε και παραπάνω, συνήθως τα ενσωματωμένα συστήματα δημιουργούνται για την διεκπεραίωση συγκεκριμένων προκαθορισμένων λειτουργιών που επαναλαμβάνονται με έναν περιοδικό ή μη τρόπο. Διατηρούνται μόνο εκείνα τα εξαρτήματα του φυσικού εξοπλισμού που είναι αναγκαία για την επιθυμητή λειτουργικότητα. Τα συστήματα αυτά ανταποκρίνονται συνεχώς σε τυχόν μεταβολές που συμβαίνουν στο περιβάλλον τους και πρέπει σε κάθε αλλαγή να εκτελέσουν κάποιους υπολογισμούς (συνήθως τους ζητείται να παραδώσουν κάποια αποτελέσματα). Σε αυτήν την περίπτωση, η αρχιτεκτονική του συστήματος περιέχει διάφορους αισθητήρες καθώς και οντότητες που προβαίνουν σε συγκεκριμένες ενέργειες όταν ενημερωθούν από τους αισθητήρες ότι συνέβη κάποια αλλαγή στο σύστημα.

Ωστόσο, παρά την χρησιμότητα τους, τα ενσωματωμένα συστήματα έχουν κάποιους περιορισμούς όσον αφορά την λειτουργία τους. Καταρχήν, συνήθως έχουν μικρή μνήμη, η οποία καταλαμβάνει αξιόλογο μερίδιο του κόστους του συστήματος. Είναι συστήματα μικρού μεγέθους και χαμηλού βάρους, περιορίζοντας έτσι τα είδη των λειτουργιών που μπορούν να εκτελέσουν. Πρέπει να έχουν υψηλή αποτελεσματικότητα, ιδιαίτερα σε περιπτώσεις που αναφερόμαστε σε πραγματικού-χρόνου συστήματα όπου οι τυχόν υπολογισμοί πρέπει να γίνουν σε σύντομο χρονικό διάστημα. Τέλος, πρέπει να είναι ικανά να λειτουργούν κάτω από αντίξοες συνθήκες, όπως ζέστη, δονήσεις κα, διότι πολλά από αυτά προορίζονται για λειτουργία σε τέτοια περιβάλλοντα (για παράδειγμα, το σύστημα ABS των φρένων των αυτοκινήτων).

Πολλές φορές, τα ενσωματωμένα συστήματα είναι πραγματικού-χρόνου, που σημαίνει ότι η συμμόρφωσή τους με τους χρονικούς περιορισμούς που απαιτούνται καθορίζει την αξιοπιστία του συστήματος. Υπάρχουν δυο ειδών τέτοιων συστημάτων, τα "Hard real-time" συστήματα και τα "Soft real-time". Στην πρώτη περίπτωση, υπάρχουν συγκεκριμένες απόλυτες διορίες που πρέπει να ικανοποιηθούν, διαφορετικά η απάντηση ή λειτουργία του συστήματος είναι ανούσια. Μπορεί να υπάρχει ορισμένος ελάχιστος και μέγιστος χρόνος ανταπόκρισης του συστήματος. Στην δεύτερη περίπτωση, η

απώλεια κάποιας διορίας δεν είναι καταστροφική. Η χρησιμότητα του αποτελέσματος που προκύπτει από μια ενέργεια του συστήματος, εξαρτάται από την διαφορά που υπάρχει μεταξύ της διορίας και της χρονικής στιγμής που έγινε διαθέσιμο το αποτέλεσμα του συστήματος. Για παράδειγμα σε έναν αποκωδικοποιητή σήματος (digital set-top box), ο χρόνος που έχουμε για να επεξεργαστούμε ένα frame πριν έρθει το επόμενο, είναι περιορισμένος (30 ms).

Στην περίπτωση τέτοιων συστημάτων, ο χαρακτηρισμός "πραγματικού-χρόνου" δεν σημαίνει απαραίτητα ότι πρόκειται για ένα γρήγορο σύστημα. Ακριβείς ορισμοί για τους παραπάνω χαρακτηρισμούς είναι:

- **Πραγματικού-χρόνου συστήματα:** Ικανοποιούν κάποιους συγκεκριμένους χρονικούς περιορισμούς...
- **Γρήγορα συστήματα:** Παράγουν αποτελέσματα με έναν γρήγορο ρυθμό...

Παρόλα αυτά, τα περισσότερα πραγματικού-χρόνου ενσωματωμένα συστήματα είναι σχεδιασμένα να λειτουργούν σε γρήγορες ταχύτητες.

Η σχεδίαση των ενσωματωμένων συστημάτων βασίζεται σε μια από τις παρακάτω προσεγγίσεις. Πρώτα από όλα, βασική μας προτεραιότητα είναι η κατανόηση του προβλήματος και της λειτουργίας που θα διεκπεραιώνει ένα ενσωματωμένο σύστημα. Στην συνέχεια, μπορούμε να χρησιμοποιήσουμε ένα συνδυασμό ειδικά σχεδιασμένου φυσικού εξοπλισμού (hardware που προορίζεται για την κάλυψη συγκεκριμένων αναγκών) και λογισμικού, πάνω σε έναν επεξεργαστή ενσωματωμένου συστήματος. Διαφορετικά, θα μπορούσαμε να χρησιμοποιήσουμε έναν "ειδικής-εφαρμογής" επεξεργαστή (επεξεργαστής που έχει βελτιστοποιηθεί για να εκτελεί το συγκεκριμένο είδος εφαρμογών) μαζί με ένα ειδικά σχεδιασμένο λογισμικό.

Κεφάλαιο 1

Περίληψη των βασικών αρχών των ενσωματωμένων συστημάτων

Στο κεφάλαιο αυτό, θα περιγράψουμε κάποιες βασικές αρχές των υπολογιστικών συστημάτων που ισχύουν και στα ενσωματωμένα συστήματα. Ακόμα, θα δούμε τις βασικότερες αρχιτεκτονικές που χρησιμοποιούνται σε τέτοια συστήματα, διάφορα χαρακτηριστικά τους καθώς και διάφορες λειτουργίες τους που έχουν ως στόχο την βελτιστοποίηση της απόδοσης τους.

Όσον αφορά την περιγραφή κάποιων τεχνικών χαρακτηριστικών των επεξεργαστών των ενσωματωμένων συστημάτων, θα περιοριστούμε στην περιγραφή του ARM επεξεργαστή[3] ο οποίος είναι ένας από τους κυρίαρχους επεξεργαστές στην συγκεκριμένη κατηγορία συστημάτων (μέχρι το τέλος του 2001 είχαν διατεθεί στην αγορά πάνω από 1 δισεκατομμύριο ARM επεξεργαστές). Άλλωστε, τα περισσότερα χαρακτηριστικά τέτοιων επεξεργαστών είναι κοινά, οπότε μετά την παρουσίαση του συγκεκριμένου επεξεργαστή, η γνώση μας για τα ενσωματωμένα συστήματα θα μπορεί να επεκταθεί και σε άλλους επεξεργαστές.

Θεμελιώδεις αρχές των υπολογιστικών συστημάτων

Καταρχήν, θα ξεκινήσουμε με την επεξήγηση των βασικών εννοιών όπως bytes, λέξεις (words), halfwords και εντολές. Ένα byte είναι η βασικότερη μορφή αναπαράστασης δεδομένων σε ένα υπολογιστικό σύστημα και αποτελείται από 8 bits. Ομοίως, μια λέξη (word), αποτελείται από 32 bits και πρέπει να είναι ευθυγραμμισμένη ανά 4 bytes. Τέλος, μια halfword

[3] Η προσομοίωση που έχουμε υλοποιήσει έχει γίνει πάνω σε ARM επεξεργαστή

αποτελείται από 16 bits και πρέπει να είναι ευθυγραμμισμένη ανά 2 bytes. Οι εντολές συνήθως είναι μια λέξη (για παράδειγμα, εντολές των 32 bit).
Τι είναι όμως μια εντολή; Ουσιαστικά, είναι η βασικότερη μονάδα εκτέλεσης ενός επεξεργαστή. Η επεξεργασία δεδομένων, η μετακίνηση δεδομένων, η εκτέλεση προγραμμάτων, όλα στηρίζονται στις εντολές. Χαρακτηρίζεται από ένα κωδικό λειτουργίας (που ονομάζεται opcode) ο οποίος περιγράφει την ενέργεια που πρέπει να εκτελεστεί. Για παράδειγμα, μια εντολή της μορφής:

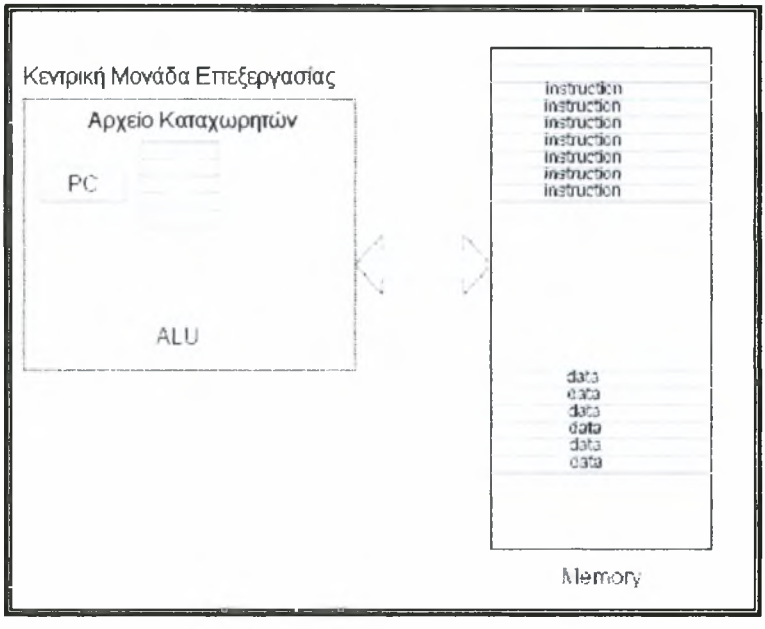
MUL x by 5 (και αποθήκευσε το αποτέλεσμα στην μεταβλητή y)

Υποδεικνύει την πράξη του πολλαπλασιασμού. Το x είναι ο τελεστέος (source operand) στον οποίο θα εφαρμοστεί η συγκεκριμένη πράξη – ουσιαστικά, η εντολή εκτελείται πάνω στο αντικείμενο αυτό – ενώ το y είναι η θέση αποθήκευσης του αποτελέσματος (result operand). Τέλος, συνήθως, στα διάφορα προγράμματα χρησιμοποιούμε και αναφορές στις επόμενες εντολές που ουσιαστικά είναι η επόμενη ενέργεια που πρέπει να εκτελεσθεί.

Μια εντολή αποτελείται από 3 βήματα: την φάση Ανάκλησης της εντολής από την μνήμη (Fetch), την φάση Αποκωδικοποίησης (Decode) και την φάση Εκτέλεσης (Execute). Κατά την φάση Ανάκλησης, ο επεξεργαστής ανακτά την εντολή προς εκτέλεση (χρησιμοποιώντας τον μετρητή προγράμματος) από την μνήμη. Αυξάνει τον μετρητή προγράμματος και στέλνει την εντολή για αποκωδικοποίηση. Η εντολή αποκωδικοποιείται προκειμένου να καθορίσουμε το είδος της λειτουργίας που αντιπροσωπεύει. Στην συνέχεια, έχουμε την εκτέλεση της εντολής, η οποία μπορεί να είναι μια αλληλεπίδραση μεταξύ της ΚΜΕ και της κύριας μνήμης, μια επεξεργασία δεδομένων, μια συνάρτηση ελέγχου (για παράδειγμα εντολές branch) ή ένας συνδυασμός όλων των παραπάνω.
Έχοντας αναλύσει όλα τα παραπάνω, η επόμενη ερώτηση που πρέπει να διερωτηθούμε είναι τι γίνεται στην περίπτωση που συμβεί ένα interrupt κατά την διάρκεια των παραπάνω φάσεων; Ουσιαστικά, αυτό δεν μπορεί να συμβεί διότι οι εντολές είναι ατομικές (atomic). Αυτό σημαίνει ότι είτε εκτελούνται ολόκληρες (χωρίς την δυνατότητα διακοπής) είτε δεν εκτελούνται καθόλου. Άρα, άμα συμβεί ένα interrupt κατά την διάρκεια μιας από τις παραπάνω φάσεις, ο επεξεργαστής θα την χειριστεί αφού τελειώσει πρώτα η εκτέλεση της εντολής που υπάρχει εκείνη την στιγμή στο σύστημα. Τέλος, ένας τρόπος

βελτιστοποίησης της εκτέλεσης των φάσεων μιας εντολής είναι η διαδικασία του pipelining που περιγράφεται αναλυτικά παρακάτω.

Οι εντολές προς εκτέλεση αποθηκεύονται στους καταχωρητές (registers) του συστήματος. Στον επεξεργαστή ενός υπολογιστικού συστήματος, έχουμε διάφορα chip (ALU- μονάδα υλοποίησης αριθμητικών πράξεων), τα οποία συνδυάζονται όπως φαίνεται στην παρακάτω εικόνα:

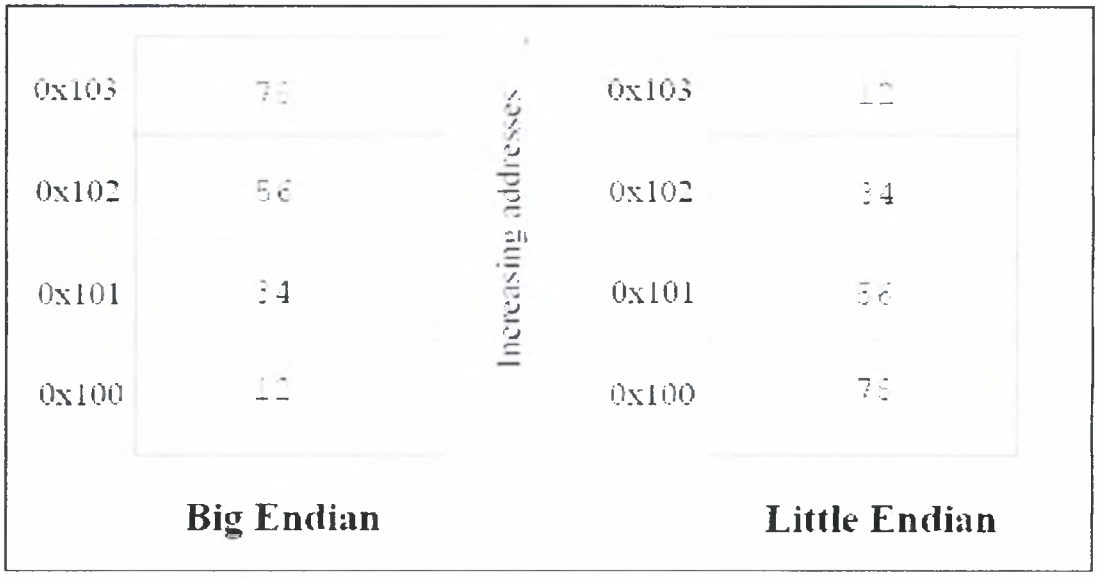


Η Κεντρική Μονάδα Επεξεργασίας, χρειάζεται κάποιο προσωρινό χώρο αποθήκευσης τον οποίο θα χρησιμοποιεί κατά την διάρκεια εκτέλεσης μιας λειτουργίας. Επομένως, μπορούμε να σκεφτόμαστε τους καταχωρητές σαν το "πρόχειρο" της ΚΜΕ. Ο αριθμός των καταχωρητών και οι λειτουργίες που υποστηρίζουν μπορεί να είναι διαφορετικός από επεξεργαστή σε επεξεργαστή. Υπάρχουν οι γενικού-σκοπού καταχωρητές, οι οποίοι είναι ορατοί στους χρήστες, χρησιμοποιούνται για την αποθήκευση δεδομένων ή διευθύνσεων και είναι αρκετά μεγάλοι σε χωρητικότητα για να μπορέσουν να αποθηκεύσουν μια ολόκληρη λέξη ή μια πλήρης διεύθυνση. Ακόμα, υπάρχει και ο μετρητής προγράμματος (program counter), ο οποίος παρέχει μια ένδειξη για το ποια εντολή εκτελείται την τρέχουσα χρονική στιγμή. Ανάλογα με την αρχιτεκτονική του συστήματος, ο μετρητής προγράμματος μπορεί να αναφέρεται άμεσα στην διεύθυνση της τωρινής εντολής ή μπορεί να έχει αποθηκευμένο έναν συγκεκριμένο αριθμό από bytes (offset) ο οποίος προστιθέμενος σε μια διεύθυνση "δείχνει" στην εντολή που εκτελείται εκείνη την στιγμή στο σύστημα. Τέλος, υπάρχουν και οι ειδικού-σκοπού καταχωρητές οι οποίοι χρησιμοποιούνται από το σύστημα και δεν μπορούν να ορισθούν από τον χρήστη. Για παράδειγμα, έχουμε τους "condition-code

registers" οι οποίοι ουσιαστικά υποδεικνύουν εάν το αποτέλεσμα της προηγούμενης ενέργειας που εκτελέστηκε ήταν ο. Ακόμα, έχουμε τους καταχωρητές CPSR (Current Program Status register) και SPSR (Saved Program Status Register) οι οποίοι ασχολούνται με την κατάσταση του προγράμματος κάθε χρονική στιγμή της εκτέλεσής του. Ο πρώτος καταγράφει την τρέχουσα κατάσταση του προγράμματος, καθώς εκτελείται, και συνήθως περιλαμβάνει και το ποια interrupts έχουν απενεργοποιηθεί ή όχι. Ο δεύτερος μας χρησιμεύει στην περίπτωση που αλλάζουμε τρόπο λειτουργίας[4] (εκτελούμε ένα switching mode από το user mode στο supervisor mode για παράδειγμα) να σώσουμε την τρέχουσα κατάσταση του προγράμματος, έτσι ώστε να μπορούμε να την ανακτήσουμε όταν επιστρέψουμε στον αρχικό τρόπο λειτουργίας.

Στην συνέχεια, θα αναφερθούμε στην μνήμη του συστήματος, η οποία μπορεί να θεωρηθεί σαν ένας γραμμικός πίνακας από bytes. Πολλά είδη δεδομένων, χρησιμοποιούν πολλαπλάσια bytes για την αναπαράσταση τους (για παράδειγμα οι ακέραιοι καταλαμβάνουν 4 bytes). Συνήθως, τα δεδομένα που είναι αποθηκευμένα στην μνήμη είναι ευθυγραμμισμένα ανά 4 bytes, διότι έτσι βελτιστοποιείται ο τρόπος πρόσβασης της KME στην μνήμη. Σε ορισμένες περιπτώσεις, επιτρέπεται η πρόσβαση σε μη ευθυγραμμισμένα δεδομένα, αλλά υπάρχει μεγάλη πιθανότητα να οδηγηθούμε σε λάθη. Τέτοια αναπαράσταση δεδομένων (ευθυγραμμισμένη) χρησιμοποιούν και οι δυο κυριότεροι τρόποι αποθήκευσης στην μνήμη, ο Big Endian και ο Little Endian (βλέπε παρακάτω φωτογραφία όπου δείχνουμε την αποθήκευση της λέξης 0x12345678).

[4] Υπάρχουν δυο τρόποι λειτουργίας (modes), το user mode – ουσιαστικά είναι το περιβάλλον όπου δουλεύει ο χρήστης – και το supervisor mode – το περιβάλλον που διαχειρίζεται το λειτουργικό σύστημα και δεν είναι ορατό στον χρήστη. Εάν για παράδειγμα, ένας χρήστης θέλει να διαβάσει κάτι από τον δίσκο, θα πρέπει να παράγει ένα interrupt έτσι ώστε να μεταβούμε στο supervisor mode (το λειτουργικό σύστημα χειρίζεται όλα τα interrupts) όπου θα εκτελεστεί η ανάγνωση από τον δίσκο.



Η κυριότερη διαφορά τους έγκειται στο γεγονός ότι στον Big Endian τρόπο αποθήκευσης, αποθηκεύουμε το πιο σημαντικό byte στην χαμηλότερη θέση μνήμης (και διαβάζουμε από αριστερά προς τα δεξιά), ενώ στον Little Endian τρόπο αποθήκευσης αποθηκεύουμε το λιγότερο σημαντικό byte στην χαμηλότερη θέση μνήμης (και διαβάζουμε από δεξιά προς τα αριστερά).

Τύποι Αρχιτεκτονικής μικροεπεξεργαστών – CISC vs. RISC

Το βασικό χαρακτηριστικό της RISC (Reduced Instruction Set Computers) αρχιτεκτονικής, είναι ότι υποστηρίζει μικρό αριθμό εντολών (συνήθως λιγότερες από 100) σε αντίθεση με την CISC (Complex Instruction Set Computers) αρχιτεκτονική όπου για την εκτέλεση των ίδιων λειτουργιών έχουμε έναν αρκετά μεγάλο αριθμό εντολών. Με άλλα λόγια, όπως φαίνεται και στην παρακάτω φωτογραφία, η βασικότερη διαφορά μεταξύ των δυο αρχιτεκτονικών έγκειται στην πολυπλοκότητα των ενεργειών και στον αριθμό των βημάτων που εκτελούνται για την διεκπεραίωση της ίδιας λειτουργίας. Ας υποθέσουμε, ότι θέλουμε να πολλαπλασιάσουμε δυο αριθμούς στις θέσεις μνήμης memo και mem1 και θέλουμε να αποθηκεύσουμε το αποτέλεσμα στην θέση memo. Οι εντολές που χρησιμοποιούνται στις δυο αρχιτεκτονικές για την παραπάνω λειτουργία φαίνονται παρακάτω:

CISC Approach	RISC Approach

mul mem0, mem1	ldr r1, mem0
	ldr r1, mem1
	mul r0, r0, r1
	str mem0, r0

Μια ακόμη διαφορά μεταξύ των δυο αρχιτεκτονικών, είναι ότι στην RISC αρχιτεκτονική χρησιμοποιούμε περισσότερους καταχωρητές, από ότι στην CISC, προκειμένου να μειώσουμε τον αριθμό των προσβάσεων που κάνει η ΚΜΕ στην μνήμη. Επιπροσθέτως, η CISC αρχιτεκτονική υποστηρίζει μεταβλητό μέγεθος εντολών (όπου μπορούμε να έχουμε εντολές με μεγάλο χρόνο εκτέλεσης), ενώ η RISC μόνο σταθερό μέγεθος εντολών (όποτε έχουμε εντολές με μικρό χρόνο εκτέλεσης). Επιπλέον διαφορές μεταξύ των δυο αρχιτεκτονικών φαίνονται στον παρακάτω πίνακα:

RISC αρχιτεκτονική	**CISC αρχιτεκτονική**
Αριθμητικές και λογικές πράξεις μπορούν να εφαρμοστούν σε τελεστές που είναι αποθηκευμένοι είτε στην μνήμη είτε σε καταχωρητές	Αριθμητικές και λογικές πράξεις εφαρμόζονται μόνο σε τελεστές που βρίσκονται στους καταχωρητές (περιεχόμενα της μνήμης πρέπει πρώτα να φορτωθούν σε καταχωρητές[5])
Όταν καλούμε διάφορες συναρτήσεις, οι παράμετροί τους αποθηκεύονται σε στοίβα	Όταν καλούμε διάφορες συναρτήσεις, οι παράμετροι αποθηκεύονται σε καταχωρητές
Μικρό μέγεθος μεταγλωττισμένου κώδικα	Μεγάλο μέγεθος μεταγλωττισμένου κώδικα
Απαιτούνται περισσότερα τρανζίστορ για την εκτέλεση διάφορων ενεργειών -> κατανάλωση περισσότερης ενέργειας	Λιγότερα τρανζίστορ χρησιμοποιούνται για την εκτέλεση διάφορων ενεργειών -> κατανάλωση λιγότερης ενέργειας

[5] Για αυτό η αρχιτεκτονική αυτή χαρακτηρίζεται σαν load/store αρχιτεκτονική

Διαδικασία βελτιστοποίησης – Pipelining

Το pipelining, είναι μια τεχνική όπου οι εντολές επικαλύπτουν η μια την άλλη κατά την διάρκεια της εκτέλεσής τους. Κάθε στάδιο της τεχνικής, ολοκληρώνει παράλληλα ένα μέρος της εκτέλεσης των εντολών. Ωστόσο, το pipelining, δεν αυξάνει (και αντίστοιχα δεν μειώνει) τον χρόνο που παίρνει μια απλή εντολή για να εκτελεστεί. Το μόνο που αυξάνει είναι το συνολικό όγκο έργου (throughput) του συστήματος, με άλλα λόγια τον συνολικό αριθμό των εντολών που εξέρχονται από το σύστημα σε μια μονάδα χρόνου.

Ο αριθμός των σταδίων σε έναν pipeline επεξεργαστή ποικίλει. Συνήθως, όμως, αποτελείται από τα παρακάτω βήματα:

- Ανάκληση εντολών από την μνήμη
- Αποκωδικοποίηση εντολής
- Εκτέλεση εντολής ή υπολογισμός μιας διεύθυνσης
- Πρόσβαση σε έναν τελεστή που είναι αποθηκευμένος στην μνήμη
- Εγγραφή του αποτελέσματος σε έναν καταχωρητή

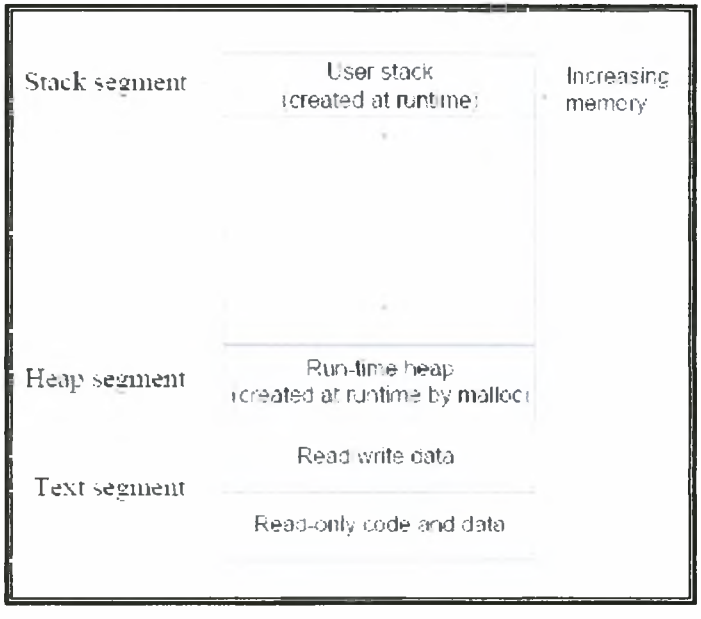
Η διάρκεια ενός σταδίου σε μια pipeline τεχνική, εξαρτάται από την χρονική διάρκεια του μεγαλύτερου από τα παραπάνω βήματα. Όλα τα στάδια πρέπει να είναι συγχρονισμένα για να προχωρούν την εκτέλεσή τους ταυτόχρονα, ενώ ο ρυθμός ολοκλήρωσης του pipeline περιορίζεται από το πιο αργό στάδιο (όσον αφορά τον ρυθμό εκτέλεσης και ολοκλήρωσης).

Στην συνέχεια θα αναφερθούμε στην RISC αρχιτεκτονική (είναι αυτή που χρησιμοποιεί ο ARM επεξεργαστής) και στις δυνατότητες που προσφέρει για pipelining. Γενικά, οι εντολές στην αρχιτεκτονική αυτή είναι πιο απλές από αυτές της CISC αρχιτεκτονικής και έτσι μπορούμε να πούμε ότι η δυνατότητα για επικάλυψη εντολών είναι αρκετά μεγάλη. Οι εντολές της RISC έχουν συνήθως το ίδιο μέγεθος και άρα αρκετές μπορούν να ανακληθούν σε μια μόνο ενέργεια (operation). Στην ιδεατή κατάσταση, κάθε στάδιο του pipeline θα έπρεπε να καταλαμβάνει 1 κύκλο μηχανής, έτσι ώστε ο επεξεργαστής να ολοκληρώνει μια εντολή κάθε κύκλο.

Ωστόσο όμως, τα πράγματα δεν είναι τόσο απλά διότι έχουμε τις εντολές διακλάδωσης (branch instructions). Τι συμβαίνει όμως σε μια εντολή διακλάδωσης; Είναι η περίπτωση όπου η επόμενη εντολή προς εκτέλεση δεν ήταν αυτή που αποκωδικοποιήθηκε στο pipeline. Επομένως, ολόκληρη η διαδικασία πρέπει να καθυστερήσει για μερικούς κύκλους μηχανής, συγκεκριμένα μέχρι να ανακληθεί και να αποκωδικοποιηθεί η σωστή εντολή. Ωστόσο, όμως, κάτι τέτοιο μειώνει την απόδοση του συστήματος.
Άρα, όπως μπορούμε να δούμε, η τεχνική του pipeline παρόλο που στις περισσότερες περιπτώσεις βελτιστοποιεί την απόδοση του συστήματος, υπάρχει και η πιθανότητα να αποτελεί ελάττωμα σε ένα σύστημα που περιέχει πολλές εντολές διακλάδωσης.

Αρχιτεκτονική του επεξεργαστή ARM

Σε ένα υπολογιστικό σύστημα, όταν ένα πρόγραμμα φορτώνεται στην μνήμη, οργανώνεται σε τρεις περιοχές που ονομάζονται τμήματα (segments), το "Text Segment", το "Stack Segment" και το "Heap Segment" (βλέπε την παρακάτω φωτογραφία).



Η πρώτη περιοχή (κάποιες φορές ονομάζεται και code segment) είναι το σημείο όπου βρίσκεται ο μεταγλωττισμένος κώδικας του προγράμματος. Η δεύτερη περιοχή, είναι ένα μέρος της μνήμης το οποίο ανατίθεται στις μεταβλητές των διαφόρων συναρτήσεων (στην περίπτωση που έχουμε εμφωλευμένες κλήσεις συναρτήσεων, οι μεταβλητές αποθηκεύονται στην στοίβα). Τέλος, η τρίτη περιοχή, παρέχει πιο σταθερές θέσεις μνήμης για τα δεδομένα ενός προγράμματος. Παραμένει ενεργή για όλη την διάρκεια εκτέλεσης ενός προγράμματος και γενικά χρησιμοποιείται

για την αποθήκευση των καθολικών (global) και των στατικών (static) μεταβλητών.

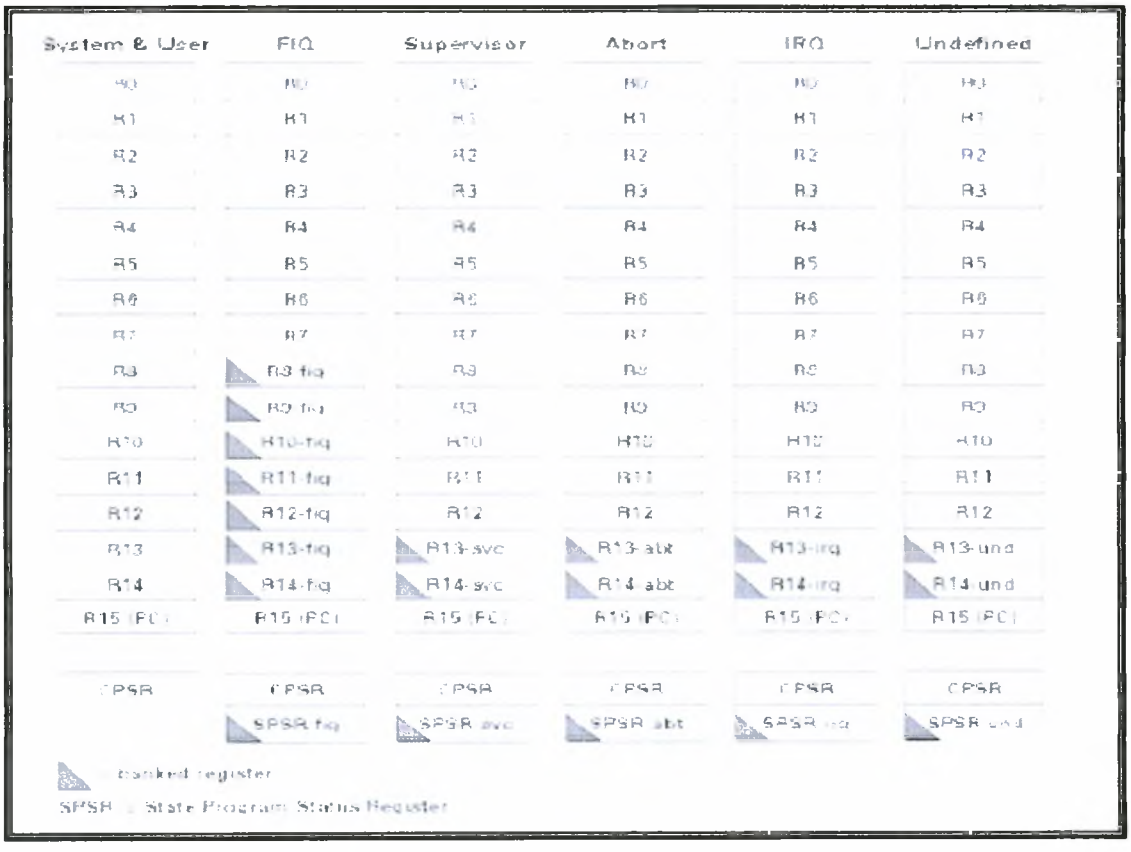
Όσον αφορά τον ARM επεξεργαστή, χρησιμοποιεί την RISC αρχιτεκτονική με εντολές των 32 bits. Περιέχει 7 βασικούς τρόπους λειτουργίας:

- **User mode:** Περιβάλλον οπού εκτελούνται κανονικά τα προγράμματα του χρήστη (δεν έχουμε πολλά προνομία όσον αφορά τους καταχωρητές, την μνήμη κα).
- **FIQ (Fast Interrupt Request) mode:** Περιβάλλον όπου μεταβαίνουμε όταν θέλουμε να χειριστούμε ένα "γρήγορο" interrupt (περιορισμός για μικρό interrupt latency[6]). Στον συγκεκριμένο τρόπο λειτουργίας, έχουμε περισσότερους κρυφούς καταχωρητές (ο όρος αυτός αναλύεται παρακάτω) διότι επιθυμούμε ένα γρήγορο response time. Με το να χρησιμοποιούμε μεγαλύτερο αριθμό καταχωρητών, μειώνουμε τον αριθμό των προσβάσεων που πρέπει να γίνουν στην μνήμη (για αποθήκευση των κοινών καταχωρητών) και έτσι επιτυγχάνουμε μικρή καθυστέρηση στην εκτέλεση της εντολής (τρόπος λειτουργίας με προνόμια).
- **IRQ (Interrupt Request) mode:** Περιβάλλον όπου μεταβαίνουμε όταν θέλουμε να εξυπηρετήσουμε ένα χαμηλής προτεραιότητας interrupt (τρόπος λειτουργίας με προνόμια).
- **Supervisor mode:** Περιβάλλον όπου μεταβαίνουμε όταν παράγεται ένα reset σήμα ή όταν αντιλαμβανόμαστε ένα interrupt για εκτέλεση εντολής λογισμικού (τρόπος λειτουργίας με προνόμια).
- **Abort mode:** Χρησιμοποιείται για χειρισμό παραβιάσεων όσον αφορά την πρόσβαση στην κύρια μνήμη (τρόπος λειτουργίας με προνόμια).
- **Undefined mode:** Χρησιμοποιείται για τον χειρισμό απροσδιόριστων εντολών (τρόπος λειτουργίας με προνόμια).
- **System mode:** Τρόπος λειτουργίας με προνόμια ο οποίος χρησιμοποιεί τους ίδιους καταχωρητές με το user mode.

[6] Interrupt latency: Ο χρόνος που μεσολαβεί από την στιγμή που έφτασε ένα interrupt στον επεξεργαστή μέχρι την χρονική στιγμή που ανακλήθηκε η πρώτη εντολή από τον αντίστοιχο interrupt handler που χειρίζεται το συγκεκριμένο είδος interrupts

Ακόμα, ο ARM έχει ένα σύνολο 37 καταχωρητών οι οποίοι έχουν μέγεθος 32 bits. Από αυτούς τους 37 καταχωρητές, οι 30 είναι καταχωρητές γενικού σκοπού, 1 χρησιμοποιείται για την αποθήκευση του μετρητή προγράμματος, 1 για την αποθήκευση του CPSR και 5 είναι αφιερωμένοι για την αποθήκευση του SPSR. Σε κάθε τρόπο λειτουργίας, μόνο ένα υποσύνολο των καταχωρητών αυτών είναι ορατό. Οι υπόλοιποι ονομάζονται "κρυφοί καταχωρητές" (banked registers). Συνήθως, ένα σύνολο 13 καταχωρητών (από τον r0 έως τον r12) είναι ορατοί σε όλους τους τρόπους λειτουργίας. Ο καταχωρητής r13 περιέχει έναν δείκτη που αναφέρεται στην στοίβα (stack pointer) , ενώ ο r14 περιέχει έναν δείκτη στην εντολή που εκτελείται εκείνη την χρονική στιγμή στο σύστημα (μας χρησιμεύει όταν για παράδειγμα επιθυμούμε να χειριστούμε ένα interrupt και αλλάζουμε τρόπο λειτουργίας, να γνωρίζουμε σε ποια εντολή θα επιστρέψουμε). Τέλος, ο καταχωρητής r15, περιέχει τον μετρητή προγράμματος.

Στην παρακάτω φωτογραφία βλέπουμε τους ορατούς και κρυφούς καταχωρητές του κάθε τρόπου λειτουργίας:



Όπως παρατηρούμε, όταν μεταβαίνουμε από έναν τρόπο λειτουργίας σε έναν άλλον (switching mode), πρέπει να αποθηκεύσουμε στην μνήμη μόνο τους κοινούς καταχωρητές που χρησιμοποιούνται από τους εκάστοτε τρόπους λειτουργίας. Οι banked registers είναι ορατοί μόνο στους τρόπους λειτουργίας για τους οποίους έχουν οριστεί και επομένως δεν υπάρχει κίνδυνος overwrite των καταχωρητών αυτών. Με αυτόν τον τρόπο, πραγματοποιώ μικρότερο αριθμό προσβάσεων στην μνήμη, αφού αποθηκεύω λιγότερους καταχωρητές.

Κεφάλαιο 2

Εισαγωγή στα Interrupts και στα Exceptions

Τι είναι ένα interrupt; Ουσιαστικά, είναι ένας μηχανισμός με τον οποίο διάφορες οντότητες μπορούν να διακόπτουν την κανονική λειτουργία του επεξεργαστή. Η πηγή ενός interrupt, μπορεί να είναι τόσο μια συσκευή εισόδου όσο και μια συσκευή εξόδου, στην περίπτωση που για παράδειγμα έχουν έτοιμα δεδομένα για επεξεργασία. Επίσης, πηγή ενός interrupt μπορεί να είναι ο χρονομέτρης (timer) του συστήματος, σε περιπτώσεις όπως όταν νέες διεργασίες χρειάζονται να χρονοπρογραμματιστούν (scheduled) ή όταν συμβαίνουν συγκεκριμένα γεγονότα που αλλάζουν την κατάσταση του timer. Τέλος, πηγή ενός interrupt μπορεί να είναι το ίδιο το πρόγραμμα εφαρμογής όταν για παράδειγμα δημιουργεί μια νέα διεργασία – παιδί.

Επιπλέον, ένας άλλος ορισμός περιγράφει το interrupt σαν ένα εξωτερικό γεγονός το οποίο συμβαίνει κατά την διάρκεια εκτέλεσης ενός προγράμματος και διακόπτει την λειτουργία του. Γενικά, αυτά τα γεγονότα έχουν σχέση με τον φυσικό εξοπλισμό (hardware) του συστήματος, όπως το πάτημα ενός κουμπιού ή η εκπνοή ενός χρονοδιακόπτη. Interrupts μπορούν να συμβούν οποιαδήποτε στιγμή. Επιτρέπουν σε λειτουργίες που είναι ευαίσθητες στον χρόνο να εκτελούνται με μεγαλύτερη προτεραιότητα από κάποιες άλλες που δεν έχουν χρονικούς περιορισμούς.

Ένα συγκεκριμένο είδος interrupts είναι τα exceptions. Τα exceptions είναι συγχρονισμένα γεγονότα που συμβαίνουν κατά την διάρκεια εκτέλεσης μιας διεργασίας και διακόπτουν την κανονική ροή εκτέλεσης των εντολών. Εάν δεν χειριστούν κατάλληλα μέσα στο πρόγραμμα, τότε μπορεί να οδηγήσουν σε δυσάρεστες καταστάσεις όπως αποτυχία του συστήματος. Επομένως, μπορούμε να καταλάβουμε ότι ο χειρισμός τέτοιων exception είναι πολύ σημαντικός, ιδίως στα ενσωματωμένα συστήματα, όπου θέλουμε να αποφύγουμε τέτοιες αποτυχίες και να βελτιώσουμε την ευρωστία του λογισμικού. Η βασική διαφορά των exceptions με τα interrupts είναι, ότι τα

exceptions συνήθως αναφέρονται σε εσωτερικά γεγονότα της ΚΜΕ, ενώ τα interrupts αναφέρονται κυρίως σε εξωτερικά γεγονότα που προέρχονται από συσκευές εισόδου/ εξόδου. Τα exceptions δηλώνουν ότι το σύστημα βρίσκεται σε μια κατάσταση λάθους (μη ασφαλής κατάσταση). Παραδείγματα συνθηκών που οδηγούν σε exception είναι τα παρακάτω:

- Χρήση μιας απροσδιόριστης εντολής.
- Προσπάθεια πρόσβασης σε μια θέση της μνήμης που δεν είναι ευθυγραμμισμένη.
- Προσπάθεια πρόσβασης σε μια θέση της μνήμης για την οποία δεν έχουμε άδεια χρησιμοποίησης.

Τέλος, interrupts από εξωτερικές συσκευές συνήθως αναφέρονται και αυτά ως exceptions.

Παρακάτω, παρουσιάζουμε τα διάφορα exceptions που μπορεί να παρουσιαστούν στον ARM επεξεργαστή. Επομένως, έχουμε διαδοχικά ότι :

Exception	**Τρόπος Λειτουργίας συστήματος (mode)**	**Περιγραφή**
Reset	Supervisor	Πυροδοτείται από το πάτημα του κουμπιού reset
Απροσδιόριστη εντολή	Απροσδιόριστο	Συμβαίνει όταν ο επεξεργαστής δεν αναγνωρίζει την εντολή
Διακοπή για εκτέλεση εντολής λογισμικού	Supervisor	Αυτό είναι ένα ορισμένο από τον χρήστη συγχρονισμένο interrupt. Επιτρέπει σε ένα πρόγραμμα που τρέχει στο user mode να ζητήσει κάποια προνόμια άλλων modes
Prefetch Abort	Abort	Συμβαίνει όταν ο επεξεργαστής

		προσπαθεί να εκτελέσει μια εντολή που δεν είχε ανακληθεί από την μνήμη
Απόρριψη Δεδομένων	Abort	Συμβαίνει όταν μια εντολή μεταφοράς δεδομένων προσπαθεί να φορτώσει ή να σώσει δεδομένα σε μια διεύθυνση μνήμης που απαγορεύεται
IRQ	IRQ	Συμβαίνει όταν έχουμε απενεργοποιήσει τα IRQ interrupts, αλλά προκύπτει ένα interrupt αυτού του είδους
FIQ	FIQ	Συμβαίνει όταν έχουμε απενεργοποιήσει τα FIQ interrupts, αλλά προκύπτει ένα interrupt αυτού του είδους

Ρουτίνες εξυπηρέτησης Interrupts

Όταν συμβαίνει ένα interrupt, ο επεξεργαστής πηγαίνει σε μια συγκεκριμένη διεύθυνση για να εκτελέσει την ρουτίνα εξυπηρέτησης του συγκεκριμένου είδους interrupt (Interrupt Service Routine - ISR). Ο φυσικός εξοπλισμός ενός συστήματος για την υποστήριξη διάφορων ειδών interrupt, μπορεί να ποικίλει μεταξύ των αρχιτεκτονικών. Κάθε επεξεργαστής έχει τον δικό του αριθμό pins για να πυροδοτήσει αυτές τις ρουτίνες.

Τα περισσότερα interrupts, συσχετίζονται με ένα λογισμικό το οποίο εκτελείται όταν συμβαίνει ένα interrupt του συγκεκριμένου είδους. Που όμως είναι αποθηκευμένοι αυτοί οι χειριστές των interrupts; Υπάρχει μια περιοχή στην μνήμη που χρησιμοποιείται για αυτόν ακριβώς τον σκοπό και ονομάζεται **Vector Table**. Έχει μέγεθος 32 bytes και ξεκινάει από την θέση 0x00. Κάθε λέξη που αποθηκεύεται μέσα σε αυτόν τον πίνακα αντιπροσωπεύει και έναν συγκεκριμένο τύπο interrupt. Ουσιαστικά, δεν περιέχει ολόκληρη την ρουτίνα εξυπηρέτησης ενός interrupt, αλλά μια εντολή διακλάδωσης ή φόρτωσης που μας πηγαίνει στην πραγματική ρουτίνα εξυπηρέτησης. Κάθε τέτοια ρουτίνα έχει πρόσβαση στο δικό της σύνολο από καταχωρητές: έχει τον δικό της καταχωρητή r13 (δείκτης σε στοίβα), τον δικό της r14 (link register) και τον δικό της SPSR.

Στην πρώτη θέση του πίνακα αυτού, υπάρχει ο χειριστής για τα interrupts τύπου FIQ. Όπως αναφέραμε παραπάνω, τα interrupts αυτά έχουν κάποιους χρονικούς περιορισμούς και απαιτούν γρήγορη εξυπηρέτηση. Για αυτόν τον λόγο, η ρουτίνα εξυπηρέτησης αυτών των interrupt βρίσκεται αποθηκευμένη ακριβώς από πάνω από τον χειριστή τους στον Vector table και κάθε φορά που συμβαίνει ένα FIQ interrupt, δεν χάνουμε χρόνο με το να εκτελούμε μια εντολή διακλάδωσης ή φόρτωσης αλλά εκτελούμε απευθείας την ρουτίνα εξυπηρέτησης.

SoftWare Interrupt εντολές

Μια Software interrupt εντολή, είναι μια εντολή που υποστηρίζεται από το σύνολο εντολών του ARM και η οποία παράγει ένα exception το οποίο μπορεί να χειριστεί μόνο σε τρόπο λειτουργίας (mode) του συστήματος με προνόμια. Από την μια, είναι παρόμοια με μια κλήση υπορουτίνας, διότι οι παράμετροι και οι επιστρεφόμενες τιμές μπορούν να αποθηκευτούν σε καταχωρητές. Από την άλλη, μπορούμε να πούμε, ότι διαφέρει από μια κλήση υπορουτίνας, διότι ο επεξεργαστής ARM αποθηκεύει πάντα τον CPSR του user mode, μεταβαίνει (εάν δεν βρίσκεται ήδη στο supervisor mode) και ξεκινά την εκτέλεσή του από

μια συγκεκριμένη διεύθυνση (πάντα από την 0x08). Γενικά, μια SWI εντολή, δίνει την δυνατότητα σε ένα πρόγραμμα που τρέχει στο user mode να ζητήσει την εκτέλεση μιας λειτουργίας η οποία μπορεί να γίνει μόνο σε ένα mode με προνόμια (για παράδειγμα εφαρμογές όταν επιθυμούν να καλέσουν ρουτίνες του λειτουργικού συστήματος). Η μορφή μιας τέτοιας εντολής φαίνεται στην παρακάτω φωτογραφία:

Όπως μπορούμε να δούμε, τα τελευταία 24 bits περιέχουν το νούμερο της SWI εντολής και χρησιμοποιούνται για να δείξουν το είδος της λειτουργίας που θα εκτελεστεί. Για παράδειγμα, η εντολή SWI 0x18 θα μπορούσε να αναφέρεται σε έναν χειριστή ο οποίος θα διαβάζει χαρακτήρες από το πληκτρολόγιο, ενώ η SWI εντολή 0x19 θα μπορούσε να υποστηρίζει την λειτουργία της εμφάνισης χαρακτήρων στην έξοδο. Τα επόμενα 4 bits είναι τα λεγόμενα mask bits που χρησιμοποιούνται για την ενεργοποίηση/ απενεργοποίηση των interrupts.

Πως όμως γίνεται η εξυπηρέτηση μιας SWI εντολής; Ας υποθέσουμε ότι αρχικά βρισκόμαστε στο User mode. Όταν συμβεί ένα exception, ο ARM επεξεργαστής :

- Αντιγράφει το CPSR του τρέχοντος mode στο SPSR.
- Θέτει τα κατάλληλα CPSR bits στην τιμή 10011 έτσι ώστε να απενεργοποιήσει περαιτέρω IRQs.
- Αποθηκεύει την τιμή του μετρητή προγράμματος (pc-4, λόγω του pipeline αρχικά ο μετρητής προγράμματος βρίσκεται 2 εντολές μπροστά. Επομένως, για να βρούμε την επόμενη εντολή προς εκτέλεσης αρκεί να αφαιρέσουμε 4 bytes από τον μετρητή προγράμματος) στον link register.
- Θέτει τον μετρητή προγράμματος στην διεύθυνση 0x008.

Για να μπορέσει να επιστρέψει από την ρουτίνα εξυπηρέτησης των Interrupts, ο επεξεργαστής πρέπει :

- Να ανακτήσει το CPSR από τον SPSR καταχωρητή (τώρα τα mode bits έχουν τιμή 1000, ενεργοποιημένα τα IRQs interrupts).
- Να ανακτήσει τον μετρητή προγράμματος από τον link register.

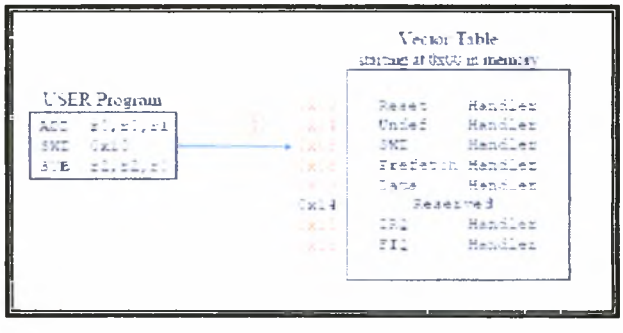
Σε modes με προνόμια, μπορούμε να ανακτήσουμε το CPSR από τον SPSR καταχωρητή και να αντιγράψουμε το περιεχόμενο του link register στον μετρητή προγράμματος, χρησιμοποιώντας μόνο μια εντολή:

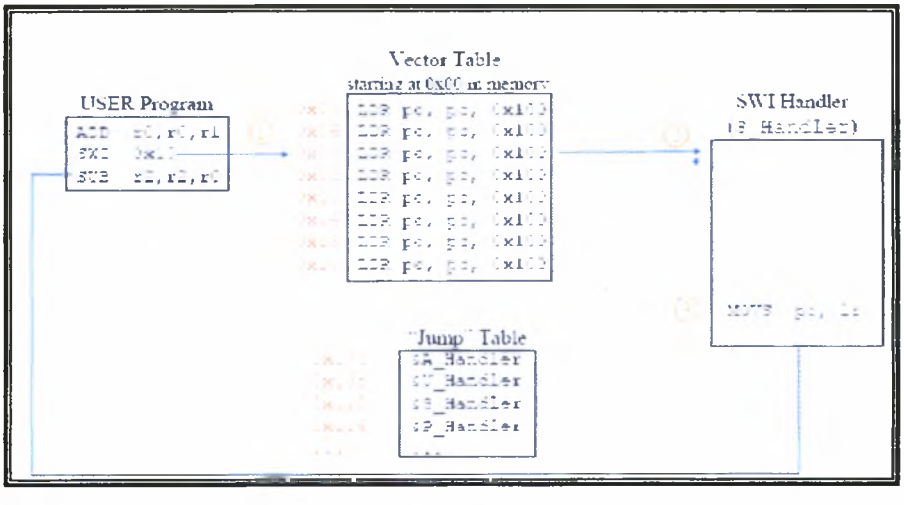
MOVS pc, lr

Τι γίνεται όμως κατά την εκτέλεση μιας SWI εντολής; Όπως αναφέραμε παραπάνω, η αρχιτεκτονική του ARM ορίζει ένα Vector Table ο οποίος έχει αποθηκευμένους όλους τους χειριστές για όλα τα πιθανά exception. Έτσι κατά την διάρκεια μιας SWI εντολής και σύμφωνα με όσα περιγράψαμε παραπάνω, εκτελείται ο παρακάτω κώδικας:

Επειδή όμως στον πίνακα δεν υπάρχει αρκετός χώρος για την αποθήκευση ολόκληρου του κώδικα ενός χειριστή, σε κάθε θέση του πίνακα υπάρχει αποθηκευμένη είτε μια εντολή διακλάδωσης η οποία μεταφέρει τον έλεγχο στον κατάλληλο SWI χειριστή (στην περίπτωση αυτή έχουμε περιορισμένη εμβέλεια επειδή το offset σε μια εντολή διακλάδωσης πρέπει να είναι

```
- lr_svc          ⇐ pc-4
- spsr_svc        ⇐ cpsr
- cpsr(mode bits) ⇐ 10011
- pc              ⇐ 0x08
```





[-32 MB, +32 MB]) είτε φορτώνουμε την διεύθυνση του πραγματικού χειριστή στον μετρητή προγράμματος χρησιμοποιώντας μια LDR εντολή. Στην δεύτερη περίπτωση, μια διεύθυνση μνήμης (που ονομάζεται softvec) αποθηκεύει την διεύθυνση του SWI χειριστή (S_HANDLER). Χρησιμοποιούμε την LDR εντολή για να φορτώσουμε το περιεχόμενο του softvec στον μετρητή προγράμματος. Αυτό γίνεται υπολογίζοντας το offset του softvec από τον SWI πίνακα και στην συνέχεια χρησιμοποιούμε την εντολή

LDR pc, [pc, #offset]

Για να φορτώσουμε την διεύθυνση του S_HANDLER στον μετρητή προγράμματος.

Έτσι, μετά από όλη την παραπάνω διαδικασία, εκτελούμε την ρουτίνα εξυπηρέτησης του συγκεκριμένου interrupt. Μόλις η εκτέλεση του χειριστή ολοκληρωθεί, ο επεξεργαστής επιστρέφει τον έλεγχο στο πρόγραμμα του χρήστη (στην εντολή που ακολουθούσε την SWI εντολή). Όπως είδαμε παραπάνω, η MOVS εντολή, ανακτά το αρχικό CPSR καθώς και τον σωστό μετρητή προγράμματος.

Interrupt vs. Polled I/O

Το polled I/ O, απαιτεί από την κεντρική μονάδα επεξεργασίας να ρωτήσει μια συσκευή (για παράδειγμα έναν διακόπτη) εάν χρειάζεται κάποιου είδους εξυπηρέτηση. Για παράδειγμα, εάν ο διακόπτης έχει αλλάξει θέση. Το λογισμικό στοχεύει σε συνεχές polling των συσκευών και είναι γραμμένο για να γνωρίζει πότε μια συσκευή εξυπηρετείται.

Το interrupt I/ O, από την άλλη μεριά, επιτρέπει σε μια συσκευή να διακόπτει τον επεξεργαστή, γνωστοποιώντας του το γεγονός ότι απαιτεί εξυπηρέτηση. Αυτό επιτρέπει στην ΚΜΕ να αγνοεί τις συσκευές μέχρι να του ζητήσουν προσοχή (μέσω διακοπών). Το λογισμικό δεν μπορεί να προγραμματίσει την εξυπηρέτηση των διακοπών, διότι αυτές μπορεί να συμβούν οποιαδήποτε στιγμή. Έτσι, δεν γνωρίζει απολύτως τίποτα για την χρονική στιγμή που

μπορεί να προκύψουν οι διακοπές. Αυτό δυσκολεύει ακόμα περισσότερο την συγγραφή κώδικα.

Ακόμα, οι επεξεργαστές μπορεί να προγραμματισθούν για να αγνοούν πιθανά interrupts. Την διαδικασία αυτή την ονομάζουμε **masking** των διακοπών. Σε κάθε χρονική στιγμή, μπορούμε να κάνουμε mask διαφορετικά είδη interrupts (όπως IRQ ή FIQ). Παρακάτω, υπάρχει ένα παράδειγμα κώδικα για polling I/ O, στην περίπτωση των διακοπών:

```
#define SWITCH_BASE         0x13000000
int main(int argc, char * argv[])
{
    volatile unsigned int *switchBank = (unsigned int *)
      SWITCH_BASE; //switchBank is now a pointer to the
      switch
    unsigned int tmpSwitchState;
    unsigned int prevSwitchState;
    /* get the current state of the switches */
    prevSwitchState = *switchBank & 0xFF;
       /* Wait for the switch to be pressed */
       while (prevSwitchState ==
              (tmpSwitchState = (*switchBank & 0xFF))) {};
       .. /* processing after the switch is pressed */
       ...
} /* end main() */
```

Γενικά, το polling I/ O, απαιτεί από τον κώδικα να κάνει busy waiting σε έναν βρόγχο μέχρι η συσκευή να είναι έτοιμη (γεγονός που οδηγεί στην σπατάλη αρκετών κύκλων μηχανής). Ωστόσο, όμως, εγγυάται γρήγορη εξυπηρέτηση και άρα χαμηλό response time.

Αντίθετα, το interrupt I/ O, δεν απαιτεί από τον κώδικα να περιμένει σε έναν βρόγχο. Οι συσκευές διακόπτουν τον επεξεργαστή όποτε αυτές είναι έτοιμες, γεγονός που επιτρέπει στον κώδικα να ασχολείται με άλλα πράγματα (άλλες διεργασίες). Ωστόσο, όμως, επειδή τα interrupts μπορεί να συμβούν οποιαδήποτε στιγμή, απαιτείται προσεκτική συγγραφή κώδικα έτσι ώστε να διασφαλίσουμε ότι η παρουσία των interrupts μέσα στο πρόγραμμά μας δεν δημιουργεί προβλήματα.

Το polling I/ O, το χρησιμοποιούμε σε περιπτώσεις που χρειαζόμαστε αναγκαστικά κάποια είσοδο από τον χρήστη ή από μια συσκευή προκειμένου

να προχωρήσουμε με την εκτέλεση του προγράμματός μας. Ακόμα, εάν ο επεξεργαστής δεν υποστηρίζει εμφωλευμένα interrupts και επιθυμώ να δω την κατάσταση μιας συσκευής, τότε και πάλι θα αναγκαστώ να χρησιμοποιήσω polling I/ O.

Παράρτημα Α

Το Εργαλείο που χρησιμοποιήσαμε

Η συγγραφή κώδικα έγινε σε γλώσσα C, ενώ χρησιμοποιήθηκε και το εργαλείο CodeWarrior IDE/ AXD Debugger για την εκτέλεση των προγραμμάτων στον ARM επεξεργαστή.

Παρακάτω, παραθέτουμε κάποιες πληροφορίες για τον τρόπο χρήσης του εργαλείου αυτού. Καταρχήν, ξεκινάμε το πρόγραμμα CodeWarrior IDE (Start Programs->ARM Developer Suite->CodeWarrior). Στην συνέχεια δημιουργούμε ένα νέο project ακολουθώντας τα παρακάτω βήματα :

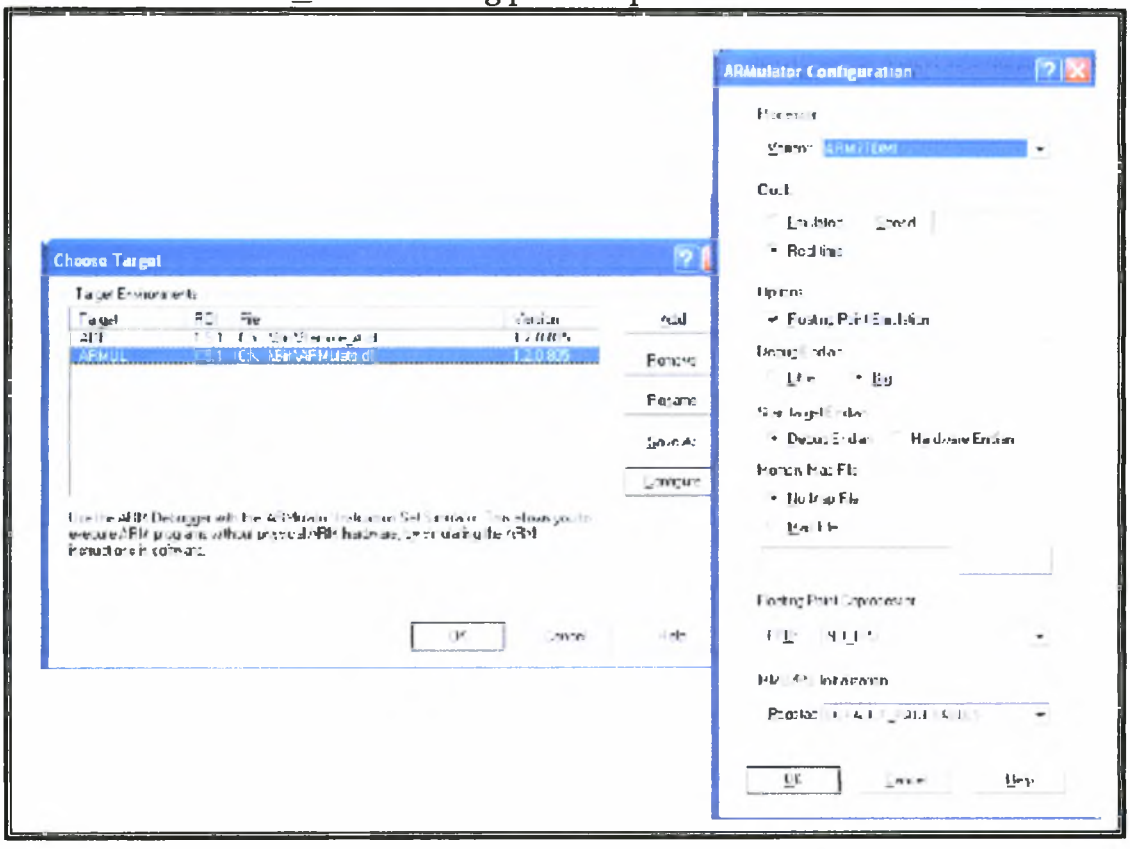
- File->New
- Επιλέξτε το πεδίο Project
- Location-> Επιλέξτε το location του καινούργιου project
- Όνομα-> Επιλέξτε ένα όνομα για το project
- Add Files to project: Επιλέξτε project-> Add Files

Συνεχίζουμε με το να εισάγουμε τις απαραίτητες ρυθμίσεις. Επομένως, έχουμε ότι :

- Κατάλληλες ρυθμίσεις για την χρήση του Big Endian τρόπου αποθήκευσης: Επιλέξτε Edit-> DebugRel Settings-> Language Settings
- Για τους ARM Assembler, C Compiler, C++ Compiler, Thumb C Compiler και Thumb C++ Compiler επιλέξτε:
  - Architecture or Processor: ARM7TDMI
  - Floating Point: Pure-endian softfp
  - Byte Order: Big Endian

Όταν είμαστε έτοιμοι να προχωρήσουμε στην μεταγλώττιση του κώδικα, επιλέγουμε:

- Project-> Debug (or press F5)
- Εμφανίζεται το AXD παράθυρο. Επιλέξτε Options->Configure Target
  - Επιλέξτε ARMUL Target (ARM emulator)
    - Click configure- σιγουρευτείτε ότι τα ισχύουν τα παρακάτω(βλέπε και την φωτογραφία):
      - Variant: ARM7TDMI
      - Clock: Real – Time
      - Floating Point Emulation is checked
      - Debug Endian is on Big
      - Also No Map File
      - NO_FPU Floating point Coprocessor



Εάν θέλουμε να παράγουμε ένα αρχείο με την συμπεριφορά του συστήματος, ακολουθούμε τα παρακάτω βήματα:

- Στον AXD Debugger, πηγαίνετε στην επιλογή File-> Load Image – στην λίστα που εμφανίζεται επιλέγουμε το δικό μας AXF αρχείο, τσεκάρουμε την επιλογή Profile και επιλέγουμε το Call graph profiling
- Επιλέξτε Options-> Profiling-> Toggle Profiling
- Πηγαίνετε στην επιλογή Execute-> Run (or F5), το εκτελούμε αυτό δυο φορές για να τρέξουμε το πρόγραμμά μας

Στην συνέχεια, έχουμε:

- Επιλέξτε Options->Profiling->Toggle Profiling
- Στο AXD Debugger παράθυρο, επιλέγουμε Options-> Profiling-> Write to File

- Δίνουμε ένα όνομα και αποθηκεύουμε το .prf αρχείο σε μια τοποθεσία
- Ανοίγουμε μια γραμμή εντολών και πληκτρολογούμε "armprof <filename.prf><profilingReport.txt>"

Το αρχείο που θα δημιουργηθεί από την εκτέλεση του προγράμματος, μοιάζει κάπως έτσι:

```
Name                    cum%    self%   desc%   calls
main                    99.4%   0.16%   95.83%  0
   qsort                        0.44%   0.75%   _
   _printf                      0.00%   0.00%   3
   clock                        0.00%   0.00%   6
   _sprintf                     0.34%   3.56%   1000
   randomise                    0.12%   0.69%   _
   hell_sort                    1.59%   3.43%   _
   insert_sort                  19.91%  59.44%  _
-------------------------------------------------------
   main                         19.91%  59.44%  _
insert_sort             79.35%  19.91%  59.44%  _
   strcmp                       59.44%  0.00%   243432
-------------------------------------------------------
   qs_string_compare            3.17%   0.00%   13021
   shell_sort                   3.43%   0.00%   14059
   insert_sort                  59.44%  0.00%   243432
strcmp                  60.05%  60.05%  0.00%   270512
```

Τέλος, εάν θέλουμε να βρούμε τον συνολικό χρόνο εκτέλεσης του προγράμματος, ακολουθούμε τα παρακάτω βήματα:

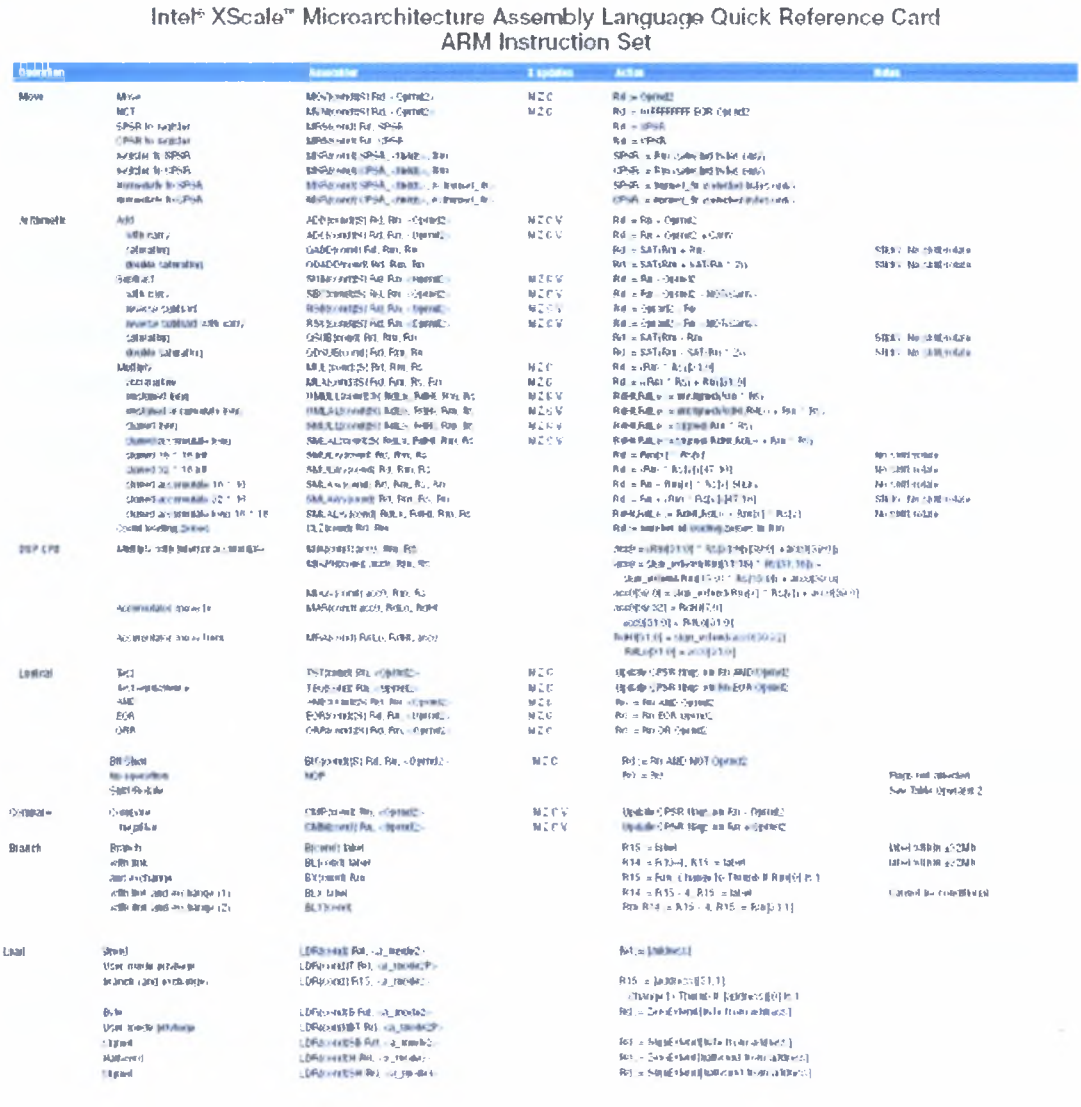
- Στο AXD παράθυρο, επιλέξτε System Views-> Debugger Internals
- Επιλέξτε Statistics
- Ο αριθμός που χαρακτηρίζεται ως "Total" σε αυτό το παράθυρο, είναι ο συνολικός χρόνος εκτέλεσης του προγράμματός μας

Παράρτημα Β

Στο παρόν παράρτημα, θα δούμε τις βασικότερες εντολές σε assembly, μερικές από τις οποίες χρησιμοποιήσαμε και στην πρώτη εφαρμογή κώδικα που γράψαμε (βλέπε Παράρτημα Γ).

Assembly Εντολές

Παρακάτω παραθέτουμε τις εντολές assembly του επεξεργαστή XScale της Intel.



Intel® XScale™ Microarchitecture Assembly Language Quick Reference Card
ARM Instruction Set

Load Multiple	Pop, or Block data load	LDM{mode} n{!}, <reglist>	Load list of registers from [Rn]	
	return and exchange	LDM{mode} n{!}, <reglist>	Load registers, R15 = [value+4][31:1]	
	and set the CPSR	LDM{mode} n{!}, <reglist>^	Load registers, and also the SPSR	Exception mode only.
	User mode registers	LDM{mode} n, <reglist>^	Load list of User mode registers from [Rn]	Privileged mode only.
Load Double		LDRD		
Store	short	STR{cond} Rd, <a_mode2>	[address] = R	
	User mode privilege	STR{cond}T Rd, <a_mode2P>	[address] = Rd	
	byte	STR{cond}B Rd, <a_mode>	[address][7:0] = Rd[7:0]	
	User mode privilege	STR{cond}BT Rd, <a_mode2P>	[address][7:0] = Rd[7:0]	
	Halfword	STR{cond}H Rd, <a_mode3>	[address][15:0] = Rd[15:0]	
Store Multiple	Push, or Block data store	STM{mode} n{!}, <reglist>	Store list of registers to [Rn]	
	User mode registers	STM{mode} n, <reglist>^	Store list of User mode registers to [Rn]	Privileged mode only.
Store Double		STRD		
Swap	short	SWP{cond} Rd, Rm, [Rn]	temp = [Rn]; [Rn] = Rm; Rd = temp	
	byte	SWP{cond}B Rd, Rm, [Rn]	temp = ZeroExtend([Rn][7:0]), [Rn][7:0] = Rm[7:0]; Rd = temp	
Coprocessor	Move to ARM register from coproc	MRC{cond} {p<pnum>,} <cp#>, Rd, CRn, CRm, <op2>		
		MRC2{cond} {p<pnum>,} <cp#>, <Rd>, <Rn>, <CRm>		
	Move to coproc from ARM reg	MCR{cond} {p<pnum>,} <cp#>, Rd, CRn, CRm, <op2>		
		MCR2{cond} {p<pnum>,} <cp#>, <Rd>, <Rn>, <CRm>		
	Load coproc reg	LDC{cond} p<pnum>, CRd, <a_mode5>	CRd = [address]	
	Store coproc reg	STC{cond} p<pnum>, CRd, <a_mode5>	[address] = CRd	
Software Interrupt		SWI{cond} {owned_24}	Software interrupt processor exception	24-bit value.
Breakpoint		BKPT <immed_16>	Prefetch abort or debug trap	Caught by debugger.
Pre-load		PLD <a_mode2>	Pre-load cache line	

Move	immediate	MOV{cond} Rd, #<immed_8r>	Rd = immed_8r	8-bit immediate value.
	Lo to Lo	MOV{S} Rd, Rm	Rd = Rm	
	Hi to Lo, Lo to Hi, Hi to Hi	MOV{cond} Rd, Rm	Rd = Rm	Not Lo to Lo
Arithmetic	Add	ADD{cond} Rd, Rn, #<immed_8r>	Rd = Rn + immed_8r	8-bit immediate value.
	Lo and Lo	ADD{S} Rd, Rn, Rm	Rd = Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi	ADD{cond} Rd, Rm	Rd = Rd + Rm	Not Lo to Lo
	immediate	ADD{S} Rd, Rn, #<immed_3>	Rd = Rn + immed_3	3-bit immediate value.
	with carry	ADC{S} Rd, Rn	Rd = Rd + Rn + C-bit	
	value to SP	ADD{cond} SP, SP, #<immed_7*4>	SP = SP + immed_7*4	9-bit immediate value (word aligned).
	value added to SP	ADD{cond} Rd, SP, #<immed_8*4>	Rd = SP + immed_8*4	10-bit immediate value (word aligned).
	value added to PC	ADD{cond} Rd, PC, #<immed_8*4>	Rd = (PC AND 0xFFFFFFFC) + immed_8*4	10-bit immediate value (word aligned).
Subtract		SUB{cond} Rd, Rn, Rm	Rd = Rn - Rm	
	immediate	SUB{S} Rd, Rn, #<immed_3>	Rd = Rn - immed_3	3-bit immediate value.
	immediate 8	SUB{S} Rd, #<immed_8>	Rd = Rd - immed_8	8-bit immediate value.
	with carry	SBC{S} Rd, Rm	Rd = Rd - Rm - NOT C-bit	
	value from SP	SUB SP, #<immed_7*4>	SP = SP - immed_7*4	9-bit immediate value (word aligned).
Negate		NEG{S} Rd, Rm	Rd = - Rm	
Multiply		MUL{S} Rd, Rm	Rd = Rm * Rd	
Compare		CMP Rn, Rm	update CPSR flags on Rn - Rm	Can be Lo-Lo, Lo-Hi, Hi-Lo or Hi-Hi
	negative	CMN Rn, Rm	update CPSR flags on Rn + Rm	
	immediate	CMP Rn, #<immed_8>	update CPSR flags on Rn - immed_8	8-bit immediate value.
	No operation	NOP	Rd = Rd	May not assembled.
Logical	AND	AND{S} Rd, Rm	Rd = Rd AND Rm	
	Exclusive OR	EOR{S} Rd, Rm	Rd = Rd EOR Rm	
	OR	ORR{S} Rd, Rm	Rd = Rd OR Rm	
	Bit Clear	BIC{S} Rd, Rm	Rd = Rd AND NOT Rm	
	Move NOT	MVN{S} Rd, Rm	Rd = NOT Rm	
	Test bits	TST Rn, Rm	update CPSR flags on Rn AND Rm	
Shift / Rotate	Logical shift left	LSL Rd, Rm, #<immed_5>	Rd = Rm << immed_5	5-bit immediate shift. Allowed shifts 0-31
	LSL Rd, Rs	Rd = Rd << Rs		
	Logical shift right	LSR Rd, Rm, #<immed_5>	Rd = Rm >> immed_5	5-bit immediate shift. Allowed shifts 1-32
	LSR Rd, Rs	Rd = Rd >> Rs		
	Arithmetic shift right	ASR Rd, Rm, #<immed_5>	Rd = Rm ASR immed_5	5-bit immediate shift. Allowed shifts 1-32
	ASR Rd, Rs	Rd = Rd ASR Rs		
	Rotate right	ROR Rd, Rs	Rd = Rd ROR Rs	

Branch	Conditional branch	B{cond} label See Table: Condition Field (ARM Ref.) All conditions	R15 = label	label must be within -256 to +256 bytes
	Unconditional branch	B label	R15 = label	label must be within ±2K bytes of current instruction
	Long branch with link	BL label label must be within 4M bytes of current instruction	R14 = R15 - 2, R15 = label	Branch or call function.
	Branch and exchange	BX Rm	R15 = Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0
	Branch with link and exchange	BLX label	R14 = R15 - 2, R15 = label	Branch or call function and change to ARM state.
	Change to ARM	label must be within 4MB of current instruction		
	Branch with link and exchange	BLX Rm	R14 = R15 - 2, R15 = Rm AND 0xFFFFFFFE	
	Change to ARM if Rm[0] = 0			

Software Interrupt		<immed_24>	Software interrupt processor exception	8-bit immediate value ignored by hardware.
Breakpoint		BKPT <immed_16>	Prefetch abort or debug trap	
Load	short immediate offset, word	LDR Rd, [Rn, #<immed_5*4>]	Rd = [Rn + immed_5*4]	
	negative	LDR{cond} Rd, [Rn, #-<immed_5*4>]	Rd = [Rn - immed_5*4]	Check bits 31:16
	byte	LDR{cond} Rd, [Rn, #<immed_5>]	Rd = ZeroExtend([Rn + immed_5][7:0])	Check bits 31:8
	short register offset, word	LDR Rd, [Rn, Rm]	Rd = [Rn + Rm]	
	negative	LDR{cond} Rd, [Rn, -Rm]	Rd = [Rn - Rm]	Check bits 31:16
	signed halfword	LDR{cond} Rd, [Rn, Rm]	Rd = SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte	LDR{cond} Rd, [Rn, Rm]	Rd = ZeroExtend([Rn + Rm][7:0])	Check bits 31:8
	signed byte	LDR{cond} Rd, [Rn, Rm]	Rd = SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
PC-relative		LDR Rd, [PC, #<immed_8*4>]	Rd = [PC AND 0xFFFFFFFC + immed_8*4]	
SP-relative		LDR Rd, [SP, #<immed_8*4>]	Rd = [SP + immed_8*4]	
Multiple		LDM{cond} Rn{!}, <reglist>	Load list of registers	Address increases, base included.

Store	short immediate offset, word	STR Rd, [Rn, #<immed_5*4>]	[Rn + immed_5*4] = Rd	
	negative	STR{cond} Rd, [Rn, #-<immed_5*4>]	[Rn - immed_5*4][15:0] = Rd[15:0]	Updates Rd[31:16]
	byte	STR{cond}B Rd, [Rn, #<immed_5>]	[Rn + immed_5][7:0] = Rd[7:0]	Updates Rd[31:8]
	short register offset, word	STR Rd, [Rn, Rm]	[Rn + Rm] = Rd	
	halfword	STR{cond}H Rd, [Rn, Rm]	[Rn + Rm][15:0] = Rd[15:0]	Updates Rd[31:16]
	byte	STR{cond}B Rd, [Rn, Rm]	[Rn + Rm][7:0] = Rd[7:0]	Updates Rd[31:8]
SP-relative, word		STR Rd, [SP, #<immed_8*4>]	[SP + immed_8*4] = Rd	
Multiple		STM{cond} Rn{!}, <reglist>	Store list of registers	Address increases, base included.
Push	Push	PUSH <reglist>	Push registers onto stack	Full descending stack.
	Push LR and register	PUSH <reglist>, LR	Push LR and registers onto stack	
Pop	Pop	POP <reglist>	Pop registers from stack	
	Pop and return	POP <reglist>, PC	Pop registers, return by address loaded to PC	
	Pop and return with exchange	POP <reglist>, PC	Pop values, and change to ARM state if bit[0] = 0	

Intel® XScale™ Microarchitecture Assembly Language Quick Reference Card
Key to Tables

CODES

{cond}	Refer to Table Condition Field {cond}
<Operand2>	Refer to Table Operand 2
{type}	Refer to Table PSR Fields
{S}	Update condition flags if S is present
<Rn>	Store, load or add/subtract the base register, used and updated using MWB and MSR
<a>	8 consecutive bits (bits [15:0] or T bracketing) [41:16]
<memory_list>	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits
<memory_list>	A 16-bit constant, formed by left-shifting an 8-bit value by four bits
<a_mode2>	Refer to Table Addressing Mode 2
<a_mode2P>	Refer to Table Addressing Mode 2 (Post-indexed only)
<a_mode3>	Refer to Table Addressing Mode 3
<a_mode4L>	Refer to Table Addressing Mode 4 (Block load or stack pop)
<a_mode4S>	Refer to Table Addressing Mode 4 (Block store or stack push)
<a_mode5>	Refer to Table Addressing Mode 5
<reglist>	A list of registers, enclosed in braces ({})
{T}	Optional; force register offset calculation in T privilege

CONDITION FIELD {cond}

Mnemonic	Description
EQ	Equal
NE	Not equal
CS / HS	Carry Set / Unsigned higher or same
CC / LO	Carry Clear / Unsigned lower
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (normally omitted)

OPERAND 2

Immediate value	#<immed_8r>	
Logical shift left immediate	Form: LSL #<immed_5>	Allowed shift: 0-31
Logical shift right immediate	Form: LSR #<immed_5>	Allowed shift: 1-32
Arithmetic shift right immediate	Form: ASR #<immed_5>	Allowed shift: 1-32
Rotate right immediate	Form: ROR #<immed_5>	Allowed shift: 1-31
Register	Form: <Rm>	
Rotate right extended	Form: RRX <Rm>	
Logical shift left register	Form: LSL <Rs>	
Logical shift right register	Form: LSR <Rs>	
Arithmetic shift right register	Form: ASR <Rs>	
Rotate right register	Form: ROR <Rs>	

PSR FIELDS - USE AT LEAST ONE SUFFIX

Suffix	Meaning	
c	Control field mask byte	PSR[7:0]
f	Flags field mask byte	PSR[31:24]
s	Status field mask byte	PSR[23:16]
x	Extension field mask byte	PSR[15:8]

ARM ADDRESSING MODES

ADDRESSING MODE 2 - WORD AND UNSIGNED BYTE DATA TRANSFER

Pre-indexed	Immediate offset	[<Rn>, #<±immed_12>]{!}	
	Zero offset	[<Rn>]	Equivalent to [<Rn>, #0]
	Register offset	[<Rn>, <±Rm>]{!}	
	Scaled register offset	[<Rn>, <±Rm>, LSL #<immed_5>]{!}	Allowed shift: 0-31
		[<Rn>, <±Rm>, LSR #<immed_5>]{!}	Allowed shift: 1-32
		[<Rn>, <±Rm>, ASR #<immed_5>]{!}	Allowed shift: 1-32
		[<Rn>, <±Rm>, ROR #<immed_5>]{!}	Allowed shift: 1-31
		[<Rn>, <±Rm>, RRX]{!}	
Post-indexed	Immediate offset	[<Rn>], #<±immed_12>	
	Register offset	[<Rn>], <±Rm>	
	Scaled register offset	[<Rn>], <±Rm>, LSL #<immed_5>	Allowed shift: 0-31
		[<Rn>], <±Rm>, LSR #<immed_5>	Allowed shift: 1-32
		[<Rn>], <±Rm>, ASR #<immed_5>	Allowed shift: 1-32
		[<Rn>], <±Rm>, ROR #<immed_5>	Allowed shift: 1-31
		[<Rn>], <±Rm>, RRX	

ADDRESSING MODE 3 - POST-INDEXED ONLY:

Post-indexed	Immediate offset	[<Rn>], #<±immed_12>	
	Zero offset	[<Rn>]	Equivalent to [<Rn>, #0]
	Register offset	[<Rn>], <±Rm>	
	Scaled register offset	[<Rn>], <±Rm, LSL #<immed_5>	Allowed shift: 0-31
		[<Rn>], <±Rm, LSR #<immed_5>	Allowed shift: 1-32
		[<Rn>], <±Rm, ASR #<immed_5>	Allowed shift: 1-32
		[<Rn>], <±Rm, ROR #<immed_5>	Allowed shift: 1-31
		[<Rn>], <±Rm, RRX>	

ADDRESSING MODE 3 - HALFWORD AND SIGNED BYTE DATA TRANSFER

Pre-indexed	Immediate offset	[<Rn>, #<±immed_8>]{!}	
	Zero offset	[<Rn>]	Equivalent to [<Rn>, #0]
	Register	[<Rn>, <±Rm>]{!}	
Post-indexed	Immediate offset	[<Rn>], #<±immed_8>	
	Register	[<Rn>], <±Rm>	

ADDRESSING MODE 5 - COPROCESSOR DATA TRANSFER

Pre-indexed	Immediate offset	[<Rn>, #<±immed_8*4>]{!}	
	Zero offset	[<Rn>]	Equivalent to [<Rn>, #0]
Post-indexed	Immediate offset	[<Rn>], #<±immed_8*4>	
Unindexed	No offset	[<Rn>, <8-bit coproc. option>]	

ADDRESSING MODE 4 - MULTIPLE DATA TRANSFER

Block load	
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before
Block store	
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before
Stack pop	
FD	Full Descending
ED	Empty Descending
FA	Full Ascending
EA	Empty Ascending
Stack push	
EA	Empty Ascending
FA	Full Ascending
ED	Empty Descending
FD	Full Descending

Παράρτημα Γ

Μετά από τις επεξηγηματικές περιγραφές που αφορούσαν τον τρόπο λειτουργίας και τις βασικές αρχές λειτουργίας των ενσωματωμένων συστημάτων και πιο συγκεκριμένα της πλατφόρμας X-board, είμαστε σε θέση να παραθέσουμε το κώδικα των δύο εφαρμογών που αναπτύχθηκαν και εκτελέστηκαν στην συγκεκριμένη πλατφόρμα.

Εφαρμογή 1η

Η πρώτη εφαρμογή αφορούσε την δημιουργία των «δικών μας» SWIs (Software Handlers). Με τον όρο «δικών μας» αναφερόμαστε στο γεγονός, πως τέτοιου είδους διαχειριστές διακοπών, έχουν υλοποιηθεί από της βιβλιοθήκες του επεξεργαστή ARM, και από την ομάδα ανάπτυξης της πλατφόρμας X-board. Έχοντας μελετήσει και κατανοήσει όλα τα παραπάνω που αναφέρονται στην εργασία, θελήσαμε να δημιουργήσουμε μόνοι μας τέτοια SWIs. Ο κώδικας της εφαρμογής που αναπτύξαμε ακολουθεί.

Πριν όμως από αυτό, θα πρέπει να αναφερθεί το γεγονός πως η υλοποίηση και η εκτέλεση της εφαρμογής των SWIs, έγινε στον ARMulator, και όχι πάνω στην πλατφόρμα ανάπτυξης του X-board. Η λειτουργικότητα που προσφέρει η εφαρμογή είναι η ίδια, με την διαφορά ότι εκτελείται στον υπολογιστή, host computer, και όχι στον ίδιο τον επεξεργαστή ARM, αλλά στο πρόγραμμα προσομοίωσης της λειτουργίας του. Από την εφαρμογή η οποία εκτελείται σε user mode, παράγονται SWIs, τα οποία δημιουργούν εξαίρεση στην κανονική ροή εκτέλεσης του προγράμματος στον επεξεργαστή, τότε αναλαμβάνει την διαχείριση της εξαίρεσης ο handler, την διαχειρίζεται και μετά το πέρας της επιτυχούς διαχείρισης της, επιστρέφει στην κανονική εκτέλεση του προγράμματος και στις εντολές που έπονται της εντολής SWI.

Κώδικας Εφαρμογής 1ης (μέρος α)

Dispatcher.c

```
#include <stdlib.h>
#include <stdio.h>
#include "userSWIs.h"

//The below correspondence between processor register numbers and simple numbers is from the ATPCS-
// - ARM-THUMB Procedure Call Standard.
//We use those numbers in SWI_Dispatcher in order to show register r0 with the pointer ptr[0].

/* Processor Register Numbers
   {"r0", 0},   {"r1", 1},    {"r2", 2},    {"r3", 3},
   {"r4", 4},   {"r5", 5},    {"r6", 6},    {"r7", 7},
   {"r8", 8},   {"r9", 9},    {"r10", 10},   {"r11", 11},
   {"r12", 12},  {"r13", REG_SP},{"r14", REG_LR},{"r15", REG_PC} */

extern unsigned int Ang_Handler;
extern unsigned int installSWI_Handler(unsigned int);

//Below we define the functions that are called by the SWI_Dispatcher to handle each SWI

void Handle_printC(char c) //Handle SWI 0x100 requests
{
        SWI_WriteC(c);
}

char Handle_getC(void) //Handle SWI 0x101 requests
{
        return SWI_ReadC();
}

void Handle_printString(char * str) //Handle SWI 0x102 requests
{
        SWI_Write0(str);
}


  void Handle_restore(void) //Handle SWI 0x103 requests
  {
          Ang_Handler=installSWI_Handler(Ang_Handler);
  }

  unsigned int Handle_getClock(void) //Handle SWI 0x104 requests
  {
          return SWI_Clock();
  }
```

```
unsigned int SWI_Dispatcher(unsigned int SWI_num, unsigned int * ptr)
{
        int status = 0; //Flag in order to know whether the SWI request was handled by our SWI_Handler
                                                //or by the Angel_Handler

        switch(SWI_num) //The SWI_num shows the code of the SWI_instruction
        {
                case (0x100):
                status = 1;
                Handle_printC(ptr[0]);
                break;

                case (0x101):
                status = 1;
                ptr[0] = Handle_getC();
                break;

                case (0x102):
                status = 1;
                Handle_printString((char *)ptr[0]);
                break;

                case (0x103):
                status = 1;
                Handle_restore();
                break;

                case (0x104):
                status = 1;
                ptr[0] = Handle_getClock();
                break;

                default:
                status = 0;
                break;
        }

        return status;
}
```

Installer.c

```
#include <stdio.h>
#include <stdlib.h>  //Declares functions for number alterations,
memory dirtibution, etc.
#include "userSWIs.h"

unsigned int Ang_Handler; //The declaration of the original(old)
SWI_handler which is the Angel SWI_Handler

extern SWIhandlerEntry(void); //A reference to the entry routine

unsigned int installSWI_Handler(unsigned int new_handler) //Our
installer which is changing the handleradress which is
                                                //stored at softvec to
the address of our new handler
{

      unsigned int offset,oldvec;
      unsigned int *SWIvec; //A pointer to the address 0x08 of the
Vector Table
      unsigned int *softvec; //The current handleraddress

      SWIvec = (unsigned *) 0x08;
      offset = (*SWIvec) & 0xfff; //We find the used offset whisch is
stored in the last 12 bits of the SWIvec

      if( offset & 0xFFFFF000 ) //We check if the offset can be
represented using 12 bits
      {                          //else we send a request to print
that the installation has failed
            SWI_Write0("Installation of handler failed");
            exit(0);
      }

      softvec = (unsigned *) (offset + 0x10); //We are finding the
address of the current SWI handler(Angel) by adding

      //the offset we found,the 0x08 address of the SWI vector and
the 8 bits that

      //the pc is ahead due to the pipelining during the execution
time
      oldvec = *softvec;                        //We are storing the
address of the old SWI handler in order to use it if

      //our SWI handler can't handle the SWI request
      *softvec = new_handler;                           //We are
setting the handleraddress to the address of our SWI handler

      return(oldvec);

}

void install_user_SWIs(void){     //A wrapper function around
installSWI_Handler to install the handler
      Ang_Handler=installSWI_Handler((int)SWIhandlerEntry);
}
```

userSWIS.h

```
#ifndef USER_SWIS_H_INCLUDED
#define USER_SWIS_H_INCLUDED

//The following SWI_numbers will be handled by our SWI_handler

void          __swi(0x100) SWI_printC(char);
char          __swi(0x101) SWI_getC(void);
void          __swi(0x102) SWI_printString(char *);
void          __swi(0x103) SWI_restore(void);
unsigned int  __swi(0x104) SWI_getClock(void);
int           __swi(0x105) SWI_findInt(void);
void          __swi(0x106) SWI_writeInt(int);
void          __swi(0x107) SWI_CopyFile(char*);
unsigned int  __swi(0x108) SWI_loadAXF(char *);
void          __swi(0x109) SWI_run(unsigned int);
void          __swi(0x200) SWI_CopyMem(unsigned int, unsigned int);

//The following SWI_numbers will be handled by Angel.

void          __swi(0x000) SWI_WriteC(char);
void          __swi(0x002) SWI_Write0(char *);
char          __swi(0x004) SWI_ReadC(void);
unsigned int  __swi(0x61) SWI_Clock(void);

void install_user_SWIs(void);

#endif /* USER_SWIS_H_INCLUDED */
```

```
swi_handler_entry.s
AREA HandlerEntry, CODE, READONLY
        EXPORT SWIhandlerEntry ;Make it visible outside of this file
        IMPORT SWI_Dispatcher  ;Make it visible inside this file
        IMPORT Ang_Handler

SWIhandlerEntry

        SUB   sp,sp,#4          ;We are subtracking 4 from sp in order to leave
room for storing SPSR for supporting nested
                                                ;SWI calls
        STMFD       sp!, {r0-r12,lr} ;We are storing gp registers and the link
register lr in the stack
        MRS         r2,spsr     ;We are moving the SPSR into the gp r2
        STR   r2, [sp, #14*4]  ;We are storing the SPSR above the gp registers
through the r2 register
        MOV         r1,sp                   ;We are setting a pointer on the
parameters of the stack through register r1
        LDR   r0, [lr, #-4]   ;We are extracting the SWI number through the
link register and store it in the register r0
        BIC         r0,r0,#0xff000000;We are getting the code of the SWI by
bit-masking
        BL          SWI_Dispatcher  ;We are going to the body of our SWI
handler
        CMP   r0,#0             ;We are checking whether for the handling of
the SWI request we are using our SWI handler or
                                                ;the old SWI handler(Angel)
        LDR         r2,[sp,#14*4]       ;We are restoring the SPSR from the
sp
        MSR         spsr_cf,r2
;http://www.arm.com/support/faqdev/1332.html
                                                ;Strictly speaking, the '_cxsf'
form should be instead of '_cf' in these cases to ensure
                                                ;compatibility with future
ARM cores which may make use of the x and s fields (unused by all current
ARM processors).

        MSREQ       cpsr_cf,r2          ;If the SWI request is to be handled
from the old SWI handler we are transferring the value of the
                                                ;register r2(SPSR) to the cpsr
        LDMFD       sp!, {r0-r12,lr} ;We are restoring all the gp registers and
the lr into their original values
        ADD         sp, sp, #4      ;We are removing the space that was used
for storing SPSR
        LDREQ       pc,Ang_Handler  ;If the SWI request is to be handled by
the old SWI handler,we are setting the pc to the
                                                ;old SWI handler(Angel)
        MOVS pc,lr          ;We are returning from the handler


        END
```

Κώδικας Εφαρμογής 1ης (μέρος β)

Dispatcher.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h> //Library for the functions memcpy,memset etc
#include <ctype.h>
#include "userSWIs.h"

                    //Below we define the max/min boundaries for an
integer.A signed integer is a 31 bit(1 bit for the signing)
#define max 2147483647   //Max integer
#define min -2147483647  //Min integer
#define EI_NIDENT 16

//The below correspondence between processor register numbers and
simple numbers is from the ATPCS-
// - ARM-THUMB Procedure Call Standard.
//We use those numbers in SWI_Dispatcher in order to show register r0
with the pointer ptr[0].

/* Processor Register Numbers
    {"r0", 0},    {"r1", 1},      {"r2", 2},      {"r3", 3},
    {"r4", 4},    {"r5", 5},      {"r6", 6},      {"r7", 7},
    {"r8", 8},    {"r9", 9},      {"r10", 10},    {"r11", 11},
    {"r12", 12},  {"r13", REG_SP},{"r14", REG_LR},{"r15", REG_PC} */


typedef struct //Format of the ELF header structure
{
      unsigned char e_ident[EI_NIDENT]; //File info(object file or
not)
      unsigned short e_type;            //Type of
file(relocatable,executable,etc)
      unsigned short e_machine;         //Target processor(Intel
x86,ARM,SPARC, etc)
      unsigned int e_version;           //Version # (to allow for
future versions of ELF)
      unsigned int e_entry;             //Program entry point(0 if no
entry point)
      unsigned int e_phoff;             //offset of program header(in
bytes)
      unsigned int e_shoff;             //offset of section header
table
      unsigned int e_flags;             //Processor-specific flags
      unsigned short e_ehsize;          //ELF header's size
      unsigned short e_phentsize;       //Entry size in pgm header
tbl
      unsigned short e_phnum;           //# of entries in pgm header
      unsigned short e_shentsize;       //Entry size in sec header
tbl
      unsigned short e_shnum;           //# of entries in sec header
tbl
      unsigned short e_shstrndx;        //sec header tbl index of str
tbl
} Elf32_Ehdr;

typedef struct //Program header entry for each segment
{
```

```
      unsigned int p_type;              //Type of segment-loadable,
dll,...
      unsigned int p_offset;            //offset in bytes from the
start of file
      unsigned int p_vaddr;             //Virtual address in memory of
segment
      unsigned int p_paddr;             //Physical address in memory
of segment
      unsigned int p_filesz;            //Number of bytes in the file
of the segment
      unsigned int p_memsz;             //Number of bytes in memory of
the process
      unsigned int p_flags;             //Indicates whether segment is
executable
      unsigned int p_align;             //Alignment information
} Elf32_Phdr;


extern unsigned int Ang_Handler;                          //Address of
the Angel handler(old handler)
extern unsigned int installSWI_Handler(unsigned int); //We create a
connection to the installSWI_Handler
extern SWIhandlerEntry;                                   //Assembly
entry routine

//Below we define the functions that are called by the SWI_Dispatcher
to handle each SWI

void Handle_printC(char c) //Handle SWI 0x100 requests
{
      SWI_WriteC(c);
}

char Handle_getC(void) //Handle SWI 0x101 requests
{
      return SWI_ReadC();
}

void Handle_printString(char * str) //Handle SWI 0x102 requests
{
      SWI_Write0(str);
}

void Handle_restore(void) //Handle SWI 0x103 requests
{
      Ang_Handler=installSWI_Handler(Ang_Handler);
}

unsigned int Handle_getClock(void) //Handle SWI 0x104 requests
{
      return SWI_Clock();
}

int Handle_findInt(void) //Handle SWI 0x105 requests
{
      char tmp[20],ar[20];
      int counter=0,ct=0,i = 0,result=0;
      tmp[counter] = SWI_ReadC();
//Read the first character from the user
```

```
      while( tmp[counter] != (char) 10)  //While the incoming
character is different from ENTER(the ASCII encoding for it is 10)
      {
            counter = counter + 1;
            if(isspace(tmp[counter-1])) { //We don't take into
account the spaces(if the user gives any)
                  tmp[counter-1] = NULL;
                  counter = counter - 1;
                  tmp[counter] = SWI_ReadC();
                  continue;
            }
            if(counter == 20) {          //We accept maximum of 20
characters from the user
                  SWI_printString("You overpassed string
boundaries");
                  break;
            }
            tmp[counter] = SWI_ReadC();  //We read the next character
from the user
      }
      if(tmp[counter] == (char) 10) { //If the incoming counter is
ENTER, stop
            tmp[counter] = NULL;
            counter = counter - 1;
      }
      while(ct<=counter)
      {
            if(tmp[ct]>(char)47 && tmp[ct]<(char)58) { //We are
locating the first integer('0' is 48 in ASCII encoding and '9' is 57)
                  if(ct>0 && tmp[ct-1] == (char) 45) {   //We are
checking if the previous character is '-'(signed integer)
                        ct = ct - 1;
                  }
                  for(i=0;i<=(counter - ct);i++) {      //We are
copying the next section(from the integer or the '-') of the input
string
                        ar[i] = tmp[i+ct];                //at a
table
                  }
                  break;
            }
            ct = ct + 1;
      }
      result = atoi(ar);                          //We are passing
this table to atoi
      if(result == max || result == (min-1)) {    //We are checking
if the integer exceeds the boundaries
            SWI_printString("You exceeded the max/min int
bounds.\n");
      }

      return atoi(ar);
}

void Handle_writeInt(int integ1) //handle SWI 0x106 requests
{
      int tmp_integ =0,counter=0,i=0,j=0;
      int number[10];
      for(i=0;i<10;i++) {
      number[i]=0;
      }
```

```
	if(abs(integ1)!=integ1) {                    //We are checking if
the integer is positive or negative
		SWI_WriteC('-');
		integ1 = abs(integ1);                  //If it is negative
we are taking the absolute value
	}

	while(integ1!=0) {
		tmp_integ=integ1%10;                   //We are finding the
least significant digit of our integer
		if(tmp_integ!=0) {
			integ1=(integ1-tmp_integ)/10;    //We are shifting
left in order to find the next digit
			number[counter]=tmp_integ;
		}
		else {
			integ1=(integ1)/10;
			number[counter]=0;
		}
		counter=counter+1;
	}

	if(counter==0) {                         //If the user gave the
number 0
		SWI_WriteC('0');
	}
	for(j=counter-1;j>=0;j--) {            //We are printing the
number the user gave(we are adding the 48-ASCII encoding for '0'-
		SWI_WriteC((char)(number[j]+48)); //in order to take the
right character
	}
	SWI_WriteC('\n');
}

void Handle_CopyFile(char * file_name) //Handle SWI 0x107 requests
{
	int handler_open;
	int handler_close;
	int file_length = 0;
	int bytes_not_read=0;
	char * char_buffer;

	if((handler_open = SWI_Open(file_name, 1)) == 0)         //We
are openning the reading file(input)
	{
		SWI_printString("Reading-File open, FAILED! \n");
		return;
	}
	else
	{
		if((handler_close = SWI_Open("out.txt", 5)) == 0)   //We
are openning the write file(output)
		{
			SWI_printString("Write-File open, FAILED! \n");
			return;
		}

		file_length = SWI_Flen(handler_open);               //We
are finding the length of the input file
```

```
            if(file_length == -1)
            {
                  SWI_printString("Reading File-Length, FAILED! \n");
            }
            else
            {
                  char_buffer = (char *)(malloc (file_length));
//Allocate temporary memory to copy the contents of the input file
                  bytes_not_read=SWI_Read(handler_open, char_buffer,
file_length);  //The bytes that were not read and copied to the

//memory from the input file
                  SWI_Write(handler_close, char_buffer, file_length-
bytes_not_read);//We are writting the contents into the output file
            }


            if(SWI_Close(handler_close) != 0)                  //We
are closing the output file
            {
                  SWI_printString("Closing Write-File, FAILED! \n");
            }

            if(SWI_Close(handler_open) != 0)                  //We are
closing the input file
            {
                  SWI_printString("Closing Read-File, FAILED! \n");
            }
      }

}

unsigned int Handle_loadAXF(char * AXF_Name) //Handle 0x108 requests
{
      int AXF_handler;
      int AXF_length;
      int i = 0;
      char * AXF_buffer;
      char * memory_buffer;
      unsigned int entry_address;

      Elf32_Ehdr ELF_header;
      Elf32_Phdr ELF_PrHeadTable;

      if((AXF_handler = SWI_Open(AXF_Name, 1)) == 0)            //We
are openning the AXF file
      {
            SWI_printString("AXF File Open, FAILED! \n");
            return 0;
      }
      else
      {
                  if((AXF_length = SWI_Flen(AXF_handler)) == -1)
//We are finding the length of the AXF file
                  {
                        SWI_printString("AXF File-Length,
FAILED!\n");
                        return 0;
                  }
```

```
                AXF_buffer = (char *)malloc(AXF_length);
//Allocate temporary memory for holding the contents of the AXF file
                SWI_Read(AXF_handler, AXF_buffer, AXF_length);
//We are loading the contents of the AXF file

                memcpy(&ELF_header,AXF_buffer,52);
//We are copying the first 52 bytes(header size) int the

//ELF header structure
                //Below we are checking if the file is an ELF
object file
                if((int)ELF_header.e_ident[0]!=0x7f ||
ELF_header.e_ident[1] != 'E' || ELF_header.e_ident[2] != 'L' ||
ELF_header.e_ident[3] != 'F')
                {
                      SWI_printString("This is NOT an ELF Object
File! \n");
                      return 0;
                }

                if((int)ELF_header.e_ident[5] != 2)
//Here we are checking the code encoding of all data(whether it is
                {
//Big Endian or Little Endian)
                      SWI_printString("Wrong Format! (It is NOT a
Big Endian!) \n");
                      return 0;
                }

                if((int)ELF_header.e_type != 2)               //We
are checking the identification of the object file type
                {
//(Relocatable,Executable or Shared object file)
                      SWI_printString("NOT an executable file!
\n");
                      return 0;
                }

                if((int)ELF_header.e_machine != 40)          //We
are checking the used architecture
                {
                      SWI_printString("NOT ARM target processor!
\n");
                      return 0;
                }

                entry_address = ELF_header.e_entry;         //We
are storing the virtual address to which the system first transfers
control

                if(entry_address == 0)
                {
                      SWI_printString("No Available Entry Point!
\n");
                      return 0;
                }

                if(entry_address < 0x12000)                 //We are
checking if the loading address is at a safe location
                {
```

```
                        SWI_printString("ERROR in Loading Address...!
\n");     //else we are printing an error message
                        return 0;
                  }

                  if(ELF_header.e_phoff == 0)               //We are
checking the program header table's file offset
                  {
                        SWI_printString("No Available Program Header
table! \n");
                        return 0;
                  }

                  AXF_buffer = AXF_buffer + ELF_header.e_phoff;
//We are moving to the program header table

                  for (i = 0; i < ELF_header.e_phnum; i++)
//We may have lot of entries
                  {
                        memcpy(&ELF_PrHeadTable, AXF_buffer,
ELF_header.e_phentsize);   //We are copying the program header table
into

//the ELF program header structure
                        if(ELF_PrHeadTable.p_type != 1)
//We are checking the type of the segment(loadable)
                        {
                              SWI_printString("NOT Loadable Segments!
\n");
                              return 0;
                        }

                        memory_buffer = (char
*)ELF_PrHeadTable.p_paddr; //We are setting a pointer to the physical
address of the memory

                               //where the segment will be written
                        memset
(memory_buffer,0,ELF_PrHeadTable.p_memsz);//We initialize with zero
all the memory blocks that will be used

                        if(SWI_Seek(AXF_handler,
ELF_PrHeadTable.p_offset) !=0 ) //We are seeking the beginning of the
segment
                        {
                              SWI_printString("Seeking File, FAILED!
\n");
                              return 0;
                        }

                        SWI_Read(AXF_handler, memory_buffer,
ELF_PrHeadTable.p_filesz); //We are loading the segment into the
memory
                     //Here while we load the segment into the
memory we see that (p_memsz-p_filesz) bytes are equal with 0
                        AXF_buffer=AXF_buffer+ELF_header.e_phentsize;
//We are moving to the next segment
                  }

      }
```

```
      if(SWI_Close(AXF_handler) != 0)
//We are closing the AXF file
            {
                  SWI_printString("Closing AXF File, FAILED! \n");
            }
      free(AXF_buffer);
//We are clearing the temporary memory space used
      return (unsigned int) entry_address;
//for the AXF file
}

unsigned int Handle_Run(unsigned int saddress) //Handle 0x109
requests
{
      if(saddress == 0)                                          //We are
checking whether the file(code) pointed by the saddress
      {                                                          //was
loaded or not
            SWI_printString("File didn't RUN! \n");
            exit(0);
      }
      return saddress;
}

void Handle_CopyMem(unsigned int address, unsigned int data) //Handle
0x200 requests
{
      unsigned int * temp_data;

      if( (address%4) !=0 )                                    //We are
checking if the address is word aligned
      {
            SWI_printString("MISALIGNED! \n");
            return;
      }
      else
      {
            temp_data = (unsigned int * ) address;           //Data
contents of the address
            SWI_printString("Data in that memory address:");
            SWI_writeInt((int )* temp_data);                 //We are
printing the data contents
            *temp_data = data;                               //We are
storing the new data contents in the address
            SWI_WriteC('\n');
      }
}



unsigned int SWI_Dispatcher(unsigned int SWI_num, unsigned int * ptr)
{
      int status = 0; //Flag in order to know whether the SWI request
was handled by our SWI_Handler
                        //or by the Angel_Handler

      switch(SWI_num) //The SWI_num shows the code of the
SWI_instruction
      {
            case (0x100):
            status = 1;
```

```
            Handle_printC(ptr[0]);
            break;

            case (0x101):
            status = 1;
            ptr[0] = Handle_getC();
            break;

            case (0x102):
            status = 1;
            Handle_printString((char *)ptr[0]);
            break;

            case (0x103):
            status = 1;
            Handle_restore();
            break;

            case (0x104):
            status = 1;
            ptr[0] = Handle_getClock();
            break;

            case (0x105):
            status = 1;
            ptr[0] = (unsigned int)Handle_findInt();
            break;

            case (0x106):
            status = 1;
            Handle_writeInt((int)ptr[0]);
            break;

            case (0x107):
            status = 1;
            Handle_CopyFile((char *)ptr[0]);
            break;

            case (0x108):
            status = 1;
            ptr[0] = Handle_loadAXF((char *)ptr[0]);
            break;

            case (0x109):
            status = 1;
            ptr[13] = Handle_Run(ptr[0]);                    //We
are storing the entry address of the code in the user link register
            break;
//which is in the stack

            case (0x200):
            status = 1;
            Handle_CopyMem((unsigned int)ptr[0], (unsigned
int)ptr[1]);
            break;

            default:
            status = 0;
            break;
    }
```

```
	return status;
}
```

Εφαρμογή 2η

Κώδικας Εφαρμογής 2ης

BufStatus.c

```
#include "BufStatus.h"
#include "uhal.h"

//Define output and input buffers
BufOutCharacteristics OutputBuffer;
BufInCharacteristics InputBuffer[MaxNumberOfInputBuffers];


//Function for intialization of the output buffer
void InitializingOutputBuffer(void) {
	OutputBuffer.head=0;
	OutputBuffer.tail=0;

	OutputBuffer.FreeSpace=OutputBufferLength-1;
	OutputBuffer.UsedSpace=0;

}


//Function for intialization of a specific input buffer
void InitializingInputBuffer(unsigned int InBufNumber) {
	InputBuffer[InBufNumber].head=0;
	InputBuffer[InBufNumber].tail=0;

	InputBuffer[InBufNumber].FreeSpace=InputBufferLength-1;
	InputBuffer[InBufNumber].UsedSpace=0;
}


void InitializationBothBuffers(void) {
	unsigned int i;

	for(i=0;i<MaxNumberOfInputBuffers;i++){
		InitializingInputBuffer(i);
	}

	InitializingOutputBuffer();
}

unsigned int getOutputBufferFreeSpace(void) {
	return(OutputBuffer.FreeSpace);
}

unsigned int getOutputBufferUsedSpace(void) {
	return(OutputBuffer.UsedSpace);
}

unsigned int getInputBufferFreeSpace(unsigned int InBufNumber) {
```

```
	return(InputBuffer[InBufNumber].FreeSpace);

}


// Function that returns used space in input buffer
unsigned int getInputBufferUsedSpace(unsigned int InBufNumber) {
	return(InputBuffer[InBufNumber].UsedSpace);
}

// Function that puts a character in output buffer
unsigned int PutCharacterInOutputBuffer(char Value) {
	int BufStatus=0;

	//Here first of all we are checking if there is free space for
writting a character in the output buffer
	if(OutputBuffer.FreeSpace!=0){
		//We are putting the character in the head of the buffer

		OutputBuffer.Data[OutputBuffer.tail]=Value;
		//We are shifting the head pointer 1 place in order to
show to the next empty entry

	OutputBuffer.tail=(OutputBuffer.tail+1)%OutputBufferLength;

		//We are setting the new values for the UsedSpace and
FreeSpace variables

		OutputBuffer.FreeSpace--;
		OutputBuffer.UsedSpace++;


		//We are returning 0(Successful placement)
		return(BufStatus);

	}
	else{
		BufStatus=1;
		//We are returning 1(Unsuccessful placement)
		return(BufStatus);

	}
}


// Function that gets a character from the output buffer
unsigned int PopCharacterFromOutputBuffer(char *CharValue) {
	int BufStatus=0;

	//Here first of all we are checking if there is a character for
poping from the output buffer
	if(OutputBuffer.UsedSpace!=0){
		//We are getting the character shown by the tail

		*CharValue=OutputBuffer.Data[OutputBuffer.head];
		//We are shifting the tail pointer 1 place in order to
show to the next character

	OutputBuffer.head=(OutputBuffer.head+1)%OutputBufferLength;
```

```
            //We are setting the new values for the UsedSpace and
FreeSpace variables

            OutputBuffer.FreeSpace++;
            OutputBuffer.UsedSpace--;


            //We are returning 0(Successful poping)
            return(BufStatus);

      }
      else{
            BufStatus=1;
            //We are returning 1(Unsuccessful poping)
            return(BufStatus);

      }
}



unsigned int PutCharacterInInputBuffer(unsigned int InBufNumber,char
Value) {
      int BufStatus=0;

      if(InputBuffer[InBufNumber].FreeSpace!=0){

      InputBuffer[InBufNumber].Data[InputBuffer[InBufNumber].tail]=Va
lue;

      InputBuffer[InBufNumber].tail=(InputBuffer[InBufNumber].tail+1)
%InputBufferLength;

            InputBuffer[InBufNumber].FreeSpace--;
            InputBuffer[InBufNumber].UsedSpace++;
            return(BufStatus);
      }
      else{
            BufStatus=1;
            return(BufStatus);
      }
}


unsigned int PopCharacterFromInputBuffer(unsigned int
InBufNumber,char *CharValue) {
      int BufStatus=0;

      if(InputBuffer[InBufNumber].UsedSpace != 0){
            *CharValue=
InputBuffer[InBufNumber].Data[InputBuffer[InBufNumber].head];

      InputBuffer[InBufNumber].head=(InputBuffer[InBufNumber].head+1)
%InputBufferLength;

            InputBuffer[InBufNumber].FreeSpace++;
            InputBuffer[InBufNumber].UsedSpace--;
            return(BufStatus);
      }
      else{
            BufStatus=1;
```

```
            return(BufStatus);

      }
}
```

Userio.c

```
#include "serial.h"                //We load the libraries of which
the functions we will use
#include "uhal.h"
#include "trongame.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "my_irq.h"
#include "timer.h"
#include "screen.h"
#include "BufStatus.h"
#include "userio.h"
#include <ctype.h>
#include "brh.h"

extern int InTimer;

//Global variable declaration
timer_p Timing;                    //Structure with the tenths,secs and
mins
volatile int TimerTenths=0;
int ticks;
int GlobalTimer;

unsigned int InputMode=0;

extern ST16C550Reg *Serial1;

//-------------------------------------------------------------------------
---------------------------
extern void uHALr_InitInterrupts(void);
extern void uHALr_InitTimers(void);
extern int  uHALr_RequestTimer(PrHandler,const unsigned char *);
extern int  uHALr_SetTimerInterval(unsigned int,unsigned int);
extern void uHALr_InstallTimer(unsigned int);
extern int  uHALr_FreeTimer(unsigned int);
extern void uHALir_InitSerial(unsigned int,unsigned int );
extern void uHALr_DisableInterrupt(unsigned int);
extern void uHALr_EnableInterrupt(unsigned int);
extern int uHALr_FreeInterrupt(unsigned int);


extern int NumPlayers;
extern int NumComputer;
//-------------------------------------------------------------------------
---------------------------

//Function that puts a string in the output buffer(with busy waiting
if the buffer is not empty enough)
void PrintStr(char *s) {
      int i=0;
      unsigned int tmp;
```

```
	int Length;

	BRHDisableInterrupt(1);	//Disable TimerInterrupts

	Length=strlen(s);		//We are finding the length of the
string

	//Wait until the transmitter holding register is empty
	while( (Serial1->ier & BIT1) != 0 );

	//If we can not put the string in the buffer...
	if( Length > getOutputBufferFreeSpace()) {
		Serial1->ier |= BIT1;	//Enable the THR interrupt
		while( (Serial1->ier & BIT1) != 0 );  //Wait until the
buffer is empty
	}

	//Put the string in the buffer
	while(s[i]!='\0') {
		tmp=PutCharacterInOutputBuffer(s[i]);
		i++;

	}

	BRHEnableInterrupt(1);		//Enable Timer Interrupts

	Serial1->ier |= BIT1;		//Enable THR interrupts

}

//Function that puts a string in the output buffer(without busy
waiting if the buffer is not empty enough)
void PrintStr_no_irq(char *s) {
	int i=0;
	char ch;

	int Length=strlen(s);

	//If we can not put the string in the buffer
	while( Length >= getOutputBufferFreeSpace() )  {
		PopCharacterFromOutputBuffer( &ch );  //Empty the buffer
until we can
		Serial1->rxtx = ch;
	}

	//Put the string in the buffer
	while(s[i]!='\0')  {
		PutCharacterInOutputBuffer(s[i]);
		i++;
	}
}

//Function that calculates the number of ticks
//It is used as the timer interrupt handler
void IncreaseTimer() {
	char String[32];

	TimerTenths++;

	//We are finding the tenths,secs and mins
	ticks=TimerTenths;
```

```
	Timing.tenths=ticks%10;

	ticks=ticks/10;
	Timing.secs=ticks%60;

	Timing.mins=ticks/60;

	// We are savinf the cureent condition,going the cursor at
	// the desirable position in terminal, putting the time in the
proper format,
	// and finally restore previous condition
    savecursor_no_irq();
	sprintf(String,"%02d:%02d.%d",Timing.mins,Timing.secs
,Timing.tenths);
	PrintStrAtLoc_no_irq( String, 70, 24 );
	restorecursor_no_irq();
}

// This function returns the char at the head of the input
buffer(with busy waiting)
char GetChar(int id) {
	char CharValue;

	//If the poping of the character is not successful do busy
waiting
	while( PopCharacterFromInputBuffer(id,&CharValue) ) {
	}

	return CharValue;
}

// This function returns the char at the head of the input
buffer(without busy waiting)
char GetCharNB(int id) {
	unsigned int tmp;
	char CharValue;

	//Getting the character from the buffer
	tmp=PopCharacterFromInputBuffer(id,&CharValue);


	//Check to see if there was a character to the buffer
	if(tmp==0) {
		return CharValue;  //If there is return the character...
	}
	else {
		return 0;          //Else return 0
	}
}

//Function for initializing the timer
void init_timer(void) {
	int tmp;
	uHALr_InitInterrupts();              //We initialize the µHAL
internal interrupt structures
	uHALr_InitTimers();                  //We reset all the timers
to a known state
	TimerTenths=0;
	uHALr_printf("InitTimer!\n");
	//We install the handler for the system timer and stop the
timer
```

```
	GlobalTimer=uHALr_RequestSystemTimer(IncreaseTimer,(const
unsigned char*)"test");
	if(GlobalTimer<=0) {
		uHALr_printf("Timer IRQ is already assigned!\n");
	}

	//We set the timer interval in microseconds
	tmp=uHALr_SetTimerInterval((unsigned int)GlobalTimer,100000);
	if(tmp<0) {
		uHALr_printf("Timer not found!\n");
	}

	//We are starting the timer
	uHALr_InstallTimer((unsigned int)GlobalTimer);
	BRHEnableInterrupt(PLAT_TIMERAINT);
	uHALr_printf("InitTimer out!\n");
}

//Function that implements the serial interrupt handler
void SerialHandling() {
	char CharValue;
	unsigned int tmp;
	unsigned int tmp1;

	if( Serial1->lsr & BIT0 ) {  //If we have a receive trasmit
interrupt
		CharValue= Serial1->rxtx;

		if(InputMode==0)
			PutCharacterInInputBuffer(0,CharValue);
		else
		{
			CharValue=tolower(CharValue); //in order to play
even with CAPS LOCK
			switch (CharValue) {
				case '2':
					PutCharacterInInputBuffer(1,CharValue);
					break;
				case '4':
					PutCharacterInInputBuffer(1,CharValue);
					break;
				case '6':
					PutCharacterInInputBuffer(1,CharValue);
					break;
				case '8':
					PutCharacterInInputBuffer(1,CharValue);
					break;
				case 'x':
					PutCharacterInInputBuffer(2,'2');
					break;
				case 'a':
					PutCharacterInInputBuffer(2,'4');
					break;
				case 'd':
					PutCharacterInInputBuffer(2,'6');
					break;
				case 'w':
					PutCharacterInInputBuffer(2,'8');
					break;
				case 'm':
					PutCharacterInInputBuffer(3,'2');
```

```
                        break;
                  case 'j':
                        PutCharacterInInputBuffer(3,'4');
                        break;
                  case 'k':
                        PutCharacterInInputBuffer(3,'6');
                        break;
                  case 'i':
                        PutCharacterInInputBuffer(3,'8');
                        break;
                  case 'v':
                        PutCharacterInInputBuffer(4,'2');
                        break;
                  case 'f':
                        PutCharacterInInputBuffer(4,'4');
                        break;
                  case 'g':
                        PutCharacterInInputBuffer(4,'6');
                        break;
                  case 't':
                        PutCharacterInInputBuffer(4,'8');
                        break;
            }
      }

      return;
}

//if we have a trasmitter empty interrupt
if(TX_EMPTY(UART2_BASE)) {
      if(getOutputBufferFreeSpace()!=0) {  //if there are
characters in the buffer...
            tmp= PopCharacterFromOutputBuffer(&CharValue);
//take characters from outbuffer
            PUT_CHAR(OS_COMPORT,CharValue);
            if(getOutputBufferUsedSpace()==0) {  //if outpout
buffer empty
                  Serial1->ier &= ~ BIT1;
//Disable THR interrupts
            }
      }
}
}

//Function that initialize the serial
void init_ser(void) {
      unsigned int tmp;

      uHALr_InitInterrupts();

      BRHDisableInterrupt(PLAT_UARTINT2);  //Disable any serial
interrupts

      //We are intializing the interrupts and the serial
      tmp=uHALr_RequestInterrupt((unsigned
int)PLAT_UARTINT2,(PrHandler)SerialHandling,(const unsigned
char*)"SerialHandling" );
      if (tmp==0){
            uHALr_printf("RequestInterrupt!!! \n");
      }
```

```
	uHALir_InitSerial(OS_COMPORT, ARM_BAUD_115200 );

	BRHEnableInterrupt(PLAT_UARTINT2);  //Enable serial interrupts

	Serial1->ier |= 1;                  //Enable THR interrupts

	return;
}
//Function that uniintialize the serial
void uninit_ser(void) {
	uHALir_InitSerial(OS_COMPORT, ARM_BAUD_115200 );   //reset port
	uHALr_DisableInterrupt(PLAT_UARTINT2);
	uHALr_FreeInterrupt(PLAT_UARTINT2);
}

//Function which is called at the end of the program for
unintializing the timer
void uninit_timer(void) {
	int tmp;

	//We disable the timer,free the interrupt and update the
internal structure
	tmp=uHALr_FreeTimer((unsigned int)GlobalTimer);
	if(tmp<0) {
		uHALr_printf("The timer is unknown!\n");
	}
}

// Function that sets which buffers received characters are placed
void SetInputMode(int mode) {
	InputMode=mode;
	return;
}

// Function that waits for "ticks" tenth of a second and returns
void Sleep(int ticks) {
	int finalTicks;

	finalTicks=TimerTenths+ticks;

	while(TimerTenths!=finalTicks){                        // Wait
until ticks time has elapsed
		//nothing
	}
	return;
}
```