ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ & ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ

Ver. 1, Rev. 5

Εγχειρίδιο Χρήσης AT91

ΧΑΡΙΔΗΜΟΣ ΒΕΡΓΟΣ
ΝΙΚΟΛΑΟΣ ΚΩΣΤΑΡΑΣ

Μάρτιος 2009

Περιεχόμενα

Πρόλογος **1**

Αρχιτεκτονική του επεξεργαστή **5**

2.1 Γενικά 5
2.2 Καταχωρητές 8
2.3 Εντολές 14
2.3.1 Εκτέλεση υπό συνθήκη 14
2.3.2 Ολισθήσεις 16
2.3.3 Εντολές μεταφοράς δεδομένων 20
2.3.4 Αριθμητικές/Λογικές πράξεις 28
2.3.5 Εντολές διακλάδωσης 31

Εισαγωγή στην Assembly του AT91 **35**

3.1 Γενικά 35
3.2 GNU Assembler 38
3.2.1 Ετικέτες 39
3.2.2 Ντιρεκτίβες 40
3.3 Παραδείγματα 46
3.3.1 Μεταφορά δεδομένων εύρους 32-bit 46
3.3.2 Άθροιση αριθμών εύρους 32-bit 47
3.3.3 Σωρός 48
3.3.4 Σύγκριση αριθμών εύρους 8 bit 52
3.3.5 Πρόσθεση αριθμών εύρους 64 bit 56
3.3.6 Διαίρεση αριθμού εύρους 32 bit με 16 bit διαιρέτη 57

Περιβάλλον εργασίας **61**

4.1 Υλικό 61
4.2 Λειτουργικό σύστημα 61
4.3 GNU tools 65

4.4 Ευκολίες 70

Εξομοιωτής **71**

5.1 Γενικά 71

5.2 Ρυθμίσεις 71

5.3 Χρήση 74

Instruction Set **79**

6.1 Αριθμητικές Εντολές 80

6.1.1 ADC 80

6.1.2 ADD 81

6.1.3 AND 82

6.1.4 BIC 83

6.1.5 CLZ 84

6.1.6 CMN 85

6.1.7 CMP 86

6.1.8 EOR 87

6.1.9 MLA 88

6.1.10 MUL 89

6.1.11 ORR 90

6.1.12 RSB 91

6.1.13 RSC 92

6.1.14 SBC 94

6.1.15 SMLAL 96

6.1.16 SMULL 97

6.1.17 SUB 98

6.1.18 TEQ 100

6.1.19 TST 101

6.1.20 UMLAL 102

6.1.21 UMULL 103

6.2 Εντολές Μετακίνησης 104

6.2.1 LDM 104

6.2.2 LDR 105

6.2.3 LDRB 107

6.2.4 LDRH 108

6.2.5 LDRSB 109

6.2.6 LDRSH 110

6.2.7 MCR 111

6.2.8 MOV 112

6.2.9 MRC 113
6.2.10 MRS 114
6.2.11 MSR 115
6.2.12 MVN 117
6.2.13 STM 118
6.2.14 STR 119
6.2.15 STRB 120
6.2.16 STRH 121
6.2.17 SWP 122
6.2.18 SWPB 123
6.3 Εντολές Διακλάδωσης 124
6.3.1 B,BL 124
6.3.2 BX 125

Πρόλογος

Η πλατφόρμα AT91 σπάει μια μακρά παράδοση του Τομέα Υλικού & Αρχιτεκτονικής του Τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής : τη παράδοση να σχεδιάζουμε εκ του μηδενός τις πλατφόρμες των εργαστηριακών μαθημάτων μας. Παρότι οι συγγραφείς βρέθηκαν στο δίλημμα «να χτίσουμε κάτι εκ του μηδενός» ή «να στηριχθούμε σε κάτι έτοιμο και να αναπτύξουμε πάνω του», υπήρξαν αρκετοί λόγοι που μας ώθησαν στη δεύτερη λύση :

- Η υποστήριξη οποιουδήποτε custom-made συστήματος είναι εξαιρετικά δύσκολη στο ακαδημαϊκό περιβάλλον που τα πρόσωπα εναλλάσσονται διαρκώς.
- Η ανάπτυξη εργαλείων υποστήριξης (εξομοιωτών, αποσφαλματωτών, ...) είναι εξαιρετικά επίπονη.
- Η ανάπτυξη εκ του μηδενός απαιτεί πολύ μεγάλη επένδυση σε χρόνο, με αμφίβολα αποτελέσματα.

Το υλικό της πλατφόρμας AT91 είναι στην ουσία μια εμπορικά διαθέσιμη αναπτυξιακή πλακέτα (AT91SAM9261EK της εταιρείας ATMEL). Στη πλακέτα αυτή μπορούμε να αναπτύξουμε εφαρμογές υλικού και λογισμικού για τον μικροελεγκτή AT91SAM9261, ο οποίος είναι μια εμπλουτισμένη υλοποίηση από την εταιρεία ATMEL του επεξεργαστή ARM926EJ-S της εταιρείας Advanced Risc Machines (ARM). Η επιλογή του συγκεκριμένου επεξεργαστικού στοιχείου δεν έγινε τυχαία. Σήμερα (τέλη 2007) όλα τα μικροϋπολογιστικά συστήματα που αναπτύσσονται (κυρίως με τη μορφή των ενσωματωμένων συστημάτων) βασίζονται είτε στην αρχιτεκτονική της ARM είτε στην αρχιτεκτονική των PowerPC. Η πρώτη όντας πιο ανοικτή, υλοποιήσιμη από πολύ μεγάλη γκάμα εταιρειών με λογικά κόστη κατέχει πάνω από το 70% της παγκόσμιας αγοράς. Ο συγκεκριμένος μικροελεγκτής χρονίζεται με ένα ρολόι των 200 MHz, διαθέτει στο ίδιο ολοκληρωμένο διαφορετική κρυφή μνήμη για δεδομένα και διαφορετική για εντολές (Harvard architecture - split caches) κάθε μία των 16K, ενσωματώνει τη μονάδα διαχείρισης μνήμης (MMU), προσφέρει δύο εναλλακτικά σετ εντολών (ARM / THUMB) με ειδικές εντολές για τη γρήγορη επεξεργασία ψηφιακού σήματος (DSP) και για την υλοποίηση της Jazelle αρχιτεκτονικής για Java. Στην

ουσία επιλέξαμε τον πιο ισχυρό μικροελεγκτή (210 MIPS) προερχόμενο από την αρχιτεκτονική που όσοι απόφοιτοί μας ασχοληθούν με την ανάπτυξη ενσωματωμένων συστημάτων και εφαρμογών είναι εξαιρετικά πιθανόν να χρησιμοποιήσουν.

Αντίθετα, όλο το λογισμικό της πλακέτας είναι προσαρμογή λογισμικού ανοικτού κώδικα (GNU licensed) για τη συγκεκριμένη αρχιτεκτονική, που έγινε από τον κ. Κωσταρά στα πλαίσια της Μεταπτυχιακής διπλωματικής του εργασίας. Παρότι κι εδώ θα μπορούσαμε να χρησιμοποιήσουμε έτοιμες λύσεις (τα Windows CE είναι σήμερα ο κύριος αντίπαλος για εφαρμογές ενσωματωμένων συστημάτων εφόσον δεν υφίστανται απαιτήσεις πραγματικού χρόνου), το ελάχιστο κόστος, η δυνατότητα μεταφοράς κώδικα σε διαφορετικές πλατφόρμες μα κυρίως η θέλησή μας να παρέχουμε ένα σύστημα στο οποίο οι μελλοντικοί φοιτητές θα μπορούν να αναπτύσσουν και δικές τους εφαρμογές υποστήριξης (ρουτίνες λειτουργικού, οδηγούς συσκευών) καθοδήγησαν την επιλογή μας. Επίσης η αναβάθμιση σε επόμενες εκδόσεις λειτουργικού είναι μια σαφώς ανοιχτή διαδικασία. Στο AT91 τη διαχείριση του υλικού αναλαμβάνει προσαρμοσμένο λειτουργικό σύστημα Linux με πυρήνα της έκδοσης 2.6.20. Προσαρμόσθηκαν επίσης όλα τα GNU εργαλεία : συμβολομεταφραστές, assemblers, αποσφαλματωτές. Έχοντας προσαρμόσει τα GNU εργαλεία στο υλικό μας, περάσαμε σε σχεδόν αυτοματοποιημένη προσαρμογή όλων των αναγκαίων πακέτων λογισμικού (π.χ. editors), προγραμμάτων οδήγησης (π.χ. USB driver) και άλλων, καταλήγοντας σε ένα πλήρως λειτουργικό σύστημα.

Η εμπειρία δέκα και πλέον χρόνων διδασκαλίας και επίβλεψης των εργαστηριακών ασκήσεων της συμβολικής γλώσσας στο προηγούμενο του AT91 σύστημα, έδειξε ότι το μεγαλύτερο μειονέκτημα ήταν η απουσία ενός εξομοιωτή. Οι λίγες εργαστηριακές ώρες δεν έφταναν στους φοιτητές για τη πλήρη κατανόηση της συμβολικής γλώσσας. Τα αποτελέσματα τραγικά : αντιγραφή των εργαστηριακών ασκήσεων και αποτυχία στις εξετάσεις. Πολλοί συμφοιτητές σας προσπάθησαν στο παρελθόν να αναπτύξουν εξομοιωτές της προηγούμενης πλατφόρμας, άλλοτε με καλά και άλλοτε με άθλια αποτελέσματα. Η υιοθέτηση λογισμικού ανοικτού κώδικα, προφανώς μας δίνει τη δυνατότητα να χρησιμοποιήσουμε κάποιον από τους διαθέσιμους εξομοιωτές. Έτσι μαζί με το AT91, σας παρέχονται το δωρεάν μέρος των αναπτυξιακών εργαλείων της εταιρείας Keil με τα οποία μπορείτε να αναπτύξετε εφαρμογές για οποιονδήποτε επεξεργαστή / μικροελεγκτή των αρχιτεκτονικών ARM και τα GNU εργαλεία για λειτουργικά συστήματα της Microsoft. Δίνεται λοιπόν η ευκαιρία στο φοιτητή να έχει ένα εξομοιούμενο AT91 σπίτι του. Εκεί να δίνει τις πρώτες λύσεις στις εργαστηριακές ασκήσεις, τις οποίες μπορεί να δοκιμάζει, να τις εκτελεί βηματικά, να διαπιστώνει λάθη και να κάνει αλλαγές μέχρι να φτάσει στο επιθυμητό αποτέλεσμα. Τον κώδικά του μπορεί ακολούθως απλά να τον μεταφέρει (και ηλεκτρονικά φυσικά, μέσω οποιασδήποτε USB συσκευής) στο πραγματικό AT91 ώστε να επιδείξει τη σωστή λειτουργία του.

Ενα τόσο δυνατό εργαλείο σα το AT91 φυσικά δε φτιάχτηκε μόνο για το εργαστήριο συμβολικής γλώσσας. Πέρα από το εγχειρίδιο χρήσης που κρατάτε αυτή τη στιγμή, αναπτύσσεται και άλλο εκπαιδευτικό υλικό γύρω από το AT91. Οι ασκήσεις συμβολικής γλώσσας είναι ένα μικρό πρώτο μέρος από αυτό. Σκοπεύουμε στη νέα πλατφόρμα να μεταφέρουμε επίσης μέρος των εργαστηρίων Αρχιτεκτονικής και μεγάλο μέρος των εργαστηρίων Μικροϋπολογιστών. Η τεράστια αυτή προσπάθεια δε θα ήταν δυνατή χωρίς την οικονομική συνδρομή που μας προσφέρθηκε μέσω της δράσης για αναμόρφωση και ανάπτυξη εκπαιδευτικού υλικού του προγράμματος ΕΠΕΑΚ ΙΙ.

Ελπίζουμε ότι το AT91 θα αποτελέσει ένα καλό εκπαιδευτικό εργαλείο.

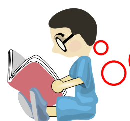
Χ. Βέργος
Μηχανικός Η/Υ & Πληροφορικής, Ph.D., Αναπληρωτής Καθηγητής.

Ν. Κωσταράς
Μηχανικός Η/Υ & Πληροφορικής, MSc

Αρχιτεκτονική του επεξεργαστή

2.1 Γενικά

Ο μικροελεγκτής *AT91SAM9261* της εταιρίας *ATMEL* ο οποίος υπάρχει στο AT91 είναι ένα ολοκληρωμένο κύκλωμα χαμηλής κατανάλωσης ενέργειας, βασισμένο στον επεξεργαστή *ARM926EJ-S*, με συχνότητα λειτουργίας τα 200 MHz. Ο ARM926EJ-S έχει σχεδιαστεί από την εταιρεία ARM και ο σχεδιασμός του είναι προσανατολισμένος για χρήση σε ενσωματωμένες εφαρμογές (embedded applications). Τα χαρακτηριστικά που τον διακρίνουν είναι η απλότητα σχεδιασμού (ανήκει σε οικογένεια *32bit RISC* επεξεργαστών) και η ευέλικτη αρχιτεκτονική του, που δίνει τη δυνατότητα ανάπτυξης μιας ευρύτατης γκάμας από εφαρμογές (υλοποίηση επικοινωνιακών πρωτοκόλλων, αλγορίθμων κρυπτογράφησης, επεξεργασία ήχου και video, παιχνιδιών κ.α). Το διάγραμμα αρχιτεκτονικής του επεξεργαστή φαίνεται στην εικόνα *Διάγραμμα Αρχιτεκτονικής* (κεφ. 2.1, σελ. 6).



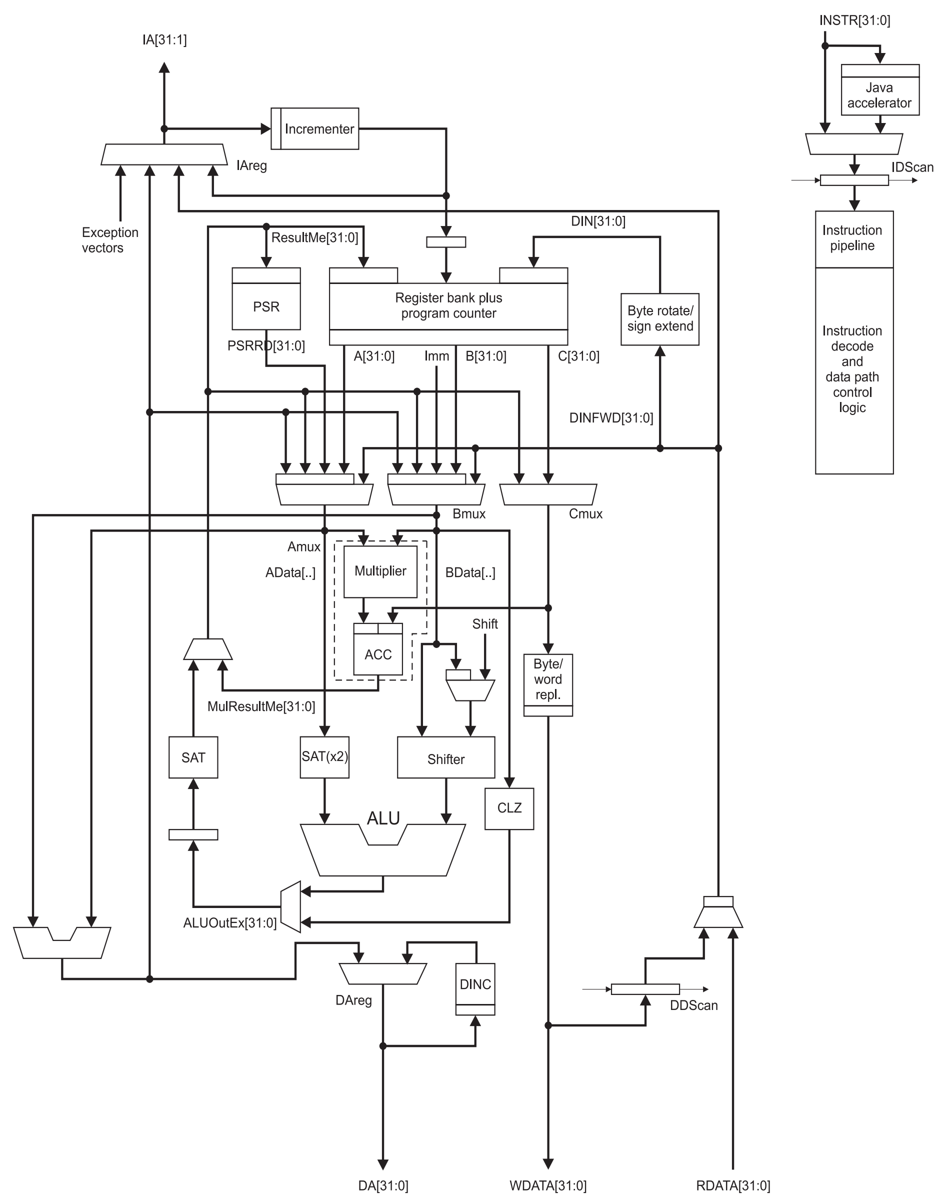
Επεξήγηση

RISC (Reduced Instruction Set Computer) ονομάζουμε κάθε αρχιτεκτονική επεξεργαστή με εξαιρετικά λίγες και απλές εντολές, κάτι που κάνει την υλοποίηση του επεξεργαστή απλούστερη και ταυτόχρονα του επιτρέπει να εκτελεί 1 εντολή ανά περίοδο ρολογιού. Εναλλακτικά, υπάρχει και η CISC (Complex Instruction Set Computer) αρχιτεκτονική, όπου οι εντολές είναι πολλές, πολύπλοκες και εκτελούν σύνθετες εργασίες.

Τα βασικά τμήματα του επεξεργαστή είναι οι **καταχωρητές** (registers - παρακάτω συμβολίζονται με R ή r) και η **Αριθμητική/Λογική Μονάδα**[1]. Οι καταχωρητές είναι μικρές μνήμες εύρους 32 δυαδικών ψηφίων (bits) η κάθε μία, οι οποίες μπορούν να προσπελαστούν με ρυθμό αντίστοιχο του κύκλου ρολογιού[2]. Οι καταχωρητές αναλαμβάνουν να αποθηκεύουν δεδομένα προερχόμενα από την εξωτερική μνήμη του συστήματος και να τροφοδοτούν τον επεξεργαστή με αυτά. Η ΑΛΜ δεν επεξεργάζεται άμεσα τα δεδομένα

[1] Arithmetic Logic Unit - ΑΛΜ - ALU

[2] Στις RISC αρχιτεκτονικές ο κύκλος ρολογιού ισούται και μπορεί να αναφέρεται εναλλακτικά και ως κύκλος μηχανής



Σχήμα 2.1: Διάγραμμα Αρχιτεκτονικής

από την εξωτερική μνήμη, αλλά πρέπει πρώτα να αποθηκευτούν σε κάποιον καταχωρητή. Τα αποτελέσματα της ΑΛΜ πριν σταλούν στην εξωτερική μνήμη, πάλι ενδιάμεσα πρέπει να αποθηκεύονται σε κάποιο καταχωρητή.

Ο επεξεργαστής για να ζητήσει μια εντολή ή ένα δεδομένο το οποίο βρίσκεται κάπου στο σύστημα μνήμης ακολουθεί μια σειρά από βήματα, ένα από τα οποία είναι η εμφάνιση της διεύθυνσης της ζητούμενης πληροφορίας στην αρτηρία διευθύνσεων (address bus) του συστήματος. Όλα τα κυκλώματα που απαρτίζουν το σύστημα μνήμης «ακούν» την αρτηρία διευθύνσεων και αναλαμβάνουν να ελέγξουν αν η διεύθυνση που εμφανίστηκε αναφέρεται σε κάποια θέση μνήμης που τους αντιστοιχεί. Το κύκλωμα που περιέχει τη συγκεκριμένη διεύθυνση εμφανίζει την αποθηκευμένη πληροφορία στην αρτηρία δεδομένων (data bus), ώστε αυτή να μεταφερθεί στον επεξεργαστή. Το εύρος της αρτηρίας διευθύνσεων είναι 32 δυαδικών ψηφίων, και συνεπώς στο AT91 έχουμε 2^{32} διαφορετικές διευθυνσιοδοτούμενες θέσεις μνήμης.

Επεξήγηση . . .

Με τον όρο *εξωτερική μνήμη* αναφερόμαστε σε κάθε μνήμη που υλοποιείται σε ολοκληρωμένα κυκλώματα εκτός αυτού του επεξεργαστή. Τα δεδομένα μιας τέτοιας μνήμης μπορούμε να τα προσπελάσουμε μέσω των αρτηριών του συστήματος, χρησιμοποιώντας τη διεύθυνση του επιθυμητού δεδομένου ως κλειδί αναζήτησης. Κάθε ολοκληρωμένο μνήμης ή περιφερειακή συσκευή που είναι συνδεδεμένη στην αρτηρία διευθύνσεων καταλαμβάνει ένα συγκεκριμένο αριθμό από θέσεις μνήμης (χώρος διευθύνσεων - address space). Στο χώρο διευθύνσεων ανάμεσα στις θέσεις που καταλαμβάνουν τα ολοκληρωμένα μνήμης και τα περιφερειακά μπορεί να μην έχει αντιστοιχιστεί τίποτα, οπότε αν προσπελάσουμε μια τέτοια θέση θα μας επιστραφεί το 0 ή θα μας ειδοποιήσει ο επεξεργαστής ότι έχει συμβεί σφάλμα.

Ο επεξεργαστής υποστηρίζει επτά διαφορετικές καταστάσεις λειτουργίας, όπου η διαφοροποίησή τους σχετίζεται με τα δικαιώματα πρόσβασης σε περιοχές εξωτερικής μνήμης. Δηλαδή, για να μπορέσουμε να προσπελάσουμε μερικές περιοχές μνήμης που έχουν δεσμευτεί για ειδική χρήση (πχ. χρήση από το λειτουργικό σύστημα) πρέπει να μεταβούμε σε κατάσταση με αυξημένα δικαιώματα. Ο λόγος ύπαρξης των διαφορετικών καταστάσεων λειτουργίας σχετίζεται με την απαίτηση του λειτουργικού συστήματος να μπορεί να διαχειρίζεται τους διαθέσιμους πόρους (resources) αποδοτικά και χωρίς να επηρρεάζεται από τις εφαρμογές των χρηστών. Οι διαφορετικές καταστάσεις του επεξεργαστή φαίνονται στον πίνακα *Processor Modes* (κεφ. 2.1 σελ. 8). Η κατάσταση του επεξεργαστή στην οποία βρισκόμαστε όταν εκτελούμε τα προγράμματά μας είναι η User.

Mode	Περιγραφή
User	Η κατάσταση εκτέλεσης προγραμμάτων
FIQ	Κατάσταση αμέσως μετά από Fast Interrupt
IRQ	Κατάσταση αμέσως μετά από Interrupt
Supervisor	Κατάσταση με αυξημένα δικαιώματα
Abort	Κατάσταση μετά από λάθος προσπέλαση στη μνήμη
Undefined	Κατάσταση μετά από εκτέλεση εντολής που δεν ανήκει στο σύνολο εντολών
System	Κατάσταση με αυξημένα δικαιώματα

Πίνακας 2.1: Processor Modes

2.2 Καταχωρητές

Ο επεξεργαστής μας διαθέτει 37 καταχωρητές, εκ των οποίων 31 είναι γενικού και 6 ειδικού σκοπού. Από αυτούς, μόνο οι 17 είναι ταυτόχρονα διαθέσιμοι και προσπελάσιμοι ανά πάσα στιγμή. Οι πρώτοι 8 (r0 έως r7) είναι κοινοί για όλες τις καταστάσεις του επεξεργαστή, ενώ οι υπόλοιποι 7 γίνονται διαθέσιμοι ανάλογα με την κατάσταση στην οποία βρίσκεται ο επεξεργαστής (αυτό σημαίνει για παράδειγμα, πως δεν μπορούμε να προσπελάσουμε τον καταχωρητή r8 της κατάστασης Fiq όταν είμαστε σε κατάσταση User, διότι ένας άλλος καταχωρητής είναι διαθέσιμος ως r8 στην κατάσταση User). Στον πίνακα *Καταχωρητές* (κεφ. 2.2 σελ. 8) καταγράφονται οι καταχωρητές που είναι διαθέσιμοι στις διάφορες καταστάσεις λειτουργίας του επεξεργαστή μας (εικόνα 2.2, σελίδα 9). Η σημειολογία όσων αναγράφονται σε αυτό το πίνακα θα γίνει πλήρως κατανοητή στις επόμενες παραγράφους.

Mode	User Mode Regs	Additional Regs
User	r0-r14, r15, CPSR	
FIQ	r0-r7, r15, CPSR	r8_fiq-r14_fiq,SPSR_fiq
IRQ	r0-r12, r15, CPSR	r13_irq-r14_irq,SPSR_irq
Supervisor	r0-r12, r15, CPSR	r13_svc-r14_svc,SPSR_svc
Abort	r0-r12, r15, CPSR	r13_abt-r14_abt,SPSR_abt
Undefined	r0-r12, r15, CPSR	r13_und-r14_und,SPSR_und
System	r0-r12, r15, CPSR	r13_sys-r14_sys,SPSR_sys

Πίνακας 2.2: Καταχωρητές

Καταχωρητές γενικού σκοπού και program counter του ARM

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

Καταχωρητές κατάστασης του ARM

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Υποδηλώνει πως ο καταχωρητής που υπήρχε σε κανονική κατάσταση λειτουργίας έχει αντικατασταθεί από τον αντίστοιχο της κατάστησης σφάλματος ή interrupt.

Σχήμα 2.2: Διαθέσιμοι καταχωρητές σε κάθε κατάσταση λειτουργίας

Το εύρος των καταχωρητών είναι 32 ψηφία, κάτι που ισοδυναμεί με μια λέξη (word) του συστήματός μας. Μπορούμε όμως στους υπολογισμούς μας να χρησιμοποιούμε ένα τμήμα της πληροφορίας κάποιου καταχωρητή, όπως τα 16 λιγότερο σημαντικά ψηφία (μισή λέξη - halfword) ή τα 8 λιγότερο σημαντικά ψηφία (byte).

Για τα προγράμματά μας μπορούμε να χρησιμοποιούμε τους καταχωρητές r0 εως r12, αλλάζοντας το περιεχόμενό τους. Οι επόμενοι 3 καταχωρητές, r13 έως r15, έχουν συγκεκριμένη λειτουργία που σχετίζεται με τον τρόπο εκτέλεσης του προγράμματος και όχι με το περιεχόμενο του προγράμματος. Πιο συγκεκριμένα, ο r15 είναι ο μετρητής προγράμματος (Program Counter - PC), ο r14 είναι ο καταχωρητής διασύνδεσης κώδικα (Branch & Link) και ο r13 είναι ο δείκτης σωρού (Stack Pointer). Ο τελευταίος καταχωρητής ειδικού σκοπού (καταχωρητής κατάστασης - Current Processor Status Register - CPSR) έχει ιδιαίτερη σημασία για τη ροή εκτέλεσης του προγράμματος. Πολλές φορές στα προγράμματά μας θέλουμε να εκτελέσουμε κώδικα υπό συνθήκη, κώδικα δηλαδή που σχετίζεται με το αποτέλεσμα κάποιας προηγούμενης αριθμητικής ή λογικής εντολής. Κάθε φορά που εκτελείται μια αριθμητική ή λογική εντολή, μπορεί να αποθηκεύουμε σε αυτόν το καταχωρητή δυαδικές σημαίες κατάστασης (status bits) που μας δείχνουν το είδος[3] του αποτελέσματος που πήραμε (εικόνα Status Register κεφ. 2.3, σελ. 12, πίνακας Status Register κεφ. 2.3, σελ. 11. Σε όλο το εγχειρίδιο χρησιμοποιούμε το συμβολισμό 0xWXYZ για τη δεκαεξαδική ποσότητα WXYZ και #WXYZ για τη δεκαδική ποσότητα WXYZ). Κάθε εντολή που έχει ενεργοποιημένη την επιλογή αλλαγής του CPSR προκαλεί την ενημερωμέρωση των δυαδικών σημαιών κατάστασης βάσει του είδους του αποτελέσματος που θα προκύψει από την εκτέλεσή της. Οι περισσότερες εντολές υποστηρίζουν την επιλογή αλλαγής του CPSR με την προσθήκη του χαρακτήρα 'S' στο τέλος του ονόματος της εντολής.

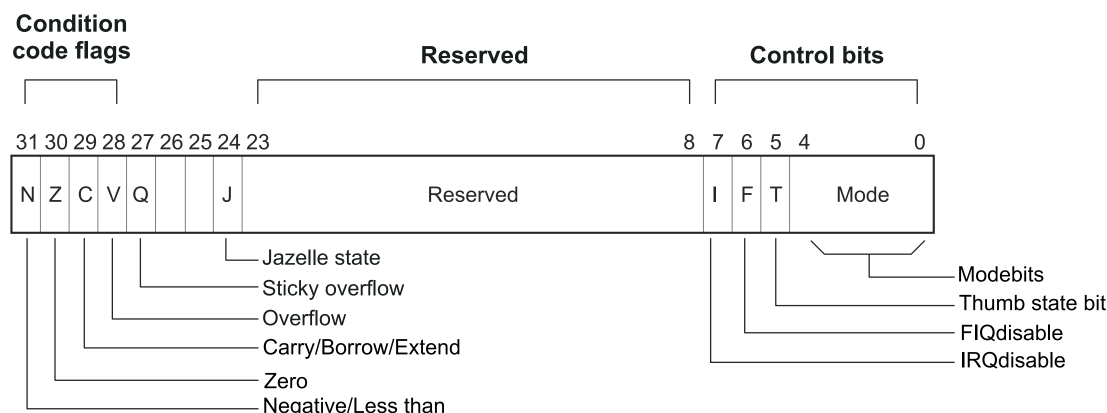
Ο r15 ονομάζεται μετρητής προγράμματος (PC) μιας και ο ρόλος του είναι να περιέχει τη διεύθυνση στην οποία υπάρχει η επόμενη προς εκτέλεση εντολή. Κάθε φορά που εκτελείται μια εντολή, ο PC μεταβάλεται αυτόματα από τον επεξεργαστή, ώστε πάντα να περιέχει τη διεύθυνση της επόμενης προς εκτέλεση εντολής. Κατά το πρώτο στάδιο εκτέλεσης μιας εντολής, ο επεξεργαστής διαβάζει το περιεχόμενο του PC και το τοποθετεί στην αρτηρία διευθύνσεων. Το σύστημα μνήμης μέσω της αρτηρίας δεδομένων του διαβιβάζει το κωδικό της εντολής προς εκτέλεση[4]. Ο PC αυξάνεται αμέσως μετά αυτόματα κατά 4, διότι κάθε εντολή στο σύστημά μας έχει μέγεθος 4 bytes. Το περιεχόμενο του PC μπορούμε να το μεταβάλουμε και εμείς γράφοντας μια τιμή σε αυτόν, αρκεί να είμαστε σίγουροι ότι η τιμή που γράψαμε αντιστοιχεί σε μια διεύθυνση στην οποία υπάρχει σωστός κώδικας

[3] Πιθανά είδη : μηδενικό ή αρνητικό αποτέλεσμα ή υπερχείλιση ή ...

[4] Αν η διεύθυνση δεν περιέχει σωστό κωδικό λειτουργίας, ο επεξεργαστής θα μεταβεί σε κατάσταση Prefetch Abort

Bit	**Περιγραφή**
N	1 αν το αποτέλεσμα της πράξης ήταν αρνητικό
Z	1 αν το αποτέλεσμα της πράξης ήταν μηδενικό
C	1 σε πρόσθεση που δημιούργησε κρατούμενο ή σε αφαίρεση που δε δημιούργησε δανεικό ή σε πράξη ολίσθησης που το τελευταίο ψηφίο που ολίσθησε εκτός του καταχωρητή ήταν 1
V	1 αν το αποτέλεσμα της πράξης δημιούργησε υπερχείλιση (overflow)
Q	Το ψηφίο αυτό επηρεάζεται από ένα σύνολο πράξεων, όπως **QADD** κ.α, και παραμένει ενεργό μέχρι να μηδενιστεί με εγγραφή στον CPSR. Δεν αποτελεί συνθήκη για την εκτέλεση εντολών, αλλά είναι βοηθητικό για την εκτέλεση ειδικών εντολών που απαιτούν αλγοριθμικό υπολογισμό.
J	1 αν ο επεξεργαστής έχει εισέλθει σε Jazelle mode και εκτελεί εντολές Java.
I	1 αν ο επεξεργαστής έχει απενεργοποιήσει τις διακοπές (interrupts) IRQ. Μόλις μηδενίσουμε το συγκεκριμένο ψηφίο τα σήματα διακοπών θα σηματοδοτήσουν άμεσα τον επεξεργαστή.
F	1 αν ο επεξεργαστής έχει απενεργοποιήσει τις γρήγορες διακοπές FIQ. Τα ψηφία **I** & **F** του CPSR δεν ενεργοποιούνται αυτόματα, αλλά τη κατάστασή τους μπορεί να τη μεταβάλλει ο χρήστης.
T	1 αν ο επεξεργαστής έχει εισέλθει σε Thumb mode και εκτελεί εντολές Thumb.
Mode	Τα 5 αυτά δυαδικά ψηφία δείχνουν σε ποια κατάσταση του επεξεργαστή βρισκόμαστε.
	0x10 User **0x11 FIQ** **0x12 FIQ** **0x13 Supervisor** **0x17 Abort** **0x1B Undefined** **0x1F System**

Πίνακας 2.3: Status Register



Σχήμα 2.3: Status Register

προς εκτέλεση[5]. Για παράδειγμα, αν έχουμε τοποθετήσει κώδικα στην περιοχή μνήμης 0x50000000 και ο PC δείχνει στη διεύθυνση 0xC0005100, τότε μπορούμε (όταν θέλουμε να εκτελέσουμε τον κώδικά μας) να μεταβάλουμε το περιεχόμενο του PC και να του γράψουμε την τιμή 0x50000000. Ο επεξεργαστής, μετά από την μεταβολή του PC, θα είναι έτοιμος να ζητήσει τα περιεχόμενα από τη θέση μνήμης 0x50000000 και να τα εκτελέσει.

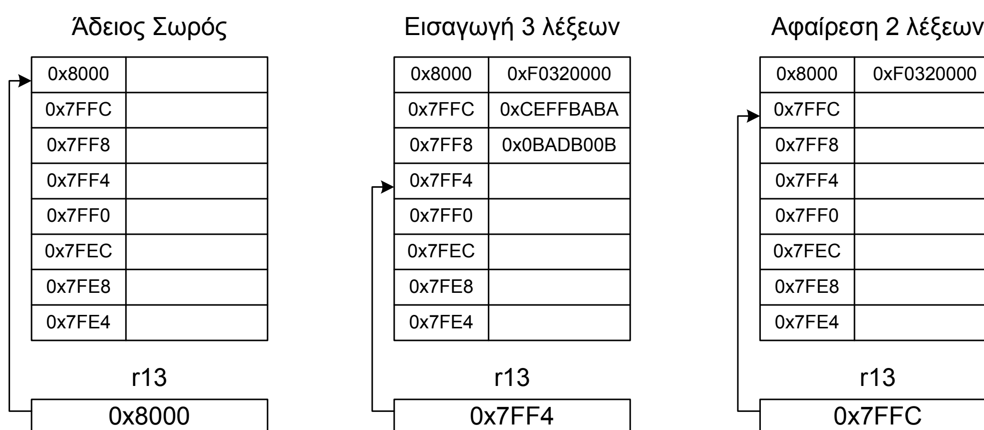
Όμως, όταν τελειώσει η εκτέλεση του κώδικα από τη θέση 0x50000000, είναι εξαιρετικά πιθανό ότι θα θέλουμε να ξαναβάλουμε στον PC τη τιμή που είχε πριν από τη μεταβολή του, δηλαδή την 0xC0005100. (Το ανάλογο παράδειγμα στη γλώσσα C είναι η κλήση μιας συνάρτησης. Μόλις τελειώσει η εκτέλεση της συνάρτησης, επιστρέφουμε στο σημείο που έγινε η κλήση της και συνεχίζουμε με την επόμενη εντολή). Κατά τη συγγραφή του κώδικα, είναι αδύνατο να γνωρίζουμε σε ποια θέση θα τοποθετηθεί αυτός καθώς αυτό είναι ευθύνη του λειτουργικού συστήματος. Πολύ περισσότερο είναι αδύνατο να γνωρίζουμε τη διεύθυνση που θα πρέπει να φορτώσουμε στον PC μετά το τέλος της εκτέλεσης του προγράμματός μας. Γι' αυτό το λόγο χρησιμοποιούμε τον καταχωρητή r14, για να αποθηκεύουμε σε αυτόν το περιεχόμενο του PC πριν εκτελέσουμε τη μετάβαση σε άλλη θέση μνήμης[6]. Έτσι, πριν γράψουμε την τιμή 0x50000000 στον PC, αντιγράφουμε το περιεχόμενό του στον r14, ο οποίος έχει το όνομα *Branch & Link*. Μόλις τελειώσει η εκτέλεση του κώδικα στη θέση 0x50000000, αντιγράφουμε το περιεχόμενο του Branch & Link στον PC, αρκεί φυσικά να μην έχουμε αλλάξει το περιεχόμενό του κατά την εκτέλεση του κώδικά μας. Για να μην χρειάζεται από τους προγραμματιστές να εκτελούν τη διαδικασία αντιγραφής και αποκατάστασης, όταν χρησιμοποιούμε συγκεκριμένες εντολές

[5]Στην περίπτωση που στην διεύθυνση που προσπαθήσει να προσπελάσει ο πεξεργαστής δεν αντιστοιχεί εξωτερική μνήμη, ο επεξεργαστής θα μεταβεί σε κατάσταση λειτουργίας Undefined

[6]Διακλάδωση

διακλάδωσης, όι διαδικασίες αυτές εκτελούνται αυτόματα από τον επεξεργαστή.

Πολλές φορές είναι απαραίτητο να περάσουμε δεδομένα στη συνάρτησή μας σαν παραμέτρους, ώστε να γίνει η επεξεργασία τους και να πάρουμε πίσω τα αποτελέσματα. Επίσης, κατά την εκτέλεση μιας συνάρτησης είναι πολύ πιθανό να χρειαστεί να χρησιμοποιήσουμε περισσότερες μεταβλητές από τους 12 διαθέσιμους καταχωρητές γενικού σκοπού, ή να δεσμεύσουμε μνήμη μέσω του λειτουργικού συστήματος, και όταν τελειώσουμε να την επιστρέψουμε σε αυτό για να μπορεί να την χρησιμοποιήσει άλλη διεργασία. Για όλους αυτούς τους λόγους, το λειτουργικό σύστημα παρέχει μια περιοχή μνήμης σε κάθε πρόγραμμα η οποία ονομάζεται σωρός (stack) και χρησιμοποιείται σαν πρόχειρο. Ο σωρός λειτουργεί σα μια LIFO (Last In First Out) δομή, σα μια δομή δηλαδή που το τελευταίο εισαχθέν στοιχείο είναι το πρώτο που μπορεί να εξαχθεί. Για παράδειγμα αν εισάγουμε στο σωρό τα δεδομένα Α, Γ, Ε & Ι με τη σειρά που καταγράφονται (δηλαδή με το Α πρώτο και το Ι τελευταίο) και εκτελέσουμε διαδικασία επαναφόρτωσης στους καταχωρητές, θα πάρουμε τα δεδομένα με την αντίστροφη σειρά, δηλαδή Ι, Ε, Γ & Α.



Σχήμα 2.4: Παράδειγμα σωρού.

Για τη διαχείριση του σωρού, υπάρχει ένας καταχωρητής που έχει σαν περιεχόμενο τη διεύθυνση της μνήμης με την πρώτη ελεύθερη θέση του σωρού, ώστε όταν εκτελούμε εγγραφή στο σωρό να εμφανίζεται η συγκεκριμένη διεύθυνση στην αρτηρία δεδομένων. Ο καταχωρητής αυτός είναι ο r13 (δείκτης σωρού - Stack pointer). Κάθε θέση στο σωρό έχει μέγεθος 32 ψηφίων ή 4 bytes, οπότε η αύξηση ή η μείωση του δείκτη σωρού γίνεται πάντα πάντα κατά 4. Λόγω της LIFO λειτουργίας του, ο σωρός έχει το χαρακτηριστικό να ξεκινά από μια διεύθυνση[7] και όσο γεμίζει με δεδομένα αυτή η διεύθυνση να μειώνεται. Αυτό

[7]Στα προγράμματά μας τη διεύθυνση αυτή την αναθέτει το λειτουργικό σύστημα

φαίνεται και στην εικόνα Stack (κεφ. 2.4, σελ. 13).

2.3 Εντολές

Οι εντολές αποτελούν τη βάση του προγράμματός μας, αφού με αυτές οδηγούμε τον επεξεργαστή στην εκτέλεση των βημάτων που χρειάζονται για τη λύση του προβλήματός μας. Οι εντολές χωρίζονται σε κατηγορίες, ανάλογα με την ενέργεια που εκτελούν και οι κατηγορίες αυτές περιγράφονται στον πίνακα *Κατηγορίες εντολών* (παρ. 2.3, σελ. 14).

Κατηγορία	**Περιγραφή**
Μεταφορά δεδομένων	Μεταφορά δεδομένων από και προς την κύρια μνήμη & τους καταχωρητές
Αριθμητικές πράξεις	Πρόσθεση, πολλαπλασιασμός...
Λογικές πράξεις	AND, OR, XOR...
Διακλαδώσεις	Αλλαγή της ροής εκτέλεσης του κώδικα
Συστήματος	Προσπέλαση συνεπεξεργαστή, προγραμματισμός εξαιρέσεων

Πίνακας 2.4: Κατηγορίες εντολών

2.3.1 Εκτέλεση υπό συνθήκη

Η αρχιτεκτονική μας έχει σχεδιαστεί για να εκτελεί απλές πράξεις με μεγάλη ταχύτητα. Για το σκοπό αυτό χρησιμοποιεί βαθμωτή επεξεργασία (pipelining[8]). Είναι γνωστό ότι διακλαδώσεις και άλματα προκαλούν «σπασίματα» της βαθμωτής επεξεργασίας και συνεπώς υποβάθμιση της απόδοσης. Η λύση της εισαγωγής υλικού για πρόβλεψη διακλαδώσεων (branch prediction) δεν είναι και η καλύτερη λύση για τη δική μας αρχιτεκτονική καθώς αυξάνει τη πολυπλοκότητα υλοποίησης και επηρρεάζει την ταχύτητα. Η λύση η οποία ακολουθήθηκε από τους σχεδιαστές της ARM είναι να δώσουν σε κάθε εντολή τη δυνατότητα εκτέλεσης υπό συνθήκη. Αυτό σημαίνει πως κάθε εντολή θα εκτελεστεί μόνο αν πληρούνται όλες οι συνθήκες που ορίζουμε. Οι συνθήκες έχουν σχέση με τις σημαίες κατάστασης του CPSR (Carry, Negative κλπ) και αναγράφονται στον πίνακα *Συνθήκες εκτέλεσης* (παρ. 2.3.1, σελ. 15). Η ενεργοποίηση των συνθηκών σε κάθε εντολή γίνεται με την αναγραφή του μνημονικού ονόματος της συνθήκης ακριβώς μετά από το μνημονικό

[8]Η τεχνική του pipelining χωρίζει την εκτέλεση μιας εντολής σε ανεξάρτητα στάδια. Μόλις ολοκληρώσει κάποια εντολή ένα στάδιό της, προωθείται στο επόμενο, ενώ η επόμενη εντολή μπορεί να απασχολεί παράλληλα το απελευθερωθέν υλικό του ολοκληρωθέντος σταδίου. Αν έχουμε 5 στάδια ολοκλήρωσης μιας εντολής, η επόμενη εντολή αντί να περιμένει 5 κύκλους μέχρι να αρχίσει να εκτελείται, με το pipelining περιμένει μόνο 1.

όνομα της εντολής (πχ. στην εντολή **ADDCS R0, R1, R2**, θα εκτελεστεί η πρόσθεση μόνο αν το Carry CPSR bit είναι ίσο με 1).

Συνθήκη	**Περιγραφή**
CS/HS	C = 1 (Unsigned Higher or Same)
CC/LO	C = 0 (Unsigned Lower)
EQ	Z = 1 (Equal)
NE	Z = 0 (Not Equal)
VS	V = 1
VC	V = 0
MI	N = 1 (Minus)
PL	N = 0 (Plus)
GE	N = V (Signed Greater Than or Equal)
GT	Z = 0 και N = V (Signed Greater Than)
HI	C = 1 και Z = 0 (Unsigned Higher)
LE	Z = 1 ή N != V (Signed Less Than or Equal)
LT	N != V (Signed Less Than)
LS	C = 0 ή Z = 1 (Unsigned Lower or Same)

Πίνακας 2.5: Συνθήκες εκτέλεσης

Στον πίνακα βλέπουμε πως η συνθήκη CS (η σημαία κρατουμένου του CPSR είναι 1) έχει το ίδιο νόημα με την συνθήκη HS. Αυτό συμβαίνει εξαιτίας του τρόπου αντιμετώπισης των συνθηκών εκτέλεσης. Κατά τη συγγραφή του προγράμματός μας είναι συνηθισμένο να χρησιμοποιείται η πράξη **CMP**, η οποία εκτελεί μια εικονική[9] αφαίρεση ανάμεσα σε δυο τιμές. Το αποτέλεσμα μιας αφαίρεσης, αν είναι θετικός αριθμός, θα οδηγήσει στην ενεργοποίηση της σημαίας κρατουμένου και την απενεργοποίηση της σημαίας μηδενικού αποτελέσματος, άρα αν στη συνέχεια ελέγξουμε την τιμή των σημαιών κρατουμένου και μηδενικού και τις βρούμε 1 & 0 αντίστοιχα θα ξέρουμε πως ο αφαιρετέος ήταν μεγαλύτερος από τον αφαιρέτη (γι'αυτό και η συνθήκη που θέλει το C = 1 & Z = 0 είναι η Unsigned Higher). Κατ' αναλογία, ακολουθούμε την ίδια αντιμετώπιση και με τις υπόλοιπες συνθήκες, δηλαδή χρησιμοποιούμε μια εντολή σύγκρισης η οποία θα ενημερώνει συγκεκριμένα ψηφία του CPSR και στη συνέχεια τα ελέγχουμε για να δούμε αν έχουν ενεργοποιηθεί τα ψηφία που καθορίζουν αν το άλμα ή η διακλάδωσή μας πρέπει να γίνει ή όχι.

[9]Το αποτέλεσμα της δεν αποθηκεύεται σε κάποιον καταχωρητή, αλλά χρησιμοποιείται μόνο για την ενημέρωση των σημαιών κατάστασης του CPSR

2.3.2 Ολισθήσεις

Οι εντολές ολίσθησης έχουν τη γενική μορφή:

<opcode> <var1>, [<var2>]

Το <opcode> είναι το μνημονικό όνομα της εντολής, το <var1> αποτελεί το πρώτο έντελο και το <var2> το δεύτερο έντελο, το οποίο δεν είναι πάντα απαιτούμενο και γι'αυτό αναγράφεται μέσα σε [ ]. Το <var1> είναι συνήθως ο καταχωρητής στον οποίο θα αποθηκευτούν τα αποτελέσματα της εντολής. Το <var2> μπορεί να είναι καταχωρητής ή σταθερά τιμή[10].

Η αρχιτεκτονική του επεξεργαστή (δες *Διάγραμμα Αρχιτεκτονικής* (κεφ. 2.1, σελ. 6)) μεταφέρει το <var2> στη μονάδα που το χρειάζεται μέσω ενός προγραμματιζόμενου ολισθητή (barrel shifter), δίνοντάς μας έτσι τη δυνατότητα να εκτελέσουμε ολίσθηση στην τιμή του <var2> στον ίδιο κύκλο με την εκτέλεση της εντολής. Ο ολισθητής μας δέχεται έντελα μέγεθους 32 ψηφίων (δηλαδή όσο και το μέγεθος κάθε καταχωρητή) και η μέγιστη ολίσθηση που μπορεί να εκτελέσει είναι κατά 31 ψηφία. Οι πράξεις ολίσθησης που παρέχει η αρχιτεκτονική του επεξεργαστή μας περιγράφονται στον πίνακα *Ολισθήσεις* (κεφ. 2.3.2, σελ. 16).

Τύπος ολίσθησης	**Περιγραφή**
LSL	Λογική ολίσθηση προς τα αριστερά
LSR	Λογική ολίσθηση προς τα δεξιά
ASR	Αριθμητική ολίσθηση προς τα δεξιά
ROR	Κυκλική ολίσθηση προς τα δεξιά
RRX	Επαυξημένη κυκλική ολίσθηση προς τα δεξιά

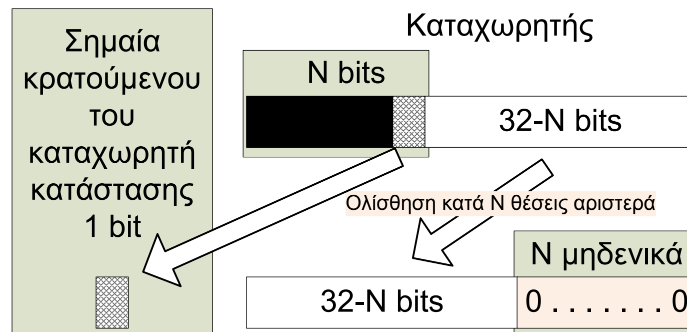
Πίνακας 2.6: Ολισθήσεις

Λογική ολίσθηση

Η λογική ολίσθηση προς τα αριστερά μεταφέρει τα ψηφία του καταχωρητή μας αριστερά και στις θέσεις των λιγότερο σημαντικών ψηφίων εισάγονται 0 (αν εκτελέσουμε ολίσθηση κατά n θέσεις θα εισαχθούν n μηδενικά). Τα n περισσότερο σημαντικά ψηφία που ολίσθησαν πέρα από την αριστερότερη θέση χάνονται, αλλά το λιγότερο σημαντικό από

[10]Οι σταθερές τιμές είναι αριθμοί που τους εισάγουμε με την εντολή χωρίς να τους έχουμε αποθηκεύσει νωρίτερα σε κάποια θέση μνήμης. Για να δηλώσουμε πως πρόκειται για σταθερά χρησιμοποιούμε το σύμβολο # πριν τον αριθμό χωρίς κενό ανάμεσα, πχ. #0x4 ή #25

αυτά αποθηκεύεται στη σημαία κατάστασης (Carry bit) του CPSR, όταν η ολίσθηση γίνεται για αριθμητική/λογική πράξη (όχι για διευθυνσιοδότηση). Για παράδειγμα, ο αριθμός 0x1035004A = 00010000 00110101 00000000 01001010 αν υποστεί ολίσθηση προς τα αριστερά κατά 4 θέσεις θα γίνει 00000011 01010000 00000100 10100000 = 0x035004A0. Στο bit κρατουμένου του καταχωρητή κατάστασης θα αποθηκευτεί το 1 (εικόνα *Ολίσθηση προς τα αριστερά* κεφ. 2.5, σελ. 17).

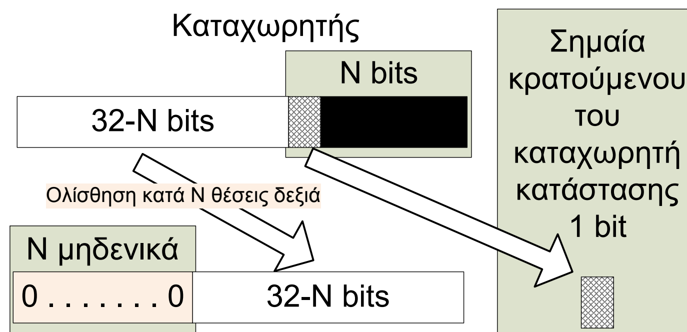


Σχήμα 2.5: Ολίσθηση προς τα αριστερά

Αν θεωρήσουμε τον δυαδικό αριθμό 0x49 = 0100_1001 = (4*16 + 9=73_{10}) με αναπαράσταση πολυωνύμου (δηλαδή $x_7 * 2^7 + x_6 * 2^6 + \ldots + x_0$, με $x_7 = 0, x_6 = 1, x_5 = 0, x_4 = 0, x_3 = 1, x_2 = 0, x_1 = 0, x_0 = 1$) θα διαπιστώσουμε πως η ολίσθηση κατά μια θέση προς τα αριστερά ισοδυναμεί με τη μετακίνηση της τιμής του κάθε x_i στο x_{i+1} (η τιμή του x_0 συμπληρώνεται με 0). Το αποτέλεσμα είναι 1001_0010 = 0x92 = (9*16 + 2 = 146_{10}), το οποίο είναι το διπλάσιο της αρχικής τιμής. Αυτό συμβαίνει διότι οι συντελεστές x_i πολλαπλασιάζονται με μια δύναμη του 2 και οι διαδοχικοί συντελεστές πολλαπλασιάζονται με διαδοχικές δυνάμεις του 2. Άρα, αν ο x_i συντελούσε κατά 2^i στην τιμή του αρχικού αριθμού, μετά την ολίσθηση θα συντελεί κατά 2^{i+1}, δηλαδή κατά το διπλάσιο. Όπως βλέπετε, διπλασιάζεται η συνεισφορά του κάθε συντελεστή στην τιμή του αριθμού και μετά την ολίσθηση έχουμε σαν αποτέλεσμα το διπλάσιο αριθμό του αρχικού. Αν εκτελέσουμε ολίσθηση κατά 2 θέσεις θα πάρουμε σαν αποτέλεσμα τον τετραπλάσιο αριθμό του αρχικού και στη γενική περίπτωση αν εκτελέσουμε αριστερή ολίσθηση κατά n θέσεις θα πάρουμε σαν αποτέλεσμα το $2^n \times$(#αρχικός αριθμός) (αριστερή ολίσθηση κατά 5_{10} θέσεις στον αριθμό 3_{10} θα δώσει σαν αποτέλεσμα το 3*2^5 = 3*32=96_{10}). Επειδή οι καταχωρητές του επεξεργαστή μας έχουν εύρος 32 ψηφίων, κατά τη διαδικασία της ολίσθησης θα αποκοπούν τα περισσότερο σημαντικά ψηφία και η τιμή που θα πάρουμε, λόγω του περιορισμού του εύρους, θα είναι mod($2^n \times$(#αρχικός αριθμός), 2^{32}), όπου το mod συμβολίζει το υπόλοιπο της διαίρεσης του $2^n \times$(#αρχικός αριθμός) με το 2^{32}. Η πράξη της αριστερής ολίσθησης

είναι πολύ χρήσιμη όταν θέλουμε να εκτελέσουμε πολλαπλασιασμό ανάμεσα σε κάποιον αριθμό με δύναμη του 2, γιατί εκτελείται σε ένα κύκλο ρολογιού, ενώ ο πολλαπλασιασμός που χρησιμοποιεί το κύκλωμα του πολλαπλασιαστή είναι αρκετά πιο χρονοβόρος και απαιτεί σημαντικά περισσότερους κύκλους.

Η δεξιά λογική ολίσθηση αντίστοιχα ισοδυναμεί με τη τη μετακίνηση της τιμής του κάθε x_i στο x_{i-1} (η τιμή του x_{31} συμπληρώνεται με 0). Τώρα, αν ο x_i συντελούσε κατά 2^i στην τιμή του αρχικού αριθμού, μετά την ολίσθηση θα συντελεί κατά 2^{i-1}, δηλαδή κατά το μισό. Επιπλέον, αποκόπτονται τα λιγότερα σημαντικά ψηφία και το αποτέλεσμα που παίρνουμε μετά την εκτέλεση της δεξιάς ολίσθησης είναι στρογγυλοποιημένο στον αμέσως μικρότερο ακέραιο του μισού της αρχικής τιμής. Στη γενική περίπτωση, η δεξιά ολίσθηση κατά n θέσεις θα παράγει το $\lfloor(\#(\text{αρχικός αριθμός})/2^n)\rfloor$ (δεξιά ολίσθηση κατά 3 θέσεις στον αριθμό 15 θα δώσει σαν αποτέλεσμα $\lfloor(15)/2^3)\rfloor = 1$). Η δεξιά ολίσθηση είναι πολύ χρήσιμη στην υλοποίηση διαίρεσης κάποιου αριθμού με δύναμη του 2, διότι ο επεξεργαστής μας δεν υποστηρίζει με υλικό την πράξη της διαίρεσης. Η υλοποίηση αυτής της πράξης με λογισμικό αυξάνει πολύ τον απαιτούμενο χρόνο ολοκλήρωσης της διαίρεσης. Γι' αυτό είναι πολύ προτιμότερο να γίνεται χρήση της παραπάνω ιδιότητας όπου είναι αυτό δυνατό (εικόνα *Ολίσθηση προς τα δεξιά* κεφ. 2.6, σελ. 18).

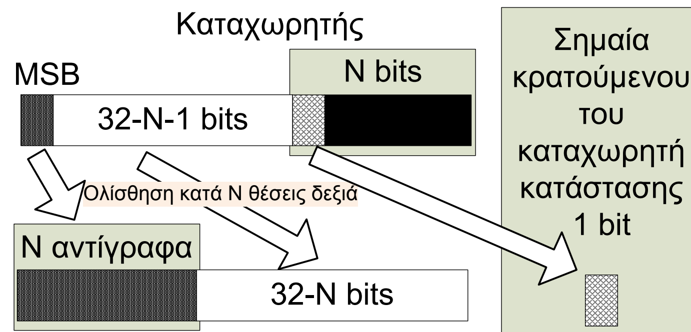


Σχήμα 2.6: Ολίσθηση προς τα δεξιά

Αριθμητική ολίσθηση

Όταν χρησιμοποιούμε αριθμητικές πράξεις με προσημασμένους αριθμούς, η δεξιά ολίσθηση δημιουργεί το πρόβλημα αλλαγής του προσήμου. Ενας αρνητικός αριθμός σε αριθμητική συμπληρώματος ως προς 2 έχει πάντα στο περισσότερο σημαντικό δυαδικό ψηφίο την τιμή 1 (πχ. ο -15_{10} σε αναπαράσταση 8 δυαδικών ψηφίων είναι 1111_0001).

Αν θέλουμε να εκτελέσουμε διαίρεση του -15_{10} με το 4_{10} περιμένουμε σαν αποτέλεσμα τον αριθμό -4_{10} (σε Ευκλείδια αναπαράσταση ισχύει ότι $-15_{10} = -4 * 4 + 1$, επειδή το υπόλοιπο είναι πάντα θετικός αριθμός), αλλά με την εφαρμογή δεξιάς ολίσθησης κατά 2 ψηφία (4 = 2^2) θα έχουμε σαν αποτέλεσμα τον αριθμό 0011_1100 = 60_{10}, το οποίο είναι λανθασμένο. Το λάθος αποτέλεσμα οφείλεται στο ότι δεν εισάγαμε 1 στα περισσότερο σημαντικά ψηφία. Αν το είχαμε προβλέψει, θα είχαμε σαν αποτέλεσμα τον αριθμό 1111_1100 = -4_{10}. Για να πάρουμε συνεπώς το επιθυμητό αποτέλεσμα χρησιμοποιούμε την αριθμητική δεξιά ολίσθηση, η οποία στη θέση των περισσότερο σημαντικών ψηφίων εισάγουν την τιμή του περισσότερο σημαντικού ψηφίου του αρχικού αριθμού. Ετσι, σε μια αριθμητική ολίσθηση N ψηφίων, αν το περισσότερο σημαντικό είναι 0 θα εισαχθούν N μηδενικά, ενώ αν είναι 1 θα εισαχθούν N άσσοι (εικόνα *Αριθμητική ολίσθηση προς τα δεξιά* κεφ. 2.7, σελ. 19).

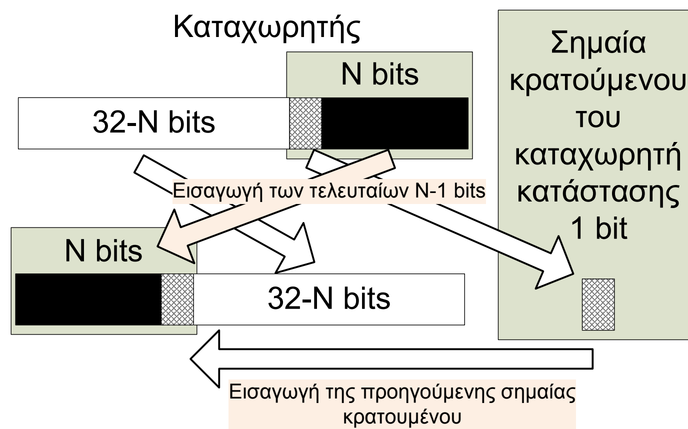


Σχήμα 2.7: Αριθμητική ολίσθηση προς τα δεξιά

Κυκλική ολίσθηση

Στις παραπάνω πράξεις, τα ψηφία που ολισθαίνουν πέρα από τα άκρα του καταχωρητή χάνονται, με εξαίρεση το τελευταίο που μπορεί να αποθηκευτεί σα σημαία κρατουμένου του CPSR, κάτω από συγκεκριμένες συνθήκες. Μερικές φορές είναι επιθυμητή η επανεισαγωγή των ψηφίων που ολισθαίνουν πέρα από τον καταχωρητή στις θέσεις που μένουν κενές. Για παράδειγμα, αν έχουμε τον αριθμό 0x97 = 1001_0111 και σα σημαία κρατουμένου του CPSR μια άγνωστη τιμή x, η κυκλική δεξιά ολίσθηση κατά 3 θέσεις θα έχει σαν αποτέλεσμα την τιμή **11x**1_0010, ενώ σα νέα σημαία κρατουμένου θα αποθηκευτεί το 1. Τα έντονα ψηφία είναι αυτά που επανεισήχθηκαν στον αριθμό, στις θέσεις όπου θα εισάγονταν 0 αν χρησιμοποιούσαμε λογική δεξιά ολίσθηση. Η αριστερή κυκλική ολίσθηση δεν υπάρχει σαν ξεχωριστή εντολή, διότι μπορεί να υλοποιηθεί μέσω της δεξιάς. Για αριστερή κυκλική ολίσθηση n θέσεων (με $n > 1$) εκτελούμε δεξιά κυκλική ολίσθηση κατά

$33-n$. Αν η ολίσθηση γίνεται για 1 θέση, τότε εκτελούμε την εντολή **ADCS** χρησιμοποιώντας ως έντελα δύο φορές τον καταχωρητή. Αν προσθέσουμε ένα αριθμό με τον εαυτό του στην ουσία τον διπλασιάζουμε ή τον ολισθαίνουμε λογικά κατά μια θέση αριστερά. Το κρατούμενο που δημιουργείται είναι το ίδιο με το περισσότερο σημαντικό bit του αρχικού αριθμού. Στο λιγότερο σημαντικό ψηφίο θα εμφανιστεί το 0, οπότε προσθέτοντας και το περιεχόμενο της σημαίας κρατουμένου στο άθροισμα, θα εμφανίσουμε εκεί το κρατούμενο που υπήρχε στον CPSR (Για παράδειγμα, αν η σημαία κρατουμένου είναι 1 και στον καταχωρητή υπάρχει η τιμή 0x49 = 0100_1001, το άθροισμα του 0x49 + 0x49 = 0x92 = 1001_0010, το κρατούμενο εξόδου είναι 0 και η πρόσθεση του 1 που υπήρχε ως σημαία κρατουμένου θα δώσει αποτέλεσμα 1001_001**1**) (εικόνα *Κυκλική Ολίσθηση προς τα δεξιά* κεφ. 2.8, σελ. 20).



Σχήμα 2.8: Κυκλική Ολίσθηση προς τα δεξιά

Η τελευταία μορφή ολίσθησης, η επαυξημένη δεξιά κυκλική ολίσθηση, δεν είναι τίποτα άλλο παρά μια κυκλική δεξιά ολίσθηση κατά μια θέση.

2.3.3 Εντολές μεταφοράς δεδομένων

Ο επεξεργαστής μας έχει τη δυνατότητα εκτέλεσης των εντολών μεταφοράς που αναφέρονται στον πίνακα *Εντολές Μεταφοράς* (κεφ. 2.3.3, σελ. 21).

Μεταφορά απλού καταχωρητή

Οι εντολές μεταφοράς δεδομένων αναλαμβάνουν να φέρουν δεδομένα από την κύρια μνήμη στους καταχωρητές, να μετακινήσουν δεδομένα ανάμεσα σε καταχωρητές και να στείλουν στην κύρια μνήμη το περιεχόμενο καταχωρητών. Κατά την εκτέλεση αυτών των εντολών δεν χρησιμοποιείται η ΑΛΜ, αλλά η Μονάδα Διασύνδεσης με την εξωτερική μνήμη.

Εντολή	**Περιγραφή**	**Εντολή**	**Περιγραφή**
LDM	Μεταφορά block από μνήμη σε καταχωρητές	**STM**	Μεταφορά block από καταχωρητές σε μνήμη
LDR	Μεταφορά από μνήμη σε καταχωρητή	**STR**	Μεταφορά από καταχωρητή σε μνήμη
MOV	Μεταφορά από καταχωρητή σε καταχωρητή	**MVN**	Μεταφορά με αντιστροφή
MRS	Μεταφορά από CPSR σε καταχωρητή	**MSR**	Μεταφορά από καταχωρητή σε CPSR
MRC	Μεταφορά από συνεπεξεργαστή σε καταχωρητή	**MCR**	Μεταφορά από καταχωρητή σε συνεπεξεργαστή
SWP	Ανταλλαγή τιμής ανάμεσα σε μνήμη και καταχωρητή		

Πίνακας 2.7: Εντολές Μεταφοράς

Η κάθε μεταφορά μπορεί να γίνει για δεδομένα εύρους 32 ψηφίων (word), 16 ψηφίων (halfword) ή 8 ψηφίων (byte). Για μεταφορές δεδομένων 16 ψηφίων χρησιμοποιείται το επίθεμα **H**, ενώ για μεταφορά byte το επίθεμα **B**. Για παράδειγμα οι εντολές **LDR, LDRH** και **LDRB** μεταφέρουν word, halfword και byte αντίστοιχα από κάποια εξωτερική μνήμη προς τους καταχωρητές.

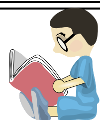
Στις εντολές μεταφοράς δεδομένων είναι απαραίτητο να ορίσουμε την πηγή των δεδομένων και τον προορισμό τους. Αν πρόκειται για μεταφορά από τη μνήμη προς καταχωρητή, τότε η μορφή είναι:

<Εντολή> Rd, [Rb, <offset>]

Ο καταχωρητής Rd (Register destination) είναι ένας από τους r0. . . r12, όπως και ο Rb (Register base). Στον Rd θα αποθηκευτεί το δεδομένο από την κύρια μνήμη. Ο Rb περιέχει τη διεύθυνση της κύριας μνήμης στην οποία βρίσκεται το δεδομένο. Για μεγαλύτερη ευελιξία, μπορούμε να χρησιμοποιήσουμε και το πεδίο <offset> (μετατόπιση) αν είναι επιθυμητό. Το offset ορίζει μια μετατόπιση η οποία προστίθεται στο περιεχόμενο του Rb. Η μορφή του <offset> είναι:

Καταχωρητής:	Μπορούμε να χρησιμοποιήσουμε οποιονδήποτε καταχωρητή από τους r0. . . r12 ([Rb, Rx])
Τιμή:	Χρησιμοποιούμε απευθείας τιμή ([Rb, #val]). Η τιμή μπορεί να είναι οποιοσδήποτε αριθμός από [0. . . 255] που να έχει δεχτεί ολίσθηση από [0. . . 31] ψηφίων (πχ. 0x35 ή 0x35 left-shifted(4) = 0x350)

Η χρήση της μετατόπισης μπορεί να μας βοηθήσει στην περίπτωση που προσπελαύνουμε στοιχεία ενός πίνακα. Ένας πίνακας ορίζεται σαν μια σειρά από διαδοχικά bytes. Αν γνωρίζουμε την διεύθυνση του πρώτου στοιχείου του πίνακα, τότε αποθηκεύουμε αυτή τη διεύθυνση στον καταχωρητή που χρησιμοποιούμε σαν Rb και στον Rx αποθηκεύουμε τη θέση του στοιχείου. Για παράδειγμα, αν ο πίνακάς μας ξεκινά από τη θέση 0xC0000380, το πρώτο στοιχείο του βρίσκεται στη θέση 0xC0000380 + 0, το πέμπτο στοιχείο στη θέση 0xC0000380 + 4 και το n στοιχείο στη θέση 0xC0000380 + $(n-1)$. Παρατηρήστε πως η αρίθμηση ξεκινά από το 0 και όχι από το 1! Ας υποθέσουμε τώρα πως κάθε στοιχείο του πίνακα δεν έχει μέγεθος ένα byte, αλλά 4 (δηλαδή πρόκειται για ένα word). Αυτό σημαίνει πως το επόμενο στοιχείο θα ξεκινά από τη θέση 0xC0000380 + 4 και το μεθεπόμενο από τη θέση 0xC0000380 + 8. Για παράδειγμα, αν στη θέση 0xC0000380 υπάρχει ο αριθμός 0xCAFEBABA και στην 0xC0000384 υπάρχει ο 0x12345678, εκτελώντας μεταφορά 4 bytes με Rb = 0xC0000380 και <offset> = 1 θα έχουμε σαν υποθετικό αποτέλεσμα τον αριθμό 0x78CAFEBA. Το υποθετικό σχετίζεται με το γεγονός πως ο επεξεργαστής δεν θα μας επιτρέψει να εκτελέσουμε προσπέλαση word στην κύρια μνήμη, αν η διεύθυνση δεν είναι ακέραιο πολλαπλάσιο του 4! Το ίδιο ισχύει και για halfword προσπελάσεις, μόνο που εκεί η διεύθυνση πρέπει να είναι ακέραιο πολλαπλάσιο του 2. Δηλαδή ο επεξεργαστής μας χρησιμοποιεί στοίχιση (alignment) των δεδομένων στη μνήμη.



Συμπληρωματικά . . .

Ο τρόπος αποθήκευσης δεδομένων μεγαλύτερων του ενός byte στη μνήμη μπορεί να ακολουθεί 2 τρόπους, οι οποίοι έχουν να κάνουν με τη σειρά των bytes. Εστω για παράδειγμα, ότι μια λέξη αποτελείται από 4 bytes και ότι τα bytes αυτά τοποθετούνται στη μνήμη διαδοχικά. Ποιο όμως θα τοποθετηθεί πρώτο και ποιο τελευταίο; Αν ακολουθούμε αρχιτεκτονική *Little Endian* στην αρχική διεύθυνση θα τοποθετηθεί το λιγότερο σημαντικό byte, στην αμέσως επόμενη το επόμενο byte και στην τέταρτη και τελευταία θα τοποθετηθεί το περισσότερο σημαντικό byte. Ας υποθέσουμε πως θέλουμε να αποθηκεύσουμε τον αριθμό 0xDEADBEEF στη θέση μνήμης 0xC000450. Στην 0x-C000450 θα τοποθετηθεί το 0xEF, στην 0xC000451 το 0xBE, στην 0xC000452 το 0xAD και στην 0xC000453 το 0xDE. Ο εναλλακτικός τρόπος αποθήκευσης ορίζει πως το πιο σημαντικό byte θα αποθηκευτεί στην πρώτη διεύθυνση και το λιγότερο σημαντικό στην τελευταία διεύθυνση. Δηλαδή, στην 0xC000450 θα τοποθετηθεί το 0xDE, στην 0xC000451 το 0xAD, στην 0xC000452 το 0xBE και στην 0xC000453 το 0xEF. Η αρχιτεκτονική αυτή ονομάζεται *Big Endian*. Η σειρά των bytes ισχύει και για την περίπτωση των halfwords, μόνο που εκεί έχουμε 2 bytes (σε αυτή την περίπτωση ο αριθμός 0xDEAD θα αποθηκευτεί με το 0xAD στην 0xC000450 και το 0xDE στην 0xC000451 με *Little Endian* αρχιτεκτονική). Ο επεξεργαστής ARM υποστηρίζει και τους δυο τρόπους αποθήκευσης, αλλά το λειτουργικό σύστημα Linux υιοθετεί την *Little Endian* αρχιτεκτονική, οπότε όλα τα προγράμματά μας θα πρέπει να ακολουθούν κανόνες *Little Endian*.

Έτσι λοιπόν, αν στο πρόγραμμά μας έχουμε χρησιμοποιήσει έναν καταχωρητή για να περιέχει τη διεύθυνση του πρώτου στοιχείου του πίνακα, έναν δεύτερο καταχωρητή για να κρατά τη θέση του στοιχείου που θέλουμε να προσπελάσουμε και το μέγεθος κάθε στοιχείου είναι ίσο με μια δύναμη του 2 (πχ. 2, 4, 8, 16...), τότε ή θα αυξάνουμε τον καταχωρητή της θέσης κατά το μέγεθος του κάθε στοιχείου (δηλαδή αν πρόκειται για το 3ο στοιχείο και κάθε στοιχείο έχει μέγεθος 8, στον καταχωρητή θέσης θα έχουμε (3-1)*8 = 16), ή θα χρησιμοποιήσουμε τη δυνατότητα του επεξεργαστή να εκτελεί ολίσθηση στα δεδομένα του καταχωρητή θέσης, πριν αυτά προστεθούν στον καταχωρητή βάσης και στον καταχωρητή θέσης θα έχουμε τη σειρά του στοιχείου (0 για το πρώτο, 1 για το δεύτερο...).

Η διαδικασία αποστολής δεδομένων προς την κύρια μνήμη είναι ακριβώς ίδια με την διαδικασία αποθήκευσης δεδομένων στους καταχωρητές, εκτός από τη σημασία του καταχωρητή Rd, όπου τώρα περιέχει την τιμή η οποία θα αποσταλεί προς την κύρια μνήμη.

Μετά από την προσπέλαση της κύριας μνήμης (είτε για αποθήκευση, είτε για ανάκτηση δεδομένων), πολλές φορές είναι επιθυμητή η τροποποίηση του Rb, ώστε να περιέχει την διεύθυνση όπου έγινε η τελευταία προσπέλαση (αν τον χρησιμοποιήσουμε για επόμενη προσπέλαση μνήμης, αυτή θα γίνει με βάση τη θέση που έγινε η τελευταία προσπέλαση). Αυτό μπορεί να γίνει αν χρησιμοποιήσουμε μια πράξη πρόσθεσης (μετά από την προσπέλαση μνήμης) ανάμεσα στον Rb και τη μετατόπιση. Μιας και αυτή η αντιμετώπιση απαιτεί επιπλέον κύκλους ρολογιού για τη πράξης της πρόσθεσης, η αρχιτεκτονική του επεξεργαστή μας είναι φτιαγμένη έτσι ώστε να μας επιτρέπεται η αλλαγή της τιμής του Rb κατά το κύκλο προσπέλασης στην κύρια μνήμη. Επιπλέον, μας δίνει τη δυνατότητα να επιλέξουμε αν θέλουμε να γίνει η τροποποίηση πριν ή μετά την προσπέλαση. Για να γίνει η τροποποίηση πριν την προσπέλαση τοποθετούμε το σύμβολο ! στο τέλος της εντολής (πχ. LDR R5, [R0, R2, LSL #2]! όπου θα γίνει η αύξηση του R0 κατά R2*4, το αποτέλεσμα θα αποθηκευτεί στον R0 και μετά θα διευθυνσιοδοτηθεί η κύρια μνήμη). Για να γίνει η τροποποίηση μετά από την προσπέλαση στην κύρια μνήμη αλλάζουμε λίγο την σημειογραφία της εντολής και περικλείουμε στα σύμβολα [ ] μόνο τον Rb, ενώ την μετατόπιση την προσθέτουμε μετά (πχ. η εντολή LDR R5, [R0], R2, LSL #2 θα διευθυνσιοδοτήσει την κύρια μνήμη με την τιμή που περιέχει ο R0 και στη συνέχεια θα αυξήσει τον R0 κατά R2*4. Παρατηρείστε πως εδώ δεν χρειάζεται το !). Παραδείγματα μεταφοράς δεδομένων από την μνήμη στους καταχωρητές υπάρχουν στον πίνακα *Data Transfer* (κεφ. 2.3.3, σελ. 24).

Υποθέστε ότι το περιεχόμενο του R1 είναι 0x8000 και του R5 είναι 0x6.

Εντολή	**Περιγραφή**
LDR R0, [R1]	Μεταφέρει το περιεχόμενο της θέσης μνήμης με διεύθυνση 0x8000 στον καταχωρητή R0.
LDR R0, [R1, R5]	Μεταφέρει το περιεχόμενο της θέσης μνήμης με διεύθυνση 0x8000 + 6 = 0x8006 στον καταχωρητή R0.
LDR R0, [R1, #10]	Μεταφέρει το περιεχόμενο της θέσης μνήμης με διεύθυνση 0x8000 + 0xA = 0x800A στον καταχωρητή R0.
LDR R0, [R1, R5, LSL #2]	Μεταφέρει το περιεχόμενο της θέσης μνήμης με διεύθυνση 0x8000 + 6*4 = 0x8018 στον καταχωρητή R0.
LDR R0, [R1, R5, LSL #4]!	Μεταφέρει το περιεχόμενο της θέσης μνήμης με διεύθυνση 0x8000 + 6*16 = 0x8060 στον καταχωρητή R0 και αποθηκεύει την τιμή 0x8060 στον R1.
LDR R0, [R1], #12	Μεταφέρει το περιεχόμενο της θέσης μνήμης με διεύθυνση 0x8000 στον καταχωρητή R0 και αποθηκεύει την τιμή 0x8000 + 12 = 0x800C στον R1.
STR R0, [R1]	Μεταφέρει στη θέση μνήμης με διεύθυνση 0x8000 το περιεχόμενο του καταχωρητή R0.
STR R0, [R1, #20]	Μεταφέρει στη θέση μνήμης με διεύθυνση 0x8000 + 20 = 0x8014 το περιεχόμενο του καταχωρητή R0.

Πίνακας 2.8: Data Transfer

Μεταφορά block καταχωρητών

Μέχρι τώρα η μεταφορά δεδομένων από τους καταχωρητές προς τη μνήμη και αντίστροφα αφορούσε δεδομένα ενός καταχωρητή. Στο AT91, υπάρχει επίσης η δυνατότητα να μεταφέρουμε προς την κύρια μνήμη όσους καταχωρητές θέλουμε με τη χρήση της εντολής **STM** (όπως και από τη μνήμη προς τους καταχωρητές με την **LDM**). Οι εντολές αυτές μπορούν να χρησιμοποιηθούν σε περιπτώσεις μεταφοράς ομάδας block δεδομένων και είναι ταχύτερες από την μεταφορά των δεδομένων ένα-προς-ένα με τη χρήση των εντολών **LDR, STR**, διότι εκμεταλλεύονται την αρχιτεκτονική του επεξεργαστή και χρησιμοποιούν την αρτηρία δεδομένων κατά ριπάς (burst mode[11]). Πρέπει να σημειώσουμε πως σε αυτό το τρόπο μεταφοράς υποστηρίζονται μόνο μεταφορές words. Μια επιπλέον χρήσιμη εφαρμογή των εντολών μεταφοράς block εμφανίζεται κατά την κλήση συναρτήσεων. Με το που ξεκινά η εκτέλεση της συνάρτησής μας, για να έχουμε τη δυνατότητα να χρησιμοποιήσουμε οποιονδήποτε καταχωρητή επιθυμούμε χωρίς να χαθεί το περιεχόμενό του, πρέπει να τον αποθηκεύσουμε στην κύρια μνήμη. Μόλις ολοκληρωθεί η εκτέλεση του κώδικα της συνάρτησής μας και αμέσως πριν εκτελέσουμε την εντολή επιστροφής στο σημείο κλήσης της συνάρτησης, καλούμε την εντολή επαναφοράς των καταχωρητών από την κύρια μνήμη, οπότε με αυτό τον τρόπο τα δεδομένα των καταχωρητών έχουν παραμείνει άθικτα. Η γενική μορφή της εντολής είναι:

<Εντολή><Mode> Rb, {<registers>}

Ο καταχωρητής που περιέχει την αρχική διεύθυνση της κύριας μνήμης είναι ο Rb. Οι καταχωρητές αναγράφονται μέσα σε {} και χωρίζονται μεταξύ τους με κόμμα ({r0, r5-r8, r15}). Αν υπάρχει μια σειρά από διαδοχικούς καταχωρητές που θα συμμετάσχουν στην μεταφορά, τότε αντί να αναγράψουμε τον καθένα, μπορούμε να σημειώσουμε τον πρώτο, μετά μια παύλα και μετά τον τελευταίο (όπως r5-r8). Σε κάθε κύκλο εκτέλεσής της η εντολή αλλάζει την διεύθυνση της κύριας μνήμης, ανάλογα με την επιλογή προσπέλασης. Ο επεξεργαστής μας δίνει 4 διαφορετικές επιλογές (Modes) στον τρόπο προσπέλασης της κύριας μνήμης :

- **Increment Before (IB)** Πριν γίνει η προσπέλαση στη μνήμη, η διεύθυνση προσπέλασης αυξάνεται κατά 4.
- **Increment After (IA)** Η διεύθυνση προσπέλασης αυξάνεται κατά 4 αφού γίνει η προσπέλαση στη μνήμη.

[11] Burst mode έχουμε όταν ο επεξεργαστής προωθεί τα δεδομένα στην αρτηρία δεδομένων σε διαδοχικούς κύκλους ρολογιού έχοντας αποκλείσει διακοπή από άλλο περιφερειακό που θα ήθελε τη χρήση της αρτηρίας για να εκτελέσει δική του μεταφορά.

- **Decrement Before (DB)** Πριν γίνει η προσπέλαση στη μνήμη, η διεύθυνση προσπέλασης μειώνεται κατά 4.

- **Decrement After (DA)** Η διεύθυνση προσπέλασης μειώνεται κατά 4 αφού γίνει η προσπέλαση στη μνήμη.

Οπως και με τις εντολές απλής προσπέλασης της κύριας μνήμης, έτσι και με τις εντολές μεταφοράς block, η αρχιτεκτονική του επεξεργαστή δίνει τη δυνατότητα να τροποποιηθεί το περιεχόμενο του καταχωρητή βάσης (Rb) με την τελευταία διεύθυνση που υπολογίζει η εντολή μας (αν πρόκειται για προσπελάσεις Increment η τιμή του Rb θα γίνει Rb + 4 * (#αριθμός καταχωρητών), αν πρόκειται για Decrement η τιμή του Rb θα γίνει Rb - 4 * (#αριθμός καταχωρητών). Η τροποποίηση ενεργοποιείται με την αναγραφή του συμβόλου ! δίπλα στον καταχωρητή βάσης. *Ιδιαίτερη προσοχή απαιτείται από τον προγραμματιστή στην περίπτωση που ο Rb συμπεριλαμβάνεται και στη λίστα καταχωρητών που μεταφέρονται, διότι αν χρησιμοποιήσουμε ανανέωση του Rb το τελικό περιεχόμενο θα είναι άγνωστο (εξαρτάται από το αν θα επιδράσει τελευταία η αλλαγή λόγω μεταφοράς ή η αλλαγή λόγω ανανέωσης).* Παραδείγματα υπάρχουν στον πίνακα Multiple Data Transfer (κεφ. 2.3.3, σελ. 26).

Το περιεχόμενο του R1 είναι 0x8000.

Εντολή	**Περιγραφή**
LDMIB R1, {R0, R2-R5}	Θα ξεκινήσει από την θέση 0x8004 και θα τοποθετήσει το περιεχόμενο της στον R0, το περιεχόμενο της 0x8008 στον R2 κλπ.
LDMDA R1, {R0, R2-R5}	Θα ξεκινήσει από την θέση 0x8000 και θα τοποθετήσει το περιεχόμενο της στον R5, το περιεχόμενο της 0x7FFC στον R4 και τέλος το περιεχόμενο της 0x7FF0 στον R0.
LDMDA R1!, {R0, R2-R5}	Θα εκτελέσει την ίδια διαδικασία με την προηγούμενη εντολή, αλλά τώρα η τελευταία υπολογισθείσα διεύθυνση θα αποθηκευτεί στον R1 (0x8000 - 4*5 = 0x7FEC).
STMIA R1, {R0, R2-R5}	Θα τοποθετήσει στο 0x8000 το περιεχόμενο του R0, στο 0x8004 το περιεχόμενο του R2 κλπ.
STMDB R1, {R0, R2-R5}	Θα τοποθετήσει το περιεχόμενο του R5 στο 0x7FFC, το περιεχόμενο του R4 στο 0x7FF8 κλπ.

Πίνακας 2.9: Multiple Data Transfer

Παρατηρείστε πως η ανάκτηση των δεδομένων ακολουθεί την ακριβώς αντίθετη επιλογή προσπέλασης από την αποστολή προς την κύρια μνήμη. Δηλαδή, μετά από μια εντολή **STMDA**, για να γίνει σωστά η ανάκτηση των δεδομένων πρέπει να χρησιμοποιήσουμε την **LDMIB**, όπως και μετά από μια **STMIA** πρέπει να χρησιμοποιήσουμε την **LDMDB**.

Μεταφορά καταχωρητή σε καταχωρητή

Οταν θέλουμε να μεταφέρουμε την τιμή ενός καταχωρητή σε άλλον ή να θέσουμε άμεσα την τιμή ενός καταχωρητή, χρησιμοποιούμε την εντολή **MOV**, της οποίας η γενική μορφή είναι **MOV Rd, <operand>**. Ο καταχωρητής Rd είναι αυτός που θα δεχθεί την τιμή του <operand>, το οποίο μπορεί να είναι οποιοσδήποτε καταχωρητής, ή καταχωρητής με εντολή ολίσθησης ή άμεση τιμή. Χρειάζεται προσοχή σε ότι αφορά τις άμεσες τιμές που μπορούμε να εισάγουμε, αφού λόγω της αρχιτεκτονικής του επεξεργαστή μας τα δεδομένα της τιμής μπορούν να καταλάβουν μόνο 8 ψηφία μέσα στην εντολή (δηλαδή η μέγιστη επιτρεπτή τιμή είναι το 255 ή 0xFF). Επειδή η τιμή περνά μέσω του ολισθητή μπορούμε να εισάγουμε όποιον αριθμό θέλουμε (μέχρι το 255) ο οποίος να έχει δεχθεί ολίσθηση μέχρι 31 θέσεις. Για παράδειγμα, οι τιμές 0xFF, 0xDA0000, 0xF0000000 είναι αποδεκτές γιατί έχουν προέλθει από τις 0xFF χωρίς ολίσθηση, 0xDA ολισθημένο αριστερά κατά 16 θέσεις και 0xF0 ολισθημένο αριστερά κατά 24 θέσεις αντίστοιχα. Οι τιμές 0xDA0010, 0xF0003200, 0x254 δεν μπορούν να εισαχθούν άμεσα, γιατί δεν αποτελούν γινόμενο (8ψήφιου δυαδικού αριθμού) * (δύναμη του 2). Με την εντολή **MVN** εκτελούμε ότι και με την εντολή **MOV**, αλλά στον καταχωρητή Rd αποθηκεύεται η αντίθετη τιμή (δηλαδή όλα τα bits που είναι 1 γίνονται 0 και το αντίθετο). Παραδείγματα υπάρχουν στον πίνακα Register to Register Transfer (κεφ. 2.3.3, σελ. 27).

Εντολή	**Περιγραφή**
MOV R1, R2	Μεταφέρει στον R1 το περιεχόμενο του R2.
MOV R1, R2, LSL #5	Μεταφέρει στον R1 το περιεχόμενο του R2, ολισθημένο αριστερά κατά 5 θέσεις.
MOVS R1, R2, LSL #5	Εκτελεί την ίδια διαδικασία με την προηγούμενη εντολή, αλλά τώρα αποθηκεύεται στο Carry bit του CPSR το τελευταίο bit που ολίσθησε εκτός του καταχωρητή.
MOV R1, #0x35	Μεταφέρει στον R1 την τιμή #0x35.
MVN R1, #0x35	Μεταφέρει στον R1 την τιμή #0xFFFF_FFCA.

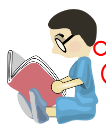
Πίνακας 2.10: Register to Register Transfer

Μεταφορές ειδικού τύπου

Στις εντολές μεταφοράς εντάσσονται και μερικές εντολές που έχουν σχέση με μεταφορά δεδομένων ανάμεσα στους καταχωρητές και ειδικές μονάδες του επεξεργαστή, όπως ο CPSR και ο συνεπεξεργαστής (CoProcessor). Όσον αφορά στον CPSR χρειάζεται λίγη προσοχή, διότι ενώ μπορούμε να αντιγράψουμε το περιεχόμενό του σε έναν καταχωρητή, δεν μπορούμε να αλλάξουμε το περιεχόμενο των bits [23-0] όταν βρισκόμαστε σε User mode. Επιπλέον, όταν θέλουμε να αποθηκεύσουμε στον CPSR διαφορετικές τιμές στις σημαίες κατάστασης πρέπει να χρησιμοποιούμε την γενική μορφή **MSR CPSR_f, Rm**, όπου το σύμβολο _f υποδηλώνει πως θέλουμε να αλλάξουμε μόνο τις σημαίες. Ο καταχωρητής Rm περιέχει την τιμή που θα σταλεί στον CPSR. Για παράδειγμα, αν το περιεχόμενο του R0 είναι 0x80000000 και εκτελέσουμε την εντολή:

MSR CPSR_f, R0

θα γίνει 1 η σημαία αρνητικού αποτελέσματος (Negative bit) και 0 οι υπόλοιπες (bits Z, C, V, Q).



Επεξήγηση

Ο συνεπεξεργαστής (CoProcessor) αποτελεί έναν υποσχεδιασμό του ολοκληρωμένου του επεξεργαστή. Είναι υπεύθυνος για των έλεγχο διαφόρων μονάδων, όπως η μονάδα διαχείρισης μνήμης (Memory Management Unit - MMU), η μονάδα προστασίας της μνήμης διαφορετικών διεργασιών (Protection Unit) και έχει το δικό του σύνολο από καταχωρητές.

Τέλος, η εντολή **SWP** αναλαμβάνει να αντιμεταθέσει το περιεχόμενο μιας θέσης μνήμης με ένα καταχωρητή. Η γενική της μορφή είναι **SWP Rd, Rm, [Rb]**, όπου ο Rd είναι ο καταχωρητής στον οποίο θα αποθηκευτεί το περιεχόμενο της θέσης μνήμης, ο Rm περιέχει την τιμή που θα σταλεί στη μνήμη και ο Rb περιέχει τη διεύθυνση της κύριας μνήμης, με την οποία θα γίνει η ανταλλαγή δεδομένων.

2.3.4 Αριθμητικές/Λογικές πράξεις

Ο επεξεργαστής μας έχει τη δυνατότητα εκτέλεσης των αριθμητικών/λογικών πράξεων που αναφέρονται στον πίνακα *Πράξεις* (κεφ. 2.3.4, σελ. 29). Εδώ πρέπει να τονίσουμε πως ο επεξεργαστής μας υλοποιεί με υλικό (hardware) μόνο πρόσθεση και πολλαπλασιασμό ακεραίων. Οι υπόλοιπες πράξεις ακεραίων καθώς και πράξεις κινητής υποδιαστολής μπορούν αν είναι αναγκαίες σε μια εφαρμογή να υλοποιηθούν με λογισμικό (software).

Η γενική μορφή των αριθμητικών/λογικών πράξεων είναι <**Εντολή**> **Rd, Rn,** <**par1**>, όπου Rd είναι ο καταχωρητής που θα δεχτεί το αποτέλεσμα της πράξης, Rn είναι ο

Εντολή	**Περιγραφή**	**Εντολή**	**Περιγραφή**
ADC	Πρόσθεση με κρατούμενο	**ADD**	Πρόσθεση
AND	Λογικό AND	**BIC**	Λογικό AND με το λογικό συμπλήρωμα (NOT) του 2^{ou} εντέλου
CLZ	Μέτρηση των 0 στις περισσότερο σημαντικές θέσεις	**CMN**	Σύγκριση με το $-(2^o$ έντελο)
CMP	Σύγκριση	**EOR**	Λογικό XOR
ORR	Λογικό OR	**RSB**	Αφαίρεση (αφαιρέτης - μειωτέος)
RSC	Αφαίρεση (αφαιρέτης - μειωτέος) με δανεικό	**SBC**	Αφαίρεση με δανεικό
SUB	Αφαίρεση	**TEQ**	Έλεγχος ισότητας
TST	Έλεγχος	**MLA**	Πολλαπλασιασμός και συσσώρευση
MUL	Πολλαπλασιασμός	**SMLAL**	Προσημασμένος πολλαπλασιασμός και συσσώρευση 32bits
SMULL	Προσημασμένος πολλαπλασιασμός 32bits	**UMLAL**	Μη-προσημασμένος πολλαπλασιασμός και συσσώρευση 32bits
UMULL	Μη-προσημασμένος πολλαπλασιασμός 32bits		

Πίνακας 2.11: Πράξεις

καταχωρητής που περιέχει το πρώτο έντελο της πράξης και <par1> μπορεί να είναι καταχωρητής, καταχωρητής που έχει δεχτεί ολίσθηση ή άμεση τιμή. Οι πράξεις αυτές, αν έχουν ενεργοποιημένη την επιλογή ανανέωσης του CPSR, αλλάζουν τις σημαίες κατάστασης ανάλογα με το αποτέλεσμα που προκύπτει. Εξαίρεση αποτελεί η εντολή **CLZ** που μετρά τα 0 που υπάρχουν στα περισσότερο σημαντικά ψηφία του Rn, μέχρι να συναντήσει ένα ψηφίο ίσο με 1 και αποθηκεύει τον αριθμό τους στον Rd (δεν χρησιμοποιεί το πεδίο <par1>). Παραδείγματα παρατίθενται στον πίνακα Αριθμητικές/Λογικές πράξεις (κεφ. 2.3.4, σελ. 30).

Οι εντολές σύγκρισης **CMP, CMN, TEQ & TST** εκτελούν τις αριθμητικές πράξεις **SUB, ADD, EOR, AND** αντίστοιχα χωρίς να αποθηκεύεται το αποτέλεσμά τους. Η εντολή **CM-**

Εντολή	**Περιγραφή**
ADD R1, R1, #0x10	Προσθέτει στον καταχωρητή R1 την τιμή 0x10.
SUB R5, R1, R2, LSL #2	Αφαιρεί από τον R1 το R2*4 και αποθηκεύει το αποτέλεσμα στον R5
RSB R5, R1, R2, LSL #2	Αφαιρεί από το R2*4 τον R1 και αποθηκεύει το αποτέλεσμα στον R5
ADC R5, R1, R2	Προσθέτει τον R1 με τον R2 συνυπολογίζοντας και τη σημαία κρατουμένου του CPSR και αποθηκεύει το αποτέλεσμα στον R5.
AND R1, R1, #0x10	Εκτελεί λογική πράξη AND ανάμεσα στον R1 και την άμεση τιμή 0x10 και αποθηκεύει το αποτέλεσμα στον R1.

Πίνακας 2.12: Αριθμητικές/Λογικές πράξεις

P ανανεώνει πάντα τις σημαίες κατάστασης του CPSR ενώ οι υπόλοιπες τρεις εντολές σύγκρισης μόνο αν υπάρχει η επιλογή S στο τέλος της εντολής. Η γενική μορφή τους είναι <**Εντολή**> **Rn**, <**par1**>, όπου Rn είναι ο καταχωρητής που περιέχει την πρώτη τιμή προς σύγκριση και το <par1> μπορεί να είναι καταχωρητής, καταχωρητής που έχει δεχτεί ολίσθηση ή άμεση τιμή. Για παράδειγμα, αν θέλουμε να εξακριβώσουμε αν ένας αριθμός είναι μεγαλύτερος από τον άλλο, εκτελούμε αφαίρεση μεταξύ τους και με βάση τις σημαίες κατάστασης που προκύπτουν ελέγχουμε αν το αποτέλεσμα ήταν αρνητικό, μηδέν ή θετικό. Παραδείγματα συγκρίσεων υπάρχουν στον πίνακα *Συγκρίσεις* (κεφ. 2.3.4, σελ. 31).

Οι εντολές πολλαπλασιασμού υποστηρίζονται από τον πολλαπλασιαστή της αρχιτεκτονικής μας, ο οποίος είναι υλοποιημένος σε υλικό. Παρότι δέχεται δεδομένα εισόδου εύρους 32 ψηφίων, μπορεί να να παράγουμε αποτελέσματα 32 ή 64 ψηφίων. Ο πλήρης πολλαπλασιασμός δύο δυαδικών αριθμών των 32 ψηφίων θα παράγει γινόμενο εύρους 64 ψηφίων, το οποίο μιας και δεν χωρά στο εύρος των καταχωρητών μας, αποθηκεύεται σε 2 καταχωρητές. Ωστόσο είναι πιθανό να μας ενδιαφέρουν μόνο τα 32 λιγότερα σημαντικά ψηφία του αποτελέσματος. Για παράδειγμα, αν θέλουμε να υπολογίσουμε το γινόμενο 5 x 6, δεν χρειαζόμαστε και τα 64 ψηφία, διότι το γινόμενο χωρά σε 32 ψηφία μόνο. Ο πολλαπλασιαστής μας είναι ικανός να μας δώσει τα 32 λιγότερο σημαντικά ψηφία ενός πολλαπλασιασμού εντέλων των 32 ψηφίων σε λιγότερους κύκλους απ' ότι απαιτεί ένας πλήρης πολλαπλασιασμός.

Εντολή	Περιγραφή
CMP R2, R4	Συγκρίνει το περιεχόμενο του R2 με το περιεχόμενο του R4. Ανάλογα με το είδος του αποτελέσματος ανανεώνονται οι σημαίες κατάστασης.
CMP R3, 0x8000	Συγκρίνει το περιεχόμενο του R3 με το δεδομένο 0x8000. Ανάλογα με το είδος του αποτελέσματος ανανεώνονται οι σημαίες κατάστασης.
CMN R1, R2	Συγκρίνει το περιεχόμενο του R1 με το -R2 (προσθέτει το συμπλήρωμα ως προς 2 του R2) και δεν επηρεάζει τις σημαίες κατάστασης.
TEQS R1, R4	Συγκρίνει ως προς την ισότητα το περιεχόμενο του R1 με το περιεχόμενο του R4. Αν τα περιεχόμενα των καταχωρητών είναι ίδια, η σημαία μηδενικού αποτελέσματος του CPSR γίνεται 1.

Πίνακας 2.13: Συγκρίσεις

Στο σύνολο εντολών μας υποστηρίζονται όλοι αυτοί οι εναλλακτικοί τρόποι χρήσης του πολλαπλασιαστή. Η εντολή **MUL** πολλαπλασιάζει 2 καταχωρητές και αποθηκεύει τα 32 λιγότερα σημαντικά ψηφία. Η εντολή **MLA** πολλαπλασιάζει 2 καταχωρητές και προσθέτει το γινόμενό τους με το περιεχόμενο ενός τρίτου πριν αποθηκεύσει τα 32 λιγότερα σημαντικά ψηφία. Οι δύο αυτές εντολές πολλαπλασιάζουν προσημασμένα και μη προσημασμένα δεδομένα. Οι υπόλοιπες εντολές πολλαπλασιασμού παράγουν γινόμενα εύρους 64 ψηφίων και διαχωρίζονται σε εντολές πολλαπλασιασμού προσημασμένων και μη προσημασμένων δεδομένων. Οι εντολές **SMULL & SMLAL** εκτελούν τις παραπάνω διαδικασίες (πολλαπλασιασμό και πολλαπλασιασμό με κατοπινή άθροιση - συσσώρευση αντίστοιχα), για προσημασμένα δεδομένα. Οι εντολές **UMULL & UMLAL** εκτελούν τις παραπάνω διαδικασίες αντίστοιχα για μη-προσημασμένα δεδομένα. Σε όλες τις περιπτώσεις το αποτέλεσμα είναι 64 ψηφίων. Παραδείγματα πολλαπλασιασμών υπάρχουν στον πίνακα *Πολλαπλασιασμοί* (κεφ. 2.3.4, σελ. 32)

2.3.5 Εντολές διακλάδωσης

Ο επεξεργαστής μας έχει τη δυνατότητα εκτέλεσης των εντολών διακλάδωσης που αναφέρονται στον πίνακα *Branches* (κεφ. 2.3.5, σελ. 32).

Ο επεξεργαστής μας υποστηρίζει μια σειρά από εντολές διακλάδωσης, που μπορούν να εκτελέσουν αλλαγή της ροής του κώδικα υπό συνθήκη κατά ±32MBytes. Η αλλαγή της ροής εκτέλεσης σε διαφορετικό σημείο του κώδικα σχετίζεται με την αλλαγή του περιεχομέ-

Εντολή	**Περιγραφή**
MUL R1, R2, R5	Πολλαπλασιάζει το περιεχόμενο του R2 με αυτό του R5 και αποθηκεύει τα 32 λιγότερο σημαντικά bits στον καταχωρητή R1.
MLA R1, R1, R4, R6	Πολλαπλασιάζει το περιεχόμενο του R1 με αυτό του R4, προσθέτει στο γινόμενο το περιεχόμενο του R6 και αποθηκεύει τα 32 λιγότερο σημαντικά bits στον καταχωρητή R1.
SMULL R0, R1, R4, R6	Πολλαπλασιάζει το περιεχόμενο του R4 με αυτό του R6 με προσημασμένη αριθμητική και αποθηκεύει τα 32 λιγότερο σημαντικά bits στον καταχωρητή R0 και τα 32 περισσότερα σημαντικά bits στον R1.
UMLAL R1, R2, R4, R6	Πολλαπλασιάζει το περιεχόμενο του R4 με αυτό του R6 με μη-προσημασμένη αριθμητική, προσθέτει στο γινόμενο την τιμή R1 + R2 * 2^{32} και αποθηκεύει τα 32 λιγότερο σημαντικά bits στον καταχωρητή R1 και τα 32 περισσότερα σημαντικά bits στον R2.

Πίνακας 2.14: Πολλαπλασιασμοί

Εντολή	**Περιγραφή**	**Εντολή**	**Περιγραφή**
B	Διακλάδωση	**BL**	Διακλάδωση με αποθήκευση του R15 στον R14
BX	Διακλάδωση με αλλαγή συνόλου εντολών	**BLX**	Διακλάδωση με αποθήκευση του R15 στον R14 και αλλαγή συνόλου εντολών

Πίνακας 2.15: Branches

νου του καταχωρητή R15 (PC). Οι εντολές διακλάδωσης έχουν τη γενική μορφή <**Εντολή**> <**label**>, όπου <label> είναι μια ετικέτα που σηματοδοτεί τη διεύθυνση της εντολής του άλματος (περισσότερες πληροφορίες κεφ. 3.2.1, σελ. 39).

Εισαγωγή στην Assembly του AT91

3.1 Γενικά

Ένα πρόγραμμα ορίζεται σαν μια σειρά από εντολές που εκτελούνται διαδοχικά και καθοδηγούν τον επεξεργαστή στην υλοποίηση μιας διεργασίας. Κάθε εντολή περιγράφει και ένα μικρό βήμα, απαραίτητο για την ολοκλήρωση της διεργασίας. Για παράδειγμα, για να υλοποιηθεί μια πρόσθεση είναι απαραίτητο να μεταφερθούν τα δεδομένα από την εξωτερική μνήμη στους καταχωρητές, να εκτελεστεί η πράξη και το αποτέλεσμα να σταλεί στην εξωτερική μνήμη για αποθήκευση. Χρειαζόμαστε δηλαδή:

1. **2 εντολές για τη μεταφορά των δεδομένων**
2. **1 εντολή για την πρόσθεση**
3. **1 εντολή για την αποθήκευση του αποτελέσματος**

Συνολικά 4 εντολές

Η υλοποίηση οποιασδήποτε διεργασίας πρέπει να διασπαστεί σε πολύ απλά βήματα, τα οποία θα εκτελούν μια από τις παρακάτω λειτουργίες:

1. Μεταφορά δεδομένων από και προς τη μνήμη
2. Επεξεργασία των δεδομένων που υπάρχουν στους καταχωρητές
3. Αλλαγή της ροής του προγράμματος με διακλαδώσεις

Οι εντολές αυτές γίνονται αντιληπτές από τον επεξεργαστή σαν μια αλληλουχία από 0 και 1 (πχ. η εντολή πρόσθεσης μπορεί να μοιάζει με 0111100111101111). Αυτό το επίπεδο συγγραφής προγραμμάτων ονομάζεται *γλώσσα μηχανής* και είναι το χαμηλότερο δυνατό[12]. Ο προγραμματισμός σε τέτοιο επίπεδο είναι εξαιρετικά δύσκολος, διότι η κάθε εντολή δεν μπορεί να γίνει άμεσα κατανοητή από κάποιον που διαβάζει το πρόγραμμα (δε συνδέεται

[12] Όσο πιο περιγραφική και αφαιρετική είναι η γλώσσα προγραμματισμού, τόσο υψηλότερο γίνεται το επίπεδο προγραμματισμού.

με κάποια περιγραφή σε λεξικολογικό επίπεδο). Επιπλέον, η συγγραφή ενός μεγάλου αριθμού από 0 και 1 είναι εξαιρετικά κουραστική και μπορεί εύκολα να οδηγήσει σε σφάλματα (παράλειψη κάποιων ψηφίων ή προσθήκη επιπλέον) τα οποία για να διορθωθούν πρέπει να γίνει επαλήθευση κάθε ψηφίου εξ αρχής. Η λύση σε αυτά τα προβλήματα ήρθε με την αντιστοίχιση των εντολών σε μικρά ονόματα (τα οποία ονομάζονται μνημονικά) και είναι επεξηγηματικά ως προς τη λειτουργία τους. Δηλαδή, στην εντολή πρόσθεσης έγινε η αντιστοίχιση του μνημονικού **ADD**, το οποίο μόλις αναγνωστεί από κάποιον υποδηλώνει άμεσα πως πρόκειται για πρόσθεση. Φυσικά, για να γίνει η μετάβαση από το λεκτικό επίπεδο στο δυαδικό επίπεδο είναι απαραίτητη η χρήση ειδικού προγράμματος που θα μεταφράζει τα μνημονικά ονόματα στην αλληλουχία των 0 και 1 (Assembler).

Η χρήση των μνημονικών ονομάτων οδηγεί αμέσως σε γλώσσα προγραμματισμού που βρίσκεται ένα επίπεδο υψηλότερο από τη γλώσσα μηχανής και το όνομα που της έχουμε δώσει είναι συμβολική γλώσσα (Assembly). Οι σχεδιαστές κάθε επεξεργαστή αναλαμβάνουν να αντιστοιχίσουν ένα μνημονικό όνομα σε κάθε εντολή που μπορεί να εκτελέσει ο επεξεργαστής και γι'αυτό το λόγο κάθε συμβολική γλώσσα συνδέεται με ένα συγκεκριμένο επεξεργαστή ή μια συγκεκριμένη ομάδα επεξεργαστών (οικογένεια). Αυτό σημαίνει πως οι εντολές της συμβολικής γλώσσας του επεξεργαστή ARM946EJ-S είναι διαφορετικές από τις εντολές του Intel Pentium 4 ή του 80C196KB, διότι κάθε επεξεργαστής υποστηρίζει διαφορετικό σύνολο εντολών. Η συμβολική γλώσσα αν και διαφέρει από τη γλώσσα μηχανής στο γεγονός ότι εισάγει χρήση λεκτικών ονομάτων, συνεχίζει να έχει το μειονέκτημα πως απαιτεί την περιγραφή κάθε μικρού βήματος, έτσι ώστε να περιγραφεί πλήρως η διεργασία του χρήστη. Για κάθε διαδικασία επεξεργασίας δεδομένων, ο προγραμματιστής πρέπει να φροντίσει και για την μεταφορά και αποθήκευση των δεδομένων προς επεξεργασία. Άρα ο προγραμματιστής σε συμβολική γλώσσα δε χρειάζεται να εστιάζει μόνο στη περιγραφή της λύσης του προβλήματος, αλλά και στο πως θα διαχειριστεί τους πόρους του συστήματος. Αυτό το μειονέκτημα το παρακάμπτουν γλώσσες προγραμματισμού σε υψηλότερα επίπεδα, όπως C, FORTRAN, BASIC κ.α.

Η συμβολική γλώσσα παρουσιάζει προβλήματα, όπως η εξειδίκευση του προγράμματος σε ένα τύπο επεξεργαστών, η μεγάλη έκταση προγραμμάτων προκειμένου να υλοποιηθεί μια διεργασία και η δυσκολία στην συντήρηση και κατανόηση ενός κώδικα[13]. Επιπλέον, οι γλώσσες υψηλότερου επιπέδου μπορούν να υλοποιήσουν τις ίδιες διεργασίες με μικρότερο χρονικό κόστος (για να αναπτύξουμε μια εφαρμογή σε συμβολική γλώσσα χρειάζεται να δαπανήσουμε πολύ χρόνο, σε σχέση με το χρόνο που θα χρειαζόμασταν αν χρησιμοποιούσαμε τη γλώσσα C πχ. μια πρόσθεση καταγράφεται στη C σαν a = b + c;). Το πρόβλημα που

[13]Με τον όρο συντήρηση εννοούμε το βαθμό ευκολίας που θα έχει η προσπάθεια επέκτασης ή κατανόησης ενός κώδικα αρκετό καιρό μετά την ανάπτυξή του και από διαφορετικά άτομα.

εμφανίζεται με τις γλώσσες υψηλού επιπέδου εντοπίζεται στο γεγονός πως ο εκτελέσιμος κώδικας που παράγεται περιέχει πλεονάζουσες εντολές, διότι η διαδικασία της μετάφρασης που ακολουθείται για κώδικα των υψηλού επιπέδου γλωσσών χρησιμοποιεί τεχνικές που παράγουν *γενικευμένο*[14] και όχι στοχευμένο στη πραγματική αρχιτεκτονική κώδικα. Για παράδειγμα, στον παρακάτω κώδικα {a = b + c; d = b + c;} ο συμβολομεταφραστής compiler της C θα παράγει τα παρακάτω βήματα ικανά να εκτελεστούν από κάθε αρχιτεκτονική :

1. Προσκόμμιση του b
2. Προσκόμμιση του c
3. Πρόσθεση.
4. Αποθήκευση στο a.
5. Προσκόμμιση του b.
6. Προσκόμμιση του c.
7. Πρόσθεση.
8. Αποθήκευση στο d.

Οι παραπάνω εντολές προφανώς μπορούν να υπολογιστούν πολύ πιο αποδοτικά με :

1. Προσκόμμιση του b.
2. Προσκόμμιση του c.
3. Πρόσθεση.
4. Αποθήκευση στο a.
5. Αποθήκευση στο d.

Με τον δεύτερο τρόπο επιταχύνουμε την εκτέλεση της έκφρασης, διότι αφαιρούμε άχρηστες εντολές. Ο μεταφραστής της C δεν μπορεί να φτάσει σε τέτοιο βαθμό βελτιστοποίησης, διότι πρόκειται για λογική βελτιστοποίηση και όχι αλγεβρική[15]. Αρα, ο βασικός λόγος εκμάθησης της συμβολικής γλώσσας είναι η δυνατότητα που προσφέρει για ανάπτυξη τμημάτων του προγράμματός μας που απαιτούν ταχύτητα ή μικρό μέγεθος κώδικα. Επιπλέον, οι μεταφραστές των γλωσσών υψηλού επιπέδου δεν υποστηρίζουν με την ίδια ευκολία όλες τις εντολές γλώσσας μηχανής του επεξεργαστή, οπότε αν χρειαστούμε κάποια από αυτές θα πρέπει να την καλέσουμε μέσω συμβολικής γλώσσας. Τέλος, η αποσφαλμάτωση ενός κώδικα (το οποίο μπορεί να έχει φτιαχτεί με οποιαδήποτε γλώσσα προγραμματισμού) γίνεται σε επίπεδο συμβολικής γλώσσας, διότι η διαδικασία περιλαμβάνει τη βηματική εκτέλεση των εντολών της γλώσσας μηχανής. Από τη στιγμή που γίνεται η μετάφραση ενός προγράμματος δεν είναι δυνατή η ανάκτηση του πηγαίου κώδικα σε μορφή υψηλής γλώσσας προγραμματισμού. Έτσι, για να ελέγξουμε περιπτώσεις σφαλμάτων κατά την εκτέλεση

[14] Γνωστός και ως ενδιάμεσος κώδικας

[15] Η λογική βελτιστοποίηση έχει σχέση με τον αλγόριθμο που επιλέγουμε για την επίλυση ενός προβλήματος.

του προγράμματος, πρέπει να χρησιμοποιήσουμε ειδικά εργαλεία (αποσφαλματωτές - debuggers) τα οποία διερμηνεύσουν τον κώδικα μηχανής μέχρι το επίπεδο της συμβολικής γλώσσας. Προφανώς η εκμάθηση της συμβολικής γλώσσας συνδέεται άμεσα με την εκμάθηση της αρχιτεκτονικής του κάθε επεξεργαστή και των ιδιαίτερων χαρακτηριστικών που έχει. Αυτή η γνώση βοηθά την αποτελεσματικότερη συγγραφή προγραμμάτων, διότι έχοντας κατανοήσει καλύτερα τον τρόπο με τον οποίο αντιδρά ο επεξεργαστής στην κάθε εντολή, μπορούμε να συνδυάσουμε τις εντολές μεταξύ τους πολύ πιο αποδοτικά.

Πληροφοριακά . . .

Οι επεξεργαστές της σειράς i386 ή i686 (πχ. Pentium 4 ή Athlon) κατασκευάζονται από διαφορετικές εταιρίες, αλλά έχουν σε ένα μεγάλο ποσοστό κοινό σύνολο εντολών. Επιπλέον, διατηρούν στο σύνολο εντολών τους και εντολές οι οποίες υπήρχαν σε παλιότερα μοντέλα, όπως πχ. στον 486. Με αυτό τον τρόπο εξασφαλίζεται η αρμονική εκτέλεση των εφαρμογών σε κάθε επεξεργαστή, όπως και η εκτέλεση παλιών εφαρμογών. Φυσικά, η κάθε εταιρία προσφέρει και επιπλέον εντολές, οι οποίες δεν υπάρχουν στους επεξεργαστές των ανταγωνιστών, οι οποίες έχουν σχέση με την αύξηση της απόδοσης. Για να γίνει εφικτή η χρήση αυτών των εντολών πρέπει να χρησιμοποιηθούν κατάλληλα προγράμματα οδηγοί (drivers) από το λειτουργικό σύστημα (προγράμματα τα οποία εκτελούνται με αυξημένα δικαιώματα σε επίπεδο πυρήνα και έχουν στόχο την αντιστοίχιση μιας γενικής διαδικασίας στις εντολές αυτές). Ενα σημαντικό τμήμα των προγραμμάτων αυτών έχει αναπτυχθεί στη συμβολική γλώσσα του αντίστοιχου επεξεργαστή.

3.2 GNU Assembler

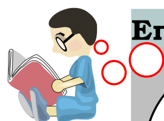
Το πρόγραμμα που θα χρησιμοποιήσουμε για τη μετάφραση από τη συμβολική γλώσσα σε γλώσσα μηχανής είναι ο GNU Assembler (gas). Στο AT91 χρησιμοποιούμε το όνομα as για τον GNU Assembler. Ο as δέχεται σαν είσοδο το πηγαίο αρχείο μας[16] και παράγει το εκτελέσιμο αρχείο. Οι εντολές της συμβολικής γλώσσας έχουν συγκεκριμένη και ομοιόμορφη δομή η οποία είναι:

[ετικέτα :]<ΚΕΝΟ>εντολή<ΚΕΝΟ>δεδομένα [<ΚΕΝΟ> @ σχόλια]

Το πρώτο πεδίο είναι η *ετικέτα* και επειδή δεν είναι υποχρεωτικό να αναγράφεται σε κάθε γραμμή, γι'αυτό και είναι ανάμεσα στα [ ]. Το <ΚΕΝΟ> μπορεί να αποτελείται από οποιονδήποτε συνδυασμό από spaces ή tabs και δεν λαμβάνεται υπ' όψιν το μέγεθός του

[16] Πηγαίο ονομάζεται το αρχείο που περιέχει τον κώδικά μας

(πάντα θεωρείται σαν ένα space). Το πεδίο της εντολής περιέχει το μνημονικό όνομα της εντολής (πχ. **ADD, MUL, ORR** κ.α.) ή κάποια εντολή προς τον ίδιο τον as για να τον ενημερώσει πως σε εκείνο το σημείο πρέπει να συνεχίσει τη μετάφραση με συγκεκριμένο τρόπο. Οι εξειδικευμένες εντολές προς τον as ονομάζονται ντιρεκτίβες. Το πεδίο των δεδομένων μπορεί να περιέχει ένα συνδυασμό από καταχωρητές, αριθμούς ή διευθύνσεις και το περιεχόμενο του πεδίου έχει σχέση με την εντολή που εκτελείται. Δηλαδή, η μορφή και τα περιεχόμενα αυτού του πεδίου καθορίζονται από την εντολή. Τέλος, το σύμβολο @ χρησιμοποιείται όταν θέλουμε να προσθέσουμε ένα σχόλιο μετά το τέλος της εντολής. Το κείμενο που ακολουθεί το @ δεν λαμβάνεται υπ' όψιν από τον as και χρησιμοποιείται από τους προγραμματιστές για να προσθέτουν σχόλια σε κρίσιμα σημεία. Η κάθε εντολή καταλαμβάνει μόνο μια γραμμή, δηλαδή δεν μπορεί να διαχωριστεί το περιεχόμενό της σε δυο ή περισσότερες γραμμές.



Επεξήγηση

Ένα ακόμη σύμβολο για σχόλια είναι το '/*...*/', όπου το κείμενο ανάμεσα στα '/*' & '*/' δεν λαμβάνεται υπ' όψιν από το μεταφραστή και μπορεί να εκταθεί σε όσες γραμμές είναι επιθυμητό. Για παράδειγμα:

```
/*
All this is a big comment
MOV R2, R2, LSL #4
ADC R2, R1, #15
⋮
*/
```

3.2.1 Ετικέτες

Πολλές φορές, κατά την εκτέλεση του προγράμματος είναι επιθυμητή η αλλαγή της ροής εκτέλεσης και η διακλάδωση σε διαφορετικό σημείο, έτσι ώστε να εκτελεστεί συγκεκριμένο τμήμα του κώδικά μας. Για να γίνει εφικτή η διακλάδωση είναι απαραίτητο να γνωρίζουμε τη διεύθυνση της εντολής στην οποία θα μεταβεί η εκτέλεση. Επιπλέον, κάθε φορά που εκτελείται ένα πρόγραμμα, οι διευθύνσεις στις οποίες βρίσκονται οι εντολές είναι διαφορετικές (εξαρτάται από το λειτουργικό σύστημα το που θα εντοπιστεί σημείο της μνήμης που να διαθέτει τον απαιτούμενο χώρο για να φορτωθεί το πρόγραμμα), άρα δεν μπορούμε να τις γνωρίζουμε εξ αρχής. Για να λυθούν αυτά τα προβλήματα, χρησιμοποιούμε σχετικές διευθύνσεις. Πιο συγκεκριμένα, αν χρειαστεί να εκτελέσουμε μια διακλάδωση σε διαφορετικό σημείο, προσθέτουμε (ισοδύναμα αφαιρούμε για άλματα / διακλαδώσεις προς τα πίσω) την απόσταση μέχρι εκείνο το σημείο στον μετρητή προγράμματος και η ροή της εκτέλεσης μεταβαίνει εκεί. Για να υλοποιήσει ο as τη λειτουργία αυτή χρησιμοποιεί σαν ενδείξεις τις ετικέτες. Κατά τη συγγραφή του προγράμματος, όταν θέλουμε να εκτελέσουμε

μια διακλάδωση σε διαφορετικό σημείο, χρησιμοποιούμε την ετικέτα που έχουμε αναθέσει σε εκείνο το σημείο και ο as υπολογίζει αυτόματα την απόσταση. Για παράδειγμα:

```
LOOP:MOV R0, R2 @ R2 - > R0
⋮
CMP R0, #4 @ R0 == 4 ???
BEQ LOOP @ If(R0 == 4) Goto LOOP
```

Το όνομα της ετικέτας μπορεί να αποτελείται από τους χαρακτήρες A. . . Z, a. . . z, 0. . . 9, _ και στο τέλος πρέπει να υπάρχει ο χαρακτήρας ':' χωρίς κανένα κενό ενδιάμεσα. Για παράδειγμα:

LOOP:	*Σωστό*
LOOP :	*Λάθος* (Υπάρχει κενό πριν το :)
_LOOP123bf_DataAREA:	*Σωστό*
@df.#Number926*:	*Λάθος* (Περιέχει μη επιτρεπτούς χαρακτήρες)

Γενικά, τα ονόματα των ετικετών είναι καλό να περιγράφουν το λόγο για τον οποίο επιλέξαμε να σημειώσουμε με την ετικέτα την εντολή. Για παράδειγμα, αν θέλουμε να υλοποιήσουμε ένα βρόχο, τα ονόματα που ταιριάζουν είναι το LOOP ή το WHILE. Αυτή η προσέγγιση βοηθά στην αναγνωσιμότητα του προγράμματος, όπως και στην αποφυγή σφαλμάτων λόγω σύγχυσης ονομάτων. Η επιλογή μιας ετικέτας με το όνομα A3bfd είναι έγκυρη από πλευρά λεξικολογικών κανόνων, αλλά δεν βοηθά στην κατανόηση του λόγου ύπαρξής της. Χρειάζεται προσοχή στην επιλογή ονόματος ετικέτας, διότι δεν πρέπει να είναι ίδιο με το μνημονικό όνομα κάποιας εντολής ή κάποιας ντιρεκτίβας.

3.2.2 Ντιρεκτίβες

Ο as μπορεί να δεχθεί εντολές κατά τη διαδικασία της μετάφρασης, ώστε να εκτελέσει κάποια συγκεκριμένη λειτουργία ή να ερμηνεύσει τον κώδικα που συναντά από εκείνο το σημείο και μετά με ειδικό τρόπο. Η γενική μορφή τους είναι **.<directive> [params]**, όπου το <directive> είναι το όνομα της ντιρεκτίβας και [params] μια λίστα με παραμέτρους που εξαρτώνται από κάθε ντιρεκτίβα. Οι εντολές αυτές παρουσιάζονται συνοπτικά στον πίνακα *Ντιρεκτίβες* (κεφ. 3.2.2, σελ. 41)[17].

[17] Για μια πλήρη λίστα από τις ντιρεκτίβες του as επισκεφτείτε το www.gnu.org

Ντιρεκτίβα	Περιγραφή
.abort	Σταματά η διαδικασία μετάφρασης.
.align	Τοποθετεί τα δεδομένα σε στοιχισμένες διευθύνσεις.
.arm	Σηματοδοτεί την έναρξη του κώδικα.
.asciz	Τοποθετεί strings αλφαριθμητικών.
.byte	Τοποθετεί bytes.
.data	Τοποθετεί τα επόμενα αριθμητικά δεδομένα σε συγκεκριμένες θέσεις.
.equ	Αντιστοιχεί μια τιμή σε ένα σύμβολο.
.global	Δηλώνει ένα καθολικά προσπελάσιμο σύμβολο.
.hword	Τοποθετεί halfwords.
.if	Εκτελεί τον κώδικα που το ακολουθεί, αν ισχύει η συνθήκη του if.
.include	Εισάγει τον κώδικα άλλου αρχείο στο τρέχον.
.text	Τοποθετεί τον κώδικα που ακολουθεί σε συγκεκριμένη θέση.
.word	Τοποθετεί words.

Πίνακας 3.16: Ντιρεκτίβες

.abort

Μόλις συναντίσει αυτή τη ντιρεκτίβα ο as, θα σταματήσει αμέσως τη μετάφραση του προγράμματος.

.align

Η μορφή αυτής της ντιρεκτίβας είναι **.align <num>**, όπου το <num> ορίζει τον αριθμό, με βάση τον οποίο θα γίνει η επιθυμητή στοίχιση. Πολλές φορές, θέλουμε τα δεδομένα να τοποθετούνται σε διευθύνσεις που αποτελούν ακέραια πολλαπλάσια κάποιου αριθμού (για παράδειγμα, οι τιμές των words τοποθετούνται σε διευθύνσεις που είναι πολλαπλάσια του 4). Με την .align μπορούμε να τοποθετήσουμε τα δεδομένα που ακολουθούν με τέτοιο τρόπο, έτσι ώστε να ξεκινούν από διεύθυνση που είναι πολλαπλάσιο του <num>. Οι θέσεις μνήμης από το τελευταίο δεδομένο που τοποθετήθηκε, μέχρι την στοιχισμένη διεύθυνση, δεν χρησιμοποιούνται. Για παράδειγμα, με την .align 6 θα αναγκάσουμε τον as να τοποθετήσει τα επόμενα δεδομένα στην διεύθυνση που είναι το πλησιέστερο πολλαπλάσιο του 6 (αν το τελευταίο δεδομένο τοποθετήθηκε στην 0x8000, το επόμενο θα τοποθετηθεί στην 0x8004 και οι 3 ενδιάμεσες θέσεις θα μείνουν αχρησιμοποίητες).

.arm

Με αυτή την εντολή ενημερώνουμε τον as ότι πρέπει να παράγει κώδικα 32 bits. Η ντιρεκτίβα αυτή πρέπει να τοποθετείται στην αρχή του προγράμματος.

.asciz

Με αυτή την εντολή ενημερώνουμε τον as πως ακολουθεί ένα αλφαριθμητικό string. Η γενική μορφή της ντιρεκτίβας είναι **.asciz "<string>"**. Για παράδειγμα, οι παρακάτω εντολές ορίζουν την ετικέτα *lesson* ως δείκτη του string "Assembly Cource" και την ετικέτα *CPU* ως δείκτη του string "AT91SAM9261":

```
lesson:
.asciz "Assembly Cource"
CPU:
.asciz "AT91SAM9261"
```

.byte

Με αυτή την εντολή ενημερώνουμε τον as πως ακολουθεί μια σειρά από bytes. Η γενική μορφή της ντιρεκτίβας είναι **.byte [,<value>]***, όπου το <value> είναι μια 8 bit τιμή. Μπορούμε να ορίσουμε μια σειρά από τιμές, οι οποίες θα διαχωρίζονται μεταξύ τους με κόμμα και θα τοποθετηθούν στην μνήμη σε διαδοχικές θέσεις, ξεκινώντας από την πρώτη τιμή, η οποία θα τοποθετηθεί στην μικρότερη διεύθυνση. Προσοχή χρειάζεται στην καταγραφή των τιμών, διότι όλες πρέπει να βρίσκονται στην ίδια γραμμή (αν είναι πολλές, χρησιμοποιείστε την ντιρεκτίβα πάλι, στην επόμενη γραμμή). Για παράδειγμα, οι εντολές που ακολουθούν τοποθετούν διαδοχικά στη μνήμη τα παρακάτω bytes:

```
.byte 0x35, 0x42, 0xaa, 0x68
.byte 0x12, 0x60, 0xba, 0xcd
```

.data

Με αυτή την εντολή ειδοποιούμε τον as πως τα δεδομένα που θα ακολουθήσουν πρέπει να τα τοποθετήσει στο τμήμα της μνήμης που δεσμεύει για να τοποθετεί μη-σταθερά δεδομένα[18].

.equ

Με την εντολή αυτή ορίζουμε ένα σύμβολο ως ενεργό και του αναθέτουμε μια τιμή. Το σύμβολο αυτό δεν χρησιμοποιείται στο πρόγραμμά μας (δεν δεσμεύεται χώρος για αυτό),

[18]Τα μη-σταθερά δεδομένα είναι αυτά, των οποίων η τιμή αλλάζει κατά την εκτέλεση του προγράμματος

αλλά χρησιμοποιείται για να αντιστοιχεί ονόματα σε τιμές (ψευδώνυμα). Η μορφή του είναι **.equ <symbol>, <value>**, όπου <symbol> είναι το όνομα και <value> η τιμή στην οποία αντιστοιχίζεται. Για παράδειγμα, με την εντολή **.equ FunnyVal, #0x12345678**, αντιστοιχούμε το όνομα FunnyVal στην τιμή 0x12345678 και μπορούμε να χρησιμοποιήσουμε το FunnyVal σαν να χρησιμοποιούσαμε τον ίδιο τον αριθμό. Χρειάζεται προσοχή ώστε να μην χρησιμοποιήσουμε μνημονικό όνομα εντολής ή ετικέτας στο όνομα του συμβόλου, διότι ο as αν βρει μια εντολή ή ετικέτα με όνομα ίδιο με του συμβόλου, θα το ερμηνεύσει σαν την τιμή που έχουμε αντιστοιχίσει στο σύμβολο.

.global

Η μορφή του είναι **.global <label>**, όπου το <label> είναι η ετικέτα που θέλουμε να ορίσουμε σαν καθολική. Με αυτή την εντολή ειδοποιούμε τον as πως πρέπει να εισάγει την ετικέτα που θέλουμε, στη λίστα των συμβόλων που μπορούν να προσπελαστούν από άλλα προγράμματα. Όταν ένας μεταφραστής παράγει το εκτελέσιμο πρόγραμμα, συνήθως δεν το κάνει σε ένα βήμα. Σαν πρώτο βήμα, παράγει ένα ενδιάμεσο αρχείο που περιέχει τον εκτελέσιμο κώδικά μας και τις μεταβλητές μας (το ενδιάμεσο αρχείο ονομάζεται object file). Αν το πρόγραμμά μας χρησιμοποιεί κώδικα από πολλά αρχεία, τότε στο πρώτο βήμα θα γίνει η παραγωγή όλων των object files (ένα για κάθε πηγαίο αρχείο κώδικα). Στο δεύτερο βήμα (γνωστό ως διασύνδεση - linking), θα αναλάβει να ελέγξει ποιες συναρτήσεις καλούνται ανάμεσα στα αρχεία και θα συμπεριλάβει μόνο αυτές στο τελικό εκτελέσιμο αρχείο (αν κάποια συνάρτηση δεν καλείται πουθενά, στο ενδιάμεσο αρχείο θα υπάρχει ο εκτελέσιμος κώδικάς της, αλλά στο τελικό δεν θα συμπεριληφθεί). Επίσης θα ανακληθούν σε αυτό το στάδιο οι απαιτούμενες βασικές συναρτήσεις από τις βιβλιοθήκες object αρχείων της γλώσσας. Για να μπορέσει να βρει τις συναρτήσεις που παρέχει κάθε ενδιάμεσο αρχείο, εξετάζει τη λίστα καθολικών συμβόλων.

.hword

Η εντολή αυτή έχει την ίδια ακριβώς λειτουργία με την *.byte*, μόνο που τώρα μπορούμε να τοποθετούμε στη μνήμη τιμές εύρους 16 bit. Για παράδειγμα, οι τιμές που εισήχθησαν με την *.byte* θα μπορούσαν να εισαχθούν σαν:

```
.hword 0x4235, 0x68aa
.hword 0x6012, 0xcdba
```

Παρατηρείστε πως η αναγραφή των αριθμών έχει φαινομενικά αλλάξει τη σειρά των bytes. Στην πραγματικότητα, η αλλαγή αυτή είναι επιβεβλημένη, διότι το περισσότερο σημαντικό byte ενός halfword τοποθετείται σε μεγαλύτερη διεύθυνση απ'ότι το λιγότερο σημαντικό.

.if

Την εντολή .if τη χρησιμοποιούμε για να ελέγξουμε την τιμή ενός συμβόλου που έχει δηλωθεί με την εντολή .equ. Όπως και η .equ, δεν παράγει εκτελέσιμο κώδικα, αλλά μπορεί να περικλείει κώδικα και να επιτρέπει στον as να τον συμπεριλάβει, αν η συνθήκη του ικανοποιείται. Για παράδειγμα, ο παρακάτω κώδικας:

```
.equ DEBUG, 1
⋮
.if DEBUG
MOV R0, #0x30
.else
MOV R0, #0x10
.endif
```

επειδή έχει αναθέσει την τιμή 1 στο σύμβολο DEBUG θα ικανοποιήσει την συνθήκη του .if και ο as θα συμπεριλάβει την εντολή MOV R0, #0x30. Αν δεν θέλαμε να ενεργοποιήσουμε αυτό το χαρακτηριστικό στο πρόγραμμά μας, ο as θα συμπεριλάμβανε την εντολή MOV R0, #0x10. Η γενική μορφή της εντολή είναι:

```
.if <expression>
⋮
.else
⋮
.endif
```

Το .else δεν είναι απαραίτητο να υπάρχει και το προσθέτουμε αν θέλουμε να χρησιμοποιήσουμε εναλλακτικό κώδικα, στην περίπτωση που η συνθήκη δεν ισχύει. Το .endif σηματοδοτεί το τέλος της περιοχής που ελέγχεται από την εντολή .if και είναι απαραίτητο να υπάρχει. Το <expression> είναι μια αριθμητική έκφραση, η οποία πρέπει να είναι θετική για να ισχύει (αν είναι 0 δεν ισχύει).

.include

Με αυτή την εντολή συμπεριλαμβάνουμε στον κώδικά μας τον κώδικα ενός άλλου πηγαίου αρχείου assembly . Η μορφή της εντολής είναι **.include "<file>"**, όπου το "<file>" είναι το όνομα του αρχείου που θέλουμε να συμπεριλάβουμε. Για παράδειγμα, αν στο αρχείο first.s υπάρχει ο κώδικας:

```
.data
ByteArray:
.word 0xcafebaba, 0xdeadbeef, ...
        ⋮
```

και στο αρχείο second.s υπάρχει ο κώδικας:

```
.include "first.c"
        ⋮
LDR R0, =ByteArray @ addr(ByteArray) -> R0
        ⋮
```

ο as θα μεταφράσει το αρχείο second.s σαν να συναντούσε τον κώδικα:

```
.data
ByteArray:
.word 0xcafebaba, 0xdeadbeef, ...
        ⋮
LDR R0, =ByteArray @ addr(ByteArray) -> R0
        ⋮
```

.text

Με αυτή την εντολή ειδοποιούμε τον as πως από εκείνο το σημείο και μετά, θα ακολουθήσει εκτελέσιμος κώδικας. Για παράδειγμα, στις παρακάτω γραμμές εναλλάσονται οι περιοχές δεδομένων και κώδικα:

```
.text
⋮
LDR R0, =ByteArray @ addr(ByteArray) -> R0
⋮
.data
ByteArray:
.byte 0x35, 0xaa, 0x22, ...
⋮
.text
LDR R1, [R0, #0x01] @ 0xaa -> R1
⋮
```

.word

Η εντολή αυτή έχει την ίδια ακριβώς λειτουργία με την *.byte*, μόνο που τώρα μπορούμε να τοποθετούμε στη μνήμη τιμές εύρους 32 bit. Για παράδειγμα, οι τιμές που εισήχθησαν με την *.byte* θα μπορούσαν να εισαχθούν σαν:

.word 0x68aa4235
.word 0xcdba6012

3.3 Παραδείγματα

Ο καλύτερος τρόπος για την εξοικείωση με τη γλώσσα Assembly και την σωστή εκμάθησή της είναι μέσω της κατανόησης προγραμμάτων και ανάπτυξης νέων. Γι'αυτό και σε αυτή την ενότητα θα παραθέσουμε μια σειρά από προγράμματα Assembly, θα εξηγήσουμε τη λογική που οδήγησε στον τρόπο ανάπτυξής τους και το πως συνδυάζονται με τις εντολές και την αρχιτεκτονική του επεξεργαστή μας.

3.3.1 Μεταφορά δεδομένων εύρους 32-bit

Μεταφορά δεδομένων εύρους 32-bit		
1	**.arm**	
2	**.text**	
3	**.global main**	
4		
5	**main:**	
6	**STMDB R13!, {R0-R12}**	*@Αποθηκεύουμε τους καταχωρητές που τυχόν θα χρησιμοποιήσουμε στο σωρό του συστήματος*
7	**LDR R0, =Value**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Value*
8	**LDR R1, [R0]**	*@Εισάγουμε στον R1 την τιμή που βρίσκεται στη διεύθυνση R0*
9		
10	**LDR R0, =Result**	*@Εισάγουμε στον καταχωρητή R0 τη διεύθυνση της θέσης μνήμης όπου θα αποθηκεύσουμε το αποτέλεσμα*
11	**STR R1, [R0]**	*@Αποθηκεύουμε την τιμή του R1 στη θέση μνήμης που περιέχει ο R0*
12	**LDMIA R13!, {R0-R12}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που τυχόν χρησιμοποιήσαμε*
13	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η συνάρτηση main. Αυτό το σημείο ανήκει στο λειτουργικό σύστημα και είναι απαραίτητο να επιστρέψουμε εκεί, ώστε να συνεχίσει η εκτέλεση των υπόλοιπων προγραμμάτων, αλλά και για να μπορούμε να εισάγουμε εντολές στο Linux shell*
14		
15	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "data" που ορίζει αυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*
16	**Value:**	
17	**.word 0xCAFEBABA**	*@Το δεδομένο 0xCAFEBABA τοποθετείται σε θέση μνήμης η οποία σημειώνεται με το όνομα Value*
18	**Result:**	
19	**.word 0**	*@Το αποτέλεσμα θα τοποθετηθεί στη θέση μνήμης Value+4 ή Result*

Στο παραπάνω πρόγραμμα εκτελέσαμε τη μεταφορά των 32 bit δεδομένων από τη θέση μνήμης *Value* και στη θέση μνήμης *Result*. Το σύμβολο "=" που βρίσκεται πριν από τις θέσεις μνήμης χρησιμοποιείται για να αποθηκευτεί στον καταχωρητή η διεύθυνση που σηματοδοτεί η ετικέτα *Value* ή η *Result*. Παρατηρείστε επίσης την αποθήκευση των καταχωρητών στο σωρό του συστήματος (ελέγχεται από τον R13) πριν αρχίσει η εκτέλεση του προγράμματός μας και την ανάκτησή τους, με την ολοκλήρωση του προγράμματος. Επίσης, σημειώστε πως είναι απαραίτητο να ονομάσουμε την αρχική συνάρτησή μας (με χρήση ετικέτας) main και να την δηλώσουμε σαν .global, ώστε να την εντοπίσει το σύστημα όταν προσπαθήσει να εκτελέσει το πρόγραμμά μας[19].

3.3.2 Άθροιση αριθμών εύρους 32-bit

Άθροιση αριθμών εύρους 32-bit		
1	**.arm**	
2	**.text**	
3	**.global main**	
4		
5	**main:**	
6	**STMDB R13!, {R0-R2}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
7	**LDR R0, =Values**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Values*
8	**LDR R1, [R0, #0]**	*@Εισάγουμε στον R1 το περιεχόμενο της μνήμης στο οποίο δείχνει ο R0*
9	**LDR R2, [R0, #4]**	*@Εισάγουμε στον R2 το περιεχόμενο της μνήμης που βρίσκεται 4 θέσεις μετά από αυτή που δείχνει ο R0*
10	**ADD R2, R2, R1**	*@Προσθέτουμε το περιεχόμενο του R2 με το περιεχόμενο του R1 και το αποθηκεύουμε στον R2*
11		
12	**LDR R0, =Result**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Result*
13	**STR R2, [R0]**	*@Αποθηκεύουμε την τιμή του R2 στη θέση μνήμης που περιέχει ο R0*
14	**LDMIA R13!, {R0-R2}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε*
15	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η υπορουτίνα main*
16		
17	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "Data" που ορίζει άυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*
18	**Values:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Values*

[19]Θεωρείστε πως το όνομα main είναι προσυμφωνημένο στα GNU εργαλεία.

19	**.word 0xFFF00000**	
20	**.word 0xCDEF**	
21	**Result:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Result*
22	**.word 0**	

Στο πρόγραμμα αυτό υλοποιήσαμε άθροιση ανάμεσα στα περιεχόμενα των θέσεων μνήμης Values και (Values+4). Η διαδικασία ξεκινά με την αντιγραφή της διεύθυνσης όπου υπάρχει ο πρώτος αριθμός στον καταχωρητή R0. Αυτό το βήμα είναι απαραίτητο, διότι χωρίς τη διεύθυνση δε μπορούμε να προσπελάσουμε το περιεχόμενο αυτής της θέσης μνήμης με άλλο τρόπο. Από τη στιγμή που γνωρίζουμε την θέση μνήμης στην οποία είναι τοποθετημένος ο αριθμός, καλούμε την εντολή μεταφοράς του δεδομένου στον καταχωρητή R1. Το δεύτερο βήμα είναι εξίσου απαραίτητο, διότι η αρχιτεκτονική του επεξεργαστή μας δεν επιτρέπει την εκτέλεση πράξεων σε θέσεις μνήμης εκτός του συνόλου καταχωρητών. Τα δυο βήματα αυτά θα τα ακολουθούμε πάντα, όταν πρόκειται για μεταφορά δεδομένων από τη μνήμη προς τους καταχωρητές. Στη συνέχεια πρέπει να υπολογίσουμε τη διεύθυνση του δεύτερου αριθμού, για τον οποίο το μόνο που γνωρίζουμε είναι πως βρίσκεται 4 bytes μετά από τον προηγούμενο[20]. Γιαύτό και διευθυνσιοδοτούμε τη θέση μνήμης που βρίσκεται 4 bytes μετά από τη διεύθυνση που είχαμε αποθηκεύσει στον R0. Η πράξη της πρόσθεσης χρησιμοποιεί τα δεδομένα που βρίσκονται στους R1, R2 και το άθροισμα το αποθηκεύει στον R2. Ο στόχος μας είναι να αποθηκεύσουμε το περιεχόμενο του R2 σε συγκεκριμένη θέση μνήμης, η οποία είναι η Result. Επαναλαμβάνουμε τα βήματα που κάναμε για να φέρουμε σε ένα καταχωρητή τη διεύθυνση μιας θέσης μνήμης και καταχωρούμε στον R0 τη διεύθυνση της Result. Με έμμεση αναφορά στη μνήμη (δηλαδή με χρήση του R0 ως δείκτη) αποθηκεύουμε εκεί τα δεδομένα που υπάρχουν στον R2. Παρατηρείστε πως η θέση μνήμης Result θα μπορούσε να αναφερθεί και σαν (Values+8) αν δε θέλαμε να χρησιμοποιήσουμε την επιπλέον ετικέτα Result.

3.3.3 Σωρός

Σωρός		
1	**.arm**	
2	**.text**	
3	**.global main**	
4	**nik**	
5	**main:**	
6	**STMDB R13!, {R0-R12, R14}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
7		

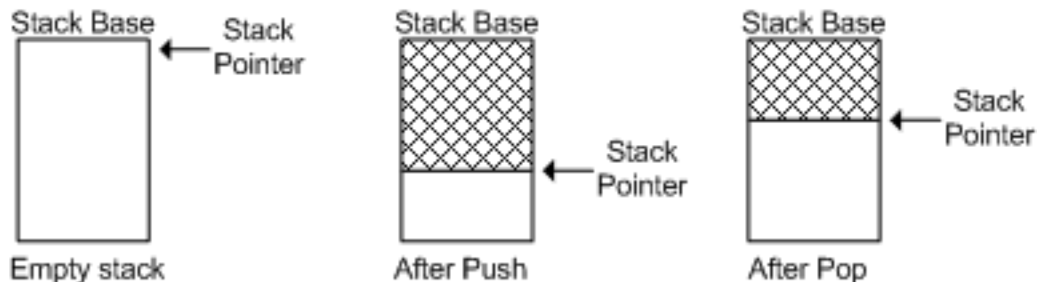
[20]Θυμηθείτε πως οι θέσεις στις οποίες τοποθετούνται οι μεταβλητές στη μνήμη δεν είναι προκαθορισμένες και το λειτουργικό σύστημα επιλέγει κάθε φορά την περιοχή που θα τις τοποθετήσει.

8	**LDR R12, =Start**	*@Εισάγουμε στον R12 τη διεύθυνση που σηματοδοτεί η ετικέτα Start*
9	**ADD R12, R12, 0x2C**	*@Προσθέτουμε στην αρχική διεύθυνση του σωρού την τιμή 0x2c* $= 44_{10}$. *Με αυτό τον τρόπο ο R12 περιέχει τη διεύθυνση του 12ου στοιχείου της λίστας*
10		
11	**MOV R0, #4**	*@Εισάγουμε την τιμή 4 στον R0*
12	**BL Push**	*@Καλούμε την υπορουτίνα Push, έχοντας στον R0 την τιμή 4*
13		
14	**MOV R0, #2**	*@Εισάγουμε την τιμή 2 στον R0*
15	**BL Pop**	*@Καλούμε την υπορουτίνα Pop, έχοντας στον R0 την τιμή 2*
16		
17	**MOV R0, #0xF**	*@Εισάγουμε την τιμή 15 στον R0*
18	**BL Push**	*@Καλούμε την υπορουτίνα Push, έχοντας στον R0 την τιμή 15*
19		
20	**MOV R0, #0xF**	*@Εισάγουμε πάλι την τιμή 15 στον R0*
21	**BL Pop**	*@Καλούμε την υπορουτίνα Pop, έχοντας στον R0 την τιμή 15*
22		
23	**LDMIA R13!, {R0-R12, PC}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε και ταυτόχρονα εισάγουμε στον PC την τιμή του R14, κάτι που θα μεταφέρει τη ροή εκτέλεσης του προγράμματος στη θέση μνήμης απ'όπου κλήθηκε η συνάρτηση main*
24		
25	**/*——– PUSH ———–*/**	
26	**push:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα Push. Κάθε αναφορά σε αυτό το όνομα θα αναφέρεται σε αυτή τη θέση μνήμης*
27	**STMDB R13!, {R4}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
28	**LDR R4, =Start**	*@Εισάγουμε στον R4 τη διεύθυνση που σηματοδοτεί η ετικέτα Start*
29		
30	**PushLoop:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα PushLoop*
31	**CMP R4, R12**	*@Συγκρίνουμε το περιεχόμενο του R4 με το περιεχόμενο του R12, με την πράξη της αφαίρεσης, R4-R12*
32	**BHI PushOvr**	*@Αν R4 > R12 τότε συνεχίζουμε την εκτέλεση από τη διεύθυνση που είναι σημειωμένη με την ετικέτα PushOvr*
33	**STR R0, [R12]**	*@Αποθηκεύουμε στη θέση μνήμης που περιέχεται στον R12 την τιμή του R0*
34	**SUB R12, R12, #4**	*@Αφαιρούμε 4 από την τιμή του R12*
35	**SUBS R0, R0, #1**	*@Αφαιρούμε 1 από την τιμή του R0 και ενημερώνουμε τον καταχωρητή κατάστασης για το είδος του αποτελέσματος (μηδενικό, αρνητικο, κλπ)*
36	**BHI PushLoop**	*@Αν η προηγούμενη αφαίρεση δεν παρήγαγε αρνητικό ή μηδενικό υπόλοιπο, συνεχίζουμε την εκτέλεση από τη διεύθυνση που είναι σημειωμένη από την ετικέτα PushLoop*
37		

38	**B PushEnd**	*@Εκτελούμε μετάβαση της ροής του κώδικα στη θέση μνήμης που έχει σημειωθεί με την ετικέτα PushEnd*
39	**PushOvr:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα PushOvr.*
40	**MVN R0, #0**	*@Αν φτάσουμε σε αυτό το σημείο, σημαίνει πως έχει σημειωθεί σφάλμα και τοποθετούμε στον R0 την τιμή 0xFFFFFFFF για να ειδοποιήσουμε την συνάρτηση που μας κάλεσε*
41		
42	**PushEnd:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα PushEnd.*
43	**LDMIA R13!, {R4}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε*
44	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η υπορουτίνα push*
45		
46	**/*——– POP ————*/**	
47	**Pop:**	
48	**STMDB R13!, {R4,R5}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
49	**LDR R4, =Start**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Start*
50	**ADD R4, R4, #0x02C**	*@Προσθέτουμε στον R4 την τιμή 4*
51		
52	**PopLoop:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα PopLoop*
53	**CMP R12, R4**	*@Συγκρίνουμε τον R12 με τον R4, με την πράξη της αφαίρεσης R12-R4*
54	**BCS PopUnd**	*@Αν R12 >= R4 εκτελούμε άλμα στην διεύθυνση που σημειώνεται από την ετικέτα PopUnd*
55	**ADD R12, R12, #4**	*@Προσθέτουμε στον R12 την τιμή 4*
56	**LDR R5, [R12]**	*@Εισάγουμε στον R5 την τιμή της διεύθυνσης μνήμης, στην οποία δείχνει ο R12*
57	**SUBS R0, R0, #1**	*@Ελλατώνουμε κατά 1 τον R0*
58	**BHI PopLoop**	*@Αν ο R0 είναι μεγαλύτερος από 0 εκτελούμε άλμα στη διεύθυνση μνήμης που σημειώνεται από την ετικέτα PopLoop*
59		
60	**B PopEnd**	*@Εκτελούμε άλμα χωρίς συνθήκη στη διεύθυνση μνήμης που σημειώνεται από την ετικέτα PopLoop*
61	**PopUnd:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα PopUnd*
62	**MVN R0, #0**	*@Εισάγουμε στον R0 την τιμή 0xFFFFFFFF*
63		
64	**PopEnd:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα PopEnd*
65	**LDMIA R13!, {R4,R5}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε*
66	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η υπορουτίνα pop*
67		
68	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "Data" που ορίζει άυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*

69	**Start:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Start*
70	**.word 0,0,0,0,0,0**	
71	**.word 0,0,0,0,0,0**	

Στο AT91 υπάρχει ήδη ο σωρός του συστήματος, τον οποίο έχουμε ήδη χρησιμοποιήσει. Σε αυτό το παράδειγμα δημιουργούμε ένα νέο δικό μας σωρό. Θυμηθείτε ότι ο σωρός είναι ένα τμήμα της μνήμης που χρησιμοποιείται για να αποθηκεύει προσωρινά δεδομένα. Η λογική που χαρακτηρίζει τη λειτουργία του βασίζεται στο ότι δεν μπορούν να προσπελαστούν άμεσα τα δεδομένα που έχουν αποθηκευτεί, εκτός του πρώτου ή του τελευταίου. Ο διαχωρισμός αυτός οδηγεί σε δυο κατηγορίες σωρού, την *Last-In First-Out (LIFO)* και την *First-In First-Out (FIFO)*. Ο σωρός που υλοποιείται στο παράδειγμά μας είναι LIFO και λειτουργία του εμφανίζεται στην εικόνα LIFO (κεφ. 3.9, σελ. 51). Ένα από τα χαρακτηριστικά της LIFO είναι ότι ξεκινά από μια θέση μνήμης και οι εισαγωγές των δεδομένων γίνονται σε μικρότερες θέσεις μνήμης και όχι σε μεγαλύτερες. Δηλαδή, αν η αρχή του σωρού είναι στη θέση 0x8000, το πρώτο στοιχείο θα αποθηκευτεί εκεί, το δεύτερο θα αποθηκευτεί στην 0x7FFC κλπ.



Σχήμα 3.9: LIFO

Στο δικό μας σωρό ο R12 θα παίζει το ρόλο του δείκτη σωρού, ενώ ο R4 κρατά τα όρια του σωρού για να αποτρέψουμε περιπτώσεις υπερχείλισης προς τα πάνω και προς τα κάτω. Ο κώδικάς μας παρέχει τις συναρτήσεις push και pop για αποθήκευση και ανάκτηση από το σωρό. Ο αριθμός των στοιχείων που θα τοποθετηθούν ή θα ανακτηθούν από το σωρό είναι μια παράμετρος αυτών των συναρτήσεων την οποία την περνάμε μέσω του καταχωρητή R0. Πιο συγκεκριμένα, η συνάρτηση push, περιμένει να βρει στον R0 όχι μόνο το πόσα στοιχεία θα εισάγει στο σωρό, αλλά έμμεσα η τιμή του R0 καθορίζει και τη τιμή τους. Τα στοιχεία που εισάγονται ξεκινούν από τον αριθμό στοιχείων εισαγωγής και καταλήγουν στο 1 (πχ. αν εισάγουμε 3 στοιχεία θα εισαχθούν οι αριθμοί 3,2,1). Όταν ξεκινά η εκτέλεση της push γίνεται έλεγχος για το αν ο σωρός είναι πλήρης, με τη σύγκριση ανάμεσα στο κατώτερο όριο και τον δικό μας stack pointer, ο οποίος βρίσκεται είναι ο καταχωρητής R12

και έχει αρχική τιμή το ανώτερο όριο του σωρού. Γίνεται η αποθήκευση του στοιχείου και ο stack pointer μειώνεται κατά 4, διότι αποθηκεύουμε words. Επιπλεόν, μειώνουμε και τον R0, ο οποίος όταν φτάσει στο 0 θα σηματοδοτήσει το τέλος της διαδικασίας εισαγωγής. Παρατηρείστε πως πρώτα αποθηκεύουμε τον αριθμό μας και μετά μειώνουμε τον stack pointer, κάτι που σημαίνει πως ο stack pointer θα δείχνει πάντα στην επόμενη ελεύθερη θέση του σωρού και η επόμενη εισαγωγή θα αποθηκεύσει το στοιχείο της σε εκείνη τη θέση. Όταν φτάσουμε να εισάγουμε αριθμό και στην τελευταία θέση του σωρού, ο stack pointer θα δείξει σε θέση μνήμης εκτός της περιοχής του σωρού, κάτι που θα ανιχνευτεί με τη σύγκριση με το κατώτερο όριο. Σε αυτή την περίπτωση τοποθετούμε την τιμή 0xffffffff στον R0, για να ειδοποιήσουμε το χρήστη πως ο σωρός είναι πλήρης.

Η αντίστροφη διαδικασία της εισαγωγής γίνεται με τη χρήση της συνάρτησης pop, η οποία αυξάνει τον stack pointer και επιστρέφει την τιμή που υπάρχει σε εκείνη τη θέση μνήμης. Παρατηρείστε πως η διαδικασία λήψης των τιμών εκτελεί τα βήματα αλλαγής του stack pointer και πρόσβασης της μνήμης με αντίθετη σειρά. Η σύγκριση που γίνεται τώρα είναι ανάμεσα στον stack pointer και το ανώτερο όριο του σωρού, οπότε αν ο stack pointer γίνει ίσος με αυτό ειδοποιούμε τον χρήστη πως ο σωρός είναι άδειος, γράφοντας την τιμή 0xffffffff στον R0.

Με το παράδειγμά μας δείξαμε τον τρόπο λειτουργίας ενός σωρού LIFO. Υπενθυμίζεται ότι το σύστημα ήδη διατηρεί ένα τέτοιο σωρό, χρησιμοποιώντας τον R13 σα stack pointer ενώ παράλληλα παρέχει τις εντολές μεταφοράς block κατ' αναλογία των push & pop. Κάθε φορά δηλαδή που χρησιμοποιούμε την **STMDB** εισάγουμε καταχωρητές στο σωρό του συστήματος και κάθε φορά που χρησιμοποιούμε την **STMIA** αποθηκεύουμε στους καταχωρητές τις τιμές που βρίσκουμε στο σωρό του συστήματος(στη διεύθυνση που δείχνει ο R13).

3.3.4 Σύγκριση αριθμών εύρους 8 bit

Σύγκριση αριθμών χωρίς βρόγχους επανάληψης

```
72  .arm
73  .text
74  .global main
75
76  main:
77  STMDB R13!, {R0-R12}    @Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε
78  LDR R0, =Values         @Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα
                            Values
```

79	**LDRB R1, [R0, #0]**	*@Εισάγουμε στον R1 το περιεχόμενο της μνήμης στο οποίο δείχνει ο R0. Προσέξτε το επίθεμα (LDR)**B** που υποδεικνύει προσπέλαση 1 byte*
80	**LDRB R2, [R0, #1]**	*@Εισάγουμε στον R2 το περιεχόμενο της επόμενης θέσης μνήμης από αυτή στην οποία δείχνει ο R0*
81		
82	**CMP R2, R1**	*@Συγκρίνουμε τον R2 με τον R1, με πράξη εικονικής αφαίρεσης R2-R1*
83	**MOVHI R1, R2**	*@Αν ο R2 > R1 τότε αποθηκεύουμε στον R1 την τιμή του R2*
84		
85	**LDRB R2, [R0, #2]**	*@Εισάγουμε στον R2 το περιεχόμενο της μεθεπόμενης θέσης μνήμης από αυτή στην οποία δείχνει ο R0*
86	**CMP R2, R1**	*@Συγκρίνουμε τον R2 με τον R1, με πράξη εικονικής αφαίρεσης R2-R1*
87	**MOVHI R1, R2**	*@Αν ο R2 > R1 τότε αποθηκεύουμε στον R1 την τιμή του R2*
88		
89	**LDRB R2, [R0, #3]**	*@Εισάγουμε στον R2 το περιεχόμενο της θέσης μνήμης που βρίσκεται 3 θέσεις μετά από αυτή στην οποία δείχνει ο R0*
90	**CMP R2, R1**	*@Συγκρίνουμε τον R2 με τον R1, με πράξη εικονικής αφαίρεσης R2-R1*
91	**MOVHI R1, R2**	*@Αν ο R2 > R1 τότε αποθηκεύουμε στον R1 την τιμή του R2*
92		
93	**LDRB R2, [R0, #4]**	*@Εισάγουμε στον R2 το περιεχόμενο της θέσης μνήμης που βρίσκεται 4 θέσεις μετά από αυτή στην οποία δείχνει ο R0*
94	**CMP R2, R1**	*@Συγκρίνουμε τον R2 με τον R1, με πράξη εικονικής αφαίρεσης R2-R1*
95	**MOVHI R1, R2**	*@Αν ο R2 > R1 τότε αποθηκεύουμε στον R1 την τιμή του R2*
96		
97	**LDR R0, =Result**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Result*
98	**STRH R1, [R0]**	*@Αποθηκεύουμε την τιμή του R1 στη θέση μνήμης που δείχνει ο R0 σαν halfword*
99	**LDMIA R13!, {R0-R12}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε*
100	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η υπορουτίνα main*
101		
102	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "data" που ορίζει αυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*
103	**Values:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Values*
104	**.byte 0x10, 0x15, 0x20**	
105	**.byte 0x25, 0x30**	
106		

107	**.align 2**	*@Είναι απαραίτητο τα επόμενα δεδομένα να τοποθετηθούν σε διευθύνσεις που είναι πολλαπλάσια του 2, διότι θα χρησιμοποιήσουμε halfwords*
108	**Result:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Result*
109	**.hword 0**	*@Εδώ θα αποθηκευτεί ο μεγαλύτερος θετικός αριθμός*

Στο παράδειγμά μας εκτελείται η σύγκριση 5 αριθμών μεταξύ τους και ο μεγαλύτερος αποθηκεύεται στην θέση μνήμης Result. Για την επίλυση του προβλήματός μας χρησιμοποιήσαμε 4 συγκρίσεις. Επειδή όμως, τα βήματα κάθε σύγκρισης είναι ίδια μεταξύ τους (δηλαδή πρόσβαση στη μνήμη για την ανάκτηση του επόμενου αριθμού προς σύγκριση, σύγκριση και τέλος αλλαγή του καταχωρητή με το μεγαλύτερο νούμερο) μπορούμε να χρησιμοποιήσουμε ένα βρόχο επανάληψης και να εκτελέσουμε τις ίδιες εντολές 4 φορές.

Σύγκριση αριθμών με βρόγχους επανάληψης		
110	**.arm**	
111	**.text**	
112	**.global main**	
113		
114	**main:**	
115	**STMDB R13!, {R0-R12}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
116	**LDR R0, =Values**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Values*
117	**LDRB R1, [R0, #0]**	*@Εισάγουμε στον R1 το περιεχόμενο της μνήμης στο οποίο δείχνει ο R0, δηλαδή το πρώτο byte*
118	**ADD R3, R0, #4**	*@Εισάγουμε στον R3 τη διεύθυνση της θέσης μνήμης που βρίσκεται 4 θέσεις μετά από αυτή στην οποία δείχνει ο R0, δηλαδή τη διεύθυνση του τελευταίου byte*
119		
120	**Loop:**	*@Ορίζουμε αυτή τη θέση μνήμης με το όνομα Loop*
121	**LDRB R2, [R0, #1]!**	*@Αποθηκεύουμε στον R2 την τιμή της θέσης μνήμης που βρίσκεται μια θέση μετά από αυτή στην οποία δείχνει ο R0 ενώ παράλληλα αυξάνουμε τον R0 κατά 1*
122	**CMP R2, R1**	*@Συγκρίνουμε τον R2 με τον R1, με πράξη εικονικής αφαίρεσης R2-R1*
123	**MOVHI R1, R2**	*@Αν ήταν R2 > R1 αποθηκεύουμε στον R1 την τιμή του R2*
124		
125	**CMP R0, R3**	*@Συγκρίνουμε τα περιεχόμενα των R0 και R3, με πράξη εικονικής αφαίρεσης R0-R3*
126	**BLO Loop**	*@Αν δεν έχουμε προσπελάσει και το τελευταίο byte μεταβαίνουμε στην εντολή που βρίσκεται στη διεύθυνση μνήμης που είναι σηματοδοτημένη με την ετικέτα Loop*
127		
128	**LDR R0, =Result**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Result*

129	**STRH R1, [R0]**	*@Αποθηκεύουμε την τιμή του R1 στη θέση μνήμης που δείχνει ο R0 σαν halfword*
130	**LDMIA R13!, {R0-R12}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε*
131	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η υπορουτίνα main*
132		
133	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "Data" που ορίζει άυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*
134	**Values:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Values*
135	**.byte 0x10, 0x15, 0x20**	
136	**.byte 0x25, 0x30**	
137		
138	**.align 2**	*@Είναι απαραίτητο τα επόμενα δεδομένα να τοποθετηθούν σε διευθύνσεις που είναι πολλαπλάσια του 2, διότι θα χρησιμοποιήσουμε halfwords*
139	**Result:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Result*
140	**.hword 0**	*@Εδώ θα αποθηκευτεί ο μεγαλύτερος θετικός αριθμός*

Η διαδικασία που ακολουθήσαμε για να παράγουμε το παραπάνω αποτέλεσμα ονομάζεται *folding* και έχει σχέση με την συγκέντρωση όμοιων τμημάτων κώδικα και εκτέλεσή τους με χρήση βρόχων. Στο παράδειγμά μας, όλες οι συγκρίσεις εκτελούνται με την ίδια εντολή, αλλά αυτό που αλλάζει είναι ο καταχωρητής R0, ο οποίος αυξάνεται αυτόματα μετά από κάθε πρόσβαση στη μνήμη και περιέχει πάντα την διεύθυνση του επόμενου στοιχείου προς ανάκτηση. Επιπλέον, χρησιμοποιήσαμε τον καταχωρητή R3 για να αποθηκεύσει το όριο που πρέπει να συναντήσουμε ώστε να φύγουμε από το βρόχο (αν δεν τεθεί σωστά το όριο, η εκτέλεση θα μείνει κολλημένη στον ίδιο βρόχο επ'άπειρο, οπότε χρειάζεται πολύ μεγάλη προσοχή στην επιλογή των ορίων). Ο έλεγχος για το όριο υλοποιείται μέσω της εντολής CMP και του ελέγχου των κατάλληλων σημαιών κατάστασης (CPSR bits). Η διακλάδωση στην ετικέτα Loop: θα γίνει 4 φορές, και μετά η εκτέλεση θα συνεχίσει με την εντολή ανάκτησης της διεύθυνσης του Result.

Με τη χρήση βρόχου υλοποιούμε την επανάληψη μιας διαδικασίας έως ώτου ικανοποιηθεί μια συνθήκη που έχουμε ορίσει. Η γενική μορφή ενός βρόχου είναι :

1. **Έναρξη.**
2. **Κώδικας που τυχόν θα επαναληφθεί.**
3. **Έλεγχος συνθήκης. Αν δεν ικανοποιείται πηγαίνουμε στο (2), διαφορετικά συνεχίζουμε τη ροή του κώδικα.**

3.3.5 Πρόσθεση αριθμών εύρους 64 bit

Πρόσθεση αριθμών		
141	**.arm**	
142	**.text**	
143	**.global main**	
144		
145	**main:**	
146	**STMDB R13!, {R0-R12}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
147	**LDR R0, =Values**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Values*
148	**LDR R1, [R0, #0]**	*@Εισάγουμε στον R1 το περιεχόμενο της μνήμης στο οποίο δείχνει ο R0*
149	**LDR R2, [R0, #8]**	*@Εισάγουμε στον R2 το περιεχόμενο της θέσης μνήμης που βρίσκεται 8 θέσεις μετά από εκεί που δείχνει ο R0*
150	**ADDS R3, R1, R2**	*@Αθροίζουμε τους R1 & R2, αποθηκεύουμε το αποτέλεσμα στον R3 και κρατάμε το κρατούμενο εξόδου στη σημαία κρατούμενου του καταχωρητή κατάστασης, γιατί θα μας χρειαστεί αργότερα*
151	**STR R3, [R0, #0x10]**	*@Αποθηκεύουμε το περιεχόμενο του R3 στη θέση του λιγότερο σημαντικού μέρους του αποτελέσματος, δηλαδή 16 θέσεις μετά από την διεύθυνση μνήμης που δείχνει ο R0*
152		
153	**LDR R1, [R0, #4]**	*@Εισάγουμε στον R1 το περιεχόμενο της θέσης μνήμης που βρίσκεται 4 θέσεις μετά από εκεί που δείχνει ο R0*
154	**LDR R2, [R0, #0xC]**	*@Εισάγουμε στον R2 το περιεχόμενο της θέσης μνήμης που βρίσκεται 12 θέσεις μετά από εκεί που δείχνει ο R0*
155	**ADC R3, R1, R2**	*@Αθροίζουμε τους R1 & R2 και την τιμή που υπάρχει αποθηκευμένη στη σημαία κράτούμενου του καταχωρητή κατάστασης και αποθηκεύουμε το αποτέλεσμα στον R3*
156	**STR R3, [R0, #0x14]**	*@Αποθηκεύουμε το περιεχόμενο του R3 στη θέση του περισσότερο σημαντικού μέρους του αποτελέσματος, δηλαδή 4 θέσεις μετά από την διεύθυνση μνήμης που σηματοδοτεί η ετικέτα Result*
157	**LDMIA R13!, {R0-R12}**	*@Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που χρησιμοποιήσαμε*
158	**MOV PC, R14**	*@Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε η υπορουτίνα main*
159		
160	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "data" που ορίζει αυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*
161	**Values:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Values*
162	**.word 0x1, 0xFF000000**	
163	**.word 0xFFFFFFFF**	
164	**.word 0xFFFFFE**	
165		

166	**Result:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Result*
167	**.word 0,0**	

Σε αυτό το πρόγραμμα ο σκοπός μας ήταν η πρόσθεση δυο αριθμών εύρους 64 bit. Η διαδικασία πρόσθεσης αριθμών μεγαλύτερων των 32 bit περιλαμβάνει τα εξής βήματα:

- **Αποθήκευση των 32 λιγότερο σημαντικών bits που δεν έχουν αθροιστεί ακόμη, σε καταχωρητές.**
- **Πρόσθεση των 2 καταχωρητών με χρήση κρατουμένου (δηλαδή εντολή ADC) και αποθήκευση του κρατούμενου εξόδου στον CPSR.**
- **Αν υπάρχουν bits που δεν έχουν προστεθεί, επανάληψη από το πρώτο βήμα.**

Στα παραπάνω βήματα χρειάζεται προσοχή στην έναρξη του αλγόριθμου, διότι το CPSR Carry bit πρέπει να είναι 0 (η πρόσθεση ξεκινά χωρίς κρατούμενο εισόδου). Επιπλέον, είναι προτεινόμενη η υλοποίηση με βρόχο επανάληψης, ώστε να μην επαναλαμβάνεται άσκοπα ο ίδιος κώδικας. Στο πρόγραμμά μας επειδή προσθέσαμε 64 bit αριθμούς, δεν ήταν απαραίτητη η χρήση βρόχου, αλλά αν προσθέταμε 128 ή 256 bit αριθμούς, θα ήταν επιβεβλημένη.

Carry In	**1**	**0**
Number1	**0xFF000000**	**0x00000001**
Number2	**0x00FFFFFE**	**0xFFFFFFFF**
Result	**0xFFFFFFFF**	**0x00000000**

Σχήμα 3.10: 64 bit addition

Στην εικόνα παρουσιάζεται η πρόσθεση που εκτελέσαμε. Παρατηρείστε πως η πρώτη πρόσθεση δημιουργεί κρατούμενο εξόδου, το οποίο εισέρχεται στην επόμενη βαθμίδα.

3.3.6 Διαίρεση αριθμού εύρους 32 bit με 16 bit διαιρέτη

Διαίρεση αριθμού		
1	**.arm**	
2	**.text**	
3	**.global main**	
4		
5	**main:**	
6	**STMDB R13!, {R0-R12}**	*@Αποθηκεύουμε τους καταχωρητές που θα χρησιμοποιήσουμε*
7		
8	**LDR R0, =Values**	*@Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα Values*
9		

```
10  LDR R1, [R0], #4        @Αποθηκεύουμε στον R1 το περιεχόμενο της θέσης μνήμης στην
                            οποία δείχνει ο R0 και αυξάνουμε τον R0 κατά 4
11  LDRH R2, [R0]           @Αποθηκεύουμε στον R2 το περιεχόμενο της θέσης μνήμης στην
                            οποία δείχνει ο R0 σαν halfword
12  CMP R2, #0              @Ελέγχουμε αν ο διαιρέτης είναι μηδέν
13  BEQ Err                 @Αν είναι 0, μεταβαίνουμε στην θέση μνήμης που σηματοδοτεί-
                            ται από την ετικέτα Err
14
15  MOV R3, #0              @Αποθηκεύουμε την τιμή 0 στον R3
16  CLZ R6, R2              @Αποθηκεύουμε στον R6 τον αριθμό των συνεχόμενων 0 που
                            βρίσκονται στις πιο σημαντικές θέσεις του R2
17  MOV R2, R2, LSL R6      @Ολισθαίνουμε προς τα αριστερά τον R2 κατά τόσες θέσεις,
                            όσες έχουμε αποθηκεύσει στον R6. Δηλαδή φέρνουμε το πιο
                            σημαντικό 1 του αριθμού στην πιο αριστερή θέση
18
19  Loop:                   @Ορίζουμε αυτή τη θέση μνήμης με το όνομα Loop
20  MOV R3, R3, LSL #1      @Ολισθαίνουμε τον R3 προς τα αριστερά κατά 1
21  CMP R1, R2              @Συγκρίνουμε τον R1 με τον R2, με πράξη εικονικής αφαίρεσης
                            R1-R2
22  ORRCS R3, R3, #1        @Αν ο R1 >= R2 τότε εκτελούμε την λογική πράξη OR του R3
                            με τον αριθμό 1
23  SUBCS R1, R1, R2        @Αν ο R1 >= R2 τότε αφαιρούμε το R2 από το R1
24
25  MOV R2, R2, LSR #1      @Ολισθαίνουμε τον R2 προς τα δεξιά κατά 1
26  SUBS R6, R6, #1         @Αφαιρούμε από το R6 τον αριθμό 1 και αποθηκεύουμε το
                            αποτέλεσμα στον R6
27  BCS Loop                @Αν ο R6 είναι μεγαλύτερος του 0, μεταβαίνουμε στη θέση
                            μνήμης που σηματοδοτείται από την ετικέτα Loop
28
29  B Done                  @Μεταβαίνουμε στη θέση μνήμης που σηματοδοτείται από την
                            ετικέτα Done
30  Err:                    @Ορίζουμε αυτή τη θέση μνήμης με το όνομα Err
31  MVN R3, #0              @Εισάγουμε στον R3 την τιμή -1
32  Done:                   @Ορίζουμε αυτή τη θέση μνήμης με το όνομα Done
33  LDR R0, =Result         @Εισάγουμε στον R0 τη διεύθυνση που σηματοδοτεί η ετικέτα
                            Result
34  STR R3, [R0], #4        @Αποθηκεύουμε στη θέση μνήμη στην οποία δείχνει ο R0 το
                            περιεχόμενο του R3 και αυξάνουμε τον R0 κατά 4
35  STRH R1, [R0]           @Αποθηκεύουμε στη θέση μνήμη στην οποία δείχνει ο R0 το
                            περιεχόμενο του R1 σαν halfword
36
37  LDMIA R13!, {R0-R12}    @Επαναφέρουμε τις αρχικές τιμές στους καταχωρητές που
                            χρησιμοποιήσαμε
38  MOV PC, R14             @Μεταφέρουμε τη ροή της εκτέλεσης στο σημείο όπου κλήθηκε
                            η υπορουτίνα main
39
```

40	**.data**	*@Ορίζουμε ότι τα παρακάτω δεδομένα θα τοποθετηθούν στην περιοχή "Data" που ορίζει άυτόματα το λειτουργικό σύστημα, για τα δεδομένα των προγραμμάτων*
41	**Values:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Values*
42	**.word 0xFFFFFFFE**	
43	**.hword 0xF**	
44		
45	**.align 4**	*@Είναι απαραίτητο τα επόμενα δεδομένα να τοποθετηθούν σε διευθύνσεις που είναι πολλαπλάσια του 4*
46	**Result:**	*@Τα παρακάτω δεδομένα τοποθετούνται σε μνήμη που σημειώνεται από την ετικέτα Result*
47	**.word 0**	
48	**.hword 0**	

Ο αλγόριθμος της διαίρεσης χρησιμοποιεί μια σειρά από αφαιρέσεις για να υπολογίσει το πηλίκο και το υπόλοιπο. Αν χρησιμοποιήσουμε έναν απλοικό αλγόριθμο, μπορούμε να αρχίσουμε να αφαιρούμε το διαιρέτη από το διαιρεταίο και να μετράμε τις αφαιρέσεις, μέχρι να γίνει ο διαιρέτης μικρότερος από το διαιρεταίο. Αυτή η προσέγγιση απαιτεί τόσες αφαιρέσεις, όσες και το πηλίκο, δηλαδή στην χειρότερη περίπτωση $2^{32} - 1$ (στην περίπτωση που διαιρέσουμε το $2^{32} - 1$ με το 1). Για να επισπεύσουμε τη διαδικασία στον παραπάνω αλγόριθμο αναλύουμε τον διαιρεταίο σε πολλαπλάσια του διαιρέτη ως εξής:

$$\mathbf{D} = x_{31} * \mathbf{d} * 2^{31} + \ldots + x_i * \mathbf{d} * 2^i + \ldots + \mathbf{r}$$

Ο όρος D συμβολίζει τον διαιρεταίο, ο d τον διαιρέτη και ο r το υπόλοιπο. Το πηλίκο σχηματίζεται από τους όρους:

$$\mathbf{q} = x_{31} * 2^{31} + \ldots + x_i * 2^i + \ldots + x_0$$

Για να σχηματίσουμε τους όρους D = x_i * d * 2^i εκτελούμε ολίσθηση στον διαιρέτη, ξεκινώντας από τη μεγαλύτερη τιμή. Η μεγαλύτερη τιμή που μπορούμε να πάρουμε είναι η κανονικοποιημένη μορφή του διαιρέτη[21], συνεπώς τιμές μεγαλύτερες από αυτή θα είναι σίγουρα μεγαλύτερες από τον διαιρεταίο και θα επιστρέψουν στον συντελεστή τους την τιμή 0, οπότε δεν χρειάζεται να τις εξετάσουμε. Αρχίζουμε να συγκρίνουμε αν η κανονικοποιημένη μορφή είναι μικρότερη ή ίση από τον διαιρεταίο και αν είναι εκτελούμε αφαίρεση σε αυτές. Για παράδειγμα, αν ο διαιρεταίος είναι ο 0xFE και ο διαιρέτης το 3, κατά την κανονικοποίηση θα μετατρέψουμε τον διαιρέτη σε 0xC0. Ο 0xC0 είναι μικρότερος από τον 0xFE, άρα θα εκτελέσουμε αφαίρεση ανάμεσά τους και ο νέος διαιρεταίος θα είναι η διαφορά τους. Διαπιστώσαμε δηλαδή πως ο διαιρεταίος μας περιέχει την τιμή x_6 * d * 2^6, όπου το x_6 = 1. Μετά την αφαίρεση, ολισθαίνουμε δεξιά τον διαιρέτη (γιατί πλέον ο διαιρεταίος είναι μικρότερος από αυτόν) και αναζητάμε την τιμή x_5 * d * 2^5, με την ίδια ακριβώς διαδικασία. Στο τέλος, όταν φτάσουμε στον τελευταίο συντελεστή, το αποτέλεσμα

[21]Δηλαδή, όταν έχει ολισθήσει τόσες θέσεις αριστερά, ώστε το πιο σημαντικό 1 να έχει έρθει στη θέση 31

της αφαίρεσης θα είναι μικρότερο από το διαιρέτη (ο οποίος θα έχει επανέρθει στην αρχική τιμή του, δηλαδή θα έχει ολισθήσει δεξιά τόσες θέσεις όσες είχε ολισθήσει αριστερά στην αρχή) και θα αποτελεί το υπόλοιπο της διαίρεσης. Ο σχηματισμός του πηλίκου μπορεί να γίνει με 2 τρόπους, με τον πρώτο να είναι η ενεργοποίηση του αντίστοιχου bit του καταχωρητή (πχ. για το συντελεστή 6 θα ενεργοποιήσουμε το 7ο bit κλπ). Ο δεύτερος τρόπος είναι η ενεργοποίηση του λιγότερου σημαντικού bit του καταχωρητή που περιέχει το πηλίκο και η ολίσθηση της τιμής σε κάθε επανάληψη του αλγορίθμου. Για παράδειγμα, αν υπάρχει ο μεγαλύτερος συντελεστής, ο καταχωρητής πηλίκου θα περιέχει την τιμή 1. Αν δεν υπάρχει ο αμέσως μικρότερος συντελεστής, ο καταχωρητής θα ολισθήσει αριστερά κατά μια θέση και στο λιγότερο σημαντικό bit δεν θα εισαχθεί τίποτα (θα μείνει 0). Στο τέλος του αλγορίθμου, το bit που εισάγαμε πρώτο θα έχει φτάσει στην 7η θέση, το επόμενο στην 6η κλπ.

Περιβάλλον εργασίας

Με το γενικό τίτλο «περιβάλλον εργασίας» αναφερόμαστε σύντομα σε αυτό το κεφάλαιο στην πλατφόρμα υλικού, στο λειτουργικό σύστημα, στα εργαλεία ανάπτυξης και αποσφαλμάτωσης και στα εργαλεία συγγραφής του κώδικα. Η περιγραφή μας παραμένει συνοπτική για το καθένα, καθώς το AT91 συμπεριφέρεται σαν οποιοδήποτε άλλο Unix σύστημα.

4.1 Υλικό

Το AT91 είναι μια εμπορικά διαθέσιμη πλακέτα ανάπτυξης συστημάτων, σχεδιασμένη από την εταιρεία ATMEL. Πέρα από το ολοκληρωμένο του μικροελεγκτή AT91SAM9261, παρέχει ένα ευρύ σύνολο από περιφερειακά (μετατροπείς από αναλογικό σε ψηφιακό, προσαρμογείς για USB, Ethernet, ...) που διευκολύνουν τη διασύνδεση του συστήματός μας.

4.2 Λειτουργικό σύστημα

Το λειτουργικό σύστημα που υιοθετήθηκε για τη πρώτη έκδοση του AT91, είναι το Linux (Snapgear distribution), με πυρήνα στην έκδοση 2.6.20. Πρόκειται για μια έκδοση του Linux αναπτυγμένη για τον επεξεργαστή ARM926EJ-S, με υποστήριξη αρκετών προγραμμάτων οδηγών, με προσαρμογές των οποίων το λειτουργικό σύστημα μπορεί να προσπελάσει τις διάφορες περιφερειακές συσκευές και εξωτερικές μνήμες (LCD οθόνη, USB θύρα, μνήμη Flash κλπ). Η πρώτη οθόνη που εμφανίζεται μετά την έναρξη λειτουργίας του συστήματος είναι η οθόνη εισαγωγής στο σύστημα στην οποία ο χρήστης πρέπει να εισάγει το όνομα χρήστη (login) **root** και αμέσως μετά το κωδικό **support**, ο οποίος δεν εμφανίζεται στην οθόνη κατά την πληκτρολόγηση. Με την εισαγωγή των στοιχείων, ο χρήστης εισέρχεται στο shell του λειτουργικού, όπου μπορεί να εισάγει οποιαδήποτε εντολή υποστηρίζεται από το Linux. Το περιβάλλον δεν είναι γραφικό και έτσι οι χρήστες πρέπει να εισάγουν εντολές στο shell του συστήματος για να εκτελεστούν οι διάφορες λειτουργίες.

Οι βασικές εντολές που υποστηρίζονται καταγράφονται στον πίνακα *Εντολές Linux* (κεφ. 4.2, σελ. 62).

Cmd	**Περιγραφή**	**Cmd**	**Περιγραφή**
as	Εκτελεί τον assembler	**cd**	Αλλαγή καταλόγου
cat	Προβολή δεδομένων αρχείου	**cp**	Αντιγραφή αρχείων
df	Ελεύθερος χώρος αποθήκευσης	**kill**	Τερματισμός διεργασίας
ld	GNU Linker	**ls**	Λίστα αρχείων καταλόγου
make	Μετάφραση κώδικα	**mkdir**	Δημιουργεία καταλόγου
mount	Δέσμευση περιφεριακού	**mv**	Μετακίνηση αρχείου
nm	GNU πληροφορίες εκτελέσιμου	**ps**	Λίστα διεργασιών
rm	Διαγραφή αρχείων	**umount**	Αποδέσμευση περιφεριακού

Πίνακας 4.24: Εντολές Linux

cat

Με την εντολή cat εμφανίζουμε στην οθόνη το περιεχόμενο ενός αρχείου. Η σύνταξη της εντολής είναι:

cat $< filename >$

όπου το $< filename >$ είναι το όνομα του αρχείου. Για παράδειγμα, η εντολή *cat /etc/-motd* θα προβάλει το λογότυπο που εμφανίζεται κατά την εισαγωγή στο σύστημα.

cd

Με την εντολή cd μεταβαίνουμε από τον τρέχοντα κατάλογο σε έναν άλλο. Η σύνταξη της εντολής είναι:

cd $< destination >$

όπου το $< destination >$ είναι ο προορισμός μας. Για παράδειγμα, η εντολή *cd ..* θα μας μεταφέρει στον κατάλογο που περιέχει τον τρέχοντα, στον οποίο βρισκόμαστε.

cp

Με την εντολή cp αντιγράφουμε ένα αρχείο σε μια άλλη θέση, στο σύστημα αρχείων. Η σύνταξη της εντολής είναι:

cp $< filename > < destination >$

όπου το $< filename >$ είναι το όνομα του αρχείου και το $< destination >$ ο προορισμός του. Για παράδειγμα, η εντολή *cp /etc/motd /storage* θα αντιγράψει το αρχείο /etc/motd στον κατάλογο /storage.

df

Με την εντολή df εμφανίζουμε στην οθόνη τον ελεύθερο χώρο στο σύστημα αρχείων (file system). Η εντολή δεν χρειάζεται παραμέτρους.

kill

Με την εντολή kill στέλνουμε σήματα (software signals) σε διεργασίες. Η σύνταξη της εντολής είναι:

kill -$< signum > < pid >$

όπου το $< signum >$ είναι ο αριθμός του σήματος που θέλουμε να στείλουμε και $< pid >$ ο αριθμός της διεργασίας. Για παράδειγμα, η εντολή *kill -9 38* θα στείλει το σήμα τύπου 9 στην διεργασία 38 (το σήμα τύπου 9 είναι σήμα τερματισμού και συνεπώς θα σταματήσει τη διεργασία).

ls

Με την εντολή ls εμφανίζουμε τα αρχεία του τρέχοντος καταλόγου. Η σύνταξη της εντολής είναι:

ls [-alF]

όπου το -alF αφορά την εμφάνιση των αρχείων σε λίστα με πληροφορίες για μέγεθος και είναι προαιρετικό.

make

Με την εντολή make μεταγλωτίζουμε πηγαίο κώδικα σε εκτελέσιμο πρόγραμμα. Το script αναλαμβάνει να καλέσει τα κατάλληλα εργαλεία και να εισάγει τις κατάλληλες παραμέτρους, ώστε να ολοκληρωθεί αυτή η διαδικασία. Η σύνταξη της εντολής είναι:

make $tool < srcfile > < destfile >$

$tool$ είναι το εργαλείο μεταγλώττισης. Για το εργαστήριο προγραμματισμού σε συμβολική γλώσσα το εργαλείο αυτό είναι ο as (gnu assembler). Για το εργαστήριο μικροϋπολογιστών θα χρησιμοποιήσετε ως εργαλείο μεταγλώττισης τον gcc (gnu C compiler). $< srcfile >$ είναι το όνομα του αρχείου με τον πηγαίο κώδικά μας. Για αρχεία assembly είναι καλό να δίνετε τη κατάληξη .s, ενώ για αρχεία γλώσσας C, την κατάληξη .c Τέλος, $< destfile >$ το όνομα του τελικού εκτελέσιμου. Για παράδειγμα, η εντολή `make as foo.s bar` μεταγλωτίζει το πηγαίο αρχείο συμβολικής γλώσσας foo.s, παράγει το ενδιάμεσο αρχείο bar.o (object αρχείο) και τελικά το εκτελέσιμο bar.

mkdir

Με την εντολή mkdir δημιουργούμε νέους καταλόγους. Η σύνταξη της εντολής είναι:

mkdir $< folder >$

όπου το $< folder >$ είναι το όνομα του καταλόγου. Για παράδειγμα, η εντολή *mkdir exer1* δημιουργεί τον κατάλογο exer1 μέσα στον τρέχοντα κατάλογο.

mount

Με την εντολή mount εισάγουμε περιφερειακά, όπως USB stick στο σύστημα αρχείων. Η σύνταξη της εντολής είναι:

mount $< device > < folder >$

όπου το $< device >$ είναι το όνομα της συσκευής και το $< folder >$ είναι το όνομα υπάρχοντος καταλόγου, μέσω του οποίου θα προσπελαύνουμε τα περιεχόμενα της συσκευής. Για παράδειγμα, για να συνδέσουμε ένα USB stick στον κατάλογο mnt θα πρέπει να δώσουμε είτε την εντολή *mount /dev/sda /mnt* είτε την εντολή *mount /dev/sda1 /mnt*, ανάλογα με τη χωρητικότητα του stick μας. Με τη χρήση του cd μπορούμε να μεταβούμε σε αυτόν τον κατάλογο και με το ls να δούμε τα αρχεία που υπάρχουν στο stick.

mv

Με την εντολή mv μετακινούμε ένα αρχείο σε μια άλλη θέση, στο σύστημα αρχείων. Η σύνταξη της εντολής είναι:

mv $< filename > < destination >$

όπου το $< filename >$ είναι το όνομα του αρχείου και το $< destination >$ ο προορισμός του. Για παράδειγμα, η εντολή *mv /etc/motd /storage* θα μεταφέρει το αρχείο /etc/motd στον κατάλογο /storage και θα το διαγράψει από την αρχική του θέση. Με αυτή την εντολή μπορούμε να μετονομάζουμε και αρχεία, αν τα μετακινούμε στον ίδιο κατάλογο (πχ. *mv /etc/motd /etc/hello.txt*).

ps

Με την εντολή ps προβάλλουμε τις διεργασίες που εκτελούνται στο σύστημα. Η εντολή δεν χρειάζεται παραμέτρους.

rm

Με την εντολή rm διαγράφουμε αρχεία και καταλόγους. Η σύνταξη της εντολής είναι:

rm [-r] $< filename >$

όπου το -r είναι προαιρετικό (χρησιμοποιείται όταν θέλουμε να διαγράψουμε ένα κατάλογο) και το $< filename >$ είναι το όνομα του αρχείου/καταλόγου που θα διαγραφεί (πχ. **rm ask1.s** θα διαγράψει το αρχείο ask1.s).

umount

Με την εντολή umount αποδεσμεύουμε περιφερειακά από το σύστημα αρχείων, όπως για παράδειγμα το USB stick. Η διαδικασία αποδέσμευσης είναι απαραίτητη για την ενημέρωση των περιφερειακών με τα σωστά δεδομένα και τον ασφαλή τους τερματισμό. Η σύνταξη της εντολής είναι:

umount $< folder >$

όπου το $< folder >$ είναι το όνομα υπάρχοντος καταλόγου, στον οποίο έχουμε δεσμεύσει μια συσκευή. Για παράδειγμα, η εντολή *umount /mnt* αποσυνδέει ένα USB stick που είχε δεσμευτεί στον κατάλογο mnt. Για να μπορέσει να ολοκληρωθεί σωστά η εντολή δε θα πρέπει αυτός ο κατάλογος ή οποιοσδήποτε υποκατάλογός του να μη χρησιμοποιείται από κανένα πρόγραμμα, ούτε καν από το λειτουργικό σύστημα. Δηλαδή θα πρέπει να εκτελέσουμε την *umount /mnt* βρισκόμενοι σε άλλο κατάλογο του συστήματός μας.

4.3 GNU tools

Τα εργαλεία ανάπτυξης περιλαμβάνουν τον Assembler, το Linker, το Debugger και τον InfoViewer. Τα δυο πρώτα εργαλεία καλούνται από το αυτοματοποιημένο αρχείο εντολών *make*, ο Debugger με την εντολή gdb (στη τρέχουσα έκδοση firmware του AT91 δυστυχώς πρέπει να τον καλείτε ως `/storage/.gnu/gdb`) και ο InfoViewer με την εντολή nm. Ο Debugger αποτελεί το βασικό εργαλείο βηματικής εκτέλεσης και εξέτασης των καταχωρητών και των μνημών[22]. Η σύνταξη της εντολής είναι **/storage/.gnu/gdb** $< executable >$, όπου το $< executable >$ είναι το όνομα του εκτελέσιμου αρχείου. Ο Debugger δέχεται εντολές μέσω της κονσόλας, που καθορίζουν τις ενέργειες που θα εκτελέσει

[22] Στο AT91 δεν υπάρχει η δυνατότητα εξέτασης συγκεκριμένης θέσης μνήμης από το shell, λόγω της ενσωμάτωσης της μονάδας διαχείρισης μνήμης (MMU) στο ολοκληρωμένο του μικροελεγκτή. Σε επίπεδο λογισμικού μόνο ιδεατές διευθύνσεις είναι διαθέσιμες και αυτές μόνο όταν μια συγκεκριμένη διαδικασία εκτελείται. Ο τερματισμός μιας διαδικασίας σημαίνει και την αποδέσμευση του χώρου μνήμης που κατέλαβε για την εκτέλεσή της.

στη συνέχεια. Οι εντολές αυτές σχετίζονται με σημεία παύσης (breakpoints) και σημεία ελέγχου (watchpoints), καθώς και πληροφορίες για το περιεχόμενο των θέσεων μνήμης και καταχωρητών.

Περιγράφουμε παρακάτω τις δυνατότητες που μας δίνονται εντός του περιβάλλοντος εκτέλεσης του Debugger, αφού δηλαδή δώσουμε την εντολή

/storage/.gnu/gdb $< file >$

όπου $< file >$ είναι το εκτελέσιμο αρχείο που προσπαθούμε να τρέξουμε βηματικά γα αποσφαλμάτωση ή για να αποδείξουμε την ορθή λειτουργία του. Το πρόγραμμα μας, κατά την εκκίνηση του Debugger, βρίσκεται σε ανενεργή κατάσταση και δεν έχει αρχίσει να εκτελείται. Για να ξεκινήσει χρησιμοποιούμε την εντολή **run**. Το πρόγραμμα θα ολοκληρωθεί και ο Debugger θα ειδοποιήσει πως το πρόγραμμα τερματίστηκε. Για να διακόψουμε τη ροή της εκτέλεσης και να επιστρέψουμε στον Debugger πρέπει να τοποθετήσουμε ένα breakpoint σε όποια γραμμή κώδικα θέλουμε να σταματήσει η εκτέλεση του προγράμματος. Μόλις σταματήσει, μπορούμε να δούμε το περιεχόμενο των καταχωρητών και οποιασδήποτε θέσης μνήμης, όπως και να αλλάξουμε τα περιεχόμενα τους. Τα breakpoints μπορούμε να τα εισάγουμε με την εντολή **break**. Οι τρόποι εισαγωγής τους εμφανίζονται στον πίνακα *Breakpoints* (κεφ. 4.3, σελ. 66).

Breakpoint	**Παράδειγμα**
break <label>	Τοποθετεί το breakpoint στη θέση που χαρακτηρίζεται από την ετικέτα <label>.
break <line>	Τοποθετεί το breakpoint στην γραμμή <line> του πηγαίου αρχείου.
break <*address>	Τοποθετεί το breakpoint στην διεύθυνση <address> της μνήμης.

Πίνακας 4.25: Καταχωρητές

Για παράδειγμα, με την εντολή *break main* τοποθετούμε ένα breakpoint στην πρώτη εντολή του προγράμματός μας, κάτι το οποίο θα μας χρησιμεύσει πολλές φορές και είναι επιθυμητό να γίνεται πριν την έναρξη εκτέλεσης του προγράμματος με τη *run*. Αν γνωρίζουμε ότι υπάρχει μια εντολή πρόσβασης στη μνήμη, στη θέση μνήμης 0x80c0 μπορούμε να τοποθετήσουμε ένα breakpoint εκεί, με την εντολή *break *0x80c0*. Αν η πρόσβαση αυτή γίνεται στη γραμμή 15 του πηγαίου κώδικα, μπορούμε να χρησιμοποιήσουμε την εναλλακτική εντολή *break 15*. Τέλος, αν θέλουμε να τοποθετήσουμε ένα breakpoint 3 εντολές μετά από μια ετικέτα (πχ. main), χρησιμοποιούμε την εντολή *break *(main+12)*, όπου το 12 σημαίνει 12 bytes μετά από την ετικέτα, λόγω του ότι κάθε εντολή καταλαμβάνει 4 bytes.

Για να προβάλλουμε τη λίστα με όλα τα breakpoints που έχουμε θέσει, χρησιμοποιούμε την εντολή *info break [n]*, η οποία παρουσιάζει αριθμημένα τα breakpoints[23] και τη θέση στην οποία βρίσκονται. Αν εισάγουμε και τον προαιρετικό αριθμό n, θα προβληθούν μόνο οι πληροφορίες για το n breakpoint (πχ. με την εντολή *info break 3* θα προβάλουμε τις πληροφορίες του τρίτου breakpoint). Η διαγραφή ενός breakpoint γίνεται με την εντολή *clear <line>*, όπου line είναι ο αριθμός της γραμμής στην οποία έχει τεθεί το breakpoint (πχ. με την εντολή *clear 12* θα διαγραφεί το breakpoint που έχει τεθεί στη γραμμή 12). Ο εναλλακτικός τρόπος διαγραφής ενός breakpoint γίνεται με την εντολή *delete <breaknum>*, όπου το breaknum είναι ο αριθμός του breakpoint (πχ. με την εντολή *delete 2* θα διαγραφεί το 2ο breakpoint).

Για να εμφανίσουμε τα περιεχόμενα μιας θέσης μνήμης χρησιμοποιούμε την εντολή *x/nfu* $< address >$, όπου το n καθορίζει το πόσες διαδοχικές τιμές θα προβληθούν, το f τον τρόπο προβολής και το u το μέγεθος των τιμών. Ο τρόπος προβολής μπορεί να είναι ένας από τους ακόλουθους:

x Δεκαεξαδική προβολή

d Προσημασμένη δεκαδική προβολή

u Μη-προσημασμένη δεκαδική προβολή

o Οκταδική προβολή

t Δυαδική προβολή

a Προβολή διεύθυνσης

c Προβολή χαρακτήρα ASCII

s Προβολή string

Ο τρόπος προβολής σχετίζεται με το πως επιθυμούμε να βλέπουμε τις τιμές και δεν επηρρεάζει το μέγεθος τους. Το επιθυμητό μέγεθος επιλέγεται με τη χρήση των παρακάτω εντολών:

b Κάθε τιμή έχει μέγεθος 1 byte

h Κάθε τιμή έχει μέγεθος 2 bytes (halfword). Ξεκινά σε διεύθυνση που είναι ακέραιο πολλαπλάσιο του 2.

w Κάθε τιμή έχει μέγεθος 4 bytes (word). Ξεκινά σε διεύθυνση που είναι ακέραιο πολλαπλάσιο του 4.

g Κάθε τιμή έχει μέγεθος 8 bytes (giantword). Ξεκινά σε διεύθυνση που είναι ακέραιο πολλαπλάσιο του 8.

Το πεδίο διεύθυνση address της εντολής μπορεί να είναι απόλυτη διεύθυνση (πχ. x/wx 0x10544), ή σχετική με τη χρήση ετικετών (πχ. x/wx (&Values + 2)). Τα παραδείγματα που αναγράφονται στον πίνακα *Προβολή τιμών* (κεφ. 4.3, σελ. 68), δείχνουν διάφορους τρόπους προβολής τιμών μνήμης.

[23] Η αρίθμηση γίνεται με βάση τη σειρά εισαγωγής, των breakpoints

Εστω ότι στο .data κομμάτι του κώδικά μας έχουμε δηλώσει :

Values:

.word 0x40302010, 0x80706050

.word 0x54204948, 0x45524548

Εντολή	**Παράδειγμα**
x/2xw &Values	Θα εμφανίσει τις τιμές 0x40302010 & 0x80706050
x/3xb ((char*)&Values+1)	Θα εμφανίσει τις τιμές 0x20, 0x30, 0x40. Προσέξτε πως χρησιμοποιήσαμε το casting (char*) για να ενημερώσουμε τον Debugger πως η διεύθυνση αναφέρεται σε bytes και το b για να πάρουμε μόνο 1 byte για κάθε τιμή. Αν δεν χρησιμοποιούσαμε το (char*) θα εμφανίζαμε τις τιμές 0x50, 0x60, 0x70, διότι η διεύθυνση θα ξεκίναγε από 1 word μετά το Values και όχι ένα byte, όπως εμείς θέλαμε.
x/2xh ((short*)&Values+1)	Θα εμφανίσει τις τιμές 0x4030, 0x6050. Εδώ χρησιμοποιήσαμε το casting (short*) για να ενημερώσουμε τον Debugger πως η διεύθυνση αναφέρεται σε halfwords και το h για να πάρουμε 2 bytes για κάθε τιμή. Αν δεν χρησιμοποιούσαμε το (short*) θα εμφανίζαμε τις τιμές 0x6050, 0x8070, διότι η διεύθυνση θα ξεκίναγε από 1 word μετά το Values και όχι ένα halfword, όπως εμείς θέλαμε.
x/dh &Values	Θα εμφανίσει την τιμή 8208 (δεκαδική τιμή του halfword 0x2010). Δεν χρησιμοποιούμε casting διότι δεν προσθέτουμε κάποια τιμή στην ετικέτα.
x/dh ((short*)&Values+3)	Θα εμφανίσει την τιμή -32656 (προσημασμένη δεκαδική).
x/uh ((short*)&Values+3)	Θα εμφανίσει την τιμή 32880 (μη-προσημασμένη δεκαδική).
x/s &Values+2	Θα εμφανίσει το string HI THERE.

Πίνακας 4.27: Προβολή τιμών

Ένας εναλλακτικός τρόπος προβολής των δεδομένων είναι με τη χρήση της εντολής *p/f* $< variable >$, όπου εδώ μπορούμε να εισάγουμε μόνο τον τρόπο προβολής και για να επιλέξουμε το μέγεθος και τη διεύθυνση χρησιμοποιούμε τροποποίηση τύπου (casting). Η τυπική λειτουργία της εντολής προβάλει ένα word, το οποίο είναι το περιεχόμενο της ετικέτας που εισάγουμε σαν παράμετρο. Για παράδειγμα, με την εντολή *p/x Values* θα εμφανιστεί η τιμή 0x40302010. Για να εμφανίσουμε το τρίτο byte, η εντολή γίνεται *p/x *((char*)&Values+2)*. Το πλεονέκτημα της εντολής αυτής είναι ότι μπορεί να προβάλει το περιεχόμενο των καταχωρητών, όπως *p/f* $\$< register >$. Για παράδειγμα, με την εντολή *p/x $r0* μπορούμε να εξετάσουμε το περιεχόμενο του R0, με την εντολή *p/x $pc* μπορούμε

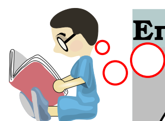
να εξετάσουμε το περιεχόμενο του R15 και με την εντολή *p/t $cpsr* μπορούμε να εμφανίσουμε δυαδικά το περιεχόμενο του CPSR. Ένας δεύτερος τρόπος προβολής όλων των καταχωρητών είναι με την εντολή *info registers*, όπου προβάλεται το περιεχόμενο όλων των καταχωρητών.

Μέσα από τον Debugger μπορούμε φυσικά και να αλλάζουμε τις τιμές των θέσεων μνήμης και των καταχωρητών, με την εντολή *set* $< expr >$, όπου expr είναι μια έκφραση ανάθεσης, όπως η set Values = 0x450 που θέτει την σημειωμένη από την ετικέτα Values θέση μνήμης στην τιμή 0x450. Ένα αντίστοιχο παράδειγμα με καταχωρητές αποτελεί η εντολή set $r0 = 0xFF, όπου αλλάζει την τιμή του R0 σε 255. Οι θέσεις μνήμης που μπορούν να επιλεγούν, σχηματίζονται και με casting ανάλογα με τη θέση τους και το εύρος της τιμής, όπως με την εντολή set *((char*)&Values+2) = 0x45, όπου θέτει το byte που βρίσκεται 2 θέσεις μετά από τη διεύθυνση που σημειώνει η ετικέτα, στην τιμή 0x45.

Για να εξετάσουμε τα αποτελέσματα που παράγονται από τις εντολές του προγράμματός μας, έχουμε τη δυνατότητα να συνεχίσουμε την εκτέλεση του προγράμματος βηματικά (δηλαδή μια προς μια τις εντολές) ή κανονικά (όπου η εκτέλεση σταματά με την ολοκλήρωση του προγράμματος ή με την συνάντηση κάποιου breakpoint). Η κανονική εκτέλεση, μετά από διακοπή, γίνεται με την εντολή continue ή μόνο c, ενώ η βηματική εκτέλεση γίνεται με την εντολή step. Σε περιπτώσεις όπου υπάρχουν κλήσεις σε υπορουτίνες, η εντολή step θα ακολουθήσει τη διακλάδωση και θα σταματήσει στην πρώτη εντολή της υπορουτίνας, κάτι το οποίο δεν είναι πάντα επιθυμητό. Γι'αυτό το λόγο μπορούμε να χρησιμοποιήσουμε την εντολή next ή n, η οποία έχει την ίδια λειτουργία με την step, αλλά όταν συναντά κλήσεις σε υπορουτίνες εκτελεί κανονικά τον κώδικα και σταματά στην επόμενη γραμμή από την κλήση (δεν κάνει βηματική εκτέλεση του κώδικα της υπορουτίνας).

Μπορούμε τέλος μέσα από τον Debugger να θυμηθούμε ένα τμήμα από εντολές του πηγαίου μας αρχείου, με τη χρήση της εντολής *list [<line_start>] [,<line_end>]*, όπου αν δεν εισάγουμε καμία παράμετρο εμφανίζονται 10 γραμμές κώδικα γύρω από την γραμμή προς εκτέλεση, αν εισάγουμε την πρώτη παράμετρο εμφανίζονται 10 γραμμές κώδικα γύρω από την γραμμή που εισάγαμε και αν εισάγουμε και τις 2 παραμέτρους εμφανίζονται οι γραμμές κώδικα ανάμεσα στην πρώτη και τη δεύτερη γραμμή. Για παράδειγμα, με την εντολή list 15 θα εμφανιστούν οι γραμμές 10. . . 19, ενώ με την εντολή list 15,18 θα εμφανιστούν οι γραμμές 15. . . 18. Παρόμοια λειτουργία έχει και η εντολή *disassemble [<addr_start>] [<addr_end>]*, με τη διαφορά πως τώρα εμφανίζεται ο κώδικας στις θέσεις μνήμης που έχει φορτωθεί. Οι παράμετροι τώρα είναι διευθύνσεις μνήμης και αν δεν εισάγουμε καμία, θα εμφανιστεί ο κώδικας που υπάρχει σε 10 θέσεις γύρω από την τρέχουσα εντολή προς εκτέλεση. Αν εισάγουμε την πρώτη θα εμφανιστεί ο κώδικας που περικλείει τις 10 θέσεις

γύρω από τη συγκεκριμένη θέση που εισάγαμε και αν εισάγουμε και τις 2 παραμέτρους θα εμφανιστεί ο κώδικας που υπάρχει ανάμεσα σε αυτές τις διευθύνσεις μνήμης. Για παράδειγμα, με την εντολή disassemble (main+20) θα εμφανιστούν οι εντολές από τη θέση main έως τη θέση main+40 (αφού κάθε εντολή καταλαμβάνει 4 bytes). Με την εντολή disassemble (main+8) (main+16) θα εμφανιστούν 3 εντολές. Η χρησιμότητα της εντολής disassemble εμφανίζεται σε περιπτώσεις που υπάρχουν συναρτήσεις που δεν έχουν επιπλέον πληροφορία για λόγους αποσφαλμάτωσης, οπότε δεν μπορεί να εντοπιστεί πηγαίος κώδικας.



Επεξήγηση

Όταν εκτελείται η διαδικασία μεταγλώτισης και σύνδεσης του αρχείου, για να παραχθεί ο εκτελέσιμος κώδικας, τα εργαλεία δέχονται μια σειρά από παραμέτρους που καθορίζουν το πως θα ολοκληρωθεί αυτή η διαδικασία και τι θα συμπεριληφθεί στο τελικό αρχείο. Μια από τις επιλογές είναι και η προσθήκη επιπλέον πληροφορίας για λόγους αποσφαλμάτωσης. Την πληροφορία αυτή χρησιμοποιεί ο gdb για να προβάλει κείμενο του πηγαίου αρχείου κώδικα, ονόματα συναρτήσεων και επιπλέον πληροφορίες για το εκτελέσιμο. Κατά την διαδικασία της σύνδεσης όμως, μπορεί να χρησιμοποιηθούν συναρτήσεις της βιβλιοθήκης του συστήματος, οι οποίες βρίσκονται σε αρχεία βιβλιοθήκης χωρίς επιπλέον πληροφορίες για λόγους μεγέθους, αφού η εισαγωγή πλεονάζουσας πληροφορίας που δεν σχετίζεται με την εκτέλεση αλλά με την αποσφαλμάτωση προσθέτει άσκοπο χώρο στο αρχείο. Γι'αυτό το λόγο, αν χρειαστεί να συμπεριλάβουμε κώδικα χωρίς πληροφορία αποσφαλμάτωσης, ο μόνος τρόπος προβολής του είναι με τη χρήση του disassemble.

4.4 Ευκολίες

Δεδομένης της μικρής οθόνης του AT91, πολλές φορές χρειάζεται μετακίνηση ως προς τον οριζόντιο άξονα, είτε σύμπτυξη ή μεγέθυνση της γραμματοσειράς ώστε να έχουμε περισσότερη πληροφορία διαθέσιμη. Στο AT91, έχουν ενσωματωθεί δύο συνδυασμοί πλήκτρων για την εκτέλεση αυτών των λειτουργιών :

- Ο συνδυασμός πλήκτρων $Alt + Control + \leftarrow$ και $Alt + Control + \rightarrow$ επιτρέπει τη μετακίνηση (scrolling) της οθόνης προς τα αριστερά και τα δεξιά αντίστοιχα.
- Ο συνδυασμός πλήκτρων $Alt + \uparrow$ και $Alt + \downarrow$ επιτρέπει τη σύμπτυξη και την αραίωση (μέσω της αλλαγής γραμματοσειράς) αντίστοιχα των γραμμών κειμένου που είναι ορατές σε μία πλήρη οθόνη του AT91.

Εξομοιωτής

5.1 Γενικά

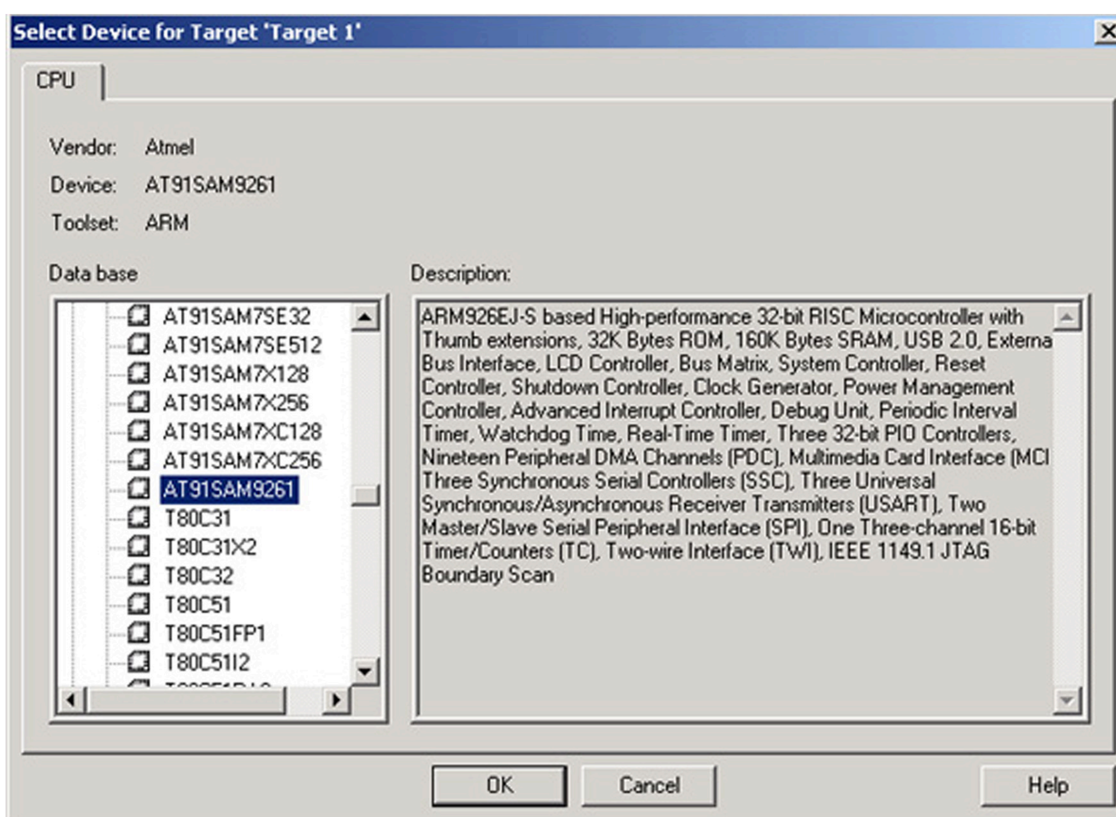
Ο εξομοιωτής του AT91 (προϊόν της εταιρίας KEIL) είναι ένα πλήρες εργαλείο ανάπτυξης και ελέγχου εφαρμογών για διάφορες αρχιτεκτονικές της εταιρείας ARM και τις υλοποιήσεις τους από πληθώρα εταιρειών. Ο πλήρης εξομοιωτής αποτελεί καταχωρημένο εμπορικό προϊόν υψηλού κόστους. Εσάς σας παρέχεται η δωρεάν έκδοσή του, η οποία είναι υπεραρκετή για εκπαιδευτικούς σκοπούς. Ο εξομοιωτής λειτουργεί σε περιβάλλον Microsoft Windows XP / Vista και διατίθεται και για εγκατάσταση στο δικό σας προσωπικό υπολογιστή, ακολουθώντας μια διαδικασία εγκατάστασης που περιγράφεται πιο κάτω. Ο εξομοιωτής επιτρέπει τη δυνατότητα ανάπτυξης και συγγραφής πηγαίου κώδικα, παραγωγής εκτελέσιμου κώδικα και αποσφαλμάτωσής του μέσω βηματικής εκτέλεσης σε συνθήκες εξομοίωσης που περιγράφουν ακριβώς την αρχιτεκτονική του επεξεργαστή μας. Με τον εξομοιωτή είναι δυνατή η ελεγχόμενη ανάπτυξη εφαρμογών χωρίς την ύπαρξη του AT91, ώστε να είναι δυνατή η συγγραφή των εργασιών και εκτός του εργαστηρίου. Πριν τη χρήση του πρέπει να γίνουν μια σειρά από ρυθμίσεις, ώστε να επιλεγεί ο αντίστοιχος με το AT91 τύπος επεξεργαστή και τα κατάλληλα εργαλεία μετάφρασης.

5.2 Ρυθμίσεις

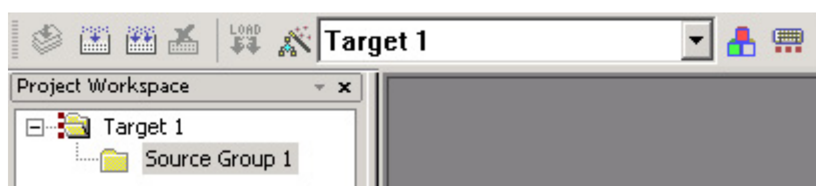
Πριν την εγκατάσταση του εξομοιωτή, είναι απαραίτητη η εγκατάσταση των GNU εργαλείων (τα οποία έχουν ως προεπιλεγμένο κατάλογο εγκατάστασης τον C:\Cygnus και προτείνεται να μην αλλαχθεί αν δεν συντρέχουν άλλοι λόγοι). Η εγκατάσταση αυτών γίνεται με την εκτέλεση του αρχείου gccarm331.exe. Ακολούθως θα πρέπει να εκτελέσετε το αρχείο mdk311.exe, το οποίο θα εγκαταστήσει όλο το περιβάλλον του εξομοιωτή, για διάφορους υποστηριζόμενους επεξεργαστές και μικροελεγκτές. Στη συνέχεια θα πρέπει να πληροφορήσουμε τον εξομοιωτή και τα εργαλεία μετάφρασης, για την αρχιτεκτονική την οποία θέλουμε να εξομοιώσουμε.

Έτσι, μετά την έναρξη του εξομοιωτή, δημιουργούμε ένα νέο Project επιλέγοντας από το menu Project την επιλογή New Project... και επιλέγουμε τον κατάλογο όπου θα απο-

θηκευτούν τα αρχεία μας. Ο εξομοιωτής θα προτρέψει στη συνέχεια να επιλέξουμε το στοχευόμενο επεξεργαστή / μικροελεγκτή για τον οποίο θα διαμορφωθεί ο κώδικάς μας. Επιλέγουμε συνεπώς την εταιρία Atmel και από τους επεξεργαστές που παρουσιάζονται τον AT91SAM9261(εικόνα *Επιλογή επεξεργαστή*, κεφ. 5.11, σελ. 72). Στην ερώτηση για το αν στο project που δημιουργήθηκε θα συμπεριληφθεί και το πρότυπο αρχείο αρχικοποίησης, επιλέγουμε όχι.



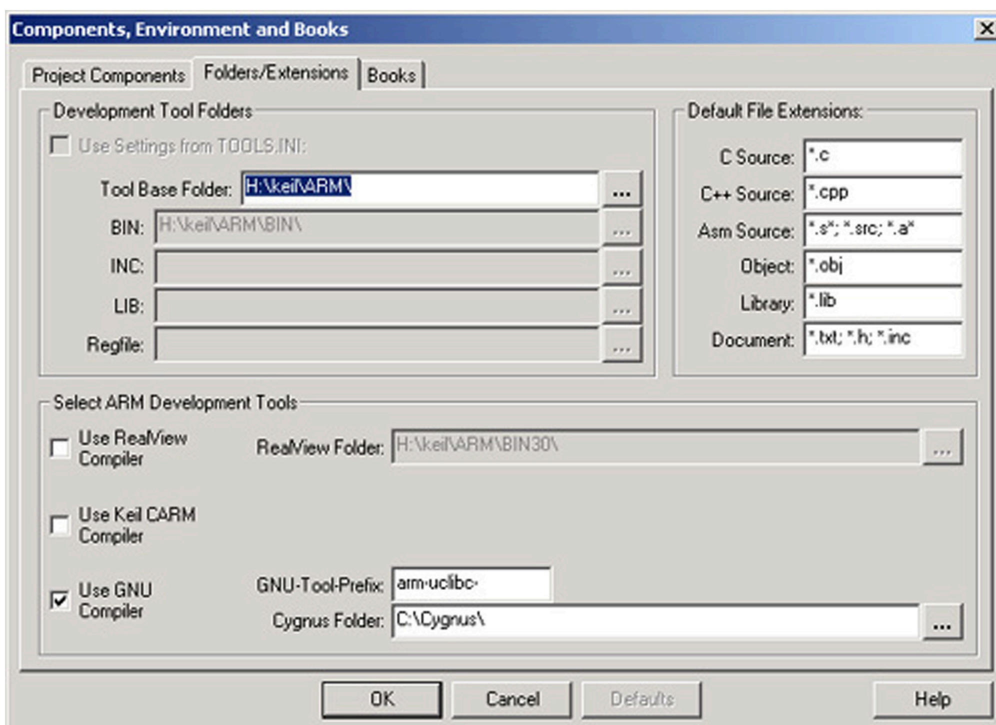
Σχήμα 5.11: Επιλογή επεξεργαστή



Σχήμα 5.12: Εισαγωγή Αρχείων

Για να δημιουργήσουμε ένα νέο αρχείο πηγαίου κώδικα, επιλέγουμε το New... από

το menu File, το οποίο θα ανοίξει ένα παράθυρο πληκτρολόγησης κειμένου. Αποθηκεύουμε το πηγαίο κώδικα στον κατάλογο που δημιουργήσαμε για το project, με την επιλογή Save As... του menu File και δίνουμε στο αρχείο μας ένα όνομα, χρησιμοποιώντας πάντα την κατάληξη .s για τα αρχεία συμβολικής γλώσσας ή .c για τα αρχεία γλώσσας C (για παράδειγμα, exer1.s, ask2.c κλπ). Μόλις ολοκληρωθεί αυτή η διαδικασία, επιλέγουμε στο Project workspace το Target 1 και με δεξί κλικ στο Source Group 1 επιλέγουμε την επιλογή Add Files to Group "Source Group 1" (εικόνα *Εισαγωγή Αρχείων*, κεφ. 5.12, σελ. 72). Επιλέγοντας το αρχείο πηγαίου κώδικα που δημιουργήθηκε αμέσως πριν στο παράθυρο που εμφανίζεται και πατώντας το Add, το αρχείο πηγαίου κώδικα που δημιουργήσαμε εισάγεται στο Project[24]. Μετά από αυτό επιλέγουμε το Close.



Σχήμα 5.13: Επιλογή Καταλόγων

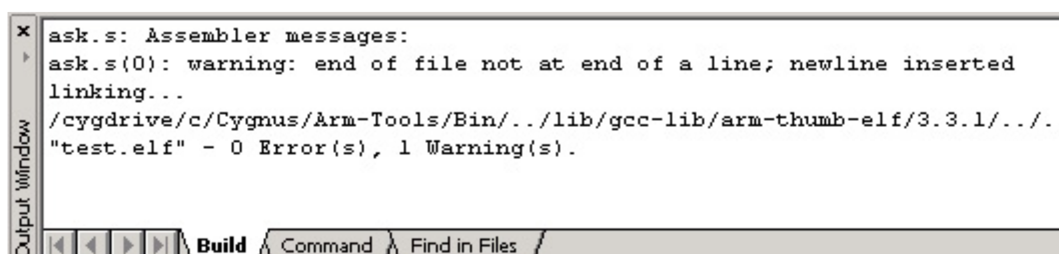
Σκοπός στη συνέχεια είναι να μεταφράσουμε το πηγαίο κώδικά μας σε εκτελέσιμο, μέσω των εργαλείων GNU. Για να ενημερωθεί ο εξομοιωτής ότι ως εργαλεία μετάφρασης θέλουμε τα GNU εργαλεία, επιλέγουμε από το Project menu το manage και εκεί το Components, Environment, Books. Στην καρτέλα που ανοίγει επιλέγουμε το Folders/Exten-

[24]Αν εισάγετε αρχείο Assembly επιλέξτε το files of type: Asm source file(*.s; *.src; *.a*), ώστε να εμφανιστεί το αρχείο.

sions tab (εικόνα *Επιλογή Καταλόγων*, κεφ. 5.13, σελ. 73). Από τις τρεις πιθανές επιλογές συνεργαζόμενων εργαλείων μετάφρασης θα πρέπει να επιλέξουμε τη τελευταία Use GNU compiler και να συμπληρώσουμε το πρώτο πεδίο (GNU-Tool-Prefix) με το κείμενο arm-uclibc- και το δεύτερο πεδίο (Cygnus Folder) με τον κατάλογο που εγκαταστάθηκαν τα GNU tools (αν χρησιμοποιήσαμε τον προεπιλεγμένο C:\Cygnus δε χρειάζεται να πειράξουμε τίποτε). Η τελευταία ρύθμιση αφορά στο μοντέλο του επεξεργαστή που θα χρησιμοποιηθεί από τον Assembler. Από το Project menu επιλέγουμε την επιλογή Options for Target. Στη καρτέλα που εμφανίζεται επιλέγουμε το Assembler tab και στο πεδίο Misc Controls προσθέτουμε το **-mcpu=arm926ej**. Αφού ολοκληρωθούν όλα τα παραπάνω βήματα, ο εξομοιωτής μας είναι έτοιμος να παράγει και να εκτελεί κώδικα, ακριβώς όπως και το AT91.

5.3 Χρήση

Η διαδικασία παραγωγής εκτελέσιμου αρχείου γίνεται με την επιλογή Build target στο Project menu. Αυτό θα ξεκινήσει τη μετάφραση του προγράμματος σας και θα εμφανίσει μηνύματα στην κονσόλα κειμένου που υπάρχει στο κατώτερο μέρος της εφαρμογής. Η τελευταία γραμμή που θα τυπωθεί αναφέρει και το πόσα σφάλματα συνάντησε κατά τη μετάφραση ο GNU assembler. Αν υπάρχει έστω και ένα error, δεν θα παραχθεί εκτελέσιμο αρχείο (εικόνα *Μηνύματα*, κεφ. 5.14, σελ. 74).



Σχήμα 5.14: Μηνύματα

Σε περίπτωση σφάλματος, ο GNU assembler θα μας ειδοποιήσει πως το σφάλμα έχει παρουσιαστεί σε συγκεκριμένη γραμμή και θα προσπαθήσει να μας εξηγήσει το είδος του σφάλματος (πχ. λανθασμένη εντολή, λανθασμένα σημεία στίξης). Από τη στιγμή που η διαδικασία ολοκληρωθεί επιτυχώς, μπορούμε να επιλέξουμε την έναρξη αποσφαλμάτωσης Start/Stop debug session από το Debug menu. Θα εμφανιστεί ένα νέο παράθυρο το οποίο περιέχει τα περιεχόμενα της μνήμης (δηλαδή το δυαδικό εκτελέσιμο κώδικα) και ξεκινά από τη διεύθυνση που υπάρχει στον PC. Ο PC (R15) μαζί με τους υπόλοιπους καταχωρητές εμφανίζονται στο αριστερό τμήμα της οθόνης και η τιμή τους μπορεί να αλλαχθεί ανά πάσα στιγμή με την επιλογή τους με το ποντίκι (η επιλογή θα εμφανίσει ένα διάστικτο ορθογώνιο

γύρω από τον καταχωρητή, το οποίο σημαίνει ότι έχει επιλεγεί) και το πάτημα του κουμπιού F2.

Το πρόγραμμά μας έχει τοποθετηθεί αυτόματα στη διεύθυνση 0x8000, οπότε για να αρχίσουμε να το εκτελούμε πρέπει να αλλάξουμε την τιμή του PC σε 0x8000. Ένα κίτρινο βελάκι δίπλα από την γραμμή κώδικα εμφανίζεται για να δείξει την επόμενη προς εκτέλεση εντολή (εικόνα *Εκτελέσιμος Κώδικας*, κεφ. 5.15, σελ. 75). Για να εκτελεστεί η εντολή

```
      6: STMDB   R13!, {R0-R12, R14}
      7:
0x00008000  E92D5FFF  STMDB     R13!,{R0-R12,R14}
      8: MOV           R0, #0
      9:
     10: main_loop:
0x00008004  E3A00000  MOV       R0,#0x00000000
     11: BL           Check
```

Σχήμα 5.15: Εκτελέσιμος Κώδικας

επιλέγουμε το Step από το Debug menu και παρατηρούμε πως ο PC αυξήθηκε κατά 4 και δείχνει στην επόμενη εντολή. Η επιλογή Step Over του Debug menu κάνει την ίδια δουλειά με την Step, αλλά όταν συναντά κλήση εντολών διακλάδωσης δεν τις ακολουθεί (θεωρεί ότι καλείται συνάρτηση, ο εξομοιωτής εκτελεί τον κώδικα της συνάρτησης και σταματά αμέσως μετά από την επιστροφή της συνάρτησης).

Σαν εξάσκηση στη χρήση του εξομοιωτή, ας δούμε το ακόλουθο παράδειγμα. Δημιουργήστε ένα νέο Project και γράψτε το παρακάτω πρόγραμμα. Δώστε του το όνομα Sample1.s και προσθέστε το στο Project.

Stack

```
1  .arm
2  .text
3  .global main
4
5  main:
6  STMDB R13!, {R0-R2, R14}
7
8  LDR R1, =Values
9  LDR R2, [R1,#0]
10 MOV R2, R2, LSL #0x02
11 STR R2, [R1, #4]
12
13 LDMIA R13!, {R0-R2, PC}
14
15 .data
16 Values:
17 .word 0x10, 0xFFFFFFFF
```

Επιλέξτε το Build target από το Project menu και αν όλες οι ρυθμίσεις είναι σωστές, θα σας εμφανίσει 0 σφάλματα και 0 ειδοποιήσεις, εκτός από ένα μήνυμα πως ο linker δεν μπορεί να βρει το σύμβολο start και θα τοποθετήσει τον κώδικα στη διεύθυνση 0x8000.

Αφού ξεκινήσετε τη διαδικασία αποσφαλμάτωσης και πριν εκτελέσετε τη 1η εντολή, βάλτε τις ακόλουθες τιμές στους καταχωρητές : R0=0xAA, R1=0xBB, R2=0xCC, R13 = 0x301000, R14 = 0x7777 και R15=0x8000. Εχοντας βάλει το μετρητή προγράμματος στη 1η εντολή του κώδικά μας, είμαστε έτοιμοι να την εκτελέσουμε. Ας δούμε όμως τι περιμένουμε να γίνει. Η εντολή αποθηκεύει τον R14 και τους τρεις καταχωρητές R2 - R0 στο σωρό. Προσέξτε ότι αυτό κάνει επιτακτική την αρχικοποίηση του R13 πριν την εκτέλεση του προγράμματός μας, όπως και σε κάθε άλλο πρόγραμμα που χρησιμοποιεί το σωρό. Σύμφωνα με τις αναθέσεις μας ο καταχωρητής σωρού δείχνει στη διεύθυνση 0x301000. Ας δούμε λοιπόν αν όντως θα εκτελεστεί σωστά αυτή η εντολή κοιτάζοντας τα περιεχόμενα των θέσεων μνήμης γύρω από αυτή τη διεύθυνση. Για να δείτε τα περιεχόμενα της μνήμης επιλέξτε το Memory Window από το menu View και στο παράθυρο που εμφανίζεται εισάγετε τη διεύθυνση 0x300FD0 στο πεδίο Address. Εκτελέστε τη 1η εντολή και παρατηρείστε τις αλλαγές στη μνήμη. Ακολούθως, το πρόγραμμά μας, φέρνει τη διεύθυνση Values στον καταχωρητή R1, την τιμή που υπάρχει σε αυτή τη διεύθυνση μνήμης στον καταχωρητή R2, πολλαπλασιάζει με 4 αυτή την τιμή και την αποθηκεύει στην επόμενη διεύθυνση από την αρχική. Όλα αυτά όμως συνδέονται με μια άλλη περιοχή της μνήμης, γύρω από τη διεύθυνση 0x8000. Χρησιμοποιώντας ένα δεύτερο παράθυρο μνήμης μεταβείτε συνεπώς στη περιοχή αυτή της μνήμης και εκτελέστε βηματικά το κώδικά σας μέχρι την γραμμή 9. Θα δείτε κατ' αυτό το τρόπο τη διεύθυνση της ετικέτας Values

που έχει αποθηκευτεί στον καταχωρητή R1 και η τιμή της είναι 0x8120. Για να δείτε τα περιεχόμενα της μνήμης εκεί επιλέξτε το 3ο Memory Window και στο παράθυρο που εμφανίζεται εισάγετε την διεύθυνση αυτή στο πεδίο Address (εικόνα *Μνήμη*, κεφ. 5.16, σελ. 77).

Σχήμα 5.16: Μνήμη

Θα παρατηρήσετε πως στην πάνω-αριστερή πλευρά του παραθύρου υπάρχουν τα 2 words του προγράμματός μας. Εκτελώντας βηματικά και τα επόμενα βήματα, μέχρι τη γραμμή 13, θα παρατηρήσετε πως τα περιεχόμενα της διεύθυνσης 0x8124 θα πάρουν την τιμή 0x00000040. Εκτός του να παρατηρείτε τα δεδομένα που αποθηκεύονται στη μνήμη, μπορείτε και να αλλάξετε την τιμή τους, επιλέγοντας το κελί μνήμης που επιθυμείτε και με δεξί κλικ επιλέγοντας Modify Memory. Επίσης, μπορείτε να αλλάξετε τον τρόπο παρουσίασης τους (σε μορφή byte, int κλπ) επιλέγοντας από το pop-up menu που εμφανίζεται με δεξί κλικ στο Memory window, μια από τις επιλογές unsigned, signed, decimal κλπ. Μπορείτε συνεπώς βάζοντας διαφορετική αρχική τιμή στη θέση μνήμης 0x8120 και επαναφέροντας τον PC στη 1η εντολή (βάζοντάς του τιμή 0x8000) να δοκιμάσετε το πως συμπεριφέρεται το πρόγραμμα και για άλλες τιμές. Η τελευταία εντολή τέλος αποκαθιστά τα αρχικά περιεχόμενα των καταχωρητών που αποθηκεύσαμε στο σωρό και επαναφέρει το δείκτη σωρού στην αρχική τιμή.

Instruction Set

Σε αυτό το κεφάλαιο παρουσιάζονται οι πιο συχνά χρησιμοποιούμενες εντολές του επεξεργαστή μας. Το υποσύνολο εντολών που παρουσιάζεται υπερκαλύπτει τις ανάγκες του εργαστηρίου προγραμματισμού σε συμβολική γλώσσα. Για κάθε εντολή δίνεται η γενική μορφή της, μια σύντομη περιγραφή για τη λειτουργία της, αναλύονται οι διαφορετικοί παράμετροι που μπορεί να δεχτεί και παρέχεται σύντομος ψευδοκώδικας που περιγράφει λεπτομερώς τις ενέργειες που εκτελεί.

6.1 Αριθμητικές Εντολές

6.1.1 ADC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 1 0 1**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή ADC (Add with Carry) προσθέτει την έξοδο του ολισθητή και το Carry bit στην τιμή του Rn. Αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

ADC{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες κατάστασης N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και οι σημαίες C & V ανάλογα με το αν έχει δημιουργηθεί κρατούμενο (υπερχείληση κατά μη-προσημασμένη πρόσθεση) ή προσημασμένη υπερχείλιση. Οι υπόλοιπες σημαίες κατάστασης δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της πρόσθεσης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της πρόσθεσης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn + shifter_operand + C_flag
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Κρατούμενο πρόσθεσης)
        V_flag = (Υπερχείλιση πρόσθεσης)

Χρήση

Η εντολή αυτή συνήθως χρησιμοποιείται για την υλοποίηση προσθέσεων ανάμεσα σε όρους που έχουν εύρος μεγαλύτερο από 32 bits. Για παράδειγμα, οι εντολές:

ADDS R4, R0, R2 @R0 + R2 − > R4

ADC R5, R1, R3 @R1 + R3 + Carry − > R5

εκτελούν την πρόσθεση των 64bit αριθμών R0 + R1 * 2^{32} με R2 + R3 * 2^{32}. Αρχικά προσθέτουμε τα 32 λιγότερο σημαντικά bits, αποθηκεύουμε το κρατούμενο και το διαδίδουμε στην επόμενη πρόσθεση.

6.1.2 ADD

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 1 0 0**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή ADD προσθέτει το αποτέλεσμα του ολισθητή στην τιμή του Rn και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

ADD{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης βάσει του αποτελέσματος της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες κατάστασης N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και οι σημαίες C & V ανάλογα με το αν έχει δημιουργηθεί κρατούμενο (υπερχείληση κατά μη-προσημασμένη πρόσθεση) ή προσημασμένη υπερχείλιση. Οι υπόλοιπες σημαίες κατάστασης δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της πρόσθεσης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της πρόσθεσης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn + shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Κρατούμενο πρόσθεσης)
        V_flag = (Υπερχείλιση πρόσθεσης)

Χρήση

Η εντολή αυτή χρησιμοποιείται για την υλοποίηση προσθέσεων ανάμεσα σε όρους με εύρος μικρότερο ή ίσο των 32 bits.

6.1.3 AND

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 0 0 0**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή AND εκτελεί τη λογική πράξη **ΚΑΙ** ανάμεσα στα ψηφία του Rn και του αποτελέσματος του ολισθητή και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

AND{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση του CPSR με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες κατάστασης N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και η σημαία C ανάλογα με το αν έχει δημιουργηθεί κρατούμενο από ολίσθηση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της λογικής πράξης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της λογικής πράξης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
  Rd = Rn AND shifter_operand
  Αν S == 1 και Rd == 15 τότε
    CPSR = SPSR
  Διαφορετικά αν S == 1 τότε
    N_flag = Rd[31]
    Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
    C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
    V_flag = (Ανεπηρέαστο)

Χρήση

Με την εντολή **AND** μπορούμε πολύ εύκολα να απομονώσουμε συγκεκριμένα ψηφία ενός καταχωρητή. Για παράδειγμα, με τις εντολές:

MOV R4, #0x82 @#0x82 − > R4
AND R5, R5, R4 @R5 AND R4 − > R5

απομονώνουμε τα bits 7 και 1 (η αρίθμηση των bits ξεκινά από το 0, το οποίο αντιστοιχεί το λιγότερο σημαντικό) και ελέγχουμε αν έχουν την τιμή 1[25].

[25]Το λογικό AND δύο ψηφίων με το ένα να είναι 1, παρέχει αποτέλεσμα ίσο με την τιμή του άλλου ψηφίου.

6.1.4 BIC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**1 1 1 0**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή BIC (Bit Clear) εκτελεί λογικό AND των ψηφίων του Rn και του συμπληρώματος της εξόδου του ολισθητή και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

BIC{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και η σημαία C ανάλογα με το αν έχει δημιουργηθεί κρατούμενο από ολίσθηση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της λογικής πράξης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της λογικής πράξης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn AND NOT shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
        V_flag = (Ανεπηρέαστο)

Χρήση

Η εντολή αυτή χρησιμοποιείται όταν θέλουμε να κάνουμε 0 συγκεκριμένα ψηφία ενός καταχωρητή. Για παράδειγμα, η εντολή:

BIC R5, R5, #0x44 @R5 AND NOT #0x44 $->$ R5

καθαρίζει τα bits 6 και 2 από τον καταχωρητή R5.

6.1.5 CLZ

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	**0 0 0 1 0 1 1 0**	**0 0 0 0**	**Rd**	**0 0 0 0**	**0 0 0 1**	**Rm**

Η εντολή CLZ (Count Leading Zeros) επιστρέφει τον αριθμό των συνεχόμενων 0, που υπάρχουν στα πιο σημαντικά ψηφία του Rm. Το αποτέλεσμα το αποθηκεύει στον καταχωρητή Rd.

Σύνταξη

CLZ{<cond>} $< Rd >, < Rm >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.
$< Rm >$	Ορίζει τον καταχωρητή, στον οποίο θα γίνει η μέτρηση των 0.

Λειτουργία

Αν Ισχύει{cond} τότε
  Αν Rm == 0 τότε
    Rd = 32
  Διαφορετικά
    Rd = 31 - #(bit position of most significant 1)

Χρήση

Η εντολή αυτή συνήθως χρησιμοποιείται όταν θέλουμε να κανονικοποιήσουμε ένα καταχωρητή, έτσι ώστε το περισσότερο σημαντικό ‘1’ του να εμφανίζεται στην θέση 31, την πιο σημαντική θέση. Τη χρησιμοποιούμε δηλαδή για να καθορίσουμε το πόσες θέσεις αριστερής ολίσθησης χρειαζόμαστε. Για παράδειγμα:

CLZ R5, R4 @ leading_zeros(R4) $->$ R5
MOV R4, R4, LSL R5 @ R4 * $2^{R5} ->$ R4

αν ο R4 περιέχει την τιμή 0x49, ο R5 θα αποθηκεύσει την τιμή 25 (δυαδική αναπαράσταση του 0x49 είναι 00000000_00000000_00000000_01001001) και μετά την ολίσθηση θα γίνει 0x92000000.

6.1.6 CMN

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 1	1	Rn	0 0 0 0	Shifter operand

Η εντολή CMN (Compare Negative) εκτελεί σύγκριση ανάμεσα στον Rn και στο αντίθετο (σε αριθμητική συμπληρώματος ως προς 2) αποτέλεσμα του ολισθητή. Οι σημαίες κατάστασης ενημερώνονται πάντα, ανάλογα με το αποτέλεσμα της σύγκρισης (η σύγκριση υλοποιείται με πρόσθεση).

Σύνταξη

CMN{<cond>} $< Rn >, < shifter_operand >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της σύγκρισης.
$< shifter_operand >$	Ορίζει τον δεύτερο όρο της σύγκρισης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει {cond} τότε

alu_out = Rn + shifter_operand

N_flag = alu_out[31]

Z_flag = (αν alu_out == 0 τότε 1 διαφορετικά 0)

C_flag = (Κρατούμενο πρόσθεσης)

V_flag = (Υπερχείληση πρόσθεσης)

Χρήση

Η εντολή αυτή χρησιμοποιείται όταν θέλουμε να ελέγξουμε το είδος του αποτελέσματος μιας πρόσθεσης (αν δηλαδή προκαλεί υπερχείλιση, κρατούμενο κλπ), ενώ δεν είναι απαραίτητη η αποθήκευση του αποτελέσματος.

6.1.7 CMP

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**1 0 1 0**	**1**	**Rn**	**0 0 0 0**	**Shifter operand**

Η εντολή CMP (Compare) εκτελεί σύγκριση ανάμεσα στον Rn και το αποτέλεσμα του ολισθητή. Οι σημαίες κατάστασης ενημερώνονται πάντα, ανάλογα με το αποτέλεσμα της σύγκρισης (η σύγκριση υλοποιείται με αφαίρεση).

Σύνταξη

CMP{<cond>} $< Rn >, < shifter_operand >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της σύγκρισης.
$< shifter_operand >$	Ορίζει τον δεύτερο όρο της σύγκρισης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει {cond} τότε
    alu_out = Rn - shifter_operand
        N_flag = alu_out[31]
        Z_flag = (αν alu_out == 0 τότε 1 διαφορετικά 0)
        C_flag = (NOT Κρατούμενο αφαίρεσης)
        V_flag = (Υπερχείληση αφαίρεσης)

6.1.8 EOR

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 0 0 1**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή EOR εκτελεί τη λογική πράξη **ΑΠΟΚΛΕΙΣΤΙΚΟ-Η** του Rn με το αποτέλεσμα του ολισθητή και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ενημερώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

EOR{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και η σημαία C ανάλογα με το αν έχει δημιουργηθεί κρατούμενο από ολίσθηση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της λογικής πράξης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της λογικής πράξης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn OR shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
        V_flag = (Ανεπηρέαστο)

Χρήση

Με την εντολή **EOR** αντιστρέφουμε συγκεκριμένα ψηφία ενός καταχωρητή. Για παράδειγμα οι :

MOV R4, #0xC0 @#0xC0 − > R4
EOR R5, R5, R4 @R5 XOR R4 − > R5

αντιστρέφουν τα ψηφία 7 και 6 (η αρίθμηση των ψηφίων ξεκινά από το 0, το οποίο αντιστοιχεί το λιγότερο σημαντικό) του καταχωρητή R5[26].

[26]Αν εκτελεστεί η πράξη EOR ανάμεσα σε 2 ψηφία με το ένα είναι 0, το αποτέλεσμα θα είναι ίσο με την τιμή του άλλου bit. Αν το ένα ψηφίο είναι 1, το αποτέλεσμα θα είναι το αντίθετο του άλλου.

6.1.9 MLA

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 0 0 1	S	Rd	Rn	Rs	1 0 0 1	Rm

Η εντολή MLA (MuLtiply and Accumulate) πολλαπλασιάζει προσημασμένα ή μη προσημασμένα έντελα για να παράγει αποτέλεσμα εύρους 32 ψηφίων, το οποίο προστίθεται σε τρίτο όρισμα, και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

MLA{<cond>}$\{S\} < Rd >, < Rm >, < Rs >, < Rn >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Επηρρεάζονται μόνο οι σημαίες N & Z.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rm >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο του πολλαπλασιασμού.

$< Rs >$ Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο του πολλαπλασιασμού.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει την τιμή που θα προστεθεί στο γινόμενο του πολλαπλασιασμού.

Λειτουργία

Αν Ισχύει{cond} τότε
- Rd = (Rm * Rs) + Rn
- Αν S == 1 τότε
  - N_flag = Rd[31]
  - Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
  - C_flag = (Ανεπηρέαστο)
  - V_flag = (Ανεπηρέαστο)

Σημειώσεις

R15 Η χρήση του R15 στη θέση των $< Rd >, < Rm >, < Rs >, < Rn >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.

Rd & Rm Η χρήση του ίδιου καταχωρητή για τον $< Rd >$ & $< Rm >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.

Πρόσημο Επειδή η εντολή MLA παράγει μόνο τα 32 λιγότερα σημαντικά ψηφία του 64ψήφιου γινομένου, ο πολλαπλασιασμός θα δώσει τα ίδια αποτελέσματα για προσημασμένους και μη προσημασμένους αριθμούς.

6.1.10 MUL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 0 0 0	S	Rd	0 0 0 0	Rs	1 0 0 1	Rm

Η εντολή MUL (MULtiply) πολλαπλασιάζει προσημασμένα ή μη προσημασμένα έντελα για να παράγει αποτέλεσμα εύρους 32 ψηφίων και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

MUL{<cond>}$\{S\} < Rd >, < Rm >, < Rs >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**S**}	Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Επηρρεάζονται μόνο οι σημαίες N & Z.
$< Rd >$	Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο του πολλαπλασιασμού.
$< Rs >$	Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο του πολλαπλασιασμού.

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = (Rm * Rs)
    Αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Ανεπηρέαστο)
        V_flag = (Ανεπηρέαστο)

Σημειώσεις

R15	Η χρήση του R15 στη θέση των $< Rd >, < Rm >, < Rs >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.
Rd & Rm	Η χρήση του ίδιου καταχωρητή για τον $< Rd >$ & $< Rm >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.
Πρόσημο	Επειδή η εντολή MUL παράγει μόνο τα 32 λιγότερα σημαντικά ψηφία του 64ψήφιου γινομένου, ο πολλαπλασιασμός θα δώσει τα ίδια αποτελέσματα για προσημασμένους και μη προσημασμένους αριθμούς.

6.1.11 ORR

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 1 0 0	S	Rn	Rd	Shifter operand

Η εντολή ORR εκτελεί το λογικό **Η** ανάμεσα στον Rn και στο αποτέλεσμα του ολισθητή και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

ORR{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες bits N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και η σημαία C ανάλογα με το κρατούμενο ολίσθησης.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της λογικής πράξης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της λογικής πράξης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn OR shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
        V_flag = (Ανεπηρέαστο)

Χρήση

Με την εντολή **ORR** μπορούμε πολύ εύκολα να θέσουμε συγκεκριμένα ψηφία σε ένα καταχωρητή. Για παράδειγμα, οι :

MOV R4, #0xC0 @#0xC0 − > R4
ORR R5, R5, R4 @R5 ORR R4 − > R5

θέτουν (κάνουν 1) τα ψηφία 7 και 6 (η αρίθμηση των bits ξεκινά από το 0, το οποίο αντιστοιχεί το λιγότερο σημαντικό) του καταχωρητή R5.

6.1.12 RSB

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 0 1 1**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή RSB (Reverse SuBtract) αφαιρεί τον Rn από το αποτέλεσμα του ολισθητή και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

RSB{<cond>}{S} $< Rd >, < Rn >, < shifter_operand >$

{<**cond**>} — Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} — Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και οπι σημαίες C & V ανάλογα με το αν έχει δημιουργηθεί κρατούμενο (υπερχείληση κατά μη-προσημασμένη αφαίρεση) ή προσημασμένη υπερχείλιση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ — Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ — Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο της αφαίρεσης.

$< shifter_operand >$ — Ορίζει τον πρώτο όρο της αφαίρεσης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = shifter_operand - Rn
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (NOT Κρατούμενο αφαίρεσης)
        V_flag = (Υπερχείλιση αφαίρεσης)

Χρήση

Πέρα από τη προφανή της χρήση, η εντολή αυτή μπορεί να χρησιμοποιηθεί και για την εύρεση του συμπληρώματος ως προς 2 :

RSB R4, R4, #0 @ -R4 $->$ R4

και για να υπολογίστεί το γινόμενο $(2^n - 1) * Rn$:

RSB R4, R4, R4, LSL #8 @ $(2^8 * R4) - R4->$ R4

6.1.13 RSC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 1 1 1**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή RSC (Reverse Subtract with Carry) αφαιρεί το (Rn + NOT CPSR C bit) από το αποτέλεσμα του ολισθητή και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

RSC{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και οι σημαίες C & V ανάλογα με το αν έχει δημιουργηθεί κρατούμενο (υπερχείληση κατά μη-προσημασμένη αφαίρεση) ή προσημασμένη υπερχείλιση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο της αφαίρεσης.

$< shifter_operand >$ Ορίζει τον πρώτο όρο της αφαίρεσης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
Rd = shifter_operand - Rn - NOT(CPSR C bit
Αν S == 1 και Rd == 15 τότε
CPSR = SPSR
Διαφορετικά αν S == 1 τότε
N_flag = Rd[31]
Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
C_flag = (NOT Κρατούμενο αφαίρεσης)
V_flag = (Υπερχείλιση αφαίρεσης)

Χρήση

Η εντολή αυτή μπορεί να χρησιμοποιηθεί για την εύρεση του συμπληρώματος ως προς 2 ενός 64ψηφιου αριθμού, αποθηκευμένου για παράδειγμα στους καταχωρητές R0 & R1, με το λιγότερο σημαντικό word στον R0, με την ακολουθία εντολών :

RSBS R2, R0, #0 @ -R0 − > R2
RSC R3, R1, #0 @ -R1 − > R3

Σημειώσεις

Αν ενεργοποιηθεί η ανανέωση των σημαιών (έχει τεθεί το S), η σημαία C γίνεται:

1 Αν δεν υπάρχει κρατούμενο

0 Αν υπάρχει κρατούμενο

Με άλλα λόγια, η σημαία C χρησιμοποιείται σαν NOT(carry). Η αντιστροφή αυτή γίνεται για τις επόμενες πράξεις. Για παράδειγμα, οι εντολές **SBC & RSC** χρησιμοποιούν τη σημαία C σαν NOT(carry), και εκτελούν απλή αφαίρεση αν είναι 1 ή αφαιρούν ένα επιπλέον αν είναι 0 (δηλαδή #(αποτέλεσμα της αφαίρεσης) -1). Οι συνθήκες εκτέλεσης HS & LO είναι ισοδύναμες με τις CS & CC αντίστοιχα.

6.1.14 SBC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 1 1 0**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή SBC (SuBtract with Carry) αφαιρεί την τιμή του αποτελέσματος του ολισθητή από το [Rn + NOT(σημαία C] και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

SBC{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και οι σημαίες C & V ανάλογα με το αν έχει δημιουργηθεί κρατούμενο (υπερχείληση κατά μη-προσημασμένη αφαίρεση) ή προσημασμένη υπερχείλιση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.

$< Rn >$ Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της αφαίρεσης.

$< shifter_operand >$ Ορίζει τον δεύτερο όρο της αφαίρεσης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn - shifter_operand - NOT(CPSR C bit
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (NOT Κρατούμενο αφαίρεσης)
        V_flag = (Υπερχείλιση αφαίρεσης)

Χρήση

Η εντολή αυτή μπορεί να χρησιμοποιηθεί για την αφαίρεση τιμών μεγαλύτερου εύρους από 32 ψηφία. Για παράδειγμα, αφαίρεση της τιμής που βρίσκεται στους καταχωρητές R0 & R1 από την τιμή που βρίσκεται στους καταχωρητές R2 & R3, επιτυγχάνεται με :

SUBS R2, R2, R0 @ R2 - R0 −> R2
SBC R3, R3, R1 @ R3 - R1 −> R3

Σημειώσεις

Αν ενεργοποιηθεί η ανανέωση των σημαιών (έχει τεθεί το S), η σημαία C γίνεται:

1 Αν δεν υπάρχει κρατούμενο
0 Αν υπάρχει κρατούμενο

Με άλλα λόγια, η σημαία C χρησιμοποιείται σαν NOT(carry). Η αντιστροφή αυτή γίνεται για τις επόμενες πράξεις. Για παράδειγμα, οι εντολές **SBC & RSC** χρησιμοποιούν τη σημαία C σαν NOT(carry), και εκτελούν απλή αφαίρεση αν είναι 1 ή αφαιρούν ένα επιπλέον αν είναι 0 (δηλαδή #(αποτέλεσμα της αφαίρεσης) -1). Οι συνθήκες εκτέλεσης HS & LO είναι ισοδύναμες με τις CS & CC αντίστοιχα.

6.1.15 SMLAL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	**0 0 0 0 1 1 1**	**S**	**RdHi**	**RdLo**	**Rs**	**1 0 0 1**	**Rm**

Η εντολή SMLAL (Signed Multiply and Accumulate Long operands) πολλαπλασιάζει προσημασμένα έντελα για να παράγει αποτέλεσμα εύρους 64 ψηφίων, το οποίο προστίθεται στον 64ψηφίο αριθμό που έχει τα 32 πιο σημαντικά ψηφία του στον RdHi και τα 32 λιγότερο σημαντικά ψηφία του στον καταχωρητή RdLo. Το πιο σημαντικό μέρος του αποτελέσματος αποθηκεύεται στον RdHi και το λιγότερο σημαντικό στον καταχωρητή RdLo.

Σύνταξη

SMLAL{<cond>}$\{S\} < RdLo >, < RdHi >, < Rm >, < Rs >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**S**}	Όταν υπάρχει ενεργοποιεί ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Επηρρεάζονται μόνο οι σημαίες N & Z ανάλογα με το αποτέλεσμα της πράξης.
$< RdLo >$	Ορίζει τον καταχωρητή που περιέχει το λιγότερο σημαντικό word που θα αθροιστεί με το γινόμενο και θα αποθηκεύσει το λιγότερο σημαντικό word του αποτελέσματος.
$< RdHi >$	Ορίζει τον καταχωρητή που περιέχει το περισσότερο σημαντικό word που θα αθροιστεί με το γινόμενο και θα αποθηκεύσει το περισσότερο σημαντικό word του αποτελέσματος.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο του πολλαπλασιασμού.
$< Rs >$	Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο του πολλαπλασιασμού.

Λειτουργία

Αν Ισχύει{cond} τότε
    RdLo = (Rm * Rs)[31:0] + RdLo
    RdHi = (Rm * Rs)[63:32] + RdHi
    Αν S == 1 τότε
        N_flag = RdHi[31]
        Z_flag = (αν RdHi == 0 και RdLo == 0 τότε 1 διαφορετικά 0)
        C_flag = (Ανεπηρέαστο)
        V_flag = (Ανεπηρέαστο)

Σημειώσεις

R15	Η χρήση του R15 στη θέση των $< RdHi >, < RdLo >, < Rm >, < Rs >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.
RdHi, RdLo & Rm	Οι καταχωρητές αυτοί πρέπει να είναι διαφορετικοί μεταξύ τους, διαφορετικά το αποτέλεσμα δεν είναι προβλέψιμο.

6.1.16 SMULL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 1 1 0	S	RdHi	RdLo	Rs	1 0 0 1	Rm

Η εντολή SMULL (Signed MULtiply Long operands) πολλαπλασιάζει προσημασμένα έντελα για να παράγει αποτέλεσμα εύρους 64 ψηφίων. Το πιο σημαντικό μέρος του γινομένου αποθηκεύεται στον RdHi και το λιγότερο σημαντικό στον καταχωρητή RdLo.

Σύνταξη

SMULL{<cond>}$\{S\} < RdLo >, < RdHi >, < Rm >, < Rs >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**S**}	Όταν υπάρχει ενεργοποιεί ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Επηρρεάζονται μόνο οι σημαίες N & Z ανάλογα με το αποτέλεσμα της πράξης.
$< RdLo >$	Ορίζει τον καταχωρητή που θα αποθηκεύσει το λιγότερο σημαντικό word του γινομένου.
$< RdHi >$	Ορίζει τον καταχωρητή που περιέχει το περισσότερο σημαντικό word του γινομένου.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο του πολλαπλασιασμού.
$< Rs >$	Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο του πολλαπλασιασμού.

Λειτουργία

```
Αν Ισχύει{cond} τότε
    RdLo = (Rm * Rs)[31:0]
    RdHi = (Rm * Rs)[63:32]
    Αν S == 1 τότε
        N_flag = RdHi[31]
        Z_flag = (αν RdHi == 0 και RdLo == 0 τότε 1 διαφορετικά 0)
        C_flag = (Ανεπηρέαστο)
        V_flag = (Ανεπηρέαστο)
```

Σημειώσεις

R15	Η χρήση του R15 στη θέση των $< RdHi >, < RdLo >, < Rm >, < Rs >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.
RdHi, RdLo & Rm	Οι καταχωρητές αυτοί πρέπει να είναι διαφορετικοί μεταξύ τους, διαφορετικά το αποτέλεσμα δεν είναι προβλέψιμο.

6.1.17 SUB

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**0 0 1 0**	**S**	**Rn**	**Rd**	**Shifter operand**

Η εντολή SUB (SUBtract) αφαιρεί την έξοδο του ολισθητή από τον Rn και αποθηκεύει το αποτέλεσμα στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

SUB{<cond>}$\{S\} < Rd >, < Rn >, < shifter_operand >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**S**}	Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν: • Αν ο Rd δεν είναι ο R15, οι σημαίες κατάστασης N & Z τίθενται ανάλογα με το αποτέλεσμα της πράξης και οι σημαίες C & V ανάλογα με το αν έχει δημιουργηθεί κρατούμενο (υπερχείληση κατά μη-προσημασμένη αφαίρεση) ή προσημασμένη υπερχείλιση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται. • Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.
$< Rd >$	Ορίζει τον καταχωρητή που θα αποθηκευτεί το αποτέλεσμα.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της αφαίρεσης.
$< shifter_operand >$	Ορίζει τον δεύτερο όρο της αφαίρεσης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = Rn - shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (NOT Κρατούμενο αφαίρεσης)
        V_flag = (Υπερχείλιση αφαίρεσης)

Χρήση

Η εντολή αυτή μπορεί να χρησιμοποιηθεί και για μείωση τιμής με ταυτόχρονο έλεγχο, αφού η λειτουργία της είναι ίδια με την εντολή **CMP**, με τη διαφορά πως η **SUBS** αποθηκεύει το αποτέλεσμα και σε κάποιον καταχωρητή.

Σημειώσεις

Αν ενεργοποιηθεί η ανανέωση των σημαιών (έχει τεθεί το S), η σημαία C γίνεται:

1 Αν δεν υπάρχει κρατούμενο

0 Αν υπάρχει κρατούμενο

Με άλλα λόγια, η σημαία C χρησιμοποιείται σαν NOT(carry). Η αντιστροφή αυτή γίνεται για τις επόμενες πράξεις. Για παράδειγμα, οι εντολές **SBC & RSC** χρησιμοποιούν τη σημαία C σαν NOT(carry), και εκτελούν απλή αφαίρεση αν είναι 1 ή αφαιρούν ένα επιπλέον αν είναι 0 (δηλαδή #(αποτέλεσμα της αφαίρεσης) -1). Οι συνθήκες εκτέλεσης HS & LO είναι ισοδύναμες με τις CS & CC αντίστοιχα.

6.1.18 TEQ

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**1 0 0 1**	**1**	**Rn**	**0 0 0 0**	**Shifter operand**

Η εντολή TEQ (Test EQuivalence) εκτελεί σύγκριση ανάμεσα στην τιμή του καταχωρητή Rn και το αποτέλεσμα του ολισθητή. Οι σημαίες κατάστασης ενημερώνονται πάντα, ανάλογα με το αποτέλεσμα της σύγκρισης. Η σύγκριση διενεργείται με λογικό EOR.

Σύνταξη

TEQ{<cond>} $< Rn >, < shifter_operand >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της σύγκρισης.
$< shifter_operand >$	Ορίζει τον δεύτερο όρο της σύγκρισης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει {cond} τότε
    alu_out = Rn EOR shifter_operand
        N_flag = alu_out[31]
        Z_flag = (αν alu_out == 0 τότε 1 διαφορετικά 0)
        C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
        V_flag = (Ανεπηρέαστο)

Χρήση

Η εντολή αυτή χρησιμοποιείται για να γίνει έλεγχος για ισότητα, χωρίς όμως να αλλάξει η τιμή της σημαίας V, κάτι που δεν αποφεύγεται με την **CMP**.

6.1.19 TST

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**1 0 0 0**	**1**	**Rn**	**0 0 0 0**	**Shifter operand**

Η εντολή TST (TeST) εκτελεί σύγκριση ανάμεσα στο καταχωρητή Rn και το αποτέλεσμα του ολισθητή. Οι σημαίες κατάστασης ενημερώνονται πάντα, ανάλογα με το αποτέλεσμα της σύγκρισης. Η σύγκριση διενεργείται με λογικό AND.

Σύνταξη

TST{<cond>} $< Rn >, < shifter_operand >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο της σύγκρισης.
$< shifter_operand >$	Ορίζει τον δεύτερο όρο της σύγκρισης (περιγραφή στο κεφ. 2.3.2, σελ. 2.3.2).

Λειτουργία

Αν Ισχύει {cond} τότε

  alu_out = Rn AND shifter_operand

    N_flag = alu_out[31]

    Z_flag = (αν alu_out == 0 τότε 1 διαφορετικά 0)

    C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)

    V_flag = (Ανεπηρέαστο)

Χρήση

Η εντολή αυτή χρησιμοποιείται για να ελέγξουμε αν μια ομάδα ψηφίων του Rn είναι ενεργοποιημένα ή μη.

6.1.20 UMLAL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	**0 0 0 0 1 0 1**	**S**	**RdHi**	**RdLo**	**Rs**	**1 0 0 1**	**Rm**

Η εντολή UMLAL (Unsigned MuLtiply and Accumulate Long operands) πολλαπλασιάζει μη-προσημασμένα έντελα για να παράγει αποτέλεσμα εύρους 64 ψηφίων, του οποίου το πιο σημαντικό μέρος προστίθεται με τον RdHi και το λιγότερο σημαντικό με τον καταχωρητή RdLo. Το πιο σημαντικό μέρος του αθροίσματος αποθηκεύεται στον RdHi και το λιγότερο σημαντικό στον καταχωρητή RdLo.

Σύνταξη

UMLAL{<cond>}$\{S\} < RdLo >, < RdHi >, < Rm >, < Rs >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**S**}	Όταν υπάρχει ενεργοποιεί την ανανέωση των σημαιών κατάστασηςμε βάση το αποτέλεσμα της εντολής. Επηρρεάζονται μόνο οι σημαίες N & Z ανάλογα με το αποτέλεσμα της πράξης.
$< RdLo >$	Ορίζει τον καταχωρητή που περιέχει το λιγότερο σημαντικό word που θα αθροιστεί με το γινόμενο και θα αποθηκεύσει το λιγότερο σημαντικό word του αποτελέσματος.
$< RdHi >$	Ορίζει τον καταχωρητή που περιέχει το περισσότερο σημαντικό word που θα αθροιστεί με το γινόμενο και θα αποθηκεύσει το περισσότερο σημαντικό word του αποτελέσματος.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο του πολλαπλασιασμού.
$< Rs >$	Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο του πολλαπλασιασμού.

Λειτουργία

Αν Ισχύει{cond} τότε

```
    RdLo = (Rm * Rs)[31:0] + RdLo
    RdHi = (Rm * Rs)[63:32] + RdHi
    Αν S == 1 τότε
        N_flag = RdHi[31]
        Z_flag = (αν RdHi == 0 και RdLo == 0 τότε 1 διαφορετικά 0)
        C_flag = (Ανεπηρέαστο)
        V_flag = (Ανεπηρέαστο)
```

Σημειώσεις

R15	Η χρήση του R15 στη θέση των $< RdHi >, < RdLo >, < Rm >, < Rs >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.
RdHi, RdLo & Rm	Οι καταχωρητές αυτοί πρέπει να είναι διαφορετικοί μεταξύ τους, διαφορετικά το αποτέλεσμα δεν είναι προβλέψιμο.

6.1.21 UMULL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	**0 0 0 0 1 0 0**	**S**	**RdHi**	**RdLo**	**Rs**	**1 0 0 1**	**Rm**

Η εντολή UMULL (Unsigned MULtiply Long operands) πολλαπλασιάζει μη-προσημασμένα έντελα για να παράγει αποτέλεσμα εύρους 64 ψηφίων. Το περισσότερο σημαντικό μέρος του γινομένου αποθηκεύεται στον RdHi και το λιγότερο σημαντικό στον καταχωρητή RdLo.

Σύνταξη

UMULL{<cond>}$\{S\} < RdLo >, < RdHi >, < Rm >, < Rs >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**S**}	Όταν υπάρχει, ενεργοποιεί την ανανέωση των σημαιών κατάστασης. Μόνο οι σημαίες N & Z επηρρεάζονται, ανάλογα με το αποτέλεσμα της πράξης.
$< RdLo >$	Ορίζει τον καταχωρητή που θα αποθηκεύσει το λιγότερο σημαντικό word του γινομένου.
$< RdHi >$	Ορίζει τον καταχωρητή που περιέχει το περισσότερο σημαντικό word του γινομένου.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει τον πρώτο όρο του πολλαπλασιασμού.
$< Rs >$	Ορίζει τον καταχωρητή που περιέχει τον δεύτερο όρο του πολλαπλασιασμού.

Λειτουργία

```
Αν Ισχύει{cond} τότε
        RdLo = (Rm * Rs)[31:0]
        RdHi = (Rm * Rs)[63:32]
        Αν S == 1 τότε
                N_flag = RdHi[31]
                Z_flag = (αν RdHi == 0 και RdLo == 0 τότε 1 διαφορετικά 0)
                C_flag = (Ανεπηρέαστο)
                V_flag = (Ανεπηρέαστο)
```

Σημειώσεις

R15	Η χρήση του R15 στη θέση των $< RdHi >, < RdLo >, < Rm >, < Rs >$ θα προκαλέσει απρόβλεπτα αποτελέσματα.
RdHi, RdLo & Rm	Οι καταχωρητές αυτοί πρέπει να είναι διαφορετικοί μεταξύ τους, διαφορετικά το αποτέλεσμα δεν είναι προβλέψιμο.

6.2 Εντολές Μετακίνησης

6.2.1 LDM

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	0	W	1	Rn	Register_list

Η εντολή LDM (LoaD Multiple) μεταφέρει block δεδομένων από την εξωτερική μνήμη στους καταχωρητές.

Σύνταξη

LDM{<cond>} $< addressing_mode >< Rn >$ {!}, $< registers >$

{<**cond**>}	Περιέχει τη συνθήκη με βάση την οποία εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
{**<addressing_mode>**}	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).
$< Rn >$	Ορίζει τον καταχωρητή που θα χρησιμοποιηθεί στον τρόπο διευθυνσιοδότησης της μνήμης. Αν χρησιμοποιηθεί ο R15 θα έχουμε απροσδιόριστα αποτελέσματα.
{!}	Αν υπάρχει, σηματοδοτεί την ανανέωση του Rn μετά την ολοκλήρωση της εντολής. Αν δεν υπάρχει, το περιεχόμενο του Rn δεν μεταβάλλεται.
$< registers >$	Ορίζει μια λίστα από καταχωρητές χωρισμένους με κόμμα, η οποία περικλείεται από { και }. Οι καταχωρητές λαμβάνουν τις τιμές τους ξεκινώντας από τον μικρότερο προς τον μεγαλύτερο (από R0 προς R15) και ο πρώτος αποθηκεύει τα δεδομένα της πρώτης θέσης μνήμης που προσπελαύνεται, ενώ ο τελευταίος της τελευταίας. Για κάθε καταχωρητή 0. . . 15 που συμετέχει στη λίστα, ενεργοποιείται και το αντίστοιχο ψηφίο του πεδίου της εντολής.

Λειτουργία

```
Αν Ισχύει{cond} τότε
        address = start_address
        for i = 0 to 14
                Αν register_list[i] == 1 τότε
                        Ri = Memory[address,4]
                        address = address + 4
        Αν register_list[15] == 1 τότε
                value = Memory[address,4]
                pc = value AND 0xFFFFFFFE
                        address = address + 4
```

Σημειώσεις

Περιορισμοί	Αν ο καταχωρητής Rn υπάρχει στη λίστα και έχει ενεργοποιηθεί η ενημέρωση του μετά την εκτέλεση της εντολής, η τελική τιμή είναι απροσδιόριστη.
Unaligned	Αν ο καταχωρητής Rd δεν περιέχει διεύθυνση που είναι πολλαπλάσιο του 4, θα σημειωθεί εξαίρεση Data Abord.

6.2.2 LDR

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	**0 1**	**I**	**P**	**U**	**0**	**W**	**1**	**Rn**	**Rd**	**addr_mode**

Η εντολή LDR (LoaD Register) μεταφέρει ένα δεδομένο 32 ψηφίων από την εξωτερική μνήμη σε ένα καταχωρητή. Η διεύθυνση της μνήμης υπολογίζεται από το πεδίο addr_mode και η αποθήκευση γίνεται στον καταχωρητή Rd. Αν η διεύθυνση δεν είναι σωστά στοιχισμένη (aligned) σε πολλαπλάσιο του 4, η τιμή ολισθαίνει δεξιά κατά 8 * ψηφία[1:0] της διεύθυνσης (δηλαδή αν αυτά τα ψηφία είναι ίσα με 3, η ολίσθηση θα γίνει κατά 24 θέσεις δεξιά). Σε ένα Little Endian σύστημα όπως το δικό μας, η διεύθυνση που δίνουμε είναι η διεύθυνση του λιγότερο σημαντικού byte του word που θέλουμε να μεταφέρουμε. Αν ο Rd είναι ο R15, τότε ο επεξεργαστής θα εκτελέσει την λήψη του word, θα το αποθηκεύσει στον program counter και θα εκτελέσει διακλάδωση σε αυτή τη διεύθυνση.

Σύνταξη

LDR{<cond>} $< Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή που θα αποθηκεύσει το word από τη μνήμη.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

```
Αν Ισχύει{cond} τότε
    Αν address[1:0] == 2'b00 τότε
        value = Memory[address,4]
    Διαφορετικά αν address[1:0] == 2'b01 τότε
        value = Memory[address,4] Rotate_right 8
    Διαφορετικά αν address[1:0] == 2'b10 τότε
        value = Memory[address,4] Rotate_right 16
    Διαφορετικά αν address[1:0] == 2'b11 τότε
        value = Memory[address,4] Rotate_right 24
    Αν Rd is R15 τότε
        pc = value AND 0xFFFFFFFE
    Διαφορετικά
        Rd = value
```

Χρήση

Αν η εντολή αυτή χρησιμοποιεί τον R15 σαν καταχωρητή βάσης, μπορεί να εκτελεστεί μεταθετός (position-independent) κώδικας, δηλαδή κώδικας που δεν θα προσπελαύνει απόλυτες διευθύνσεις μνήμης, αλλά διευθύνσεις σχετικές με τη διεύθυνση που εκτελείται τώρα. Για παράδειγμα, αν γνωρίζουμε πως οι τιμές που θα επεξεργαστούμε βρίσκονται 85 bytes μετά από την εντολή μεταφοράς, μπορούμε να προσθέσουμε την τιμή του PC και το 85 και να μεταφέρουμε από εκείνη τη διεύθυνση τα δεδομένα. Επειδή το λειτουργικό σύστημα κάθε φορά τοποθετεί τον κώδικα του προγράμματός μας όπου εντοπίσει ελεύθερο χώρο στη μνήμη, οι διευθύνσεις στις οποίες τοποθετείται ο εκτελέσιμος κώδικάς μας δεν είναι ίδιες κάθε φορά, οπότε μας εξυπηρετεί το γεγονός πως προσπελαύνουμε δεδομένα σε σχετικές και όχι σε απόλυτες διευθύνσεις.

Σημειώσεις

Περιορισμοί Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.

Unaligned Αν ο καταχωρητής Rd δεν περιέχει διεύθυνση που είναι πολλαπλάσιο του 4, θα σημειωθεί εξαίρεση Data Abord.

6.2.3 LDRB

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	**0 1**	**I**	**P**	**U**	**1**	**W**	**1**	**Rn**	**Rd**	**addr_mode**

Η εντολή LDRB (LoaD Register with Byte) μεταφέρει ένα δεδομένο 8 ψηφίων από την εξωτερική μνήμη σε ένα καταχωρητή, επεκτείνοντάς το στα 32 ψηφία με προσθήκη 24 ’0’ στα περισσότερο σημαντικά ψηφία του.

Σύνταξη

LDR{<cond>}$B < Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή στον οποίο αποθηκευθεί το δεδομένο.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

Αν Ισχύει{cond} τότε

Rd = Memory[address, 1]

Σημειώσεις

Περιορισμοί Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.

6.2.4 LDRH

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	1	Rn	Rd	addr_mode	1 0 1 1	addr_mode

Η εντολή LDRH (LoaD Register with Halfword) μεταφέρει ένα δεδομένο 16 ψηφίων από την εξωτερική μνήμη σε ένα καταχωρητή, επεκτείνοντάς το στα 32 ψηφία με προσθήκη 16 '0' στα περισσότερο σημαντικά ψηφία του.

Σύνταξη

LDR{<cond>}$H < Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή στον οποίο θα αποθηκευθεί το δεδομένο.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

Αν Ισχύει{cond} τότε
    Αν address[0] == 0 τότε
        data = Memory[address, 2]
    Διαφορετικά
        data = UNPREDICTABLE
    Rd = data

Σημειώσεις

Περιορισμοί	Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.
Unaligned	Αν η διεύθυνση δεν είναι πολλαπλάσιο του 2, η τιμή που αποθηκεύεται στον καταχωρητή είναι απροσδιόριστη και θα προκληθεί εξαίρεση Data Abort.

6.2.5 LDRSB

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	1	Rn	Rd	addr_mode	1 1 0 1	addr_mode

Η εντολή LDRSB (LoaD Register with Signed Byte) μεταφέρει ένα δεδομένο 8 ψηφίων από την εξωτερική μνήμη σε ένα καταχωρητή, επεκτείνοντάς το στα 32 ψηφία με προσθήκη 24 ψηφίων προσήμου στα περισσότερο σημαντικά ψηφία του.

Σύνταξη

LDR{<cond>}$SB < Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή στον οποίο θα αποθηκευθεί το δεδομένο.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

Αν Ισχύει{cond} τότε

data = Memory[address, 1]

Rd = SignExtend(data)

Σημειώσεις

Περιορισμοί Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.

6.2.6 LDRSH

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	1	Rn	Rd	addr_mode	1 1 1 1	addr_mode

Η εντολή LDRSH (LoaD Register with Signed Halfword) μεταφέρει ένα δεδομένο 16 ψηφίων από την εξωτερική μνήμη σε ένα καταχωρητή, επεκτείνοντάς το στα 32 ψηφία με προσθήκη 16 ψηφίων προσήμου στα περισσότερο σημαντικά ψηφία του.

Σύνταξη

LDR{<cond>}$SH < Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή στον οποίο θα αποθηκευθεί το δεδομένο.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

```
Αν Ισχύει{cond} τότε
        Αν address[0] == 0 τότε
                data = Memory[address, 2]
        Διαφορετικά
                data = UNPREDICTABLE
        Rd = SignExtend(data)
```

Σημειώσεις

Περιορισμοί	Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.
Unaligned	Αν η διεύθυνση δεν είναι πολλαπλάσιο του 2, η τιμή που αποθηκεύεται στον καταχωρητή είναι απροσδιόριστη και θα προκληθεί εξαίρεση Data Abort.

6.2.7 MCR

31 28	**27 26 25 24**	**23 21**	**20**	**19 16**	**15 12**	**11 8**	**7 5**	**4**	**3 0**
cond	**1 1 1 0**	**opcode_1**	**0**	**CRn**	**Rd**	**cp_num**	**opcode_2**	**1**	**CRm**

Η εντολή MCR (Move to Coprocessor from Register) μεταφέρει την τιμή του καταχωρητή Rd, στον συνεπεξεργαστή cpu_num. Αν δεν υπάρχει ο συγκεκριμένος συνεπεξεργαστής, η εντολή θα δημιουργήσει την εξαίρεση Undefined Instruction.

Σύνταξη

MCR{<cond>} $< coproc >, < opcod_1 >, < Rd >, < CRn >, < CRm > \{, < opcod_2 >\}$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< coproc >$	Ορίζει το όνομα του συνεπεξεργαστή και θέτει την κατάλληλη τιμή στο πεδίο cp_num της εντολής. Οι γενικές τιμές είναι από p0 εώς p15.
$< opcod_1 >$	Εντολή που εξαρτάται από τον συνεπεξεργαστή.
$< Rd >$	Ορίζει τον καταχωρητή, του οποίου η τιμή θα μεταφερθεί στο συνεπεξεργαστή. Αν χρησιμοποιήσουμε τον R15, το αποτέλεσμα θα είναι απροσδιόριστο.
$< CRn >$	Ο καταχωρητής του συνεπεξεργαστή που θα αποθηκεύσει την τιμή.
$< CRm >$	Δεύτερος καταχωρητής του συνεπεξεργαστή που θα συμμετάσχει στη μεταφορά.
$< opcod_2 >$	Εντολή που εξαρτάται από τον συνεπεξεργαστή. Αν αγνοηθεί, θεωρείται 0.

Λειτουργία

Αν Ισχύει{cond} τότε

Send Rd to Coprocessor[cpu_num]

6.2.8 MOV

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**1 1 0 1**	**S**	**0 0 0 0**	**Rd**	**Shift_operand**

Η εντολή MOV (move) μεταφέρει το αποτέλεσμα του ολισθητή στον καταχωρητή Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

$\text{MOV}\{<\text{cond}>\}\{S\} < Rd >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει, ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες N & Z τίθενται ανάλογα με την τιμή που μετακινείται και η σημαία C ανάλογα με το αν έχει δημιουργηθεί κρατούμενο από ολίσθηση. Οι υπόλοιπες σημαίες κατάστασης δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί η τιμή.

$< shifter_operand >$ Ορίζει την πηγή της τιμής (περιγραφή στο κεφ. 2.3.2, σελ. 16).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
        V_flag = (Ανεπηρέαστο)

Χρήση

Η εντολή αυτή χρησιμοποιείται για να μετακινήσει την τιμή από έναν καταχωρητή σε έναν άλλο, να τοποθετήσει μια σταθερά σε ένα καταχωρητή, να εκτελέσει ολίσθηση χωρίς να ακολουθήσει κάποια άλλη πράξη ή να εκτελέσει διακλάδωση αν ο Rd είναι ο R15.

6.2.9 MRC

31 28	27 26 25 24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	**1 1 1 0**	**opcode_1**	**1**	**CRn**	**Rd**	**cp_num**	**opcode_2**	**1**	**CRm**

Η εντολή MRC (Move to Register from Coprocessor) μεταφέρει ένα έντελο από τον συνεπεξεργαστή cpu_num στον καταχωρητή Rd.Αν δεν υπάρχει ο συγκεκριμένος συνεπεξεργαστής, η εντολή θα δημιουργήσει την εξαίρεση Undefined Instruction.

Σύνταξη

MRC{<cond>} $< coproc >, < opcod_1 >, < Rd >, < CRn >, < CRm > \{, < opcod_2 >\}$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< coproc >$	Ορίζει το όνομα του συνεπεξεργαστή και θέτει την κατάλληλη τιμή στο πεδίο cp_num της εντολής. Οι γενικές τιμές είναι από p0 εώς p15.
$< opcod_1 >$	Εντολή που εξαρτάται από τον συνεπεξεργαστή.
$< Rd >$	Ορίζει τον καταχωρητή, στον οποίο θα μεταφερθεί η τιμή από τον συνεπεξεργαστή. Αν χρησιμοποιήσουμε τον R15, θα ενημερωθούν οι σημαίες κατάστασης αντί του καταχωρητή.
$< CRn >$	Ο καταχωρητής του συνεπεξεργαστή που περιέχει την τιμή.
$< CRm >$	Δεύτερος καταχωρητής του συνεπεξεργαστή που θα συμμετάσχει στη μεταφορά.
$< opcod_2 >$	Εντολή που εξαρτάται από τον συνεπεξεργαστή. Αν αγνοηθεί, θεωρείται 0.

Λειτουργία

Αν Ισχύει{cond} τότε
    data = value from Coprocessor[cpu_num]
    Αν Rd is R15 τότε
        N_flag = data[31]
        Z_flag = data[30]
        C_flag = data[29]
        V_flag = data[27]
    Διαφορετικά
        Rd = data

6.2.10 MRS

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 0
cond	**0 0 0 1 0**	**R**	**0 0**	**0 0 0 0**	**Rd**	**0 0 0 0 0 0 0 0 0 0 0 0**

Η εντολή MRS (Move PSR to Register) μεταφέρει την τιμή του CPSR ή του SPSR στον καταχωρητή Rd.

Σύνταξη

MRS{<cond>} $< Rd >, CPSR$
MRS{<cond>} $< Rd >, SPSR$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

$< Rd >$ Ορίζει τον καταχωρητή, στον οποίο θα μεταφερθεί η τιμή από τον αντίστοιχο PSR.

Λειτουργία

Αν Ισχύει{cond} τότε
    Αν R == 1 τότε
        Rd = SPSR
    Διαφορετικά
        Rd = CPSR

6.2.11 MSR

Immediate operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 0
cond	**0 0 1 1 0**	**R**	**1 0**	**field_mask**	**1 1 1 1**	**rotate_imm**	**8_bit_immediate**

Register operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	**0 0 0 1 0**	**R**	**1 0**	**field_mask**	**1 1 1 1**	**0 0 0 0**	**0 0 0 0**	**Rm**

Η εντολή MSR (Move to xPSR) μεταφέρει μια τιμή (άμεση τιμή περιεχόμενη στην εντολή ή προερχόμενη από τον Rd) στον CPSR ή τον SPSR.

Σύνταξη

MSR{<cond>}$CPSR_ < fields >, \# < immediate >$
MSR{<cond>}$CPSR_ < fields >, < Rm >$
MSR{<cond>}$SPSR_ < fields >, \# < immediate >$
MSR{<cond>}$SPSR_ < fields >, < Rm >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

$< fields >$ Μπορεί να είναι ένας συνδιασμός των παρακάτω:

c Θέτει το ψηφίο 16 της εντολής (control).

x Θέτει το ψηφίο 17 της εντολής (extension).

s Θέτει το ψηφίο 18 της εντολής (status).

f Θέτει το ψηφίο 19 της εντολής (flags).

$< immediate >$ Η άμεση τιμή που θα μεταφερθεί στον CPSR ή SPSR. Οι αποδεκτές τιμές έχουν εύρος 8 ψηφία ή έχουν προέλθει από ολίσθηση μιας τιμής 8 ψηφίων (από 0 ως 31 θέσεις).

$< Rm >$ Είναι ο καταχωρητής, του οποίου η τιμή θα μεταφερθεί στον CPSR ή SPSR.

Λειτουργία

```
Αν Ισχύει{cond} τότε
        Αν opcode[25] == 1 τότε
                operand = 8_bit_immediate ROR (rotate_imm*2)
        Διαφορετικά
                operand = Rm
        Αν R == 0 τότε
                Αν field[0] == 1 AND PrivilegedMode τότε
                        CPSR[23:0] = operand[23:0]
                Αν field[1] == 1 AND PrivilegedMode τότε
                        CPSR[15:8] = operand[15:8]
                Αν field[2] == 1 AND PrivilegedMode τότε
                        CPSR[23:16] = operand[23:16]
                Αν field[3] == 1 τότε
```

```
        CPSR[31:24] = operand[31:24]
Διαφορετικά
    Αν field[0] == 1 AND HasSPSR τότε
        SPSR[23:0] = operand[23:0]
    Αν field[1] == 1 AND HasSPSR τότε
        SPSR[15:8] = operand[15:8]
    Αν field[2] == 1 AND HasSPSR τότε
        SPSR[23:16] = operand[23:16]
    Αν field[3] == 1 AND HasSPSR τότε
        SPSR[31:24] = operand[31:24]
```

6.2.12 MVN

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	**0 0**	**I**	**1 1 1 1**	**S**	**0 0 0 0**	**Rd**	**Shift_operand**

Η εντολή MVN (MoVe Negative) μεταφέρει το συμπλήρωμα ως προς 1 της εξόδου του ολισθητή στον Rd. Οι σημαίες κατάστασης ανανεώνονται κατ' επιλογή, ανάλογα με το αποτέλεσμα.

Σύνταξη

MVN{<cond>}$\{S\} < Rd >, < shifter_operand >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**S**} Όταν υπάρχει, ενεργοποιεί την ανανέωση των σημαιών κατάστασης με βάση το αποτέλεσμα της εντολής. Δυο τύποι ανανέωσης μπορούν να συμβούν:

- Αν ο Rd δεν είναι ο R15, οι σημαίες κατάστασης N & Z τίθενται ανάλογα με την τιμή που μετακινείται και η σημαία C ανάλογα με το αν έχει δημιουργηθεί κρατούμενο από ολίσθηση. Οι υπόλοιπες σημαίες δεν επηρρεάζονται.
- Αν ο Rd είναι ο R15, ο SPSR του ενεργού processor mode αντιγράφεται στον CPSR. Αν εκτελεστεί ενώ είμαστε σε User ή System mode δημιουργείται σφάλμα, επειδή αυτά τα modes δεν έχουν SPSR.

$< Rd >$ Ορίζει τον καταχωρητή που θα αποθηκευτεί η τιμή.

$< shifter_operand >$ Ορίζει την πηγή της τιμής (περιγραφή στο κεφ. 2.3.2, σελ. 16).

Λειτουργία

Αν Ισχύει{cond} τότε
    Rd = NOT shifter_operand
    Αν S == 1 και Rd == 15 τότε
        CPSR = SPSR
    Διαφορετικά αν S == 1 τότε
        N_flag = Rd[31]
        Z_flag = (αν Rd == 0 τότε 1 διαφορετικά 0)
        C_flag = (Bit από ολίσθηση (αν έχει γίνει), αλλιώς 0)
        V_flag = (Ανεπηρέαστο)

Χρήση

Αν η εντολή αυτή χρησιμοποιείται για να υπολογιστεί το συμπλήρωμα ως προς 1 της τιμής ενός καταχωρητή.

6.2.13 STM

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	0	W	0	Rn	Register_list

Η εντολή STM (STore Multiple) μεταφέρει block δεδομένων από τους καταχωρητές στην εξωτερική μνήμη.

Σύνταξη

STM{<cond>} $< addressing_mode >< Rn >$ {!}, $< registers >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

{**<addressing_mode>**} Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

$< Rn >$ Ορίζει τον καταχωρητή που θα χρησιμοποιηθεί στον τρόπο διευθυνσιοδότησης της μνήμης. Αν χρησιμοποιηθεί ο R15 θα έχουμε απροσδιόριστα αποτελέσματα.

{!} Αν υπάρχει, σηματοδοτεί την ανανέωση του Rn μετά την ολοκλήρωση της εντολής. Αν δεν υπάρχει, το περιεχόμενο του Rn δεν μεταβάλλεται.

$< registers >$ Ορίζει μια λίστα από καταχωρητές χωρισμένους με κόμμα, η οποία περικλύεται από { και }. Οι καταχωρητές δίνουν τις τιμές τους ξεκινώντας από τον μικρότερο προς τον μεγαλύτερο (από R0 προς R15) και ο πρώτος αποθηκεύεται στην πρώτη θέση μνήμης που προσπελαύνεται, ενώ ο τελευταίος στην τελευταία. Για κάθε καταχωρητή 0...15 που συμετέχει στη λίστα, ενεργοποιείται και το αντίστοιχο ψηφίο του πεδίου της εντολής.

Λειτουργία

```
Αν Ισχύει{cond} τότε
    address = start_address
    for i = 0 to 14
        Αν register_list[i] == 1 τότε
            Memory[address,4] = Ri
            address = address + 4
```

Σημειώσεις

Περιορισμοί Αν ο καταχωρητής Rn υπάρχει στη λίστα και έχει ενεργοποιηθεί η ενημέρωση του μετά την εκτέλεση της εντολής, η τελική τιμή είναι απροσδιόριστη.

Unaligned Αν ο καταχωρητής Rd δεν περιέχει διεύθυνση που είναι πολλαπλάσιο του 4, θα σημειωθεί εξαίρεση Data Abord.

6.2.14 STR

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	**0 1**	**I**	**P**	**U**	**0**	**W**	**1**	**Rn**	**Rd**	**addr_mode**

Η εντολή STR (STore Register) αποθηκεύει τη τιμή ενός καταχωρητή (32 ψηφία) και στην εξωτερική μνήμη. Η διεύθυνση της μνήμης υπολογίζεται από το πεδίο addr_mode και το δεδομένο προέρχεται από τον καταχωρητή Rd.

Σύνταξη

STR{<cond>} $< Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή που περιέχει το δεδομένο που θα αποθηκευτεί στη μνήμη.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

Αν Ισχύει{cond} τότε

  Memory[address,4] = Rd

Χρήση

Αν η εντολή αυτή χρησιμοποιεί τον R15 σαν καταχωρητή βάσης, μπορεί να εκτελεστεί position-independent κώδικας, δηλαδή κώδικας που δεν θα προσπελαύνει απόλυτες διευθύνσεις μνήμης, αλλά διευθύνσεις σχετικές με τη διεύθυνση που εκτελείται τώρα. Για παράδειγμα, αν γνωρίζουμε πως οι τιμές που θα επεξεργαστούμε βρίσκονται 85 bytes μετά από την εντολή μεταφοράς, μπορούμε να προσθέσουμε την τιμή του pc και το 85 και να μεταφέρουμε σε εκείνη τη διεύθυνση τα δεδομένα. Επειδή το λειτουργικό σύστημα μπορεί να φορτώσει στη μνήμη τον κώδικα του προγράμματός μας όπου εντοπίσει ελεύθερο χώρο, δεν είναι ίδιες οι διευθύνσεις που φορτώνεται κάθε φορά, οπότε μας εξυπηρετεί το γεγονός πως προσπελαύνουμε δεδομένα σε σχετικές και όχι σε απόλυτες διευθύνσεις.

Σημειώσεις

Περιορισμοί	Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.
Unaligned	Αν ο καταχωρητής Rd δεν περιέχει διεύθυνση που είναι πολλαπλάσιο του 4, θα σημειωθεί εξαίρεση Data Abord.

6.2.15 STRB

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	**0 1**	**I**	**P**	**U**	**1**	**W**	**0**	**Rn**	**Rd**	**addr_mode**

Η εντολή STRB (STore Register Byte) μεταφέρει ένα δεδομένο 8 ψηφίων από ένα καταχωρητή στην εξωτερική μνήμη.

Σύνταξη

STR{<cond>}$B < Rd >, < addressing_mode >$

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

$< Rd >$ Ορίζει τον καταχωρητή που περιέχει το δεδομένο που θα αποθηκευθεί στη μνήμη.

$< addressing_mode >$ Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

Αν Ισχύει{cond} τότε

Memory[address, 1] = Rd[7:0]

Σημειώσεις

Περιορισμοί Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.

6.2.16 STRH

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	0	Rn	Rd	addr_mode	1 0 1 1	addr_mode

Η εντολή STRH (STore Register Halfword) μεταφέρει ένα δεδομένο 16 ψηφίων από ένα καταχωρητή στην εξωτερική μνήμη.

Σύνταξη

STRH{<cond>}$H < Rd >, < addressing_mode >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή που περιέχει το δεδομένο που θα αποθηκευθεί στη μνήμη.
$< addressing_mode >$	Ορίζει τον τρόπο διευθυνσιοδότησης της μνήμης (περιγραφή στο κεφ. 2.3.3, σελ. 20).

Λειτουργία

```
Αν Ισχύει{cond} τότε
    Αν address[0] == 0 τότε
        data = Rd[15:0]
    Διαφορετικά
        data = UNPREDICTABLE
    Memory[address,2] = data
```

Σημειώσεις

Περιορισμοί	Αν το $< addressing_mode >$ ορίζει ενημέρωση του καταχωρητή βάσης μετά την πρόσβαση στη μνήμη και χρησιμοποιούμε τον ίδιο καταχωρητή στα Rd & Rn, η τελική τιμή του Rd είναι απροσδιόριστη.
Unaligned	Αν η διεύθυνση δεν είναι πολλαπλάσιο του 2, η τιμή που αποθηκεύεται στον καταχωρητή είναι απροσδιόριστη και θα προκληθεί εξαίρεση Data Abort.

6.2.17 SWP

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm

Η εντολή SWP (SWaP) ανταλλάσει τιμές ανάμεσα σε μια θέση μνήμης και ένα καταχωρητή. Η τιμή του καταχωρητή Rm αποθηκεύεται στη θέση μνήμης που υποδεικνύεται από τον καταχωρητή Rn και η τιμή που υπήρχε εκεί αποθηκεύεται στον καταχωρητή Rd. Αν χρησιμοποιηθεί ο ίδιος καταχωρητής στη θέση των Rn & Rm, θα γίνει ανταλλαγή τιμών ανάμεσα στον ίδιο καταχωρητή και τη μνήμη.

Σύνταξη

SWP{<cond>} $< Rd >, < Rm >, [< Rn >]$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή στον οποίο θα αποθηκευθεί το δεδομένο της μνήμη.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει το δεδομένο που θα αποθηκευθεί στη μνήμη.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τη διεύθυνση της μνήμης.

Λειτουργία

```
Αν Ισχύει{cond} τότε
        Αν Rn[1:0] == 0'b00 τότε
                temp = Memory[Rn,4]
        Διαφορετικά αν Rn[1:0] == 0'b01 τότε
                temp = Memory[Rn,4] ROR 8
        Διαφορετικά αν Rn[1:0] == 0'b10 τότε
                temp = Memory[Rn,4] ROR 16
        Διαφορετικά αν Rn[1:0] == 0'b11 τότε
                temp = Memory[Rn,4] ROR 24


        Memory[Rn,4] = Rm
        Rd = temp
```

6.2.18 SWPB

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 1 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm

Η εντολή SWPB (SWaP Byte) ανταλλάσει τιμές εύρους 8 ψηφίων ανάμεσα σε μία θέση μνήμης και έναν καταχωρητή (χρησιμοποιώντας τα λιγότερο σημαντικά ψηφία του). Η τιμή του καταχωρητή Rm αποθηκεύεται στη θέση μνήμης που υποδεικνύεται από τον καταχωρητή Rn και η τιμή που υπήρχε εκεί αποθηκεύεται στον καταχωρητή Rd. Αν χρησιμοποιηθεί ο ίδιος καταχωρητής στη θέση των Rn & Rm, θα γίνει ανταλλαγή τιμών ανάμεσα στον ίδιο καταχωρητή και τη μνήμη.

Σύνταξη

SWP{<cond>}$B < Rd >, < Rm >, [< Rn >]$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rd >$	Ορίζει τον καταχωρητή στον οποίο θα αποθηκευθεί το δεδομένο της μνήμη.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει το δεδομένο που θα αποθηκευθεί στη μνήμη.
$< Rn >$	Ορίζει τον καταχωρητή που περιέχει τη διεύθυνση της μνήμης.

Λειτουργία

Αν Ισχύει{cond} τότε

temp = Memory[Rn,1]
Memory[Rn,1] = Rm[7:0]
Rd = temp

Σημείωση

Οι εντολές SWP & SWPB μπορούν να χρησιμοποιηθούν για την υλοποίηση σημαφόρων.

6.3 Εντολές Διακλάδωσης

6.3.1 B,BL

31 28	27 26 25	24	23 0
cond	1 0 1	L	signed_immed_24

Οι εντολές **B** (Branch) & **BL** (Branch and Link) οδηγούν τον επεξεργαστή σε διακλάδωση και αλλαγή της ροής εκτέλεσης του προγράμματος.

Σύνταξη

B{L}{<cond>} $< target_address >$

L Ενεργοποιεί την αποθήκευση μιας διεύθυνσης επιστροφής στον καταχωρητή R14.

{<**cond**>} Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.

$< target_address >$ Ορίζει την διεύθυνση όπου θα γίνει η διακλάδωση. Η διεύθυνση υπολογίζεται με τα ακόλουθα βήματα :

1. Γίνεται προσημασμένη επέκταση της άμεσης τιμής $< target_address >$ που παρέχεται στην εντολή (ψηφία 23 έως 0 του κωδικού λειτουργίας) από τα 24 στα 32 ψηφία.
2. Το αποτέλεσμα ολισθαίνει αριστερά κατά 2 ψηφία.
3. Η ολισθημένη τιμή προστίθεται στο (PC + 8).

Η παραπάνω διαδικασία μας επιτρέπει να εκτελέσουμε διακλάδωση σε μια απόσταση $\pm 32MB$ από τη θέση που δείχνει ο PC.

Λειτουργία

Αν Ισχύει{cond} τότε
    Αν L == 1 τότε
        LR = address of next instruction
    PC = PC + SignExtend(immed_24) LeftShift 2

Χρήση

Η εντολή BL χρησιμοποιείται για να υλοποιήσει μια κλήση υπορουτίνας. Η επιστροφή από την υπορουτίνα γίνεται με την αντιγραφή του R14 στον R15. Η τιμή που δίνεται στην εντολή δεν χρειάζεται να είναι αριθμός αλλά ετικέτα, διότι ο Assembler εκτελεί την μετάφραση της ετικέτας σε μετατόπιση.

6.3.2 BX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	Rm

Οι εντολή **BX** (Branch and eXchange) οδηγεί τον επεξεργαστή σε διακλάδωση στη διεύθυνση που βρίσκεται αποθηκευμένη στον καταχωρητή Rm, ενώ μπορεί και να αλλάξει κατάσταση εκτέλεσης εντολών από ARM σε Thumb. Η διεύθυνση της μνήμης πρέπει να είναι πολλαπλάσιο του 4.

Σύνταξη

BX{<cond>} $< Rm >$

{<**cond**>}	Περιέχει τη συνθήκη βάσει της οποίας εκτελείται η εντολή. Αν δεν έχει συμπληρωθεί, η εντολή εκτελείται πάντα.
$< Rm >$	Ορίζει τον καταχωρητή που περιέχει τη διεύθυνση διακλάδωσης.

Λειτουργία

Αν Ισχύει{cond} τότε
    T Flag == Rm[0] τότε
        LR = address of next instruction
    PC = Rm AND 0xFFFFFFFE

Χρήση

Η εντολή BX μπορεί να χρησιμοποιηθεί σε περιπτώσεις όπου μια διεύθυνση της κύριας μνήμης έχει αποθηκευτεί σε ένα καταχωρητή και θέλουμε να εκτελέσουμε διακλάδωση σε εκείνη τη θέση. Εκτός από την λειτουργία αλλαγής είδους εκτελέσιμου κώδικα ARM, Thumb, είναι ακριβώς ίδια στη λειτουργία με μια MOV στον PC.