Register File

Name	Use
R0	argument / temporary variable / *return value*
R1 - R3	argument / temporary variable
R4 - R8	saved variables
R9	saved variable / *static base pointer SB*
R10 - R11	saved variables
R12	temporary variable
R13	***stack pointer SP***
R14	***link register LR***
R15	***program counter PC***
CPSR	*Current Program Status Register CPSR*

Condition Flags

Ο καταχωρητής CPSR δεν είναι κανονικά ανάμεσα στους άλλους 16 του register file, είναι υποτίθεται ξεχωριστός και έχει τις σημαίες συνθήκης. Αποθηκεύονται στα 4 msb του CPSR

Flag	Name/Description
N	Negative
Z	Zero
C	Carry
V	Overflow

Οι εντολές υπό συνθήκη εκτελλούνται ανάλογα την κατάσταση των σημαιών και ορίζονται mnemonics βάσει ακριβώς την κατάσταση των σημαιών.
Η εντολές έχουν πεδίο συνθήκης 4bit cond. Τα bit του μετρώνται 3 downto 0. Οι συνθήκες με το lsb 0 είναι συμπληρωματικές αυτών που έχουν lsb 1 (δεν ισχύει για τις δύο τελευταίες).

cond	Mnemonic	Name	CondEX/flag
0000	EQ	Equal	Z
0001	NE	Not Equal	$\bar{Z}$
0010	CS/HS	Carry Set / unsigned Higher or Same	C
0011	CC/LO	Carry Clear / unsigned LOwer	$\bar{C}$
0100	MI	MInus / Negative	N
0101	PL	PLus / Positive or Zero	$\bar{N}$
0110	VS	overflow / oVerflow Set	V
0111	VC	no overflow / oVerflow Clear	$\bar{V}$
1000	HI	unsigned HIgher	$\bar{Z}C$
1001	LS	unsigned Lower or Same	$\bar{C} + Z$
1010	GE	signed Greater or Equal	$\overline{N \oplus V}$
1011	LT	signed Less Than	$N \oplus V$
1100	GT	signed Greater Than	$\bar{Z}\overline{N \oplus V}$
1101	LE	signed Less or Equal	$Z + (N \oplus V)$
1110	AL (or none)	ALways / unconditional	*Ignored*
1111	none	none	*Unconditional*

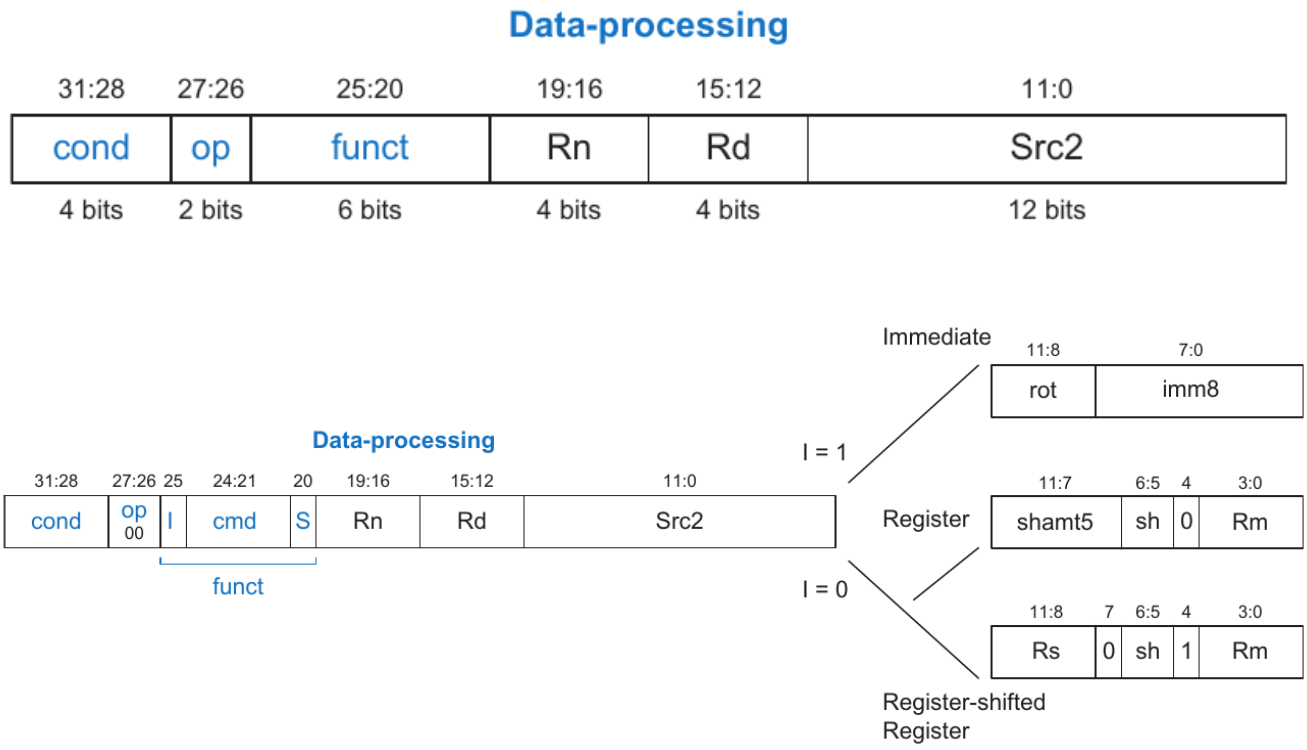
- τα higher είναι για unsigned, τα greater είναι για signed
- τα lower είναι για unsigned, τα less είναι για signed
- τα same είναι για unsigned, τα equal είναι για signed
- οι unsigned χρησιμποιούν το flag C και Z, ενώ οι signed V, N, Z

- Σε unsigned αν A-B>=0 παράγεται πάντα κρατούμενο κατά την αφαίρεση, εξ ου και C=1
- Το carry προκύπτουν αν βγει 1 σε κάποια μεγαλύτερη δύναμη bit από αυτή που ορίζουν οι δύο τελεστέοι της πράξης. Δηλαδή η πρόσθεση 001001 + 001110 εστιάζει στα 4bit, γιατί μέχρι εκεί εμφανίζουν οι τελεστέοι δύναμη. Αν το αποτέλεσμα έχει 1 στο bit 5 τότε θα είναι overflow. Πράγματι 001001+01110 = 1001+1110 = 010111 = 10111. Πρακτικά κάνει 0111 και C=1
- Το overflow προκύπτει όταν οι τελεστέοι έχουν κοινό πρόσημο και το αποτέλεσμα ανάποδο. Όλα αυτά στο ίδιας δύναμης bit. Δηλαδή στο πάνω παράδειγμα: 001001+01110 = 1001+1110 γιατί μας νοιάζουν μόνο τα 4bit και 1001+1110 = 0111 και 1 που δε μας νοιάζει γιατί βλέπουμε ότι το πρόσημο άλλαξε και V=1

Data-Processing Instructions

Όσες έχουνε (S) σε παρένθεση σημαίνει ότι μπορούν να εκτελεστούν υπό συνθήκη

Έχουν γενική μορφή:



Τα πεδία cond και op υπάρχουν για κάθε τύπο εντολής. Το funct υπάρχει σε data processing και memory, όπως και οι τελεστές

- ***cond*** : conditional execution βασισμένο στα flags του CPSR (βλέπε πίνακα με mnemonics)
- ***op*** : ο τύπος / μορφή της εντολής, που για **εντολές επεξεργασίας δεδομένων είναι op = 00**
- ***funct***: function code είναι η κωδικοποίηση της εντολής
    - *I* : είναι '1' για immidiate και '0' για register
    - *cmd* : command για τη δήλωση της πράξης
    - *S* : είναι 1 αν ενημερώνει τις σημαίες συνθήκης του CPRS
- ***Rn*** : καταχωτητής προέλευσης του πρώτου τελεστέου (του αριστερά από αυτούς της προέλευσης)
- ***Rd*** : καταχωρητής προορισμού στο περιεχόμενο του οποίο αποθηκεύεται το αποτέλεσμα της πράξης
- ***Src2*** : προσδιορίζει τον δεύτερο (δεξιά) τελεστέο προέλευσης
    - **Άμεσος τελεστέος**
        - *rot* : δήλωση άμεσης ποσότητας περιστροφής/rotation προς τα δεξιά του τελεστέου 8 bit για να γίνει τελεστης 32 bit
        - *imm8* : πεδίο αποθήκευσης άμεσου τελεστέου 8 bit
    - **Καταχωρητής - μπορεί να κάνει shift από σταθερά**
        - *Rm* : καταχωρητής προέλευσης δεύτερου (δεξιά) τελεστέου
        - *shamt5* : shift amount των 5bit για τη δήλωση της ποσότητας ολίσθησης (τιμές 0 έως 31)
        - *sh* : δηλώνει τον τύπο της ολίσθησης
        - ανάμεσα από το *Sh* και το *Rm* υπάρχει ένα bit με τιμή '0'
    - **Καταχωρητής - μπορεί να κάνει shift από άλλο register**
        - *Rm* : καταχωρητής προέλευσης δεύτερου (δεξιά) τελεστέου
        - *sh* : δηλώνει τον τύπο της ολίσθησης
        - *Rs* : άλλος register που δείχνει την ποσότητα ολίσθησης
        - ανάμεσα από το *Sh* και το *Rm* υπάρχει ένα bit με τιμή '1'

Arithmetic Instructions

ADD(S)

Προσθετει το περιεχόμενο των καταχωρητών r1 και r2 και το αποτέλεσμα το αποθηκεύει στον r0

```
ADD r0, r1, r2
```

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*0100*	0	r1	r0	00000	00	0	r2

Προσθέτει μία σταθερά στο περιεχόμενο του r1 και αποθηκεύει το αποτέλεσμα στον καταχωρητή r0. Η σταθερά είναι μη προσημασμένη 8-12 bit (256 έως 4096 τιμές) και μπορεί να είναι σε δεκαδική ή δεκαεξαδική μορφή

```
ADD r0, r1, #42
ADD r0, r1, #0xFF0 ; xB2 == 4080 == b(1111 1111 0000) (12 bit)
```

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*0100*	0	r1	r0	0000	00101010

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*0100*	0	r1	r0	1110	11111111

Δηλαδή rot=14 και κάναμε ror = 2 x 14 = 28 θέσεις

SUB(S)

Αφαιρεί το περιεχόμενο των καταχωρητών r1 και r2 και το αποτέλεσμα το αποθηκεύει στον r0

```
SUB r0, r1, r2  ; πάντα κάνει r0 = r1 - r2
```

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*0010*	0	r1	r0	00000	00	0	r2

Αφαιρεί μία σταθερά από το περιεχόμενο του r1 και αποθηκεύει το αποτέλεσμα στον καταχωρητή r0. Η σταθερά είναι μη προσημασμένη 8-12 bit (256 έως 4096 τιμές) και μπορεί να είναι σε δεκαδική ή δεκαεξαδική μορφή. Η σταθερά πάντα αφαιρείται!

```
SUB r0, r1, #16
SUB r0, r1, #0xB2 ; xB2 == 178 (τουλάχιστον 8bit)
```

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*0010*	0	r1	r0	0000	00010000

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*0010*	0	r1	r0	0000	10110010

Logical Instructions

AND(S)

Λογικό and μεταξύ δύο τελεστέων προορισμού και το αποτέλεσμα ένας τελεστέος προέλευσης. Συνήθως είναι registers. Λειτουργεί bit προς bit και πάντα οι τελεστέοι είναι 32bit

```
AND r3, r1, r2
AND r4, r6, #BC000003   ; binary: 1011 1100 0000 0000 0000 0000 0000 0011
```

-

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*0000*	0	r1	r3	00000	00	0	r2

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*0000*	0	r6	r4	0011	11101111

Πραγματοποιήθηκε rotate right κατά rot x 2 = 3 x 2 = 6 θέσεις

EOR(S) (xor)

Exclusive or μεταξύ δύο τελεστέων προορισμού και το αποτέλεσμα ένας τελεστέος προέλευσης. Συνήθως είναι registers. Λειτουργεί bit προς bit και πάντα οι τελεστέοι είναι 32bit

```
EOR r3, r1, r2
EOR r4, r6, #0xBC000003
```

-

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*0001*	0	r1	r3	00000	00	0	r2

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*0001*	0	r6	r4	0011	11101111

Πραγματοποιήθηκε rotate right κατά rot x 2 = 3 x 2 = 6 θέσεις

MVN(S) (not)

Δέχεται μόνο έναν τελεστέο προέλευσης και έναν προορισμού. Κάνει αντιστροφή όλα τα bit του τελεστέου προέλευσης (είτε immidiate είτε register) και εγγράφει το αποτέλεσμα στον προορισμό (πάντα μνήμη)

```
MVN r7, r2
MVN r3, #0x3A0000008 = b(1010 0000 0000 0000 0000 0000 0000 1000)
```

-

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*1111*	0	*0000*	r7	00000	00	0	r2

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*1111*	0	*0000*	r3	0010	10001010

Πραγματοποιήθηκε rotate right κατά rot x 2 = 2 x 2 = 4 θέσεις

*Παρατήρησε ότι σε αυτήν την εντολή **δε χρησιμοποιείται ο τελεστέος Rn (αναγκαστικά Rn=0)**, ενώ ο δεύτερος τελεστέος στο υποπεδίο **Rm εκλαμβάνεται ως πρώτος***

Shift Instructions

MOV

Γράφει την τιμή μίας σταθεράς ή μίας μεταβλητής (ουσιαστικά το περιεχόμενο κάποιου άλλου register) στο περιεχόμενο ενός register
Αριστερά είναι ο τελεστέος προορισμού και δεξιά είναι προέλευσης. Χρησιμοποιείται για αρχικοποίηση και ανάθεση (αντιγραφή) τιμών.

```
MOV r4, #8
MOV r5, #0xFF0    ; xFF0 = 4080 άρα θεωρώ 12bit πεδίο σταθεράς
MOV r4, r5      ; πλέον R4=4080
```

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*1101*	0	*0000*	r4	0000	00001000

-

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*1101*	0	*0000*	r5	1100	11111111

-

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*1101*	0	*0000*	r4	00000	00	0	r5

Παρατήρησε ότι σε αυτήν την εντολή ***δε χρησιμοποιείται ο τελεστέος Rn (αναγκαστικά Rn=0)****, ενώ ο δεύτερος τελεστέος στο υποπεδίο* ***Rm εκλαμβάνεται ως πρώτος***

LSL

Λογικό shift left - το οποίο ισοδυναμεί και με asl - γιατί δεν το νοιάζει το πρόσημο, κάνει απλά shift left.
Αριστερά είναι ο τελεστέος προέλευσης στον οποίο εγγραφεται το αποτέλεσμα και δεξιά είναι ο τελεστέος προορισμού με το shift amount (shamt)

Το shamt είναι ένας άμεσος τελεστέος, απρόσημος ακέραιος που αυτή τη φορά προσδιοπρίζει **bit** και θα έχει τιμή από 0 έως 31. Η τιμή 32 δε θα είχε νόημα, όπως ούτε ιδιαίτερο νόημα έχει και η τιμή 0, γιατί τότε απλά δε γίνεται κανένα shift

Η LSL προσφέρεται ως πολλαπλασιασμό επί δύο όσες φορές όσες το shift

```
LSL r0, r5, #7      ; 7 bit shift left στο περιεχόμενο του r5 και αποθήκευση στον r7
```

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*1101*	0	*0000*	r0	00111	00	0	r5

Παρατήρησε:

- το πεδίο sh=00 είναι αυτό που παραπέμπει σε LSL, το οποίο είναι ίδιο με τη MOV. Γενικώς η **MOV, με register, είναι ολόιδια απλά χωρίς shamt5**
- έχει ίδιο command code με τις άλλες πράξεις ολίσθησης cmd=1101
- παρότι είναι πράξη με imidiate έχουμε I=0 και χρησιμοποιείται το format που στις άλλες εντολές χρησιμοποιείται σε register πράξεις

Στην παρακάτω δομή, δηλαδή **register shifted by register** ό,τι είναι στο Rm (r8) θα δεχθεί shift όσο και τα 8 lsb του περιεχομένου του Rs και το αποτέλεσμα θα αποθηκευτεί στο Rn

```
MOV r6, #0xF001001C     ; bin = 1111 0000 0000 0001 0000 0000 0001 1100
LSL r4, r8, r6          ; shamt = 0001 1100 = 0x1C = 28
                        ; ο r8 θα δεχθεί αριστερό shift 28 bit
                        ; το αποτέλεσμα θα αποθηκευθεί στον r4
```

cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm
1110	00	*0*	*1101*	0	*0000*	r4	r6	0	00	*1*	r8

Παρατήρησε

- Δομή register shifted by register (που είναι η τρίτη κατηγορία του Src2)
- Ο Rn πάλι δε χρησιμοποιείται, ενώ σαν πρώτος (αριστερά) τελεστέος προέλευσης

θεωρείται ο Rm και ως δεύτερος (δεξιά) ο Rs
- υπάρχουν δύο μη χρησιμοποιημένα bit δεξιά και αριστερά του πεδίου sh. Το δεξιά υπάρχει και στις άλλες εντολές με register, απλά έχει τιμή 0, ενώ εδώ έχει τιμή 1

ASR(S)

Η ASR προσφέρεται για διαίρεση με το δύο όσες φορές όσες το shift, αγνοώντας το υπόλοιπο

Αριθμητική ολίσθηση στα δεξιά, που σημαίνει ότι **λαμβάνει υπόψη και το πρόσημο**

```
ASR r0, r5, #0xB    ; 11 bit δεξιά ολίσθηση
                    ; αν η τιμή ξεκίναγε από '1' θα προστεθούν 11 '1' μπροστά
```

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*1101*	0	*0000*	r0	01011	10	0	r5

Παρατήρησε:

- το πεδίο sh=10 είναι αυτό που παραπέμπει σε ASR
- έχει ίδιο command code με τις άλλες πράξεις ολίσθησης cmd=1101
- παρότι είναι πράξη με immidiate έχουμε I=0 και χρησιμοποιείται το format που στις άλλες εντολές χρησιμοποιείται σε register πράξεις

Η παρακάτω είναι πράξη **register shift by register** και είναι ίδιας λογικής με αυτήν στην LSL

```
ASR r5, r1, r12
```

cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm
1110	00	*0*	*1101*	0	*0000*	r5	r12	0	10	*1*	r1

Conditional Instruction

CMP

Αφαιρεί από τον πρώτο/αριστερό τελεστέο προέλευσης (πάντα περιεχόμενο καταχωρητή) τον δεύτερο/δεξιά τελεστή προέλευσης (είτε σε καταχωρητή είτε άμεσος)
Το αποτέλεσμα ενημερώνει κατευθείαν τον CPSR

```
MOV r3, #18
MOV r4, #0x12   ; x12 = 18
CMP r4, r3      ; CPSR flag = Z (zero), δηλαδή είναι ίσα
```

cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	*0*	*1010*	*1*	r4	0000	00000	00	0	r3

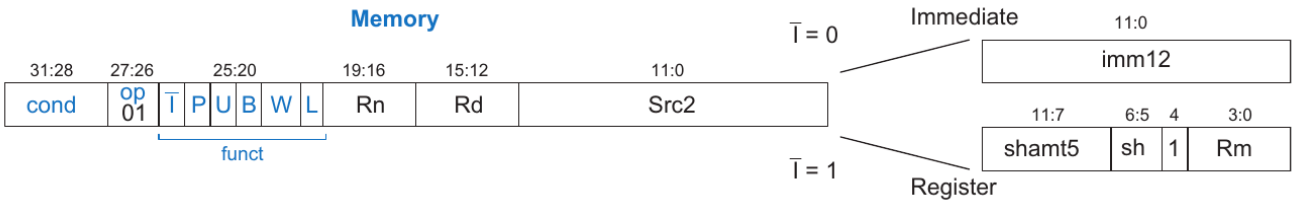
```
MOV r5, #25
CMP r5, #19     ; CPSR flag Z = 1, δηλαδή είναι ίσα
```

cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	*1*	*1010*	*1*	r5	*0000*	0000	00011001

```
MOV r1, #9      ; 9 = b1001
MOV r2, #2      ; 2 = b0010
CMP r1, r2      ; 9-2=7=b0111 και θα έχω C=1
                ; θα ενεργοποιεί τα mnemonics CS/HS, HI

MOV r1, #-7     ; -7 = b1001
MOV r2, #2      ; 2 = b0010
CMP r1, r2      ; -7-2=-9: δε μπορεί να αναπαρασταθεί με 4bit σαν signed
                ; δίνει αποτέλεσμα 0111 και V=1, ενεργοποιώντας V=1, N=0
                ; θα ενεργοποιεί mnemonics LT, LE

MOV r1, #5      ; 5 = b0101
MOV r2, #13     ; 13 = b1101
CMP r1, r2      ; 5-13=1000=-8
                ; δεν αναπαριστάται κανονικά στους unsigned
                ; flags: C=0 , mnemonics: LO, LS

MOV r1, #5      ; 5 = b0101
MOV r2, #-3     ; -3 = b1101
CMP r1, r2      ; 5+3=8 που δεν αναπαρίσταται σε signed 4bit
                ; αποτέλεσμα -8=b1000 υπερχείλιση προς τα κάτω
                ; flags: N=1, V=1, Z=0 , mnemonics: GE, GT

MOV r2, #0x80000000 ; binary: 100...000 (32 bit)
MOV r3, #0x00000001 ; binary: 111...111 (32 bit)
CMP r2, r3          ; 0x80000000 + 0xFFFFFFFF = 0x17FFFFFFF
                    ; δηλαδή 0x7FFFFFFF=b011...111 και 1 έξτρα
                    ; για signed είναι 011...111 με overflow (negativest - 1 = sign flip)
                    ; για unsigned είναι 011...111 με carry (maximum + κάτι)
ADDEQ r4, r5, #78   ; OXI: γιατί Z=0 και NE
ANDHS r7, r8, r9    ; NAI: γιατί C=1
SUBMI r10, r11, r12 ; OXI: γιατί N=0 (bit 31 είναι 0)
EORLT r12, r7, r10  ; NAI: γιατί N=0, V=1 άρα N xor V = 1
```

Memory Instructions

Έχουν γενική μορφή:



Τα πεδία cond και op υπάρχουν για κάθε τύπο εντολής. Το funct υπάρχει σε data processing και memory, όπως και οι τελεστές

- ***cond*** : conditional execution βασισμένο στα flags του CPSR (βλέπε πίνακα με mnemonics)
- ***op*** : ο τύπος / μορφή της εντολής, που για **εντολές επεξεργασίας μνήμης είναι op = 01**
- ***funct***: function code είναι η κωδικοποίηση της εντολής
    - $\bar{I}$: inverse immediate για τη δήλωση **ύπαρξης άμεσου τελεστέου, όταν** $\bar{I} = 0$
    - *P* & *W* : ορίζουν τον τρόπο διευθυνσιοδότητης μνήμης και **όταν PW=10 η διευθυνσιοδότηση γίνεται με σχετική απόσταση / offset**
    - *B* & *L* : για να καθορίζουν τον τύπο εντολής. Για **BL=01 η εντολή είναι LDR**. Για **BL=00 η εντολή είναι STR**.
    - *U* : ορίζει την πράξη της σχετικής απόστασης. Για **U=1 είναι πρόσθεση**, ενώ για **U=0 είναι αφαίρεση**.
- ***Rn*** : καταχωτητής βάσης (αυτός που συνήθως βρίσκεται μέσα σε [...])
- ***Rd*** : είναι ο καταχωρητής προέλευσης στην εντολή store και ο καταχωρητής προορισμού στην εντολή load.
- ***Src2*** *όπου για immidiate* ***imm12*** : πεδίο προσδιορισμού σχετικής απόστασης / offset ως άμεσος τελεστέος 12 bit

Για το offset (τελευταίο πεδίο) θα εξετάσουμε μόνο τον άμεσο τελεστέο και όχι το register. Για αυτό και τα πεδία PW θα είναι πάντα PW=10, όπως και το $\bar{I} = 0$ *στην περίπτωσή μας*

LDR

Η εντολή αυτή φορτώνει δεδομένα από τη μνήμη στους καταχωρητές. Η μνήμη είναι 32bit words, byte addressible, little-endian, άρα προσμετράτε από το lsbyte στο msbyte και από κάτω προς τα πάνω.

```
MOV r5, #0          ; ο r5 θα χρησιμοποιηθεί σαν καταχωρητής βάσης

LDR r7, [r5, #8]    ; αποθήκευση στο r7 το περιεχόμενο μνήμης που απέχει 2 λέξεις από βάση
                    ; η βάση είναι το 0 και το 8 σημαίνει 8 bytes παραπέρα
                    ; συν 8bytes είναι 2 λέξεις πάνω
```

cond	op	IPUBWL	Rn	Rd	imm12
1110	01	011001	r5	r7	000000001000

STR

Η εντολή εγγράφει περιεχόμενο ενός καταχωρητή από το regfile σε μία θέση μνήμης. **Προσοχή** γιατί έχει παρόμοιο συντακτικό με τη LDR αλλά είναι ανάποδα. Δηλαδή ο αριστερά τελεστέος είναι προέλευσης και ο δεξιά είναι ο προορισμού

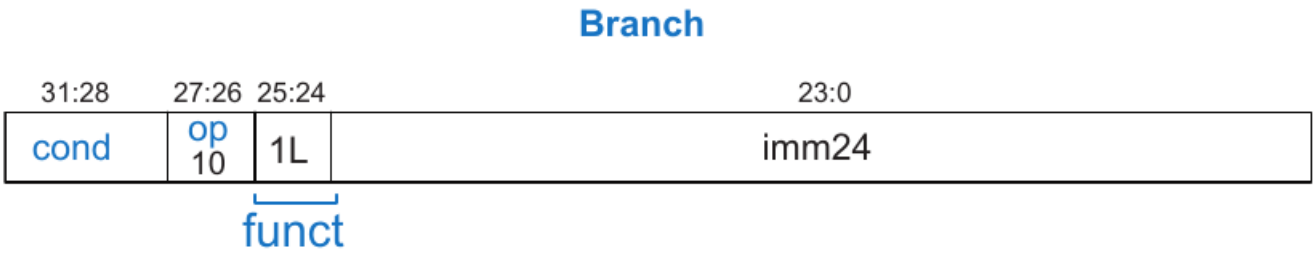
```
MOV r5, #0x9FFFA654
MOV r9, #0x000000FF     ; xFF = 255
STR r9, [r5, #-0x10]    ; να σου θυμήσω ότι x10 = 16 = 4 words
                        ; έτσι 255 εγγράφηκε στην 4η λέξη κάτω από το base
                        ; δηλαδή στη μνήμη διεύθυνσης 0x9FFFA644
```

cond	op	IPUBWL	Rn	Rd	imm12
1110	01	010000	r5	r9	000000010000

Branch Instructions

Εκτελούνται υποσυνθήκη βάσει των μνημονικών συνθήκης

Έχουν γενική μορφή:



Τα πεδία cond και op υπάρχουν για κάθε τύπο εντολής.

- ***cond*** : conditional execution βασισμένο στα flags του CPSR (βλέπε πίνακα με mnemonics)
- ***op*** : ο τύπος / μορφή της εντολής, που για **εντολές επεξεργασίας διακλαδωσης είναι op = 10**
- ***funct*** *ή* ***1L***: function code είναι η κωδικοποίηση της εντολής και για εντολές διακλάδωσης είναι δύο bit. Το πρώτο (αριστερά) είναι πάντα 1, ενώ το δεύτερο είναι το L, όπου για **L=1 έχω την εντολή branch and link / BL**
- ***imm24*** : πεδίο προσδιορισμού **διεύθυνσης στόχου της διακλάδωσης / target address**, ως άμεσος τελεστέος.

B(S)

Απλή διακλάδωση / simple branch. Η εντολή ορίζεται ετικέτα όπου από εκεί θα συνεχιστεί η εκτέλεση, που σε machine code είναι διευθύνσεις εντολών.
Η ARM assembly έχει formatting. Οι εντολές θέλουν κενό, ή tab, ενώ τα branch labels όχι.

Επίσης, είναι πολύ σημαντικό να καταλάβει κανείς ότι **οι εντολές που βρίσκονται τα labels, *που η διεύθυνσή τους ονομάζεται Branch Target Address / BTA*, δεν είναι κάποια ειδική εντολή με label, οπότε η θέση μνήμης τους είναι της επόμενης εντολής**. Και αυτό γιατί η

εντολή B είναι αυτή που υλοποιεί και προσδιορίζει τη μεταπήδηση σε συγκεκριμένο branch. Το label υπάρχει στην Assembly για δική μας διευκόλυνση.

Στα παραδείγματα έχω και τις υποθετικές διευθύνσεις μνήμης και στα σχόλια τη σειρά εκτέλεσης

```
0x8004  ADD r1, r2, r3      ; (1)
0x8008  B TARGET            ; (2) διακλάδωση στο TARGET, που είναι στη BTA
0x800C  EOR r2, r4, r3      ; ΔΕΝ ΕΚΤΕΛΕΙΤΑΙ
0x8010  AND r8, r2, #3      ; ΔΕΝ ΕΚΤΕΛΕΙΤΑΙ
                            ; θέση του PC+08 από την εντολή που κάλεσε το B
0x8014  SUB r9, r8, #-0xB   ; ΔΕΝ ΕΚΤΕΛΕΙΤΑΙ
0x8018  ADD r10, r2, #2     ; ΔΕΝ ΕΚΤΕΛΕΙΤΑΙ
0x801C TARGET               ; Αυτή δεν είναι χωριστή εντολή, αλλά μαζί με την επόμενη
0x801C  SUB r1, r3, r4      ; (3) εκτελείται και είναι η BTA
```

Εστιάζουμε στη δεύτερη εντολή που αφορά τη διακλάδωση.

- Ο PC κατά την εντολή είναι PC=0x8008
- Η PC+08 είναι δύο εντολές κάτω, η εντολή `AND` στη διεύθυνση 0x8010
- Η BTA είναι η εντολή στη διεύθυνση 0x801C, 5 εντολές μετά την PC και 3 εντολές μετά την PC+08
- Άρα το πεδίο imm24 θα έχει την τιμή 3 (αφορά απόσταση σε words)

cond	op	1L	imm24
1110	10	10	000000000000000000000011

Μπορεί να εκτελεστεί και υποσυνθήκη

```
 MOV r0, #4
 ADD r1, r0, r0     ; αποτέλεσμα 8
 CMP r1, r0         ; 1000+1100=0100=4 και 1
                    ; flags C=1
                    ; mnemonics NE, HS, HI, GE, GT
 BEQ EQUAL          ;
 ADDLE r3, r5, r7   ; ΔΕΝ ΕΚΤΕΛΕΙΤΑΙ
 ADDHS              ; εκτελείται
EQUAL
 SUB r3, r4, r5     ; ΔΕΝ ΜΠΑΙΝΕΙ ΚΑΝ ΣΤΟΝ ΚΛΑΔΟ
```

cond	op	1L	imm24
0000	10	10	000000000000000000000001

Και επιτρέπονται και jumps προς τα πίσω εντολές

```
0x8040 TEST LDR r5, [r0, r3]    ; 6 words πίσω (αφαίρεση) από το PC+08
0x8044      STR r5, [r1, r3]
0x8048      ADD r3, r3, #1
0x804C      CMP r3, r5
0x8050      BHS TEST            ; εντολή branch
0x8054      LDR r3, [r1], #4
0x8058      SUB r4, r3, #9      ; PC+08 από εντολή branch
```

cond	op	1L	imm24
0010	10	10	111111111111111111111010

BL(S)

Χρησιμοποιείται για κλήση functions και procedures/subroutines. Διαθέτουν *εισόδους:* ***ορίσματα / arguments*** και *εξόδους:* ***επσιτρεφόμενη τιμή / return value***.

- **caller**: *η συνάρτησή που καλεί την άλλη*

- **callee**: *η συνάρτηση που καλείται*

Η διαδικασία function call χρησιμοποιεί το stack και έχει τα ακόλουθα βήματα:

1. caller βάζει τα arguments στους καταχωρητές r0-r3
2. ο caller αποθηκεύει τη διεύθυνση επιστροφής (PC+04) στον link register (LR/r14) χρησιμοποιώντας την εντολή BL για να μπει στη συνάρτηση, με διακλάδωση *(γίνεται αυτόματα με την κλήση της BL)*
3. η callee έχει χρέος να να προστατεύσει τους καταχωρητές r4-r11 και r14 και γενικά περιοχές μνήμη που χρησιμοποιεί ο caller (π.χ. μπορεί να καλέσει άλλη συνάρτηση)
4. callee βάζει τη return value στον καταχωρητή r0 πριν την ολοκλήρωση

Είναι σημαντικό να θυμάται κανείς ό,τι ισχύει για τις BTA, που εξηγήθηκε στην εντολή B.

Στα παραδείγματα έχω και τις υποθετικές διευθύνσεις μνήμης και στα σχόλια τη σειρά εκτέλεσης

```
0x8004  MAIN
0x8004   MOV r1, #3             ; (1): μπορεί να θεωρηθεί input συνάρτησης
0x8008   BL SIMPLE;             ; (2): LR=0x800C, PC=0x8018
0x800C   MOV r0, #2             ; (4): μπορεί να θεωρηθεί input συνάρτησης
0x8010   BL COMPLICATED         ; (5): LR=0x8014, PC=0x801C
0x8014   ADD r4, r0, r0         ; (9): ο r0 είναι η επιστροφή της complicated = 1
0x8018  SIMPLE MOV PC, LR       ; (3): PC=0x800C
0x801C  COMPLICATED             ; (6)
0x801C   SUB r0, r1, r0         ; (7): r0 είναι ο register επιστροφής
0x8020   MOV r15, r14           ; (8): r15=PC και r14=LR, άρα PC=0x8014
```

- *Η κωδικοποίηση της εντολής* `BL SIMPLE`

cond	op	1L	imm24
0000	10	11	00000000000000000000000010

- *Η κωδικοποίηση της εντολής* `BL COMPLICATED`

cond	op	1L	imm24
0000	10	11	00000000000000000000001

Μερικά παραδείγματα κακής χρήσης που οι registers δεν προστατεύονται από την callee

```
MAIN
 MOV r0, #2             ; (1)
 MOV r2, #4             ; (2)
 MOV r1, r2             ; (3)
 EOR r8, r1, r2         ; (4)
 MVN r4, r8             ; (5)
 BL DANGER              ; (6)
 CMP r4, #0x3F          ; (12): χωρίς την DANGER θα ήταν ίσα, αλλά η DANGER την άλλαξε
 ADDEQ r7, r8, r8       ; ΔΕΝ ΕΚΤΕΛΕΙΤΑΙ
DANGER
 SUB r4, r2, r0         ; (7): η r4 έπρεπε να προστατευτεί!
 ADD r8, r4, r1         ; (8): η r8 έπρεπε να προστατευτεί!
 SUB r3, r4, r8         ; (9)
 MV r0, r3              ; (10): τιμή επιστροφής
 MOV PC, LR             ; (11)
```

Stack

Για να αποφευχθούν τα side-effects των συναρτήσεων (π.χ. το να πειράζουν registers που δεν πρέπει) χρησιμοποείται stack

Το stack είναι ***descending***, δηλαδή μεγαλώνει από μεγαλύτερες διευθύνσεις μνήμης προς μικρότερες, ή από πάνω προς τα κάτω. Το ***top of the stack*** είναι η μικρότερη διεύθυνση μνήμης στο stack και αντιστοιχεί στην πιο πρόσφατα αποθηκευμένη λέξη δεδομένων. Τη διεύθυνση αυτή θα έχει σαν περιέχόμενο ο ***r13 / stack pointer***.

Το offset όμως που χρησιμοποιείται σχέση με το stack pointer είναι θετικό και δείχνει προς τα πάνω

```
DIFFOFSUMS
 SUB SP, SP, #12        ; αφαίρεση 12 bytes από τη διεύθυνση του stack pointer
                        ; χώρος για 3 words στο stack
 STR r9, [SP, #8]       ; 8 bytes πάνω από τον stack pointer - 2 words πάνω
 STR r8, [r13, #4]      ; 4 bytes πάνω από τον stack pointer - 1 word πάνω
 STR r4, [SP]           ; αποθήκευση στον ίδιο τον stack pointert
                        ; δεν πρέπει να πειράξουμε data από [SP, #12]-άλλο stack frame
 ADD r8, r0, r1         ; επιτρέπεται
 ADD r9, r2, r3         ; επιτρέπεται
 SUB r4, r8, r9         ; επιτρέπεται
 MOV r0, r4             ; τιμή επιστροφής

 LDR r4, [SP]
 LDR r8, [SP, #4]
 LDR r9, [r13, #8]      ; επαναφέρμα τις τιμές στους registers
 ADD r13, r13, #12      ; dealocate το stack frame
 MOV r15, r14           ; return to caller
```

Οι καταχωρητές χωρίζονται σε **preserved** και **nonpreserved**.

- οι **preserved** δεν πρέπει να αλλάζουν από συναρτήσεις, οπότε οποιαδήποτε callee θέλει να τους χρησιμοποιήσει πρέπει να τους σώσει στο stack. Εξ ου και λέγονται ***callee-save***
- οι **nonpreserved** είναι ελεύθεροι να αλλάζουν από οποιαδήποτε συνάρτηση, οπότε οποιαδήποτε caller τις χρησιμοποιεί πριν κλήση συνάρτησης πρέπει να τις αποθηκεύει στο stack, εξ ου και λέγονται ***caller-save***

Preserved	**Nonpreserved**
Saved registers: r4 - r11	Temporary register: r12
Stack pointer: r13	Argument registers: r0 - r3
Return address: r14	CPSR
Stack above stack pointer	Stack below stack pointer

Να σημειώσω ότι το stack πάνω από τον SP θεωρείται χώρο μέσα στο stack, ενώ από κάτω είναι εκτός του stack

Machine Language Intricacies

Μετατροπή σταθερών σε 32 bit

Σταθερές των 8 bit (πχ εντός των εντολών, όπως οι imm8 μπορούν να μετατραπύν σε 32 bit. Συνήθως είναι μέσα σε δύο πεδία εντολών: το imm8 που είναι η εντολή και rot που είναι μία μικρη σταθερά 4 bit

imm32 = imm8 right_shift by (2 x rot)

Όσο κάνει αυτή η περιστροφή στα δεξιά μπορεί να υπολογιστεί και αντίστοιχη στροφή στα αριστερά για να το σκέφτεσαι πιο εύκολα:

left rotation = 32 - right rotation

Εντολές Branch και PC+08 διευθυνσιοδότηση

Στις εντολές B και BL στο πεδίο imm24 αποθηκεύεται με έναν ιδιαίτερο τρόπο η **Branch Target Address / BTA**. Όταν καλείται η εντολή branch, αν πρόκειται να εκτελλεστεί και να κάνει *jump* / *branch-out* τότε υπολογίζεται η BTA σε σχέση με τη διεύθυνση PC+08, δηλαδή την απόσταση της BTA σε σχέση με την εντολή που βρίσκεται δύο θέσεις μετά από την εντολή branch.

Συγκεκριμένα μέσα στο πεδίο αποθηκεύεται απόσταση, δηλαδή words. Για να υλοποιηθεί το jump και να βρεθεί εν τέλει η BTA **πρέπει η απόσταση/words να μετατραπεί σε μέγεθος/bytes** και μετά να προστεθεί στο PC+08. Για να γίνει αυτό **πραγματοποιείται sign-extension και πολλαπλασιασμό με 4, που υλοποιείται ως shift-left κατά 2**.

Τελικά για να υπολογιστεί η BTA από το imm24:

BTA = PC + 8 + (imm24 x 4)

Ανάγνωση του PC

Για ιστορικούς λόγους η ανάγνωση / read του PC / r15 επιστρέφει την διεύθυνση της εντολής που εκτελλείται εκείνη τη στιγμή συν 8