

1. INSTRUKTIONSOVERSIKT.....	2
1.1. INSTRUKTIONSFORMAT	3
1.1. IMMEDIATEOPERANDENS FORMAT	3
1.2. BEGRÄNSNINGAR HOS OPERAND 2	3
1.3. BEGRÄNSNINGAR HOS ADRESSER	3
1.2. PSEUDOINSTRUKTIONER	4
1.3. ADDRESSERINGSMODE	4
2. REGISTERUPPSÄTTNING.....	5
2.1. BANKADE REGISTER	5
2.2. STATUSREGISTER	5
3. SUFFIX (ARM). VILLKORSSTYRNING.....	6
4. SÄTTA STATUSFLAGGOR.....	6
5. SKIFTOOPERATIONER.....	7
5.1. SKIFTOOPERATIONEN LSL	7
5.2. SKIFTOOPERATIONERNA LSR OCH ASR	7
6. LOGISKA/ARITMETISKA INSTRUKTIONER.....	8
6.1. JÄMFÖRA CMP, CMN, TEQ, TST	8
6.2. LOGISKA OPERATIONER AND, ORR, EOR, BIC	8
6.3. INVERTERING. LOGISKT NOT (MVN).....	8
6.4. ARITMETISKA OPERATIONER ADD, ADC, SUB, SBC, RSB,RSC.....	8
6.5. MULTIPLIKATION	9
6.6. MUL, MLA	9
6.7. MULTIPLIKATION MED STORA TAL: MULL, MLAL.....	9
7. KONTROLLINSTRUKTIONER.....	10
7.1. ABSOLUTA OCH VILLKORSSTYRDA HOPP (FÖRGRENINGAR): B ("BRANCH").....	10
7.2. ABSOLUTA OCH VILLKORSSTYRDA HOPP TILL SUBROUTINER: BL ("BRANCH WITH LINK").....	10
7.3. STACKHANTERING - SKYDDA REGISTER.....	11
8. DATAFÖRFLYTTNING (DATAKOPIERING):.....	12
8.1. FLYTTA IN VÄRDE I ETT REGISTER.....	12
8.2. FÖRFLYTTNING MELLAN REGISTER MOV, MVN	12
8.3. LÄSNING/SKRIVNING TILL STATUSREGISTER MSR, MRS	12
8.4. STORLEKSSUFFIXEN B OCH H PÅ LDR OCH STR.....	12
8.5. LOAD LDR, STORE STR.....	13
8.6. FLYTTNING (KOPIERING) AV DATABLOCK: LDM, STM	15
9. ASSEMBLERDIREKTIV.....	16
9.1. EQUAL (EQU).....	16
9.2. ORIGIN (ORG)	16
9.3. AVSLUT (END).....	16
9.4. DEFINE CONSTANT (DC).....	16
9.5. DEFINE STORAGE (DS).....	16
9.6. ALIGN	16
9.7. KOMMENTARER.....	17

1. Instruktionsöversikt

Mnemonic	Instruction	Action
ADC	Add with carry	$Rd = Rn + Op2 + Carry$
ADD	Add	$Rd = Rn + Op2$
AND	AND	$Rd = Rn \text{ AND } Op2$
B	Branch	$R15 = \text{address}$
BIC	Bit Clear	$Rd = Rn \text{ AND NOT } Op2$
BL	Branch with Link	$R14 = R15, R15 = \text{address}$
BX	Branch and Exchange	$R15 = Rn, T \text{ bit} = Rn[0]$
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags: $= Rn + Op2$
CMP	Compare	CPSR flags: $= Rn - Op2$
EOR	Exclusive OR	$Rd = (Rn \text{ AND NOT } Op2) \text{ OR } (Op2 \text{ AND NOT } Rn)$
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	$Rd = (\text{address})$
MCR	Move CPU register to coprocessor register	$cRn = rRn \{<op>cRm\}$
MLA	Multiply Accumulate	$Rd = (Rm \times Rs) + Rn$
MOV	Move register or constant	$Rd = Op2$
MRC	Move from coprocessor register to CPU register	$Rd = cRn \{<op>cRm\}$
MRS	Move PSR status/flags to register	$Rd = PSR$
MSR	Move register to PSR status/flags	$PSR = Rm$
MUL	Multiply	$Rd = Rm \times Rs$
MVN	Move negative register	$Rd = \text{Not } Op2$
ORR	OR	$Rd = Rn \text{ OR } Op2$
RSB	Reverse Subtract	$Rd = Op2 - Rn$
RSC	Reverse Subtract with Carry	$Rd = Op2 - Rn - \text{Not Carry Flag}$
SBC	Subtract with Carry	$Rd = Rn - Op2 - \text{Not Carry Flag}$
STC	Store coprocessor register to memory	$\text{address} = cRn$
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	$<\text{address}> = Rd$
SUB	Subtract	$Rd = Rn - Op2$
SWI	Software Interrupt	OS call
SWP	Swap register with memory	$Rd = [Rn], [Rn] := Rm$
TEQ	Test bitwise equality	CPSR flags: $= Rn \text{ EOR } Op2$
TST	Test bits	CPSR flags: $= Rn \text{ AND } Op2$

1.1. Instruktionsformat

Instruktion $Rd, Rn, Op2$

- Rd "Register destiny", dvs det register där resultatet hamnar.
 Rn "Register with arbitrary number", dvs godtyckligt register, dock oftast inte R15 (dvs PC).
 Op2 "Operand - two types" är en mycket komplex operand och kan delas upp i följande fall:
- Direkt värde ("immediate vale") – ett 32 bitar värde som skapas genom att vänsterrottera ett 8-bitars värde ett antal steg:
 MOV R3, #0xFF0000 ; lagrar FF samt information om skiftning
 MOV R3, #4
 - Skift- och rotationsoperationer utförda på ett register (Rm). Antalet skift/rotationer anges direkt med ett tal som får uppta max 5 bitar (dvs ett tal 0 - 32)
 ADD R2, R4, Rm, LSL #4
 - Ett godtyckligt register: Rm
 ADD R4, R3, Rm
 - Skift- och rotationsoperationer utförda på på ett register (Rm). Antalet skift/rotationer finns lagrat i ett annat register (Rs):
 ADD R2, R4, Rm, LSL Rs

1.1. Immediateoperandens format

Immediateoperanden kan skrivas in i tre olika talformat

Decimalt #45 #-34 (även negativ inskrivning)
 Hexdecimalt #0x2F
 Binärt #0010010b

1.2. Begränsningar hos operand 2

Operand 2 kan vara en av följande:

- Godtyckligt register
- 8-bitarsvärde. Maximalt tal = 255

12-bitar är avsatta till datavärde och manipulation av värdet. "Manipulationsoperatorn" (skift, rotation) måste också inrymmas i 12-bitars värdet. Följande exempel visar på mekanismen.

Då MOV används kan talet max vara 255 i normalfall och man kan inte skriva in negativa tal i decimal form!

```
MOV R0, #0x123 ; Assemblatorn ger felrapport. 0x123 är ett 12-bitars
                ; värde men kan inte skapas med vänsterrottera.
MOV R0, #0x120 ; Assemblatorn kan skapa detta 12-bitars värde m h a
                ; vänsterrotation.
MOV R0, #0x12000000 ; Assemblatorn kan skapa detta 32-bitars värde m h a
                ; vänsterrotation.
```

Då instruktioner utför beräkningar används kan talet vara från -255 till 255

```
CMP R3, #45 ; talet man jämför mot är begränsat.
ADD R2, R2, #-255
```

1.3. Begränsningar hos adresser

Hos instruktioner som tar en adress som operand finns också begränsningar då man inte kan få plats med en hel adress i instruktionen.

Nedanstående översätts till LDR R3,[PC,#X], där X blir en offset från nuvarande position till adressen man vill läsa från. X är det tal som verkligen lagras i minnet. X kan vara mellan -4095 till 4095

POS LDR R3, ADRESS ; skillnad mellan POS och ADRESS lagras

När det gäller *BRANCH*-instruktioner av olika slag lagras offseten från nuvarande PC. Längden på offseten blir något större än för *LDR*-fallet, då man inte behöver lagra något destinationsregister.

POS B ADRESS ; skillnad mellan POS och ADRESS lagras

1.2. Pseudoinstruktioner

Trots uppenbara begränsningar i immediatevärdet kan man ladda in 32-bitars tal i register. Detta sker med sk. pseudoinstruktioner. Det är inte riktiga instruktioner, utan ett mellanting mellan assemblerdirektiv och instruktioner. Pseudoinstruktionen översätts till riktiga instruktioner av assemblern. En pseudoinstruktionen kan alltså bli mer än en riktig instruktion

Med tillägget '=' kan man säga att *LDR* blir en pseudoinstruktion.

LDR R2, =TAL ; tal kan vara 32-bitar stor

Översätts till:

MOV R2, #tal ; om talet är litet (mindre än 8-bitar)

LDR R2, [PC, #offset] ; om talet är större än 8-bitar

Själva talet man vill lagra läggs ut på minnet av assemblern. Vanligtvis läggs det efter själva koden. Offseten som assemblern räknar ut blir avståndet från nuvarande programräknare och var talet lagras.

Även då *LDR* eller *STR* använder direkt adresseringsmode översätts den till en annan.

LDR R2, ADRESS ; Läs från adressen ADRESS och lägg i R2

Översätts till:

LDR R2, [PC, #offset]

Där offsetvärdet är skillnaden mellan instruktionens adress och minnesplatsen ADRESS. Värdet beräknas av assemblern.

Instruktionen *ADR* som används för att läsa in en adress i ett register är också en pseudoinstruktion. Fungerar på liknande sätt som *LDR*

ADR R4, ADRESS ; Liknande sätt som LDR Rn, =tal

1.3. Adresseringsmode

De olika sätt att nå data brukar med ett gemensamt ord kallas *adresseringsmode*. Antalet adresseringsmode säger i princip på hur många olika sätta man kan flytta in information i registren. Ofta skrivs instruktionens operander på olika sätt beroende på vilket adresseringsmode som nyttjas. Dessa gäller för ARM. Alla instruktioner stödjer inte alla adresseringsmode.

<u>ADRESSERINGMODE</u>	<u>EXEMPEL</u>	<u>FÖRKLARING</u>
Register	ADD R1, R2, R3	Alla data i register
Immediate	MOV R3, #34	Data är ett värde man skrivit in omedelbart i instruktionen
Base + index (offset)	LDR R0, [R5, #8] LDR R3, [R4] LDR R2, [R3, R3]	Data finns på en adress. Adressen till data finns i ett register som adderas med ett konstant tal
Auto - index	LDR R0, [R5, #8]! LDR R0, [R5], #8 LDR R0, [R5], R2	Liknande föregående, men ökar registret som innehåller adressen. Bra vid tabeller
Direct	LDR R4, PLACE	Läs data från den adress som utgörs av labeln.

2. Registeruppsättning

ARM State General Registers and Program Counter

System & 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_irq	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)

Stackpekare
 ("Stack pointer"):
 R13 = SP
 (Ej standard men brukligt.)

Länkregister
 ("Subroutine link register"):
 R14 = LR

Programpekare
 ("Program Counter"):
 R15 = PC

Statusregister
 ("Program Status Register"):
 R16 = CPSR
 Innehåller N-,Z-,C-,V-flaggor och "mode"-bitar

ARM State Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

▲ = banked register

2.1. Bankade register

Arm-processorn arbetar i olika *mode*. Beroende på vilket mode man är i har man lite olika rättigheter att ändra olika register. Dom olika moden har även olika registeruppsättningar, dvs är man supervisor har man en uppsättning, är man user en annan. Detta kallas att man har *bankade register*. Detta berör endast de markerade registren ovan.

På det här varje mode bland annat egna statusregister (CPSR), länkregister (LR) och stackpekare (SP).

2.2. Statusregister

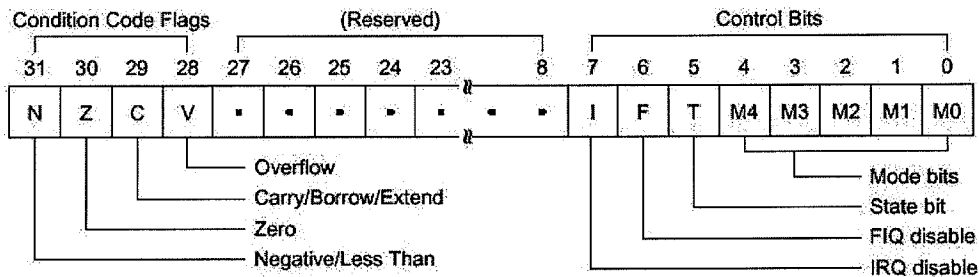


Figure 2-6. Program Status Register Format

Dom lägsta 5 bitarna M4-M0 i CPSR avgör vilket mode man är i.

10000 User	10111 Abort
10001 FIQ	11011 Undefined
10010 IRQ	11111 System
10011 Supervisor	

3. Suffix (ARM). Villkorsstyrning

Alla instruktioner på ARM går att utföra villkorsstyrt genom att lägga till suffix på instruktionerna.

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Z "Zero"-flaggan: om resultat av föregående instruktion = 0 \Rightarrow Z = 1

C "Carry"-flaggan: om resultat av föregående instruktion ger "carry" \Rightarrow C = 1

N "Negative"-flaggan: om resultatet av föregående instruktion ger ett värde med MSB = 1 \Rightarrow N = 1

V "Overflow"-flaggan: om resultatet av föregående instruktion ger "overflow" \Rightarrow V = 1

Exempel på villkorligt utförda instruktioner.

```
ADDEQ    R1,R2,#45 ;R1=R2+45 om Z=1
BEQ     TEST      ; hopp till TEST om Z=1
```

Flervalskonstruktion

```
CMP     R1,R2      ; testa R1 mot R2
MOVEQ   R3,R1      ; om R1=R2
ADDGT   R3,R1,#1   ; om R1>R2
ADDLT   R3,R1,#4   ; om R1<R2
```

4. Sätta statusflaggor

Alla instruktioner på ARM kan sätta statusflaggorna med tillägget av S efter instruktionen. Flaggor sätts efter talet som hamnar i destinationsoperanden.

Sätter statusflaggor samtidigt som addition utförs

```
ADDS    R2,R1,#1   ; Sätter statusflaggor beroende på värdet i R2
```

Villkor utfört utan compare (CMP)

```
SUBS    R1,R1,#1   ; R1=R1-1, testa på R1
MOVEQ   R3,R1      ; om R1=0, annars hoppa över
```

5. Skiftoperationer

Alla instruktioner som i listan med instruktioner kan ha operand av typen 2 (dvs Op2) kan också samtidigt använda olika typer av skiftinstruktioner i samma instruktion. Detta är en smart grej, som ARM-processorn är relativt ensam om. Man kan se skiftoperationerna som tillägsoperatörer, som alltid opererar på den sista operanden.

ASL	= Aritmetiskt vänsterskift ("Arithmetic Shift Left")
LSL	= Logiskt vänsterskift ("Logic Shift Left") = ASL
ASR	= Aritmetiskt högerskift ("Arithmetic Shift Right")
LSR	= Logiskt högerskift ("Logic Shift Right")
ROR	= Logisk högerrotation ("Rotate Right")

Med hjälp av dessa skiftoperationer kan en hel del smarta tidsbesparande operationer utföras. Här visas de två vanligaste LSL och LSR

Biten som skiftas ut till vänster hamnar i Carry-flaggan.

5.1. Skiftoperationen LSL

Med hjälp av LSL kan ett värde enkelt multipliceras med 2^n (där $n = 0, 1, \dots, 30, 31$).

For example, the effect of LSL #5 is shown in Figure 3-6.

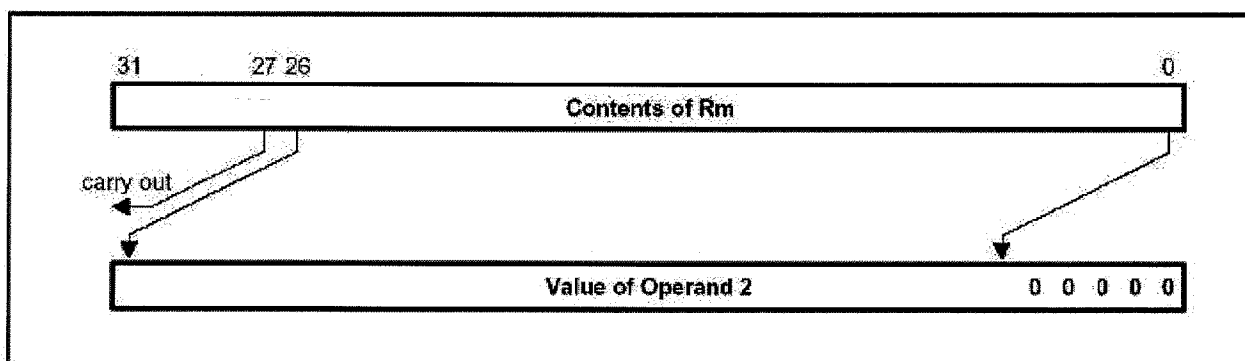


Figure 3-6. Logical Shift Left

Vänsterskift kan användas för att tillverka ett värde som upptar mer än de 8 bitars datakraft som finns tillhanda:

```
MOV    R1, #8-bit_val      ; R1= 8-bitars värde
MOV    R1, R2, LSL #5bit_val ; R1= R2 x 25bit_val (max 231)
```

Vänsterskift kan användas för att placera en "etta" i rätt position:

```
MOV    R1, #1              ; R1= "1" i bit 0
MOV    R2, R1, LSL #7     ; R2= "1" i bit 7
```

Dynamiskt vänsterskift. Antal steg man ska skifta ligger i ett register:

```
ADD    R1, R2, R4, LSL Rn ; R4 vänsterskiftas enligt värde i Rn, efter
                          ; det utförs additionen
```

5.2. Skiftoperationerna LSR och ASR

Med hjälp av LSR kan ett positivt värde enkelt divideras med 2^n (där $n = 0, 1, \dots, 30, 31$). Om talet ska betraktas som ett tal med tecken ska aritmetiskt högerskift användas istället. Ovanstående exempel är också relevanta för LSR.

6. Logiska/Aritmetiska instruktioner

6.1. Jämföra CMP, CMN, TEQ, TST

Instruktionerna CMP, CMN, TEQ, och TST är "oförstörande", dvs de producerar inga synliga resultat utan används för att testa innehållet i ett register. Några exempel:

```
CMP    Rn, Rm          ; Jämför Rn med Rm. Utför internt beräkningen Rn - Rm
                          ; och sätter därefter flaggorna
CMP    Rn, Op2         ; Jämför Rn med Op2. Op2 mellan -255 till 255
TEQ    Rn, Rm          ; Rn BITVIS XOR Rm ?. Påverkar Z-flaggan.
TST    Rn, Op2         ; Rn BITVIS OCH Op2 ? Påverkar Z-flaggan.
```

; Rn och Rm är två godtyckliga register.

6.2. Logiska operationer AND, ORR, EOR, BIC

De logiska instruktionerna AND, ORR (eller), EOR (Exklusiv eller) och BIC utför bitoperationer på alla bitar i operanden eller operanderna. Några exempel: (EOR och ORR används på liknande sätt som AND)

```
AND    Rd, Rn, Rm      ; Rd:= Rn AND Rm
AND    Rd, Rn, Op2     ; Rd:= Rn AND Op2. Op2 mellan -255 till 255
BIC    Rd, Rn, Rm      ; Rd:= Rn AND (NOT Rm)
BIC    Rd, Rn, Op2     ; Rd:= Rn AND (NOT Op2)
```

Om villkorsflaggorna ska aktiveras måste suffixet S läggas till, exempelvis.

wait_for_zero

```
SUBS   Rd, Rm, Rn      ; Rd:= Rm - Rn
BNE    wait_for_zero   ; Om Rd <> 0 så hoppa, annars fortsätt
```

Om suffixet S inte används måste en "compare"-instruktion användas:

wait_for_zero

```
SUB    Rd, Rm, Rn      ; Rd:= Rm - Rn
CMP    Rd, #0          ; Jämför Rd med värdet 0
BNE    wait_for_zero   ; Om Rd <> 0 så hoppa, annars fortsätt
```

6.3. Invertering. Logiskt NOT (MVN)

ARM har ingen specifik instruktion för NOT utan man får använda registerflyttningsinstruktionen MOV. Instruktionen går att utföra inverterat och heter då MVN.

Invertering av ett register:

```
MVN    R1, R1          ; R1 inverteras. R1= NOT R1
```

Invertering av ett värde:

```
MVN    Rd, Op2        ; Rd:= NOT Pp2. Op2 mellan 0-255
```

6.4. Aritmetiska operationer ADD, ADC, SUB, SBC, RSB, RSC

Instruktionerna ADD, ADC, SUB, SUBC, RSB, RSC, utför addition och subtraktion. Några exempel:

```
ADD    Rd, Rn, Rm      ; Rd:= Rn + Rm
ADD    Rd, Rn, Op2     ; Rd:= Rn + Op2
```

6.5. Multiplikation

Det finns tre grupper av instruktioner vid multiplikation av heltal

- Vanlig multiplikation. Här förutsätts att resultatet får plats i 32-bitar
- Multiplikation Long. Går att arbeta med stora tal genom att nyttja fler register.
- Multiplikation med ackumulation. Vanligt vid vektoroperationer samt digitala filter

6.6. MUL, MLA

Vanlig multiplikation:

```
MUL    Rd, Rs, Rm           ; Rd:= Rm x Rs
```

Multiplikation med accumulation. Vanligt vid vektoroperationer:

```
MLA    Rd, Rn, Rs, Rm       ; Rd:= Rm x Rs + Rn
```

Faktiskt exempel (-10) x 20 = -200:

```
LDR    R0, =0xFFFFFFFF6    ; -10
MOV    R1, #20              ; 20
MUL    R2, R0, R1          ; 0xFFFFFFFF38 (-200) i R2
```

Faktiskt exempel 0xFFFFFFFF6 x 20 = 0x13FFFFFFF38:

```
LDR    R0, =0xFFFFFFFF6    ; 4294967286
MOV    R1, #20              ; 20
MUL    R2, R0, R1          ; 0xFFFFFFFF38 (-200) i R2
                               ; 0x13 blir "overflow"
```

6.7. Multiplikation med stora tal: MULL, MLAL

Eftersom multiplikation mellan 32-bitars tal kan ge upphov till produkter som inte ryms i ett 32-bitars register finns speciella instruktioner för detta (MULL, MLAL). Suffixet L betyder "Long" och syftar just på att 64 bitar är "längre" än 32 bitar.

Finns i två uppsättningar – en för positiva tal och en för tal med tecken. Detta fås med prefixet U eller S:

Generella exempel:

```
UMLA   RdLo, RdHi, Rn, Rs, Rm ; RdHiLo:= Rm x Rs + Rn
UMUL   RdLo, RdHi, Rs, Rm     ; RdHiLo:= Rm x Rs
```

Faktiskt exempel 0xFFFFFFFF6 x 20 = 0x13FFFFFFF38:

```
LDR    R0, =0xFFFFFFFF6    ; 4294967286
MOV    R1, #20              ; 20
UMULL  R2, R3, R0, R1       ; 0xFFFFFFFF38 (-200) i R2
                               ; 0x13 hamnar i R3
```

7. Kontrollinstruktioner

7.1. Absoluta och villkorsstyrda hopp (förgreningar): B ("Branch")

Ett program fortsätter automatiskt mot högre adresser. Ibland måste en programdel hoppas förbi ("förbihopp") och ibland måste en annan del repeteras ("återhopp").

```
B      label          ; Hoppa alltid ("Branch Always") till adressen där
                        ; etiketten ("label") finns.
```

B har egentligen suffixet AL, vilket inte behöver skrivas ut. Hoppet kan villkorsstyras m h a suffix enligt tabellen. Här följer två vanliga exempel på suffix (tillägg):

```
BEQ    label          ; Hoppa om föregående resultat är noll, dvs Z = 1
BNE    label          ; Hoppa om föregående resultat är skilt från noll,
                        ; dvs Z <> 0.
```

En del suffix styr hur ett aritmetiskt resultat ska tolkas. Här följer två vanliga exempel på suffix:

```
CMP    Rn,Rm          ; Jämför Rn med Rm
BHI    res_hi         ; Positiva heltal. Om Rn > Rm så hoppa till res_hi.
BGT    res_gt         ; Pos/neg. tal. Om Rn > Rm så hoppa till res_gt.
```

Suffixet HS ("Higher or Same") finns inte med på listan med suffix men fungerar också.

7.2. Absoluta och villkorsstyrda hopp till subrutiner: BL ("Branch with Link")

Subrutiner används för att huvudprogrammet ska bli överskådligt men också för att en del programdelar är generella och återkommer i många program.

```
BL     Sub_label      ; Hoppa alltid till subrutin. Sub_label är exempel på
                        ; en label! Återhoppadress sparas alltid i LR.
```

Sub_label

```
      ; Här följer programkoden i subrutinen.
      ; Återhopp genom att placera LR i PC.
MOV    PC,LR
```

Hoppet till subrutiner kan också villkorsstyras m h a suffix enligt tabellen? Här följer två vanliga exempel på suffix:

```
BLEQ   Sub_label      ; Hoppa till subrutin om föregående resultat är noll.
BLNE   Sub_label      ; Hoppa till subrutin om föregående resultat inte är
                        ; noll.
```

7.3. Stackhantering - skydda register

I detta exemplet utgås från att stacken är initerad på höga adresser och ska därmed växa neråt (mot lägre adresser). Viktigt att spara undan *arbetsregister* samt *länkregister* (för att slippa problem med nästlade subrutinanrop).

För detaljerad information om dataflyttningsinstruktionerna STMFD och LDMFD, se kapitel 8.6.

```
; SUBRUTIN Delay_ms
; Anropas med värde i r0, vilket ska ge motsvarande fördröjning i millisekunder.
; -----
DELAY_CALIB    EQU    100           ; Innerslingan som ska ta en millisekund

Delay_ms       STMFD  SP!, {R0-R1, LR}      ; Spara undan R0, R1 och LR på stacken
do_delay_ms    LDR    R1, =DELAY_CALIB      ; Ladda "fördröjningsvärde".
loop_ms        SUBS   R1, R1, #1
               BNE   loop_ms
               SUBS   R0, R0, #1           ; Minska millisekundräknaren
               BNE   do_delay_ms
               LDMFD  SP!, {R0-R1, PC}     ; Återställer innehållet i R0 och R1.
               ; Läger in LR till PC samtidigt.
; -----
```

8. Dataförflyttning (datakopiering):

På ARM finns två huvudgrupper av dataförflyttning.

- Mellan register. Detta sker internt i processorn. (MOV)
- Mellan minne och register. Detta kräver access till minnet (LDR, STR)

Följande är möjligt att göra med dessa instruktioner

- Kopiera värde mellan register
- Flytta in värde (immediate) i register
- Läsa från minnesadresser till register
- Skriva från register till minnesadresser

Det är även möjligt att skriva från flera register till minnet, dvs blockförflyttning av data.

8.1. Flytta in värde i ett register

Detta kan göras med två instruktioner. Använder man MOV-instruktionen ligger verkligen datat lagrat som en del av instruktionen. Talet är då begränsat till 8-bitar OM det inte slutar med nollor (se kapitel 1.2). Endast positiva tal går att skriva in då man anger talet decimalt.

Vill man lägga in större tal kan man ta till psudoinstruktionen LDR som egentligen flyttar från minne till register (se kapitel 1.2)

8.2. Förflyttning mellan register MOV, MVN

Instruktionerna MOV och MVN används för att kopiera mellan register eller för att lägga in "små tal" i register. Några vanliga exempel:

```
MOV    R1,R2          ; R1 får R2:s värde
MOV    R2,#45         ; R2=45
MOV    R3,#45         ; R3=45
MVN    R1,R1          ; R1= NOT R1
```

; Talet kan maximalt vara 8-bitar.

8.3. Läsning/skrivning till statusregister MSR, MRS

För att läsa av och skriva till statusregistret CPSR finns två speciella instruktioner.

```
MRS    R1, CPSR      ; läser av CPSR och lägger i R1
MSR    R1, CPSR      ; skriver R1:s värde till CPSR
```

8.4. Storlekssuffixen B och H på LDR och STR

Genom att lägga till suffixen B eller H till instruktionerna LDR och STR kan man bestämma storleken på läsning eller skrivning. Ofta väldigt viktigt vid arbete med specialregister (SFR). Felaktig skrivning (och läsning) resulterar i odefinierat resultat

- B = "Byte" (8 bitar)
- H = "Halfword" (16 bitar)
- Inget suffix. Betyder 32-bitar

Skrivning till Port 0:s kontrollregister (PCON0) på 11 bitar. Anta adress till PCON0 i R2.

```
STRh   R1,[R2]      ; skriver 16-bitar av R1 till adressen i R2
```

Avläsning av Port 0:s dataregister (PDAT0) som är 1 byte. Anta adress till PDAT0 i R2.

```
LDRb   R1,[R2]      ; Läser av PDAT0 till R1
```

8.5. Load LDR, Store STR

LDR ("Load from memory into Register")

ladda register.. Läsning från minnet till ett register. Kan även användas som psuedoinstruktion för att läsa in stora tal i register.

STR ("Store from Register into memory")

lagra register. Skrivning från ett register till minnet

Här följer några exempel som täcker de vanligaste fallen rörande **LDR** och **STR**

Ladda ett register med ett värde. Generella exempel:

```
LDR    Rd,=32bit_val      ; Rd tilldelas 32-bitars värde
```

Ladda ett register med ett värde som ligger på en adress. Generella exempel:

```
LDR    Rd,ADDRESS        ; Rd tilldelas det värde som ligger på ADDRESS
STR    Rd,ADDRESS        ; Rd:s värde hamnar på ADDRESS
```

Base register adressering. Ladda ett register med ett värde, som finns på en adress. Generella exempel:

```
LDR    Rn,=32bit_val     ; Rn tilldelas 32-bitars värde, som
                          ; ska bli ett adressvärde.

LDR    Rd, [Rn]          ; Rd:= Innehållet på adress i Rn
LDRb   Rd, [Rn]          ; Rd:= Innehållet på adress i Rn.
                          ; Relevant om innehållet på adressen är ett
                          ; 8-bitars värde, exv en 8-bitars port.

STR    Rd, [Rn]          ; Innehållet på adress enligt Rn tilldelas
                          ; värdet i Rd, dvs Rd -> M(Rn)

STRh   Rd, [Rn]          ; 16 bitar av Rd lagras på adress enligt Rn.
                          ; Relevant för exv 16-bitars port.
```

Base + index (offset) adressering. Ladda ett register med ett värde, som finns på en adress + förskjutning. Generella exempel:

```
LDR    Rn,=Array        ; Rn tilldelas en 32-bitars adress till
                          ; första elementet i en lista.
                          ; Listan måste finnas i minnet.

LDR    Rd, [Rn,#2]      ; Rd:= element från adressen [Rn + 2].
                          ; ex Rn=0x00001000. Då läser man från adress
                          ; 0x00001000+2 = 0x00001002

LDR    Rd, [Rn,Rm]      ; Rd:= element från adressen [Rn + Rm]
STRh   Rd, [Rn,#2]      ; 16 bitar av Rd till adressen [Rn+2]
STR    Rd, [Rn,Rm]      ; Rd lagras på adressen [Rn + Rm]
```

; Offset räknas alltid i byte

Auto + index adressering och "write back". Ladda ett register med ett värde, som finns på en adress + förskjutning. Automatisk uppdatering av offset.

Uppdatering av basregister Rn sker före skrivning eller läsning. Kräver utropstecken!

LDR Rd, [Rn, #4]! ; Rn:= Rn + 4, Rd:= data från adress Rn.

STR Rd, [Rn, #4]! ; Rn:= Rn + 4, Rd skrivs till adress i Rn.

Uppdatering av basregister Rn sker efter skrivning eller läsning. Inget utropstecken!

LDR Rd, [Rn], #4 ; Rd:= data på adress Rn. Rn:= Rn + 4

STR Rd, [Rn], #4 ; Rd skrivs till adress i Rn. Rn:= Rn + 4

Skiftoperationer kan användas tillsammans med LDR men skiftvärdet måste ges explicit (dvs inte av ett register).

8.6. Flyttning (kopiering) av datablock: LDM, STM

Förflyttning av datablock kan utföras med hjälp av instruktionerna LDM ("Load Multiple Registers") och STM ("Store Multiple Registers"). Instruktionen LDM laddar in ett datablock från minnet och instruktionen STM sparar ett antal register i minnet. Instruktionerna är sinnrikt uppbyggda och kan också användas för att utföra s.k. stackoperationer. Begreppet stack förklaras inte här.

Registerlistan

Vare sig datatrafiken går från minnet till registren eller från registren till minnet måste de register som är aktiva i datatrafiken anges på ett speciellt sätt. Register placeras i en lista och listan markeras med "krullparenteser" på följande sätt:

Ex på registerlistor:

- {R2, R3, R4} Register R2, R3 och R4 är delaktiga i dataöverföring.
- {R2 - R4} Register kan anges som intervall. Likvärdigt med ovanstående lista.
- {R2 - R4, LR} Intervall och enstaka register kan kombineras. I detta exempel finns dessutom länkregistret med.

Baspekare ("Base register")

Ett register måste alltid användas för att peka på den adress där blocket finns. Vanligtvis används register R0 till R12. Vid stackhantering används oftast register R13 (= SP). Det finns fyra uppsättningar LDM-STM-instruktioner.

Name	Stack	Other
Pre-Increment Load	LDMED	LDMIB
Post-Increment Load	LDMFD	LDMIA
Pre-Decrement Load	LDMEA	LDMDB
Post-Decrement Load	LDMFA	LMDA
Pre-Increment Store	STMFA	STMIB
Post-Increment Store	STMEA	STMIA
Pre-Decrement Store	STMFD	STMDB
Post-Decrement Store	STMED	STMDA

Other = Datablock

Pilarna visar två vanliga kombinationer vid stackhantering och kopiering av datablock. Notera att LDMFD och LDMIA är olika namn på samma instruktion.

Suffixen ED, FD, EA FA används vid stackoperationer och är likvärdiga med suffixen IB, IA, DB, DA - som används vid datakopiering. De olika suffixen kodas till samma instruktion men att visa hur instruktionerna ska paras ihop på rätt sätt.

Datakopiering

Vid datakopiering behövs två baspekare - en som pekare på källan ("source") och en som pekar på slutstationen ("destination"). I nedanstående exempel flyttas 8 stycken "word" (dvs 32-bitars tal).

; Exempel på datakopiering.

; Data flyttas från adress source_pnt till adress dest_pnt

```
LDR    R0,=source_pnt      ; R0 = pekare (adress) till datakällan
LDR    R1,=dest_pnt       ; R1 = pekare (adress) till destination
LDMIA R0,{R4-R11}        ; Ladda 8 ord ("words") från källan
STMIA R1,{R4-R11}        ; och placera på destinationen
```

Stackhantering

Det är viktigt att välja passande skriv och läsinstruktion så verkligen det sista man skrivit blir det första man läser. För att göra livet enklare för assemblerprogrammerare har passande instruktioner samma ändelse. Här används instruktionerna LDMFD och STMFD som utgör ett naturligt par vid stackhantering. Vi antar här att stackpekaren SP är satt till lämpligt värde. Eftersom SP flyttas neråt vid skrivning med STMFD är stacken antagligen definierad på en hög adress i minnet.

Stackexempel

```
PUSH      STMFD  SP!,{R0-R1,LR}      ; Spara undan R0-R1 och LR på stacken
POP       LDMFD  SP!,{R0-R1,PC}     ; Återställ innehållet i R0-R1.
                                                ; Läger in LR till PC samtidigt.
```

Utropstecknet efter registret betyder att innehållet i registret automatiskt ändras (uppdateras). Det är en s.k. "write back operation".

9. ASSEMBLERDIREKTIV

Vid programmeringen behövs ett antal direktiv (styrningar), som behövs för att assemblern ska kunna generera den maskinkod som processor "förstår". Här följer en lista på de vanligaste direktiven vid "enkel" assembly-programmering.

9.1. Equal (EQU)

Direktivet EQU innebär att en textsträng ersätts av en annan likvärdig (ekvivalent) textsträng. Exempel:

```
ROM_START    EQU    0x01FF0000        ; Programmet börjar på denna adress
RAM_END      EQU    0x01FFFFFF
STACK       EQU    RAM_END+1
```

9.2. Origin (ORG)

Direktivet ORG betyder att adresserna styrs till detta värde. Exempel, fortsättning föregående:

```
        ORG    ROM_START
main    ; Etiketten main får automatiskt adressen 0x01ff0000
```

9.3. Avslut (END)

Direktivet END placeras alltid sist i källfilen. Alla instruktionerna efter detta direktiv kommer inte att tolkas av assemblern.

9.4. Define Constant (DC)

Dessa direktiv används för att lagra (konstant) information i programmet. Det rör sig oftast tabelldata och textsträngar

Exempel på inlagda konstanter:

```
TAL1      DC32    0x3243DFAA    ; istället för DC32 kan DCD användas
TAL2      DC16    0xFF32        ; istället för DC16 kan DCW användas
TAL3      DC8     32            ; istället för DC8 kan DCB användas
```

Exempel på tabelldata:

```
BAUD_RATES DC32    9600, 19200, 38400, 57600, 115200; 32 bitars tal lagras.
```

Exempel på en textsträng. Alla textsträngar avslutas med ett "osynligt" nollvärde. Denna nolla används som indikering på att strängen är slut – "nullterminated string".

```
WELCOME_SW DCB    "Välkommen" ; Define Constant Byte. 10 byte krävs.
```

9.5. Define Storage (DS)

Dessa direktiv används för att reservera plats för data som skapas under programkörning. All data kan näppeligen inte rymmas i processorns register.

Exempel på tabelldata:

```
MODELNO   DS8     9            ; Plats för en lista med 9 element.
          ; Varje element får uppta 8 bitar
```

9.6. ALIGN

Följande exempel visar på problemet med 32-bitars instruktioner. I ett programområde allokeras (dvs bokas plats) för ett antal byte. Om programkod ska placeras efter denna allokering måste man garantera att programmet hamnar på jämna 32-bitars steg, dvs adressen måste ha slutsiffran (LSD) 0, 4, 8 eller C.

```
        ORG    TEXT_START    ; Area för textmeddelande
message DCB    "Hälsning från S3F441FX. "
        ALIGNROM    5        ; Efterföljande adresserna läggs upp i 32-bitars steg
          ; 2^5 = 32
```

Assemblern brukar vanligast kolla att adresserna är i "align".

9.7. KOMMENTARER

Kommentarer placeras fortlöpande i källkoden. Det underlättar förståelse en månad senare när man vill använda koden utan att tränga in i detaljer. Det kan också användas för att tillfälligt plocka bort kod.

Radkommentarer: ; och //

; Detta är en kommentar som är giltig på denna rad.

; Semikolon kan placeras var som helst på en rad.

En typ av kommentarer som används i programspråket C++ har också spridit sig till assembly-programmering:

// Detta är en "C++-kommentar" som också är giltig på denna rad.

// Divisionstecknena kan placeras var som helst på en rad.

Blockkommentarer: /*.....*/

/* Denna kommentarer kan användas för ett textblock.

Det kan finnas radkommentarer i blocket!!

Blockkommentaren kan placeras på en textraden eller på en egen rad.

*/