ARM32-instruktioner. Prel. sammanfattning

Detta kursmaterial är preliminärt och kan innehålla ”dumheter”, som förhoppningsvis upptäcks efterhand. Materialet kommer att korrigeras och kompletteras. Materialet kommer sedan att vara tillåtet hjälpmedel vid tentamen. Det är därför viktigt att du påpekar fel eller andra brister.

Mnemonic	Instruction	Action
ADC	Add with carry	Rd: = Rn + Op2 + Carry
ADD	Add	Rd: = Rn + Op2
AND	AND	Rd: = Rn AND Op2
B	Branch	R15: = address
BIC	Bit Clear	Rd: = Rn AND NOT Op2
BL	Branch with Link	R14: = R15, R15: = address
BX	Branch and Exchange	R15: = Rn, T 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 AND NOT Op2) OR (Op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd: = (address)
MCR	Move CPU register to coprocessor register	cRn: = rRn {<op>cRm}
MLA	Multiply Accumulate	Rd: = (Rm × 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 × Rs
MVN	Move negative register	Rd: = Not Op2
ORR	OR	Rd: = Rn OR Op2
RSB	Reverse Subtract	Rd: = Op2 - Rn
RSC	Reverse Subtract with Carry	Rd: = Op2 - Rn - Not Carry Flag
SBC	Subtract with Carry	Rd: = Rn - Op2 - Not Carry Flag
STC	Store coprocessor register to memory	address: = CRn
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	<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 EOR Op2
TST	Test bits	CPSR flags: = Rn AND Op2

Inledning

Denna skrift utgör en personlig sammanfattning av de väsentligaste som kan sägas om de instruktioner, som behövs för att skriva ”små” assemblyprogram. En mer omfattande beskrivning finns i följande verk:

”USER’S MANUAL. S3F441FX 32-Bit CMOS RISC Microprocessor. Revision 2” (pdf-dokument).
(Kap 2 och 3 beskriver hur ARM ska programmeras. Alla figurer i denna skrift finns där. Du kan hämta pdf-dokumentet på min hemsida.)

”ARM Architecture Reference Manual” (praktverk och pdf-dokument)
Finns som pdf-dokument (DDI 0100E), att hämta hos Arm Ltd (http://www.arm.com/)

”Programming Techniques” (ARM-dokument DIU 0021A)
I detta pdf-dokument finns många exempel men innehållet är lite föråldrat.

”AR7TDMI. Technical Reference Manual” (ARM-dokument DDI 0210B)
Pdf-dokumentet beskriver hårdvaran men inleds med en mycket koncis framställning av ARM-instruktionerna. Den bästa formelsamling som jag hittat – dock något tuff för nybörjare.

Kommentar

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:

1. Direkt värde (“immediate vale)” – ett 32 bitar värde som skapas genom att högerrotera ett 8-bitars värde ett jämnt antal steg:
2. 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):

```
Rm LSL #5bit_Imm   Logical shift left
Rm LSR #5bit_Imm   Logical shift right
Rm ASR #5bit_Imm   Arithmetic shift right
Rm ROR #5bit_Imm   Rotate right
```

3. Ett godtyckligt register: Rm
4. Skift- och rotationsoperationer utförda på på ett register (Rm). Antalet skift/rotationer finns lagrat i ett annat register (Rs) :

```
Rm LSL Rs          Logical shift left
Rm LSR Rs          Logical shift right
Rm ASR Rs          Arithmetic shift right
Rm ROR Rs          Rotate right
Rm RRX Rs          Rotate right extended
```

Kommentar till kommentaren

Op2, dvs operanden av typ 2, verkar var en riktig ”typ”. Men situationen är inte riktigt så elakartad som den ser ut. Assemblatorn, dvs det program som översätter ditt assembly-program, till maskinkod (dvs sifferkod) kan ibland automatiskt välja ”rätt” variant av operand typ 2.,

Jag har därför i denna skrift lite slarvigt givit följande förenklade uppdelning av Op2
1. Godtyckligt register
2. Op2 (”12-bitars värde”)

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.

```
        MOV    R0,#123        ; Assemblatorn ger felrapport. 0x123 är ett 12-bitars
                              ; värde men kan inte skapas med högerrotation.

MOV     R0,#0x120      ; Assemblatorn kan skapa detta 12-bitars värde m h a
                              ; högerrotation.

MOV     R0,#0x12000000 ; Assemblatorn kan skapa detta 32-bitars värde m h a
                               ; högerrotation.
```

Suffix (ARM)

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

Kommentar

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 resultet av föregående instruktion ger ett värde med MSB = 1 $\Rightarrow$ N = 1
V “Overflow"-flaggan: om resultet av föregående instruktion ger “overflow” $\Rightarrow$ V = 1

Detta behöver man inte tänka på! Genom att använda suffixen på rätt sätt “mjölkas” flaggorn på relevant information. Notera att användaren (dvs programmeraren) själv måste bestämma om ett tal ska tolkas som ett postivt heltal (“unsigned”) eller som ett tal med tecken (“signed”). Suffixet AL kan utelämnas.

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_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)

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

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.

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 etikett!
; Återhoppsadress sparas alltid i LR.


Sub_label
; Här följer programkoden i subrutinen.


; Återhopp till det anropade programmet görs alltid på följande sätt:
        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.
```

Logiska och grundläggande aritmetiska instruktioner: MOV, MVN, CMP, CMN, TEQ, TST, AND, ORR, EOR, BIC, ADD, ADC, SUB, SBC, RSB,RSC

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

```
        MOV     Rd,Rn           ; Rd:= Rn
        MOV     Rd,Op2          ; Rd:= Op2
        MVN     Rd,Op2          ; Rd:= NOT Pp2
; 12bit_value kan förstås vara mindre än 12 bitar. Rd och Rn är godtyckliga
; register. Dck inte R15.
```

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. Testar Rn – Rm >= 0?,
                                ; Rn – Rm = 0?, Rn – Rm < 0 Påverkar alla flaggor.
        CMP     Rn,Op2          ; Jämför Rn med Op2.
        TEQ     Rn,Rm           ; Rn BITVIS XOR Rm ?. Påverkar Z-flaggan.
        TST     Rn,Op2          ; Rn BITVIS OCH Op2 ? Påverkar Z-flaggan.
; 12bit_value kan förstås vara mindre än 12 bitar. Rn och Rm är två godtyckliga
register.
```

De logiska instruktionerna AND, ORR, EOR och, BIC utför bitoperationer på alla bitar i operanden eller operanderna. Några exempel:

```
        AND     Rd,Rn,Rm        ; Rd:= Rn BITVIS OCH Rm
        AND     Rd,Rn,Op2       ; Rd:= Rn BITVIS OCH Op2
        BIC     Rd,Rn,Rm        ; Rd:= Rn BITVIS OCH MED BIVIS INV Rm
        BIC     Rd,Rn,Op2       ; Rd:= Rn BITVIS OCH MED BITVIS INV 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
```

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
; 12bit_value kan förstås vara mindre än 12 bitar. Rm och Rn är två register.
```

Skiftoperationer: ASL, LSL, ASR, LSR, ROR

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 tilläggsoperatorer, 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 smarta tidsbesparande operationer utföras. Jag visar här de två vanligaste LSL och LSR

Skiftoperationen LSL

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

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

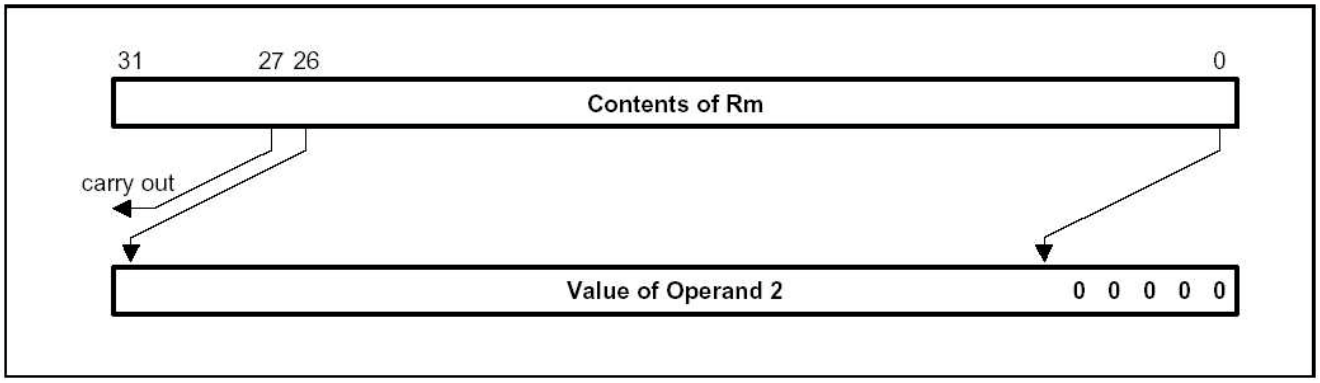

Figure 3-6. Logical Shift Left

Vänsterskift kan användas för att tillverka ett värde som upptar mer än de 12 bitars datakraft som finns tillhanda:

```
MOV     Rd,#8-bit_val         ; Rd:= 8-bitars värde
MOV     Rd,Rd,LSL#5bit_val    ; Rd:= Rd x (max 2^31)
```

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

```
MOV     Rd,#1                 ; Rd:= ”1” i bit 0
MOV     Rd,Rd,LSL#7           ; Rd:= ”1” i bit 7
MOV     Rd,Rd,LSL Rn          ; Rd högerskiftas enligt värde i Rn
```

Skiftoperationerna LSR och ASR

Med hjälp av LSR kan ett positivt värde enkelt divideras med 2^n (där n = 0,1,….,30,31). Om talet ska betraktas som ett tal med tecken ska aritmetiskt högerskift användas istället.

Ovanstående exempel är också relevanta för LSR.

Multiplikation. Multiplikation med ackumulation: MUL, MLA, 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.

MUL, MLA

Generella exempel:

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

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

```
LDR     R0,=0xFFFFFFF6       ; -10
MOV     R1,#20               ; 20
MUL     R2,R0,R1             ; 0xFFFFFF38 (-200) i R2
```

Faktiskt exempel 0xFFFFFFF6 x 20 = 0x13FFFFFF38:

```
LDR     R0,=0xFFFFFFF6       ; 4294967286
MOV     R1,#20               ; 20
MUL     R2,R0,R1             ; 0xFFFFFF38 (-200) i R2
                             ; 0x13 blir ”overflow”
```

MULL, MLAL

Det är sympatiskt att dessa instruktioner 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 0xFFFFFFF6 x 20 = 0x13FFFFFF38:

```
LDR     R0,=0xFFFFFFF6       ; 4294967286
MOV     R1,#20               ; 20
UMULL   R2,R3,R0,R1          ; 0xFFFFFF38 (-200) i R2
                             ; 0x13 hamnar i R3
```

Dataförflyttning (datakopiering): LDR, STR, LDRB, STRB, LDRH, STRH, LDRSB, LDRSH, LDM, STM

Data måste kunna laddas in i register och data måste kunna flyttas ut till olika adresser i minnet. Ordet flyttas är lite vilseledande, det rör sig snarare om kopiering.

LDR, ladda register (”Load from memory into Register ”), innebär att ett värde laddas in i ett register. Värdet kan skapas direkt (”immediate value”) eller också kan det finnas på en adress. Om inget av suffixen B, H, SB eller SH används rör det sig om det 32-bitars värde. I många sammanhang, t.ex. adresser till olika in- och utenheter, behövs bara adresser med 8 eller 16 bitars längd.

B = ”Byte” (8 bitar), H = ”Halfword” (16 bitar),
SB = ”Signed Byte” (8 bitar med tecken), SH = Signed Halfword (16 bitar med tecken)

(Suffixen SB och SH berörs inte i denna lilla skrift.)

STR, lagra register (”Store from Register into memory”), innebär att ett värde sparas på en adress. Endast suffixen B eller H kan användas.

Förflyttning av datablock kan utföras med hjälp av instruktionerna LDM (”Load Multple Registers”) och STM (”Store Multiple Registers”). Dessa instruktioner berörs kortfattat i nästa avsnitt.

LDR, STR med suffixen B och H

Om man vet att ett register ska laddas med ett ”litet” värde (12 bitar) brukar man använd instruktionen MOV. Instruktionen MOV användas också för att flytta (kopiera) data mellan register. Instruktionerna LDR och STR används för dataöverföring mellan processorns register och minnet.

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

Ladda ett register med ett värde. Generella exempel:

```
        LDR     Rd,=32bit_val           ; Rd tilldelas 32-bitars värde
; Värdet kan sålunda ligga i intervallet 0 – 0xFFFFFFFF.
        LDRB    Rd,=32bit_val ;         ; Rd tilldelas de låga 8 bitarna av ett
                                        ; 32-bitars värde. (Dumt exempel.)
```

Indirekt 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 enligt Rn
        LDRB    Rd,[Rn]                 ; Rd:= Innehållet på adress enligt Rn.
                                        ; Relevant om innehållet på adressen är ett
                                        ; 8-bitars värde, exv en 8-bitars port.
```

Indirekt adressering med ”offset”. Ladda ett register med ett värde, som finns på en adress + förskjutning. (Exempel på Array finns deklarerad på nästa sida.) 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 med offset 2 byte.
        LDR     Rd,[Rn,Rm]              ; Rd:= element med offset som ges av Rm
; Offset räknas alltid i byte
```

Indirekt adressering med ”offset” och ”write back”. Ladda ett register med ett värde, som finns på en adress + förskjutning. Automatisk uppdatering av offset. Generella exempel:

```
        LDR     Rn,=Array               ; Rn tilldelas ett 32-bitars adressvärde.
        LDR     Rd,[Rn],#4              ; Rd:= element på adress enl. Rn. Rn:= Rn + 4
        LDR     Rd,[Rn,#4]!             ; Rd:= element med offset 4 byte. Rn:= Rn + 4
```

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

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

Indirekt adressering. Lagra ett register på en adress. Generella exempel:

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

Indirekt adressering med ”offset”. Ladda ett register med ett värde, som finns på en adress + förskjutning. Generella exempel:

```
        LDR     Rn,=Array               ; Rn tilldelas ett 32-bitars adressvärde.
        STRH    Rd,[Rn,#2]              ; Element med offset 16bitar ersätts med
                                        ; 16 bitar av Rd.
        STR     Rd,[Rn,Rm]              ; Element med offset enl. Rm ersätts av Rd.
; Offset räknas alltid i byte.
```

Indirekt adressering med ”offset” och ”write back”. Ladda ett register med ett värde, som finns på en adress + förskjutning. Automatisk uppdatering av offset. Generella exempel:

```
        STR     Rd,[Rn],#4              ; Element på adress enl. Rn ersätts av Rd.
                                        ; Uppdatering (”write back”): Rn:= Rn + 4
        STR     Rd,[Rn,#4]!             ; Element med offset 4 byte tilldelas Rd.
                                        ; Uppdatering (”write back"):= Rn + 4
```

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

Datalista (”array”)

Här följer deklaration av lämplig datalista (”array”) som passar till exemplen där adressen Array används. (Direktiven ORG och DC förklaras i nästa avsnitt.) Etiketten Array placeras på lämpligt ställe eller styrs önskad adress med hjälp av direktivet ORG. Notera att namnvalet ”Array” är godtyckligt. Du kan exv. kalla etiketten Array:

```
; Array = basadress. Exv. värdet 234567 ligger på offset2x4byte = 8 byte
Array  DC32   12345, 67891, 234567, 789102 ; Dessa 32-bitars tal placeras efter
                                           ;varandra med start på adressen Array
```

Texthantering (”textsträngar”)

Stränghantering, dvs text, utgör också exempel på datalistor. Här följer ett annat exempel som utgår från textsträngen WELLCOME_SW som deklareras på nästa sida:

```
        LDR     R1,=WELLCOM_SW          ; R1 pekar nu på första tecknet i strängen
write_str_
        LDRB    R0,[R1],#1              ; R0:= tecken enl adress i R1. R1:= R1 + 1
                                        ; R1 pekar alltid på tecken som ska hämtas.
        CMP     R0,#0                   ; Ett osynligt tecken med ASCII-värdet 0
        BEQ     ready_str               ; ligger alltid sist i strängen (listan).
        BL      Put_char                ; En subrutin som kan skriva ut tecknet i R0
        B       write_str               ; Fortsätt med nästa tecken.
ready_str
; Put_char mottar följande tecken: ’V’,’ä’,’l’,’k’,’o’,’m’,’m’,’e’,’n’
; Det sista tecknet, 0, visar att strängen är avslutad (”terminerad”). Detta
; tecken används av programmet för att sluta anropa subrutinen Put_char.
```

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

Förflyttning av datablock kan utföras med hjälp av instruktionerna LDM (”Load Multple 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	LDMDA
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 stack-hantering 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.

Suffixen F och E syftar på "full" eller "empty"stack. Om stacken är fylld måste stackpekaren flyttas före ("pre") datakopiering. Om stacken är tom kan data först placeras, sedan ("post") ändras stackpekaren. Suffixet A syftar på en stack som är "ascending" (dvs växer mot högre adresser) medan däremot suffixet D syftar på en stack som "descending", dvs växer mot lägre adresser. Det betyder att assembly-programmeraren har fyra olika stacktyper att välja. Ett programmerare som använder ett högnivåspråk (exv C) behöver inte tänka på detta. En "C-stack" använder oftast en stack som växer mot lägre adresser. Stackpekaren pekar vanligtvis på det sista elementet på stacken. Det betyder att kombinationen STMFD-LDMFD är aktuell.

Suffixen IB, IA, DB, DA antyder att det *inte* rör sig om stackhantering . Suffixen betyder "Increment Before", "Increment After", "Decrement Before", "Decrement After".

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
```

I många fall räcker det inte med en dataöverföring för att flytta alla data. Man kan tänka sig ett fall där data består av ett stort antal block och varje block kan bestå av ett antal ord. Vi kan utgå från ovanstående exempel och tänka oss att 100 sådan block ska flyttas.

```
; 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
       MOV    R2,#100               ; 100 block ska kopieras
repeat
       LDMIA  R0!,{R4-R11}          ; Ladda 8 ord ("words") från källan
       STMIA  R1!,{R4-R11}          ; och placera på destinationen
       SUBS   R2,R2,#1              ; Minska R2 med 1 och "enabla" Z-flaggan
       BNE    repeat                ; Hoppa till repeat om R2 skild från 0.
; Nu har 100 block på vardera 8 "words" kopierats.
```

Utropstecknet efter registret betyder att innehållet i registret automatiskt ändras (uppdateras). Det är en s.k. "write back operation". Notera hur instruktionerna LMDIA och STMIA utgör ett naturligt par.

Stackhantering - skydda register

Jag utgår från en stack som växer mot lägre adresser, dvs "descending stack". Jag låter stackpekaren pekar på det sista elementet (dataordet) på stacken och måste därför flyttas före kopiering av data från registren. Det betyder att operationen blir en "pre-decrement store", dvs STMFD.

När data hämta från stacken pekar redan stackpekaren på det första elementet som sk laddas in i registren. Stackpekaren flyttas efter hämtningen av data. Stackpekaren måste dessutom ökas efter utrymme på stacken frigörs. Det blir sålunda en LDMFD-instruktion.

I nedanstående exempel struntar vi vilken startadress stacken har. Vi hoppar rakt in i ett viktigt exempel, nämligen hur subrutin bör se ut.

```
;-----------------------------------------------------------------------
;      SUBRUTIN Delay_ms
; Anropas med värde i r0, vilket ska ge motsvarande fördröjning
; i millisekunder. Användaren måste själv kalibrera innerslingan.
; Anropas med värde i r0. Påverkar register r1.
DELAY_CALIB   EQU    100           ; Innerslingan som ska ta en millisekund
Delay_ms
       STMFD  SP!,{R1}             ; Spara undan R1 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!,{R1}             ; Återställ innehållet i R1.
       MOV    PC,LR                ; Tillbaka till huvudprogrammet.
;-----------------------------------------------------------------------.
```

Principen är att man sparar innehållet i de register man använder. I ovanstående exempel används R1 och det första subrutinen gör är att spara innehållet på stacken. I andra assembly-språk kallas detta förfarande för PUSH. Strax innan subrutinen avslutas återställas innehållet i register R1. Det kallas ibland för POP. Först "pushas" R1 och sedan poppas" det. Detta förenklar stackhanteringen.

Om subrutinen innehåller ett subrutinanrop måste återhoppsadressen för det aktuella subrutinanropet också sparas. I nedanstående exempel finns en subrutin LED-blink. I denna subrutin finns ett anrop av en annan subrutin, Delay_ms.

```
;-----------------------------------------------------------------------

;       SUBRUTIN LED_blink

; Väntar ett antal millisekunder enligt värde i r0. Tänder sedan alla
; lysdioderna under lika låg tid. Påverkar register R1, R2, R3 och LR

LED_blink

        STMFD   SP!,{R1-R3,LR}        ; Spara de register som påverkas!

; Spara fördröjningsvärdet

        MOV     R2,R0

; Dröj ett antal millisekunder angivna av R0

        BL      Delay_ms        ; Anrop till subrutin Delay_ms. Nu förstörs det
                                ; aktuella innehållet i LR.

; Tänd alla lysdioder

        LDR     R1,=rP0         ; Adress till port 0

        MOV     R0,#0x0F        ; En 1:a i en bit ger en 1:a på utgången,

        STRB    R0,[R1]

; Dröj ett antal millisekunder angivna av R0

        BL      Delay_ms        ; Anrop till subrutin Delay_ms. Nu förstörs det
                                ; aktuella innehållet i LR.

; Släck alla lysdioder

        LDR     R1,=rP0         ; Adress till port 0

        MOV     R0,#0x00        ; En 0:a i en bit ger en 1:a på utgången,

        STRB    R0,[R1]

; Hämta tillbaka återhoppsadress och hoppa tillbaka

        LDMFD   SP!,{R1-R3,LR}        ; Återställ alla register

        MOV     PC,LR                 ; Hoppa tillbaka med sparad LR

;-----------------------------------------------------------------------
```

Återhoppet till anropande programdelen kan automatiskt gör genom att placera PC i registerlistan. De två sista kan då utföras med en instruktion:

```
        LDMFD   SP!,{R1-R3,PC}        ; Återställ alla register även programpekaren!
```

Stackhantering - parameteröverföring

I ovanstående exempel används processorns register för att överföra information till en subrutin. Detta förfaringssätt blir allt vanligare vid "modern" programmering när man har tillgång till en processor med en hyfsad uppsättning av interna register. Det traditionella sättet att överföra information till en subrutin är att gå via stacken. Jag ber att få återkomma med en beskrivning hur dettas sker i samband med laboration 4, som kommer att handla om hur språket C handskas med parameteröverföring. Det innebär att detta material kommer att kompletteras vid ett senare tillfälle.

ASSEMBLERDIREKTIV - ENLIGT "EWARM": ORG, END, EQU, DCB, DC8, DC16, DC32, DSB, DS8, DS16, DS32,

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

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

```
ROM_START     EQU    0x01FF0000          ; Programminnet börjar på denna dress
```

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

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

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

Direktiven Define Constant (DC)

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

Exempel på tabelldata:

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

Exempel på två textsträngar. Dessa behandlas som två olika listor. Alla textsträngar avslutas med ett ”osynligt” nollvärde. Denna nolla används som indikering på att strängen är slut – ”nullterminated string”.

```
WELLCOME_SW   DCB    ”Välkommen”    ; Define Constant Byte. 10 byte krävs.
WELLCOME_UK   DC8    ”Wellcome”     ; DCB betyder DC8. 1 byte är  alltid 8 bitar.
```

Direktiven 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
```

Direktivet 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
```

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

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 använda för tillfället 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 använd 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.
```

Blockkomentarer: /* . . . . . */

```
/* Denna kommentarer kan användas för ett textblock.
   Det kan finnas radkommentarer i blocket!!
   Blockkommentaren kan placeras på en textraden eller på en egen rad.
*/
```

KOMPLETTERINGAR