Assemblerprogrammering för ARM – del 1

Ur innehållet:

- Datatyper och ordlängder
  - Variabeldeklarationer
- Programkonstruktioner
  - Tilldelningar
  - Uttrycksevaluering
  - Ovillkorliga programflöden
  - Funktion med returvärde
- Läsanvisningar:
  - Arbetsbok kap 1 och 2
  - Quick-guide, instruktionslistan

Datayper och ordlängder

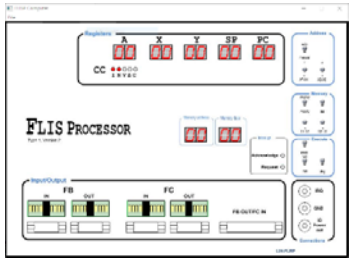
ordlängd	Java	C	ARM	FLISP
8 bitar	`byte`	`signed char`	`BYTE`	`BYTE`
16 bitar	`short`	`(signed) short`	`HALFWORD`	-
32 bitar	`int`	`(signed) int`	`WORD`	-
64 bitar	`long`	`(signed) long long`	`DWORD`	-
32 bitar	`float`	`float`	`FLOAT`	-
64 bitar	`double`	`double`	`DOUBLE`	-
1 bit	`boolean`	-	-	-
16 bit	`char`	-	-	-

Exempel på deklarationer...

Java	C	ARM	FLISP
`byte b;`	`signed char b;`	`b: .SPACE 1`	`b: RMB 1`
`short s;`	`short s;`	`s: .SPACE 2`	`s: RMB 2`
`int i;`	`int i;`	`i: .SPACE 4`	`i: RMB 4`

Adressrum

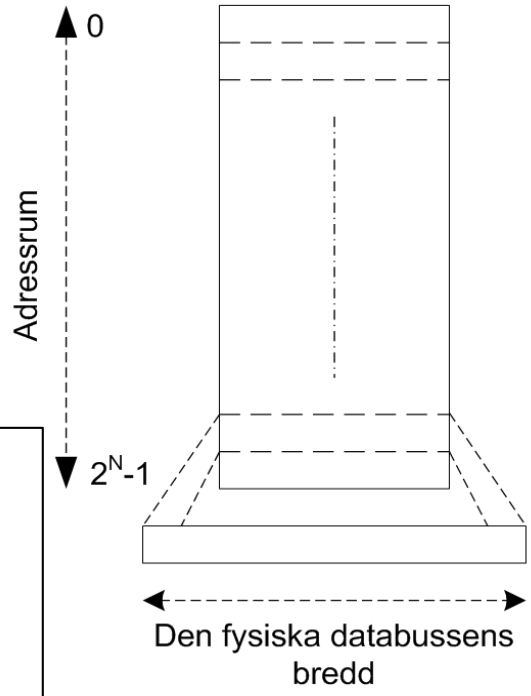
Adressrummet begränsas av adressbussens storlek...



8-bit data,
8-bit adress
2^8 bytes = 256 bytes

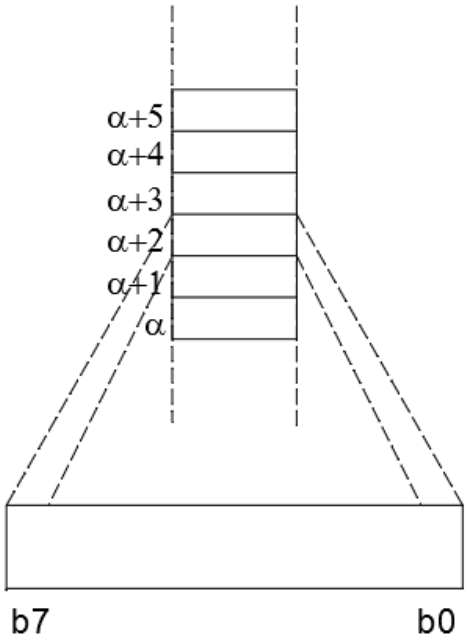
32-bit data,
32-bit adress
2^{32} bytes = 4 294 967 296 bytes
= 4 Giga bytes

64-bit data,
max 64-bit adress
Dagens implementering: 48 bitar
2^{48} bytes = 281 474 976 710 656 bytes
= 281 Tera bytes

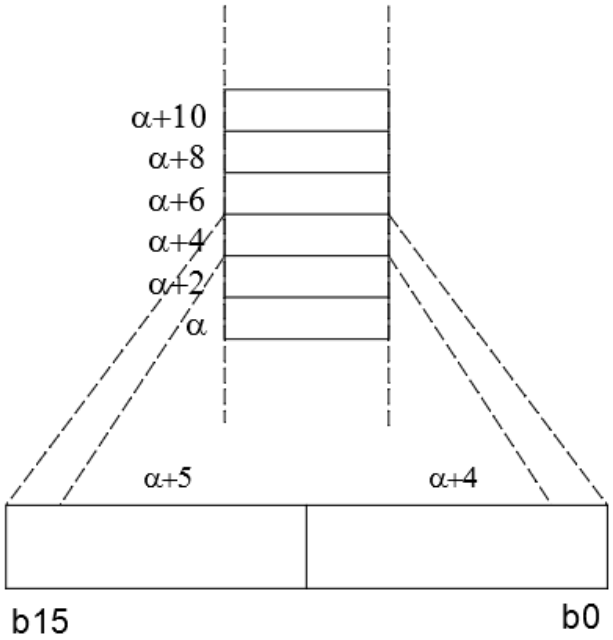


Organisation av minnet

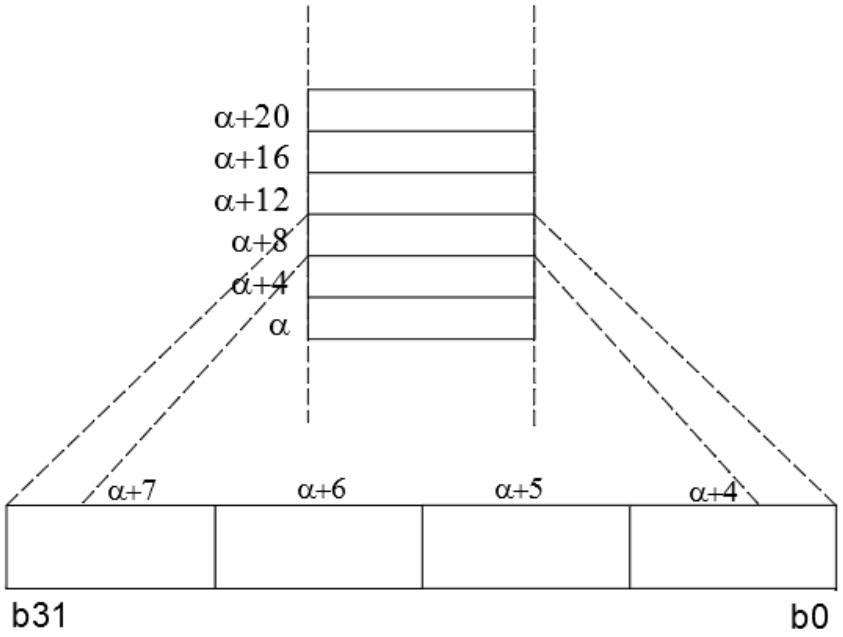
Genom att packa flera bytes i varje minnes-åtkomst reduceras antalet läsningar/skrivningar i minnet. Skillnaden i arbetstakt kan utnyttjas – prestanda ökar



”byte”-organiserat



”halfword”-organiserat

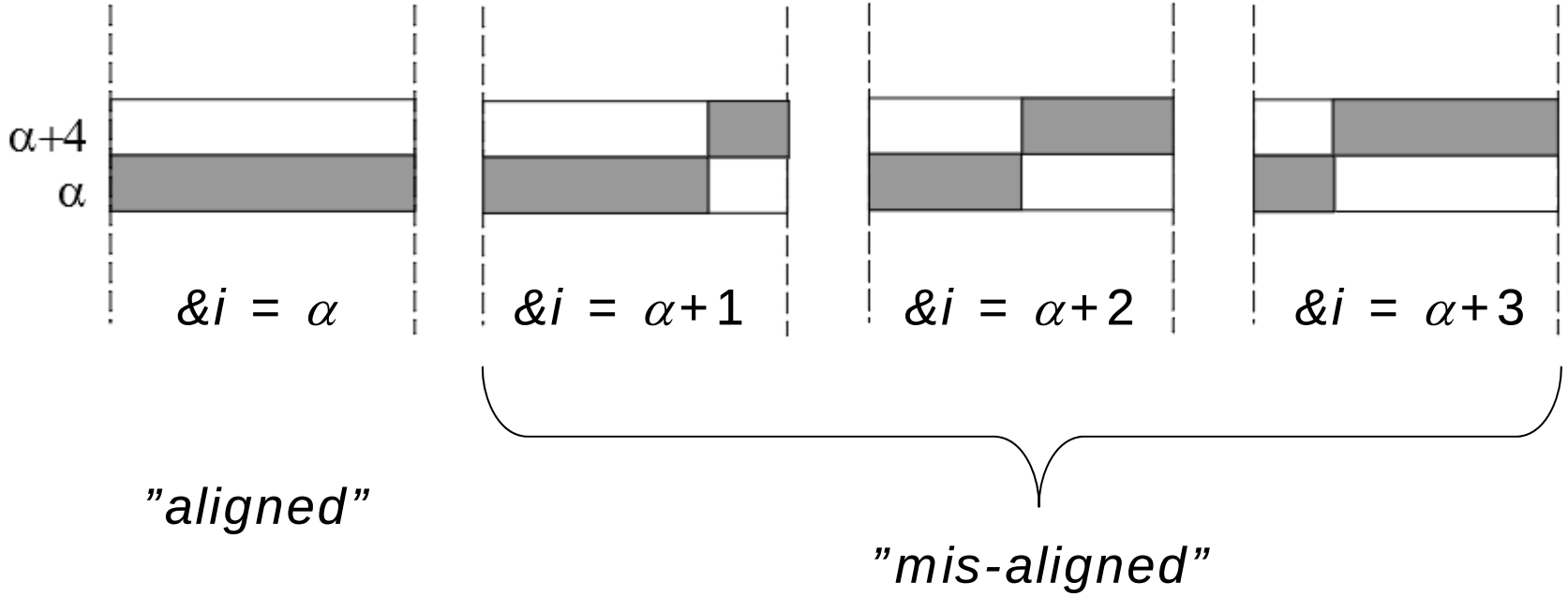


”word”-organiserat

Rättning – ”alignment”

Datatyper som är större än 1 byte kan kräva onödigt många läsningar/skrivningar beroende på var dom placerats.

Exempel: en variabel `int i;`*, kan placeras på minst 4 olika sätt i minnet:*



Deklarationer

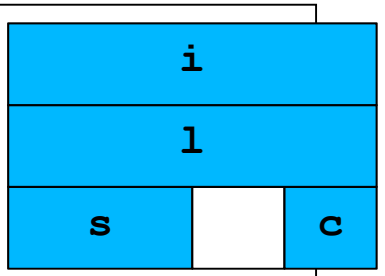
```
char      c;     /* 8-bi
short     s;     /* 16-b
long      l;     /* 32-b
int       i;     /* 32-bitars c    yp, storlek word */
```

```
.ALIGN 1 @ align to half (2 byte)
.ALIGN 2 @ align to word (4 byte)
.ALIGN   @ default, align to word
```

```
char  c;

short s;

long  l;

int   i;
```

Assemblerspråk:

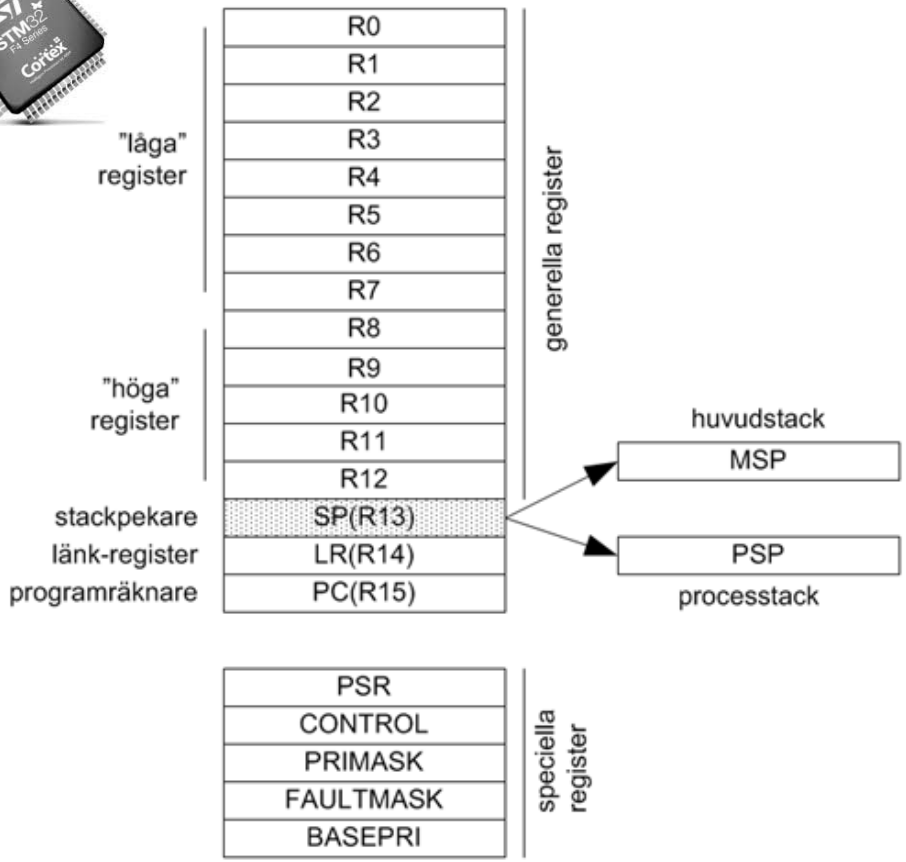
```
c: .SPACE   1
   .ALIGN   1
s: .SPACE   2
   .ALIGN   2
l: .SPACE   4
i: .SPACE   4
   .GLOBL   c,s,l,i
```

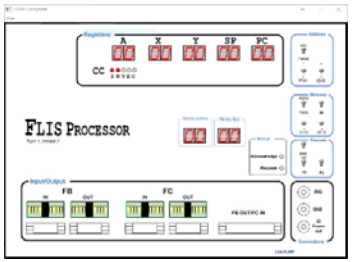


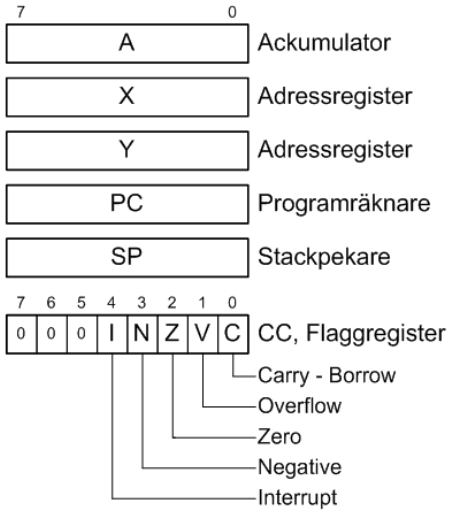
ISA – ”Instruction Set Architecture”

- Vilka operationer kan utföras ?
  Instruktionsgrupper
- Hur lagras operanderna förutom i minnet ?
  Korttidslagring dvs. *register*
- Hur nås operander i minnet?
  Adresseringssätt
- Vilka typer/storlekar av operander kan hanteras ?
  Generella/speciella register, registerstorlek

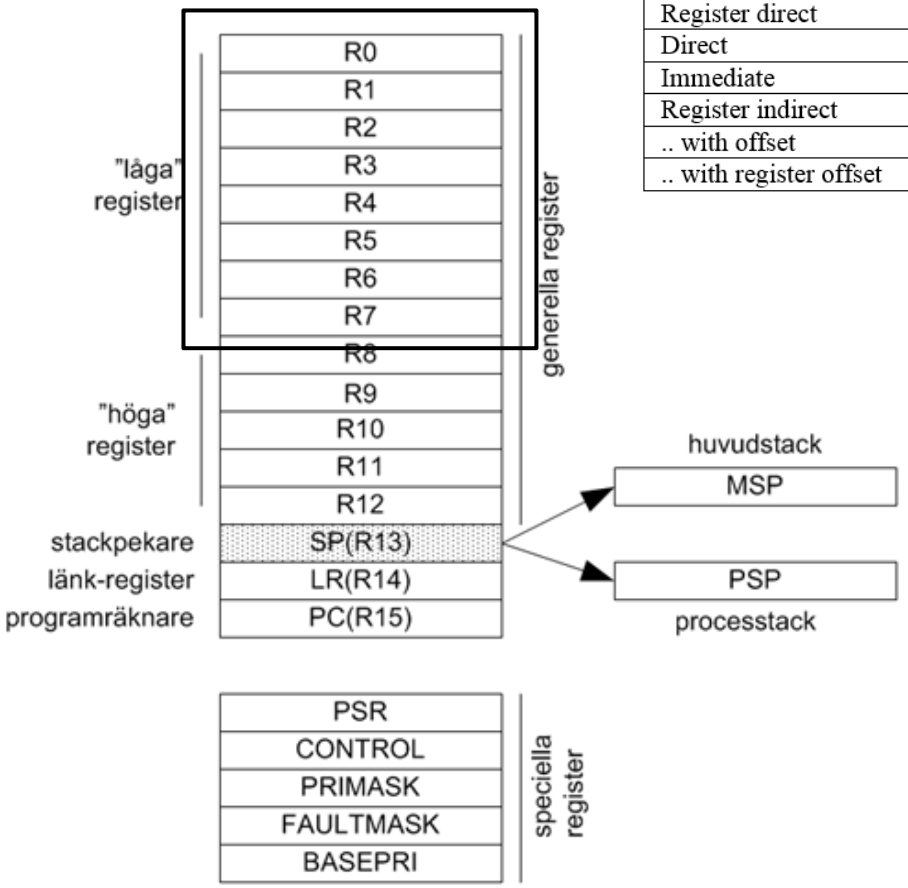
Register


R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP(R13)
LR(R14)
PC(R15)
"låga" register
"höga" register
generella register
stackpekare
länk-register
programräknare
huvudstack
MSP
PSP
processtack
PSR
CONTROL
PRIMASK
FAULTMASK
BASEPRI
speciella register


FLIS Processor


7
0
A
Ackumulator
X
Adressregister
Y
Adressregister
PC
Programräknare
SP
Stackpekare
7 6 5 4 3 2 1 0
0 0 0 I N Z V C
CC, Flaggregister
Carry - Borrow
Overflow
Zero
Negative
Interrupt

Register och adresseringssätt



Namn	Syntax	Exempel	RTN
Register direct	`Rx`	`MOV R0,R1`	R0←R1
Direct	`Symbol`	`LDR R0,symbol`	R0←M(symbol)
Immediate	`#const`	`MOV R0,#0x15`	R0←0x15
Register indirect	`[Rx]`	`LDR R0,[R1]`	R0←M(R1)
.. with offset	`[Rx,#offset]`	`LDR R0,[R1,#4]`	R0←M(R1+4)
.. with register offset	`[Rx,Ri]`	`LDR R0,[R1,R2]`	R0←M(R1+R2)

Exempel:

Med deklarationen: **`int i;`**

```
LDR R0,=i          @ R0←i
LDR R0,[R0]        @ R0←M(i)
```

samma sak som...

```
LDR R0,i           @ R0← M(i)
```

FLISP:

```
LDX #i             @ X←i
LDA ,X             @ A←M(i)
```

samma sak som...

```
LDA i              @ A←M(i)
```

Heltalstyper i C

Typ	Förklaring
`char`	Minsta adresserbara enhet som kan rymma en basal teckenuppsättning. Det är dock en heltalstyp som kan vara signed eller unsigned, detta är implementationsberoende.
`signed char`	Tolkas som heltal med tecken. Måste **minst** kunna representera talområdet [−127, +127], dvs. minst 8 bitar.
`unsigned char`	Tolkas som heltal utan tecken. Måste **minst** kunna representera talområdet [0, 255], dvs. minst 8 bitar.

`short`
`short int`
`signed short`
`signed short int`
	Kort heltalstyp med tecken. Måste **minst** kunna representera talområdet [−32767, +32767], dvs. minst 16 bitar. Observera att talområdet [−32768, +32767] som fås vid 2-komplementsrepresentation, också är tillåtet.

`unsigned short` `unsigned short int`	Kort heltalstyp utan tecken. Måste **minst** kunna representera talområdet [0, +65535], dvs. minst 16 bitar.

`long`
`long int`
`signed long`
`signed long int`
	Lång heltalstyp med tecken. Måste **minst** kunna representera talområdet [−2147483647, +2147483647], dvs. minst 32 bitar.

`unsigned long` `unsigned long int`	Lång heltalstyp utan tecken. Måste **minst** kunna representera talområdet [0, +4294967295], dvs. minst 32 bitar.

`int`
`signed`
`signed int`
	Implementationsbestämd heltalstyp med tecken. Måste **minst** kunna representera talområdet [−32767, +32767], dvs. minst 16 bitar. Observera att talområdet [−32768, +32767] som fås vid 2-komplementsrepresentation, också är tillåtet.
	Observera speciellt att `int` kan vara synonym med `short` eller `long`.

`unsigned` `unsigned int`	Typen `int` utan tecken.

Rättningsvillkor ("alignment")

Av prestandaskäl har vi restriktioner på var olika datatyper kan placeras i minnet:

`int` (32-bitar, word) får bara finnas på en adress jämnt delbar med 4, s.k "word alignment")

`short` (16-bitar, halfword) får bara finnas på en adress jämnt delbar med 2, s.k "halfword alignment")

`char` (8 bitar) kan finnas på såväl udda som jämn adress

Assemblerdirektiv:

`.ALIGN 1` @ LC = (LC + 1) & ~1)

dvs. "avrundar" uppåt till jämn adress

`.ALIGN 2` @ LC = (LC + 3) & ~3)

"avrundar" till address delbar med 4

```
LDR R0,symbol  @ R0← M(symbol)
```

översätts av assemblatorn till den verkliga instruktionen:

```
LDR R0,[PC,offset]
```

offset = (PC-symboladress) << 2, dvs. antal "Words"

....

```
        .ALIGN
symbol:   .WORD  symbolvärde
```

...på motsvarande sätt översätts

```
LDR R0,=symbol
```

till den verkliga instruktionen

```
ADD R0,PC,#offset
```

så att *adressen* till `symbol` hamnar i R0

Instruktioner för tilldelningar

Instruktion	Betydelse	Datatyp
`LDRB`	Load byte	`unsigned char`
`LDRSB`	Load signed byte	`signed char`
`LDRH`	Load halfword	`unsigned short`
`LDRSH`	Load signed halfword	`signed short`
`LDR`	Load word	`unsigned/signed int`
`MOV`	Move	`(0..0xFF)` liten konstant
`STRB`	Store byte	`char`
`STRH`	Store halfword	`short`
`STR`	Store word	`int`

ARM är en så kallad ”load/store”-arkitektur vilket innebär att endast load/store instruktioner läser/skriver i minnet.

Övriga instruktioner opererar direkt på register.

Tilldelningar

Om det är en liten konstant (8 bitar) kan vi använda MOV-instruktionen

```
char  c;
   c = 1;

short s;
   s = 256;
(=0x0100)

int   i;
   i = 65535;
(=0x00010000)
```

ARM:

```
c:      .SPACE    1
 MOV    R0,#1
 LDR    R7,=c
 STRB   R0,[R7]
.....
s:      .SPACE    2
 LDR    R0,=256
 LDR    R7,=s
 STRH   R0,[R7]
.....
i:      .SPACE    4
 LDR    R0,=65535
 LDR    R7,=i
 STR    R0,[R7]
.....
```

FLISP:

```
c:      RMB     1
 LDA    #1
 STA    c
.....
s:      RMB     2
 LDA    #1
 STA    s
 LDA    #0
 STA    s+1
.....
i:      RMB     4
 LDA    #0
 STA    i
 LDA    #1
 STA    i+1
 LDA    #0
 STA    i+2
 STA    i+3
```

Tilldelningar: `a = b;`

Tilldelningar görs alltid via något register, grundregel, load/store-operationer anpassade efter datatyp....

Exempel:

Följande deklarationer är gjorda på global nivå, visa hur deklarationerna uttrycks i assemblerspråk.

```
char        c1,c2;
short       s1,s2;
int         i1,i2;
```

visa också hur följande tilldelningssatser kan kodas i ARM-v6 assemblerspråk:

```
c1 = c2;
s1 = s2;
i1 = i2;
```

Vi löser på tavlan...

Datatyper med olika storlekar

Olika datatyper i höger/vänsterled kan kräva typkonvertering

Exempel:

```
char c;
int  i;
```

större till mindre typ ger trunkering:

`c = i;` dvs. `c = (char) i;`

mindre till större typ kräver teckenutvidgning

`i = c;` dvs. `i = (int) c;`

Decimalt	char	short	int
-128	0x80	0xFF80	0xFFFFFF80
127	0x7F	0x7F	0x7F

Dvs. vid teckenutvidgning kopieras teckenbiten till alla bitar med högre signifikans

Instruktioner för teckenutvidgning

`SXTB Rx,Ry`

?	?	?	0xxxxxxx
00000000	00000000	00000000	0xxxxxxx
?	?	?	1xxxxxxx
11111111	11111111	11111111	1xxxxxxx

Sign Extend Byte
8 till 32 bitar

`UXTB Rx,Ry`

?	?	?	0xxxxxxx
00000000	00000000	00000000	0xxxxxxx
?	?	?	1xxxxxxx
00000000	00000000	00000000	1xxxxxxx

Unsigned Extend Byte
8 till 32 bitar

`SXTH Rx,Ry`

?	?	0xxxxxxx	xxxxxxxx
00000000	00000000	0xxxxxxx	xxxxxxxx
?	?	1xxxxxxx	xxxxxxxx
11111111	11111111	1xxxxxxx	xxxxxxxx

Sign Extend
Halfword
16 till 32 bitar

`UXTH Rx,Ry`

?	?	0xxxxxxx	xxxxxxxx
00000000	00000000	0xxxxxxx	xxxxxxxx
?	?	0xxxxxxx	xxxxxxxx
00000000	00000000	0xxxxxxx	xxxxxxxx

Unsigned Extend
Halfword
16 till 32 bitar

Loadinstruktioner med teckenutvidgning

Teckenutvidgningen sker direkt

Unsigned Extend Byte

`LDRB Rx,[Ry,#k]`

?	?	?	0XXXXXXX
00000000	00000000	00000000	0XXXXXXX
?	?	?	1XXXXXXX
00000000	00000000	00000000	1XXXXXXX

Signed Extend Byte

`LDRSB Rx,[Ry,Rz]`

?	?	?	0XXXXXXX
00000000	00000000	00000000	0XXXXXXX
?	?	?	1XXXXXXX
11111111	11111111	11111111	1XXXXXXX

Dvs. antingen:

```
LDRB   Rx,[Ry,#k]
SXTB   Rx,Rx
```

eller

```
MOV    Rz,#k
LDRSB  Rx,[Ry,Rz]
```

På samma sätt med `LDRH` *och* `LDRSH`

Tilldelningar med typkonvertering

Exempel:

typkonvertering ....

```
signed char c
short       s
int         i
```

visa också hur följ
assemblerspråk:

a) `c = s;`

b) `s = c;`

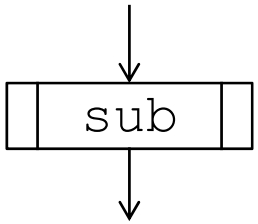
c) `s = i;`

```
c = s;
LDR     R3,=s       @ adress (&s) till R3
LDRH    R3,[R3]     @ (s) till R3, bit 15-31 nollställs
LDR     R2,=c       @ adress (&c) till R2
STRB    R3,[R2]     @ R3 (byte) till (c)
s = c;
LDR     R3,=c       @ adress (&c) till R3
LDRB    R3,[R3]     @ (c) till R3, bit 8-31 nollställs
SXTB    R3,R3       @ teckenutvidgning BYTE->WORD, R3
LDR     R2,=s       @ adress (&s) till R2
STRH    R3,[R2]     @ R3 (halfword) till (s)
s = i;
LDR     R3,=i       @ adress (&i) till R3
LDR     R3,[R3]     @ (i) till R3
LDR     R2,=s       @ adress (&s) till R2
STRH    R3,[R2]     @ R3 (halfword) till (s)
```

Anm: typen `char` kan enligt C-standard vara `unsigned` eller `signed` beroende på kompilatorn. För GCC-ARM gäller att `char = (unsigned char)`

Subrutiner (”funktioner”) och ”goto”

C-operator	Betydelse	Operation	Instruktion	RTN	*FLISP*	*RTN*
`f()`	Funktions-anrop	Branch and link	`BL <label>`	`LR←PC, PC←label`	`BSR <label>` `JSR <label>`	`SP←SP-1, (SP)←PC`
`return`		Branch and exchange	`BX Rx`	`PC←Rx`	`RTS`	`PC←(SP), SP←SP+1`
`goto`	ovillkorlig	Branch	`B <label>`	`PC←label`	`BRA <label>` `JMP <label>`	`PC←label`



C:

```
...
    sub ();
...
```

Exempel:

```
    BL     sub
@ returadress -> LR
    ...

sub:
    ...
    BX     LR
```

Returadressen sparas i register. Snabbt, men klarar ej rekursion

Returvärde från funktion

Konvention: Returvärde i R0 för `char`, `short` och `int`

Exempel:

```
char  f1( void );
short f2( void );
int   f3(void);
```

Alla dessa funktioner lämnar returvärdet i R0.

```
BL      f1          @ returvärdet från f1 nu i R0
LDR     R3,=c       @ adress (&c) till R3
STRB    R0,[R3]     @ R0 (byte) till (c)

BL      f2          @ returvärdet från f2 nu i R0
LDR     R3,=s       @ adress (&s) till R3
STRH    R0,[R3]     @ R0 (halfword) till (s)

BL      f3          @ returvärdet från f3 nu i R0
LDR     R3,=i       @ adress (&i) till R3
STR R0,[R3]         @ R0 (word) till (s)
```

Exempel:

Utgå från deklaration

```
char c;
short s;
int  i;
```

koda tilldelningarna

```
c = f1();
s = f2():
i = f3();
```

Uttrycksevaluering

DEST = (type of DEST) (uttryck); "värdeuttryck"

Exempel:

```
int a,b,c;
a = b+c;
```

(uttryck); "sanningsuttryck", dvs ==0 eller != 0

Exempel:

```
(a+1);      /* falskt om a==-1 */
(a <= b);   /* falskt om a>b */
```

Instruktioner för unära operationer

C-operator	Betydelse	Datatyp	Instruktion
~	bitvis not	signed/unsigned	`MVN`
–	negation	signed/unsigned	`NEG (RSB)`

DEST = *operation* (type of DEST) OPERAND;

Exempel: Antag följande deklarationer på ”toppnivå”:

```
signed char   c,d;
```

Koda följande tilldelningssatser i assembler:

```
c = ~d;
c = -d;
```

```
LDR     R3,=b
LDRB    R3,[R3]
SXTB    R3,R3
MVN     R3,R3   (NEG R3,R3)
LDR     R2,=a
STRB    R3,[R3]
```

FLISP:

```
c:  RMB 1
d:  RMB 1

 LDA    d
 COMA
 STA    c
.....
 LDA    d
 NEGA
 STA    c
.....
```

Instruktioner för binära operationer

C-operator	Betydelse	Datatyp	Instruktion
+	addition	signed/unsigned	`ADD Rd,Rs1,Rs2`
–	subtraktion	signed/unsigned	`SUB Rd,Rs1,Rs2`
*	multiplikation	signed/unsigned	`MUL Rd,Rs1,Rs2`
/	division	unsigned	finns ej som instruktion, mjukvaruimplementerat
%	modulus		
&	bitoperation AND	signed/unsigned	`AND Rd,Rs1,Rs2`
|	bitoperation OR	signed/unsigned	`ORR Rd,Rs1,Rs2`
^	bitoperation EOR	signed/unsigned	`EOR Rd,Rs1,Rs2`
<<	rotation vänster	signed/unsigned	`LSL Rd,Rs1,Rs2`
>>	rotation höger	signed	`ASR Rd,Rs1,Rs2`
		unsigned	`LSR Rd,Rs1,Rs2`

Observera att alla operationer utförs på 32-bitars tal i register

Kodgenerering, binära operationer

DEST = (type of DEST) OPERAND1 *operation* (type of DEST) OPERAND2;

```
LDRx    R1,OPERAND1
   (ev. teckenutvidga)
LDRx    R2,OPERAND2
   (ev. teckenutvidga)
operation R0,R1,R2
LDR     R7,=DEST
STRx    R0,[R7]
```

Anm: `LDRB` och `LDRH` teckenutvidgar alltid som unsigned

Exempel: Addition av 16-bitars tal

Eftersom alla instruktioner opererar på 32-bitar måste ingående operander först anpassas:

```
short int sa,sb,sc;
   ...
 sa = sb + sc;
```

```
sa: .SPACE    2
sb: .SPACE    2
sc: .SPACE    2
...
    LDR    R0,=sb
    LDRH   R0,[R0]
    SXTH   R0,R0
    LDR    R1,=sc
    LDRH   R1,[R1]
    SXTH   R1,R1
    ADD    R0,R0,R1
    LDR    R1,=sa
    STRH   R0,[R1]
```

```
 unsigned short int  usa,usb,usc;
   ...
  usa = usb + usc;
```

```
usa:   .SPACE    2
usb:   .SPACE    2
usc:   .SPACE    2
...
    LDR    R0,=usb
    LDRH   R0,[R0]
@   UXTH   R0,R0      @ teckenutvidga
    LDR    R1,=usc
    LDRH   R1,[R1]
@   UXTH   R1,R1      @ teckenutvidga
    ADD    R0,R0,R1
    LDR    R1,=usa
    STRH   R0,[R1]
```

Exempel: Uttrycksevaluering

Exempel:

Utgå från att följande deklarationer är gjorda på global nivå.

```
char   f(void);
int    a,b;
```

Visa en kodsekvens, i ARM assemblerspråk, som evaluerar följande uttrycks värde till register R0.

```
f() + a * ~( b & 4 );
```

Vi löser på tavlan...