

Korsutveckling för ARM/Thumb – del 1

Ur innehållet:

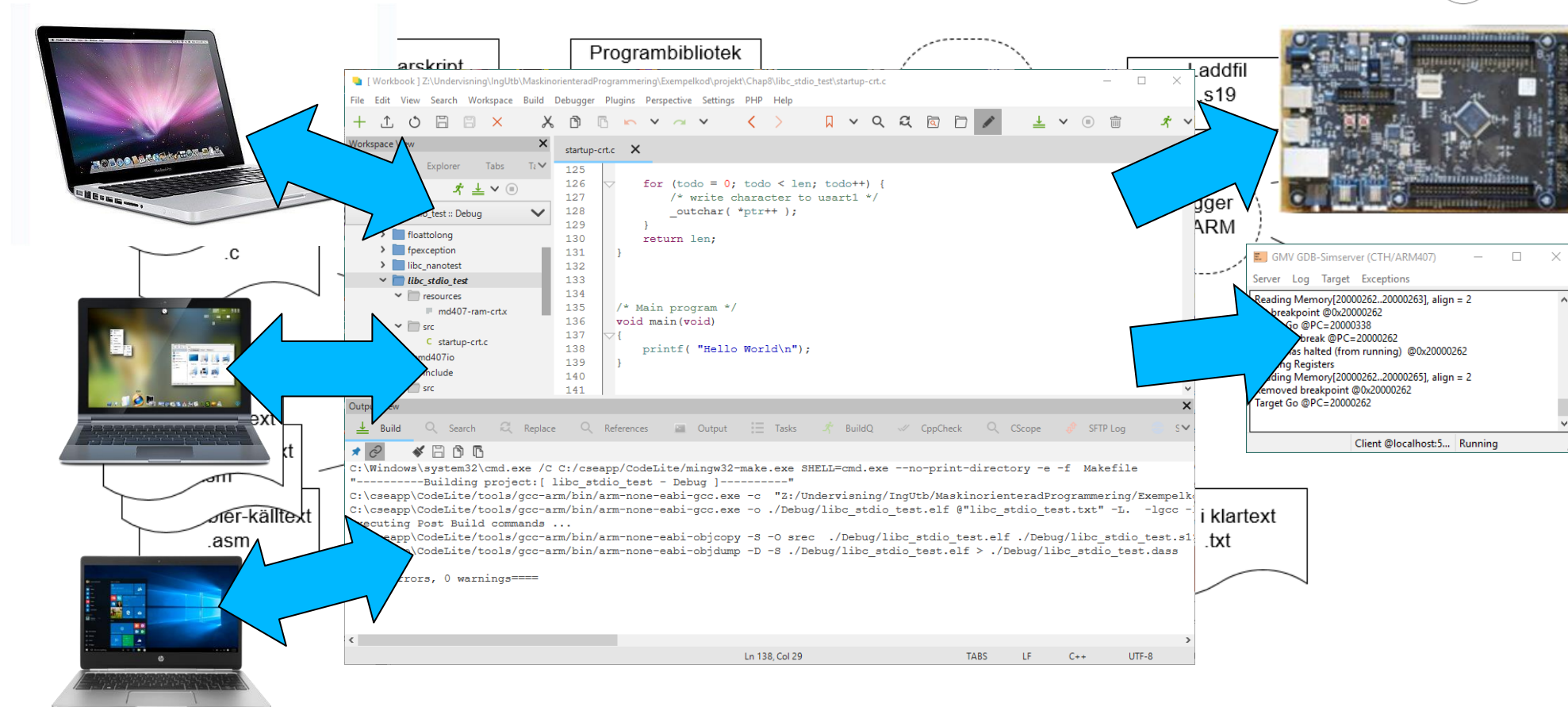
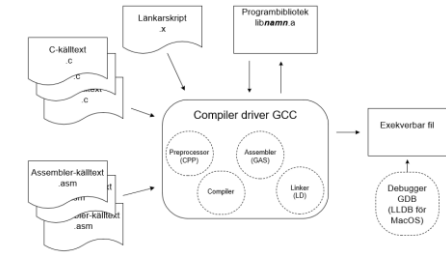
- Programutveckling i C och assembler
 - Programutvecklingsmiljö för GCC/ARM
 - Korskompilering
- Programmering i C och kodgenerering för ARM
 - Tilldelningar
 - Uttrycksevaluering
 - Ovillkorliga programflöden
 - Funktion med returvärde



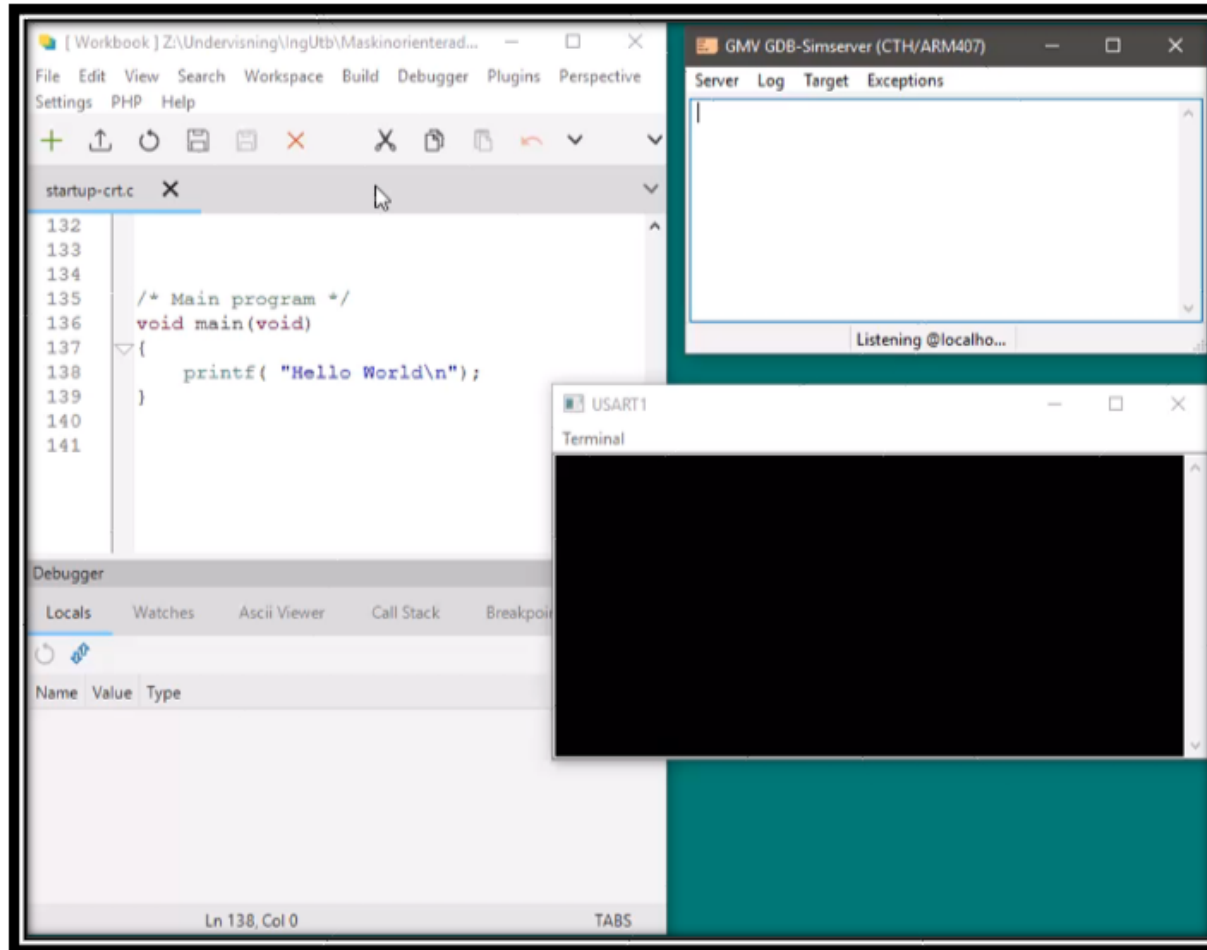
Målsättningar:

- Bli bekant med korsutvecklingsmiljön för ARM
- Utforma och testa ett enkelt C-program för MD407

Värddator och måldator: Korsutveckling i C och assembler



"Hello World" från MD407

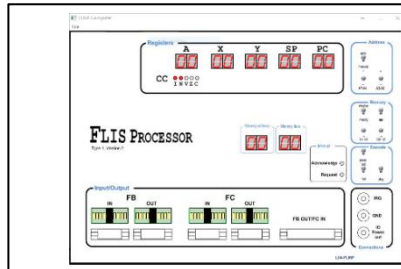


Användningen av `printf` kräver att vi tillhandahåller speciella drivrutiner för terminalen.

Drivrutinerna måste anpassats för laborationssystemet MD407.

Adressrum

Adressrummet begränsas av adressbussens storlek...



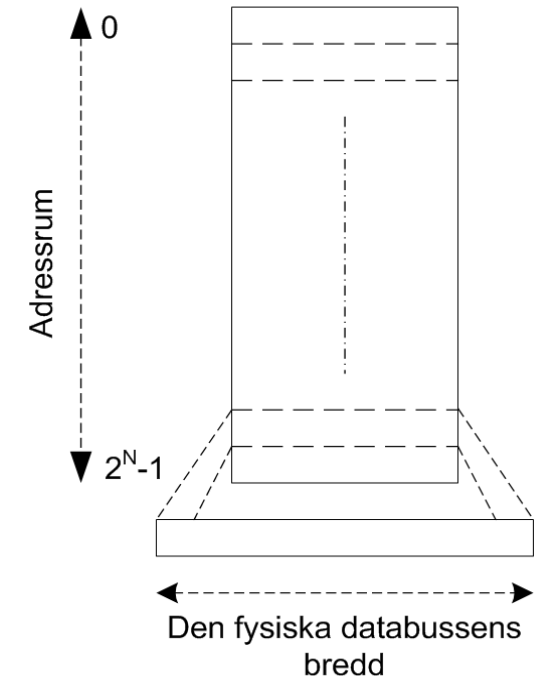
*8-bit data,
8-bit adress
 2^8 bytes = 256 bytes*



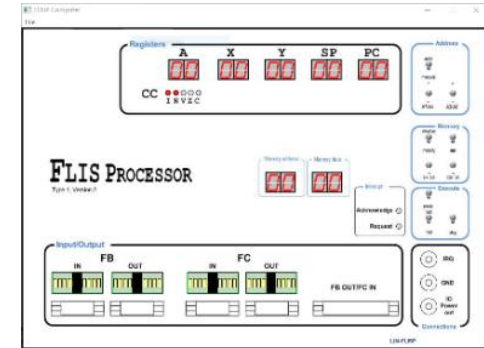
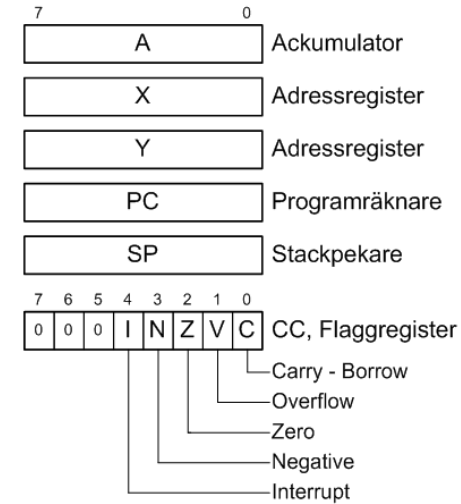
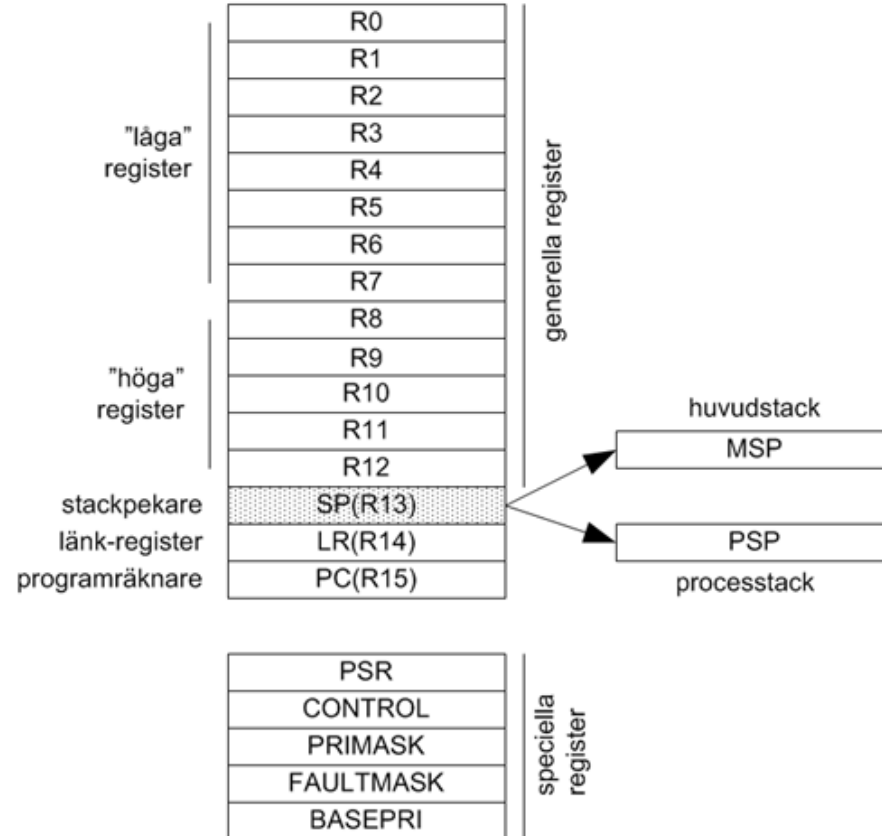
*32-bit data,
32-bit adress
 2^{32} bytes = 4 294 967 296 bytes
= 4 Giga bytes*



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



Register



Datatyper och ordlängder

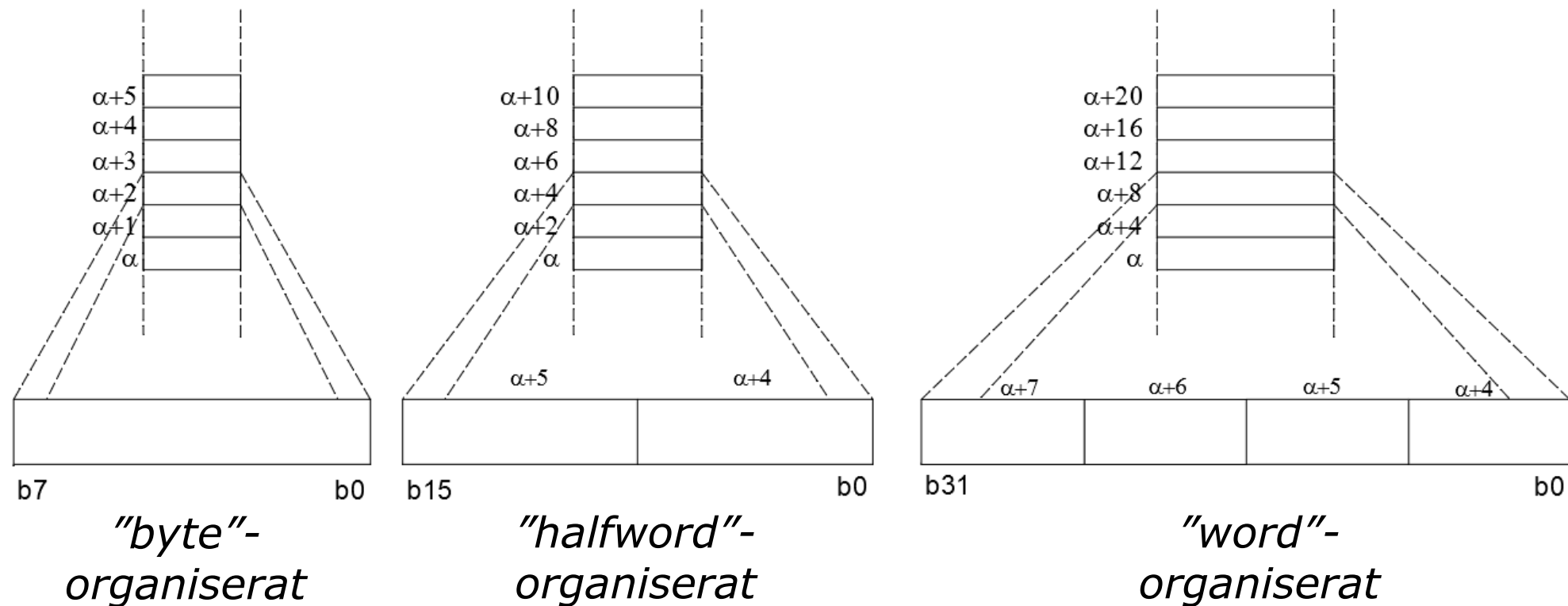
ordlängd	Java	C	ARM	FLISP
8 bitar	byte	signed char	BYTE	BYTE
16 bitar	short	(signed) short	HALFWORD	–
32 bitar	int	(signed) int	WORD	–
64 bitar	long	(signed) long long	DWORD	–
32 bitar	float	float	FLOAT	–
64 bitar	double	double	DOUBLE	–
1 bit	boolean	-	-	-
16 bit	char	-	-	-

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

Organisation av minnet

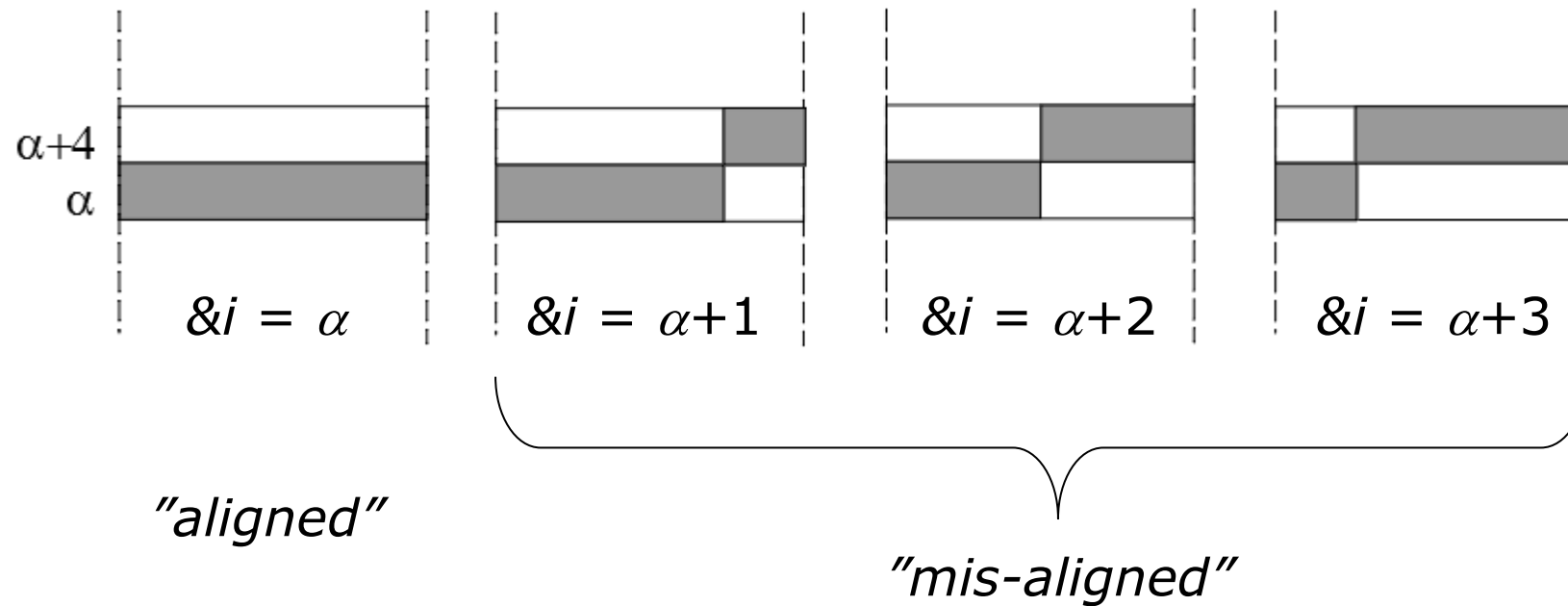
Genom att packa flera bytes i varje minnesåtkomst reduceras antalet läsningar/skrivningar i minnet. Skillnaden i arbetstakt kan utnyttjas – prestanda ökar



Rättning – "alignment"

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

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



Deklarationer

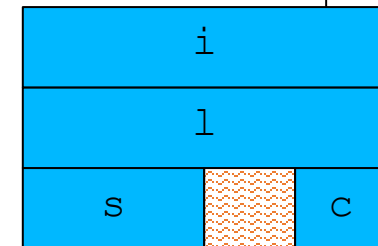
```
char    c;    /* 8-bitars data, korlek word */
short   s;    /* 16-bitars data, korlek word */
long    l;    /* 32-bitars data, korlek word */
int     i;    /* 32-bitars data, korlek word */
```

```
.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

- ⊕ Hur lagras operanderna förutom i minnet ?

Korttidslagring dvs. *register*

- ⊕ Hur nås operander i minnet?

Adresseringssätt

- ⊕ Vilka typer/storlekar av operander kan hanteras ?

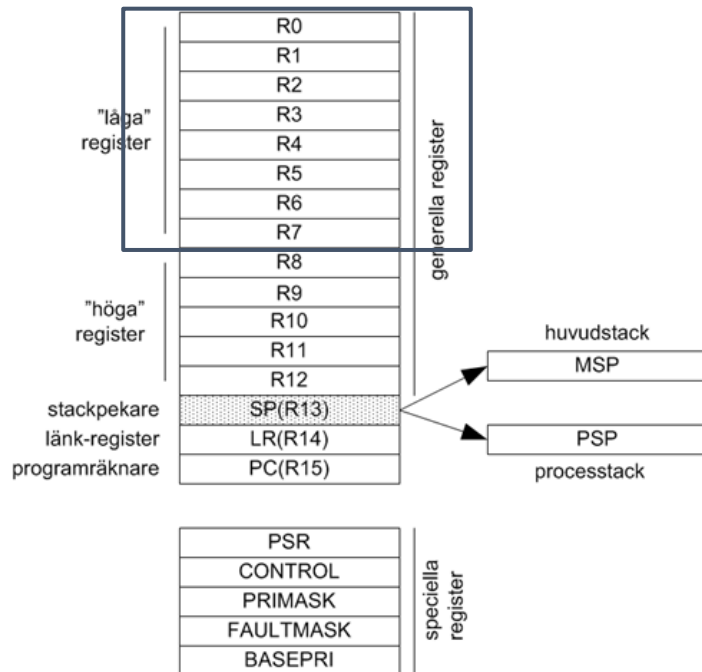
Generella/speciella register, registerstorlek

- ⊕ Vilka operationer kan utföras ?

Instruktionsgrupper

Register och adresseringssätt

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



Exempel:

Med deklarationen: `int i;`

`LDR R0,=i` @ $R0 \leftarrow i$

`LDR R0,[R0]` @ $R0 \leftarrow M(i)$

samma sak som...

`LDR R0,i` @ $R0 \leftarrow M(i)$

FLISP:

`LDX #i` @ $X \leftarrow i$

`LDA ,X` @ $A \leftarrow M(i)$

samma sak som...

`LDA i` @ $A \leftarrow M(i)$

Primära instruktionsgrupper

Load/Store för att läsa från, och skriva till, minnet

Load Till, Från
Store Från, Till

Uttrycksevaluering

Operation Till, Från1{, Från2}

Exempel: C

```
char    c,d;  
  
void foo( void )  
{  
    c = ~d;  
}
```

```
char    c,d;  
  
void foo( void )  
{  
    c = ~d;  
}
```

Programflödeskontroll

B Symbol
Bxx Symbol (xx=villkor)
BX Register

Diverse

MOV Till, Från mellan register
MOV Till, #konstant konstant till register

ARM/Thumb assembler

```
c: .SPACE 1  
d: .SPACE 1  
  
foo:  
    LDR    R0,=d  
    LDRB   R0,[R0]  
    MVN    R1,R0  
    LDR    R0,=c  
    STRB   R1,[R0]  
    BX     LR
```

Instruktioner för tilldelningar

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.

Instruktion	Betydelse	Datatyp
LDRB	Load byte	unsigned char
LDRH	Load halfword	unsigned 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

Exempel: MOV kan ge en liten kodförbättring

```
MOV    R0, #1    @ koda med 16 bitar
```

```
LDR    R0, =1    @ koda totalt med 16+32 bitar
```

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...

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;
```

```
i1:    .SPACE    4  
i2:    .SPACE    4  
c1:    .SPACE    1  
c2:    .SPACE    1  
s1:    .SPACE    2  
s2:    .SPACE    2  
.GLOBL i1,i2,c1,c2,s1,s2
```

```
@ c1 = c2;  
        LDR      R3,=c2  
        LDRB     R3,[R3]  
        LDR      R2,=c1  
        STRB     R3,[R2]
```

```
        LDR      R3,=s2  
        LDRH     R3,[R3]  
        LDR      R2,=s1  
        STRH     R3,[R2]
```

@ i1=i2;

```
        LDR      R3,=i2  
        LDR      R3,[R3]
```

@ eller kortare...

```
        LDR      R3,i2  
  
        LDR      R2,=i1  
        STR      R3,[R2]
```

Datatyper med olika storlekar

Olika datatyper i höger/vänsterled kan kräva typkonvertering

Exempel:

```
signed 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	0xFFFFFFFF80
127	0x7F	0x7F	0x7F

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

Loadinstruktioner med teckenutvidgning

Teckenutvidgningen sker direkt

Unsigned Extend Byte

?	?	?	0XXXXXXXX
00000000	00000000	00000000	0XXXXXXXX

LDRB Rx, [Ry, #k]

?	?	?	1XXXXXXXX
00000000	00000000	00000000	1XXXXXXXX

Signed Extend Byte

LDRSB Rx, [Ry, Rz]

?	?	?	0XXXXXXXX
00000000	00000000	00000000	0XXXXXXXX
?	?	?	1XXXXXXXX
11111111	11111111	11111111	1XXXXXXXX

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 följande
tilldelningssatser kan kodas i
ARM-v6 assemblerspråk:

a) c = s;

b) s = c;

c) i = s;

```
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)

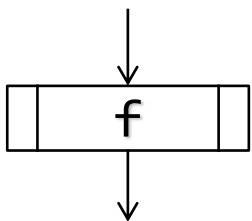
s = c;
LDR    R3,=c           @ adress (&c) till R3
LDRB   R3,[R3]         @ (c) till R3, bit 8-31 nollställs
SXTB   R3,R3           @ teckenutvidgning BYTE->WORD, R3
LDR    R2,=s           @ adress (&s) till R2
STRH   R3,[R2]         @ R3 (halfword) till (s)

i = s;
LDR    R3,=s           @ adress (&s) till R3
LDRH   R3,[R3]         @ (s) till R3
SXTH   R3,R3           @ teckenutvidgning HALFWORD->WORD, R3
LDR    R2,=i           @ adress (&i) till R2
STR    R3,[R2]         @ R3 (word) till (i)
```

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:
...
    f();
...
```

Exempel: ARM

```
    BL    f
@ returadress -> LR
...
f:
...
    BX    LR
```

Exempel: FLISP

```
    JSR    f
...
f:
...
    RTS
```

Returadressen sparas i register. Snabbt, men klarar ej fler nivåer eller 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.*

Exempel:

Utgå från deklARATIONERNA:

```
char c;  
short s;  
int i;  
koda tilldelningarna  
c = f1();  
s = f2();  
i = f3();
```

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)

Funktionsparametrar

Konventionerna säger att vi använder register R0,R1,R2,R3, (i denna ordning) för parametrar som skickas till en subrutin. Övriga parametrar läggs på stacken (behandlas senare i kursen).

Exempel:

Följande deklARATIONER är gjorda:

```
int    a,b;
```

Visa hur följande funktionsanrop kodas i assemblerspråk:

```
b = isupper(a);
```

```
int isupper (int c)
{
    if ( c >= 'A' && c <= 'Z')
        return 1;
    return 0;
}
```

Assembler:

```
LDR R0, =a
LDR R0, [R0]
BL isupper
LDR R1, =b
STR R0, [R1]
```

Uttrycksevaluering

I ett *uttryck* kan *konstanter*, *variabler*, *funktionsvärden* och *operatorer* ingå.

Exempel:

```
int  a, b, c;  
a = b+c / 4 - ( b>>2 ) * c + foo();
```

Kan också betraktas som "sanningsuttryck", dvs ==0 eller != 0

Exempel:

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

Operatorer

Aritmetik

Basic assignment	a = b
Addition	a + b
Subtraction	a - b
Unary plus (integer promotion)	+a
Unary minus (additive inverse)	-a
Multiplication	a * b
Division	a / b
Modulo (integer remainder)	a % b
Increment	Prefix ++a
	Postfix a++
Decrement	Prefix --a
	Postfix a--

Logik

Logical negation (NOT)	!a
Logical AND	a && b
Logical OR	a b

Bitvis

Bitwise NOT	~a
Bitwise AND	a & b
Bitwise OR	a b
Bitwise XOR	a ^ b
Bitwise left shift	a << b
Bitwise right shift	a >> b

Jämförelse

Equal to	a == b
Not equal to	a != b
Greater than	a > b
Less than	a < b
Greater than or equal to	a >= b
Less than or equal to	a <= b

Sammansättning

Addition assignment	a += b	a = a + b
Subtraction assignment	a -= b	a = a - b
Multiplication assignment	a *= b	a = a * b
Division assignment	a /= b	a = a / b
Modulo assignment	a %= b	a = a % b
Bitwise AND assignment	a &= b	a = a & b
Bitwise OR assignment	a = b	a = a b
Bitwise XOR assignment	a ^= b	a = a ^ b
Bitwise left shift assignment	a <<= b	a = a << b
Bitwise right shift assignment	a >>= b	a = a >> b

Instruktioner för unära operationer

C-operator	Betydelse	Datatyp	Instruktion
~	bitvis not	signed/unsigned	MVN Rd, Rs
-	negation	signed/unsigned	NEG Rd, Rs

Observera att alla operationer utförs på 32-bitars tal i register
använd `LDRH` för unsigned short och `LDRB` för unsigned char operander
Men om signed (short eller char) används måste vi teckenutvidga före operationen...

Exempel:

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

```
void foo( void )
```

```
{
```

```
    c = -d;
```

```
}
```

```
LDR    R3, =d
LDRB   R3, [R3]
SXTB   R3, R3
NEG     R3, R3
LDR     R2, =c
STRB   R3, [R2]
```

FLISP:

```
c:  RMB 1
```

```
d:  RMB 1
```

```
LDA     d
```

```
COMA
```

```
STA     c
```

Instruktioner för binära operationer

Binära operationer utförs med tre registerfält där samma register kan förekomma i flera fält.

Alla operationer utförs på 32-bitars tal i register:

Instruktion Till, Från1, Från2

C-operator	Betydelse	Datatyp	Instruktion
+	addition	signed/unsigned	ADD Rd, Rs1, Rs2 ADD Rd, Rs, #k där 0<k<8 ADD Rds, #n där 0<n<256
-	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 unsigned	ASR Rd, Rs1, Rs2 LSR Rd, Rs1, Rs2

Exempel:

```
...  
mask = mask << n;  
...
```

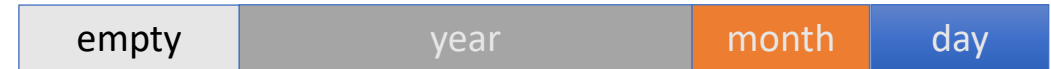
```
R0 ← mask  
R1 ← n  
LSL R0, R0, R1  
mask ← R0
```

Bitvis operationer

Exempel:

Packa samman data i en variabel

Packa (och packa upp) (DAY/MONTH/YEAR) i variabler med typen `int`.



Visa en function: `int pack( int year, int month, int day)` i assembler, som packar tre heltal representerande år, månad och dag, i ett heltal. Det returnerade heltalet ska packas enligt enligt följande figur:

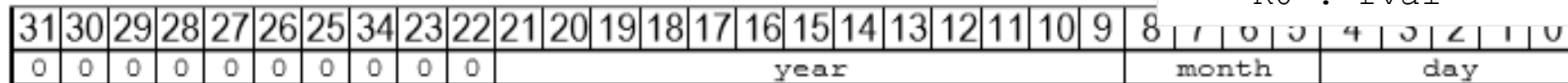
Konventioner:

R0 : year

R1 : month

R2 : day

R0 : rval



```
int pack( int year, int month, int day) {  
    int rval;  
    rval = year << 9;  
    rval |= month << 5;  
    rval |= day;  
    return rval;  
}
```

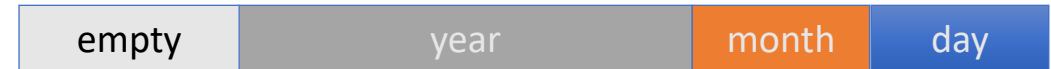
```
pack:  
    MOV    R3, #9  
    LSL    R0, R0, R3    @ R0 ← year << 9  
    MOV    R3, #5  
    LSL    R1, R1, R3  
    ORR    R0, R0, R1    @ R0 ← R0 | month << 5  
    ORR    R0, R0, R2    @ R0 ← R0 | day  
    BX     LR            @ return( rval );
```

Bitvis operationer

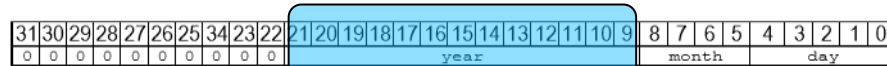
Exempel:

Packa upp data från en variabel

Packa (och packa upp) (DAY/MONTH/YEAR)
i variabler med typen int.



Visa en function: `int unpack_year( int date )` i C och Thumb1 assembler, som packar upp heltalet som representerar år, och returnerar detta.



```
int unpack_year( int date ) {  
    int rval;  
    rval = date >> 9;  
    rval &= 0x1FFF;  
    return rval;  
}
```

Konventioner:

R0 : date

R0 : rval

```
unpack_year:  
    MOV    R2, #9  
    ASR    R0, R0, R2    @ R0 ← date >> 9  
    LDR    R2, =0x1FFF  
    AND    R0, R0, R2  
    BX     LR            @ return( rval );
```

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...

Exempel: Uttrycksevaluering

Exempel:

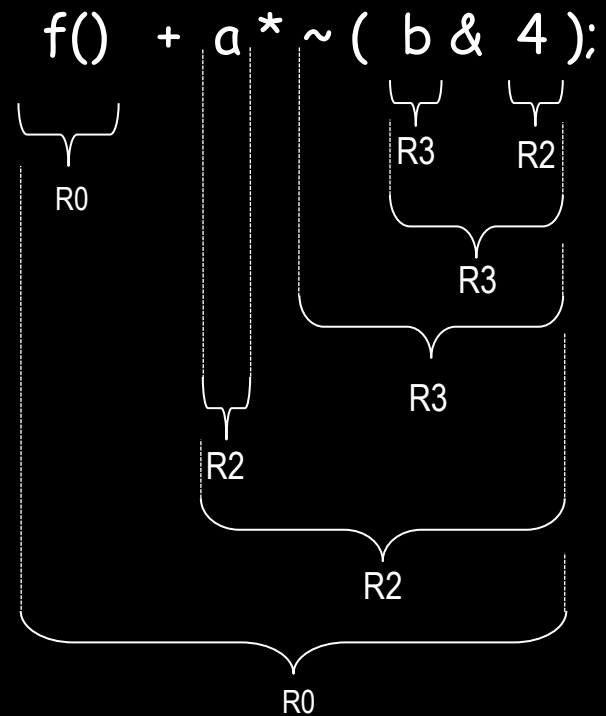
Utgå från att följande deklarationer är gjorda på global nivå.

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

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

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

Vi löser på tavlan...



```
LDR    R3,b
MOV     R2,#4
AND     R3,R3,R2
MVN     R3,R3
LDR     R2,a
MUL     R2,R2,R3
ADD     R0,R0,R2
```