Mera assembler och I/O-hantering

Assemberdirektiv

Simulatorn ARMsim

Syntetiska instruktioner

Stack alignment

Polling och avbrott

Assemblerdirektiv

- Information till assemblern om hur programmet ska översättas
- Information om var olika delar ska placeras i minnet, t ex
- Genererar ingen exekverbar kod
- Börjar med punkt

Vanliga direktiv

- **`.global`** - gör en symbol känd utanför filen
- **`.extern`** – label deklarerad i en annan fil
- **`.text`** - textsegment (programkod, instruktioner)
- **`.data`** – datasegment
- **`.end`** - slut på programfilen

Särskilda direktiv för data

- **`.word`** - allokerar ett ord (32 bitar) i minnet och ger det värdet som anges

Exempel

```
variabel1:  .word  12
```

- **`.byte`** - allokerar en byte (8 bitar) i minnet och ger det värdet som anges

Exempel

```
tecken: .byte  'f'
```

- **`.ascii`** – allokerar plats för och fyller en sträng med angivet innehåll

Exempel

```
str: .ascii "Hello world\n"
```

- **`.asciz`** – allokerar plats för och fyller en sträng med angivet innehåll samt terminerar den med nulltecknet (ascii-kod 0)

Exempel

```
str: .asciz "Hello world"
```

- **`.space`** – allokerar plats angivet antal byte

Exempel

```
buff:   .space 80
```

Simulatorn ARM-sim

Hjälpmedel om man vill öva hemma men inte har någon PI

- Stöjder inte hela ARMv6 (är ARMv5)
  - **PUSH** och **POP** finns inte
- Skriv kod i editor och ladda in i ARMsim
- Länk till nedladdningssida för programmet finns på It's.

Ekvivalenta instruktioner för **PUSH** och **POP**

STore in Memory Decrement Before, **sp** är pekaren

```
STMDB   sp!, {r4, lr}  /* PUSH */
```

LoaD from Memory Increment After, **sp** är pekaren

```
LDMIA   sp!, {r4, pc}     /* POP */
```

Stack alignment

- Konvention: 8 bytes stack alignment

- När man programmerar under ett OS kan det underlätta om man behandlar main också som en funktion, dvs

```
main:
    PUSH {lr}
:
:
    POP {pc}
```

Pseudo-instruktioner, syntetiska instruktioner

- ARM är en RISC-processor med mycket begränsad instruktionsrepertoar

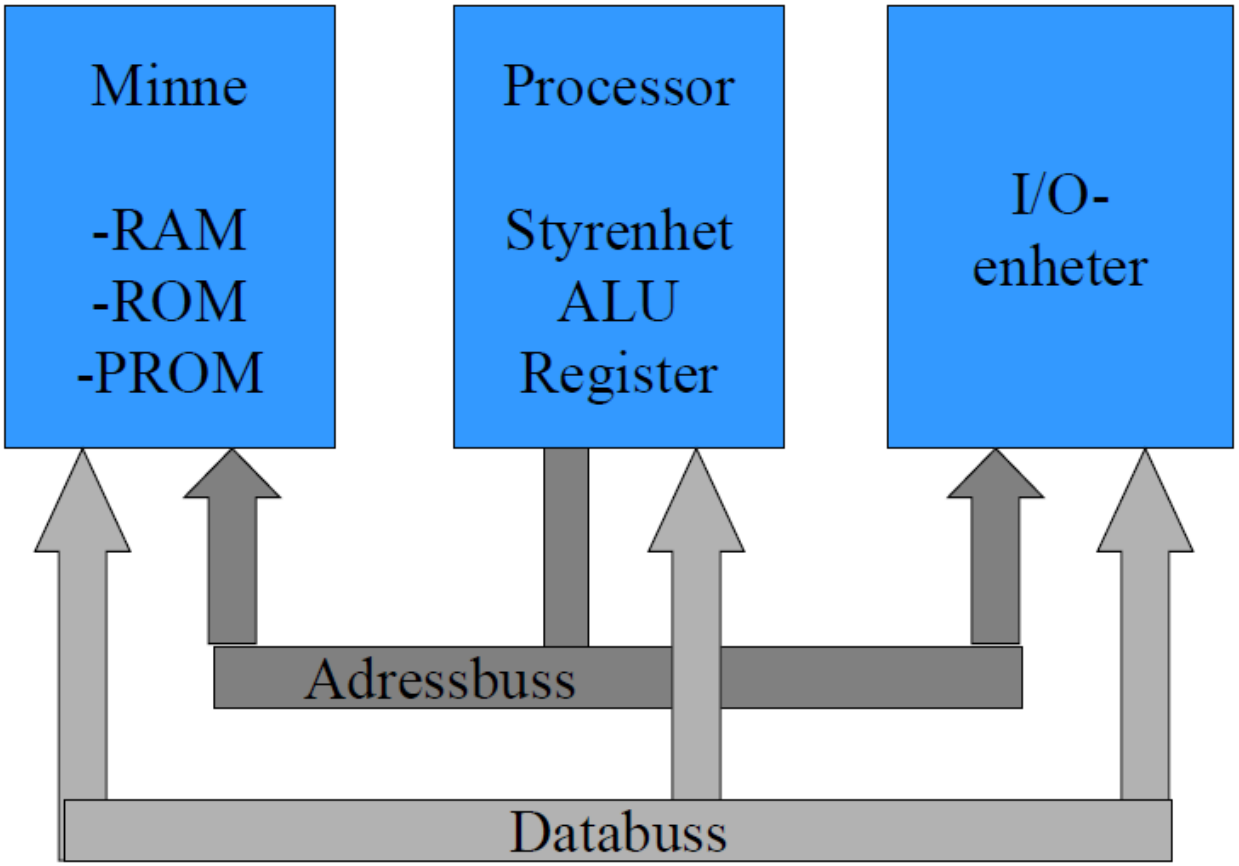
- Vissa instruktioner översätts av assemblern till en eller flera andra maskin-instruktioner

Exempel:
Eftersom instruktionerna är 32 bitar kan inte stora ”immediate” tal läggas med i instruktionen, men man kan ändå skriva det i koden

`LDR r3,=49500` översätts till **`LDR r3, [pc, #offset]`**
(Konstanten läggs på pc+offset i minnet)

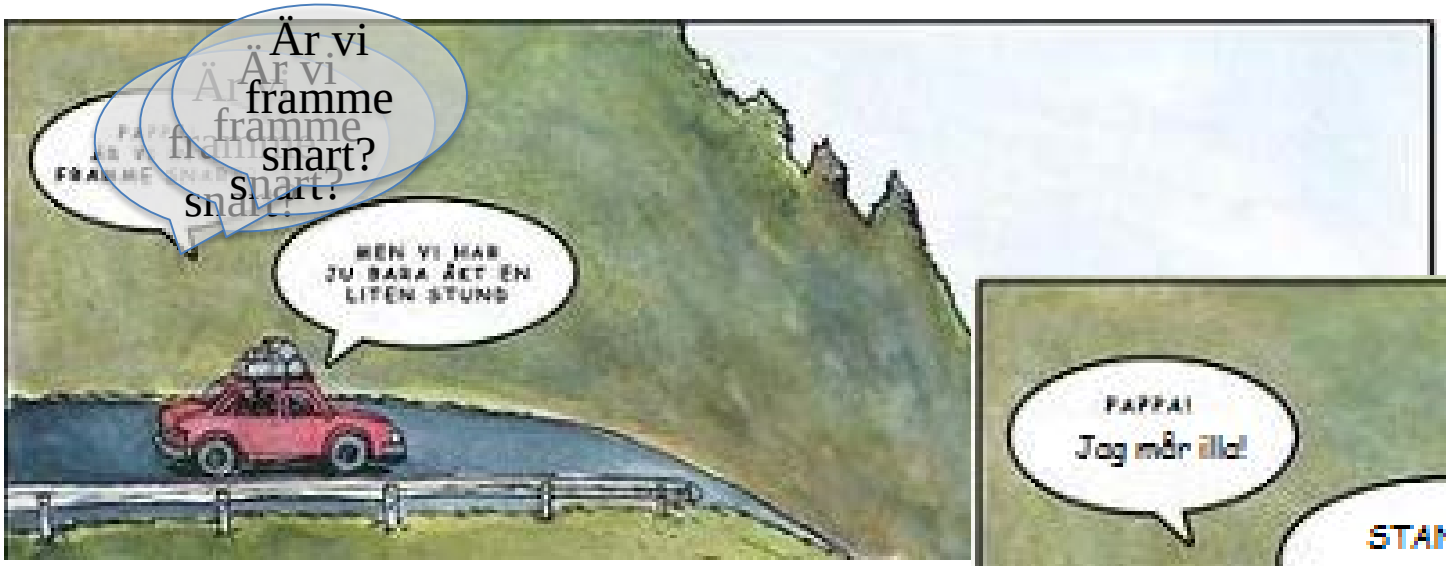
`LDR r3,=200` översätts däremot till **`MOV r3, #200`**

I/0-hantering



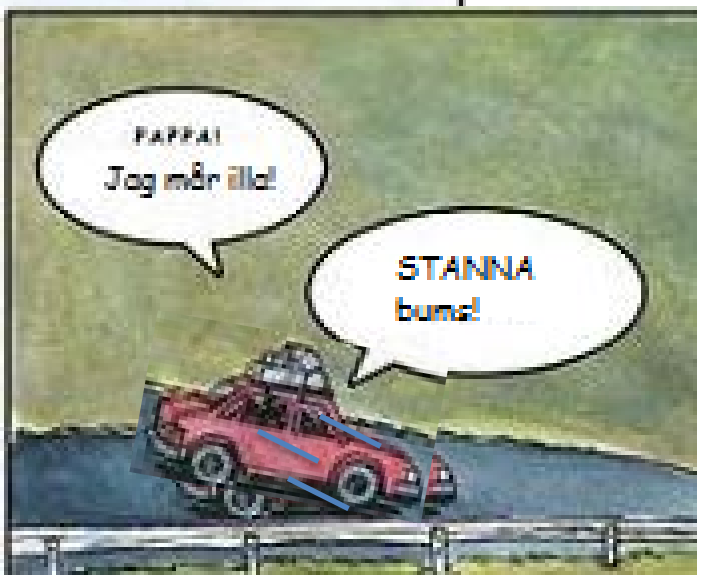
Polling och avbrott

Olika metoder för I/O-enheter att påkalla uppmärksamhet



Polling

Avbrott



Programmerad I/O –Polling

- Sker genom att programmet låter CPUn läsa en viss bit hos enheten med jämna mellanrum (upprepad förfrågning)

Fördel:

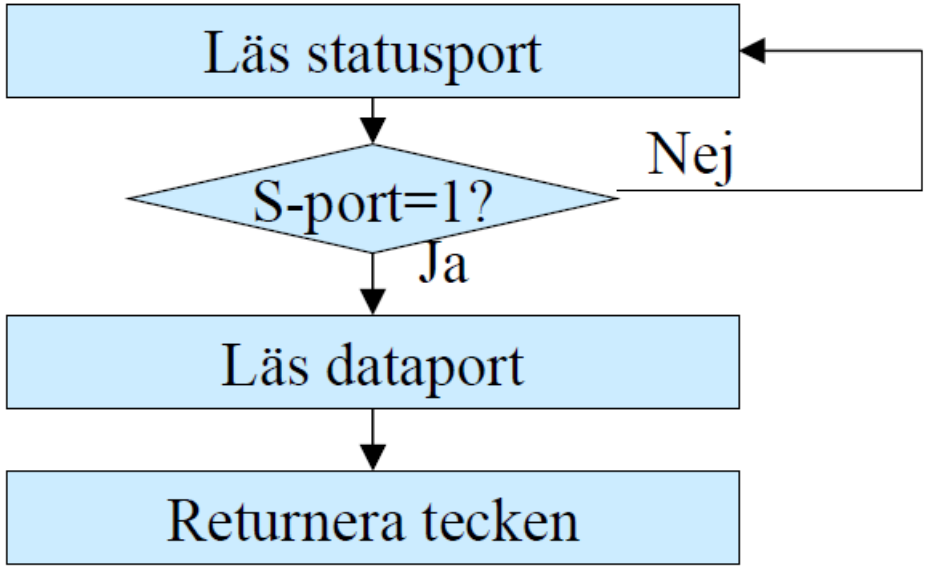
Metoden är enkel och ger ett predikterbart beteende

Nackdel:

Förfrågan måste ske ofta för god tillförlitlighet $\Rightarrow$ olämpligt för tidskritiska tillämpningar

Polling - Exempel

Tangentbord med en statusport och en dataport.



Adressering av hårdvara

- Hårdvara för I/O-enheter är kopplade till minnesadressrymden, och kan adresseras (läsas/skrivas) som vilket minne som helst från mjukvara
- Oftast endast tillgängliga i privilegierad mode om systemet har OS (under avbrott)
- För att hålla ned antalet register i hårdvaran är informationen ofta mycket komprimerad
- Varje enskild bit i registren har ofta en helt egen betydelse

Bitmaskning

- För att få tag på information om enskild bit(ar) ur en byte (word) kan bitmasker användas.

- Exempel1: Kolla om bit 2 i **`r0`** är satt (mask `0x04`)

Mask

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0	0	0	0	0	1	0	0

```
MOV     r4, #0x04
ANDS    r4, r0, r4 /* blir noll om biten inte är satt i r0 */
BNE …           /* vidtag lämplig åtgärd om biten var satt */
…
```

Bitmaskning, forts.

- Exempel 2: Nollställ enbart bit 3 i **`r0`**
  (låt övriga vara oförändrade)

(mask **`0x`**FFFFFF**`F7`**)

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Mask	1	1	1	1	0	1	1	1

(+ resten av bitarna ettor)

```
LDR         r4, =0xFFFFFFF7
AND         r0, r0, r4   /* sätter bit 3 till 0 och lämnar de övriga
                            oförändrade */
```

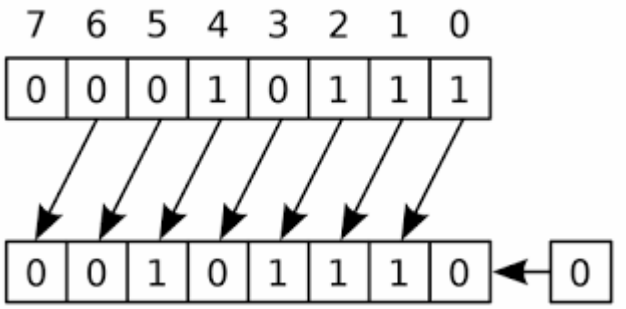
Funderare!

Vi vill ettställa bit 4 i ett register och lämna de övriga bitarna oförändrade.

- Vilken bitmask ska vi använda?
- Vilken logisk operation ska vi använda?

Bitskift

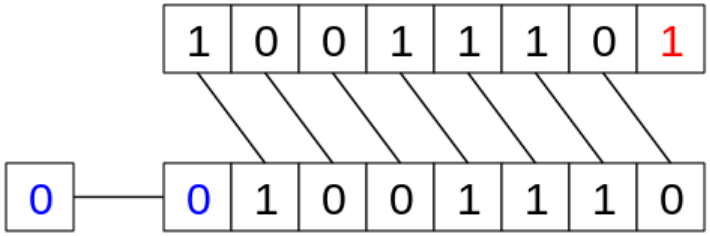
- Finns särskilda logiska instruktioner för bitskift
- Logiska bitskift är särskilt bekväma för att utföra multiplikation eller division med en 2-potens. Nollor skiftas in som nya bitar.



Talet 23 vänsterskiftas 1 steg

Resultat: 46 (multiplikation med 2)

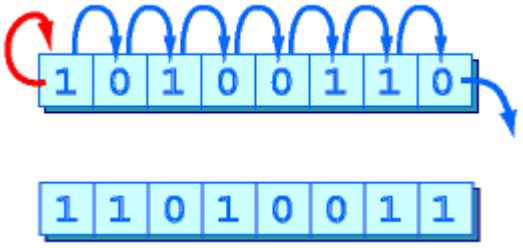
Bitskift, forts.



Högerskifta talet 157

Resultat: 78 (heltalsdivision med 2)

- Finns även *aritmetiskt högerskift*, som är teckenbevarande (dvs kopia av MSB skiftas in)



Högerskifta talet -90

Resultat: -45 (heltalsdivision med 2)

ARM-instruktioner för bitskift

ARM-instruktion	*Förklaring*
`LSL rd, rm, rn`	Vänsterskiftar **`rm`** antal bitpositioner som **`rn`** anger, resultat i **`rd`**
`LSL rd, rm, #steps`	Vänsterskiftar **`rm`** antal bitpos som **`steps`** anger, resultat i **`rd`**
`LSR rd, rm, rn`	Högerskiftar **`rm`** antal bitpositioner som **`rn`** anger, resultat i **`rd`**
`LSR rd, rm, #steps`	Högerskiftar **`rm`** antal bitpos som **`steps`** anger, resultat i **`rd`**
`ASR rd, rm, rn`	Teckenbevarande aritmetiskt högerskift
`ASR rd, rm, #steps`	Teckenbevarande aritmetiskt högerskift

Bitskift som tillägg i annan instruktion

ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}

Exempel:
Hantera en array av word
Ett word = 4 byte
Flytta pekare index*4,
Motsvarar vänsterskift 2 steg

```
    .data
numbers:
    .word  2, 5, 8, 3, 9, 12, 0
sum:
    .space  4

    .text
    .global main
main:
    LDR         r1, =numbers
    MOV         r0, #0
    MOV         r3, #0          /* index counter */
again:
    LDR         r2,[r1, r3, LSL #2]
    CMP         r2, #0
    BEQ         finish
    ADD         r0, r0, r2
    ADD         r3, r3, #1
    B           again
finish:
    LDR         r1, =sum
    STR         r0, [r1]
halt:
    BAL         halt
    .end
```

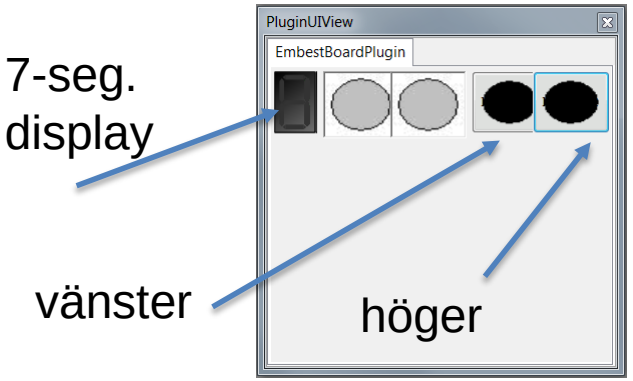
Simulatorn har inga skift som “stand-alone instructions” utan bara som tillägg

Polling - Exempel

- Skriv ett program som utför vissa beräkningar och samtidigt pollar knappenheten för att se om någon knapp tryckts ned.

- Vid knapptryckning på höger knapp ska displaysiffran räknas upp, vid tryck på vänster knapp ska den räknas ned

- I simulatorn görs I/O-operationer med hjälp av **`SWI`**



Kodexempel, polling

```
/* .equ works like #define in C */

    .equ SWI_SETSEG8, 0x200      /* display on 8 Segment */
    .equ SWI_SETLED, 0x201       /* LEDs on/off */
    .equ SWI_CheckBlack, 0x202   /* check Black button, returns 0 if no
                                    pressed,  1, if right pressed, 2 if
                                    left pressed */


/* Patterns for 8 segment display:
   byte values for each segment of the 8 segment display */
    .equ SEG_A, 0x80
    .equ SEG_B, 0x40
    .equ SEG_C, 0x20
    .equ SEG_D, 0x08
    .equ SEG_E, 0x04
    .equ SEG_F, 0x02
    .equ SEG_G, 0x01

/* bit patterns for black buttons */
    .equ LEFT_BLACK_BUTTON,0x02
    .equ RIGHT_BLACK_BUTTON,0x01
```

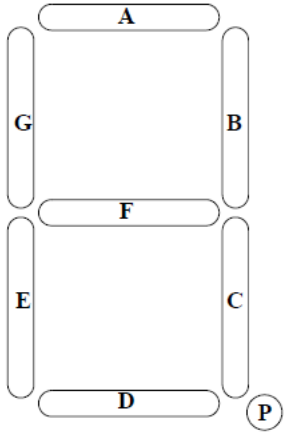


Table 6: Segment byte values

Display byte values	
A	0x80
B	0x40
C	0x20
P	0x10
D	0x08
E	0x04
F	0x02
G	0x01

Kodexempel polling, forts

```
    .data
digits:
    .word SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_G  /* 0 */
    .word SEG_B | SEG_C                                  /* 1 */
    .word SEG_A | SEG_B | SEG_F | SEG_E | SEG_D          /* 2 */
    .word SEG_A | SEG_B | SEG_F | SEG_C | SEG_D          /* 3 */
    .word SEG_G | SEG_F | SEG_B | SEG_C                  /* 4 */
    .word SEG_A | SEG_G | SEG_F | SEG_C | SEG_D          /* 5 */
    .word SEG_A | SEG_G | SEG_F | SEG_E | SEG_D | SEG_C  /* 6 */
    .word SEG_A | SEG_B | SEG_C                          /* 7 */
    .word SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G /* 8 */
    .word SEG_A | SEG_B | SEG_F | SEG_G | SEG_C          /* 9 */
    .word 0                                      /* Blank display */

    .end
```

Kodexempel, polling forts.

```
cntUp9:
    ADD     r0, r0, #1
    CMP     r0, #10
    BNE     lCntUp9End
    MOV     r0, #0
lCntUp9End:
    BX      lr


cntDown9:
    SUB     r0, r0, #1
    CMP     r0, #-1
    BNE     lCntDown9End
    MOV     r0, #9
lCntDown9End:
    BX      lr
```

Kodexempel, polling forts.

```
display7Segment:
   STMDB    sp!, {r0, r1, lr}      /* PUSH {r0, r1, lr} */
   LDR      r1, =digits
   LDR      r0, [r1, r0, LSL #2]
   SWI      SWI_SETSEG8
   LDMIA    sp!, {r0, r1, pc}  /* POP {r0, r1, pc} */
```

```
compute:
   LDR      r0, =0x1fffff  /* Initialize counter value */
lDelay:
   SUBS     r0, r0, #1 /* Decrease counter by 1 */
   BNE      lDelay          /* Test if ready   */
   BX       LR          /*  Return to polling loop */
```

Kodexempel polling, forts.

```
.global main
main:
   MOV   r4, #0       /* Keep counter in r4 */
lLoop:
   MOV   r0, r4
   BL    display7Segment
   BL    compute
   SWI   SWI_CheckBlack   /* Read black buttons */
   CMP   r0, #0       /* 0 means no button pressed, loop */
   BEQ   lLoop
   CMP   r0, #RIGHT_BLACK_BUTTON
   BEQ   lRight
lLeft:
   MOV   r0, r4       /* move counter value to arg. reg. */
   BL    cntDown9     /* update counter */
   MOV   r4, r0       /* move updated counter value back */
   BAL   lLoop
lRight:
   MOV   r0, r4       /* move counter value to arg. reg. */
   BL    cntUp9       /* update counter */
   MOV   r4, r0       /* move updated counter value back */
   BAL   lLoop
```