Datorteknik TSEA82 + TSEA57
Fö6

Stacken

LINKÖPINGS UNIVERSITET

Datorteknik Fö6 : Agenda

- Repetition
- Stacken
- Lab 2, lite tips
- Tid för frågor

Repetition

Adresseringsmoder

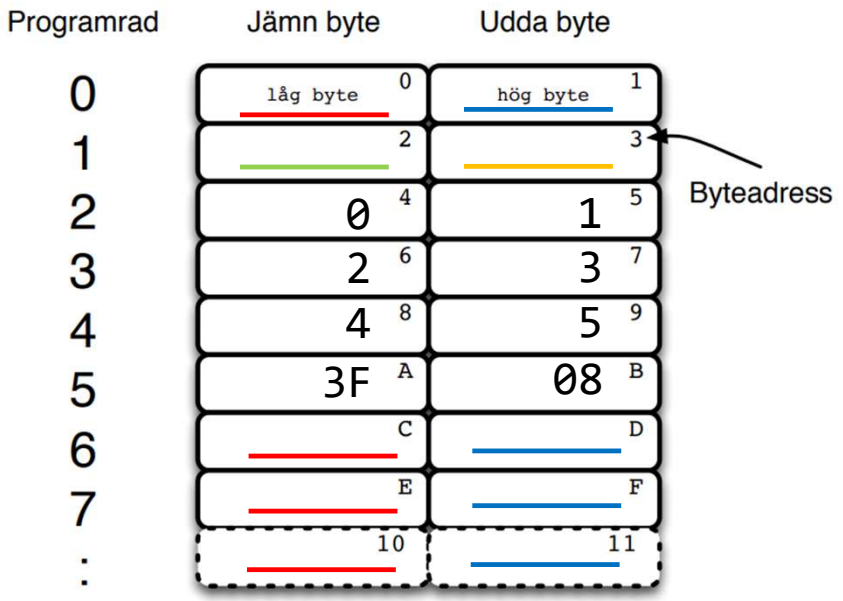
AVR-mikrokontrollern kan hantera följande adresseringsmoder

Mod	Exempelkod			
1.Omedelbar	`subi`	`r20,$12`		
2.Register direkt	`com`	`r16`	`add`	`r20,r21`
3.Data direkt (Absolut)	`lds`	`r20,$A3`	`sts`	`$A4,r20`
4.Data indirekt (Indirekt)	`ld`	`r20,X`	`st`	`Y,r16`
- med förskjutning (offset)	`ldd`	`r20,Y+$05`	`std`	`Z+3,r20`
- med post-inkrement	`ld`	`r20,X+`	`st`	`Y+,r16`
- med pre-dekrement	`ld`	`r20,-X`	`st`	`-Y,r16`

Tabeller i flash-minnet

Med instruktionen lpm och Z-pekaren kan man peka ut enstaka byte i Flash-minnet.
Innehållet i Z utgör den effektiva adressen EA.

```
    .org    0x0000
    jmp     LOOKUP
TAB:
    .db     0,1,2,3,4,5     ; jämnt antal byte
    .dw     0x083F
LOOKUP:
    ldi     ZH,HIGH(TAB*2) ; ladda tabellstart
    ldi     ZL,LOW(TAB*2)
    lpm     r16,Z           ; hämta 0
    call    PROCESS         ; gör nåt ...
    adiw    ZH:ZL,1         ; peka ut nästa
    lpm     r16,Z           ; hämta 1
    call    PROCESS         ; gör nåt
    ...                     ; osv
```



Angående load/store, pekare och minnen

Load och store-instruktionerna har beroende på variant begränsade möjligheter
för att ibland bara använda vissa pekare, och kan bara arbeta mot antingen SRAM (dataminne)
eller Flash (programmine) beroende på instruktion.

Instruktion	Pekare och/eller konstant	Minne
ldi	konstant värde	---
lds, sts	konstant adress	SRAM
ld, st	X,X+,-X, Y,Y+,-Y, Z,Z+,-Z	SRAM
ldd, std	Y+konstant, Z+konstant	SRAM
lpm	Z,Z+	Flash

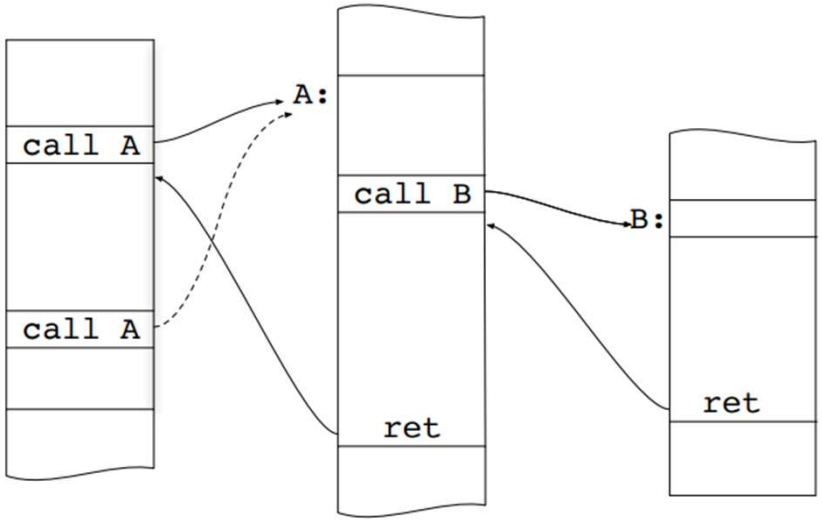
Stacken

Subrutiner

Var gång ett program behöver utföra en uppgift flera gånger
så lönar det sig att skapa en subrutin (ett underprogram) för
den uppgiften, och sedan anropa den subrutinen från de platser
där uppgiften ska utföras.

Det underlättar både skrivandet,
och läsandet, av programkoden.

För att åstadkomma det behövs tre saker:
-Programflödet behöver anropa subrutinen.
-Återhoppsadressen måste sparas.
-Subrutinen behöver, när den är färdig, kunna
hoppa tillbaka till återhoppsadressen.



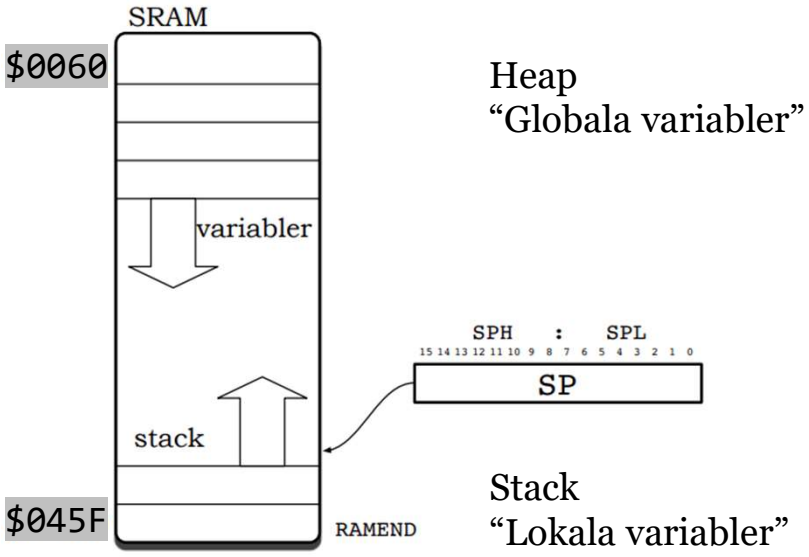
Stacken

För att anropa en subrutin används instruktionen `call` / `rcall` (Absolute Call / Relative Call).
Observera, att man **anropar alltid** en subrutin, man **hoppar aldrig** till en subrutin.

Instruktionerna `call` / `rcall` fungerar som `jmp`, med skillnaden att återhoppsadressen sparas på en stack

Stacken är bara en del av arbetsminnet (SRAM).
Vanligen så används slutet av minnet och stacken växer mot lägre adresser.

En stackpekare, SP, pekar på nästa lediga plats i stacken, och SP räknas ned / upp när något läggs dit / tas bort på stacken.



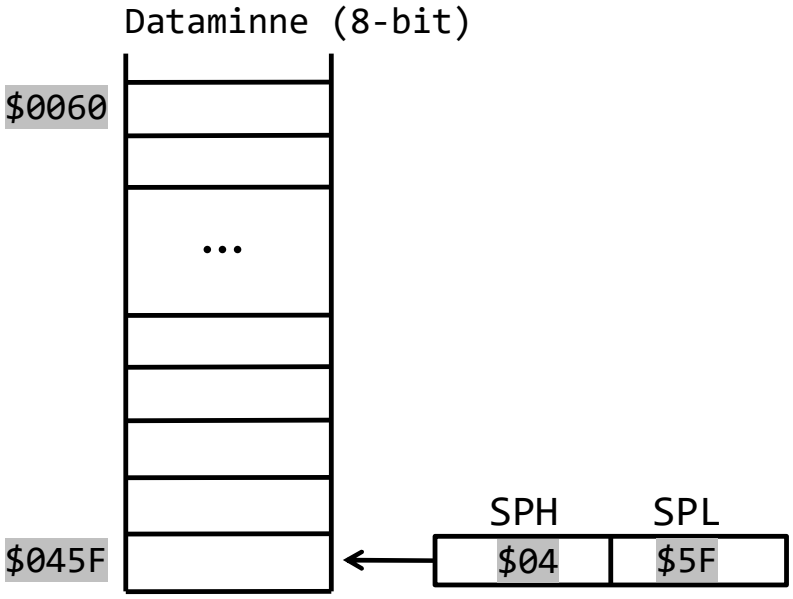
Stacken : initiering

Stacken, eller rättare sagt, stackpekaren SP måste initieras innan användning, vanligen till slutet av minnet (RAMEND) med följande programkod:

```
ldi    r16,HIGH(RAMEND)
out    SPH,r16
ldi    r16,LOW(RAMEND)
out    SPL,r16
```

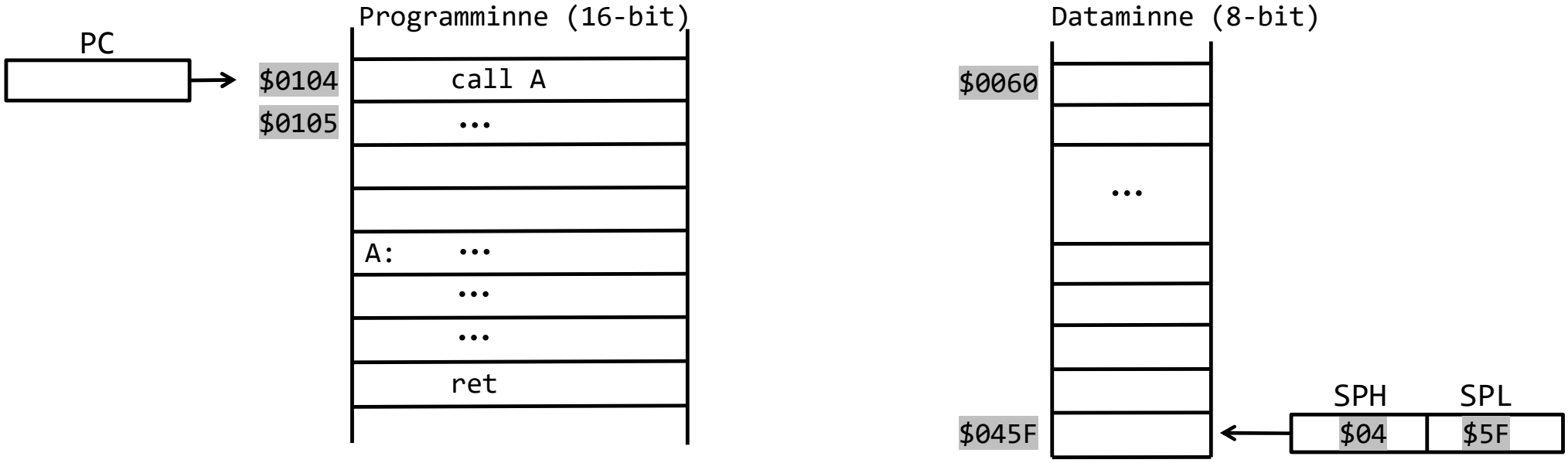
Konstanten RAMEND definieras automatiskt beroende på val av processor. För ATmega16:

```
.equ   RAMEND = $045F
```



Stacken: `call` och `ret`

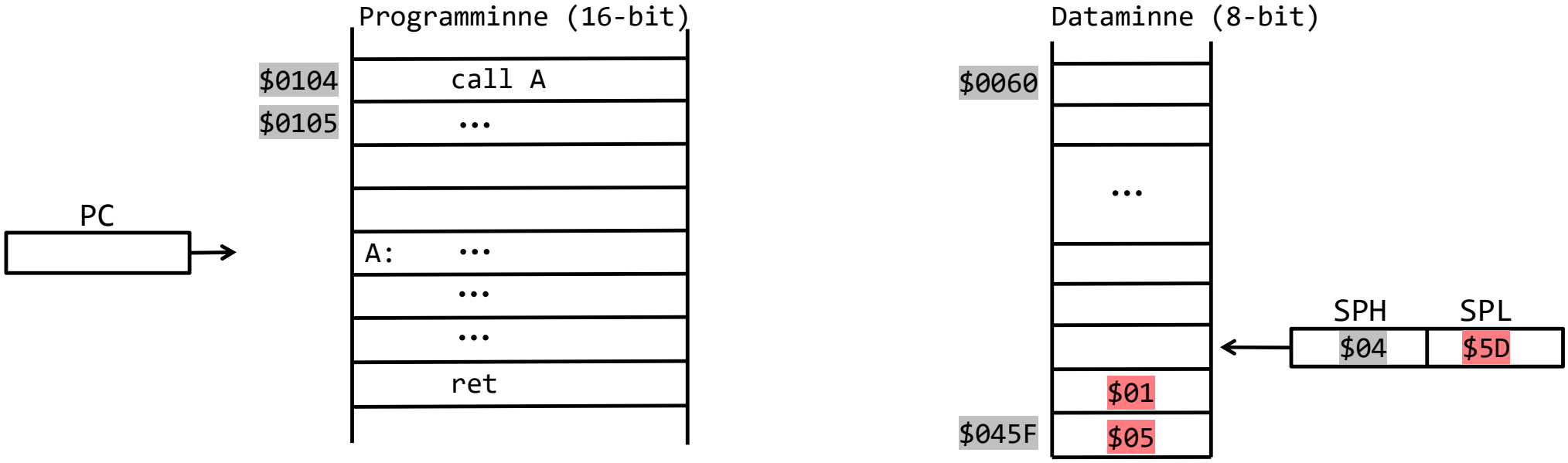
När en subrutin anropas (`call/rcall`) läggs återhoppsadressen automatisk upp på stacken, stackpekaren räknas ned och program-exekvering fortsätter vid subrutinen:



Instruktion	Innebörd
`call` *addr*	$M(SP) := PC + 1, SP := SP - 2, PC := addr$
`ret`	$SP := SP + 2, PC := M(SP)$

Stacken: `call` och `ret`

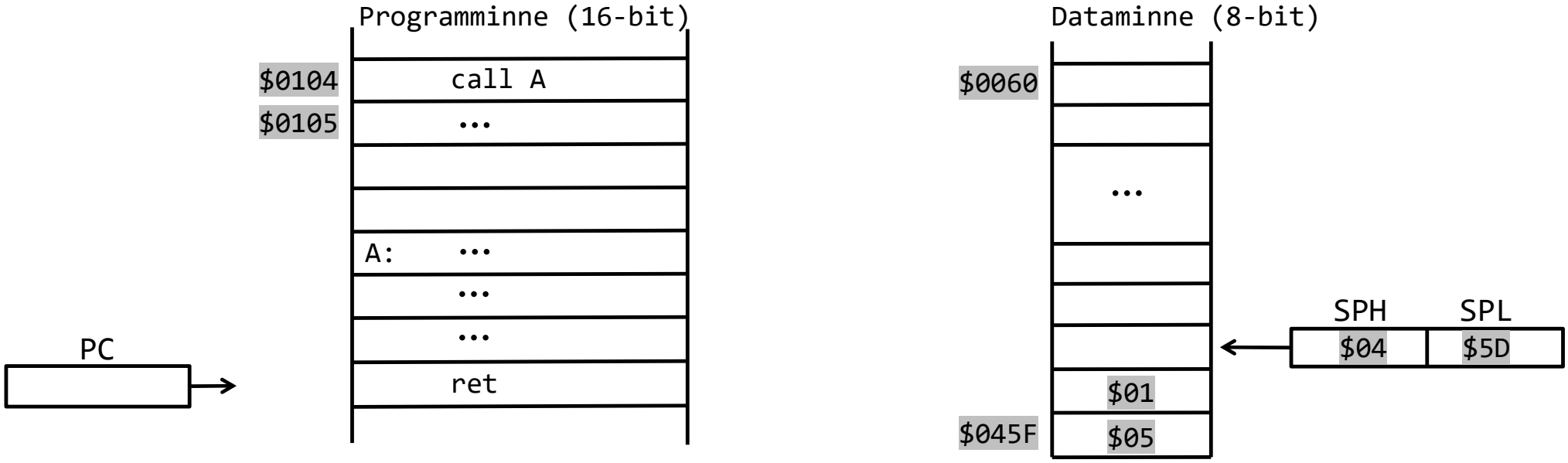
När en subrutin anropas (`call/rcall`) läggs återhoppsadressen automatisk upp på stacken, stackpekaren räknas ned och program-exekvering fortsätter vid subrutinen:



Instruktion	Innebörd
`call` *addr*	$M(SP) := PC + 1, SP := SP - 2, PC := addr$
`ret`	$SP := SP + 2, PC := M(SP)$

Stacken: `call` och `ret`

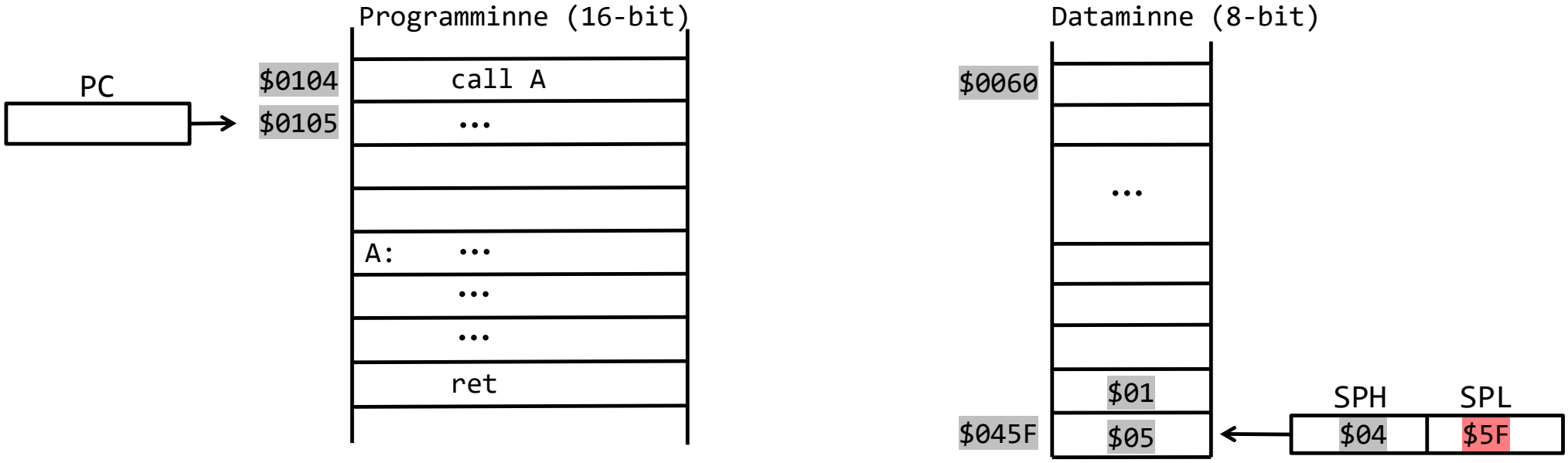
När en subrutinen avslutas returnerar den med instruktionen `ret`, varpå stackpekaren räknas upp och programräknaren PC sätts till återhoppsadressen från stacken:



Instruktion	Innebörd
`call` *addr*	$M(SP) := PC + 1, SP := SP - 2, PC := addr$
`ret`	$SP := SP + 2, PC := M(SP)$

Stacken: `call` och `ret`

När en subrutinen avslutas returnerar den med instruktionen `ret`, varpå stackpekaren räknas upp och programräknaren PC sätts till återhoppsadressen från stacken:



Instruktion	Innebörd
`call` *addr*	$M(SP) := PC + 1, SP := SP - 2, PC := addr$
`ret`	$SP := SP + 2, PC := M(SP)$

Observera, innehållet på stacken förändras inte vid läsning, men betraktas nu som inaktuellt.

Stacken : push och pop

Instruktionerna `call` och `ret` påverkar alltså automatiskt innehållet på stacken, förändrar stackpekarens (`SP`) värde och styr programflödet genom att programräknarens (`PC`) värde läggs upp på stacken vid `call` och återhämtas vid `ret`.

Man kan även påverka stacken manuellt genom lägga dit eller hämta värden, via instruktionerna `push` och `pop`. Stackpekaren `SP` uppdateras automatiskt.

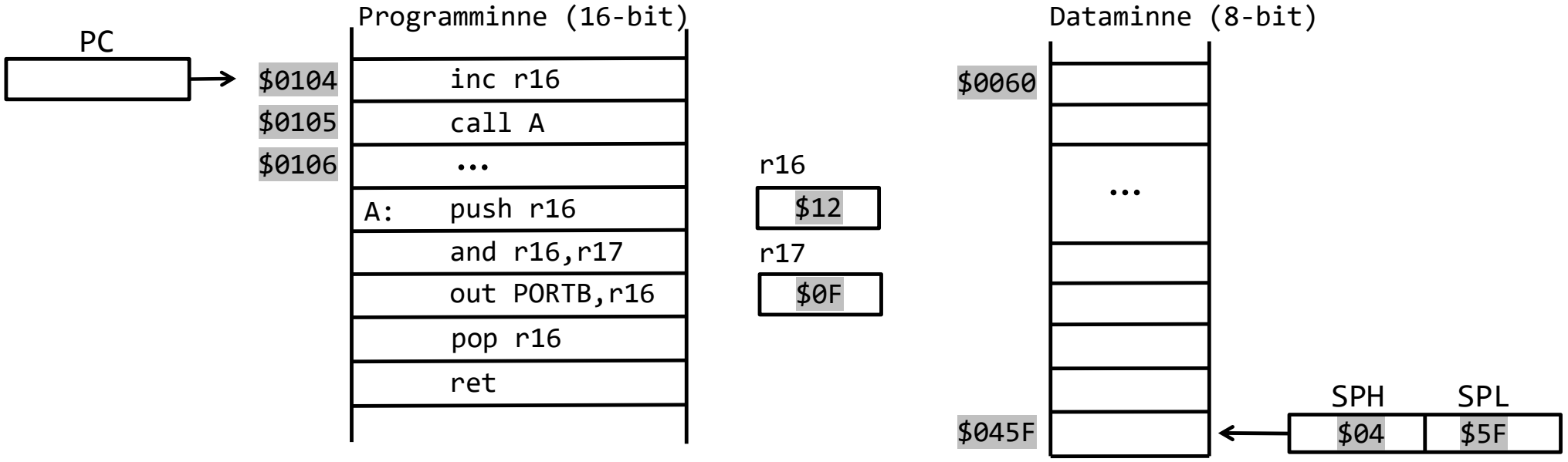
Tex:

```
push    r16     ; värdet i r16 läggs på stacken, därefter minskas SP med 1
pop     r22     ; SP ökas med 1, därefter hämtas (kopieras) värdet på stacken till r22
```


Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

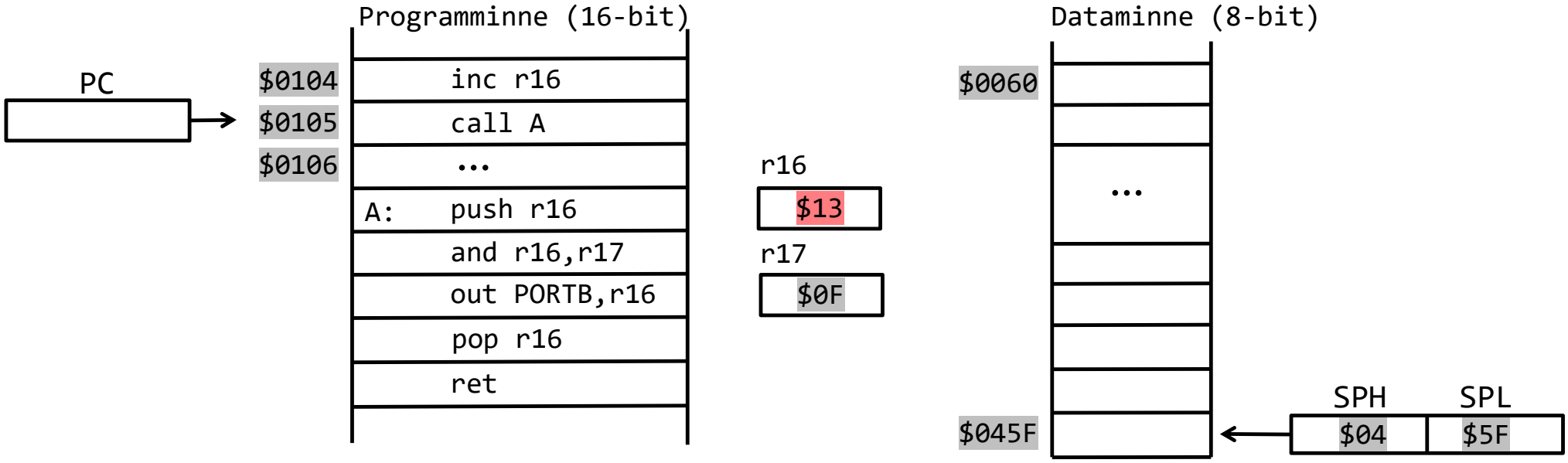
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

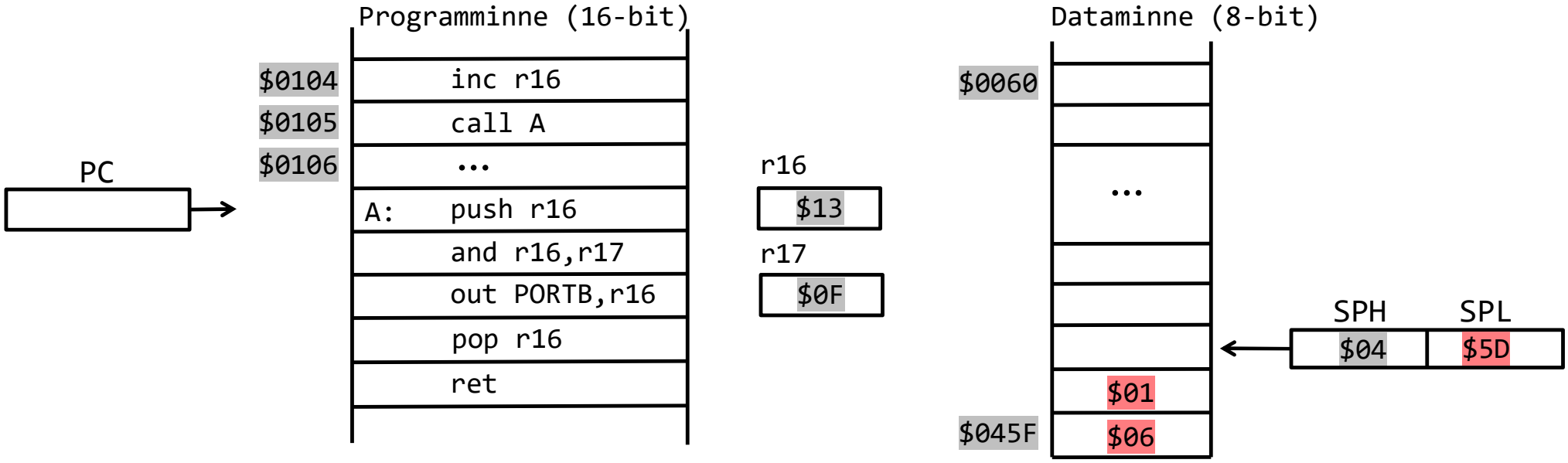
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

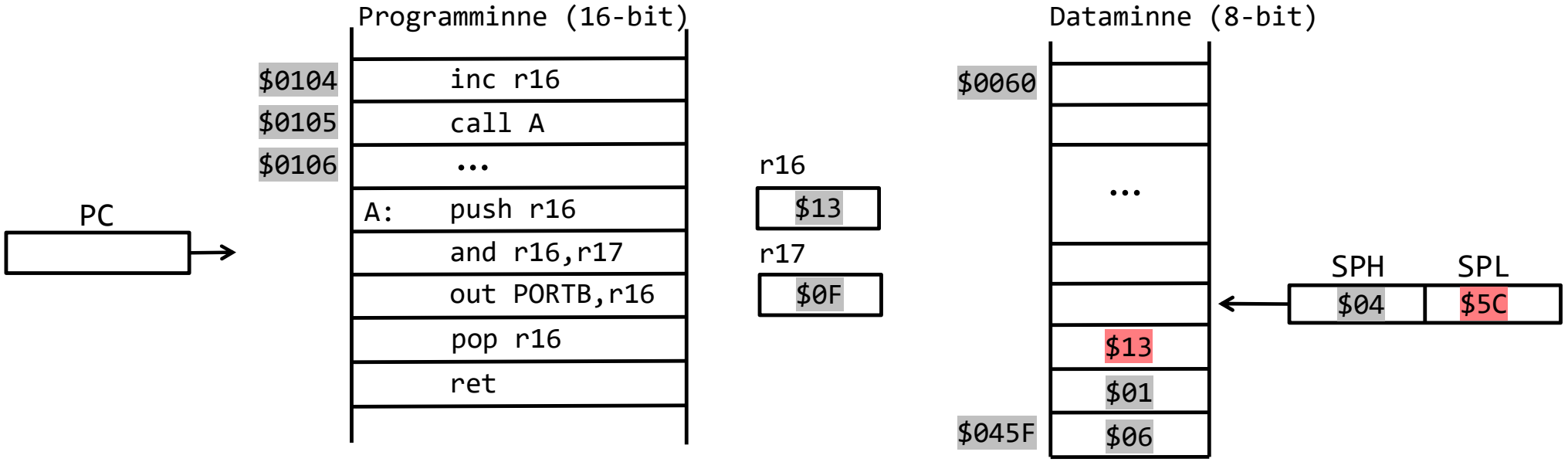
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

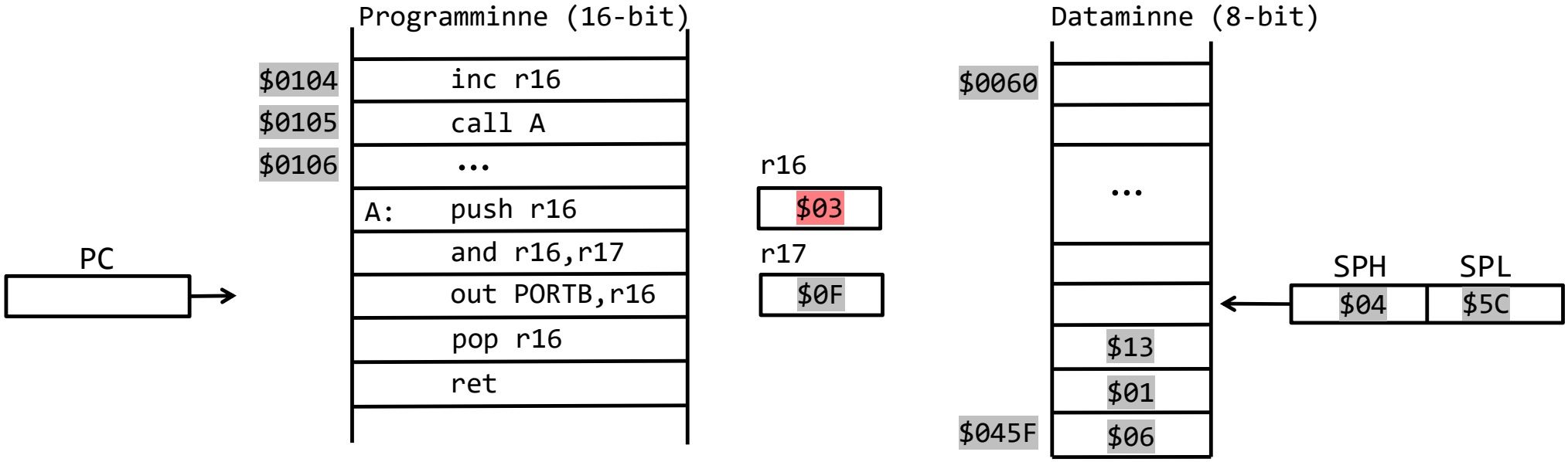
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

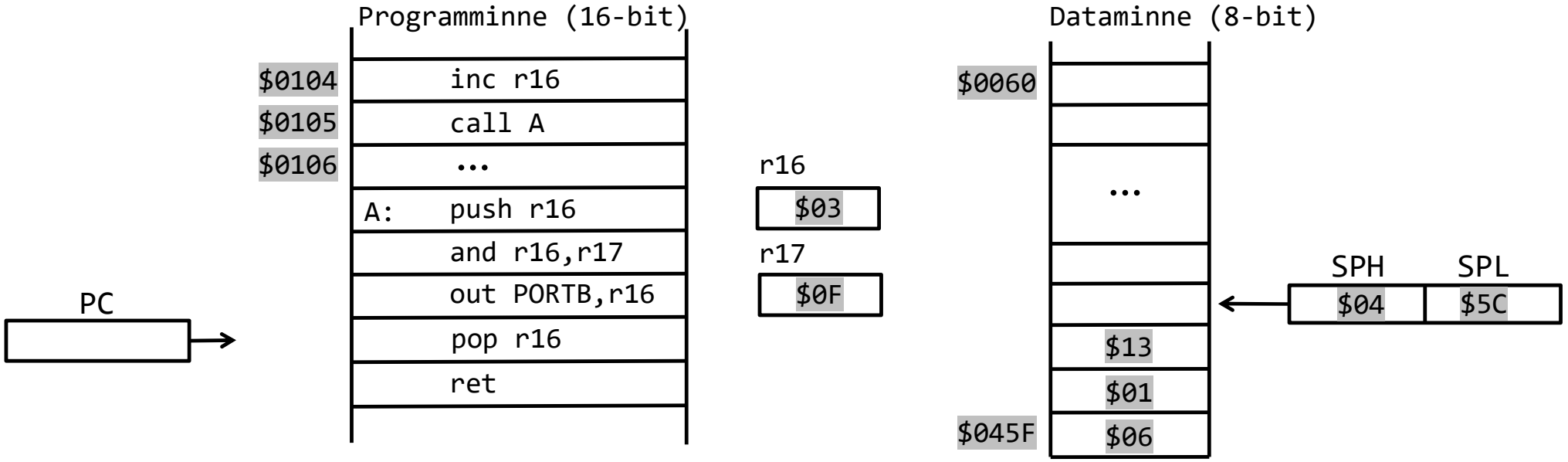
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

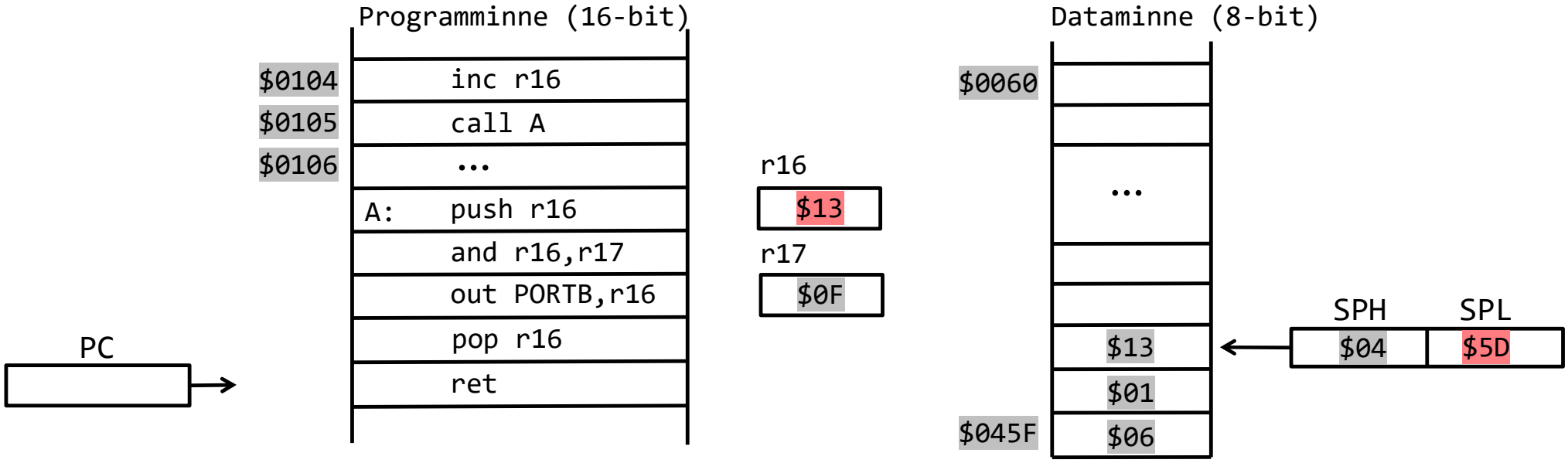
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

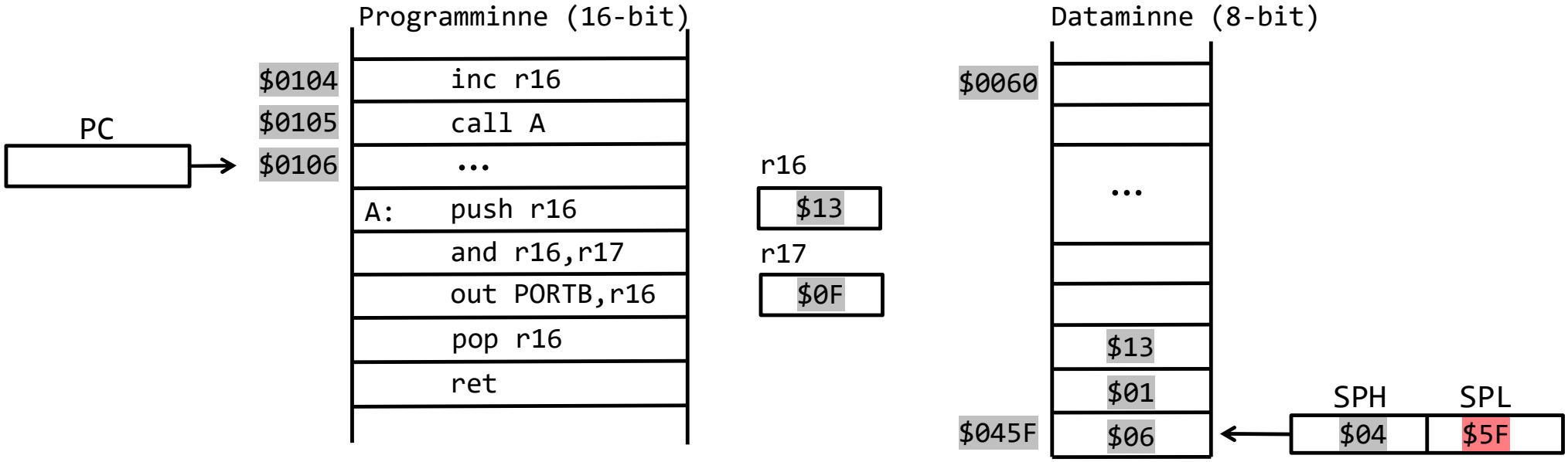
Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

Stacken : push och pop, lokala register

Det är god programmeringssed att skydda kontexten i ett program. Dvs de register som en subrutin förändrar sparas på stacken. I subrutinen får man då ”lokala” register. Innan retur återhämtas sedan kontexten.



Instruktion	Innebörd
push Rr	M(SP):=Rr, SP:=SP-1
pop Rd	SP:=SP+1, Rd:=M(SP)

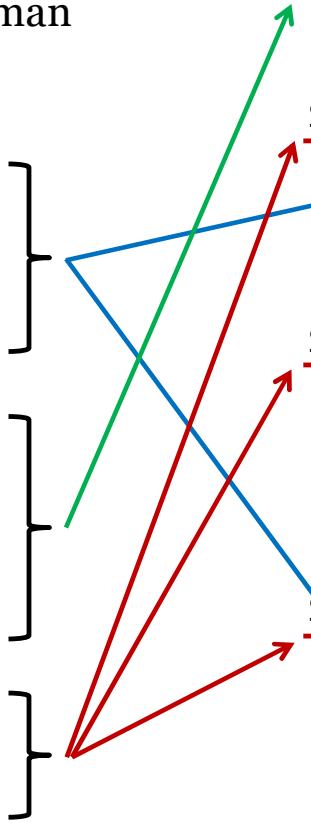
Stacken : subrutiner

Exempel: Gör en subrutin för att beräkna summan av `10` konsekutiva tal, börja med talet i `r17`. Returnera summan i `r17`.

Spara undan kontexten för de ”lokala” register som förändras i subrutinen. Observera ordningen, Det som läggs dit sist på stacken hämtas först.

För att den som läser koden lätt ska förstå vad koden gör och snabbt och enkelt ska veta vilka in- och ut-parametrar som används, skriver man lämpligen ett ”funktionshuvud” i form av en kommentar som talar om detta.

Använd samma stam-namn på alla labels i subrutinen. Det undviker namnkonflikter och blir tydlig att dessa labels tillhör subrutinen.



```
; Sub SUM
; Calculate sum of 10 consecutive numbers
; starting with number in r17
; IN: r17, starting number
; OUT: r17, sum
SUM:
    push    r16     ; save context
    push    r18
    ldi     r16,10  ; set loop index
    clr     r18     ; clear sum
SUM_LOOP:
    add     r18,r17 ; add to sum
    inc     r17
    dec     r16     ; next in loop
    brne    SUM_LOOP
    mov     r17,r18 ; set return value
SUM_EXIT:
    pop     r18     ; restore context
    pop     r16
    ret
```

Stacken : subrutiner

För att beräkna summan av de 10 talen
4+5+6+7+8+9+10+11+12+13
kan man anropa subrutinen med 4 i `r17`:
(parameteröverföring via register)

```
    ldi     r17,4
    call    sum
    ...
```

Resultatet, summan, kommer tillbaka i `r17`,
så därför sparas inte `r17` på stacken.

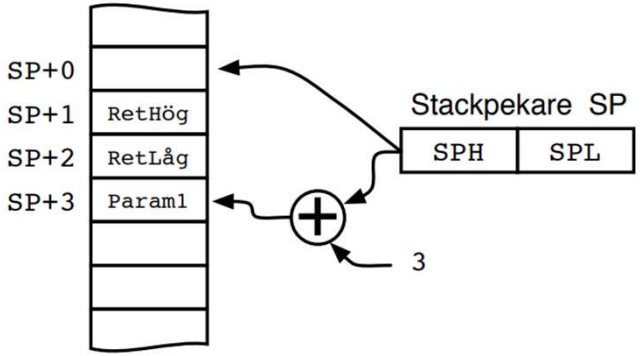
Subrutinen ska bara ha **EN** utgång, som man kan hoppa till från andra delar av subrutinen. Dvs, bara **EN** `ret`-instruktion per subrutin. Om man använder `ret`-instruktionen på flera platser för samma subrutin så måste kontexten återställas på alla dessa platser. Det blir stökigt.

```
; Sub SUM
; Calculate sum of 10 consecutive numbers
; starting with number in r17
; IN: r17, starting number
; OUT: r17, sum
SUM:
    push    r16     ; save context
    push    r18
    ldi     r16,10  ; set loop index
    clr     r18     ; clear sum
SUM_LOOP:
    add     r18,r17 ; add to sum
    inc     r17
    dec     r16     ; next in loop
    brne    SUM_LOOP
    mov     r17,r18 ; set return value
SUM_EXIT:
    pop     r18     ; restore context
    pop     r16
    ret
```

Parameter/Retur-stacken

Om man vill göra en mer generell lösning kan man använda stacken för parameteröverföring.
Här läggs parametern på toppen av stacken:

```
ldi     r25,4
push    r25
call    sum
pop     r25
...
```



Register `r25` används endast temporärt
Resultatet, summan, kommer tillbaka på toppen av stacken och kan sedan överföras till t ex `r25`.

```
; Sub SUM
; Calculate sum of 10 consecutive numbers
; starting with number in parameter stack
; IN: first pos in parameter stack
; OUT: sum in first pos in parameter stack
SUM:
    in      ZH,SPH      ; copy stack pointer
    in      ZL,SPL      ;   to Z
    push    r17         ; get parameter
    ldd     r17,Z+3     ;   from stack
    push    r16         ; save context
    push    r18
    ...
    ...         ; samma som förut
    ...
SUM_EXIT:
    pop     r18         ; restore context
    pop     r16
    std     Z+3,r17     ; store sum in
    pop     r17         ;   return stack
    ret
```

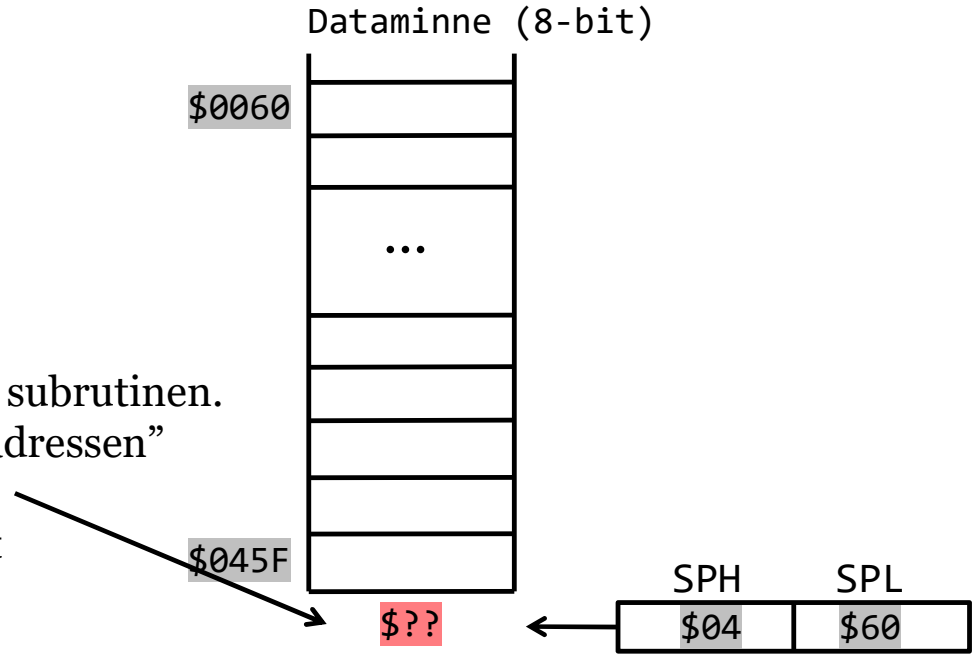
Vanliga misstag med subrutiner/stacken

Observera att en subrutin måste ***alltid anropas*** med instruktionen `call` (eller `rcall`), annars sparas inte återhoppsadressen på stacken.

För vad händer om man gör:

```
    jmp     subrutin            subrutin:
                                  ...
eller t ex                        ...
                                  ret
    brne    subrutin
```

Det går bra ända tills man kommer till `ret`-instruktionen i subrutinen.
Då kommer stackpekaren räknas upp och läsa ”återhoppsadressen”
som inte alls är någon återhoppsadress utan nåt annat.
Som konsekvens kommer programmet att hoppa helt galet
och möjligen krascha.



Vanliga misstag med subrutiner/stacken

Instruktionerna push och pop måste alltid matcha varandra, på så sätt att det är lika många push som pop och att man återhämtar element från stacken i omvänd ordning jämfört med hur dom lades upp.

För vad händer om man gör:

```
push    r16
push    r17
...
pop     r16
pop     r17
ret
```

Jo, `r16` och `r17` har bytt värde med varandra.

Eller så här:

```
push    r16
push    r17
...
pop r17
ret
```

Nu ligger `r16` kvar på stacken. Ska det vara så?

Labb 2

Lite tips

Lab 2, lite tips

I morse-labben ska man i princip göra tabelluppslagning i två tabeller. Dels från en text-sträng, dels från morse-tabellen.

Endast hex-koderna för de olika morse-koderna behöver lagras i morse-tabellen.

Mellanslag, space, kan lättast specialhanteras.

Endast `Z`-pekaren kan användas tillsamman med instruktionen lpm för läsning från programminnet. Eftersom läsning ska ske från två tabeller, gör en lokal `Z` i `LOOKUP`.

Man kan ange ASCII-tecken som argument, t ex `subi r16,'A'`.

```
TEXT:
     .db      "ANDNI ANDNI ANDNI",0
MORSE:
     .db      $60,$88,$A8,$90,$40,...
     ...
     ldi      ZH,HIGH(TEXT*2)
     ldi      ZL,LOW(TEXT*2)
     ...
MAIN_LOOP:
     lpm      r16,Z+
     cpi      r16,' '
     breq     DO_SPACE
     call     LOOKUP
SEND_LOOP:
     lsl      r16
     ...           ; om C=0 => beep(N)
     ...           ; om C=1 => beep(3N)
     ...           ; om fler => hoppa SEND_LOOP
     jmp      MAIN_LOOP
LOOKUP:
     ...           ; push ZH,ZL
     ...           ; ldi Z=MORSE*2
     subi     r16,'A'
     ...           ; Z=Z+r16
     lpm      r16,Z
     ...           ; pop ZL,ZH
     ret
```

Tid för Frågor

Anders Nilsson

www.liu.se

LINKÖPINGS UNIVERSITET