Mera assembler, ARM

Stack och subrutiner

Assemblerdirektiv

Logiska instruktioner

AND (logiskt OCH)

ARM-instruktion	*Förklaring*
`AND rd,rn,rm`	Bitvis logiskt OCH mellan två tal i register
`AND rd,rn,#K`	Bitvis OCH mellan ett tal i register och en konstant

OR (Logiskt ELLER)

ARM-instruktion	*Förklaring*
`ORR rd,rn,rm`	Bitvis logiskt ELLER mellan två tal i register
`ORR rd,rn,#K`	Bitvis ELLER mellan ett tal i register och en konstant

Logiska instruktioner (forts.)

NOT

ARM-instruktion	*Förklaring*
`MVN rd,rn`	Bitvis ICKE Varje bit inverteras (Eg. Move Not)

XOR (exklusivt eller)

ARM-instruktion	*Förklaring*
`EOR rd,rn,rm`	Bitvis XOR, ger etta i bitpos med olika värde
`EOR rd,rn,#K`	

Hopp

Branch, kan vara villkorligt

ARM-instruktion	*Förklaring*
`B<cond> label`	Hoppa till label (om cond uppfyllt)

Olika villkor (condition , 4bitar)

Värde	Kod (Betydelse)	Flaggor	Värde	Kod (Betydelse)	Flaggor
0	**BEQ** (EQual)	Z=1	8	**BHI** (unsigned HIgher)	(C=1)&(Z=0)
1	**BNE** (Not Equal)	Z=0	9	**BLS** (us.Lower or Same)	(C=0)|(Z=1)
2	**BHS** (us. Higher or Same)	C=1	10	**BGE** (s. Greater or Equal)	N=V
3	**BLO** (unsigned LOwer)	C=0	11	**BLT** (s. Less Than)	N!=V
4	**BMI** (Minus, < 0)	N=1	12	**BGT** (s. Greater Than)	(Z=0)&(N=V)
5	**BPL** (PLus, <=0)	N=0	13	**BLE** (s. Less or Equal)	(Z=1)&(N!=V)
6	**BVS** (oVerflow Set)	V=1	14	**BAL** (ALways) som **B**	
7	**BVC** (oVerflow Clear)	V=0	15	Används ej!	

Not: Även **BCC**(=**BLO**) och **BCS**(=**BHS**) funkar

Jämförelse, förberedelse för hopp

- Det finns instruktioner för att jämföra registervärden med varandra eller konstanter.
- Jämförelseinstruktioner genererar inget resultat, men påverkar flaggorna i CPSR
- Efter en jämförelse kan en villkorlig hoppinstruktion exekveras

ARM-instruktion	*Förklaring*
`CMP rn,rm`	Beräknar **`rn`** – **`rm`** och låter resultatet påverka CPU-flaggorna
`CMP rn,#K`	Beräknar **`rn`** – **`K`** och låter resultatet påverka CPU-flaggorna

Exempel med villkorligt hopp

```
loop:
.
.
  CMP   r1, #10   /* Jämför r1 med 10 */
  BNE   loop    /* Upprepa om r1 inte är 10 */
.
.
```

Vårt första assemblerprogram

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

    .text
    .global main
/* This is a comment */
main:
   LDR         r1, =numbers
   MOV         r0, #0
again:
    LDR        r2,[r1]
    CMP        r2, #0
    BEQ        finish
    ADD        r0, r0, r2
    ADD        r1, r1, #4
    B          again
finish:
    LDR        r1, =sum
    STR        r0, [r1]
halt:
    BAL        halt

    .end
```

Hoppinstruktioner och struktur

Exempel 1: Sekvens

```
a = a + b;
c = a – b;
d = a * f;
```

Instruktionerna utförs i ordning

Exempel2: Iteration och selektion

```
a = 0;
while (a<10)
   a = a + 1;
if (b==5)
   c = 10;
else
   c = 0;
d = a * f;
```

Den här instruktionen utförs flera gånger

Beroende på villkoret görs olika saker

Hoppinstruktion forts.

- Nästa instruktion är normalt nästa i programmet

  `PC = PC + 4;`

  Programräknare (PC) räknas upp och nästa instruktion hämtas

- För att göra ett hopp till en annan del av koden, ladda programräknaren med ett nytt värde

  `PC = adress dit hopp ska ske`

Subrutiner (funktioner)

Program:

```
int main()
{
  int a, b, c;
  a = 5;
  b = 6;
  c = myAdd(a,b);
  return 0;
}

int myAdd(int x, int y)
{
  return x+y;
}
```

Exekvering:

a = 5

b = 6

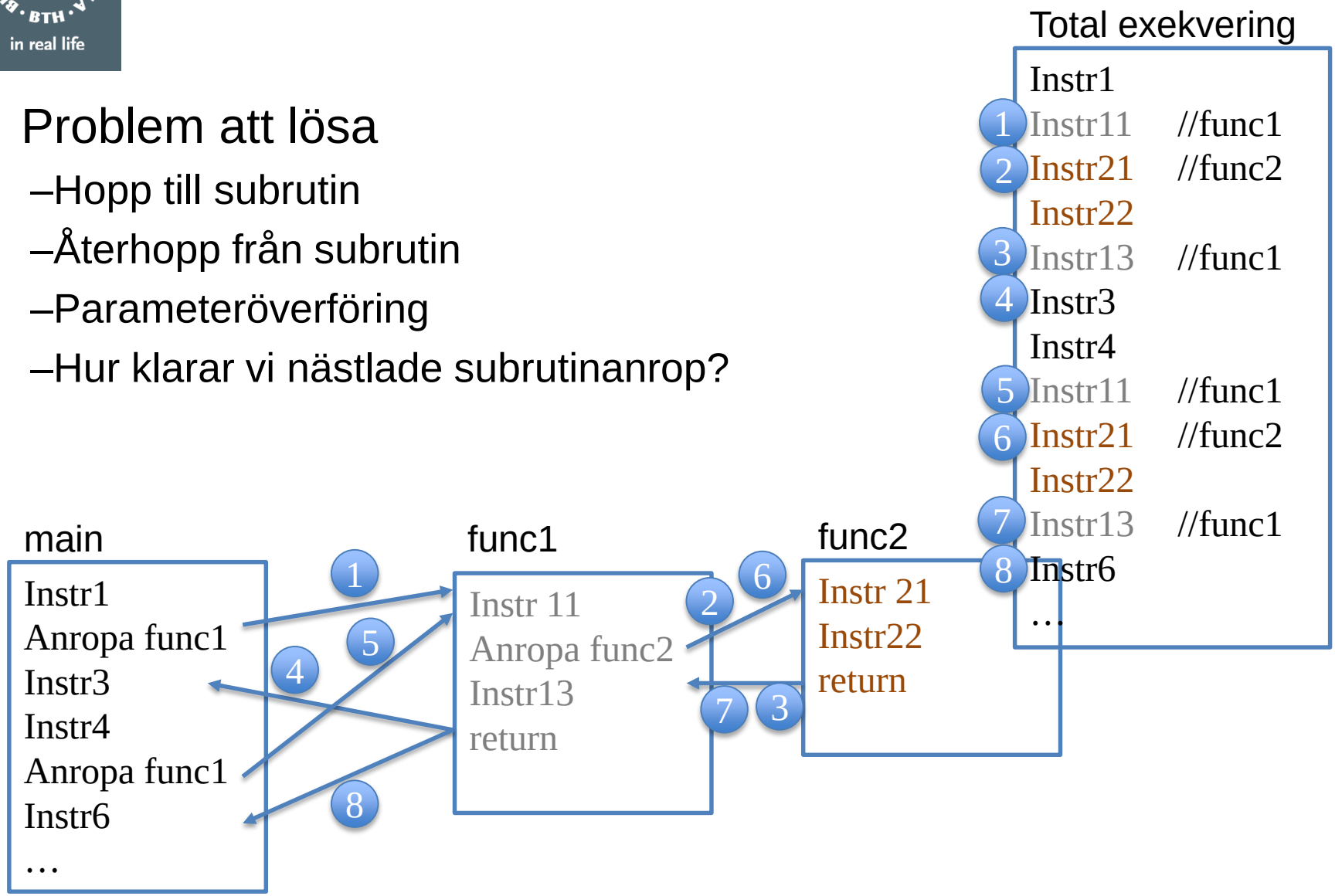
Funktionsanrop
+parameteröverföring

c = myAdd(a, b)

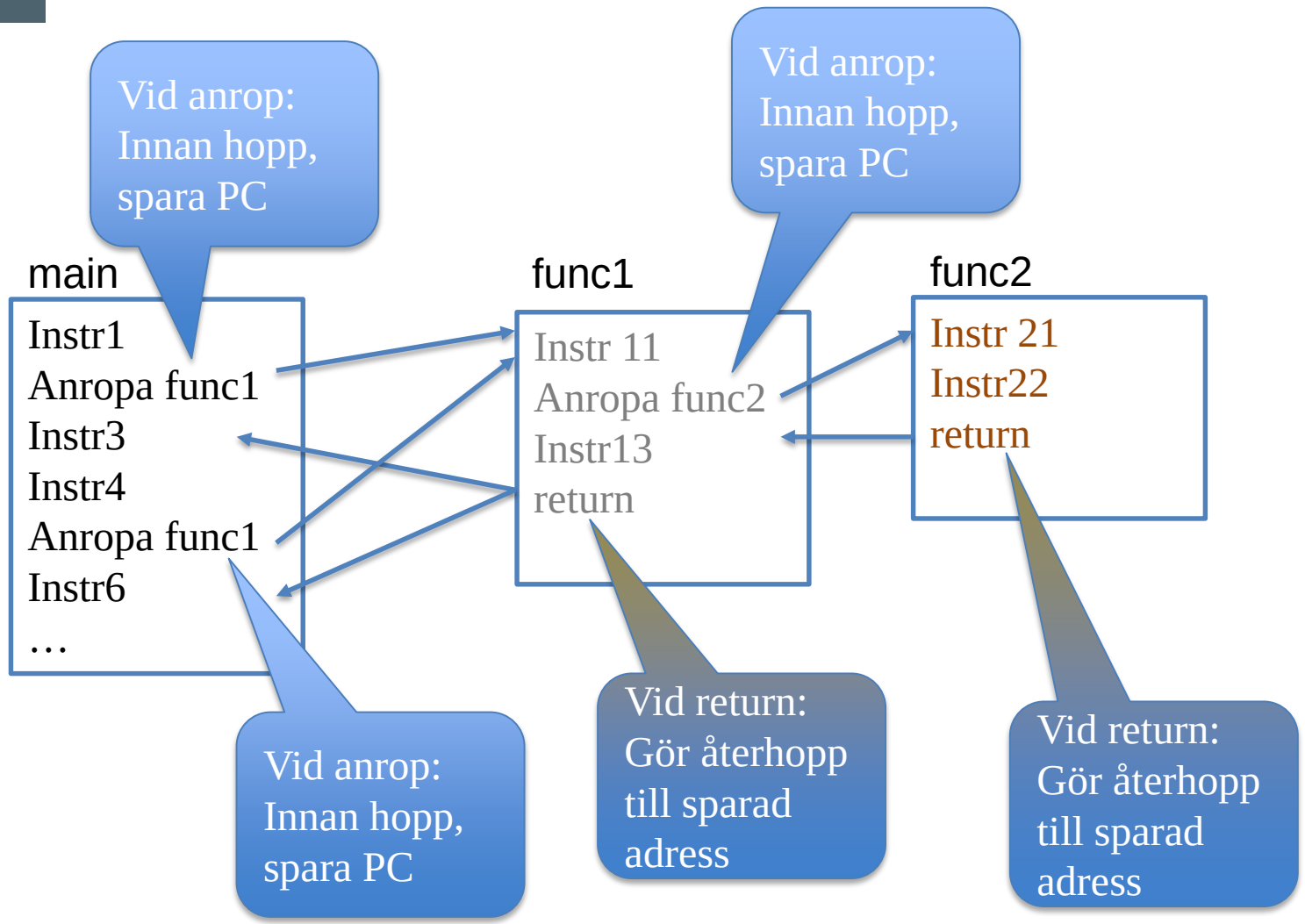
Återhopp +
parameteröverföring

Subrutiner forts.

- Problem att lösa
  - –Hopp till subrutin
  - –Återhopp från subrutin
  - –Parameteröverföring
  - –Hur klarar vi nästlade subrutinanrop?



Subrutiner forts.



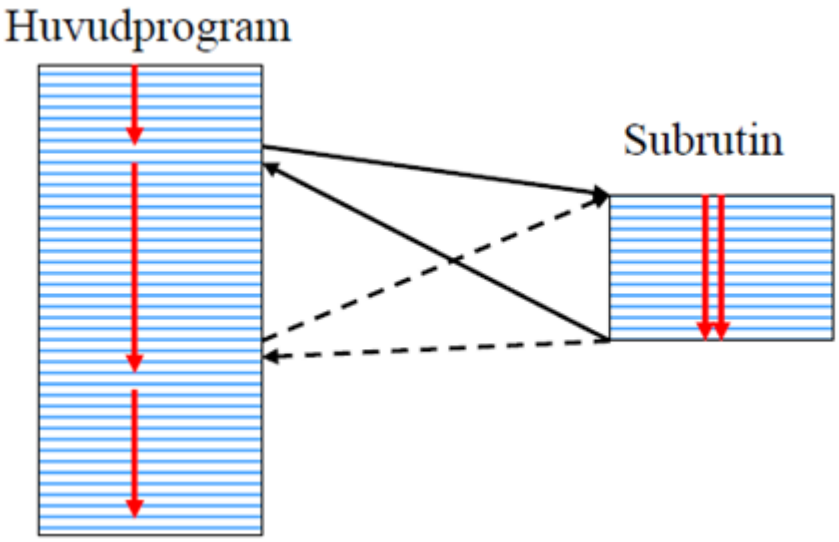
Subrutinanrop i ARM-assembler

- För att kunna anropa subrutiner måste vi veta
  - Var finns koden vi ska hoppa till?
  - Hur ska vi efter subrutinen veta vart vi ska hoppa för att komma tillbaka?

- Instruktionen **`BL <label>`** sparar automatiskt innehållet i PC i register $14 (**`lr`**).

- Återhoppet fås då genom **`BX lr`**

Subrutinanrop



Subrutinen ska hoppa till olika ställen beroende på varifrån den anropades
Återhoppsadressen måste alltså sparas!

Kan subrutiner se ut hur som helst?

- Subrutiner som anropar andra subrutiner kan ställa till problem!

- Vi måste hålla reda på en kedja av återhoppsadresser, men bara ett register i CPUn är vikt för återhoppsadress

Exempel på nästlade subrutinanrop

```
sub1:
   MOV r0,#0
   .
   .
   BL sub2 /* sparar
    återhoppsadress i lr */
   STR r0, [r1, #16]
   BX lr
```

```
sub2:
   ADD r0,r0,#7
   BX lr
```

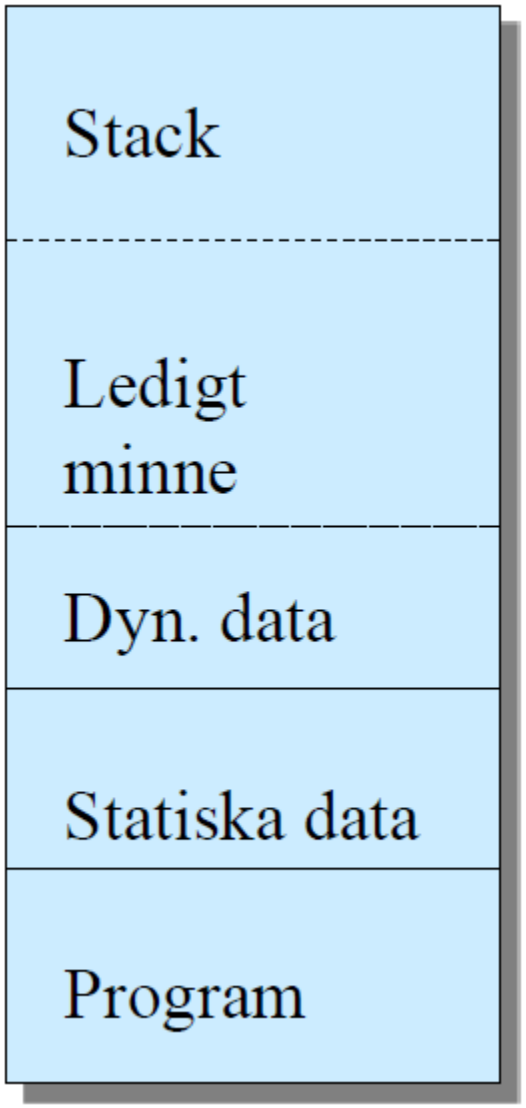
Fungerar inte!

Återhoppsadressen till huvudprogrammet har tappats bort och vi hittar inte tillbaka!

Analys av exemplet

- Subrutinen **`sub1`** anropar en annan subrutin
- **`BL`**-instruktionen skriver över **`lr`**
- Nu är det omöjligt att komma tillbaka till stället som **`sub1`** anropades ifrån
- Lösning: Innehållet från **`lr`** måste sparas någonstans innan nästa anrop!
- Var?

Minnesorganisation



Stack. Växer nedåt!
(Mot lägre adress)

Heap. Växer uppåt!

Variabler, .data

Programkod

Stack

- En stack är en struktur av typen LIFO (Last In First Out) (jfr stapel med tallrikar)

- Bara två typer av operationer är tillåtna på en stack:
  - **PUSH** – Lägg ett element överst på stacken
  - **POP** – Hämta översta elementet på stacken
  - I ARM assembler kan ett element vara en hel lista av registervärden

- Processorn håller reda på var översta elementet på stacken finns genom ett särskilt register
  - Stackpekaren (**sp**) (stack pointer)

Instruktioner för stackoperationer i ARM

PUSH – tar en sekvens av register och lägger dem på stacken

Exempel :

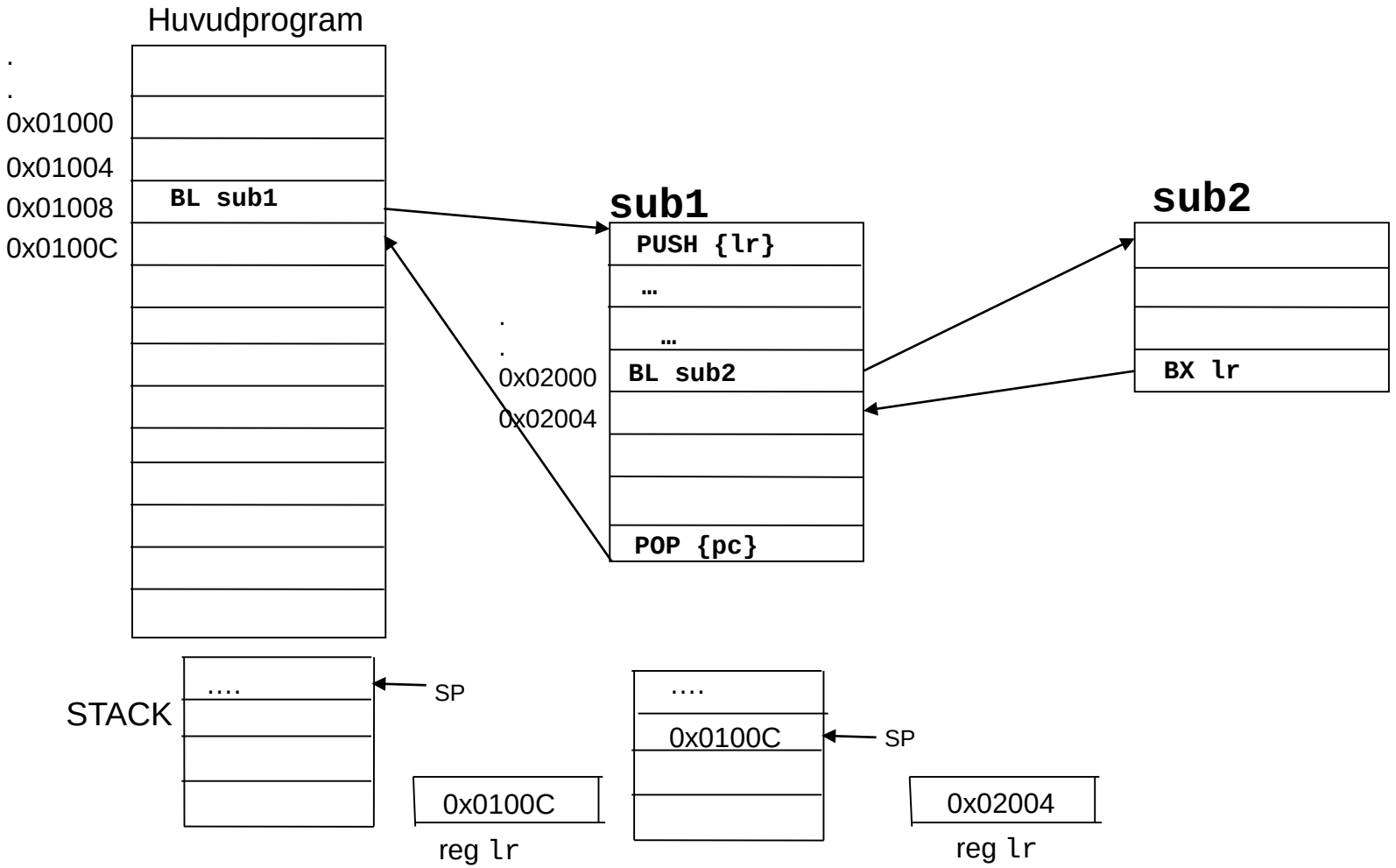
```
PUSH {r4, lr}
```

POP – tar en sekvens av värden från stacken och placerar i angivna register

Exempel :

```
POP {r4, pc}
```

Subrutinanrop



Register i ARM

Register	Namn	Kommentar
`$0`	**`r0`**	Primärt argument och returvärde
`$1`	**`r1`**	Sekundärt argument och högt word för 64-bit returvärde
`$2-$3`	**`r2-r3`**	Argument 3 och 4
`$4-$11`	**`r4-r11`**	Generella register, får ej förändras av subrutiner.
`$12`	**`r12`**	Scratchregister för funktioner
`$13`	**`sp`**	Stackpekare
`$14`	**`lr`**	Länkregister, används vid subrutinanrop
`$15`	**`pc`**	Programräknare

De logiska instruktionerna fungerar bara för **`r0–r7`**

Stacken och subrutiner

- Vad kan en subrutin vilja spara på stacken?
  - **`lr`** – registret med återhoppsadressen (om rutinen innehåller subrutinanrop)
  - **`r0-r3`** – parametrarna till rutinen
  - **`r4-r11`** – ska sparas om de används
  - lokala variabler
  - plats för argument till anropad rutin

Exempel på en enkel subrutin (löv)

Enkel rutin som ökar värdet som skickas som argument i r0 med 1. Resultatet returneras i r0.

```
/* Anropar inga subrutiner */
/* Om bara register r12 används behöver vi inte använda
   stacken */

.global increment  /* anger att rutinen ska vara globalt känd */

increment: /* label, gör att vi kan använda namnet som adress */
   ADD  r12,r0,#1      /* addera argumetet med 1 */
   MOV  r0, r12    /* lägg över i r0 */
   BX   lr         /* återhopp */
```

Lite mer komplicerad subrutin

- Den här rutinen ska öka på sitt argument med 4, och därefter använda förra rutinen för att öka på värdet med ytterligare 1.Öka till sist värdet med 4 en gång till och returnera i r0.
  (Lite krystat kanske, men fantasin är dålig)

```
/* subrutinens funktionella kod */

/* här måste något göras före */
   MOV  r4,r0
   ADD  r4,r4,#4       /* öka med 4 */
   MOV  r0,r4
   BL increment    /* hoppa till subrutinen */
   ADD  r4,r0,#4       /* öka med 4 igen */
   MOV  r0, r4
/* här måste något göras efter */
```

Prolog - programkod

- För att det här ska fungera problemfritt måste vi använda stacken.

```
.global komplex

komplex:
   PUSH    {r4, lr}    /* Lägg de registervärden som kan
                          förändras på stacken */
```

Den här kodsnutten ska alltså stå före den förra!

Återställ registeroch stack - Epilog

```
…
/* återställ alla register */

   POP {r4,pc}
```

I ARM assembler kan återhoppet göras genom att ”poppa” återhoppsadressen direkt i **pc**

Den här kodsnutten ska alltså stå sist!

Hela rutinen

```
.global komplex

komplex:
   PUSH {r4, lr}     /* Prolog */
   MOV   r4,r0
   ADD   r4,r4,#4    /* öka med 4 */
   MOV   r0,r4
   BL    increment   /* hoppa till subrutinen */
   ADD   r4,r0,#4    /* öka med 4 igen */
   MOV   r0, r4
   POP   {r4,pc}     /* Epilog */
```

Hur skriver man en subrutin - Sammanfattning

- Deklarera subrutinen
  - Spara register på stacken
    - Gör rutinens funktionella kod
  - Återställ registren (inkl. återhopp)

Rekursiv subrutin i assembler

Antag att vi vill skriva en rekursiv subrutin i assembler för MIPS som summerar de *n* första udda talen. (*n* är parameter)

Exempel:

Det första udda talet är 2*1 - 1 = 1

Det andra udda talet är 2*2 - 1 = 3

Det tredje udda talet är 2*3 - 1 = 5

Det *n*:te udda talet är då 2**n*-1

Idén med rekursion

- Beräkna summan som ett element plus en summa av resten av elementen.
- Summan av de *n* första udda talen blir då 2*n*-1 + Summan av de (*n*-1) första udda talen
- I ett högnivåspråk kan det se ut så här:

C-rutin

```
int sumOdd(int n)
{
    if (n == 0)
      return 0;
    else
      return 2*n-1 + sumOdd(n-1);
}
```

Rekursiv assemblerrutin
(prolog)

```
.global   sumOdd

/* Prolog  */
sumOdd:
   PUSH   {r4,lr}
```

Rekursiv assemblerrutin
(forts)

```
/* Subrutinens funktionella kod */
    CMP    r0, #0
    BEQ    lBase      /* hoppa ur om argumentet är noll */
    ADD    r4,r0,r0   /* beräkna 2*n (=n+n) till r4 */
    SUB    r4,r4,#1    /* beräkna 2*n-1 till r4 */
    SUB    r0,r0,#1   /* minska argumentet med 1 */
    BL     sumOdd     /* anropa funktionen själv */
    ADD    r4,r4,r0   /* 2*n-1 + returvärdet */
    BAL    lReturn     /* ovillkorligt hopp */
lBase:
    MOV    r4,#0      /* basfall */
lReturn:
    MOV    r0,r4  /* flytta returvärde till returregister */
```

Rekursiv assemblerfunktion
(forts, epilog)

```
/* Epilog – återställ register och återvänd */

   POP {r4,pc}
```

Allmän anropskonvention

Calling Convention

- Register **`r0–r3`** används för att skicka de fyra första parametrarna vid anrop till subrutiner (vid fler överförs de via stack)
- Register **`r0`** (och ev. **`r1`**) används för att returnera värden från funktioner
- Register **`r4-r11`** bevaras av subrutinen
- Register **`r12`** används vid dynamisk länkning och kan inte bevaras mellan subrutinanrop, kan användas som lokalt scratchregister i subrutin
- **`sp`** är stackpekaren som pekar ut det översta elementet på stacken, **`lr`** är det register som instruktionen **`BL`** använder för att bevara returadressen vid subrutinanrop, **`pc`** är programräknaren

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.

Exempelkod för sumOdd för ARMsim (demo)

```
.text
.global sumOdd

sumOdd:
    STMDB  sp!, {r4, lr}    /* PUSH */
    CMP    r0, #0
    BEQ    lBase
    ADD    r4, r0, r0       /* Beräkna (2*n) */
    SUB    r4, r4, #1       /* Beräkna (2*n-1) */
    SUB    r0, r0, #1       /* Bestäm argument (n-1) */
    BL     sumOdd
    ADD    r4, r4, r0       /* 2*n - 1 + returvärden från förra */
    BAL    lEnd
lBase:
    MOV    r4, #0
lEnd:
    MOV    r0, r4                /* Lägg returvärdet i returregistret */
    LDMIA  sp!, {r4, pc}         /* POP */

.global main
main:
    MOV    r0, #5
    BL     sumOdd
    MOV    r1, r0
halt:   BAL   halt
.end
```