Datorteknik

Tomas Nordström

Föreläsning 5

För utveckling av verksamhet, produkter och livskvalitet.

HÖGSKOLAN HALMSTAD

Föreläsning 5

- Subrutiner
- Stack
- Parameteröverföring

HÖGSKOLAN HALMSTAD

Upprepning av kod?

- Ibland har man kod som upprepas på flera ställen i sitt program.
- Man kan då kopiera och klistra in koden på varje ställe, men det är…
  - Minneskrävande och oflexibelt. Måste ändra på varje ställe om vi ändrar funktionen hos kodpartiet
- Vi skulle i stället kunna sätta ut ett programläge (label) och varje gång hoppa till denna kod, fast..
  - Vart hoppar man då man är färdig med kodpartiet? Hoppar man tillbaks till där man anropade kan man inte anropa från olika ställen!
- Vill kunna anropa funktionen och sedan fortsätta där vi är oavsett varifrån koden anropas!

Lösning: Subrutin

- Vi har programräknaren (PC) som alltid håller reda på var vi är.

  Spara undan (PC) vid hopp till kodpartiet.

  Återställ (PC) då kodpartiet är färdigt!
- Detta kallas att man gör ett *subrutin*- eller *funktionsanrop*.
- Kodpartiet som man anropar kallas subrutin eller funktion. Beror på om den har returvärde (funktion) eller ej (subrutin).
- I nästan alla arkitekturer finns stöd i instruktionsuppsättningen för just denna mekanism

Subrutinanrop och retur

- En subrutin anropas genom instruktionen BL som står för Branch with Link. Detta sparar automatiskt PC i länkregistret R14 (LR, Link Register). Det är detta ”with link” står för.

```
BL Delay_ms
```

- Återhopp från subrutinen till anropande kod görs genom att lägga tillbaka den sparade programräknaren från länkregistret

```
MOV PC,LR
```

- Notera att namnen PC och LR går att använda istället för att skriva R15 resp R14.

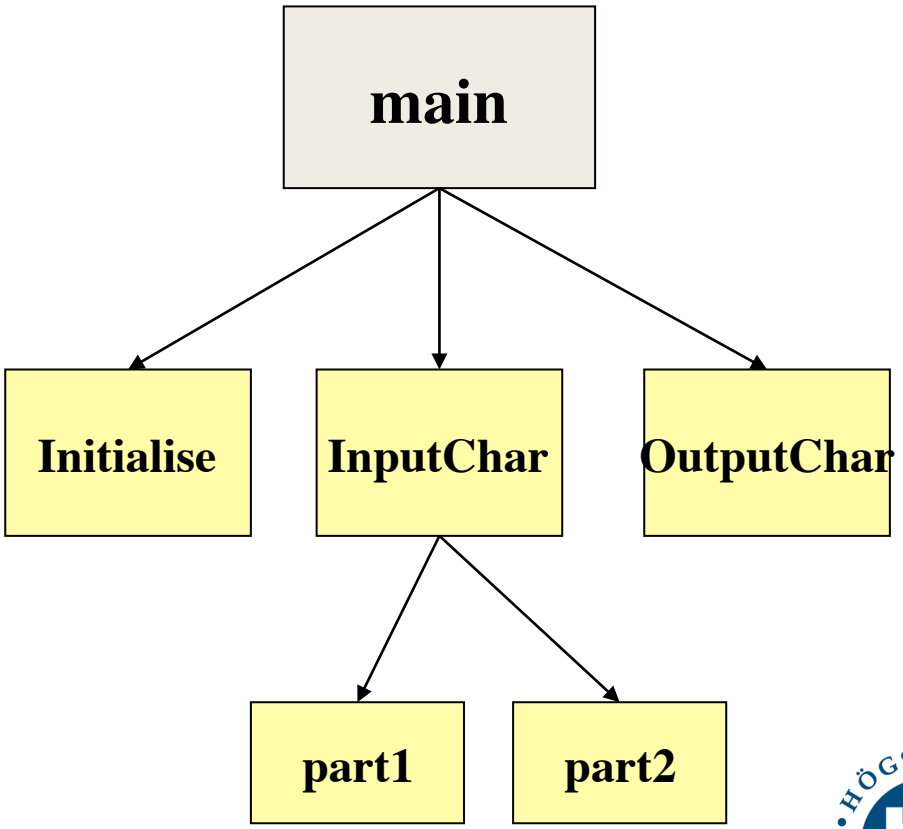
Subrutinanrop exempel

```
        BL TEST ;Anrop av TEST. Sker med speciell instruktion BL
        <KOD>
        BL TEST ;Anrop från annat ställe


TEST <SUBRUTINKOD>
        MOV PC, R14 ;Återgång från TEST R14=LR
```

Exampel2

```
    BL Initialise
    MOV R0,#3
    BL InputChar
    BL OutputChar
    SUBS R0,R0,#1
    BNE repeat
    …
    …
Initialise
    …
InputChar
    BL part1
    BL part2
    …
OutputChar
    …
```



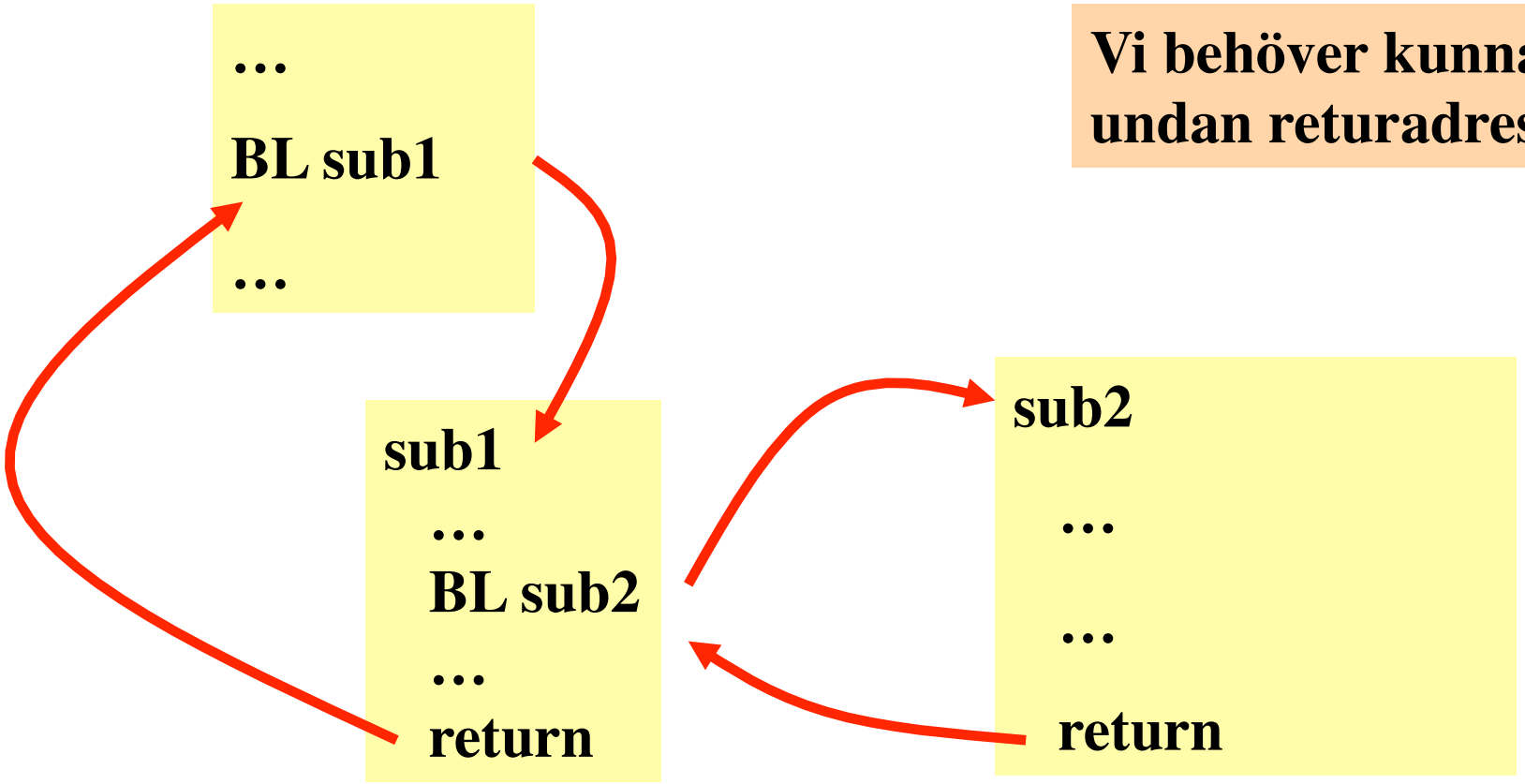
Nästlade subrutiner

- Ofta låter man subrutiner anropa andra subrutiner **(nästlade** subrutiner)
- Om (PC) sparas i LR kommer detta värde att skrivas över vid det "nya" subrutinanropet

```
        BL TEST          ;Anrop av TEST. (R14)=(PC), (PC)=TEST
        …
TEST BL NEXT             ;Nästlat anrop av NEXT. (R14)=TEST, (PC) NEXT
        <utför något>
        MOV PC, R14      ;Return från TEST. Här blir fel!
NEXT <utför något>
        MOV PC, R14 ;Return från NEXT
```

- PROBLEM: På retum-raden i TEST kommer PC att få värdet av TEST, dvs inte komma tillbaka till anropande kod.
- Lösning: Spara (LR) innan nästlat anrop.

Nästlade subrutiner



Vi behöver kunna lagra undan returadressen!!

Lokala variabler

- I en funktion (C) eller en metod (JAVA) finns något som heter lokala variabler. Dessa är ju, som ordet säger, lokala i subrutinen. Fördelen är att anropande kodavsnitt kan nyttja samma register som subrutinen. Registervärdena blir lokala i subrutinen.
- För att få en liknande funktion i assembler väljer man att spara undan innehållet i de register som används i subrutinen för att kunna återställa dem senare. Spara även LR ifall nästlade anrop förekommer.

```
        BL TEST
        …
TEST    Spara undan Rl-R4 + LR någonstans i minnet
        <KOD som använder Rl-R4,LR>
        Återställ Rl-R4 + LR ifrån minnet
        MOV PC, LR     ;Återgång från TEST
```

Var ska vi spara undan register?

- Vi måste kunna spara undan register som används plus LR vid subrutinens början för att säkerställa följande:
  - Subrutinen ska kunna använda registren utan att förstöra för anropande kod.
  - Tillåta nästlade anrop.
- Får varje register ett speciellt minnesutrymme blir det ändå problem vid nästlade anrop om anropad subrutin använder samma register.
  - Måste ha någon form av unikt utrymme för varje subrutin.
  - Fungerar om man lägger registerinformationen på hög och hela tiden läser tillbaka ovanifrån.

Lösningen kallas **STACK**.

Stack

- En stack är en form av lagringsutrymme som definieras av att det man lägger dit sist är det man kan plocka ut.
- Mekanismen att det sista man lägger dit är det första man tar bort kallas LIFO. *Last In, First Out*. Jämför en tallrikshög eller en kortlekshög
- En stack är en abstrakt datatyp, som definieras av vilka operationer man kan göra på den och inte hur den egentligen är konstruerad.

Operationer på stacken

- Det *enda* vi kan göra på vår stack är att lägga dit respektive plocka bort information.
- Lägga dit något på stacken kallas för att ''*pusha*" något på stacken.
- Ta bort något kallas för att ''*poppa*" något.
- De två funktionerna vi kan göra på stacken är PUSH respektive POP.
- Detta är en vanlig mekanism på många processorer och kan vara implementerad på olika sätt.

Implementering av stacken *i minnet*

- Stacken byggs vanligtvis upp av en sekvens av minnesplatser, tillsammans med en mekanism som håller reda på var toppen av stacken är. Kallas stackpekare.
- Stackpekaren (SP) håller reda på var man ska skriva resp. läsa. Ofta ett speciellt register. På ARM används R13. Detta register måste initieras i början av koden! Anta STACK_START är definierat innan i koden.

```
LDR SP,=STACK_START
```

- När man sedan PUSH:ar på stacken ökas stackpekaren med en position, vid POP minskas den med en position.

Operationer på stacken

	PUSH #12	PUSH #16	PUSH #15	POP
0x10000010	#12	#12	#12	#12
0x1000000C	////////////	#16	#16	#16
0x10000008	////////////	////////////	#15	#15
0x10000004	////////////	////////////	////////////	////////////
0x10000000	////////////	////////////	////////////	////////////

- Talet 15 ligger kvar på stacken (i minnet) även om vi gör POP. Fast vi kan inte nå det, eftersom det enda vi får göra på stacken är PUSH och POP.
- I detta exempel växer stacken (vid PUSH) mot lägre adresser. Inte nödvändigt. Den kan även växa mot högre adresser så länge principen 'Sist data som skrivs är först data ut' (LIFO-principen).

Multiple-Register Transfer

- Transfer multiple registers between memory and processor in a single instruction
- More efficient than several single transfers
- Use a base address register, Rn
- Used for moving blocks of data
- Also used for saving context & Stacks

Syntax

```
LDM{<cond>}<addressing mode> Rn{!},<registers>{^}
STM{<cond>}<addressing mode> Rn{!},<registers>{^}
```

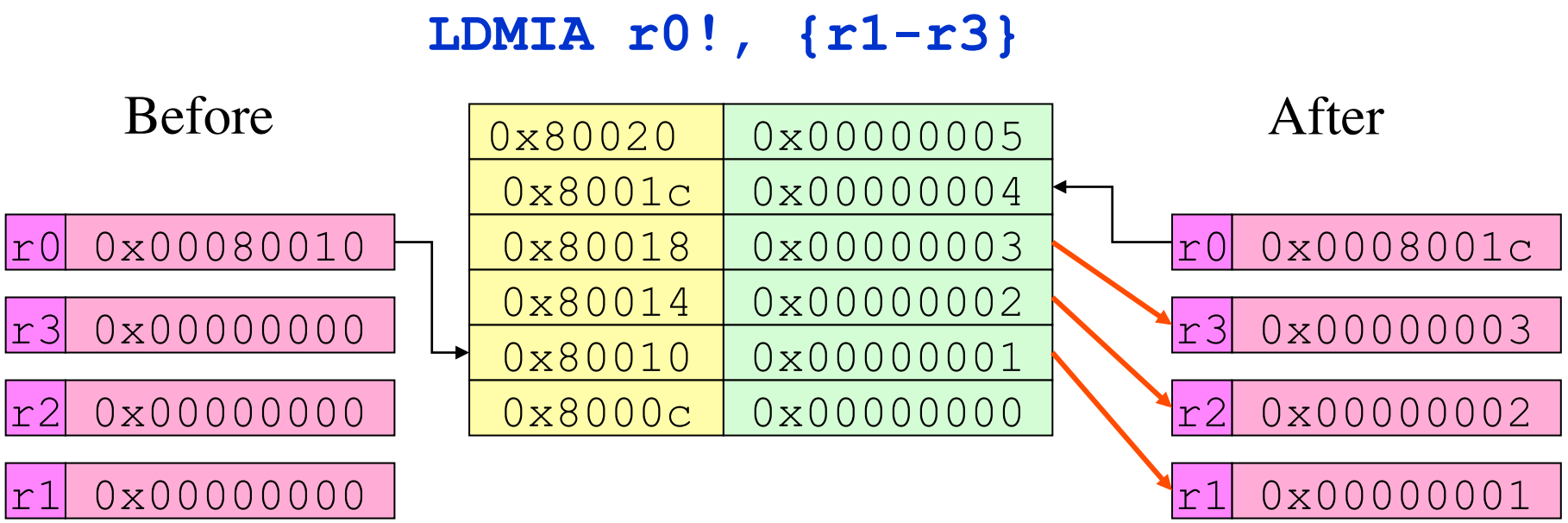
MODES

Mode	Description	Start address	End address	Rn!
IA	Increment after	Rn	Rn+4*N-4	Rn+4*N
IB	Increment before	Rn+4	Rn+4*N	Rn+4*N
DA	Decrement after	Rn-4*N+4	Rn	Rn-4*N
DB	Decrement before	Rn-4*N	Rn-4	Rn-4*N

N is number of registers in <registers>
! denotes writeback
^ denotes modified operation

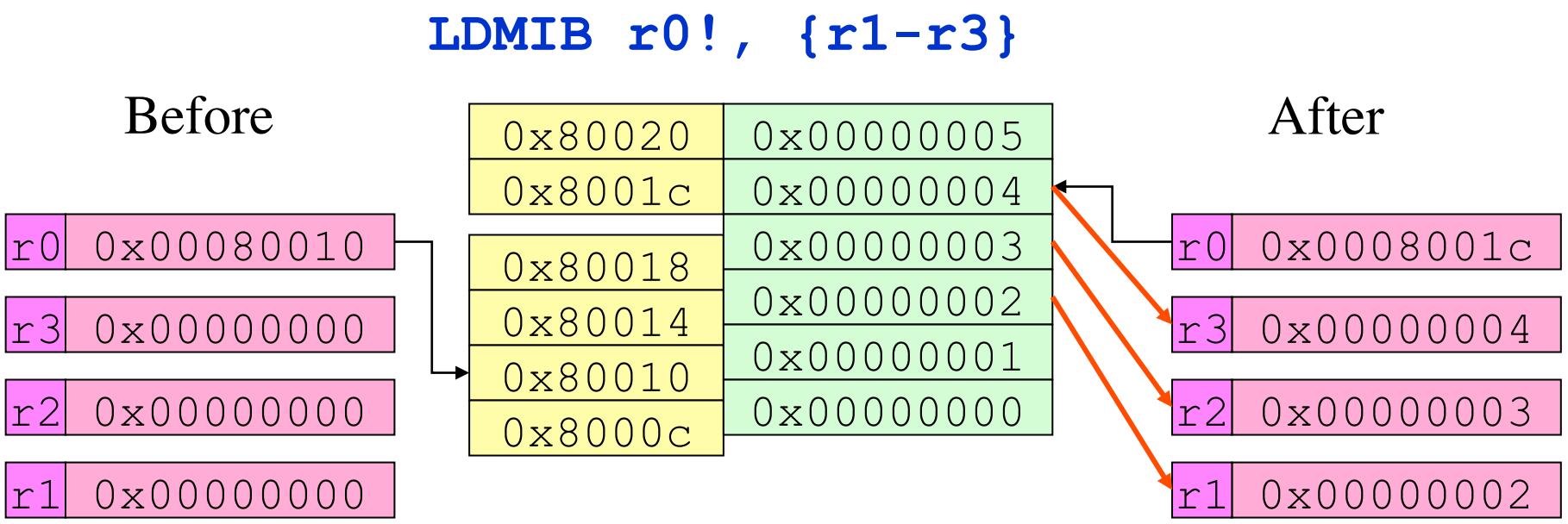
LDMIA

- `LDMIA` – Load Multiple Increment After



LDMIB

- `LDMIB` – Load Multiple Increment Before



Stack med LDM/STM version 1

- Med en stack som går ifrån hög adress till låg och pekar på ett giltigt värde (och inte på en tom position)
- PUSH mha STMDB

```
STMDB SP!,{R1-R3} ; Spara R1 till R3 på stacken
```

- POP mha LDMIA (samma som LDM)

```
LDMIA SP!,{R1-R3} ; Hämta R1,R2,R3 ifrån stacken
```

Att Skapa en Stack med LDM/STM Version 2

- Flytta innehållet i flera register samtidigt till och från minnet.

LDM*xx* Load Multiple	STM*xx* Store Multiple
Där xx är:	
FD= Full Descending	FA= Full Ascending
ED=Empty Descending	EA = Empty Ascending

Descending	skrivning mot lägre adresser
Ascending	skrivning mot högre adresser

- Genom att välja en av ovanstående sätt för både skrivning och läsning implementerar man PUSH (STMxx) och POP (LDMxx) på ARM.

Att flytta multipla register med ARM

- Tillägg av ! efter källregistret uppdaterar det. Observera att man kan skriva LR och SP i stället för R14 och R13.

PUSH:

```
STMFD R13!, {RO}      ;spara RO på stacken. Uppdatera SP
STMFD R13!, {RO-R4}   ;spara RO-R4 på stacken. Uppdatera SP
STMFD R13 !, {RO-R4, LR}    ;spara RO-R4, LR på stacken.
                            ;Uppdatera SP
```

POP:

```
LDMFD SP!, {RO}       ;hämtar RO från stacken. Uppdatera SP
LDMFD SP!, {RO-R4}    ;hämtar RO-R4 från stacken. Uppdatera SP
LDMFD SP!, {RO-R4, PC} ; hämtar RO-R4, LR från stacken.
                       ; Uppdatera SP
```

- Det behöver inte vara samma register som man lägger tillbaka i som de man sparade från, men bör vara samma antal register.
- Här nyttjas att direkt lägga tillbaka LR i PC

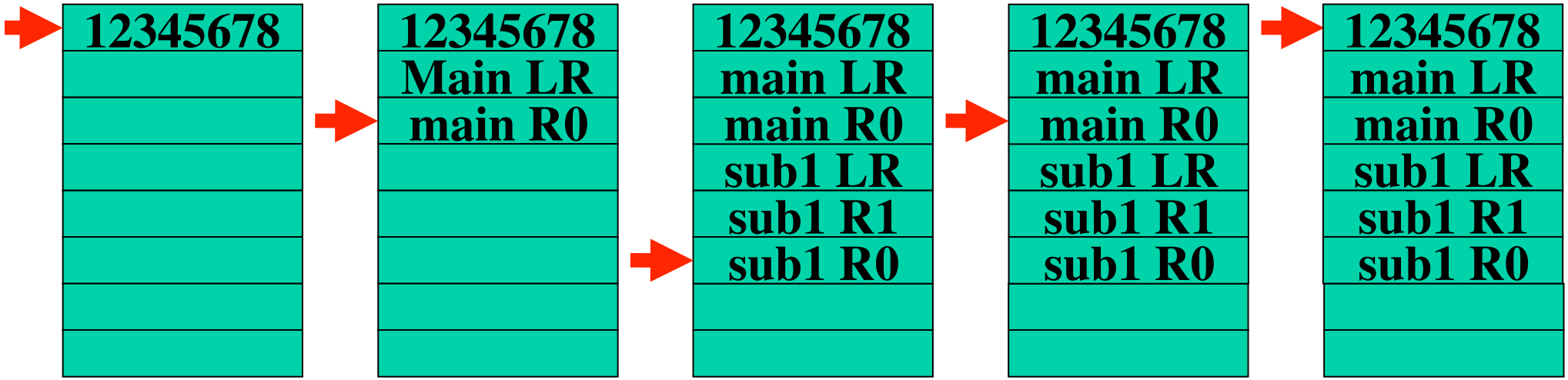
Stack Operation

- Pop: Load multiple instruction
- Push: Store multiple instruction
- There are four stack addressing methods:

Mode	Description	Pop	Push
FA	Full Ascending	LDMFA	STMFA
FD	Full Descending	LDMFD	STMFD
EA	Empty Ascending	LDMEA	STMEA
ED	Empty Descending	LDMED	STMED

Ascending – stack grows towards higher addresses
Descending – stack grows towards lower addresses
Full – stack pointer points at a full location
Empty – stack pointer points at an empty location

The Stack



Inside main Before calling sub1

Inside sub1 Before calling sub2

Inside sub2

Inside sub1 after calling sub2

Inside main after calling sub1

```
STMFD R13!,{R0,R14} ; first in sub1
        STMFD R13!,{R0,R1,R14} ; first in sub2
                    LDRFD R13!,{R0,R1,R15} ; return from sub2
                                LDRFD R13!,{R0,R15} ; return sub1
```

Template for a subroutine

Subroutine address

Push the lr and any working registers used in the subroutine

```
sub1
    STMFD R13!,{R0,R14}  ; Push R0 and
                         ; R14 - Link Register
    .......
                                  Use working registers locally
    .......
    LDMFD r13!,{r0,r14}  ; Pop r0 and LR
    MOV PC, r14          ; Put link register back into PC
```

Pop lr and working registers

Restore pc from link register

ARM: stackhantering
Lokala variabler vid subrutiner

```
TEST STMFD R13!, {RO-R4, R14} ;spara RO-R4, LR
        ; beräkning som använder RO till R4 genom att dessa bara
        ; är "variabler" som används i subrutinen sparar vi undan
        ; dess värden för att senare återställa.
        ; Bör alltid göras!
        …
        LDMFD SP!, {RO-R4, PC} ;hämtar RO-R4, LR. Uppdatera SP
```

- Notera: Det behöver inte vara samma register som man lägger tillbaka i som de man sparade från. I exemplet ovan läggs LR tillbaka i PC

```
alt)
        LDMFD SP!,{RO-R4, LR};hämtar RO-R4, LR. Uppdatera SP
        MOV PC,LR
```

Nested Subroutine Linkage

```
; In main
      BL sub1
      …
      …
sub1
   STMFD R13!,{R0,R14} ; Push R0 and
                       ; R14 - Link Register
   .......
   LDR R0,             ; Using R0 locally
   .......
   BL sub2             ; Call a nested subroutine
   .......
   LDMFD r13!,{r0,r14} ; Pop r0 and LR
   MOV PC, r14         ; Put link register back into PC

sub2
   STMFD R13!,{R0,R1,R14} ; Push R0, R1 and R14 - Link Register
   .......
   LDR R1,                ; Using R1 locally
   LDR R1,                ; Using R0 locally

   .......
   LDMFD r13!,{r0,R1,r14} ; Pop r0, r1 and LR
   MOV PC, r14            ; Put link register back into PC
```

Save lr (r14) and any working registers

Pop lr (r14) and any working registers

Save lr (r14) and any working registers

Pop lr (r14) and any working registers

Vi har även PUSH och POP instr.

Pop registers from stack	POP <registers>
Pop registers and PC from stack	POP <registers, PC>
Push registers onto stack	PUSH <registers>
Push LR and registers onto stack	PUSH <registers, LR>

- These stacks are full, descending stacks. The stack grows downwards, and the SP points to the last entry on the stack.
- Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

Subrutinanrop med PUSH/POP

```
subroutine PUSH {r5-r7,lr} ; Push work registers and lr
        ; code
        BL somewhere_else
        ; code
        POP {r5-r7,pc} ; Pop work registers and pc
```

Parameterpassning

- Funktioner i C eller metoder i Java kan ju ta inparametrar.
- Det lättaste sättet att utbyta information mellan anropande kod och subrutin är att använda ett register.
- Anropande kod lägger ett tal i ett register, som subrutinen använder.
- Subrutinen kan sedan lämna ett returvärde till anropande kod på samma sätt. Genom ett register. Viktigt är då att man inte sparar undan värdet på det. Skulle man det kommer ju ingen förändring av registret ske, och det är ju det man vill i detta fallet.
- Ett mer generellt sätt är att använda stacken. Denna metod används oftast av kompilatorer från högnivåspråk.

Exempel: subrutinanrop Parameter i register

```
        MOV R0, #BufferLen ; Length of Buffer in R0
        LDR R1, =BufferA ; Buffer A beginning address in R1
        LDR R2, =BufferB ; Buffer B beginning address in R2
        BL Swap; Call subroutine

; Swap - Swaps two strings - Assumes R0=Buffer len,
;         R1=addr to BufferA, R2=addr to BufferB
Swap    STMFD SP!, {R4,R5,LR} ; Save working regs and LR
        CMP R0,#0
Loop    BLE Done                ; Repeat until counter is zero
        LDRB R4,[R1]            ; Get the two buffer values
        LDRB R5,[R2]
        STRB R5,[R1],#1         ; Swap and point to next position
        STRB R4,[R2],#1
        SUB R0,#1               ; Reduce counter
        B Loop
Done    LDMFD SP!, {R4,R5,PC} ;Restore R4,R5, and return from subr.
```

Exempel: subrutinanrop med parameter i parameterblock

```
        BL Subr ; Call the subroutine
        ; parameters directly at the subroutine call
        DCD BufferLen ;Buffer Length
        DCD BufferA ;Buffer A starting address
        DCD BufferB ;Buffer B starting address
        <Code after call>

Subr    LDR R0, [LR], #4 ; Read BufferLen
        LDR R1, [LR], #4 ; Read address of Buffer A
        LDR R2, [LR], #4 ; Read address of Buffer B
        ; LR points to next instruction
        <subroutine code>
        MOV PC,LR ; Return
```

Exempel: subrutinanrop parametrar på stacken

```
        MOV R0, #BufferLen ; Read Buffer Length
        PUSH {R0} ; Save BufferLen on the stack
        LDR R0, =BufferA ; Read Address of Buffer A
        PUSH {R0} ; Save Buffer A on the stack
        LDR R0, =BufferB ; Read Address of Buffer B
        PUSH {R0} ; Save Buffer B on the stack
        BL Subr
        ADD SP,SP,#3*4 ; Clean up stack from inparam
        …
Subr    PUSH {R0, R1, R2, R11, LR} ; save working registers to stack
        ADD R11,SP,#5*4   ; Set up a frame pointer to point to inparam
        LDR R0, [R11, #0] ; Buffer B starting address
        LDR R1, [R11, #4] ; Buffer A starting address
        LDR R2, [R11, #8] ; Buffer Length in D0
        ... ; Main function of subroutine
        POP {R0, R1, R2, R11, LR} ; Recover working registers
        MOV PC, LR ; Return to caller
```

Vid parameterpassning på stacken

- Minska stackpekaren så att plats finns för parametrarna och returvärden och lagra sedan parametrarna utifrån SP:s värde. Eller PUSHa ut parametrarna
- I subrutinen så kommer man åt parametrarna som offset ifrån SP
- Lagra resultatet genom ett lämpligt offset ifrån SP
- Städa stacken antingen före eller efter retur ifrån subrutinen, ta bort parametrar och hantera returvärdet.

ARM Stack Frame Methodology

1. Caller pushes N parameters on the stack
2. Caller executes call (BL), return address is in R14
3. Subroutine does context save using PUSH, including R11 and R14
4. Subroutine sets R11 (frame pointer, aka FP) to the SP value to establish a fixed reference to the stack frame
5. Subroutine subtracts 4*X from SP to allocate space for X words of local variable space
   a. Must always allocate local variable space in multiples of words!
6. Subroutine code executes
   a. Parameters are accessed by using FP as base register with positive offset (i.e., the last parameter pushed on stack is addressed by [FP,# (4*registers_pushed_in_context_save)])
   b. Local variables are accessed by using FP as base register with negative offset (i.e. the first word of local variable space is addressed by [FP,#-4])
7. Subroutine deallocates local variables (ADD SP, #4*X)
8. Subroutine does POP to restore context and return
9. Caller adds value to SP to deallocate the passed parameter space
   a. ADD SP, #4*N
   b. Caller clean-up supports use of a variable number of arguments, so it is a common implementation in practice

Summering

Subrutin:

- Kodavsnitt som vi vill utföra flera gånger. Sparar (PC) så att vi kan hoppa tillbaka till där vi var innan subrutinen utfördes. Viktigt att tänka på att spara undan viktiga register så att vi inte förstör information. Kan sparas innan anrop eller i själva subrutinen. Bäst är i subrutinen.

Stack:

- En sk *Abstrakt Datatyp*. Ett minnesutrymme för temporär lagring. Kräver en sk stackpekare samt funktioner för att lägga dit resp plocka bort. Kallas ofta PUSH och POP. LIFO =Last In, First Out. "Lägg på hög och plocka ovanifrån". Finns ofta speciella instruktioner för stacken på många processorer.