

TSEA28 Datorteknik Y (och U)

Föreläsning 5

Kent Palmkvist, ISY



Dagens föreläsning

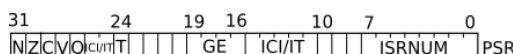
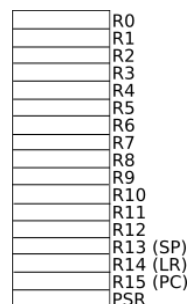
- Darma, ARM Cortex-M (labbdatorn)
 - Instruktionsuppsättning
 - Hur utföra vanliga operationer
- Exempel hur program fungerar
- Vanliga fel vid programmering

Praktiska kommentarer

- Labanmälan finns tillgänglig
 - Kursrummet på lisam, nere till vänster
 - Lab 1 del A deadline anmälan 250203, kl 23.30
 - Anmälan krävs för att få vara i labbet under schemalagd tid
- Labförberedelser
 - Bra att vara förberedd innan 1:a lektionen
 - Vi räknar inte med att ni kan processorbeskrivningen utantill, men att ni vet var man hittar den informationen i manualen
- Programvaran installerad på alla linux maskiner (thinlinc, ssh, asgård, olympen, egypten)
 - Endast användbart fört kodinmatning och assemblering
- MUX2: Maskiner med distansutrustning
 - Emulerar tangentbord etc. mha arduino (kommandon tsea28lab1, tsea28lab2 etc.)
 - Se Introduktion till Darma, video på lisam, samt förra föreläsningen för mer information.

Programmerarens vy av DARMA

- CPU med 13 generella register, PC, LR, SP samt statusregister PSR
- 256 KB Flash programminne (ROM, bara läsning)
 - Programkod
 - Konstanter
- 32KB RAM för data (skriv och läs)
 - Variabler
 - Stack etc.
- Minnesmappad I/O



00000000	ROM (FLASH)
0003FFFF	
20000000	RAM
20007FFF	
40000000	IO-enheter GPIO A-F UART mm.
43FFFFFF	
E0000000	Intern IO
E00FFFFFF	

Skillnad mellan kod Darna och andra miljöer

- Code Composer studio använder TI assembler, boken använder Kiel assembler (dvs ARM officiella version)
 - Olika sätt beskriva konstanter, alignment, etc.

Kiel:	TI:
DCB value	.field value,8 ; 8 bit värde placerat i programminnet
DCW value	.field value,16 ; 16 bit värde placerat i programminnet
DCD value	.field value,32 ; 32 bit värde placerat i programminnet
ALIGN	.align 4
DCB "text"	.string "text"
AREA	.text
	.global
EQU	.equ
LDR r0, =value	Finns ej, måste välja mellan MOV, MVN, MOVT och LDR

Konstanter i ARM/Thumb instruktionsformat

- En instruktion kan beskrivas med maximalt 32 bitar
 - Både opcode och argument måste få plats på 32 bitar.
 - Argument måste därför vara mindre än 32 bitar
- Konstanter i instruktioner (argument) måste vara kortare än 32 bitar
 - Dessutom flera argument till samma instruktion (t ex register och värde)
- I många fall kan en 32-bitars konstant konstrueras som ett 8-bitars värde med 0:or på båda sidor för att skapa ett 32-bitars värde
 - Ex: 0x00000093 ses som 10010011 med 24 nollor till vänster
 - Ex: 0x00027400 ses som 10011101 med 10 nollor till höger och 14 nollor till vänster
 - Ex: 0x0E003000 går ej plats då 1110000000000011 är fler än 8 bitar
 - Ex: -3 går ej plats då det översätts till 0xffffffff

Assemblerinstruktioner i Darma

Ladda små konstanter till register

- Små (max 8 bitar) och enkla konstanter (8 bitar skiftade vänster)
 - MOV rd,#konstant ; Placera värdet i register rd
 - MVN rd,#konstant ; Placera inverterade värdet i register rd
; (dvs -konstant-1)
- Exempel
 - MOV r0,#0x0 ; Sätt r0=0x00000000
 - MVN r0,#0x0 ; Sätt r0=0xffffffff
 - MOV r0,#0x000FF000
- Knep med negativa tal (kuriosa)
 - MVN r0,#~(negativkonstant) ; assembler beräknar rätt mönster
 - Exempel: MVN r0,#~(-3) ; ger MVN r0,#0x00000002 vilket ger r0=0xffffffffd (-3)

Ladda en 16 bitars konstant till register

- Kan även använda 16-bitars konstant för laddning av konstant
 - `MOV rd,#Konstant16bit`
 - De övre 16 bitarna (bit 31-16) nollställs i registret
- Exempel
 - `MOV r0,#0xa357`
 - `MOV r0,#12345 ; 12345 = 0x3039, dvs 16 bitar räcker`

Ladda en lång konstant till register (32 bit)

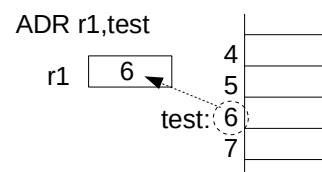
- Längre konstanter (värde känt vid assemblering) kan laddas i två steg
 - Sätt först de 16 lägsta bitarna (nollställer samtidigt de övre 16 bitarna i registret)
`MOV rd,#(longkonstant & 0xffff) ; 16 lägsta bitarna av värdet ; placeras i register rd`
 - Sätt de översta 16 bitarna
`MOVT rd,#(longkonstant >> 16) ; Placera 16 högsta bitarna av värdet`
- Exempel
`MOV r0,#(0x12345678 & 0xffff) ; Sätter r0=0x00005678`
`MOVT r0,#(0x12345678 >> 16) ; sätter översta 16 bitarna, ; resultat r0=0x12345678`

Viktig begränsning hos konstantvärdet

- `&` = logisk and, `>>` = logisk skift höger
 - C-kod syntax
- `&` och `>>` beräknas innan assemblering
 - Måste vara kända konstanter
 - Går inte använda adresser i programminnet
 - Inte tilldelade adresser förrän vid länkning
 - Kan inte vara registervärden
- Går att skriva kod utan att använda `&` och `>>`
 - Exempel:
 - `MOV r0,#5678` ; Sätter `r0=0x00005678`
 - `MOVT r0,#0x1234` ; sätter översta 16 bitarna, resultat `r0=0x12345678`

Ladda register med en address i programminnet

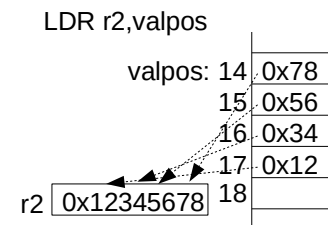
- Programminnets exakta adresser inte kända vid assemblering
 - Behöver kunna räknas fram av länkaren
 - Pseudoinstruktion finns för detta
- `ADR rd,label`
 - Lagra adressen till label i register `rd`
 - Label måste ligga max +/-4095 från nuvarande PC-värde (placering av instruktionen)
 - Kan peka på konstanter (tabeller och data) lagrat i programminnet
 - Kan inte peka på variabler i RAM-minnet (för långt bort!)
- Assembler översätter till `add` eller `sub` av PC-register



data: .field 0x1234,32
prog: adr r0,data
ldr r1,[r0]

Ladda en lång konstant, forts.

- LDR r0,constlabel
 - Ladda värde från minne, constlabel får vara placerad i programminnet max 4095 adresser bort från PC
 - Implementeras som läsning med offsetadressering mha PC
 - LDR r2,valpos
assembler => ldr r2,[PC,#offset]
- Constlabel .field 0x12345678,32
 - Definiera datavärde (32 bitar lång) placerat i programminnet på plats constlabel (little endian)
 - Vanligt samla ihop alla konstanter i ett block placerat efter programkoden

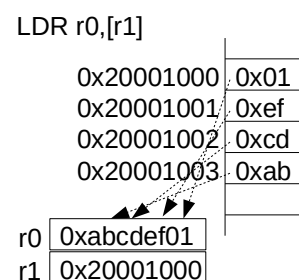


; definiera ett 32-bitars värde
; i programminnet på plats valpos

valpos: .field 0x12345678,32

Läsning från minnet (32 bit)

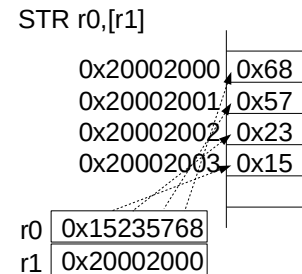
- Data och I/O-enheter ligger långt från programminnet
 - Använd ett register som innehåller (pekar på) adressen som ska läsas
 - Sätt värdet i pekarregistret innan minnesaccess
 - Många adresseringsmoder att välja bland
- LDR rd,operand ; läs värde från minne
 - Operand anger hur minnesadressen ska bestämmas
- Exempel läs från dataminnet på adress 0x20001000



LDR r0,[r1] ; antag r1=0x20001000

Skrivning till minnet (32 bit)

- STR rn,operand ; skriv värde i rn till minne
- Exempel skriv r0 till dataminnet
 - ; Antag r0=0x15235768
 - ; Antag r1=0x20002000
 - STR r0,[r1] ; sätter minne[0x20002000]=0x15235768
- 32-bitars data kan för vissa ARM-processorer kräva adress jämnt delbar med 4 (både för LDR och STR)
 - DARMA kan inte använda udda adress vid läsning av programminnet!
 - DARMA kan inte använda udda adress vid läsning och skrivning i dataminnet!



Läsning och skrivning i minnet, byte

- Kan läsas/skrivas från valfri adress (även udda adress)
 - 8 lägsta bitarna (bit 7-0) i registret används
- Exempel
 - LDRB r0,[r1] ; läs en byte från adress som r1 pekar på
 - STRB r0,[r1] ; skriv en byte (bit 7-0 i r0) till adress som r1 pekar på
- Registerbitar 31-8 vid läsning av byte från minne
 - LDRB r0,[r1] ; sätt bit 31-8 = 0
 - LDRSB r0,[r1] ; sätt bit 31-8 = teckenbit på indata (bit 7)
 - LDRSB = LDR Signed Byte, dvs teckenförlänger

Läsning och skrivning i minnet, halvord

- Kan läsas/skrivas från valfri jämn adress (även ej jämnt delbar med 4)
 - 16 lägsta bitarna (bit 15-0) i registret används
- Exempel
 - `LDRH r0,[r1]` ; läs två byte med start på adress som r1 pekar på
 - `STRH r0,[r1]` ; skriv två byte med start på adress som r1 pekar på
- Registerbitar 31-16 vid läsning av byte från minne
 - `LDRH r0,[r1]` ; sätt bit 31-16 = 0
 - `LDRSH r0,[r1]` ; sätt bit 31-16 = teckenbit på indata (bit 15)
 - `LDRSH` = `LDR Signed Halfword`, dvs teckenförlänger
- Använder little-endian, bit 7-0 i `[r1]`, bit 15-8 i `[r1+1]`

Adresseringsmoder i Cortex-M

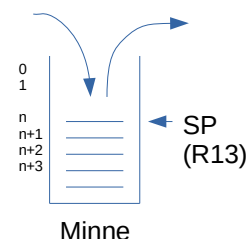
- Access till minne på en adress som anges i instruktionen finns inte hos ARM
 - `LDR r0,0x231450` ; läs på adress 0x231450 FINNS INTE hos arm
- Adress som används för minnesaccess kan delvis räknas ut samtidigt (konstanter begränsade till 12 bit 2-komplement)
 - Register indirekt: `LDR r0,[r1]`
 - hämta värde från adress som finns i r1
 - Register indirekt med Offset: `LDR r0,[r1,#8]`
 - hämta värde från adress beräknad av $r1 + 8$
 - Register indirekt register indexed: `LDR r0,[r1,r2]`
 - hämta värde från adress beräknad av $r1 + r2$
 - Register indirekt register index med skalning: `LDR r0,[r1,r2,LSL#2]`
 - hämta värde från adress beräknad som $r1 + (r2 \ll 2)$ dvs $r1 + r2*4$

Adresseringsmoder i Cortex-M, forts.

- Minnesaccesser kan även uppdatera registret som innehåller adressen
 - Register indirekt post-increment: `ldr r0,[r1],#4`
 - läs från minnesadress som finns i r1, öka därefter r1 med 4
 - Register indirekt pre-increment: `ldr r0,[r1,#4]!`
 - Öka först värdet i r1 med 4, läs sedan från minnesadress som anges av det nya värdet i r1
- Användbart för hantering av data i strängar och sekvenser
 - Läs tecken i en sträng och öka samtidigt positionspekare
 - `LDRB r0,[r1],#1` ; Läs en byte från position som anges i r1, öka ; sedan position (dvs r1) med 1

Stackhantering i ARM Cortex-M

- R13 (SP) pekar på översta använda adress i stacken
 - Lägst adress som används. Uppdateras av PUSH och POP
- PUSH {registerlista}
 - Sparar register på stacken (kan vara många)
 - Exempel: `PUSH {r0,r1}` ; Spara r0 och r1 på stack
 - Placerar r1 först, r0 sedan på stack (använder alltid högst registernummer först)
- POP {registerlista}
 - Hämtar värden från stack till register
 - Exempel: `POP {r3,r4}` ; Hämta först r3, sedan r4 från stack (använder alltid lägst registernummer först)



Att placera annat data i programminnet

- Programminnet innehåller program och konstanter
- Konstanter kan placeras i programminnet mellan subrutinerna
- Olika sätt att beskriva konstanta värden
 - Label: `.field value,size`
 - Size 8, 16 eller 32
 - Label: `.string "sträng",byte,byte`
 - Sträng översätts till ascii-tecken, efterföljande byte kan vara t ex 13, 10 för CR, LF
- Se till att nästa rutin eller data börjar på adress delbar med 4
 - `.align 4`
- Hitta adress till konstanterna
 - `Adr rd,label`

Adresseringslägen, cortex-M, forts.

- Vilka behöver man kunna?
 - Direkt (ladda konstanter)
 - Registerindirekt (använda r0-r12 som pekare)
 - push/pop på stack
 - Alla andra går att lösa mha beräkning i r0-r12 registren
 - Dock mindre effektivt

Beräkningsinstruktioner

- Arbetar bara på register (aldrig direkt mot minnet)
- ADD,SUB ; addition och subtraktion
 - ADDS, SUBS för att uppdatera flaggor
 - Exempel: ADD r2,r2,r3 ; $r2=r2+r3$, inga flaggor uppdateras
 - SUBS r1,r2,#2 ; $r1=r2-2$, uppdaterar Z,N,C,V flaggorna
- CMP ; jämför värde i register
 - Påverkar alltid flaggor, ändrar inte register r0-r12
 - Exempel: CMP r0,r1 ; beräkna r0-r1, uppdatera flaggor enligt resultatet
- AND,ORR,EOR ; bitvis logisk operation
 - ANDS, ORRS, EORS för att uppdatera N och Z flaggor
- LSL, LSR, ASR, ROL, ROR ; skiftning/rotation
- MOV ; flytta värde mellan register
 - MOVS för att uppdatera N och Z flaggorna

Hopp

	cc	flaggvärden	betydelse
• b label			
- Hoppa i koden till positionen label (ovillkorligt)	EQ	Z=1	lika
	NE	Z=0	inte lika
	CS eller HS	C=1	större eller lika, positiva heltal
	CC eller LO	C=0	mindre än, positiva heltal
• bcc label	MI	N=1	negativt
- Villkorligt hopp till positionen label	PL	N=0	positivt eller noll
	VS	V=1	spill
	VC	V=0	inget spill
	HI	C=1 och Z=0	större än, positiva heltal
	LS	C=0 eller Z=1	mindre eller lika, positiva heltal
	GE	N = V	större än eller lika, 2-komplement
	LT	N != V	mindre än, 2-komplement
	GT	Z=0 och N=V	större än, 2-komplement
	LE	Z=1 och N!=V	mindre än eller lika, 2-komplement

Subrutiner, anrop och återhopp

- BL label ; Branch and link
 - Hoppa i koden till positionen label och lagra samtidigt adressen till instruktionen efter bl i register LR (r13)
 - LSB i R13 sätts till 1 automatiskt (för att indikera thumbkod)
- Återhopp
 - BX LR ; Branch and exchange LR
 - Återställ PC (R15) till värdet i LR (r13)
 - Viktigt: r13 måste ha LSB = 1 (måste vara en udda adress, sätts av bl)

Subrutinanrop i flera nivåer

- En subrutin som anropar en annan subrutin måste först spara undan länkregistret
 - PUSH {LR}
- Innan återhopp måste LR återställas
 - POP {LR}
 - BX LR

Exempel: stack och subrutiner

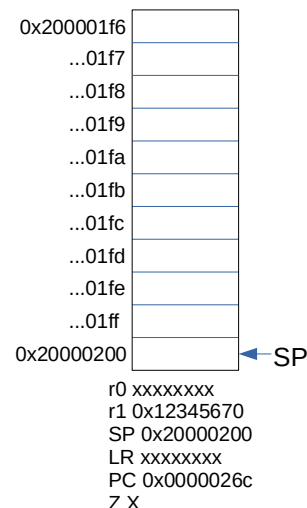
- Lite exempelkod

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
→ 00026c    MOV    r0,#66
   000270    BL     0x00000344
   000274    ...

   000344    PUSH   {r1}
   000348    MOV    r1,#0
   00034c    ADD    r1,r1,r0
   000350    SUBS   r0,r0,#1
   000354    BNE    0x00034c
   000358    MOV    r0,r1
   00035c    POP    {r1}
   000360    BX     LR

```



- Går igenom koden

- Exempel på hur man kan ta reda på funktion hos en given maskinkod

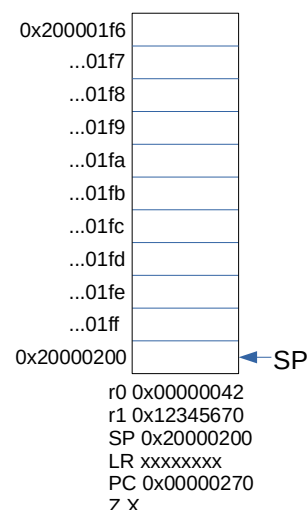
Exempel: stack och subrutiner, forts. 1

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
→ 00026c    MOV    r0,#66
   000270    BL     0x00000344
   000274    ...

   000344    PUSH   {r1}
   000348    MOV    r1,#0
   00034c    ADD    r1,r1,r0
   000350    SUBS   r0,r0,#1
   000354    BNE    0x00034c
   000358    MOV    r0,r1
   00035c    POP    {r1}
   000360    BX     LR

```



- Ladda r0 med $66_{10} = 42_{16}$
 - Flaggor påverkas inte
 - Räkna upp PC till nästa instruktion

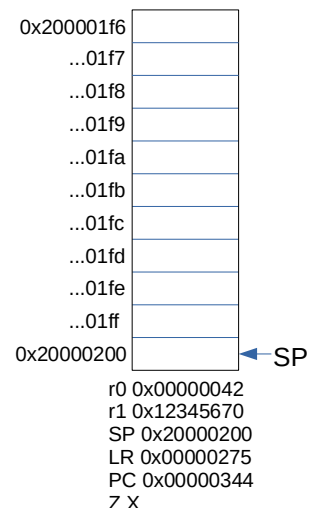
Exempel: stack och subrutiner, forts. 2

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x000034c
000358    MOV    r0,r1
00035c    POP     {r1}
000360    BX     LR

```



- Kopiera återhoppadress (nästa PC) till LR
 - Instruktion efter BL är på 0x00000274, med LSB=1 fås 0x00000275 (placeras i LR)
 - Sätt PC till subrutinens adress

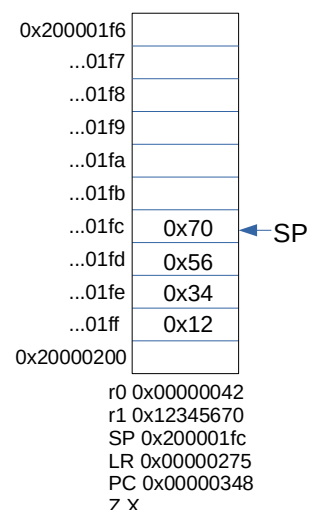
Exempel: stack och subrutiner, forts. 3

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x000034c
000358    MOV    r0,r1
00035c    POP     {r1}
000360    BX     LR

```



- Push nuvarande r1-värde till stack
 - Minska SP med 4, sätt M[0x200001ff], M[0x200001fe], M[0x200001fd], M[0x200001fc] med little-endian ordning

Exempel: stack och subrutiner, forts. 4

```

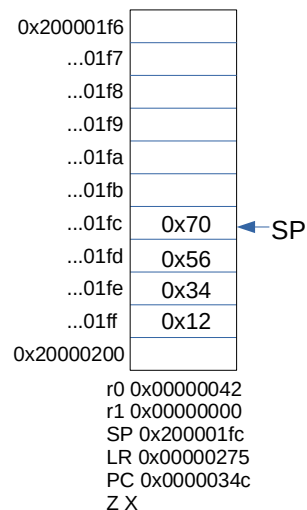
; Antagande. SP = 0x20000200,
; r1=0x12345670 från main

00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
→ 000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```

- Sätt r1 till 0



Exempel: stack och subrutiner, forts. 5

```

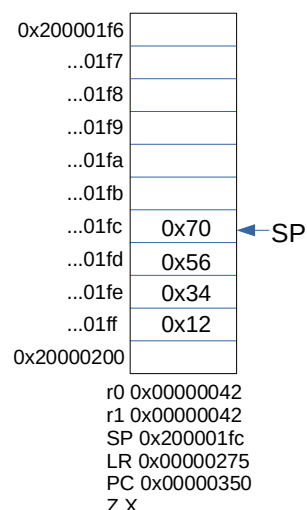
; Antagande. SP = 0x20000200,
; r1=0x12345670 från main

00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
→ 000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```

- Beräkna $r1 = r1 + r0$
 - $r1 = 0x000000, r0 = 0x000042 \Rightarrow r1 = 0x000042$
 - Inga flaggor ändras då det inte är ADDS



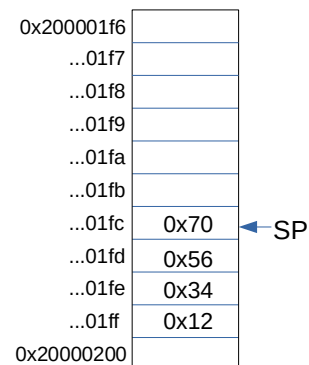
Exempel: stack och subrutiner, forts. 6

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
→ 000350    SUBS   r0,r0,#1
000354    BNE    0x000034c
000358    MOV    r0,r1
00035c    POP     {r1}
000360    BX     LR

```



- Beräkna $r0 = r0 - 1$, uppdatera flaggor
 - Resultat skilt noll $\Rightarrow Z=0$

```

r0 0x00000041
r1 0x00000042
SP 0x200001fc
LR 0x00000275
PC 0x00000354
Z 0

```

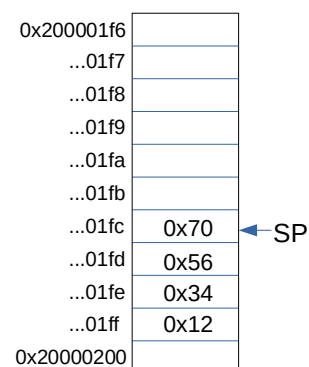
Exempel: stack och subrutiner, forts. 7

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
→ 000350    SUBS   r0,r0,#1
000354    BNE    0x000034c
000358    MOV    r0,r1
00035c    POP     {r1}
000360    BX     LR

```



- Om $Z = 0$ hoppa till 0x0000034c
 - $Z = 0 \Rightarrow PC = 0x0000034c$

```

r0 0x00000041
r1 0x00000042
SP 0x200001fc
LR 0x00000275
PC 0x0000034c
Z 0

```

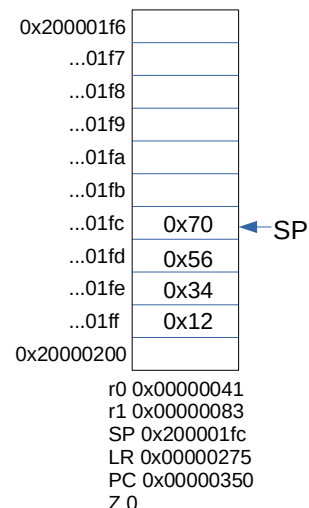
Exempel: stack och subrutiner, forts. 8

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
→ 00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```



- Beräkna $r1 = r1 + r0$
 - $r1 = 0x42, r0 = 0x41 \Rightarrow r1 = 0x83$

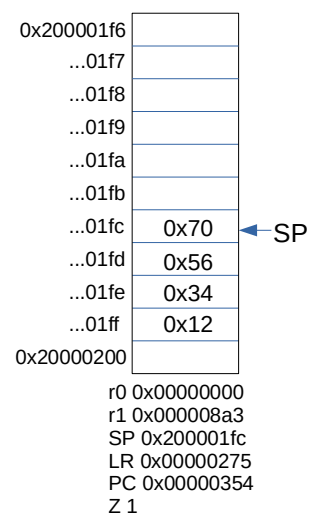
Exempel: stack och subrutiner, forts. 9

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```



- Loop address 0x034c - 0x0354 exekveras tills $r0 = 1$ lagts till $r1$
 - $r1$ blir totalt $0x42 + 0x41 + \dots + 0x02 + 0x01 = 0x08A3$
 - Senaste SUBS med $r0 = 1$ ger $r0 = 0, Z = 1$

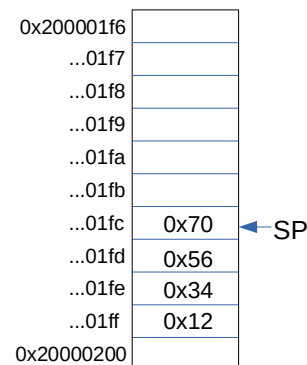
Exempel: stack och subrutiner, forts. 10

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
→ 000354    BNE    0x00034c
000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```



- Om Z = 0 sätt PC till 0x00034c
 - Z = 1 därför blir PC = 0x000358 (nästa instruktion)

```

r0 0x00000000
r1 0x000008a3
SP 0x200001fc
LR 0x00000275
PC 0x00000358
Z 1

```

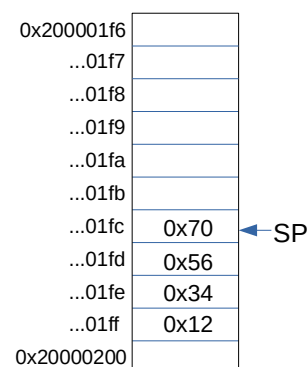
Exempel: stack och subrutiner, forts. 11

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
→ 000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```



- Kopiera värdet i r1 till r0
 - r1 = 0x000008A3 => r0 = 0x000008A3

```

r0 0x000008a3
r1 0x000008a3
SP 0x200001fc
LR 0x00000275
PC 0x0000035c
Z 1

```

Exempel: stack och subrutiner, forts. 12

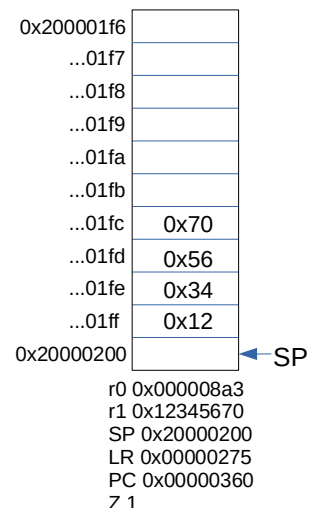
```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
→ 00035c    POP    {r1}
000360    BX     LR

```

- Pop av värde tillbaks till r1
 - Kopiera M[0x200001fc],M[0x200001fd], M[0x200001fe],M[0x200001ff] till r1
 - Öka SP med 4



Exempel: stack och subrutiner, forts. 13

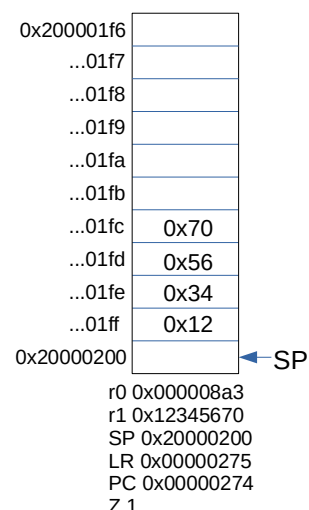
```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...

000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
→ 00035c    POP    {r1}
000360    BX     LR

```

- Återhopp från subrutin
 - Kopiera LR till PC, nollställ LSB i PC

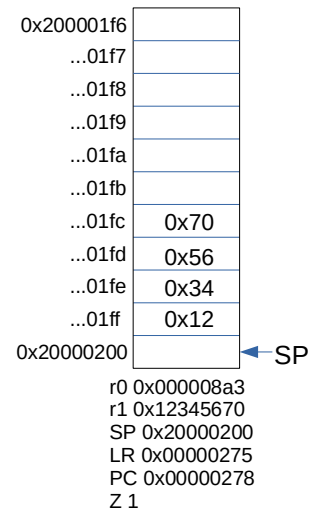


Exempel: stack och subrutiner, forts. 14

```

; Antagande. SP = 0x20000200,
; r1=0x12345670 från main
00026c    MOV    r0,#66
000270    BL     0x00000344
000274    ...
000344    PUSH   {r1}
000348    MOV    r1,#0
00034c    ADD    r1,r1,r0
000350    SUBS   r0,r0,#1
000354    BNE    0x00034c
000358    MOV    r0,r1
00035c    POP    {r1}
000360    BX     LR

```



- Fortsätter med resten av programmet

- Subrutinen har beräknat
- r1 har använts men återställts

$$r0 = \sum_{x=1}^{r0} x$$

Vanliga fel vid programmering

- Glömma # eller 0x
- Ett # eller 0x för mycket (på fel ställe)
- Glömma ange ordlängd på instruktion
 - LDR istället för LDRB respektive STR istället för STRB
- Läsa/skriva 16 eller 32 bitars ord på udda adresser
- Tar bort fel antal element på stacken före retur från subrutin
- Villkorliga subrutinanrop (finns inte i vår CPU!)
- Inte avsluta subrutin med bx lr

Exempel på olämplig/felaktig kod

- Verkar försöka kontrollera om $r2=r3$ och skriva ut meddelande annars kopieras värde in i $r0$
- Just nu skrivs det ut korrekt 1 gång
- Ser ni problemen?
- Programmet hänger sig till slut

```

huvudprogrammet      ; Ej fungerande variant
    bl calculate
...
calculate             ; En subrutin
    push {lr}
    add r1,r1,r0
    add r1,r1,r2
    cmp r2,r3
    beq printmessage
    mov r3,r0
    pop {pc}

printmessage          ; En till subrutin
    adr r4,errorstring ; peka på sträng
    bl printit
printit:              ; Ytterligare subrutin
... ; Lite mer kod här...
    b huvudprogrammet
  
```

Källor till problem i exemplet

- Svår att felsöka
- Subrutiner hoppar till varandra via BEQ/BNE
- Viktigt fel: Växande stack
- Återhopp till huvudprogram sist i subrutin utan att återställa stack
- Till slut kommer stacken skriva över data och/eller program
- Sparar inte LR innan subrutin anropas

```

huvudprogrammet      ; Ej fungerande variant
    bl calculate
...
calculate             ; En subrutin
    push {lr}
    add r1,r1,r0
    add r1,r1,r2
    cmp r2,r3
    beq printmessage
    mov r3,r0
    pop {pc}

printmessage          ; En till subrutin
    adr r4,errorstring ; peka på sträng
    bl printit
printit:              ; Ytterligare subrutin
... ; Lite mer kod här...
    b huvudprogrammet
  
```

Bättre variant

```
Huvudprogrammet ; Fungerande variant
bl calculate
...
calculate: ; En subrutin
push {lr}
add r1,r1,r0
add r1,r1,r2
cmp r2,r3
bne dontprint
bl printmessage
dontprint:
pop {lr}
bx lr

printmessage ; En till subrutin
push {lr}
adr r4,errorstring ; peka på sträng
bl printit
pop {lr}
bx lr

printit: ; Ytterligare subrutin
... ; Lite mer kod här...
bx lr
```

olämplig/felaktig kod

```
huvudprogrammet ; Ej fungerande variant
bl calculate
...
calculate ; En subrutin
push {lr}
add r1,r1,r0
add r1,r1,r2
cmp r2,r3
beq printmessage
mov r3,r0
pop {pc}

printmessage ; En till subrutin
adr r4,errorstring ; peka på sträng
bl printit

printit: ; Ytterligare subrutin
... ; Lite mer kod här...
b huvudprogrammet
```

Varför den nya versionen är bättre

- En subrutin bör göra enbart en sak och sedan hoppa tillbaka
- Om fler saker behöver göras låt subrutinen ropa på flera andra subrutiner
- Detta angreppssätt möjliggör återanvändning av rutiner
- Enda väg in i en subrutin: BL
- Enda vägen ut ur en subrutin: BX LR
- Alla andra hopp inom subrutin eller huvudprogrammet

```
huvudprogrammet ; Fungerande
variant
bl calculate
...
calculate: ; En subrutin
push {lr}
add r1,r1,r0
add r1,r1,r2
cmp r2,r3
bne dontprint
bl printmessage
dontprint:
pop {lr}
bx lr

printmessage ; En till subrutin
push {lr}
adr r4,errorstring ; peka på sträng
bl printit
pop {lr}
bx lr

printit: ; Ytterligare subrutin
... ; Lite mer kod här...
bx lr
```

Live demo av ccstudio via thinlinc + ssh

Logga in på thinlink
ssh -X muxen2-103.edu.liu.se
module load courses/TSEA28
tsea28_active
ccstudio



Görs bara om ni INTE är i labbet

Skapa tomt ccsproject
Lägg till lab1.asm från /courses/TSEA28
(eller ladda ned från labbsidan)
Editera lab1.asm
Assemblera lab1.asm

- Titta på inspelning som finns under kursdokument i kursrummet på lisam
Labbförberedelse_hemifrån_2021.mp4 (OBS MUXEN2-101 t om MUXEN2-116 numera)

Live demo av Darma (med kort)

Logga in på maskin I MUX1/MUX2
module load courses/TSEA28
ccstudio
öppna tidigare projekt
Starta debugläge
Go main
Kör programmet
Undersök register
Undersök minne
Ändra register
Stega igenom
Sätt brytpunkt

Kommentar: Måste köra initiering
av port innan porten
kan läsas/skrivas

