Assemblerprogrammering - fördjupning

Ur innehållet:

- ”Trampoliner” – tabellerade funktionsadresser
- Aktiveringspost med ARM Cortex M4
- Mer om parameteröverföring
- Registerspill
- Kodgenerering - ISA
- "Kodoptimering"

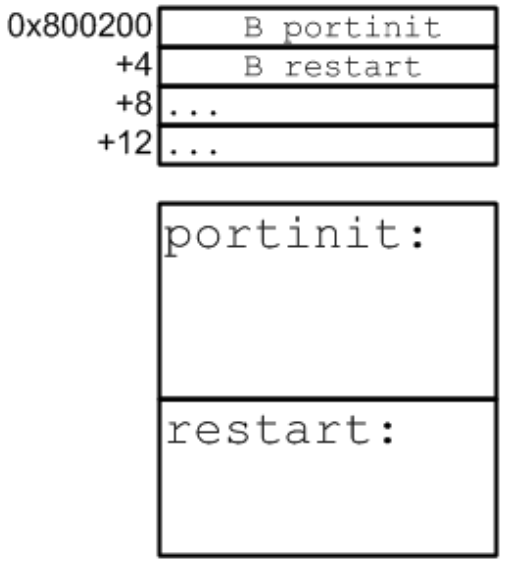
Pekare till C-funktioner på fast adress i minnet

```
void func(void);      /* funktion utan parametrar och returvärde */

void * func(void);    /* funktion utan parametrar,  returnerar pekare */

void (*func)(void);   /* pekare till funktion utan parametrar och returvärde */
```

EXEMPEL:
En parameterlös subrutin `portinit` för att initiera hårdvaran i MD407 finns med ingång på adress adress 0x08000200 i minnet. Visa hur denna kan anropas:

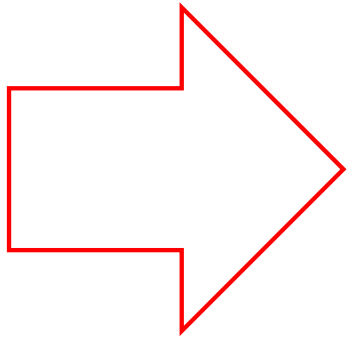
a) från ett C-program
b) i assemblerspråk

Vi löser på tavlan...



Mer ”trampoliner” – ”Switch/case”

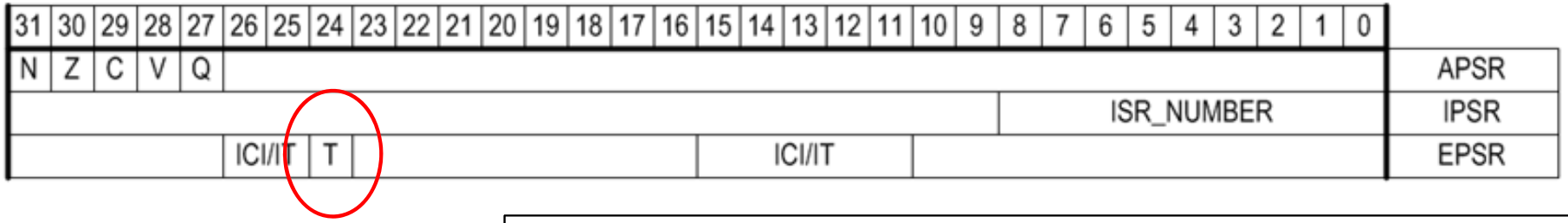
```
…
switch( a ) {
  case 0:  x = y + z;
           break;
  case 1:  x = y * z;
           break;
  case 3:  x = y – z;
           break;
  default: x = 0;
           break;
}
…
```



```
…
    LDR     R0,=switch_a
    LDR     R1,a
    LSL     R1,#2
    LDR     R2,[R0,R1]
    BX      R2

case_0: ...
    B       case_exit
case_1: ...
    B       case_exit
case_3: ...
    B       case_exit
case_default: ...
    B       case_exit
case_exit:
 ...

switch_a:
   .WORD    case_0+1
   .WORD    case_1+1
   .WORD    case_3+1
   .WORD    case_default+1
a: .SPACE   4
```

Instruktioner som ändrar PC: Thumb-state

Flera olika ARM-processorer kan exekvera såväl 32-bitars ARM som Thumb- instruktions-uppsättningar. Man kan då blanda ARM- och Thumb-kodade funktioner.
För alla instruktioner som påverkar PC kopieras bit A0 till EPSR, bit T. Då denna är 1 tolkas instruktionen på denna adress som Thumb-instruktion, annars tolkas den som en ARM-instruktion



Jämför med uppgift 6.2 i arbetsboken.

UPPGIFT 6.2 "INVALID STATE" UNDANTAG

Följande kod illustrerar vad som händer då man försöker att, felaktigt, ändra exekveringstillstånd från *Thumb* till *ARM*.

```
@ invstate.asm
main:
   LDR   R0,=func1
   BLX   R0
   LDR   R0,=func2
   BLX   R0
   .thumb_func    @ anger att func1 ska exekveras i "Thumb-state"
func1:
   BX LR
func2:
   BX LR
```

Lagringsklasser och synlighet

Variabler med permanent adress i minnet:

```
int gi;     /* global synlighet */

static int si;    /* synlig i denna källtext */

void sub(void)   {
    static int si; /* synlig i funktionen sub */
}
```

I assemblerspråk:

```
            .GLOBL    gi
gi:         .SPACE    4
.si:        .SPACE    4
.sub_si:    .SPACE    4
```

alternativt sätt för global variabel:

```
            .comm gi,4,4
```

(`.comm` *`sym`*`,`*`length`*`,`*`alignment`*`)`

Symboler med begränsad synlighet tilldelas *unika* namn av kompilatorn.
Sådana kompilatorgenererade symbolnamn dyker ofta upp med inledande '.' i assemberkälltexten.
Samma C-symboler kan därför användas i olika sammanhang utan konflikt

Permanenta och tillfälliga variabler

Exempel:

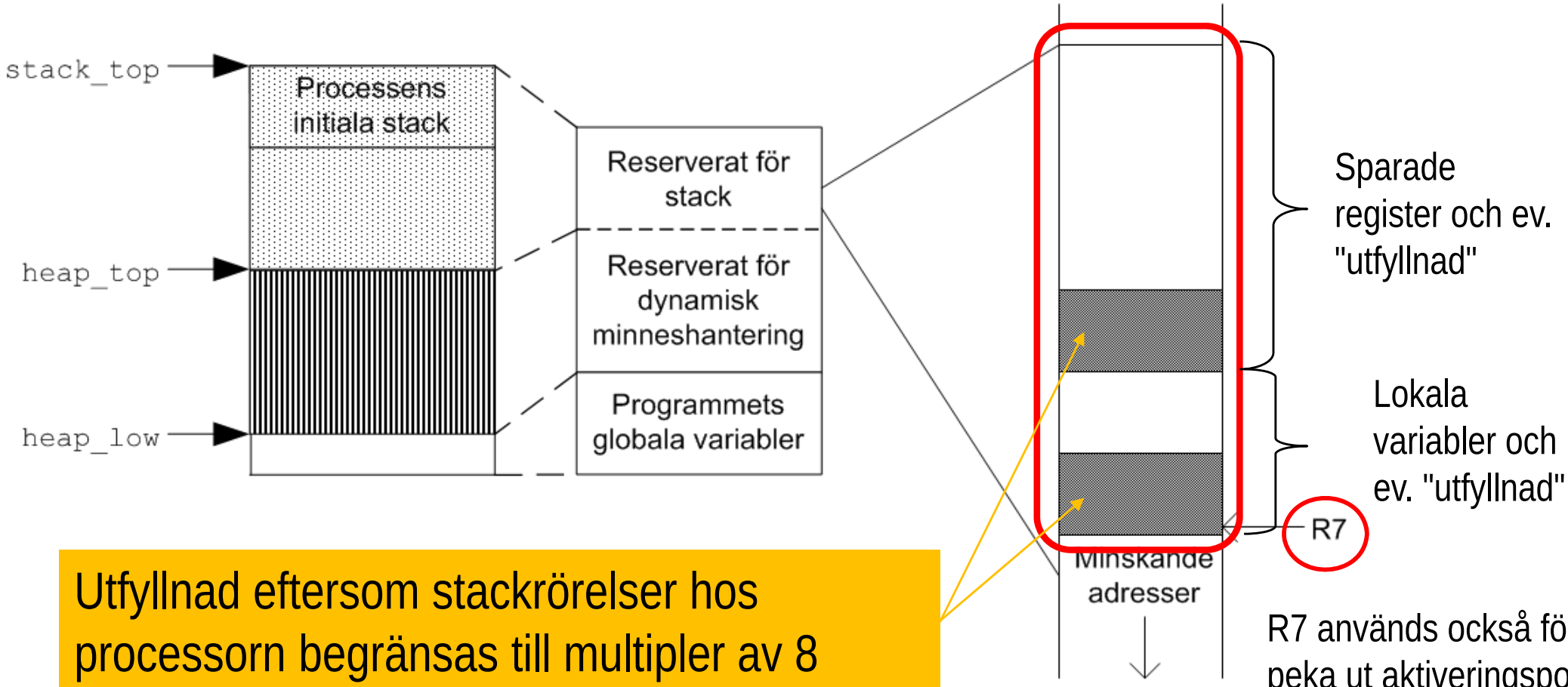
```
char  gc;

void  sub( void )
{
  char   lc;
  lc = 5;
}
```

```
        .comm gc,1,1
sub:
        PUSH  {R7,LR}
        SUB   SP,SP,#8
        MOV   R7,SP
        ADD   R3,R7,#7   @ R3 = &lc
        MOV   R2,#5
        STRB  R2,[R3]
        MOV   SP,R7
        ADD   SP,SP,#8
        POP   {R7,PC}
```

Av prestandaskäl, stacken ändras i multiplar av 8.
Här används därför 8 bytes för en variabel som egentligen bara kräver 1 byte

I stället för "registerallokering" av `lc` kan man reservera en position på stacken:
I subrutinen refereras här då variabeln `lc` som **`[r7,#7]`.**

Aktiveringspost, ARM Cortex M4



Utfyllnad eftersom stackrörelser hos processorn begränsas till multipler av 8

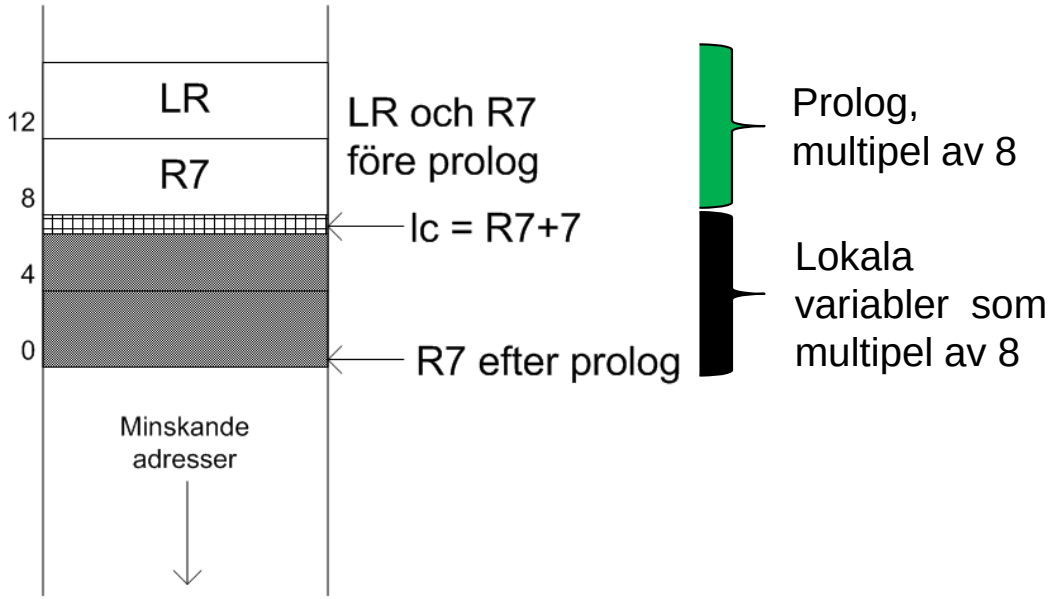
Aktiveringspost – lokal variabel

Observera att såväl stackens utrymme för lokala variabler, som *prolog*, måste vara en multipel av 8

```
sub:
    PUSH    {R7,LR}
    SUB     SP,SP,#8
    MOV     R7,SP
    ADD     R3,R7,#7
    MOV     R2,#5
    STRB    R2,[R3]
    MOV     SP,R7
    ADD     SP,SP,#8
    POP     {R7,PC}
```

prolog

epilog



Parameteröverföring till, och returvärden från subrutiner

Parametrar, två metoder kombineras

Via register: snabbt, effektivt och enkelt, begränsat antal
Via stacken: generellt

Returvärden

Via register: för enkla datatyper som ryms i processorns register `R0` och ev. `R1`
Via stacken: sammansatta datatyper (poster och fält)

Register	Användning	
R15 (PC)	Programräknare	
R14 (LR)	Länkregister	
R13 (SP)	Stackpekare	
R12 (IP)	Dessa register är avsedda för variabler och som temporära register. Om dom används måste dom sparas och återställas av den anropade (*callee*) funktionen	
R11		
R10		
R9		
R8		
R7	Speciellt använder GCC R7 som pekare till aktiveringspost (*stack frame*) Också dessa register är avsedda för variabler och temporärbruk Om dom används måste dom sparas och återställas av den anropade (*callee*) funktionen	
R6		
R5		
R4		
R3	parameter 4 / temporärregister	Dessa register sparas normalt sett inte över funktionsanrop men om, så är det den anropande (*caller*) funktionens uppgift
R2	parameter 3 / temporärregister	
R1	parameter 2 / resultat 2 /temporärregister	
R0	parameter 1 / resultat 1/ temporärregister	

Detaljerna styrs av konventioner ”application binary interface” (ABI) ”Procedure call standard”

Parameteröverföring via register

Konventionerna sägar att vi använder register R0,R1,R2,R3,
(i denna ordning) för parametrar som skickas till en subrutin.
Övriga parametrar läggs på stacken.

Exempel:

```
int     a,b,c,d;
```

Följande funktionsanrop görs:

```
sub(a,b,c,d);
```

Tänkbar komplikation: ”Registerspill"

Dvs. register behövs i den anropade subrutinen.

Visa hur en prolog skapar en aktiveringspost med kopior av parametrarna. Dvs. Skapa ”tillfälliga variabler” – och använd stacken för temporär lagring.

Vi löser på tavlan

Parameteröverföring via stacken

Exempel: Följande deklarationer är gjorda:

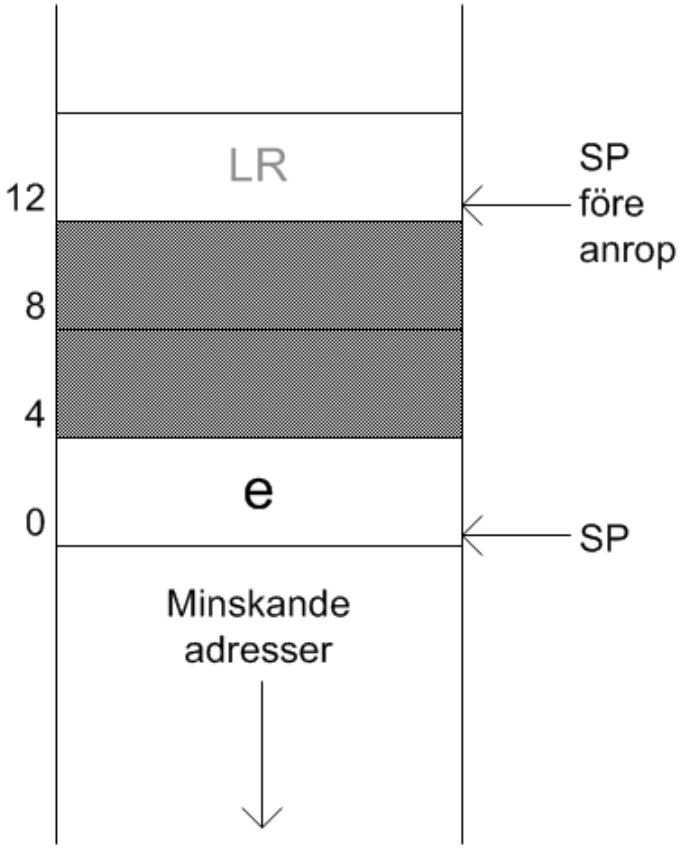
```
int      a,b,c,d,e;
```

Visa hur följande funktionsanrop kodas i assemblerspråk:

```
sub(a,b,c,d,e);
```

Lösning:

```
...
SUB    SP,SP,#12
LDR    R3,e
STR    R3,[sp]
LDR    R0,a
LDR    R1,b
LDR    R2,c
LDR    R3,d
BL     sub
ADD    SP,SP,#12
...
```



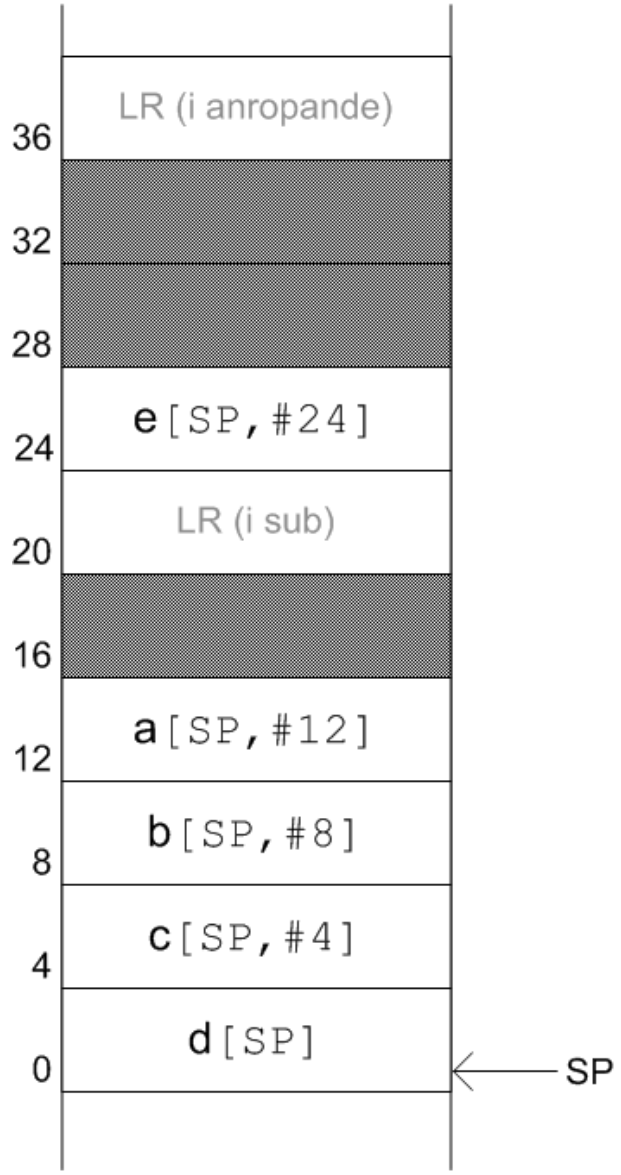
Stackens utseende i sub

Funktionen:

```
void sub(int a,int b,int c,int d,int e);
```

I exemplet har funktionerna inga lokala variabler (endast LR på stacken)

Aktiveringsposten kan sedan utvidgas ytterligare med lokala variabler.



Returvärden via register

Storlek	C-typ	Register
8-32 bitar	char/short/int	R0
64 bitar	long long	R1:R0

Exempel: Visa hur följande funktionsanrop kodas i assemblerspråk:

```
int sub(int a)
{
   return a+1;
}
```

respektive

```
long long sub(int a)
{
   return a+1;
}
```

Lösning:

```
@ (int)
sub:
        ADD    R0,R0,#1
        BX     LR
@ (long long)
sub:
        ADD    R0,R0,#1
        MOV    R1,#0
        ADC    R1,R1,R1
        BX     LR
```

Returvärden via stack

Krävs typiskt då en subrutin ska returnera en komplett post, eller ett helt fält. Den anropande funktionen ska då först reservera utrymme för returvärdet. En pekare till detta utrymme läggs sedan som första parameter vid anropet:

Exempel:

```
typedef struct coord COORD;
typedef struct coord{
     int x,y,z;
     COORD *ptr;
} COORD;

COORD sub( void )
{
    static COORD start;
    return start;
}
```

GCC genererar följande kod:

```
@ förbered stack för returvärde
    SUB     SP, SP,#16
    MOV     R0,SP
    BL      sub
    …

sub:
    PUSH    {R4,R7}
    LDR     R2,=start
    MOV     R4,R0
    LDMIA   R2,{R0,R1,R2,R3}
    STMIA   R4,{R0,R1,R2,R3}
    MOV     R0,R4
    POP     {R4,R7}
    BX      LR
```

Kodgenerering, C - ISA

`-mthumb`

`-march=armv6-m`

`-march=armv7-m`

`-march=armv7e-m`

`-mfloat-abi=hard`
`-mfpu=fpv4-sp-d16`

Instruktion	Storlek	Cortex M0	Cortex M0+	Cortex M1	Cortex M3	Cortex M4	Cortex M7	Ark.
ADC, ADD, (ADR), AND, ASR, B, BIC, BKPT, BLX, BX, CMN, CMP, CPS, EOR, LDM, (LDMIA, LDMFD), LDR, LDRB, LDRH, LDRSB, LDRSH, LSL, LSR, MOV, MUL, MVN, NOP, ORR, POP, PUSH, REV, REV16, REVSH, ROR, RSB, SBC, SEV, STM, (STMIA, STMEA), STR, STRB, STRH, SUB, SVC, SXTB, SXTH, TST, UXTB, UXTH, WFE, WFI, YIELD	16-bit	x	x	x	x	x	x	v6
BL, DMB, DSB, ISB, MRS, MSR	32-bit	x	x	x	x	x	x	
CBNZ, CBZ, IT	16-bit				x	x	x	v7
ADC, ADD, AND, ASR, B, BFC, BFI, BIC, CDP, CLREX, CLZ, CMN, CMP, DBG, EOR, LDC, LDMA, LDMDB, LDR, LDRB, LDRBT, LDRD, LDREX, LDREXB, LDREXH, LDRH, LDRHT, LDRSB, LDRSBT, LDRSHT, LDRT, MCR, LSL, LSR, MLS, MCRR, MLA, MOV, MOVT, MRC, MRRC, MUL, MVN, NOP, ORN, ORR, PLD, PLDW, PLI, POP, PUSH, RBIT, REV, REV16, REVSH, ROR, RRX, RSB, SBC, SBFX, SDIV, SEV, SMLAL, SMULL, SSAT, STC, STMDB, STR, STRB, STRBT, STRD, STREX, STREXB, STREXH, STRH, STRHT, STRT, SUB, SXTB, SXTB, TBB, TBH, TEQ, TST, UBFX, UDIV, UMLAL, UMULL, USAT, UXTB, UXTH, WFE, WFI,YIELD	32-bit				x	x	x	
PKH, QADD, QADD16, QADD8, QASX, QDADD, QDSUB, QSAX, QSUB, QSUB16, QSUB8, SADD16, SADD8, SASX, SEL SHADD16, SHADD8, SHASX, SHSAX, SHSUB16, SHSUB8, SMLABB, SMLABT, SMLATB,SMLATT, SMLAD, SMLALBB, SMLALBT, SMLALTB, SMLALTTSMLALD, SMLAWB, SMLAWT, SMLSD, SMLSLD, SMMLA, SMMLS, SMMUL, SMUAD, SMULBB, SMULBT, SMULTT, SMULTB, SMULWT, SMULWB, SMUSD, SSAT16, SSAX, SSUB16, SSUB8, SXTAB, SXTAB16, SXTAH, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSAX, UHSUB16, UHSUB8, UMAAL, UQADD16, UQADD8, UQASX, UQSAX, UQSUB16, UQSUB8, USAD8, USADA8, USAT16, USAX, USUB16, USUB8, UXTAB, UXTAB16, UXTAH, UXTB16	32-bit					x	x	v7e (DSP)
VABS, VADD, VCMP, VCMPE, VCVT, VCVTR, VDIV, VLDM, VLDR, VMLA, VMLS, VMOV, VMRS, VMSR, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VPOP, VPUSH, VSQRT, VSTM, VSTR, VSUB	32-bit					SP FPU	SP FPU	32-b FP
FP- dubbel precision	32-bit						DP FPU	64-b FP

Kodgenereringen styrs med flaggor till kompilatorn, viktiga parametrar är

- val av instruktionsuppsättning (ISA) för anpassning till aktuell måldator
- kodförbättring, dvs. optimering m.a.p något kriterie.

Kodoptimering - optimalitetskriterier

`-O0` optimera med avseende på kod som ska "debuggas". Innebär att stacken används för lagring av såväl parametrar som lokala variabler, inte speciellt "bra" kod.

`-O` Optimera för snabb kod, kan t.e.x "rulla upp" små loopar...

`-Os` Optimera för minimal kodstorlek

`-O1` Optimera mera för snabb kod

`-O2` Optimera ännu mera för snabb kod

`-O3` Optimera fullt för snabb kod

`-fexpensive-optimization` Tillämpa även de mest tidsödande optimeringsmetoderna

```
littleloop: -O0
  PUSH    {R7,LR}
  SUB     SP,SP,#8
  ADD     R7,SP,#0
  MOV     R3,#0
  STR     R3,[R7,#4]
  B       .L14
.L15:
  LDR     R3,=g
  LDR     R3,[R3]
  ADD     R2,R3, #1
  LDR     R3,=g
  STR     R2,[R3]
  LDR     R3,[R7,#4]
  ADD     R3,R3,#1
  STR     R3,[R7,#4]
.L14:
  LDR     R3,[R7,#4]
  CMP     R3,#3
  BLE     .L15
  MOV     SP,R7
  ADD     SP,SP,#8
  POP     {R7,PC}
```

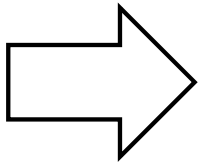
```
int g;
void littleloop(void)
{
  for(i=0;i<4;i++)
    g++;
}
```

```
littleloop: (-O3)
 LDR     R2,=g
 LDR     R3,[R2]
 ADD     R3,R3,#4
 STR     R3,[R2]
 BX      LR
```

Kodoptimering - flyktiga variabler

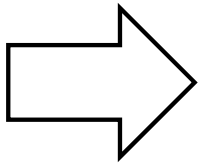
Globala variabeln `delay_count` uppdateras av en avbrottsrutin

```
static volatile int delay_count;
void main(void)
{
  while(1)
  {
    if( delay_count == 0)
      break;
  }
}
```



```
main:
  LDR    R2,=delay_count
.L21:
  LDR    R3,[R2]
  CMP    R3,#0
  BNE    .L21
  BX     LR
```

```
static int delay_count;
void main(void)
{
  while(1)
  {
    if( delay_count == 0)
      break;
  }
}
```



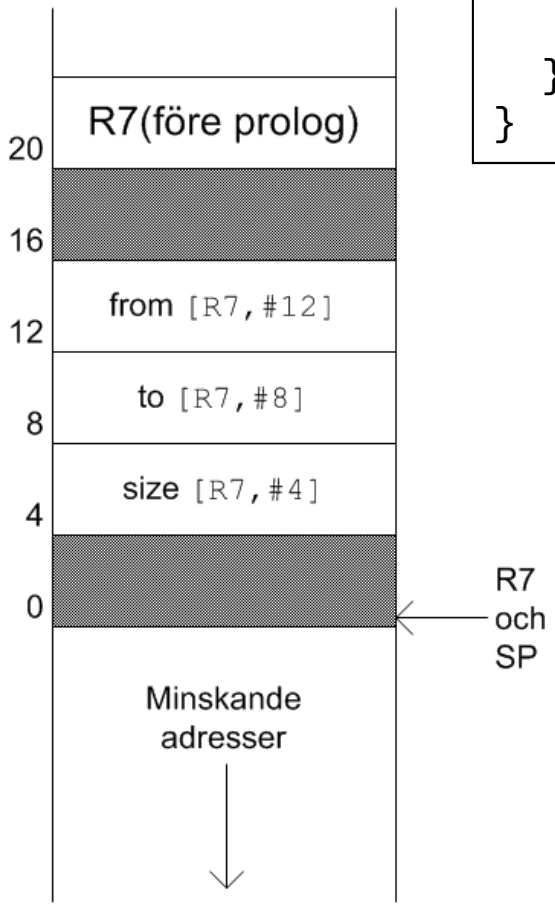
```
main:
  LDR    R3,=delay_count
  LDR    R3,[R3]
  CMP    R3,#0
  BEQ    .L20
.L23:
  B      .L23
.L20:
  BX     LR
```

Kodoptimering – "för hand" - exempel

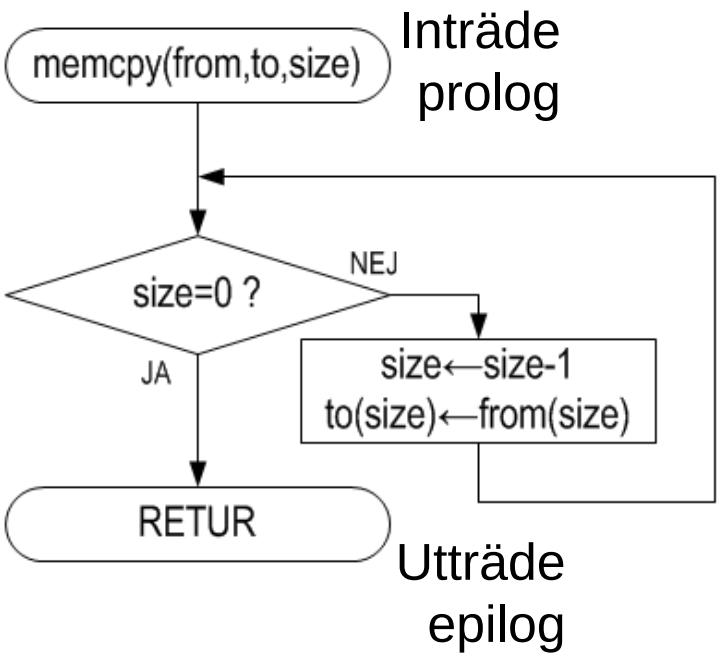
Anropssekvens: (formellt):

```
LDR     R0,=from
LDR     R1,=to
LDR     R2,size
BL      memcpy
```

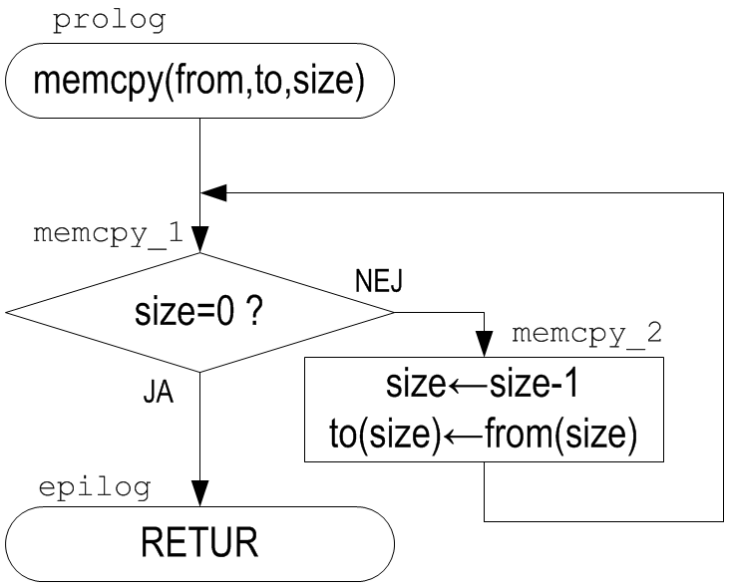
Stackens utseende i ”memcpy” efter prolog



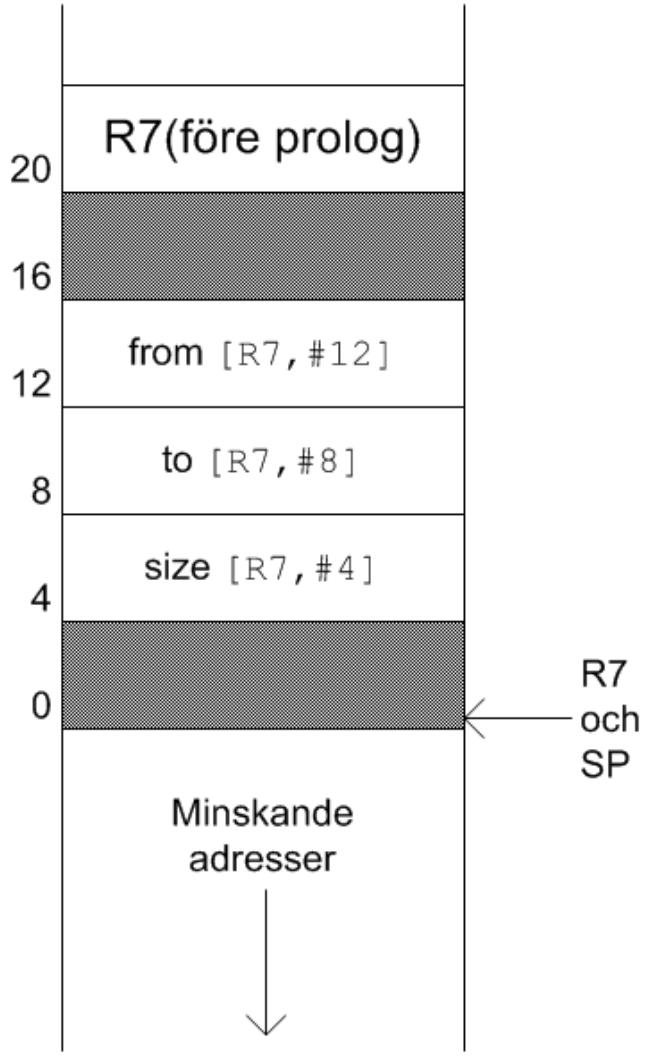

```
void memcpy(  unsigned char from[],
    unsigned char to[],
    unsigned int size )
{
  while (size > 0){
    size = size - 1;
    to[size] = from[size];
  }
}
```



Kompilatorflagga –O0 ger "Naiv kodning", av blocken




```
memcpy:
memcpy_prolog:
      PUSH  {R7}
      SUB   SP,SP,#20
      MOV   R7,SP
      STR   R0,[R7,#12]
      STR   R1,[R7,#8]
      STR   R2,[R7,#4]
memcpy_1:
      LDR   R3,[R7,#4]
      CMP   R3,#0
      BEQ   memcpy_epilog
memcpy_2:
      LDR   R3,[R7,#4]
      SUB   R3,R3,#1
      STR   R3,[R7,#4]
      LDR   R2,[R7,#8]
      LDR   R3,[R7,#4]
      ADD   R3,R3,R2
      LDR   R1,[R7,#12]
      LDR   R2,[R7,#4]
      ADD   R2,R2,R1
      LDRB  R2,[R2]
      STRB  R2,[R3]
      B     memcpy_1
memcpy_epilog:
      ADD   SP,SP,#20
      POP   {R7}
      BX    lr
```



Kodförbättring, ingen onödig minnesanvändning, dvs. *registerallokering*, "förbättrad för hand"

Registerallokering:
`R0:` `from`, ersätter [R7,#12]
`R1:` `to` , ersätter [R7,#8]
`R2:` `size` , ersätter [R7,#4]
`R3:` temporär
`R4:` temporär

Vi gör dessa substitutioner och kommenterar därefter ut de instruktioner som då blir överflödiga...

```
memcpy:
memcpy_prolog:
      PUSH  {R7}
      SUB   SP,SP,#20
      MOV   R7,SP
      STR   R0,[R7,#12]
      STR   R1,[R7,#8]
      STR   R2,[R7,#4]
memcpy_1:
      LDR   R3,[R7,#4]
      CMP   R3,#0
      BEQ   memcpy_epilog
memcpy_2:
      LDR   R3,[R7,#4]
      SUB   R3,R3,#1
      STR   R3,[R7,#4]
      LDR   R2,[R7,#8]
      LDR   R3,[R7,#4]
      ADD   R3,R3,R2
      LDR   R1,[R7,#12]
      LDR   R2,[R7,#4]
      ADD   R2,R2,R1
      LDRB  R2,[R2]
      STRB  R2,[R3]
      B     memcpy_1
memcpy_epilog:
      ADD   SP,SP,#20
      POP   {R7}
      BX    lr
```

```
memcpy:
memcpy_prolog:
      PUSH  {R4,LR}
@     SUB   SP,SP,#20
@     MOV   R7,SP
@     STR   R0,[R7,#12]
@     STR   R1,[R7,#8]
@     STR   R2,[R7,#4]
memcpy_1:
@     LDR   R3,[R7,#4]
      CMP   R2,#0
      BEQ   memcpy_epilog
memcpy_2:
@     LDR   R3,[R7,#4]
      SUB   R3,R3,#1
@     STR   R3,[R7,#4]
@     LDR   R2,[R7,#8]
@     LDR   R3,[R7,#4]
      MOV   R3,R0
      ADD   R3,R3,R2
      MOV   R4,R1
      ADD   R4,R4,R2
@     ADD   R3,R3,R2
@     LDR   R1,[R7,#12]
@     LDR   R2,[R7,#4]
@     ADD   R2,R2,R1
      LDRB  R3,[R3]
      STRB  R3,[R4]
      B     memcpy_1
memcpy_epilog:
@     ADD   SP,SP,#20
      POP   {R4,PC}
@     BX    lr
```

vi jämför med GCC...

```
void memcpy(  unsigned char from[],
    unsigned char to[],
    unsigned int size )
{
  while (size > 0){
    size = size - 1;
    to[size] = from[size];
  }
}
```

vår förbättrade kod..

```
memcpy:
memcpy_prolog:
     PUSH{R4,LR}
memcpy_1:
     CMP       R2,#0
     BEQ       memcpy_epilog
memcpy_2:
     SUB       R3,R3,#1
     MOV       R3,R0
     ADD       R3,R3,R2
     MOV       R4,R1
     ADD       R4,R4,R2
     LDRB      R3,[R3]
     STRB      R3,[R4]
     B         memcpy_1
memcpy_epilog:
     POP       {R4,PC}
```

GCC-genererad kod (-O3)

```
@void memcpy(  unsigned char from[],
@       unsigned char to[],
@       unsigned int size )
memcpy:
     B         .L1
.L2:
     SUB       R2,R2,#1
     LDRB      R3,[R0,R2]
     STRB      R3,[R1,R2]
.L1:
     CMP       R2,#0
     BNE       .L2
     BX        LR
```

Vi inser att vi får svårt att göra jobbet bättre än kompilatorn...

Fler exempel på kodförbättring...

- Code motion
  - För operationer på ”loop-invarianter”
- Dead code elimination
  - Kod som aldrig exekveras tas bort
- Out-of-order
  - Instruktioner arrangeras om för att undvika ”stall”
  - Villkorliga brancher (branch prediction)
- Loop unrolling
  - Undvik iterationer – ersätt med repetitioner.