

# Maskinorienterad programmering, fördjupning

## Ur innehållet:

- Mer om parameteröverföring och registerspill
- Aktiveringspost med ARM Cortex M4
- Kodgenerering - ISA
- "Kodoptimering"

## Målsättningar:

- Att få en större förståelse för maskinorienterad programmering

## Tillfälliga variabler på stacken

I stället för registerallokering av lokala variabler kan man reservera en position på stacken.

**Exempel:**

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

f:

```
PUSH    {LR}
SUB     SP, SP, #1
MOV     R2, #5
ADD     R3, SP, #0    @ R3 = &lc
STRB    R2, [R3]
ADD     SP, SP, #1
POP     {PC}
```

I subrutinen refereras här då variabeln **lc** som **[SP, #0]**.

# Aktiveringspost – prolog och epilog

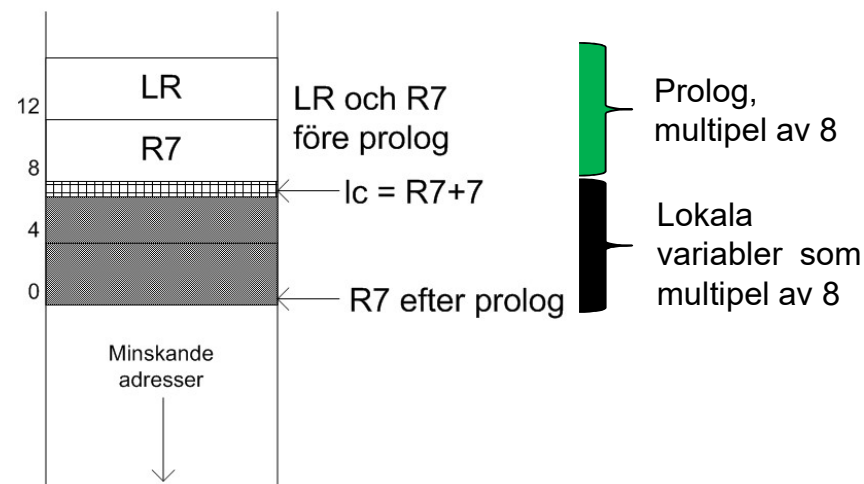
Men `SUB SP, SP, #1` fungerar inte speciellt bra. Stacken *måste* vara rättad för nästa eventuella "push"-operation. Hårdvaran har optimerats för stackrörelser i multipler av 8.

Utrymmet som skapas på stacken kallas *aktiveringspost (stack frame)*. För att inte användningen av SP ska låsas upp använder GCC register R7 som pekare till aktiveringsposten.

```
f:
    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}
```

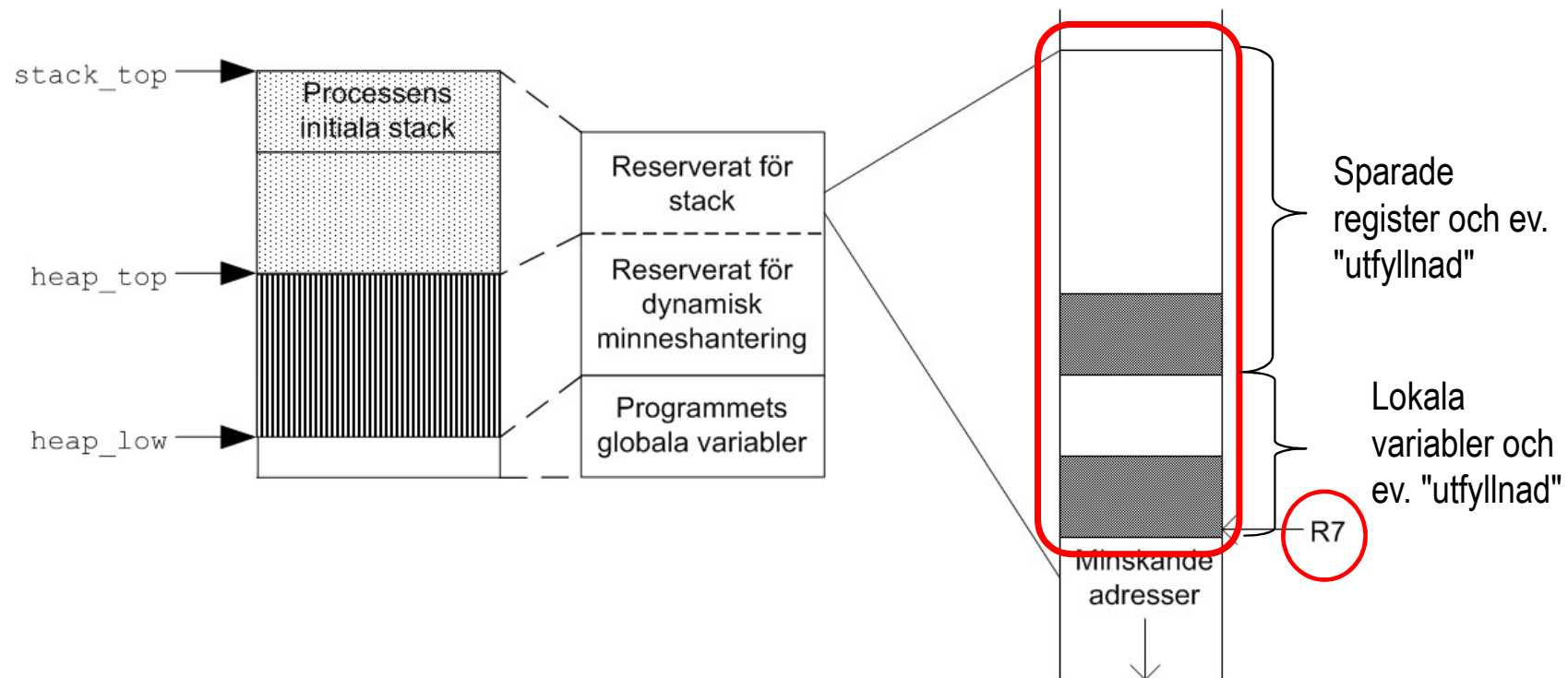
prolog

epilog



Stackens totala utrymme för den lokala variabeln blir här 8 bytes, 7 bytes är "utfyllnad"

# Aktiveringspost, ARM Cortex M4



# 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.<br>Om dom används måste dom sparas och återställas av den anropade ( <i>callee</i> ) funktionen |                                                                                                                                   |
| R11      |                                                                                                                                                                     |                                                                                                                                   |
| R10      |                                                                                                                                                                     |                                                                                                                                   |
| R9       |                                                                                                                                                                     |                                                                                                                                   |
| R8       |                                                                                                                                                                     |                                                                                                                                   |
| R7       | Speciellt använder GCC R7 som pekare till aktiveringspost ( <i>stack frame</i> )                                                                                    |                                                                                                                                   |
| R6       | Också dessa register är avsedda för variabler och temporärbruk                                                                                                      |                                                                                                                                   |
| R5       | Om dom används måste dom sparas och återställas av den anropade ( <i>callee</i> ) funktionen                                                                        |                                                                                                                                   |
| R4       |                                                                                                                                                                     |                                                                                                                                   |
| R3       | parameter 4 / temporärregister                                                                                                                                      | Dessa register sparas normalt sett inte över funktionsanrop men om, så är det den anropande ( <i>caller</i> ) 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 stacken

**Exempel:** Följande deklarationer är gjorda:

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

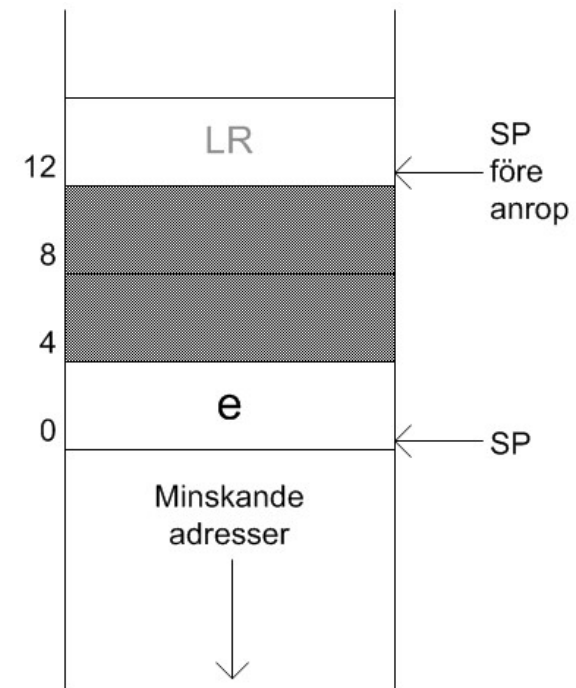
Visa hur följande funktionsanrop kodas i assemblerspråk:

```
f(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     f  
ADD    SP, SP, #12  
...
```

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

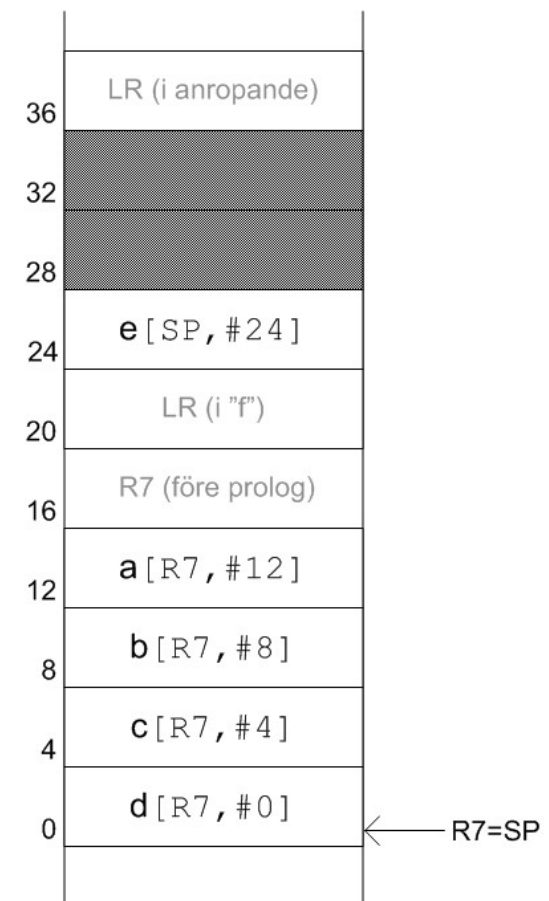


# Tänkbar komplikation: "Registerspill"

Dvs. register som används för parametrar behövs i den anropade subrutinen.

Vi löser problemet genom att, i prologen, spara även parametrarna i aktiveringsposten:

```
f:
    PUSH {LR,R7}
    SUB   SP, SP, #16
    MOV   R7, SP
    STR   R0, [R7, #12]
    STR   R1, [R7, #8]
    STR   R2, [R7, #4]
    STR   R3, [R7, #0]
    ...
    ...
    ADD   SP, SP, #16
    POP   {PC, R7}
```



# Returvärden via register

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

# Returvärden via stack

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

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

GCC genererar följande kod:

```
@ förbered stack för returvärde
SUB    SP, SP, #16
MOV    R0, SP
BL     f
...

f:
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
```

-mthumb

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

Maskinorienterad programmering - fördjupning, klipp 3, 2021:1

# 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++;
}
    
```

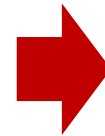
```

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

# Kodoptimering - flyktiga variabler

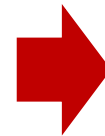
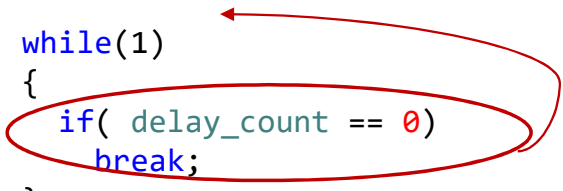
Globala variabeln `delay_count` uppdateras av en avbrottsrutin

```
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;
    }
}
```



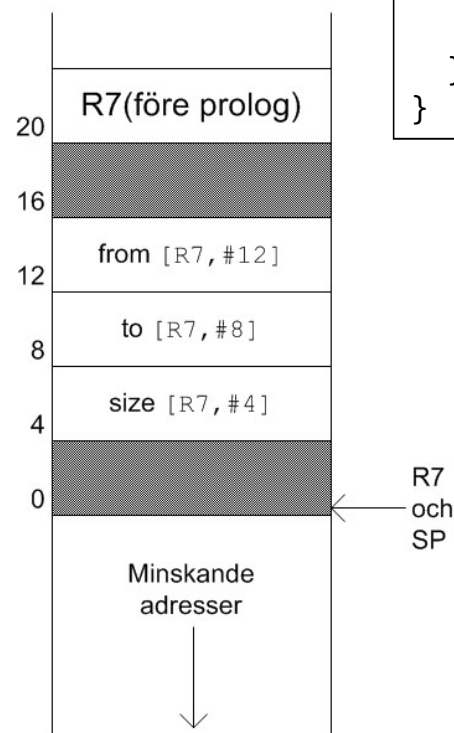
```
main:
    LDR    R3,=delay_count
    LDR    R3,[R3]
    CMP    R3,#0
    BEQ    .L20
.L23:
    B      .L23
.L20:
    BX     LR
```

# Kodoptimering – "för hand" - exempel

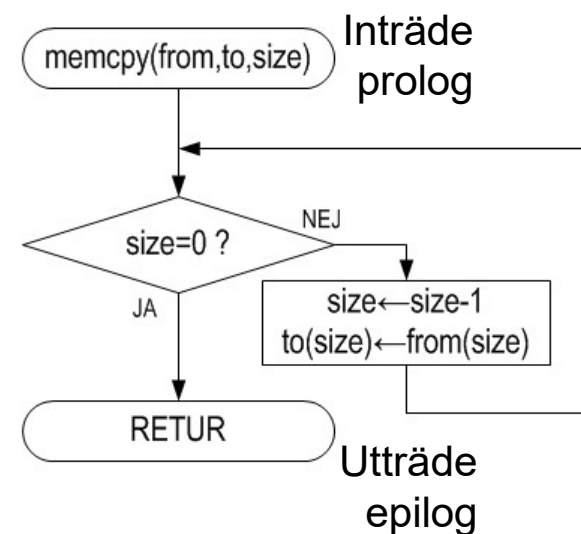
Anropssekvens: (formellt):

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

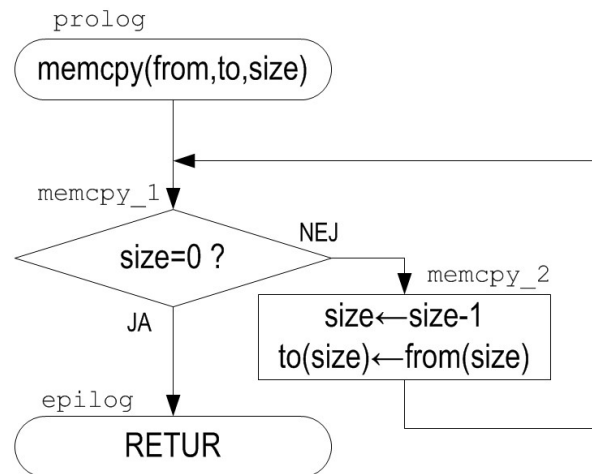
Stackens utseende  
i "memcpy" efter  
prolog



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



## Kompilatorflagga -O0 ger "Naiv kodning", av blocken



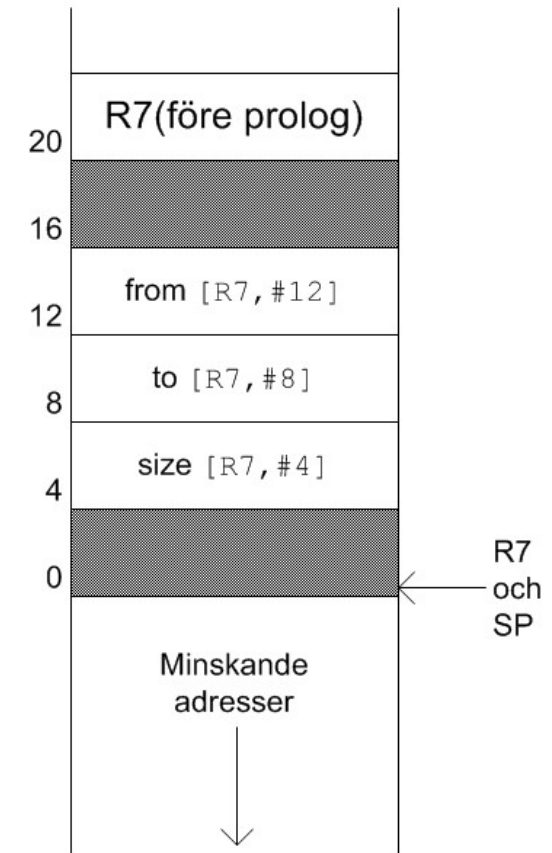
```

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 R2,R2,#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...

vår förbättrade kod..

```
memcpy:
memcpy_prolog:
    PUSH {R4,LR}
memcpy_1:
    CMP     R2,#0
    BEQ     memcpy_epilog
memcpy_2:
    SUB     R2,R2,#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
```

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

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.