Korsutveckling för ARM/Thumb – del 2

Ur innehållet

- Programflöde
- Subrutiner, parametrar och returvärden
- Inkapsling av data och funktioner
- Variabler och minnesanvändning

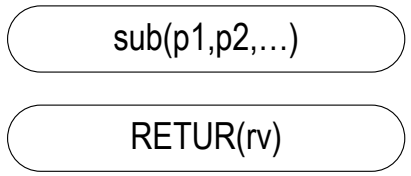
Läsanvisningar:

- Arbetsbok kap 2
- Quick-guide, instruktionslistan

Målsättningar:

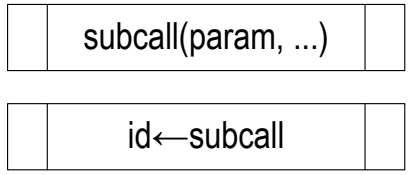
- Koda och testa enkla C-funktioner.
- Utforma och testa assemblerkod med MD407-simulator (ETERM8)

Flödesdiagram för programstrukturer



Inträde i och utträde ur subrutiner.

Inträde, typiskt med subrutinens namn och symbolisk representation av eventuella parametrar då sådana finns.
Utträde, (”RETUR”) typiskt med angivande av returnvärde (rv) om sådant finns.

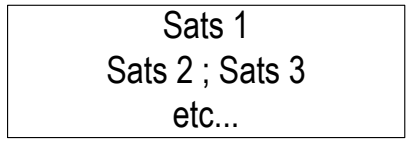


Subrutinanrop.

Med parametrar, symbolisk representation av eventuella aktuella parametrar som skickas med subrutinanropet.
Med returvärde, tilldelningsoperator placeras framför den anropade subrutinen.

Engångsinitieringar.

Placeras typiskt i omedelbar anslutning till en inträdessymbol



Exekveringsblock.

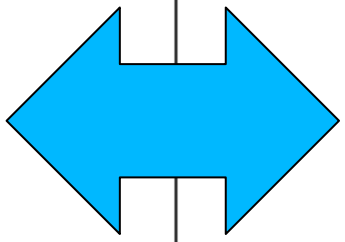
En eller flera satser som ordnats och exekveras sekvensiellt.

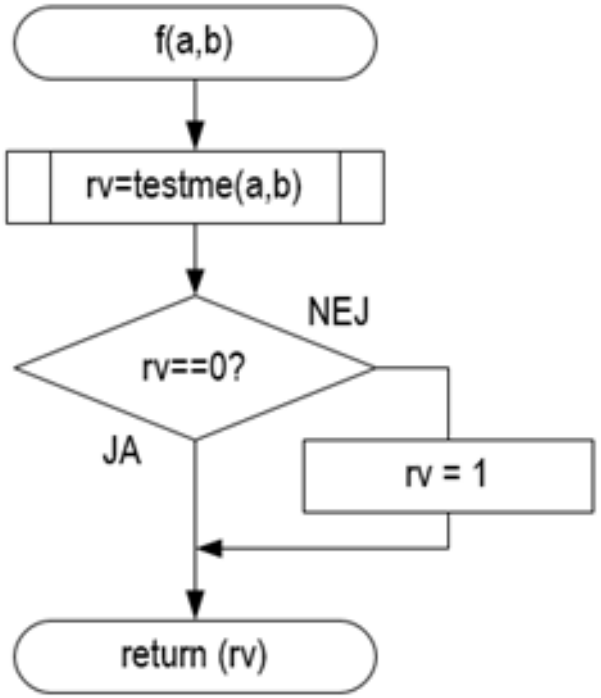
Villkorsblock.

Ett uttryck testas, utfallet kan vara sant eller falskt och exekveringsväg väljs därefter.

Exempel:

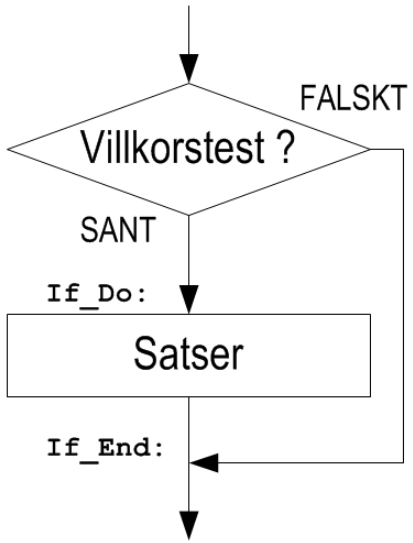
```
int f ( int a, int b )
{
    if ( testme( a,b ) )
        return 1;
    return 0;
}
```



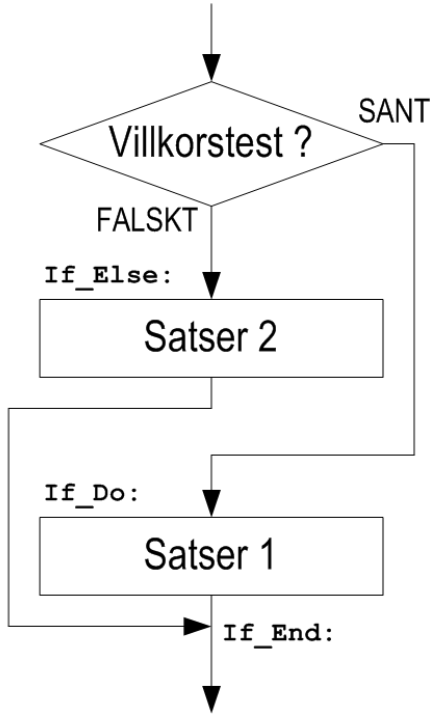


Kontrollstrukturer

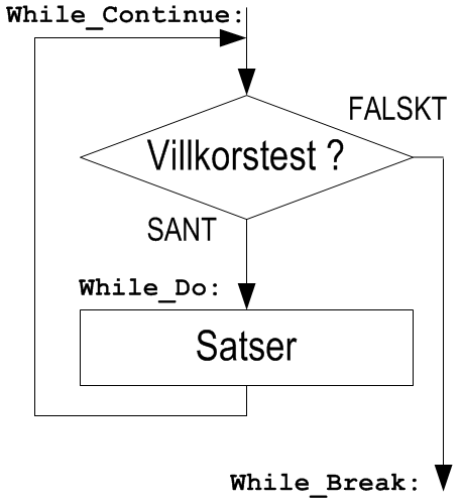
```
if( villkor )
{
  Satser
}
```



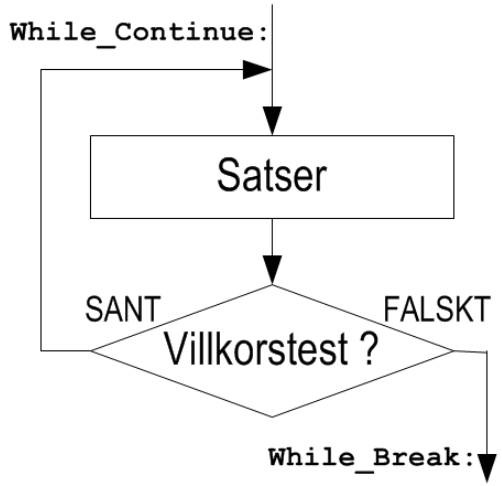

```
if( villkor ){
   Satser1
}else{
   Satser2
}
```




```
while(  villkor  )
{
  Satser
}
```

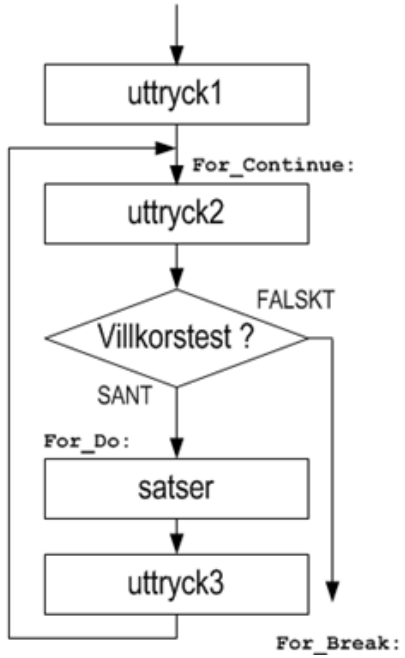



```
do{
  Satser
} while( villkor );
```

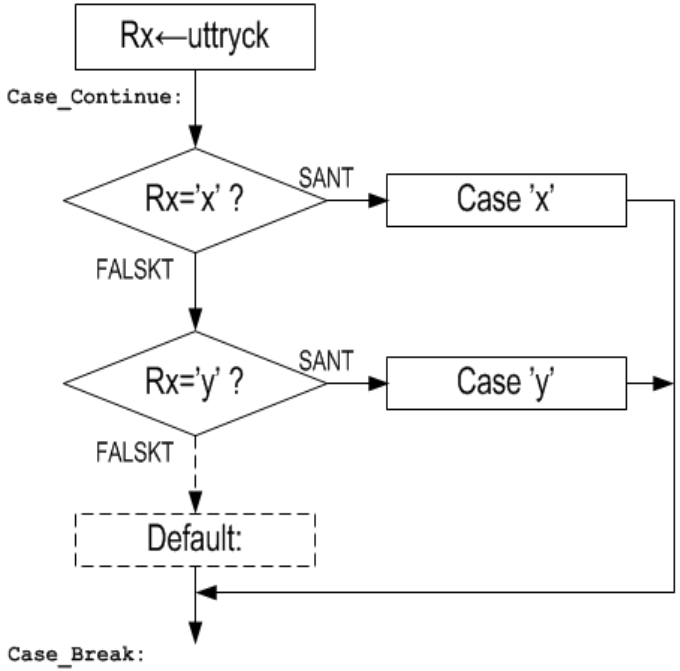


Kontrollstrukturer (forts)

```
for( uttryck1;
     uttryck2;
     uttryck3 )
         { satser }
```




```
switch( uttryck )
{
    case 'x':
        break;
    case 'y':
        break;
    default: ...
}
```



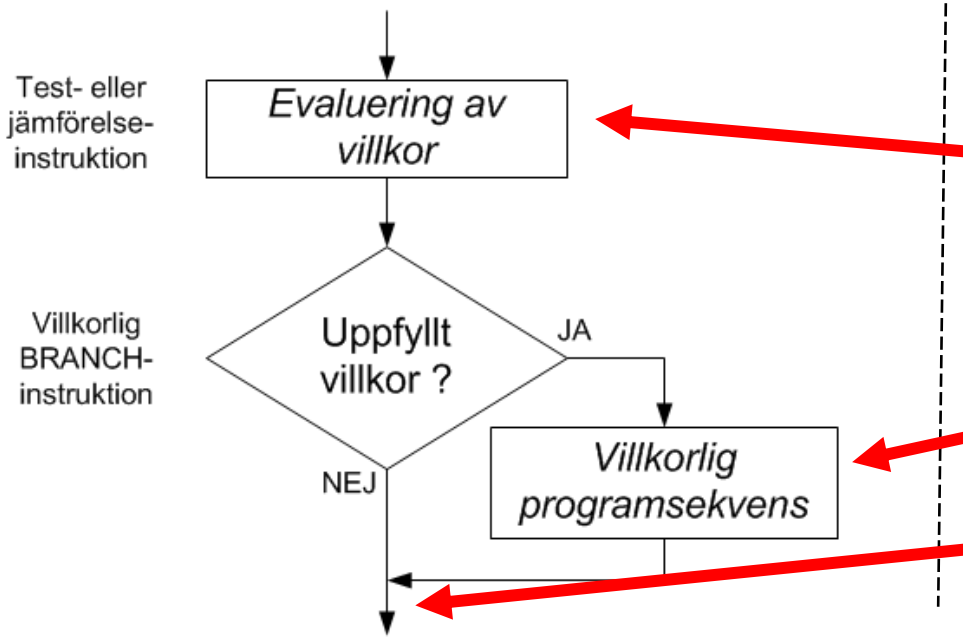
Villkorsblock – kan ändra programflöde

FLISP Assembler:

```
     LDA  i
     CMPA j
     BEQ  L1
     BRA  L2
L1:
     ...
L2:
```

Exempel:

```
if( i==j )
   L1;
L2;
```



THUMB Assembler:

```
    LDR   R0,i
    LDR   R1,j
    CMP   R0,R1
    BNE   L2
    ...
L2:
```

Kan vi göra det bättre än så här?

”Jämförelse” eller ”test”?

Att ”jämföra med 0” kallas också för ”test”, det finns en speciell instruktion för det.

```
if( i != 0 ) ↔ if( i )

if( i == 0 ) ↔ if( !i )
```

THUMB Assembler:

```
LDR R0,i
CMP R0,#0
alternativt:
LDR R0,i
TST R0,R0   @ R0 & R0
```

FLISP Assembler:

```
LDA  i
CMPA #0
alternativt:
LDA  i
TSTA
```

Test-instruktionen påverkar endast flaggor N och Z.

Registeranvändning

I princip kan man använda de generella registren som man vill men det är mycket bättre att följa ARM's rekommendationer:

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

Funktionsparametrar

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.

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	

Exempel:

Följande deklarationer är gjorda:

```
void f(int x,int y, int z, int w);
int    a,b,c,d;
```

Visa hur följande funktionsanrop kodas i assemblerspråk:

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

Assembler:

```
LDR   R0,a
LDR   R1,b
LDR   R2,c
LDR   R3,d
BL    f
```

Funktionens returvärde

Konventionerna sägar att vi använder register R0 för 8,16 och 32-bitars returvärde...

...och registerparet R0:R1 för 64 bitars returvärde

Exempel:

Funktionen:

```
int rintval( void )
{
   return 0x12345678;
}
```

kodas:

```
rintval:
   LDR    R0,=0x12345678
   BX     LR
```

Exempel:

Funktionen:

```
long long rval( void )
{
   return 0x1234567800000000;
}
```

kodas:

```
rval:
   MOV      R0,#0
   LDR      R1,=0x12345678
   BX    LR
```

Registerspill

Problem: Det register man behöver är upptaget.
Lösning: Man lagrar temporärt registrets innehåll på annan plats (annat register eller i minnet), detta kallas ”registerspill”.

Exempel:

Funktionen `int testme( int a, int b )` är definierad.
Visa hur följande funktion kan kodas i assemblerspråk:

```
int f ( int x, int y )
{
   if ( testme( x,y ) )
      return 1;
   return 0;
}
```

Vilket/vilka register måste sparas inför anropet av `testme`?

Vi löser på tavlan...

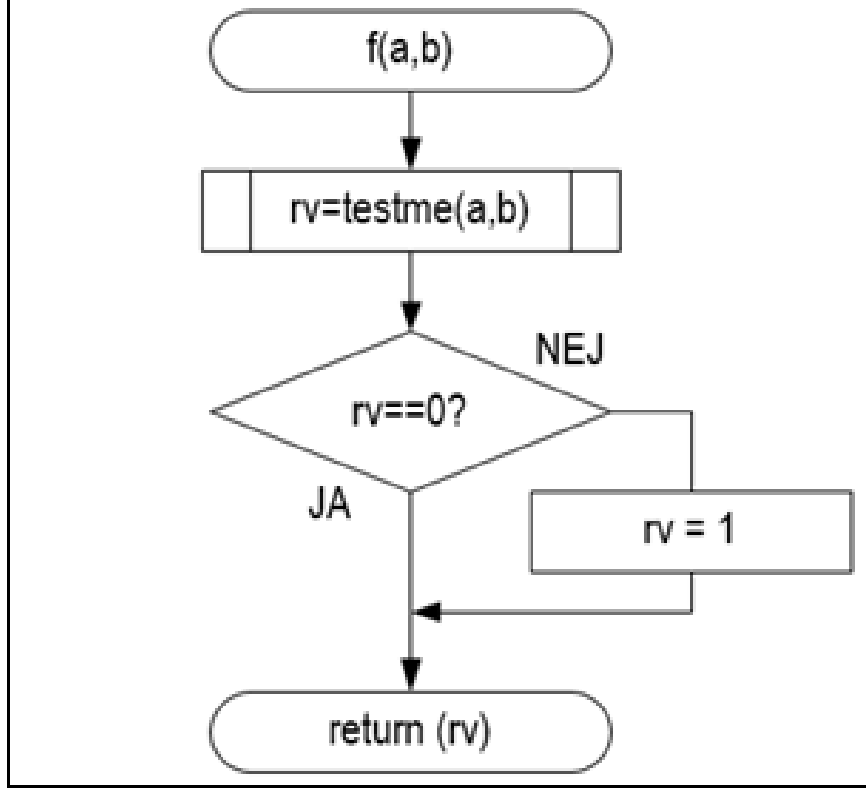
Registerspill

Problem: Det register man behöver är upptaget.
Lösning: Man lagrar temporärt registrets innehåll på annan plats (annat register eller i minnet), detta kallas "registerspill".

Exempel:
Funktionen `int testme( int a, int b )` är definierad.
Visa hur följande funktion kan kodas i assemblerspråk:

```
int f ( int x, int y )
{
   if ( testme( x,y ) )
      return 1;
   return 0;
}
```

Vilket/vilka register måste sparas inför anropet av `testme`?




```
f:
      PUSH {LR}
      BL   testme
      CMP  R0,#0
      BEQ  .L1
      MOV  R0,#1
.L1:
      POP  {PC}
```

Instruktioner för villkorlig programflödeskontroll

C-operator	Betydelse	Datatyp	Instruktion	Komplement-instruktion
==	Lika med	signed/unsigned	BEQ	BNE
!=	Skild från	signed/unsigned	BNE	BEQ
<	Mindre än	signed	BLT	BGE
		unsigned	BCC	BCS
<=	Mindre än eller lika	signed	BLE	BGT
		unsigned	BLS	BHI
>	Större än	signed	BGT	BLE
		unsigned	BHI	BLS
>=	Större än eller lika	signed	BGE	BLT
		unsigned	BCS	BCC

Relationer och tal med tecken

Vid relationer (> eller <) måste vi ta hänsyn till om operanderna är `signed` eller `unsigned`:

<	Mindre än	signed	BLT
		unsigned	BCC

Antag att följande deklarationer gjorts på ”toppnivå”:

```
unsigned int   i,j;
int            k,l;
```

Koda följande programsekvens i assembler:

```
if( i < j )
{
    if ( k < l )
        L1;
}
L2;
```

BCS och BGE är komplementvillkor

BCC och BGE är komplementvillkor

FLISP Assembler:

```
    LDA   i
    CMPA  j
    BCC   .L2  ("BHS")
    LDA   k
    CMPA  l
    BGE   .L2
.L1:
...
.L2:
```

THUMB Assembler:

```
    LDR     R3,i        @ R3<-i
    LDR     R2,j        @ R2<-j
    CMP     R3,R2       @ APSR <-(i-j)
    BCS     .L2         (“BHS”)
    LDR     R3,k        @ R3<-k
    LDR     R2,l        @ R2<-l
    CMP     R3,R2       @ APSR <-(k-l)
    BGE     L2
.L1:
…
.L2:
```

Funktioner från standard C-biblioteket

Funktionen `isupper()`, kan kodas i C på följande sätt:

```
int isupper (int c)
{
   if ( c >= 'A' && c <= 'Z')
      return 1;
   return 0;
}
```

Visa hur funktionen kan kodas i Thumb assembler

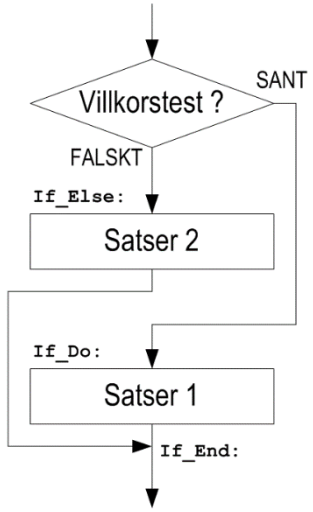
```
isupper:
   CMP   R0,#'A'    @ c >= 'A' ?
   BLT   .L1
   CMP   R0,#'Z'    @ c <= 'Z' ?
   BGT   .L1
.L2:
   MOV   R0,#1      @ return 1;
   BX    LR
.L1:
   MOV   R0,#0      @ return 0;
   BX    LR
```

Vi använder *komplementvillkor* för testen.

<	Mindre än	signed	BLT	BGE
		unsigned	BCC	BCS

>	Större än	signed	BGT	BLE
		unsigned	BHI	BLS

If (...) {...} else { ...}




```
int   i,j,k;
if ( i > k )
   k = i;
else
   k = j;
```

THUMB Assembler:

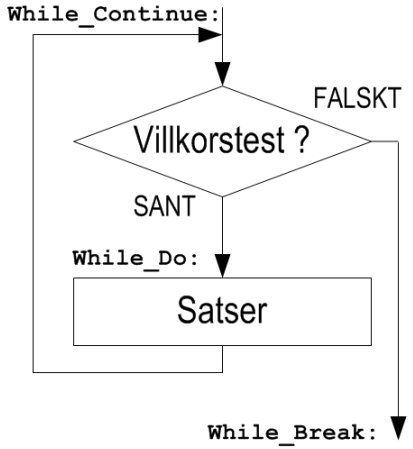
```
   LDR  R1,i
   LDR  R2,j
   CMP  R1,R2
   BGT  If_Do
If_Else:
   MOV  R0,R2
   B    If_End
If_Do:
   MOV  R0,R1
If_End:
   LDR  R2,=k
   STR  R0,[R2]
        ...
```

FLISP Assembler:

```
    LDA  i
    CMPA j
    BGT  If_Do
If_Else:
    LDA  j
If_Do:
    STA  k
```

>	Större än	signed	BGT
		unsigned	BHI

while (...) {...}




```
int i;
...
while ( i < 20 )
{
   ...
}
```

Vid kodning av ”while”-iteration används det *komplementära* villkoret

THUMB Assembler:

```
While_Continue:
   LDR R0,i
   CMP R0,#20
   BGE While_Break
While_Do:
   ...
   B   While_Continue
While_Break:
```

FLISP Assembler:

```
While_Continue:
   LDA  i
   CMPA #20
   BGE  While_Break
While_Do:
     ...
While_Break:
```

>=	Större än eller lika	signed	BGE
		unsigned	BCC

Funktionsparametrar, 8 eller 16 bitar

All aritmetik utförs med 32 bitar, alla parametrar måste vara 32 bitar.
`short` och `signed char` ska därför typkonverteras före anrop.

Exempel:
Funktionen

```
signed char f( signed char a, signed char b);
```

och de globala variablerna

```
signed char x,y,z;
```

är definierade.
Visa hur funktionsanropet:

```
z = f( x, y );
```

kodas i assemblerspråk.

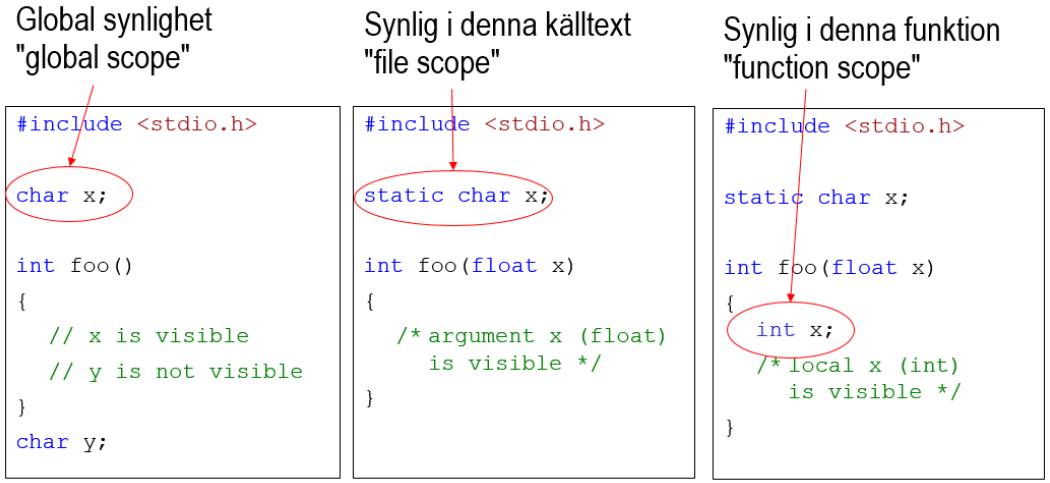
Thumb Assembler:

```
    LDR   R0,=x
    LDRB  R0,[R0]
    SXTB  R0,R0
    LDR   R1,=y
    LDRB  R1,[R1]
    SXTB  R1,R1
    BL    f
    LDR   R1,=x
    STRB  R0,[R1]
```

Variabler och minnesanvändning

Ett deklarerat objekt karakteriseras av

- Synlighet och åtkomlighet
- Varaktighet: permanent plats i minnet eller tillfällig (register/stack)
- Typ av bindning: ingen, intern eller extern



Senare deklaration, parameter eller lokal, "döljer" tidigare...

Synlighet – ("visibility")

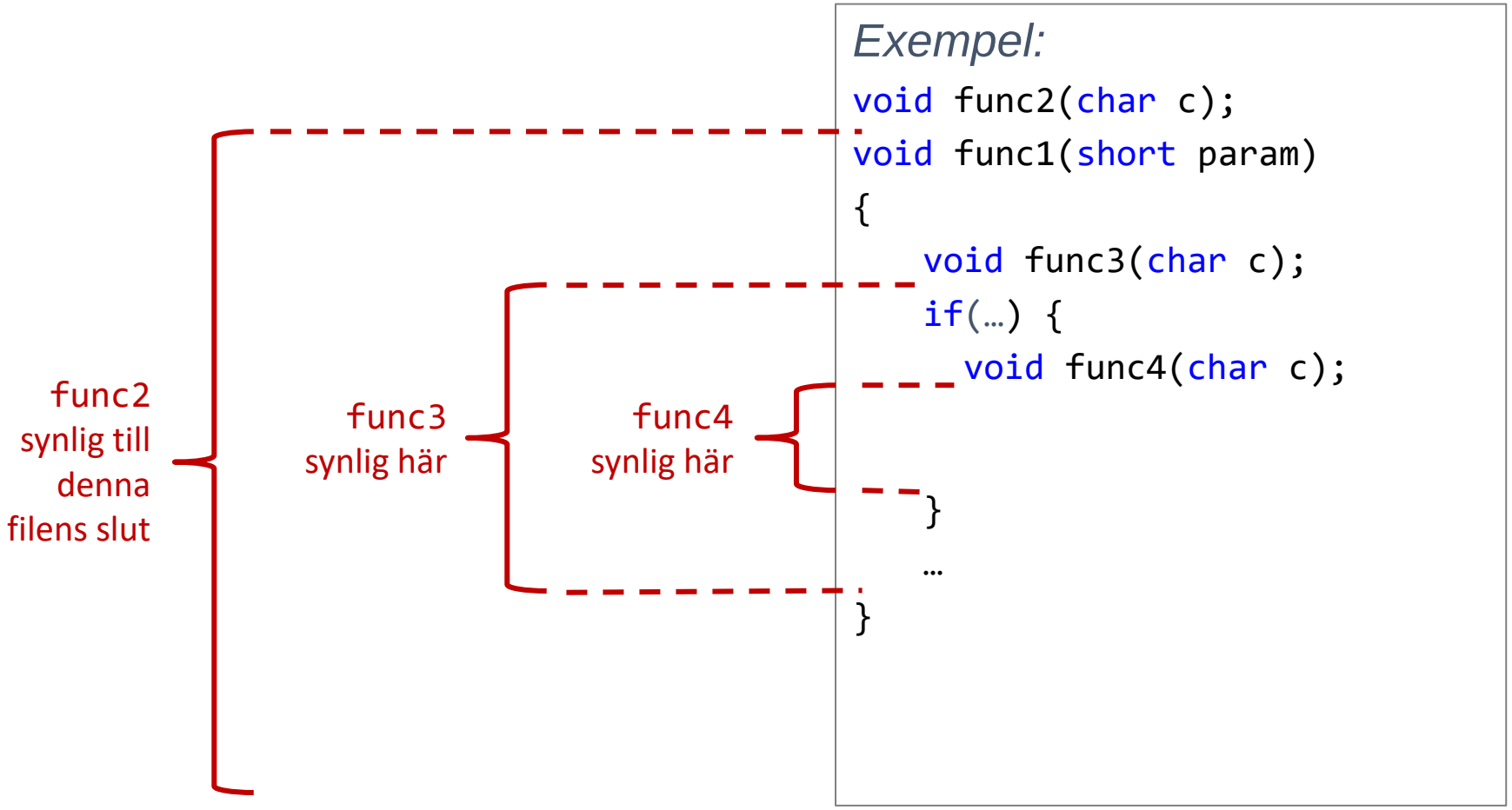
Deklarationer (variabler och funktioner) och uttryck som *typdefs/defines* är synliga först *efter* dess introduktion.

Exempel:

```
void func2(char c);     // Introducera deklaration av func2.
void func2(char c);     // Vi kan upprepa den (flera gånger) …
void func1(short param)
{
   void func2(char c); // Vi kan också ha den här...
   if(…) {
     void func2(char c);   // ... och här ...
     return func2('a');
   }
   …
   return;
}
```

Synlighet

Synligheten kan begränsas av deklarationens block.



Varaktighet ("storage duration")

Varje deklarerat objekt har en *varaktighet* under vilken det kan användas.
Det finns fyra olika typer av varaktighet i C:

- automatic: reserverat minnesutrymme finns i det block objektet deklarerats. ("tillfällig lagring")
- static: reserverat minnesutrymme under hela programexekveringen, initieras innan exekvering av main-funktionen.("permanent lagring")
- allocated: minnesutrymme som reserverats med funktioner för dynamisk minneshantering (`malloc/free`).
- thread: reserverat minnesutrymme under hela exekveringen av den tråd där objektet deklarerats. Jämför med "static" men inte hela programmet.

Lagringsklasser ("storage class")

Med lagringsklassen specificeras objektets varaktighet och bindning
En lagringsklass kan anges med ett *tillägg* i deklarationen:

`auto` – tillfällig lagring, ingen bindning
`register` – tillfällig lagring, ingen bindning; adressoperator kan inte användas
`static` – permanent lagring intern bindning (dock ej om deklaration i block)
`extern` – permanet lagring och extern bindning (om inte redan deklarerad)

Exempel:

```
static int var1 = 1;
extern int var2;
void func()
{
   auto int var3 = 0;  /* auto är default här */
   register auto int var4 = 0; /* lämplig för registerallokering*/
}
```

Permanent lagring

Variabler med permanent adress i minnet:

```
int i; /* global synlighet */

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

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

I ARM/Thumb assemblerspråk:

```
            .GLOBL    i
i:          .SPACE    4
.i:         .SPACE    4
.func_i:    .SPACE    4
```

alternativt sätt för global variabel:

```
            .comm gi,4,4
(.comm sym,length,alignment)
```

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

Tillfällig lagring – i register eller på stacken

Exempel:
Gör en lämplig registerallokering och koda tilldelningssatserna i funktionen:

```
void   f (int a, int b, int c )
{
    int d:
    int e;
    d = a;
    e = b;
    …
}
```

Lösning:
Vi ser att R0,R1 och R2 redan är upptagna av parametrar, R3 upplåter vi för uttrycksevaluering i funktionen och gör registerallokeringen:
`d=R4, e=R5.`
Vi får då följande kodning:

```
@void   f (int a, int b, int c )
f:
@{
@    int d:
@    int e;
     PUSH    {R4,R5,LR}
     MOV     R4,R0
     MOV     R5,R1
     …
     POP     {R4,R5,PC}
@}
```

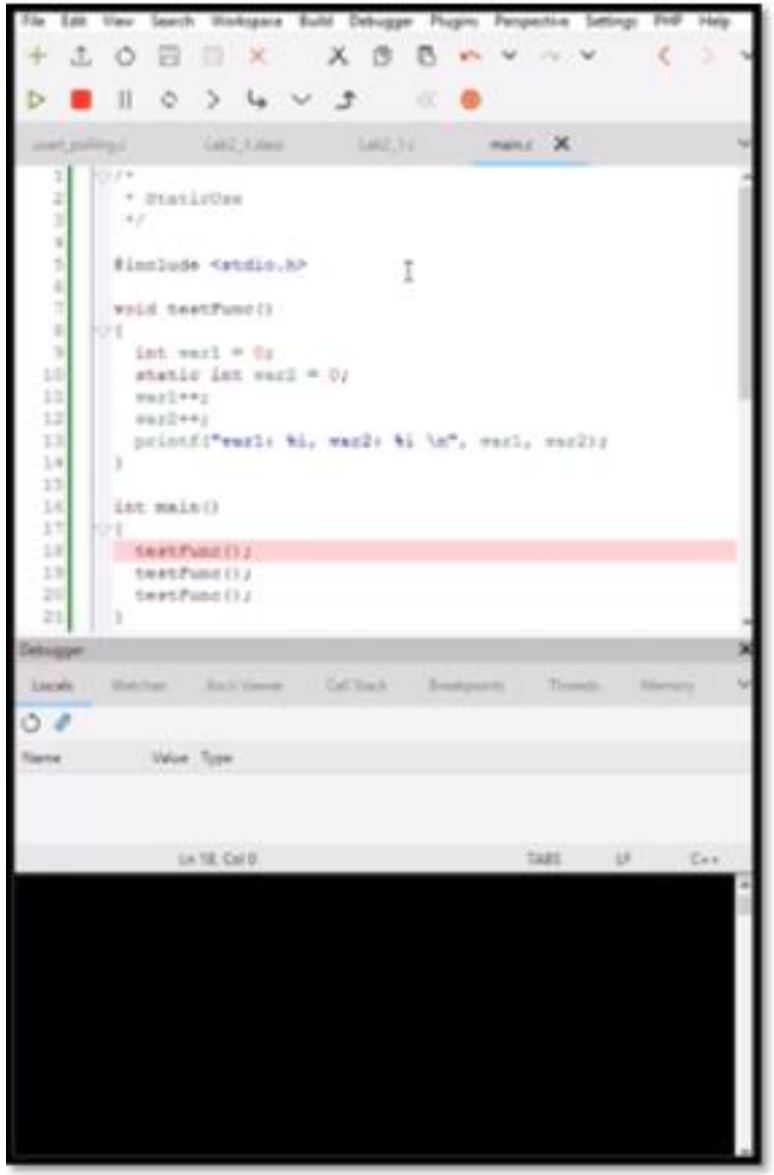
R7	Speciellt använder GCC R7 som pekare till aktiveringspost (*stack frame*)
R6	Också dessa register är avsedda för variabler och temporärbruk
R5	Om dom används måste dom sparas och återställas av
R4	den anropade (*callee*) funktionen
R3	parameter 4 / temporärregister — Dessa register sparas p

Permanent/tillfällig lagring

```c
#include <stdio.h>

void testFunc()
{
  int var1 = 0;
  static int var2 = 0;
  var1++;
  var2++;
  printf("var1: %i, var2: %i \n", var1, var2);
}

int main()
{
  testFunc();
  testFunc();
  testFunc();
}
```



Bindning – ("linkage")

Med "bindning" menas hur en variabel (eller funktion) kan refereras:

Ingen bindning, variabeln kan refereras enbart inom dess åtkomlighet.

Synlig i denna funktion
"function scope"

```
#include <stdio.h>

static char x;

int foo(float x)
{
  int x;
  /* local x (int)
     is visible */
}
```

Extern bindning, variabeln kan refereras från externa delar (samtliga källfiler) i programmet.

Global synlighet
"global scope"

```
#include <stdio.h>

char x;

int foo()
{
  // x is visible
  // y is not visible
}
char y;
```

Intern bindning, variabeln kan refereras från den källfil (translation unit) den deklarerats.

Synlig i denna källtext
"file scope"

```
#include <stdio.h>

static char x;

int foo(float x)
{
  /* argument x (float)
     is visible */
}
```

”Höga” register R8-R12, för temporär lagring

Det är möjligt att använda även register R8-R12 för att ”spilla” register.

R12 (IP)	
R11	Dessa register är avsedda för variabler och som temporära register.
R10	Om dom används måste dom sparas och återställas av
R9	den anropade (*callee*) funktionen
R8	
R7	Speciellt använder GCC R7 som pekare till aktiveringspost (*stack fram*

Vi har dock här begränsat oss till användning av ARM-V6 (huvudsakligen 16-bitars instruktionsformat).
Dessa instruktioner använder bara tre bitar i instruktionsformatets registerfält. Man måste då använda en speciell MOV-variant, för att kopiera data från R0-R7 till R8-R12, och vice versa.

LDR – Load register

En basadress bestäms genom att innehållet i ett indexregister adderas till innehållet i basregistret. Därefter kopieras ett WORD, från denna adress, till destinationsregistret

Syntax:

```
LDR     Rt,[Rn,Rm]
```

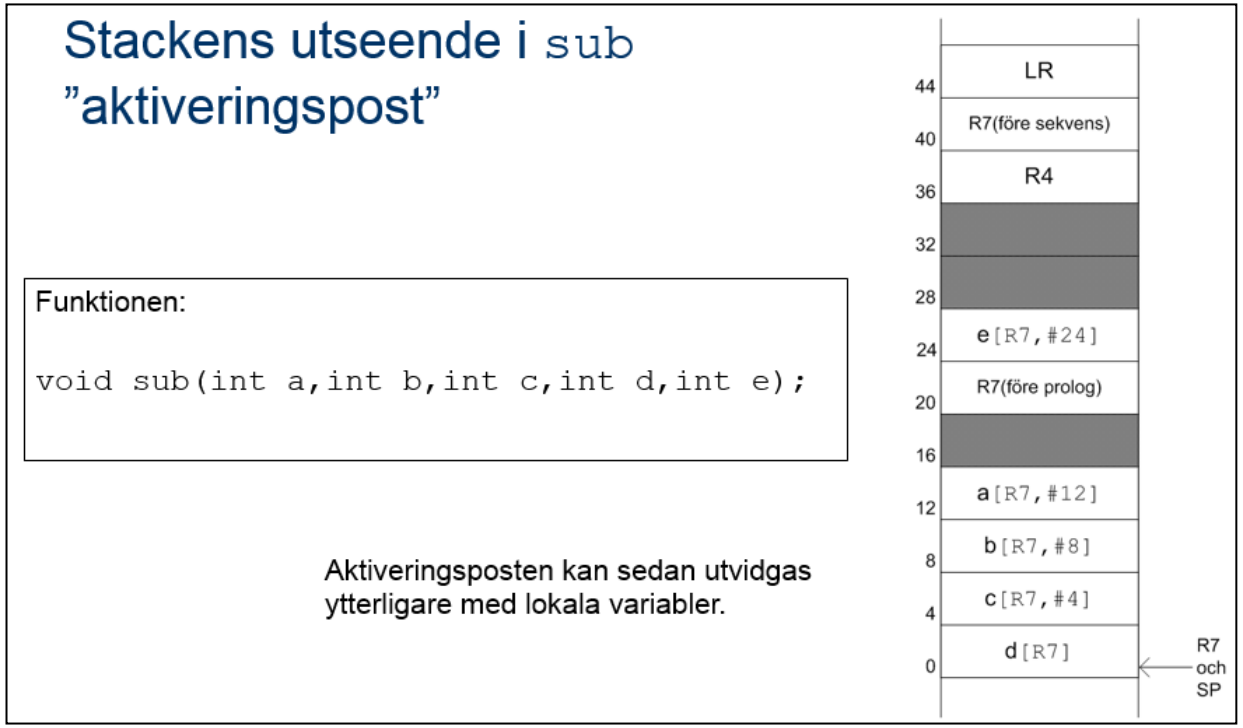
Rt	Destination, (R0-R7).
Rn	Basregister för adressberäkningen, (R0-R7).
Rm	Indexregister för adressberäkningen, (R0-R7) .

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								

När inte registren räcker till

Då registren inte räcker till måste parametrar och lokala variabler lagras i minnet, stacken är då det lämpligaste stället.

Detta är också sant då C-kompilatorn genererar kod som ska användas med en debugger.



Registeranvändning med lokala variabler

Exempel:

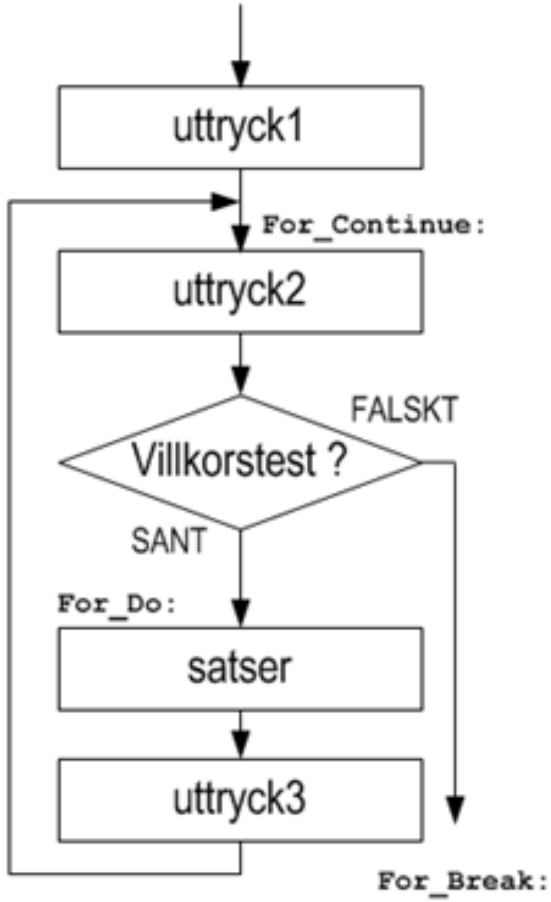
Funktionen `int g( int )` är definierad.

Visa hur följande funktion kan kodas i assemblerspråk:

```
int f( int val )
{
    int i;
    int bits = 0;

    for( i=0 ; i< val; i++ )
    {
        bits = bits | g(i);
    }
    return bits;
}
```

Vi löser på tavlan...



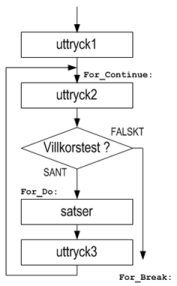
Registeranvändning med lokala variabler

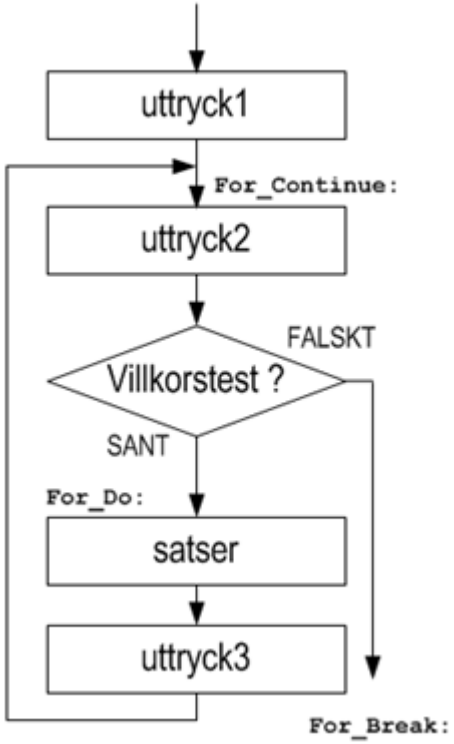
Exempel:
Funktionen `int g( int )` är definierad.
Visa hur följande funktion kan kodas i assemblerspråk:

```
int f( int val )
{
    int i;
    int bits = 0;

    for( i=0 ; i< val; i++ )
    {
        bits = bits | g(i);
    }
    return bits;
}
```

Vi löser på tavlan...






```
@ Registerallokering:
@ R0, parameter och returvärde
@ R4 "i"
@ R5 "bits"
@ R6 "val" spill från R0

f:
        PUSH    {R4,R5,R6,LR}
@ 'prolog'
        MOV     R6,R0           @ spillregister R6
        MOV     R5,#0           @ bits = 0;
@ 'uttryck 1'
        MOV     R4,#0           @ i = 0;
@ 'For_continue' - 'uttryck 2'
.L1:
        CMP     R4,R6           @ i - val
        BGE     .L2             @ i < val, komplementvillkor
@ 'satser'
        MOV     R0,R4
        BL      g               @ g(i);
        ORR     R5,R5,R0        @ bits = bits | g(i);
@ 'uttryck 3'
        ADD     R4,R4,#1        @ i++;
        B       .L1
@ 'For_Break'
.L2:
        MOV     R0,R5           @ "bits" -> R0 (returvärde)
        POP     {R4,R5,R6,PC}
```