Assembly programming for ARM – part 2

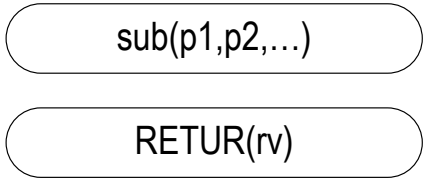
Contents

- Programflöde
- Subrutiner, parametrar och returvärden
- Tillfälliga (lokala) variabler

Läsanvisningar:

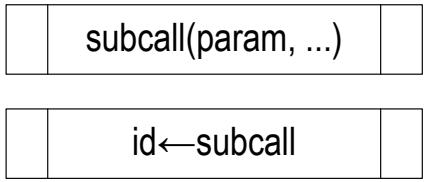
- Arbetsbok kap 2
- Quick-guide, instruktionslistan

Flowchart for program structures



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.

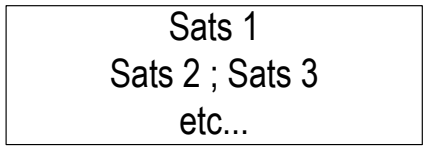


Subrutinanrop.

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

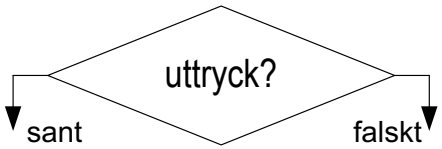
Engångsinitieringar.

Placeras typiskt i omedelbar anslutning till en inträdessymbol



Exekveringsblock.

En eller flera satser som ordnats och exekveras sekvensiellt.

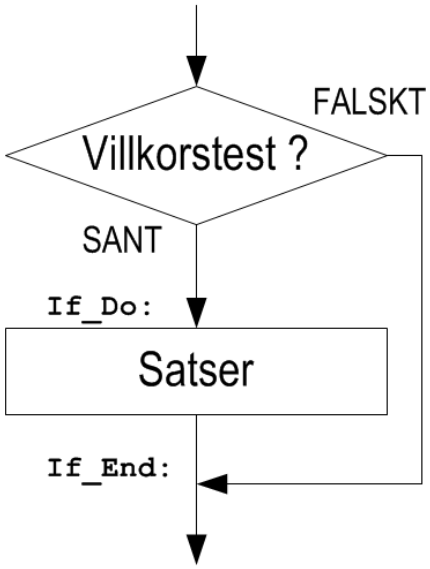


Villkorsblock.

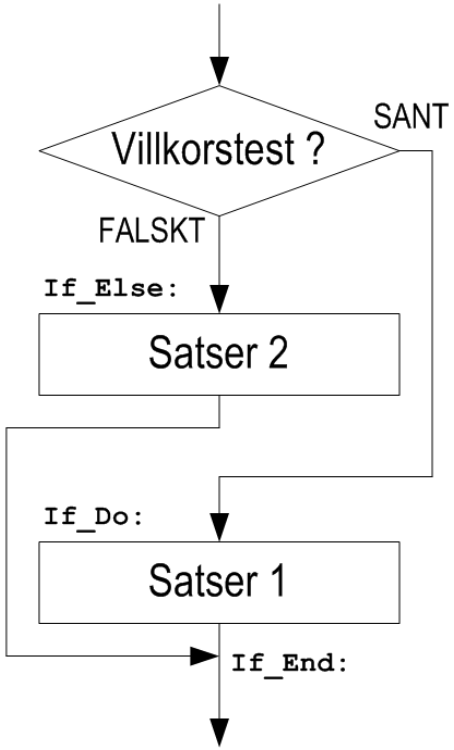
Ett uttryck testas, utfallet kan vara sant eller falskt och exekveringsväg väljs därefter.

Control structures

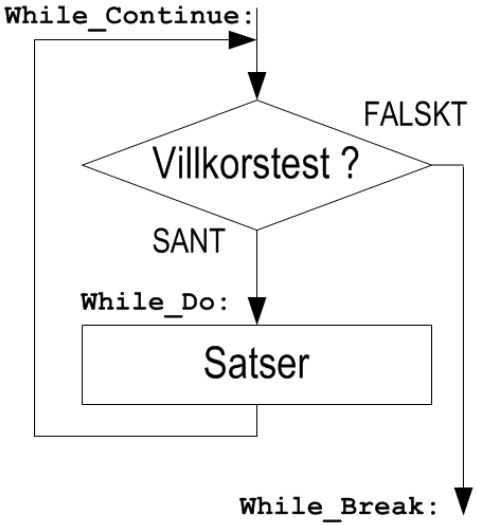
```
if( villkor )
{
   Satser
}
```



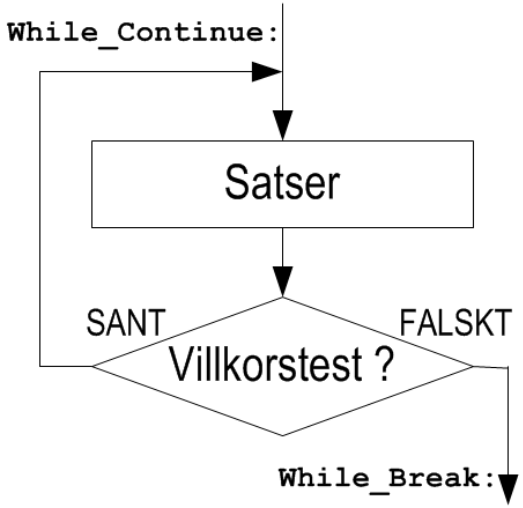

```
if( villkor ){
    Satser1
}else{
    Satser2
}
```




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

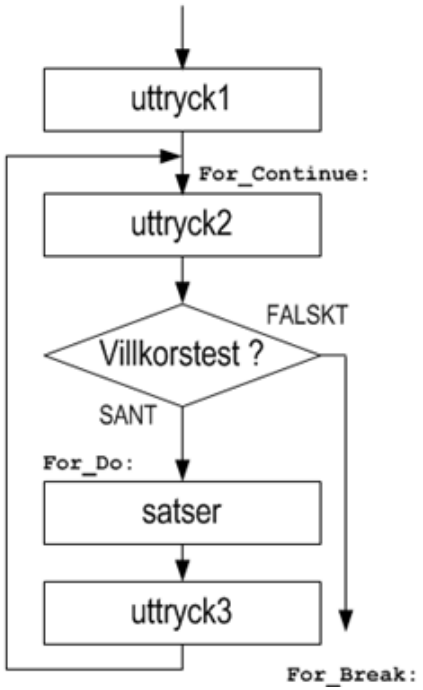



```
do{
   Satser
} while( villkor );
```

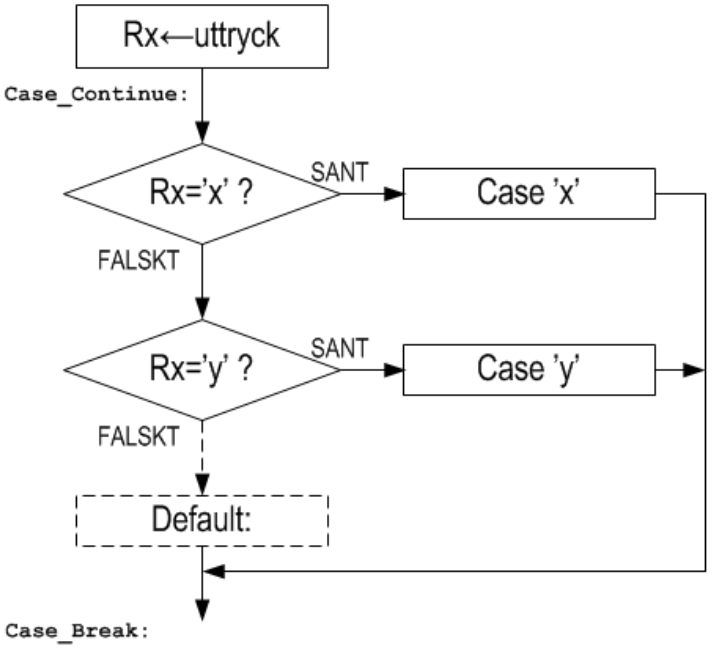


Control structures (cont.)

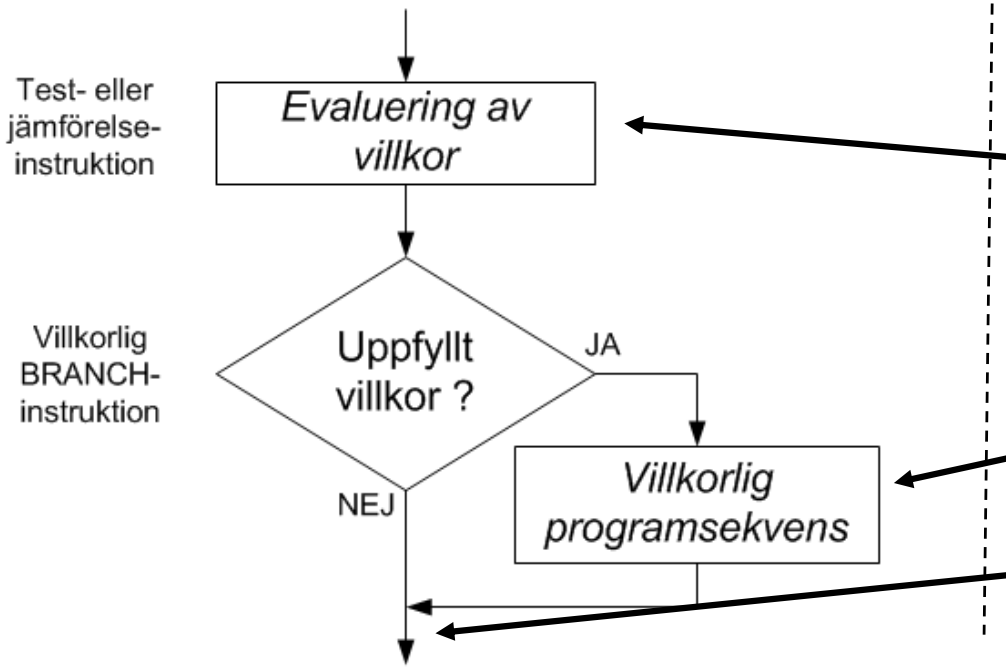
```
for( uttryck1;
     uttryck2;
     uttryck3 )
        { satser }
```




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



Conditional block – can change program flow



Exempel:

```
if( a==b )
   L1;
L2;
```

Assemblerspråk:

```
    LDR   R0,a
    LDR   R1,b
    CMP   R0,R1
    BEQ   L1
    B     L2
L1:
    ...
L2:
```

Can we do better than 2 branch instructions?

B L2end

Instructions that program flow: conditions

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

Relations

<	Mindre än	signed	BLT
		unsigned	BCC

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

Antag att följande deklarationer gjorts på ”toppnivå”:

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

Koda följande programsekvens i assembler:

```
if( a < b )
   {
      if ( c < d )
         L1;
   }
   L2;
```

```
    LDR R0, a
    LDR R1, b
    CMP R0, R1
    BCC Lx
    B   L2
Lx: LDR R0, c
    LDR R1, d
    CMP R0, R1
    BLT L1
    B   L2
L1: …
L2: …
```

Vi löser på tavlan...

”Compare” or ”test”?

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

```
if( a != 0 ) ↔ if( a )

if( a == 0 ) ↔ if( !a )
```

Assembler:

```
LDR R0,a
CMP R0,#0    @ R0 - 0
```

alternativt:

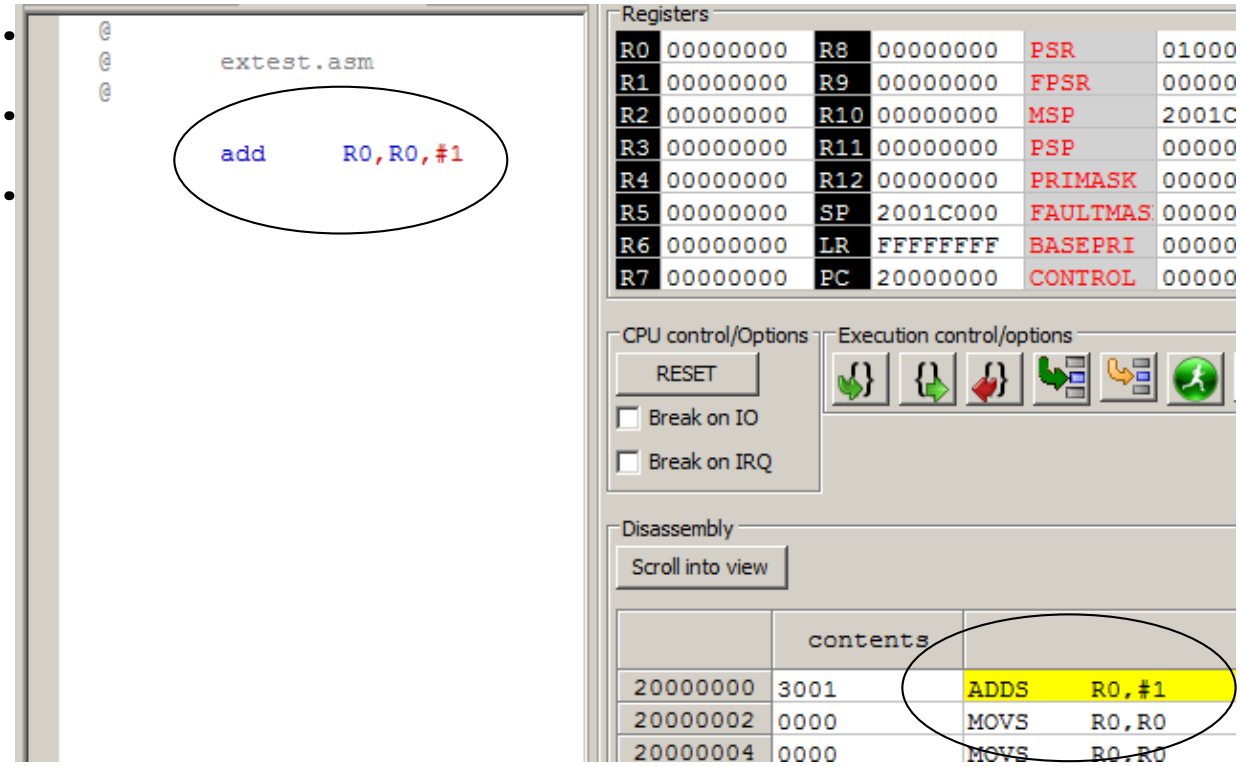
```
LDR R0,a
TST R0,R0    @ R0 & R0
```

Test-instruktionen påverkar endast flaggor N och Z.

Setting flags

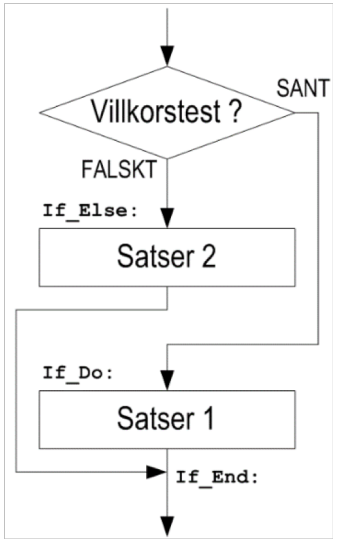
Instruktioner för aritmetik- logik- och skiftoperationer påverkar flaggsättningen:

```
ADD     SUB     MUL ..
AND     ORR     EOR ..
ASR     LSL     ROR ..
```



Vid disassemblering visas dom ofta med ett 'S' tillagt i instruktionsnamnet,
`ADD` *blir* `ADDS` *osv.*
Har liten praktisk betydelse,
se även Quick Guide...

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

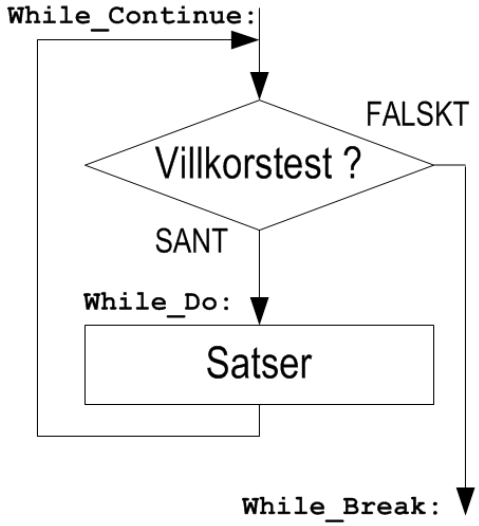



```
int   a,b,c;
if ( a > b )
   c = a;
else
   c = b;
```

```
        LDR   R1,a
        LDR   R2,b
        CMP   R1,R2
        BGT   If_Do
If_Else:
        MOV   R0,R2
        B     If_End
If_Do:
        MOV   R0,R1
If_End:
        ...
```

>	Större än	signed	BGT
		unsigned	BHI

while (...) {...}




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

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

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

>=	Större än eller lika	signed	BGE
		unsigned	BCC

Registers

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)	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	

Function parameters

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 (behandlas senare i kursen).

Exempel:
Följande deklarationer är gjorda:

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

Visa hur följande funktionsanrop kodas i assemblerspråk:

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

Assembler:

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

Function return value

Konventionerna sägar att vi använder register R0 för 8,16 och32-bitars returvärde, registerparet R1:R0 för 64 bitars returvärde

Exempel:

Funktionen:

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

kodas:

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

Exempel:

Funktionen:

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

kodas:

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

Function parameters, 8 or 16 bits

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 minc( signed char a, signed char b)
```
och de globala variablerna
```
signed char x,y,z
```
är definierade. Visa hur funktionsanropet:
```
x = minc( x, y );
```
kodas i assemblerspråk.

Assembler:
```
    LDR   R0,=x
    LDRB  R0,[R0]
    SXTB  R0,R0
    LDR   R1,=y
    LDRB  R1,[R1]
    SXTB  R1,R1
    BL    minc
    LDR   R1,=x
    STRB  R0,[R1]
```

Register spilling

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;
}
```

Which register do we need to store before calling testme?

Vi löser på tavlan...

Temporary (local) variables

Exempel:

Gör en lämplig registerallokering för funktionen:

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

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

Om **f** inte är trivial behövs R0-R3 för uttrycksevaluering etc. Enkel grundregel är då : Använd R4,R5 osv till lokala variabler, konventionen säger att dessa *kan* var upptagna av anropande (caller) funktion och därför *måste* dom ”spillas”.

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}
    …
    POP     {R4,R5,PC}
@}
```

Save R4 and R5 values from out of f!

Restore R4 and R5 before leaving f

Register utilization for local variables

Exempel:

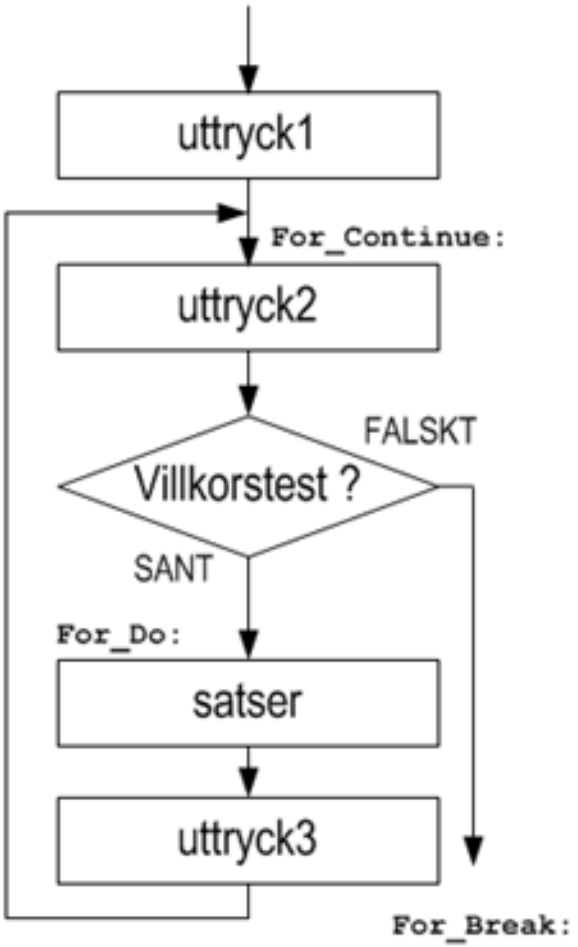
Funktionen `int g( int )` är definierad.

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...



”Higher” registers R8-R12, for temporary storage

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

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 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).

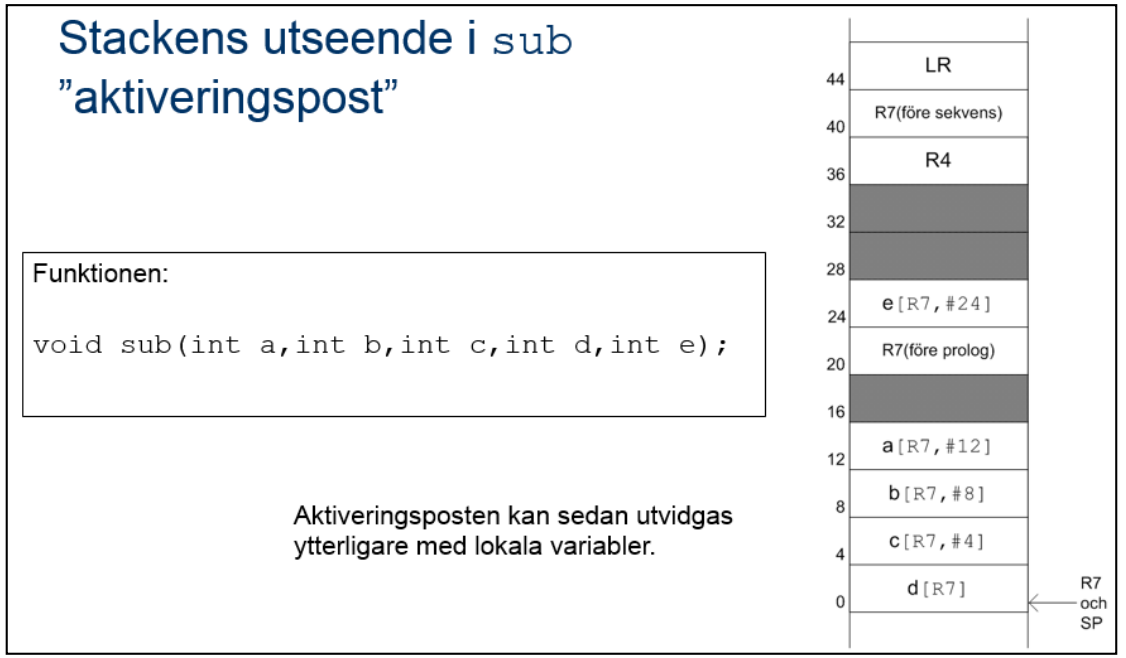
`Rm` Indexregister för adressberäkningen, (R0-R7) .

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								

When there are not enough registers…

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.



Vi återkommer till hur register spills till minnet i slutet av kursen.

Summary

This lecture:

- Program flow (conditional instructions)
- Loops
- Subroutines (parameters, return value)
- Register spilling
- Temporary variable