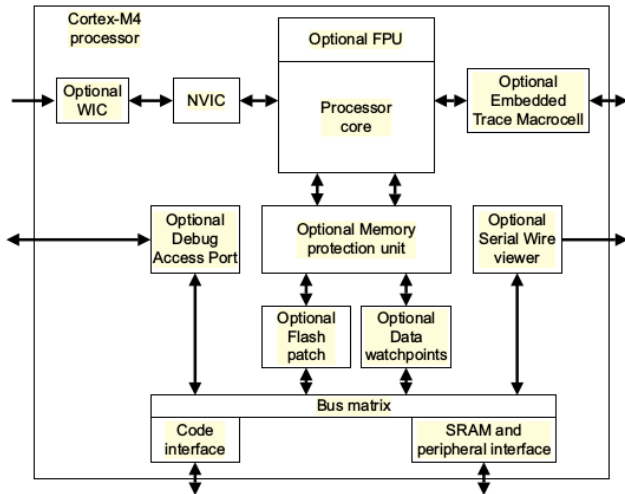


# ARM Cortex-M4 Architecture and Instruction Set

M J Brockway

February 9, 2018

# Block diagram



This diagram is from the [Cortex M4 Generic User Guide](#), published by ARM. *Further reading:* Chapter 1

# Programmer's model

Ref: *Generic User Guide* section 2.1. Click the link above, or find it on the module home page.

## Processor modes

**main** stack - running application software. Limited access to some instructions, system timer. System is in this mode immediately after reset.

**handler** mode - handling as exception. Returns to thread mode after all exceptions handled.

## Stacks

- ▶ *full descending*: Stack pointer is *decremented* when an item is pushed on; the stack grows 'downwards'
- ▶ Two stacks: *main* and *process*; a special register controls which.

### 2.1.3 Core registers

The processor core registers are:

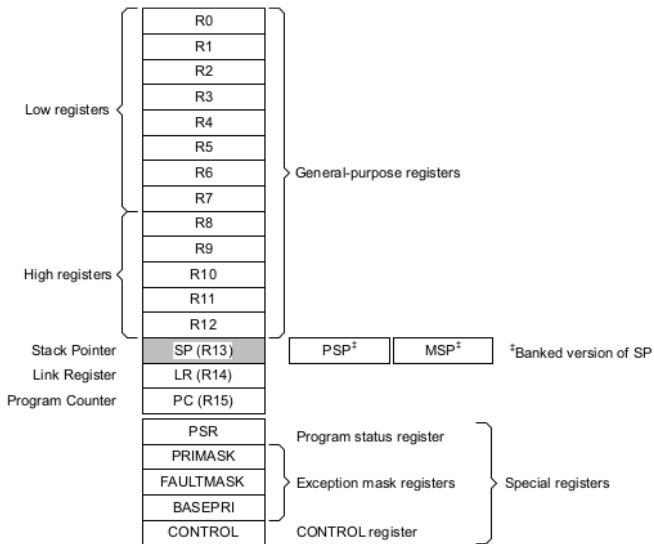


Table 2-2 Core register set summary

# Programmer's model

Registers ...

- R0-R12 32-bit general-purpose registers for data operations. Avoid using higher registers for application programming!
- R13 the stack pointer: *main* or *process* stack depending on setting in CONTROL register.
- R14 is the *link register*(LR): stores the return information for subroutines, function calls, and exceptions.
- R15 is the *program counter*: stores the address of the next instruction to be fetched.
- PSR The *program status register* contains information about status of last operation. Bits 27-31 are the *condition codes*; the ones we shall be most concerned with are
  - N : the last operation produced an arithmetically *negative* result
  - Z : the last operation produced a *zero* result
  - C : the last operation produced a *carry* (or *borrow*)
  - V : the last operation produced an arithmetic *overflow*

# Programmer's model

## Registers (ctd)

- ▶ **CONTROL** manages privilege level of the current instruction, and which of the two stacks to use.

## Exceptions and interrupts

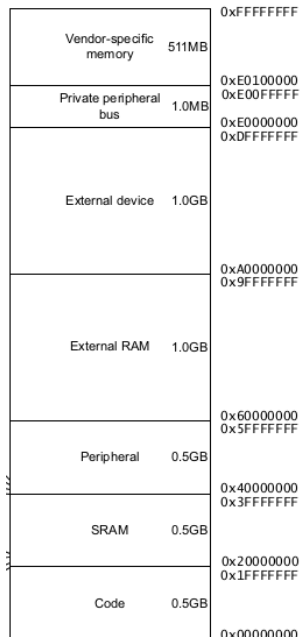
**NVIC** The processor and the *nested vectored interrupt controller* manage prioritized handling.

## Data types

- ▶ 32-bit *words*, 16-bit *halfwords*, 8-bit *bytes*.
- ▶ Can be big- or little-endian;
- ▶ instruction memory and private peripheral but (PPB) accesses are always little-endian.

# Memory model

- ▶ Ref: Generic User Guide sec 2.2;
- ▶ With 32-bit addressing, the memory space is 4 Gb.



# Memory model

- ▶ **code** 00000000-1FFFFFFF is where executable code goes. On-chip FLASH. Data can go here, but not recommended.
- ▶ **SRAM** 20000000-3FFFFFFF is primarily for application data, but code can go here.
- ▶ **peripherals** 40000000-5FFFFFFF - on-chip peripherals
- ▶ **external RAM** 60000000-9FFFFFFF - eg DDR, FLASH, LCD
- ▶ **external device** A0000000-DFFFFFFF - external peripherals
- ▶ **private peripheral bus** E0000000-E00FFFFF
- ▶ **vendor-specific** E0100000-FFFFFFF



# Cortex-M4 Image of a Program

From low to high memory, ...

1. Interrupt vector table - addresses of interrupt and exception handlers
2. C startup routine
3. Application code and data
4. C library code

On reset, the CPU

1. Reads initial 'stack pointer' address
2. Reads interrupt vector for 'reset'
3. The reset handler branches to start of application program
4. The program executes ...

# Introducing Cortex-M4 Machine Instructions

Reading: GUG chapter 3

- ▶ The size of most instructions is 32 bits (4 bytes) or 16 bits (2 bytes), comprising a *THUMB-2 operation code* and sometimes and additional *operand*.
- ▶ The program as loaded into RAM is thus an array of 32-bit words. The address of the  $n^{\text{th}}$  instruction (counting from 0) is therefore the base address of the loaded program +  $32n$ .
- ▶ Shall use *ARM assembly language* as a human-readable version of these instructions.
- ▶ The following is a simple program: simpler than 'Hello world', as it does not do any I/O. It simply uses registers R0, R1 to compute  $10 + 9 + 8 + \dots + 1$ , leaving the result in register R1.

# Cortex-M4 Machine Instructions - simple example

```
PRESERVE8 ; Indicate the code here preserve
           ; 8 byte stack alignment
THUMB     ; Indicate THUMB code is used
AREA      |.text|, CODE, READONLY ; Start of CODE area
EXPORT __main
ENTRY
__main    FUNCTION
          ; initialize registers
MOV  r0, #10 ; Starting loop counter value
MOV  r1, #0  ; starting result
          ; Calculating 10+9+8+...+1 ...
loop
  ADD  r1, r0 ; R1 = R1 + R0
  SUBS r0, #1 ; Decrement R0, update flag ('S' suffix)
  BNE  loop ; If result not zero jump to loop
          ; Result is now in R1
deadloop
  B    deadloop ; Infinite loop
ENDFUNC
END ; End of file
```

# Simple example: pseudocode

main function:

Put value 10 in register 0

Put value 0 in register 1

Repeat:

    Add contents of R0 to R1, leaving result in R1;

    Subtract 1 from contents of R0, updating  
        condition codes according to the result;

    Check condition codes:

        if 'Z' not set, repeat, else drop out of loop

Forever run on the spot (branch to self)

# Simple example with machine code and relative addresses

```
1 00000000                PRESERVE8
2 00000000                ; 8 byte stack alignment
3 00000000                THUMB
4 00000000                AREA
5 00000000                EXPORT                __main
6 00000000                ENTRY
7 00000000                __main FUNCTION
8 00000000                ; initialize registers
9 00000000 F04F 000A        MOV                r0, #10
10 00000004 F04F 0100       MOV                r1, #0
11 00000008                ; Calculating 10+9+8+...+1
12 00000008                loop
13 00000008 4401                ADD                r1, r0
14 0000000A 3801                SUBS               r0, #1
15 0000000C D1FC                BNE                loop
16 0000000E                ; Result is now in R1
17 0000000E                deadlock
18 0000000E E7FE                B                deadlock
19 00000010                ENDFUNC
20 00000010                END
```

# MOV, MVN

## Syntax

- ▶  $\text{MOV}\{S\}\{cond\} \text{ Rd, operand2}$
- ▶  $\text{MOV}\{cond\} \text{ Rd, \#imm16}$
- ▶  $\text{MVN}\{S\}\{cond\} \text{ Rd, operand2}$
- ▶ The MOV instructions copy the value of the second operand to Rd; MVN copies the *complement* of this value to Rd.

where:

- ▶ S is an optional suffix. If included, the condition code flags are updated on the result of the operation.
- ▶ *cond* is an optional condition code (see below).
- ▶ Rd is the destination - always a register.
- ▶ operand2 is a flexible second operand: one of
  - ▶ a constant:  $\#const$  where 32-bit *const* is either a left-shifted byte or a constant of one of the forms 00XY00XY, XY00XY00, XYXYXYXY.
  - ▶ a register with optional shift.
- ▶ *imm16* is any value in the range 0-65535.

## MOV, MVN - Examples

- ▶ `MOV R3, #0x1F0000` - R3 set to 0x1F0000
- ▶ `MVN R3, #0x1F0000` - R3 set to 0xFFE0FFFF
- ▶ `MOV R3, R0` - value in R0 copied to R3
- ▶ `MOVS R3, R0` - value in R0 copied to R3 and condition flags set according to result: Eg Z is set if result is 0; N is set if result is negative
- ▶ `MVNS R3, R0` - complement of value in R0 copied to R3 and condition flags set according to result
- ▶ `MVNSEQ R3, R0` - if Z is set complement of value in R0 copied to R3 and condition flags set according to result

Some condition codes

`EQ` *equal*:  $Z==1$

`NE` *not equal*:  $Z==0$

`HS` *higher or same, unsigned*:  $C==1$

`HI` *higher, unsigned*:  $C==1$  and  $Z==0$

`GE` *greater than or equal, signed*:  $N==V$

`GT` *greater than, signed*:  $N==V$  and  $Z==0$

Exercise: Find definitions of LO, LS, LE, LT.

# Arithmetic

## Syntax

- ▶  $op\{S\}\{cond\} \{Rd,\} Rn, operand2$

where:

- ▶  $op$  is one of
  - ADD** - add:  $Rd \leftarrow Rn + operand2$
  - ADC** - add with carry:  $Rd \leftarrow Rn + operand2 + C$
  - SUB** - subtract:  $Rd \leftarrow Rn - operand2$
  - SBC** - subtract:  $Rd \leftarrow Rn - operand2 - !C$
  - RSB** - reverse subtract:  $Rd \leftarrow operand2 - Rn$
- ▶ Optional  $S$  suffix and  $cond$  condition are as we defined above.
- ▶  $Rd, Rn$  are registers:  $Rd$  is the destination. Its specification is optional: if absent,  $Rd=Rn$ .
- ▶  $operand2$  is a flexible second operand, a byte-sized immediate value or a register with optional shift (see GUG ch 3).
- ▶ **ADD**, **SUB** also allow  $operand2$  to be any value in the range 0-4095 (12 bits).
- ▶  $N,Z,C,V$  flags updated (with  $S$  option) according to result.



# Arithmetic Examples

- ▶ `ADD R0, R1, R1, LSL #2`:  $R0 \leftarrow R1 + (R1 \ll 2)$
- ▶ `ADDEQ R0, R1, #1` - If Z flag is set,  $R0 \leftarrow R1 + 1$
- ▶ `ADDS R0, #1` -  $R0 \leftarrow R0 + 1$
- ▶ `ADC R3, R4, R5` -  $R3 \leftarrow R4 + R5 + C$
- ▶ `SUBS R0, R0, #1` -  $R0 \leftarrow R0 - 1$ ; update condition flags
- ▶ `SBCNES R0, R1, R2` - If Z is clear,  $R0 \leftarrow R0 - R2 - !C$ ; update condition flags
- ▶ `RSBS R0, R1` -  $R0 \leftarrow R1 - R0$ ; update flags

# Logic

## Syntax

- ▶  $op\{S\}\{cond\} \{Rd,\} Rn, operand2$

where:

- ▶  $op$  is one of
  - AND** - bitwise AND:  $Rd \leftarrow Rn \& operand2$
  - ORR** - bitwise OR:  $Rd \leftarrow Rn \mid operand2$
  - EOR** - bitwise exclusive OR:  $Rd \leftarrow Rn \oplus operand2$
  - BIC** - bitwise AND NOT:  $Rd \leftarrow Rn \& \neg(operand2)$ ; clears bits in  $Rd$  marked by  $operand2$
  - ORN** - bitwise OR NOT:  $Rd \leftarrow Rn \mid \neg(operand2)$
- ▶ Optional  $S$  suffix and  $cond$  condition are as we defined above.
- ▶  $Rd, Rn$  are registers:  $Rd$  is the destination. Its specification is optional: if absent,  $Rd$  is  $Rn$ .
- ▶  $operand2$  is a flexible second operand, a byte-sized immediate value or a register with optional shift (see GUG ch 3).
- ▶ With  $S$  option,  $N, Z$  updated according to result (possibly also  $C$  during evaluation of  $operand2$ )

# Logic Examples

- ▶ `ORREQ R0, R1, #0x1F` - If Z is set,  $R0 \leftarrow R1 \oplus 0x1F$
- ▶ `EORS R0, R1` -  $R0 \leftarrow R0 \hat{=} R1$ ; set flags
- ▶ `BIC R3, R4` -  $R3 \leftarrow R3 \& \neg R4$ : bits set in R4 are cleared in R3

# Shifts

## Syntax

- ▶  $op\{S\}\{cond\}$  Rd, Rn, Rs
- ▶  $op\{S\}\{cond\}$  Rd, Rn, #n
- ▶  $RXX\{S\}\{cond\}$  Rd, Rn

where:

- ▶  $op$  is one of
  - $ASR$  - arithmetic shift right
  - $LSL$  - logical shift left
  - $LSR$  - logical shift right
  - $ROR$  - rotate right
- ▶ Optional  $S$  suffix and  $cond$  condition are as we defined above.
- ▶ Rd is the destination register; Rn is register holding the value to be shifted
- ▶ Register Rs holds value of shift length - only the lowest order byte applies
- ▶ #n is a shift length:

$ASR, LSR$   $1 \leq n \leq 32$ ;

$LSL, ROR$   $0 \leq n \leq 31$ ;

- ▶  $RXX$  sets Rd to bits in Rn rotated right 1 bit.
- ▶ With  $S$  option, N,Z updated according to result; if shift length  $> 0$ , C updated to last bit shifted out.

## Comparisons

These always update the condition flags, without needing an S suffix.  
There is no destination register: the only purpose is to update the flags.

Syntax:

- ▶ `CMP{cond} Rn, operand2`
- ▶ `CMN{cond} Rn, operand2`
- ▶ `TST{cond} Rn, operand2`
- ▶ `TEQ{cond} Rn, operand2`

where:

- ▶ Rn register holds the first operand;
- ▶ operand2 is a flexible second operand, as seen above.
- ▶ CMP subtracts the operands and CMN adds the operands, setting the condition flags but discarding the result. Cf SUBS, ADDS.
- ▶ `CMP R0, R1` sets Z and C if  $R0 == R1$ , sets N if  $R1 > R0$ , sets C if  $R0 > R1$
- ▶ TST bitwise ANDs the operands and TEQ bitwise EORs the operands, setting the condition flags but discarding the result. Cf ANDS, EORS.

# Branching

- ▶ To do anything other than run a fixed sequence of instructions, the CPU needs to be able to decide on the result of some test what instruction to execute next.
- ▶ In normal operation, an instruction is *fetch*ed from the address pointed at by the *Program Counter* ( $PC = R15$ ), and the PC immediately incremented by the size of that instruction: 2 or 4 bytes. Then the “next” instruction is the next in the RAM.
- ▶ A branch instruction overrides this by *setting* the PC to some other value - some other valid instruction address, we hope.
- ▶ A *conditional* branch: only happens if some condition is met.
- ▶ The “next” instruction to be fetched is the one whose address was loaded into the PC by the branch instruction.

# Branching

## Syntax

- ▶  $B\{cond\}$  label
- ▶  $BL\{cond\}$  label
- ▶  $BX\{cond\}$  Rm
- ▶  $BLX\{cond\}$  Rm

where:

- ▶ *cond* is an optional condition code, EQ, NE, LT, LE, GE, GT, ... etc
- ▶ Rm is a register containing the address to branch to.
- ▶ A *label* is declared in the assembly language code as an unindented symbol: eg `_main`, `loop`, `deadloop` on slide 11

# Memory Access Instructions with immediate offset

## Syntax

1.  $op\{type\}\{cond\} Rt, [Rn]$  - no offset or write-back
2.  $op\{type\}\{cond\} Rt, [Rn, \#offset]$  - immediate offset, no write-back
3.  $op\{type\}\{cond\} Rt, [Rn, \#offset]!$  - pre-indexed
4.  $op\{type\}\{cond\} Rt, [Rn], \#offset$  - post-indexed

where:

- ▶  $op$  is one of
  - LDR - load register  $Rt$  from [...]
  - STR - store register  $Rt$  to [...]
- ▶  $type$  is one of
  - B - unsigned byte (0-extended to 32 bits)
  - SB - signed byte (on load, sign-extended to 32 bits)
  - H - unsigned halfword (0-extended to 32 bits)
  - SH - signed halfword (on load, sign-extended to 32 bits)
    - ▶ omitted if whole word is to be loaded or stored
- ▶ Optional  $cond$  condition is EQ, NE, ... as we defined above.
- ▶  $Rt$  is register to load to or store from.



# Examples

- ▶ `LDR R8, [R10]` - load R8 from memory address R10
- ▶ `STR R2, [R5, #4]` - store R2 to memory address R5+4; value in R5 is unchanged
- ▶ `LDR R2, [R5, #4]!` - Add 4 onto contents of R5, then load R2 from memory address R5
- ▶ `STR R2, [R5], #-4` - load R2 from memory address R5 then decrement contents of R5 by 4

# Data Definitions

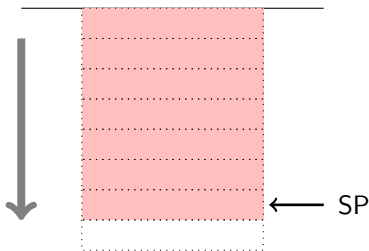
These instructions assume we have a register containing an address pointing at an item of data, or perhaps an array of data. One way this comes about is via the *assembler directives* which define and reserve memory for data.

`{label} DCD expr{, expr} ...`

`{label} DCDU expr{, expr} ...`

- ▶ defines a 32-bit word or a sequence of words with optional label giving address.
- ▶ `expr` is a numeric expression
- ▶ The version *without* U adds padding so that the data is aligned on a *word boundary*, an address that is a multiple of 4. Usually the *unaligned* version is the simpler one to use.
- ▶ Example
  - ▶ `numArray DCDU 1000000, 999999, 999998, 999997`
  - ▶ `LDR R0, =numArray ;loads address of data into R0`
  - ▶ `LDR R1, [R0, #4] ;loads 999999 into R1 Why #4?`

# The Cortex-M4 Stack



The subroutine stack is *full, descending*

- ▶ It “grows” downwards from higher to lower memory addresses
- ▶ The stack pointer SP, register R13, points at the last word on the stack.
- ▶  $SP - 4$  is address of next available space on the stack.

Alternatives

- ▶ ascending - “grows” upwards
- ▶ empty - SP points at next available space rather than last occupied space.

# Full Descending Stack operations

**PUSH** Decrement SP, then store data to where SP points

**POP** Retrieve data from where SP points, then increment SP.

(Questions: how would these be different for an ascending full stack? an ascending empty stack?)

# Full Descending Stack operations

## Cortex-M4 Syntax

- ▶ `PUSH{cond} regs-list`
- ▶ `POP{cond} regs-list`

where:

- ▶ *cond* is an optional condition code, as before: the operation occurs only if the condition is true
- ▶ *regs-list* is a comma-separated list in braces `{ }` of registers or ranges or registers (eg R0-R3, etc)
- ▶ PUSH stores the register contents with higher-numbered registers at higher addresses, starting at `SP-4`. At the end, `SP` points at the lowest stored value.
- ▶ POP loads the registers from the stack, assuming higher-numbered registers are at higher addresses, and post-increments `SP` by 4 times the number of registers popped.

# Full Descending Stack operations

- ▶ Do not include the SP in the register list;
- ▶ Special restrictions apply to including the PC in the register list.
- ▶ You normally *do* need to push th LR if you want to be able to nest subroutine calls.
- ▶ These instructions do not change the condition code flags.

## Examples

- ▶ PUSH {R0,R4-R7}
- ▶ PUSH {R0-R2,LR}
- ▶ POP {R5-R8}

Question: describe the effect of following the last two examples one after the other.

## Subroutines - basic idea

The black code contains a call to the function defined by the brown code:

```
....  
BL myFunc  
....
```

myFunc FUNCTION

```
....  
....; do some complicated stuff  
BX LR  
ENDFUNC
```

- ▶ The **brown** code defines a function labelled myFunc
- ▶ A branch to myFunc starts executing it.
- ▶ The last thing myFunc does is BX LR: branch to the address stored in the link register...
- ▶ ... so the calling code should brach with BL myFunc - branch to myFunc storing the return address in the link register. This is the PC value of the instruction *after* BL myFunc.

## Subroutines - example

A simple function to compute largest (unsigned) integer  $\leq \sqrt{x}$ :

```
typedef unsigned int uint32_t;
```

```
uint32_t usqt(uint32_t x) {  
    uint32_t n = 0x8000,  
             m = 0;  
    while (n != 0) {  
        m += n;  
        if (m*m > x) {  
            m -= n;  
        }  
        n >>= 1;  
    }  
    return m;  
}
```




## Subroutines - example

Here is an assembly language version:

```
1:      PRESERVE8 ; Indicate the code here preserve
2:              ; 8 byte stack alignment
3:      THUMB     ; Indicate THUMB code is used
4:      AREA     |.text|, CODE, READONLY
5:      EXPORT  __main
6:      ENTRY
7:  __main      FUNCTION
8:              ; initialize registers
9:      MOV     r0, #1000 ; Starting loop counter value
10:     BL     usqt
11:     deadlock
12:     B      deadlock ; Infinite loop
13:     ENDFUNC
14:
```

## Subroutines - example

```
15: usqt          FUNCTION
16:             ; compute highest 32-bit uint < value in r0
17:             ; return value in r0
18:             PUSH {r1-r3}
19:             MOV r1, #0x8000 ; n = 0x8000
20:             MOV r2, #0      ; m = 0
21: usqLoop
22:             ADD r2, r2, r1  ; m += n
23:             MUL r3, r2, r2  ; r3 = m*m
24:             CMP  r3, r0
25:             BLS  jump       ; if unsigned lower or same as x
26:             SUB  r2, r2, r1 ; skip this step
27: jump
28:             LSRS r1, r1, #1
29:             BNE usqLoop
30:             MOV  r0, r2      ; copy result into r2
31:             POP  {r1-r3}
32:             BX  LR
33:             ENDFUNC
34:             END              ; End of file
```



## Subroutines - example

- ▶ Line 10 in the main function calls the subroutine function `usqt` with BL `usqt`
- ▶ The function is actually defined on lines 15-33. You will need to satisfy yourself these are equivalent to the C code above.
- ▶ Line 32 ends the function call with a branch to the address saved in LR by the line 10.
- ▶ The function gets the parameter `x` in register `r0`, uses registers `r1`, `r2`, `r3` as “local variables” and returns the result in `r0`.
- ▶ IT IS GOOD PRACTICE therefore to PUSH these registers onto the stack at the beginning (line 18) and restore them with a POP at the end of the call, line 31.
- ▶ It is also common practice to push LR onto the stack (and pop it off at the end) as well: ESSENTIAL if this function itself calls another function.
- ▶ In fact, **push and pop all registers the function alters in any way.**
- ▶ In this case, register `r0` is excluded because it is used to pass the return value back to the calling code.

## Calling a subroutines from C - main.c

```
1: int fact(int x);  
2: int x, y;  
3:  
4: int main() {  
5:   x = 4;  
6:   y = fact(x);  
7:   return 0;  
8: }
```

## Calling a subroutines from C - main.c

```
1:    PRESERVE8 ; Indicate the code here preserve
2:                ; 8 byte stack alignment
3:    THUMB      ; Indicate THUMB code is used
4:    AREA      |.text|, CODE, READONLY ; Start of CODE area
5:    EXPORT fact
6:    ENTRY
7: fact        FUNCTION
8:            ; expects arg in R0; returns val in R0
9:    PUSH {R1,R2, LR} ; save working registers and LR
10:   MOVS R1, R0 ; if nonzero, recursive call ...
11:   BNE recvCall ; else ...
12:   MOV R0, #1
13:   B return
14: recvCall
15:   SUB R0, #1
16:   BL fact
17:   MUL R0, R0, R1
18: return
19:   POP {R1,R2, LR} ; restore saved registers
20:   BX LR
21:   ENDFUNC
22:   END ; End of file
```