

Functions (10A)

Copyright (c) 2014 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

<https://thinkingeek.com/arm-assembler-raspberry-pi/>

Instructions for procedures

B {cond} **label** ; branch to label
BX {cond} **Rm** ; branch **indirect** to location specified by Rm
BL {cond} **label** ; branch to *subroutine* at label
BLX{cond} **Rm** ; branch to *subroutine* **indirect** specified by Rm

BL (Branch and Link)

BL {cond} label ; branch to *subroutine* at label

The call to subroutine instruction

The address of the subroutine is specified by the label

Saves the the return address

(the address of the next instruction)

in the **LR** (Link Register, **R14**)

The range of the BL instruction is

-16MB to +16MB from the current instruction

May use W suffix to get the maximum branch range (width selection)

BX (Branch eXchange)

BX {cond} **Rm** ; branch **indirect** to location specified by **Rm**

A branch indirect instruction

The branch instruction is specified by **Rm**

This instruction causes a UsageFault exception

If bit[0] of **Rm** is 0

Rm[0] = 1, the processor switched to the **Thumb** execution mode

Rm[1] = 0, the processor continues to the **ARM** execution mode

To return from subroutine

```
BX    LR  
MOV  PC, LR
```

Instructions for procedures

```
uint32_t  Num;

void Change1(void) {
    Num = Num + 25;
}

void main(void) {
    Num = 0;
    while (1) {
        Change1();
    }
}
```

```
uint32_t  Num;

void Change2(void) {
    if (Num <25600) {
        Num = Num + 25;
    }
}

void main(void) {
    Num = 0;
    while (1) {
        Change2();
    }
}
```

```
uint32_t  Num;

void Change3(void) {
    if (Num <100) {
        Num = Num + 1;
    } else {
        Num = -100;
    }
}

void main(void) {
    Num = 0;
    while (1) {
        Change3();
    }
}
```

My notations

```
uint32_t Num =0;
```

address content
&NUM Num

In C,
Num is a content of a location

```
NUM    EQU    0
```

address
NUM 0

In assembly,
NUM is an address of a location

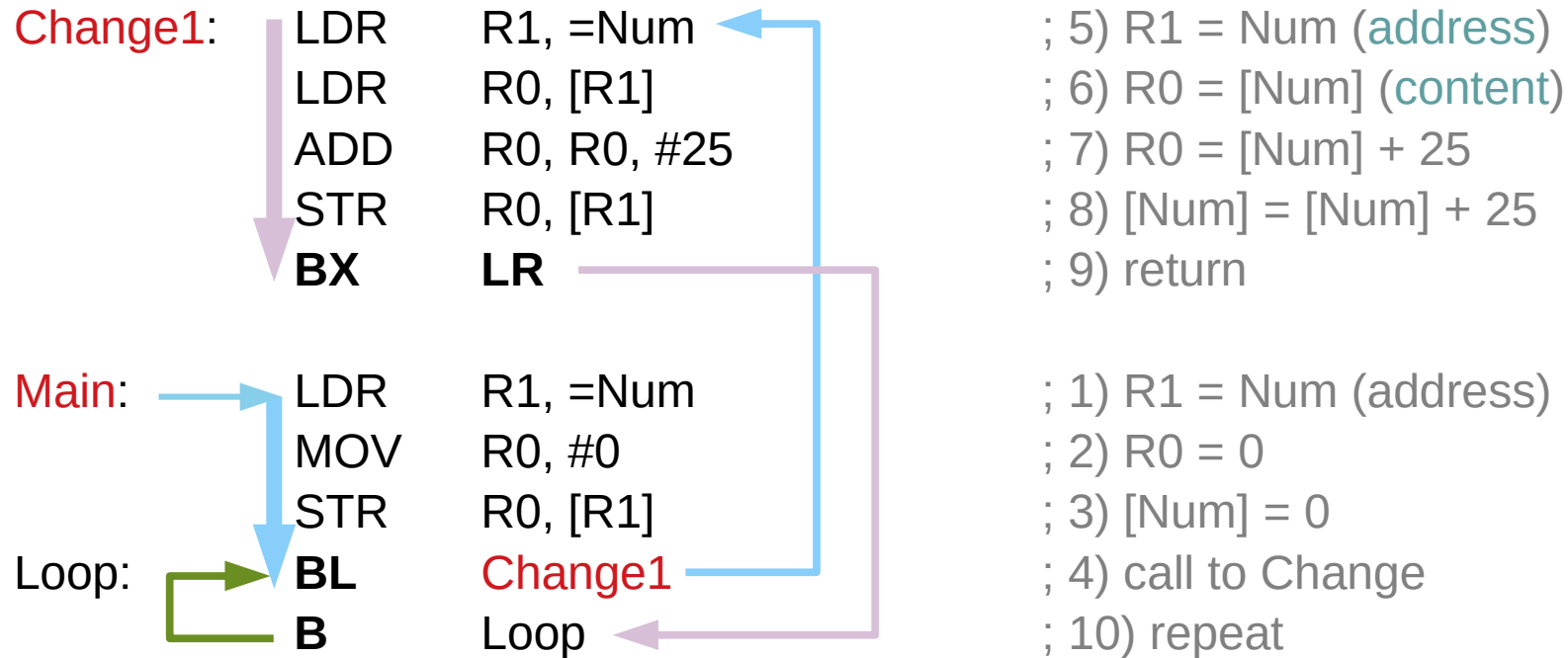
```
LDR    R1, =Num ; R1 = Num
```

considering R1
as the content of a location

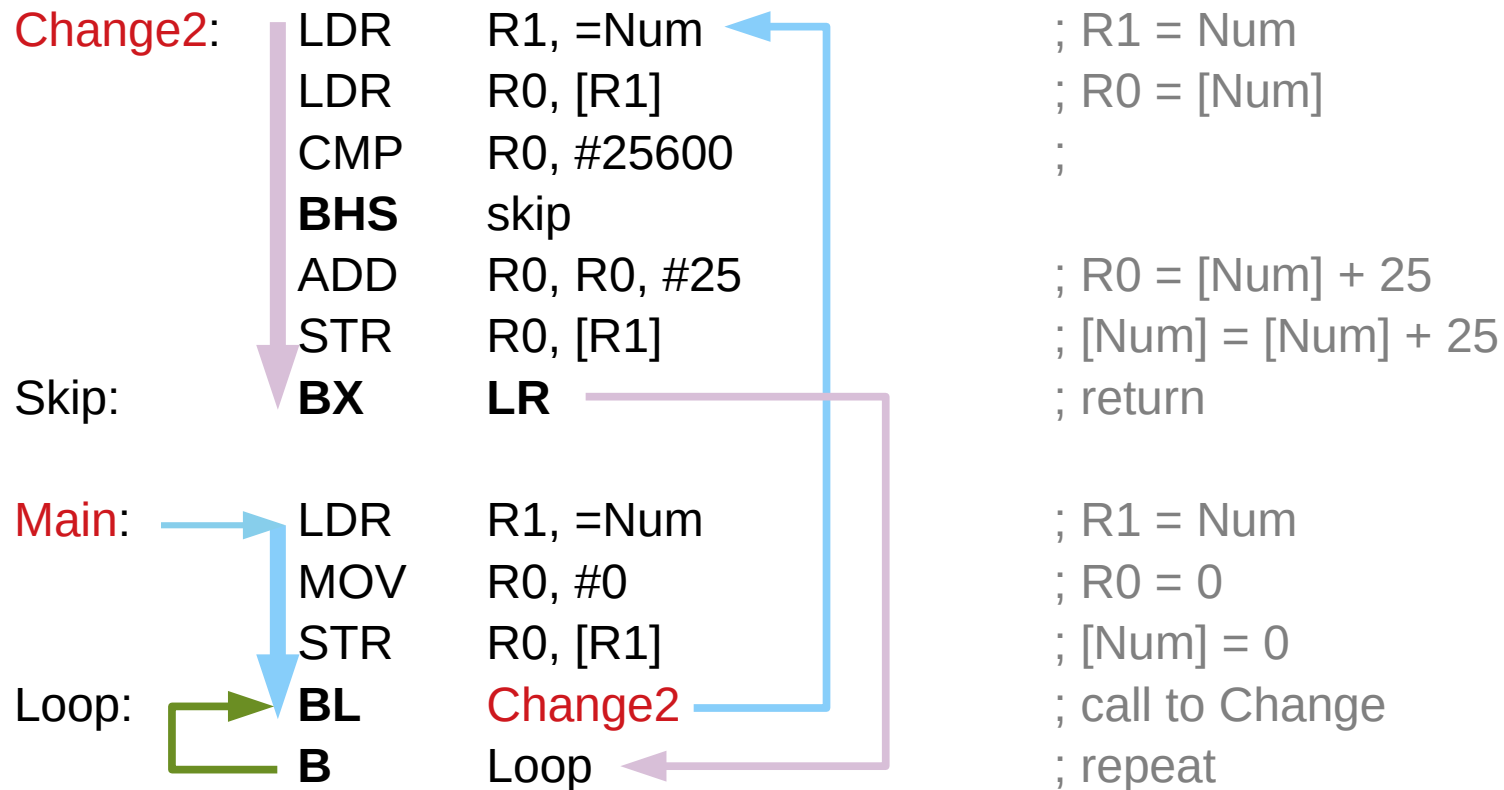
then the value of the content R1 is
the address NUM

R1

Using Change1 function



Using Change2 function



Using Change3 function

Change3:	LDR R1, =Num	; R1 = Num
	LDR R0, [R1]	; R0 = [Num]
	CMP R0, #100	;
	BGE else	
	ADD R0, R0, #1	; [Num] = [Num] + 1
	B skip	
else:	MOV R0, #-100	; R0 = -100
skip:	STR R0, [R1]	; [Num] = [Num] + 1 or -100
	BX LR	; return
Main:	LDR R1, =Num	; R1 = Num
	MOV R0, #0	; R0 = 0
	STR R0, [R1]	; [Num] = 0
Loop:	BL Change3	; call to Change
	B Loop	; repeat

Pointer access to an array

Supporting Procedures

1. put **parameters** in a place where the procedure can access them
2. transfer control to the procedure
3. acquire the **storage resources** needed for the procedure
4. perform the desired task
5. put the **result value** in a place where the calling program can access it
6. return control to the points of origin, since a procedure can be called from several points in a program

Argument registers and return register

R0, R1, R2, R3 : four **argument registers** to pass parameters

Func (**arg1, arg2, arg3, arg4**)



the callee assumes that the caller provides
the necessary arguments in registers **R0, R1, R2, R3**

When more than 4 arguments are passed,
the extra arguments are passed on the **stack**,

the **SP** points to them at the entry to the function.

Link register

LR : one **link register** containing the **return address** register to the point of origin

The *link* portion of the name **LR** means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address

this link stored in register **LR (R14)** is called the **return address**

the return address is needed because the same procedure could be called from several parts of the program

Instructions for procedures

BL ProcedureAddress
return address

*BL stores the return address to LR register
PC+4 → LR*

transfer control to the procedure

jumps to an address and simultaneously saves (links)
the address of the following instruction in register **LR**

MOV PC, LR

return control to the points of origin

Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

Passing Arguments

```
int main (void)
{
    ...
    leaf_example (1, 2, 3, 4);
    ...
}
```

```
; g : R0, h : R1, i : R2, j : R3
```

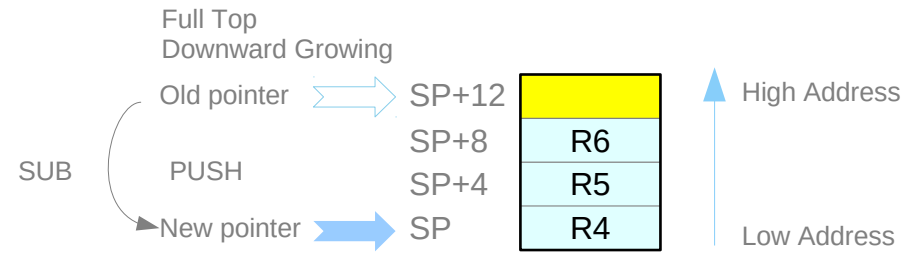
```
MOV    R0, #1        ; g = 1
MOV    R1, #2        ; h = 2
MOV    R2, #3        ; i = 3
MOV    R3, #4        ; j = 4
```

```
BL     leaf_example
```

Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

Function Prologue

```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i+j);
    return f;
}
```



; g : R0, h : R1, i : R2, j : R3

```
SUB    SP, SP, #12    ; adjust stack to make room for 3 items
STR    R6, [SP, #8]   ; save register R6 for a later use      ; (g+h) - (i+j)
STR    R5, [SP, #4]   ; save register R5 for a later use      ; (i+j)
STR    R4, [SP, #0]   ; save register R4 for a later use      ; (g+h)
```

Function Body

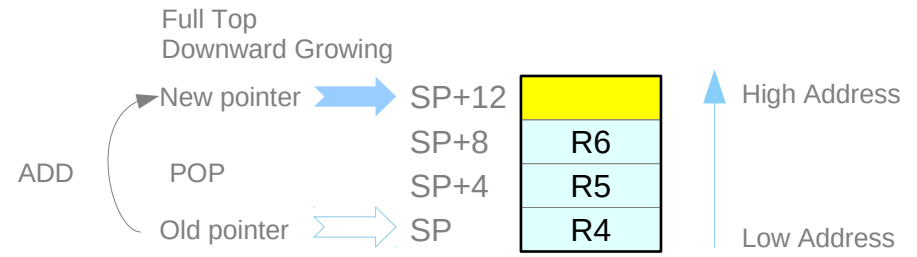
```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i+j);
    return f;
}
```

```
ADD    R5, R0, R1    ; R5 = g + h
ADD    R6, R2, R3    ; R6 = i + j
SUB    R4, R5, R6    ; R4 = R5 - R6

MOV    R0, R4        ; returns f (R0 = R4)
```

Function Epilogue

```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i+j);
    return f;
}
```



```
LDR    R4, [SP, #0]    ; restore R4 for the caller
LDR    R5, [SP, #4]    ; restore R5 for the caller
LDR    R6, [SP, #8]    ; restore R6 for the caller
ADD    SP, SP, #12     ; adjust stack to delete 3 items
```

```
MOV    PC, LR          ; jump back to calling procedure
```

Argument, scratch, variable, return result registers

R0, R1, R2, R3, R12 :

argument or scratch registers

that are not preserved by the **callee** on a procedure call

R4, R5, R6, R7, R8, R9, R10, R11 :

8 **variable registers** that must be preserved on a procedure call
(if used, the **callee** must save and restore them)

R0, R1 :

return result registers

The called performs the calculations,
places the result (if any) in **R0** and **R1**
and returns control to the caller using **MOV PC, LR**

Argument, scratch, variable, return result registers

Registers that is preserved across a procedure

variable registers **r4 – r11**

stack pointer register **sp**

link register **lr**

stack above the stack pointer

Registers that is not preserved across a procedure

argument registers **r0 – r3**

intra procedure call scratch register **r12**

stack below the stack pointer

ARM Register Conventions

Names	Reg No	Usage	preserved
a1-a2	0-1	Argument / return result/ scratch register	no
a3-a4	2-3	Argument / scratch register	no
v1-v8	4-11	Variables for local routine	yes
ip	12	Intra procedure call scratch register	no
sp	13	Stack pointer	yes
lr	14	Link register (Return address)	yes
pc	15	Program counter	n.a.

Calling Convention (ARM32)

in the prologue, **push** **r4 ~ r11** to the stack,
push the **return address** in **r14** to the stack
(this can be done with a single **STM** instruction)

scratch registers to be used
LR

copy any passed **arguments** (in **r0 ~ r3**)
to the local scratch registers (**r4 ~ r11**);

r0 ~ r3 **r4 ~ r11**

allocate other **local variables** to the remaining
local **scratch registers** (**r4 ~ r11**);

r4 ~ r11

using **BL**, do calculations and / or call other subroutines
assuming **r0 ~ r3**, **r12** and **r14** will not be preserved;

put the result in **r0**;

In the epilogue, pop **r4 ~ r11** from the stack,
and pull the **return address** to the program counter **r15**.
(this can be done with a single **LDM** instruction)

[https://en.wikipedia.org/wiki/Calling_convention#ARM_\(A32\)](https://en.wikipedia.org/wiki/Calling_convention#ARM_(A32))

PUSH, POP Synonyms

PUSH{cond} reglist
POP{cond} reglist

Synonyms

PUSH = **STMDB R13!** = **STMFD R13!**
POP = **LDMIA R13!** or even **LDM** = **LDMFD R13!**

Assume

the base register **SP (R13)**

the adjusted address **written back** to the base register

registers are stored on the stack in **numerical order**

with the lowest numbered register at the lowest address.

Full Descending Stack with SP (=R13)

More than 4 arguments

the extra arguments are passed on the stack,
the SP points to them at the entry to the function.
In the prolog, you're pushing registers to be saved,
and this changes the SP, so you need to account for it.

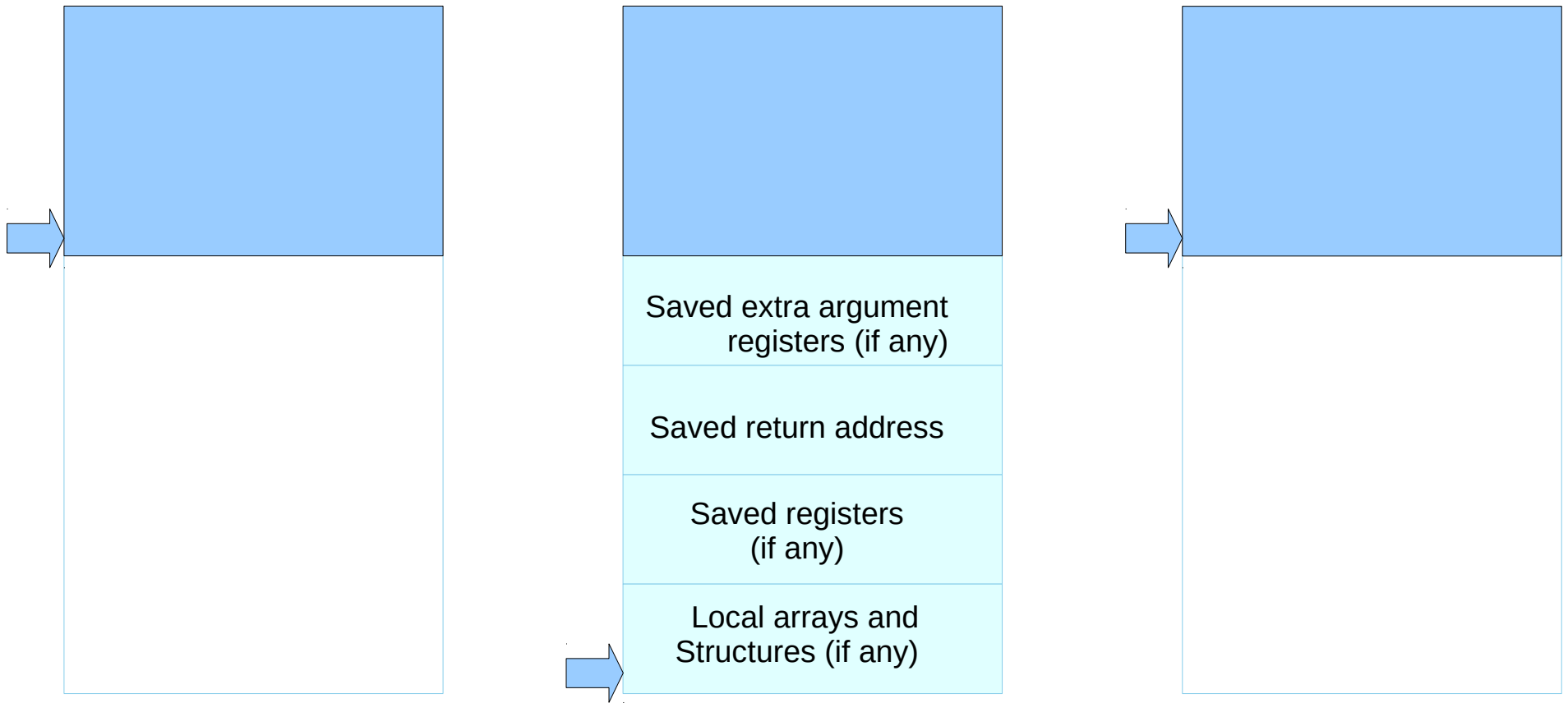
r4, r5, r6, r7, r8 and **lr** are 6 registers,
so you need to adjust your SP offsets
by $6 \times 4 = 24$ bytes.

```
push {r4-r8,lr}      // 6 regs are pushed // SP is decremented by  $6 \times 4 = 24$  bytes  
ldr r6, [sp, #(0+24)] // get first stack arg  
ldr r7, [sp, #(4+24)] // get second stack arg
```

If you do more manipulations with SP, e.g. allocate space for stack vars, you might have to take that into account too.

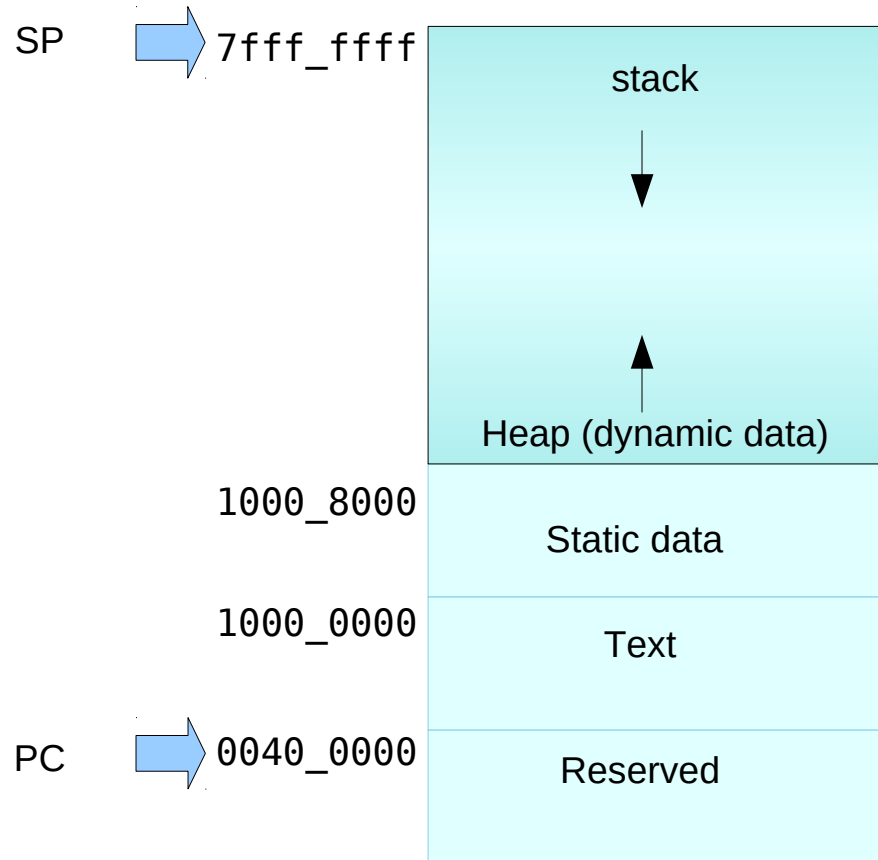
<https://stackoverflow.com/questions/15071506/how-to-access-more-than-4-arguments-in-an-arm-assembly-function>

Stack allocation



Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

Memory map



Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

Recursive procedure

```
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

```
fact(3)
    fact(2)
        fact(1)
            return(1)
        return(2*1)
    return(3*2)
```

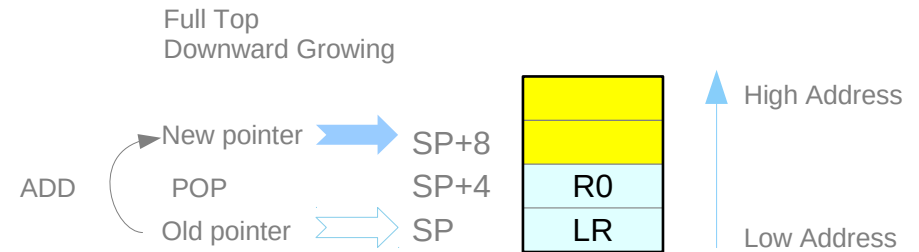
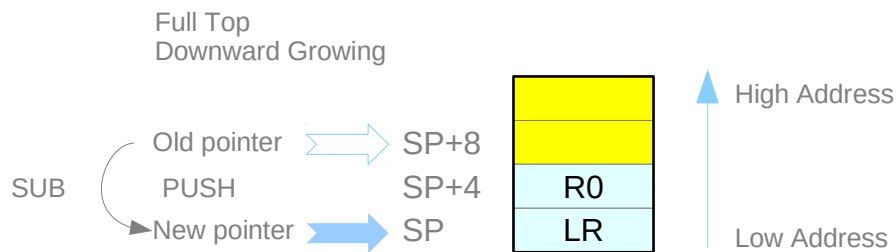
Recursive procedure

fact:

```
SUB    SP, SP, #8           ; adjust stack for 2 items
STR    LR, [SP, #0]        ; save the return address
STR    R0, [SP, #4]        ; save the argument n

CMP    R0, #1              ; compare n to 1
BGE  L1                  ; if n >= 1, go to L1

MOV    R0, #1              ; return 1
ADD    SP, SP, #8          ; pop 2 items off stack
MOV  PC, LR              ; return to the caller
```



Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

Recursive procedure

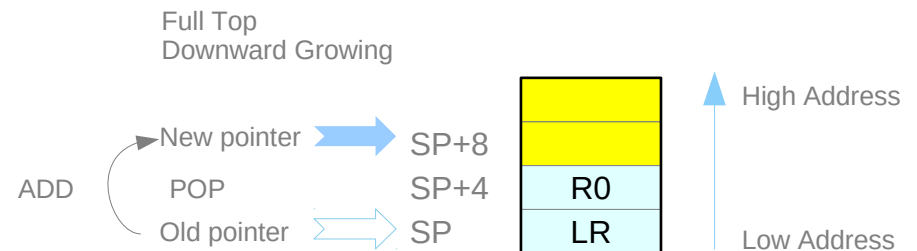
L1:

```
SUB    R0, R0, #1           ; n >= 1 argument gets (n-1)
BL   fact                 ; call fact with (n-1)

MOV    R12, R0              ; save the return value
LDR    R0, [SP, #4]         ; return from BL ; restore argument n
LDR    LR, [SP, #0]        ; restore the return address
ADD    SP, SP, #8          ; adjust stack pointer to pop 2 items

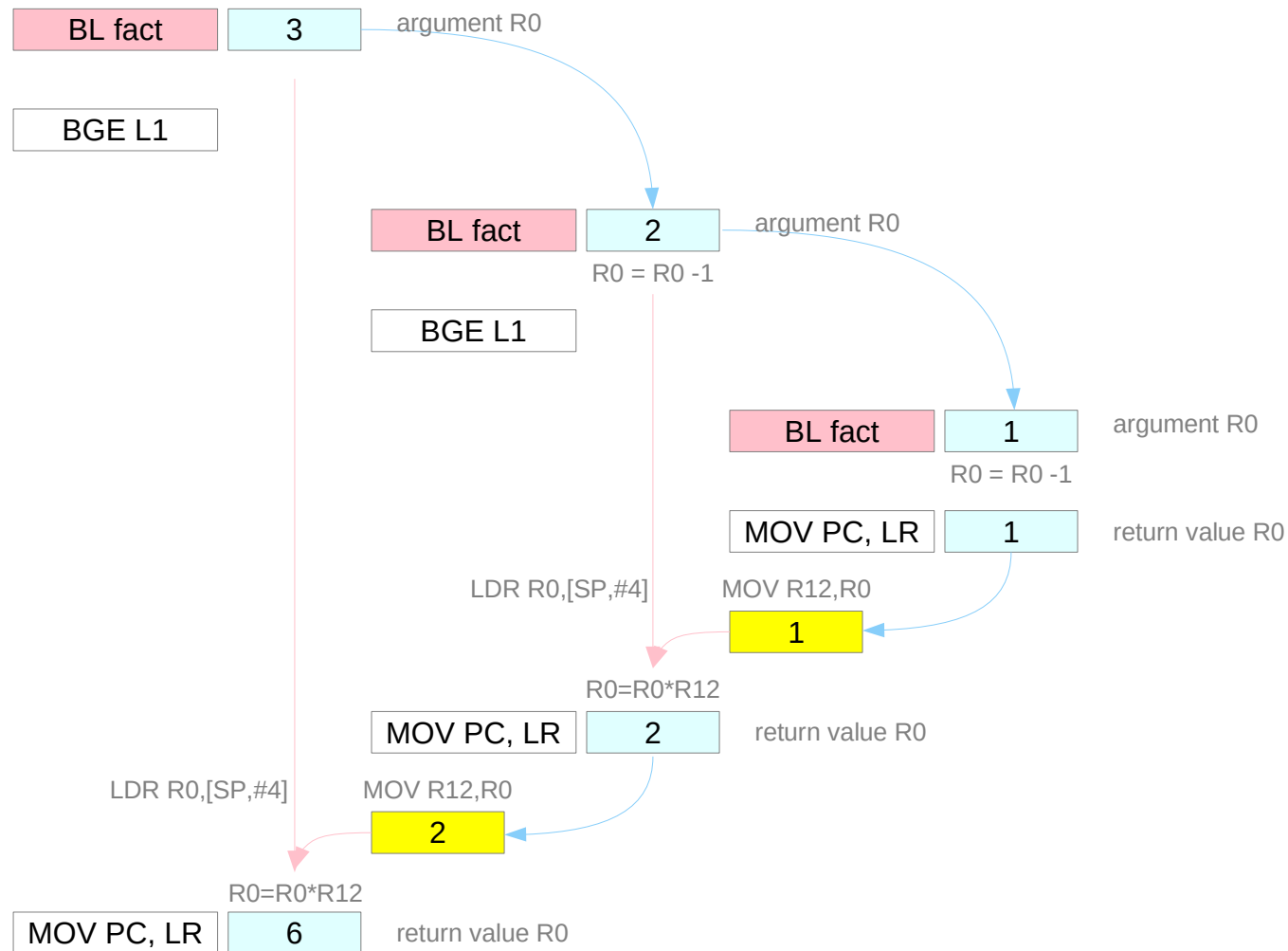
MUL    R0, R0, R12          ; return n * fact (n-1)
MOV    PC, LR              ; return to the caller
```

R12 : IP Intra procedure call scratch register



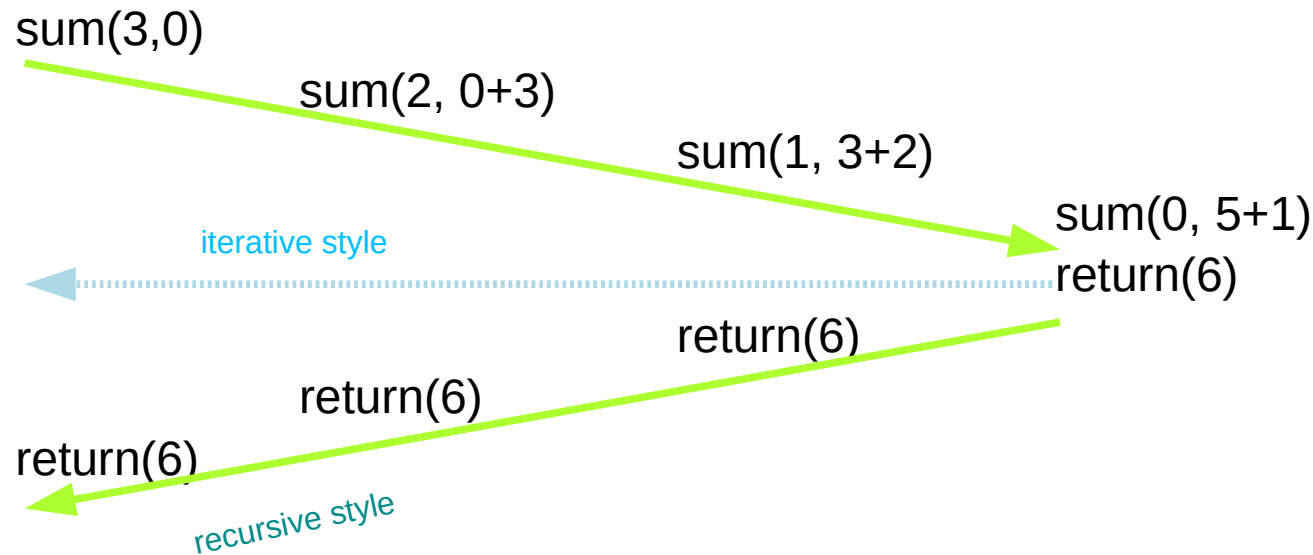
Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

Recursive procedure



Recursive Procedure and Iterative Implementation

```
int sum (int n, int acc) {  
    if (n > 0)  
        return sum(n-1, acc+n);  
    else  
        return acc;  
}
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Recursive Procedure and Iterative Implementation

```
sum:  CMP    R0, #0           ; test if n <= 0
      BLE   sum_exit      ; go to sum_exit if n <= 0;
      ADD   R1, R1, R0    ; add n to acc
      SUB   R0, R0, #1    ; subtract 1 from n
      B     sum           ; go to sum

sum_exit:  MOV   R0, R1      ; return value acc
          MOV   PC, LR     ; return to caller
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

String Copy Procedure

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0')    // copy & test byte
        i += 1;
}
```

String Copy Procedure

```
strcpy: SUB    SP, SP, #4      ; adjust stack for 1 more item
        STR    R4, [SP, #0]  ; save R4
        MOV    R4, #0        ; i = 0 + 0
L1:     ADD    R2, R4, R1     ; address of y[i] in R2
        LDRBS  R3, [R2, #0]  ; R3 = y[i] and set condition flag
        ADD    R12, R4, R0   ; address of x[i] in r12
        STRB   R3, [R12, #0] ; x[i] = y[i]
        BEQ    L2           ; if y[i] == 0, go to L2
        ADD    R4, R4, #1    ; i = i+1
        B     L1           ; go to L1
L2:     LDR    R4, [SP, #0]   ; y[i] == 0 : end of string, restore old R4
        ADD    SP, SP, #4    ; pop 1 word off stack
        MOV    PC, LR       ; return
```

Swap (1)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Swap (2) - using RN directive

v **RN** 0 ; 1st argument address of v
k **RN** 1 ; 2nd argument index k
temp **RN** 2 ; local variable
temp2 **RN** 3 ; temporary for v[k+1]
vkAddr **RN** 12 ; to hold address of v[k]

R0	v	; 1st argument address of v
R1	k	; 2nd argument index k
R2	temp	; local variable
R3	temp2	; temporary for v[k+1]
R12	vkAddr	; to hold address of v[k]

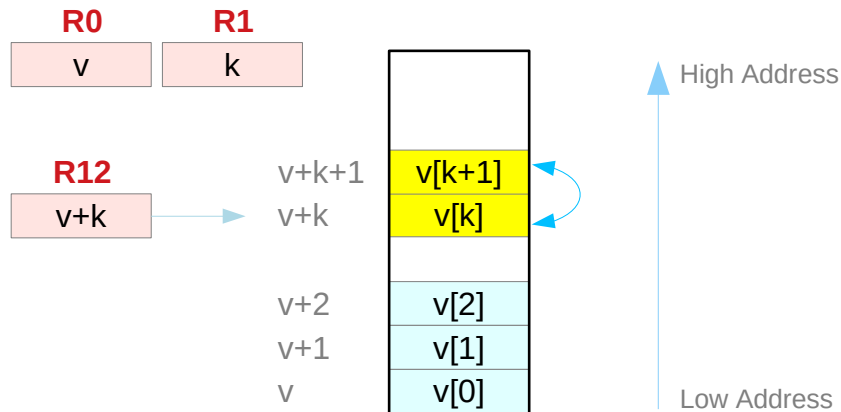
Swap (3)

```
swap:  ADD    vkAddr, v, k, LSL #2    ; reg vkAddr = v + (k * 4)
      ; reg vkAddr has the address of v[k]
      LDR    temp, [vkAddr, #0]      ; temp = v[k]
      LDR    temp2, [vkAddr, #4]    ; temp2 = v[k+1]
      ; refers to next element of v
      STR    temp2, [vkAddr, #0]    ; v[k] = temp2
      STR    temp, [vkAddr, #4]    ; v[k+1] = temp

      MOV    PC, LR                ; return to calling routine
```

Swap (4)

```
swap:  ADD    R12, R0, R1, LSL #2    ; reg vkAddr = v + (k * 4)
      LDR    R2, [R12, #0]         ; reg vkAddr has the address of v[k]
      LDR    R3, [R12, #4]         ; temp = v[k]
      STR    R3, [R12, #0]         ; temp2 = v[k+1]
      STR    R2, [R12, #4]         ; refers to next element of v
      STR    R2, [R12, #4]         ; v[k] = temp2
      STR    R3, [R12, #0]         ; v[k+1] = temp
      MOV    PC, LR                ; return to calling routine
```



R0	v	; 1st argument address of v
R1	k	; 2nd argument index k
R2	temp	; local variable
R3	temp2	; temporary for v[k+1]
R12	vkAddr	; to hold address of v[k]

Sort (1)

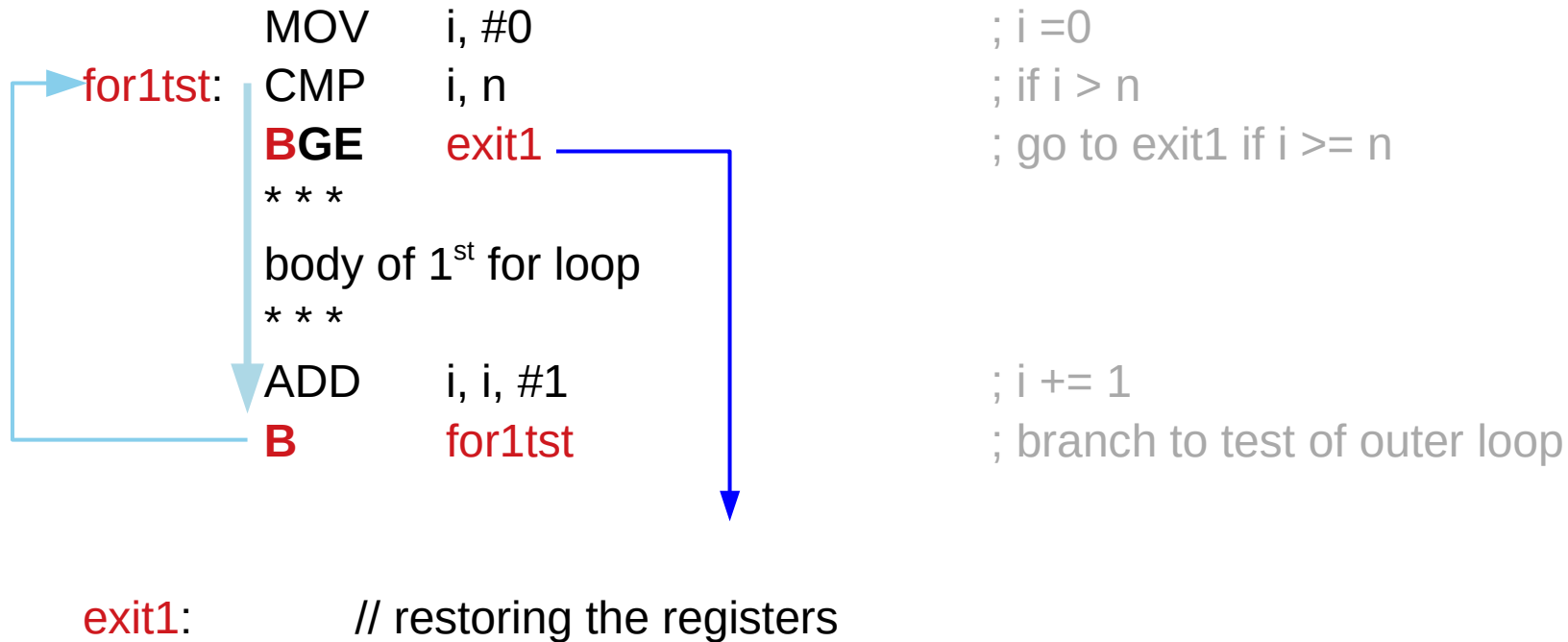
```
void sort(int v[], int n)
{
    int i, j;
    for (i=0, i<n, ++i) {
        for (j=i-1; j >= 0 && v[j] > v[j+1]); --j) {
            swap(v, j);
        }
    }
}
```

Sort (2) – using RN directive

v	RN 0	; 1st argument address of v
n	RN 1	; 2nd argument index n
i	RN 2	; local variable i
j	RN 3	; local variable j
vjAddr	RN 12	; to hold address of v[j]
vj	RN 4	; to hold a copy of v[j]
vj1	RN 5	; to hold a copy of v[j+1]
vcopy	RN 6	; to hold a copy of v
ncopy	RN 7	; to hold a copy of n

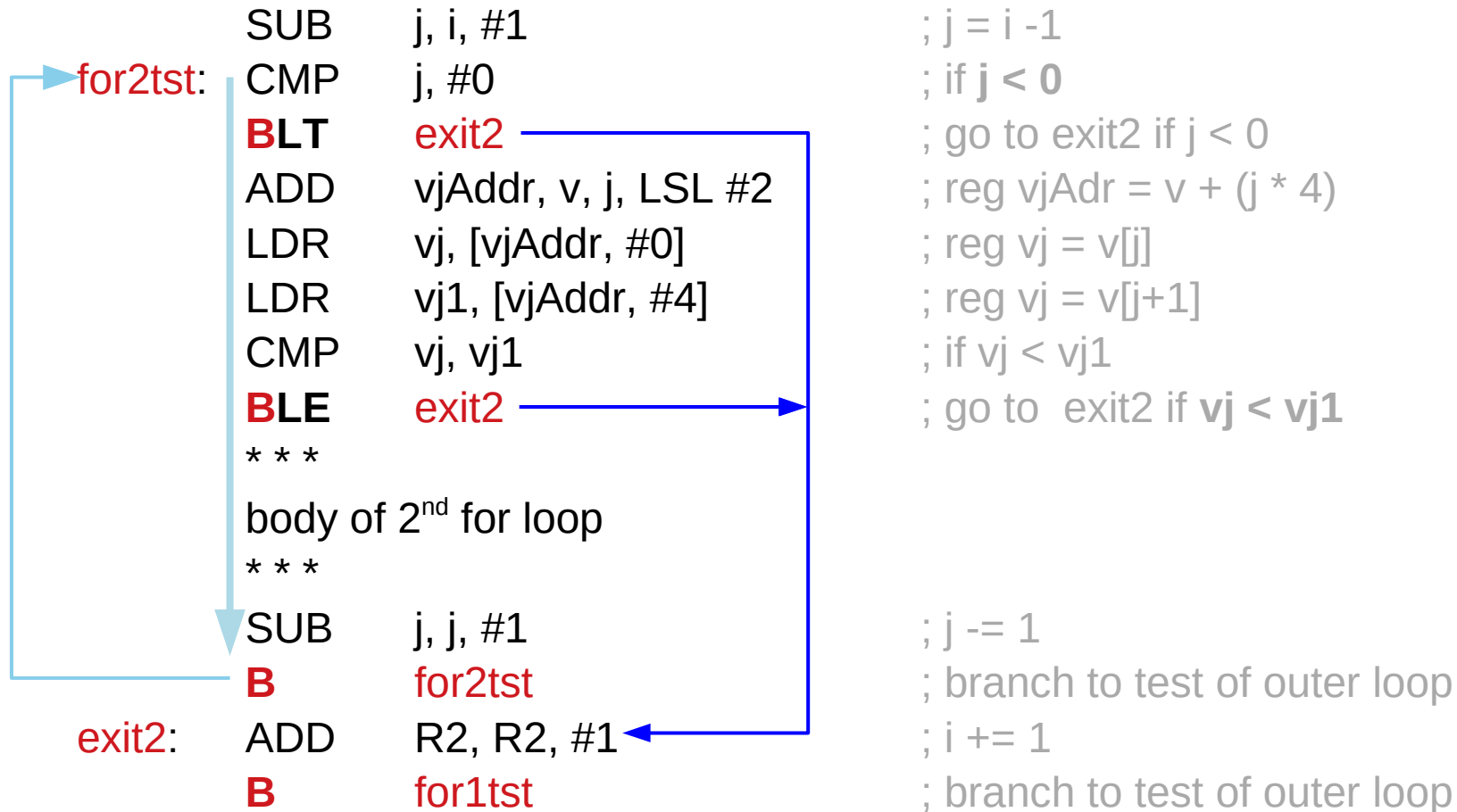
Sort (3) outer loop

for (i=0, i<n, ++i)



Sort (4) inner loop

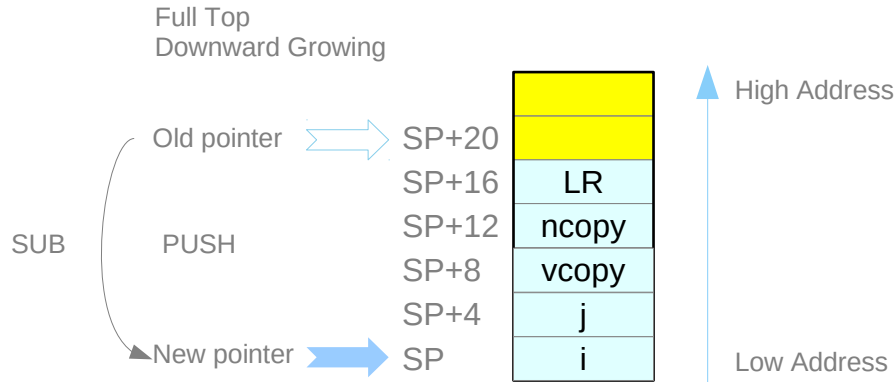
```
for ( j=i-1; j >= 0 && v[j] > v[j+1]); --j )
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Sort (5) Saving registers

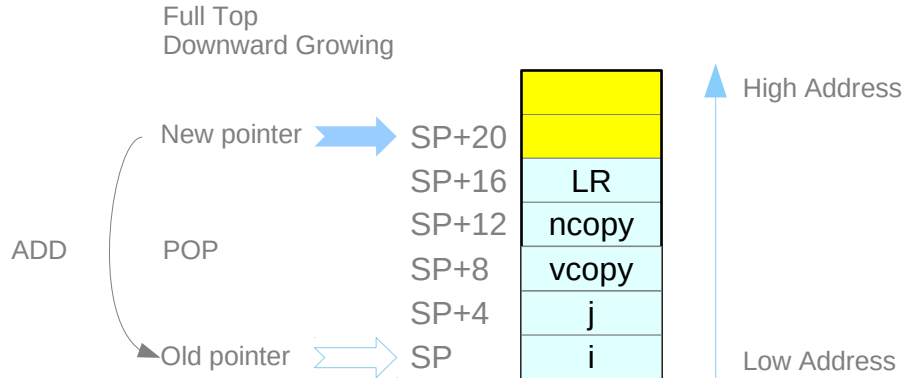
```
sort:  SUB    SP, SP, #20      ; make room on stack for 5 registers
      STR    LR, [SP, #16]   ; save LR on stack
      STR    ncopy, [SP, #12] ; save ncopy on stack
      STR    vcopy, [SP, #8] ; save vcopy on stack
      STR    j, [SP, #4]     ; save j on stack
      STR    i, [SP, #0]    ; save i on stack
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Sort (6) Restoring registers

```
exit1:  LDR    i, [SP, #0]           ; restore i from stack
        LDR    j, [SP, #4]       ; restore j from stack
        LDR    vcopy, [SP, #8]   ; restore vcopy from stack
        LDR    ncopy, [SP, #12]  ; restore ncopy from stack
        LDR    LR, [SP, #16]    ; restore LR from stack
        ADD    SP, SP, #20       ; restore stack pointer
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

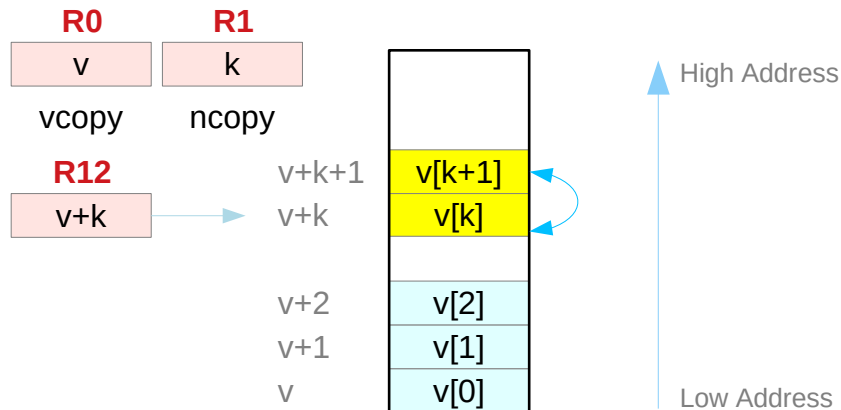
Sort (7) Calling swap

```
swap(v, j);
```

```
MOV    vcopy, v           ; copy parameter v into vcopy (save R0)
MOV    ncopy, n          ; copy parameter n into ncopy (save R1)
```

```
BL   swap
```

```
MOV    R0, vcopy         ; first swap parameter is v
MOV    R1, j              ; second swap parameter is j (new n)
```



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Sort full listing (1)

Saving Registers

```
sort:      SUB  SP, SP, #20      ; make room on stack for 5 registers
           STR  LR, [SP, #16]   ; save LR on stack
           STR  R7, [SP, #12]  ; save ncopy on stack
           STR  R6, [SP, #8]   ; save vcopy on stack
           STR  R3, [SP, #4]   ; save j on stack
           STR  R2, [SP, #0]   ; save i on stack
```

Procedure Body

for1tst:

for2tst:

exit2:

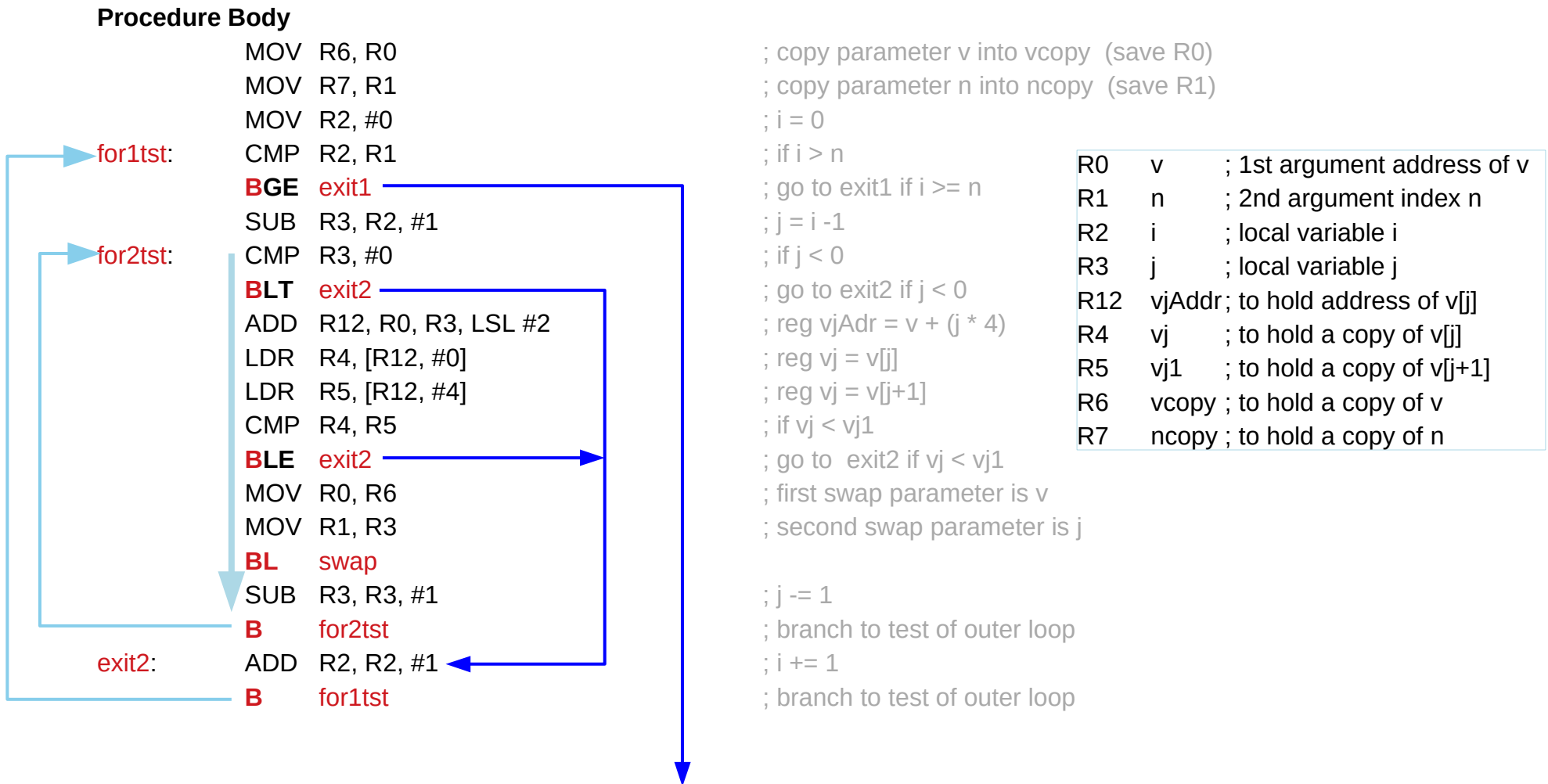
R2	i	; local variable i
R3	j	; local variable j
R6	vcopy	; to hold a copy of v
R7	ncopy	; to hold a copy of n

Restoring Registers

exit1:

Procedure Return

Sort full listing (2)



Sort full listing (3)

Restoring Registers

```
exit1:    LDR  R2, [SP, #0]    ; restore i from stack
          LDR  R3, [SP, #4]    ; restore j from stack
          LDR  R6, [SP, #8]    ; restore vcopy from stack
          LDR  R7, [SP, #12]   ; restore ncopy from stack
          LDR  LR, [SP, #16]   ; restore LR from stack
          ADD  SP, SP, #20     ; restore stack pointer
```

Procedure Return

```
MOV  PC, LR    ; return to calling routine
```

R2	i	; local variable i
R3	j	; local variable j
R6	vcopy	; to hold a copy of v
R7	ncopy	; to hold a copy of n

Hello world (1)

```
; hello01.s
.data

greeting:
.asciz "Hello world"

.balign 4
return: .word 0

.text
```

address	contents
greeting	H e l l o w o r l d 0
return	0

```
.global main
main:
ldr r1, address_of_return       ; r1 ← &address_of_return
str lr, [r1]                   ; *r1 ← lr

ldr r0, address_of_greeting     ; r0 ← &address_of_greeting
                              ; First parameter of puts

bl puts                         ; Call to puts
                              ; lr ← address of next instruction

ldr r1, address_of_return       ; r1 ← &address_of_return
ldr lr, [r1]                   ; lr ← *r1
bx lr                          ; return from main

address_of_greeting:       .word greeting
address_of_return:        .word return
```

```
; External
.global puts
```

address	contents
address_of_greeting	greeting
address_of_return	return

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

Hello world (2)

```
; printf01.s
.data

.balign 4
message1: .asciz "Hey, type a number: "
; First message

.balign 4
message2: .asciz "I read the number %d\n"
; Second message

.balign 4
scan_pattern: .asciz "%d"
; Format pattern for scanf

.balign 4
number_read: .word 0
; Where scanf will store the number read

.balign 4
return: .word 0

.text

.global scanf
```

	LSByte		MSByte	
	Byte 0	Byte 1	Byte 2	Byte 3
address	contents			
message1	H	e	y	,
	t	y	p	e
	a	n	u	m
	b			
message2	e	r	:	0
	I	r	e	a
	d	t	h	e
	n	u	m	b
	e			
scan_pattern	r	%	d	\
	n	0		
number_read	%	d	0	
return			0	
			0	

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

Hello world (3)

```
.global main
```

```
main:
```

```
ldr r1, address_of_return      ; r1 ← &address_of_return
str lr, [r1]                    ; *r1 ← lr

ldr r0, address_of_message1    ; r0 ← &message1
bl printf                       ; call to printf

ldr r0, address_of_scan_pattern ; r0 ← &scan_pattern
ldr r1, address_of_number_read  ; r1 ← &number_read
bl scanf                         ; call to scanf

ldr r0, address_of_message2    ; r0 ← &message2
ldr r1, address_of_number_read  ; r1 ← &number_read
ldr r1, [r1]                   ; r1 ← *r1
bl printf                       ; call to printf

ldr r0, address_of_number_read  ; r0 ← &number_read
ldr r0, [r0]                    ; r0 ← *r0

ldr lr, address_of_return       ; lr ← &address_of_return
ldr lr, [lr]                    ; lr ← *lr
bx lr                           ; return from main using lr

message1: .asciz "Hey, type a number: "
message2: .asciz "I read the number %d\n"
scan_pattern: .asciz "%d"
number_read: .word 0
return: .word 0
```

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

Hello world (4)

```
address_of_message1 : .word message1
address_of_message2 : .word message2
address_of_scan_pattern : .word scan_pattern
address_of_number_read : .word number_read
address_of_return : .word return
```

```
; External
.global printf
```

```
$ ./printf01
```

```
Hey, type a number: 123 ↵
I read the number 123
```

```
$ ./printf01 ; echo $?
```

```
Hey, type a number: 124 ↵
I read the number 124
124
```

address	contents
address_of_message1	message1
address_of_message2	message2
address_of_scan_pattern	scan_pattern
address_of_number_read	number_read
address_of_return	return

message1
message2
scan_pattern
number_read
return

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

mult_by_5 function (1)

```
.balign 4
return2:          .word 0

.text

; mult_by_5 function

mult_by_5:
    ldr r1, address_of_return2    ; r1 ← &address_of_return
    str lr, [r1]                  ; *r1 ← lr

    add r0, r0, r0, LSL #2        ; r0 ← r0 + 4*r0

    ldr lr, address_of_return2    ; lr ← &address_of_return
    ldr lr, [lr]                  ; lr ← *lr
    bx lr                          ; return from main using lr

address_of_return2: .word return2
```

address	contents
address_of_return2	return2

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

mult_by_5 function (2)

```
; printf02.s
.data

.balign 4
message1: .asciz "Hey, type a number: "
; First message

.balign 4
message2: .asciz "%d times 5 is %d\n"
; Second message

.balign 4
scan_pattern : .asciz "%d"
; Format pattern for scanf

.balign 4
number_read: .word 0
; Where scanf will store the number read

.balign 4
return: .word 0

.balign 4
return2: .word 0
anf
```

	LSByte		MSByte	
	Byte 0	Byte 1	Byte 2	Byte 3
address				
contents				
message1	H	e	y	,
		t	y	p
		e		a
				n
				u
				m
				b
message2	%	d		
		t	i	m
		e	s	
			5	
			%	d
				\
				n
				0
scan_pattern	%	d		
		0		
number_read	0			
return	0			

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

mult_by_5 function (3)

`.global main`

`main:`

```
ldr r1, address_of_return      ; r1 ← &address_of_return
str lr, [r1]                   ; *r1 ← lr

ldr r0, address_of_message1    ; r0 ← &message1
bl printf                      ; call to printf

ldr r0, address_of_scan_pattern ; r0 ← &scan_pattern
ldr r1, address_of_number_read ; r1 ← &number_read
bl scanf                       ; call to scanf

ldr r0, address_of_number_read ; r0 ← &number_read
ldr r0, [r0]                   ; r0 ← *r0
bl mult_by_5

mov r2, r0                     ; r2 ← r0
ldr r1, address_of_number_read ; r1 ← &number_read
ldr r1, [r1]                   ; r1 ← *r1
ldr r0, address_of_message2    ; r0 ← &message2
bl printf                      ; call to printf

ldr lr, address_of_return      ; lr ← &address_of_return
ldr lr, [lr]                   ; lr ← *lr
bx lr                          ; return from main using lr
```

```
message1: .asciz "Hey, type a number: "
message2: .asciz "%d times 5 is %d\n"
scan_pattern : .asciz "%d"
number_read: .word 0
return: .word 0
return2: .word 0
```

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

mult_by_5 function (4)

```
address_of_message1 : .word message1
address_of_message2 : .word message2
address_of_scan_pattern : .word scan_pattern
address_of_number_read : .word number_read
address_of_return : .word return
```

```
; External
.global printf
```

address	contents
address_of_message1	message1
address_of_message2	message2
address_of_scan_pattern	scan_pattern
address_of_number_read	number_read
address_of_return	return

<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

Dynamic activation

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

The stack

```
sub sp, sp, #8           ; sp ← sp - 8. This enlarges the stack by 8 bytes
str lr, [sp]             ; *sp ← lr
```

... // Code of the function

```
ldr lr, [sp]             ; lr ← *sp
add sp, sp, #8           ; sp ← sp + 8.
                        ; This reduces the stack by 8 bytes
                        ; effectively restoring the stack
                        ; pointer to its original value
```

bx lr

```
str lr, [sp, #-8]!       ; preindex: sp ← sp - 8; *sp ← lr
```

... // Code of the function

```
ldr lr, [sp], #+8       ; postindex; lr ← *sp; sp ← sp + 8
bx lr
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Factorial implementation (1)

```
; factorial01.s
.data

message1:    .asciz "Type a number: "
format:      .asciz "%d"
message2:    .asciz "The factorial of %d is %d\n"

.text

factorial:

is_nonzero:

end:
```

		LSByte				MSByte	
		Byte 0	Byte 1	Byte 2	Byte 3		
address	contents						
message1	Type a number: 0	T	y	p	e	a	n
format	%d 0	%	d	0			
message2	The factorial of	T	h	e	f	a	c
	%d is %d\n 0	%	d	\	n	0	

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Factorial implementation (2)

```
.text
factorial:
    str lr, [sp,#-4]!           ; Push lr onto the top of the stack
    str r0, [sp,#-4]!         ; Push r0 onto the top of the stack
                                ; Note that after that, sp is 8 byte aligned
    cmp r0, #0                ; compare r0 and 0
    bne is_nonzero            ; if r0 != 0 then branch
    mov r0, #1                ; r0 ← 1. This is the return
    b end

is_nonzero:
    sub r0, r0, #1            ; Prepare the call to factorial(n-1)
                                ; r0 ← r0 - 1
    bl factorial

    ldr r1, [sp]              ; After the call r0 contains factorial(n-1)
                                ; Load r0 (that we kept in th stack) into r1
                                ; r1 ← *sp
    mul r0, r0, r1            ; r0 ← r0 * r1

end:
    add sp, sp, #+4           ; Discard the r0 we kept in the stack
    ldr lr, [sp], #+4         ; Pop the top of the stack and put it in lr
    bx lr                     ; Leave factorial
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Factorial implementation (3)

```
.global main
main:
    str lr, [sp,#-4]      ; Push lr onto the top of the stack
    sub sp, sp, #4       ; Make room for one 4 byte integer in the stack
                        ; In these 4 bytes we will keep the number
                        ; entered by the user
                        ; Note that after that the stack is 8-byte aligned
    ldr r0, address_of_message1 ; Set &message1 as the first parameter of printf
    bl printf            ; Call printf

    ldr r0, address_of_format ; Set &format as the first parameter of scanf
    mov r1, sp           ; Set the top of the stack as the second parameter
                        ; of scanf
    bl scanf            ; Call scanf

    ldr r0, [sp]         ; Load the integer read by scanf into r0
                        ; So we set it as the first parameter of factorial
    bl factorial        ; Call factorial

    message1:           .asciz "Type a number: "
    format:             .asciz "%d"
    message2:           .asciz "The factorial of %d is %d\n"
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Factorial implementation (4)

```
mov r2, r0           ; Get the result of factorial and move it to r2
                    ; So we set it as the third parameter of printf
ldr r1, [sp]         ; Load the integer read by scanf into r1
                    ; So we set it as the second parameter of printf
ldr r0, address_of_message2 ; Set &message2 as the first parameter of printf
bl printf          ; Call printf

add sp, sp, #+4     ; Discard the integer read by scanf
ldr lr, [sp], #+4   ; Pop the top of the stack and put it in lr
bx lr             ; Leave main
```

```
address_of_message1: .word message1
address_of_message2: .word message2
address_of_format:   .word format
```

address	contents
address_of_message1	message1
address_of_message2	message2
address_of_format	format

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Factorial implementation (5)

```
factorial:
    str lr, [sp,#-4]!           ; Push lr onto the top of the stack
    str r4, [sp,#-4]!         ; Push r4 onto the top of the stack
                                ; The stack is now 8 byte aligned
    mov r4, r0                ; Keep a copy of the initial value of r0 in r4

    cmp r0, #0                ; compare r0 and 0
    bne is_nonzero            ; if r0 != 0 then branch
    mov r0, #1                ; r0 ← 1. This is the return
    b end

is_nonzero:
    sub r0, r0, #1            ; Prepare the call to factorial(n-1)
                                ; r0 ← r0 - 1
    bl factorial

    mov r1, r4                ; After the call r0 contains factorial(n-1)
                                ; Load initial value of r0 (that we kept in r4) into r1
                                ; r1 ← r4
                                ; r0 ← r0 * r1
    mul r0, r0, r1

end:
    ldr r4, [sp], #+4         ; Pop the top of the stack and put it in r4
    ldr lr, [sp], #+4         ; Pop the top of the stack and put it in lr
    bx lr                     ; Leave factorial
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Factorial implementation (6)

```
str lr, [sp,#-4]!           ; Push lr onto the top of the stack
str r4, [sp,#-4]!          ; Push r4 onto the top of the stack

ldr r4, [sp], #+4          ; Pop the top of the stack and put it in r4
ldr lr, [sp], #+4          ; Pop the top of the stack and put it in lr

push {r4, lr}
pop {r4, lr}
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

References

- [1] <ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf>
- [2] <https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf>