

Stack Frames (12A)

Copyright (c) 2014 - 2021 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

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

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

Nested and recursive function calls

Nested function call

```
int main(void) {  
    f1( ... );  
}  
  
void f1 ( ... ) {  
    f2 ( ... );  
}  
  
void f2 ( ... ) {  
}
```

The diagram illustrates nested function calls. It shows three function definitions: `main`, `f1`, and `f2`. `main` calls `f1`, `f1` calls `f2`, and each function returns to its caller. Arrows indicate the call sequence: from `main` to `f1`, from `f1` to `f2`, and return paths from `f2` back to `f1` and from `f1` back to `main`.

Recursive function call

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

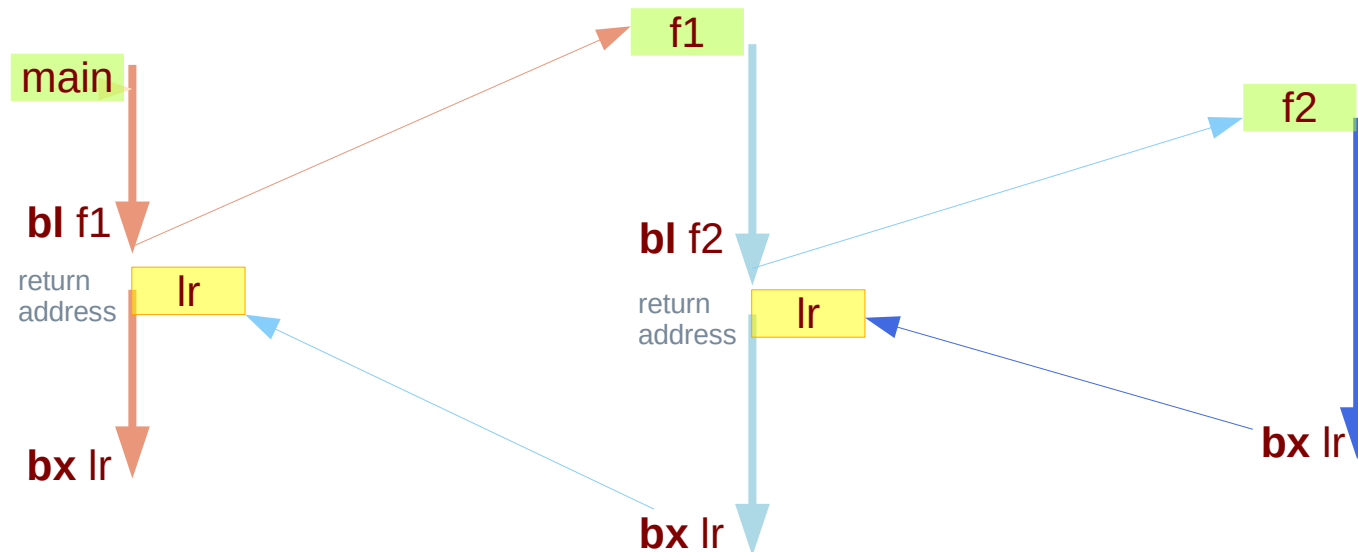
The diagram illustrates recursive function calls. It shows three instances of the `fact` function. Each instance calls itself with a smaller argument. Arrows indicate the call sequence: from the leftmost `fact` to the middle `fact`, and from the middle `fact` to the rightmost `fact`. Return paths are shown from the rightmost `fact` back to the middle `fact`, and from the middle `fact` back to the leftmost `fact`.

Nested and recursive function calls

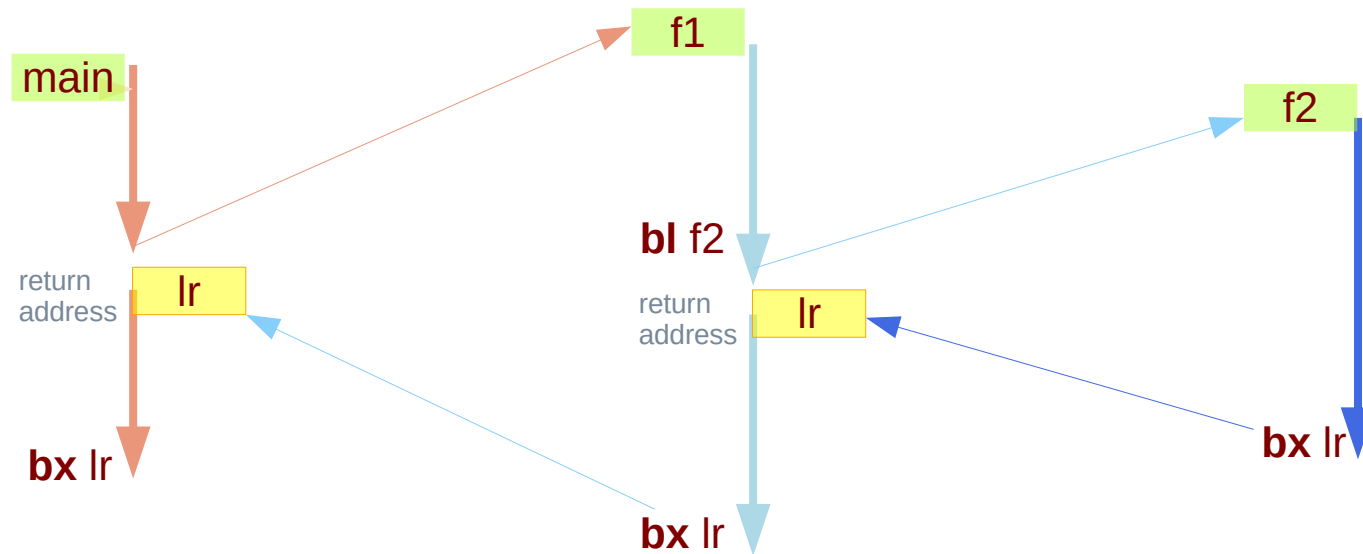
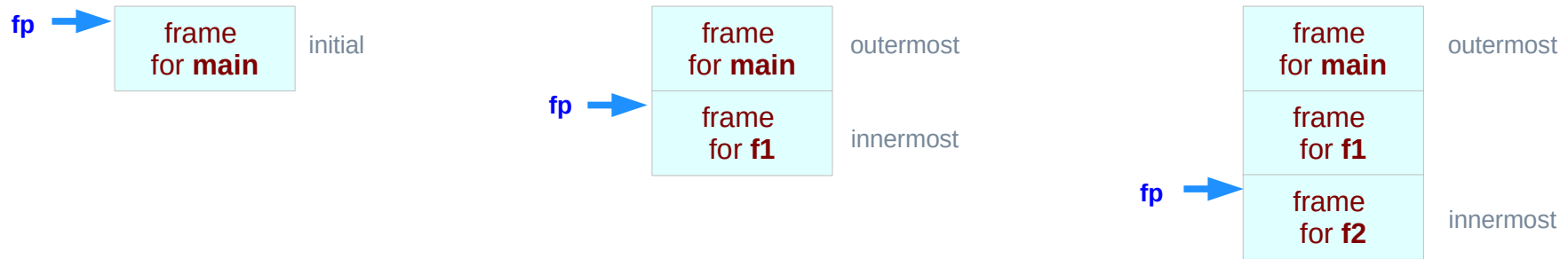
At least, **LR** must not be overwritten

save the followings in a **frame**

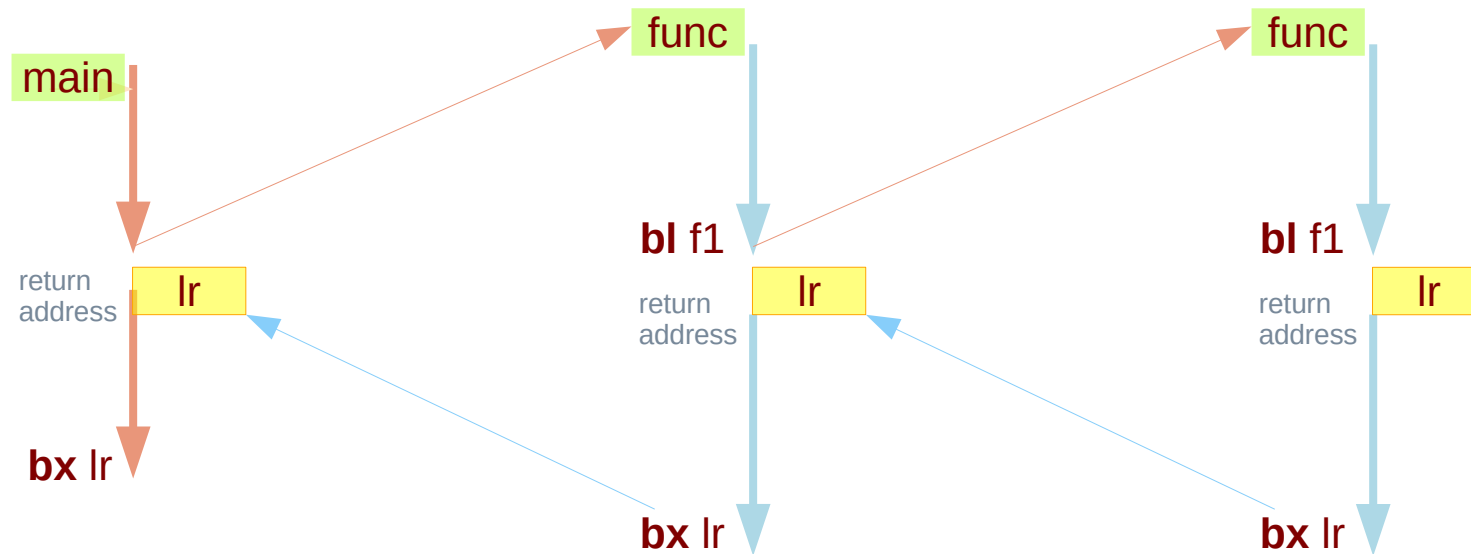
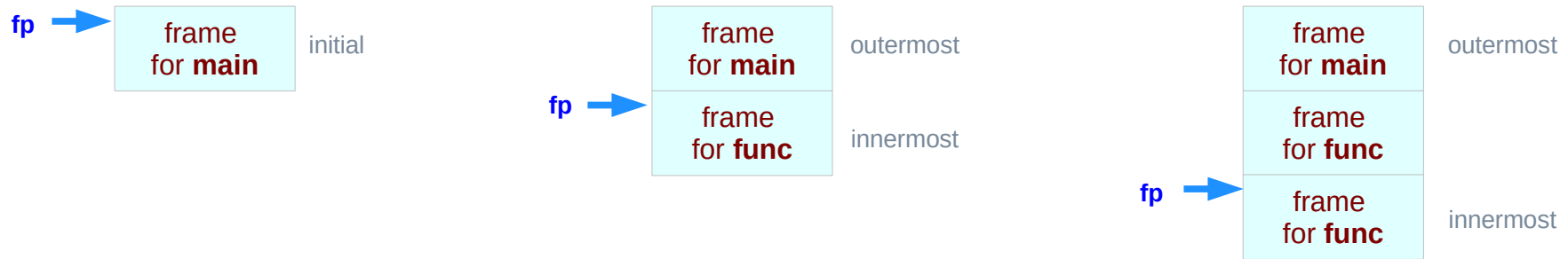
- return address
- arguments
- local variables.



Nested and recursive function calls



Recursive function calls



Stack frames (1)

local variables

- created upon entry to **function**.
- destroyed when function returns.

each **invocation** of a function has its own instantiation of **local variables**.

- recursive and nest calls to a function require several instantiations to exist simultaneously.
- functions return only after all functions it calls have returned last-in-first-out(**LIFO**) behavior.
- a **LIFO** structure called a **stack** is used to hold each instantiation.

the portion of the stack used for an **invocation** of a function is called the function's **stack frame** or **activation record**

<https://www.cs.princeton.edu/courses/archive/spring03/cs320/notes/7-1.pdf>

Stack frames (2)

a stack frame

a frame of data that gets pushed onto the stack.

a call stack

divided up into contiguous pieces called **stack frames** which represent a **function call** and its **argument** data.

- **return address**
- **arguments**
- **local variables.**

architecture-dependent.

processor knows the size of each frame and moves the **stack pointer** accordingly as **frames** are pushed and popped off the stack.

<https://stackoverflow.com/questions/10057443/explain-the-concept-of-a-stack-frame-in-a-nutshell>

Stack frames (3)

when your program is started,
the **call stack** has only one frame,
that of the function **main()**.
the **initial frame** or the **outermost frame**.

each time a function is called,
a new frame is added.
each time a function returns,
the frame for that function call is eliminated.

for a recursive function,
there can be many frames for the same function.

the frame for the currently executing function
is called the **innermost frame**.
the most recently created frame

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_StackFrames.html

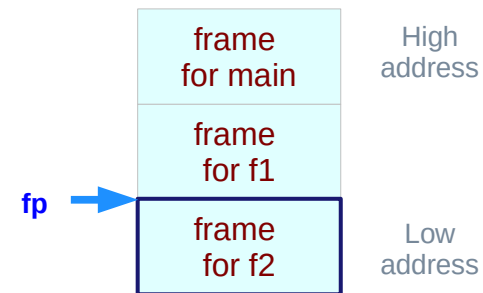
Stack frames (4)

A **stack frame** consists of many bytes

stack frames are identified by their addresses.

the address of the frame depends on architectures

Usually this address is kept in a register called the **frame pointer register fp** while execution is going on in that frame.

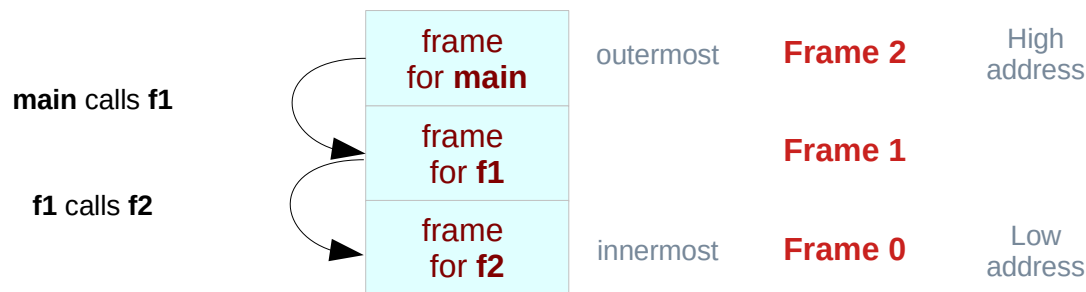


http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_StackFrames.html

Stack frames (5)

GDB assigns numbers to all existing stack frames, starting with **0** for the **innermost** frame, **1** for the frame that called it, and so on upward.

These numbers don't really exist in your program; they're assigned by GDB to give you a way of designating stack frames in GDB commands.



http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_stackframes.html

Stack frames (6)

a call stack

a stack data structure that stores information about the **active subroutines** of a computer program.

Although maintenance of the **call stack** is important for the proper functioning of most software, the details are normally **hidden** and **automatic** in high-level programming languages.

Many computer instruction sets provide **special instructions** for manipulating stacks.

also known as an

- **execution stack**
- **program stack**
- **control stack**
- **run-time stack**
- **machine stack**

https://en.wikipedia.org/wiki/Call_stack

Stack frames (7)

A **call stack** is used for several related purposes, but the main reason for having one is to keep track of the point to which each **active subroutine** should return control when it finishes executing.

An **active subroutine** is one that has been called, but is yet to complete execution, after which control should be handed back to the point of call.

Such **activations** of subroutines may be nested to any level (recursive as a special case), hence the **stack structure**.

https://en.wikipedia.org/wiki/Call_stack

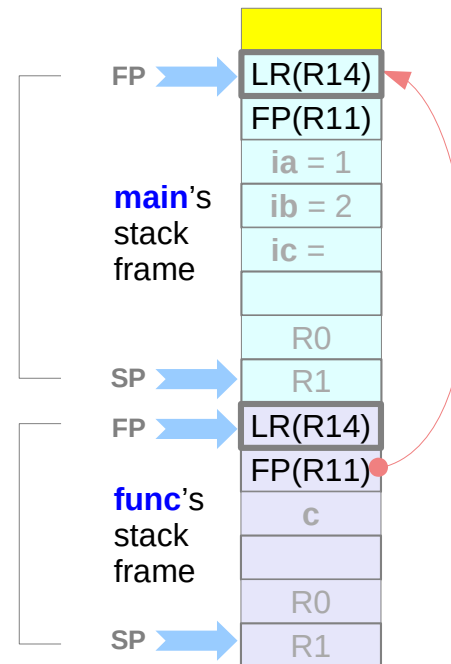
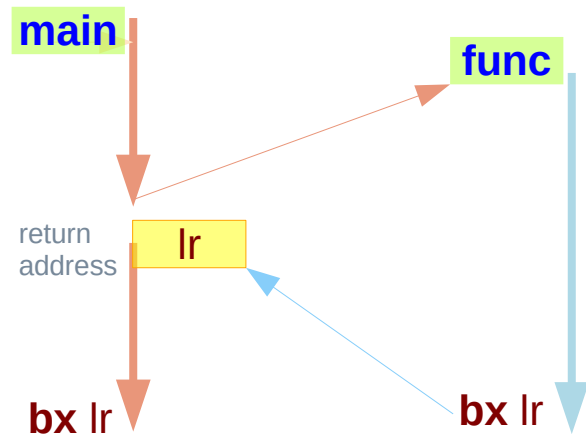
Register Use Convention

High Address	↑	R15	PC	Program counter
		R14	LR	Link address / scratch register
		R13	SP	Lower end of current stack frame
		R12	IP	Scratch register / specialist use by linker
Low Address		R11	FP	Frame Pointer

Frame pointer and stack pointer registers

LR (R14, link register,)
PC (R15, program counter)
FP (R11, frame pointer)
SP (R13, stack pointer)

where you were
where you are
where the stack bottom was
where the stack top is



<https://stackoverflow.com/questions/15752188/arm-link-register-and-frame-pointer>

Stack frames in intel processors

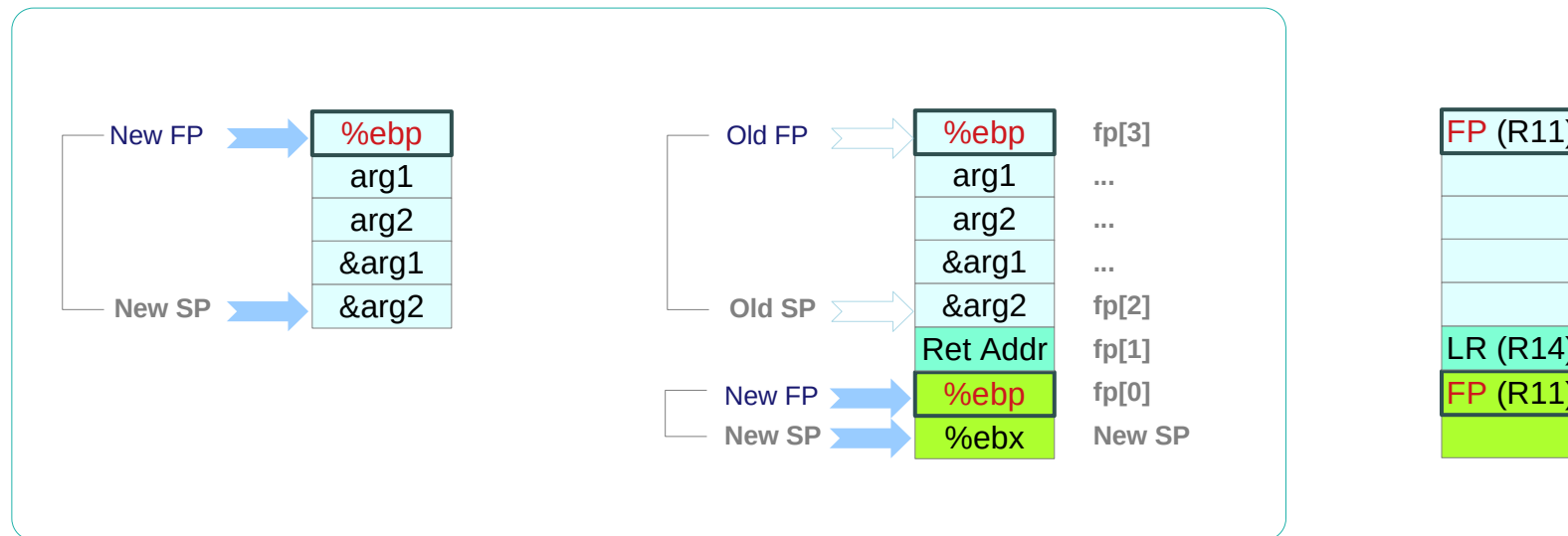
The basic frame layout is,

fp[0] saved **FP**, where we stored this frame.

fp[1] saved **LR**, the return address for this function.

fp[2] previous **SP**, before this function eats stack.

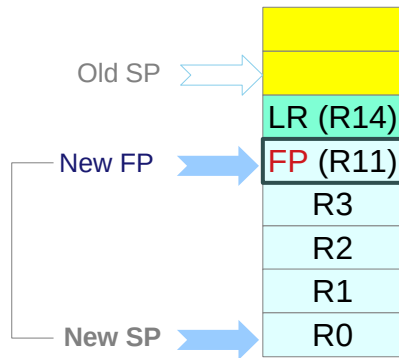
fp[3] previous **FP**, the last stack frame, many optional registers...



<https://stackoverflow.com/questions/15752188/arm-link-register-and-frame-pointer>

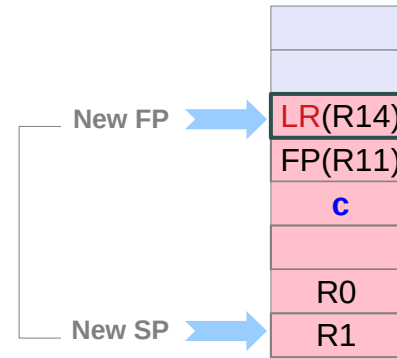
Frame pointers in a stack frame

The frame starts from the saved FP



Intel style stack frame

Sometimes, the frame starts from the saved LR

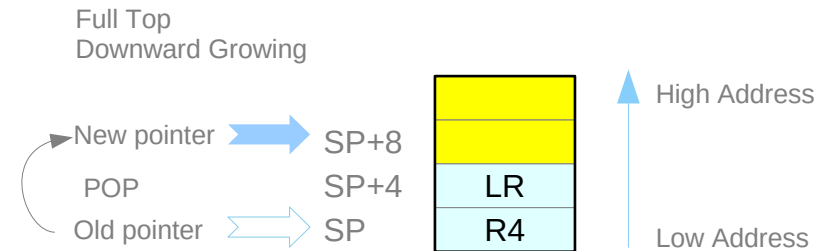
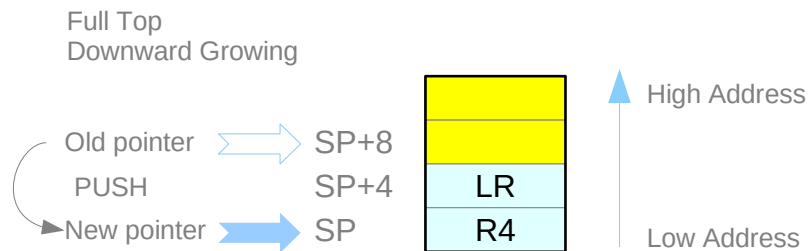


<https://stackoverflow.com/questions/15752188/arm-link-register-and-frame-pointer>

Stack frame skeleton (1)

```
function:                ; keep callee-saved registers

    push {r4, lr}        ; keep the callee saved registers
    ...                  ; code of the function
    pop {r4, lr}         ; restore the callee saved registers
    bx lr                ; return from the function
```



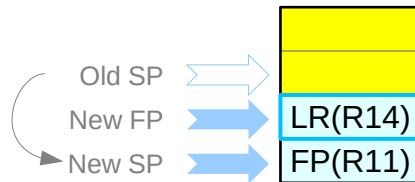
<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Stack frame skeleton (2)

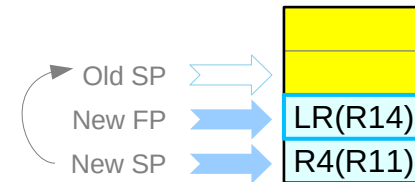
```
function:                                ; keep callee-saved registers

push {fp, lr}
add  fp, sp, #4
...                                     ; code of the function
sub  sp, r11, #4
pop  {fp, pc}
bx  lr                                  ; return from the function
```

Full Top
Downward Growing



Full Top
Downward Growing



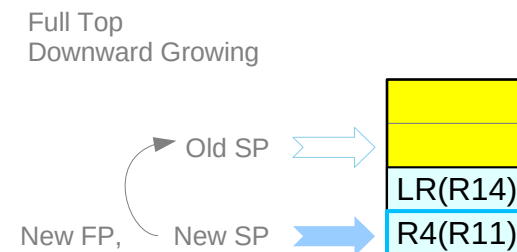
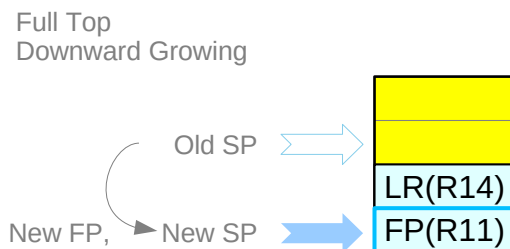
<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Stack frame skeleton (3)

```
function:                                ; keep callee-saved registers

push {fp, lr}                             ; keep fp and all callee-saved registers.
mov fp, sp                                 ; set the dynamic link
...                                         ; code of the function
mov sp, fp                                 ; Undo the dynamic link
pop {fp, lr}                               ; Restore fp and callee-saved registers

bx lr                                       ; return from the function
```



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

Stack frame skeleton (4)

function:

```
push {r4, r5, fp, lr}
mov fp, sp
```

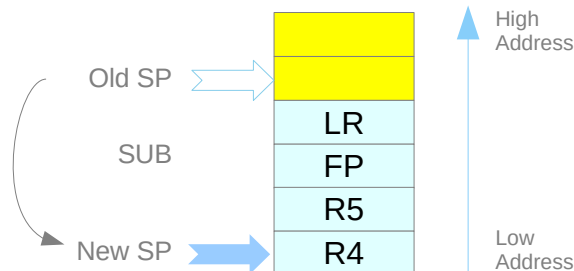
...

```
mov sp, fp
pop {r4, r5, fp, lr}
```

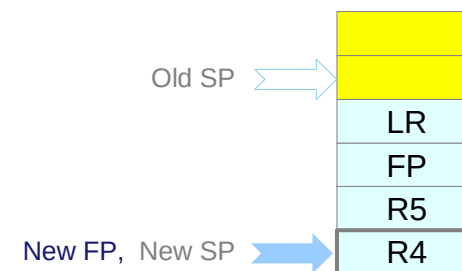
bx lr

```
; keep callee-saved registers
; keep the callee saved registers.
; we added r5 to keep the stack 8-byte aligned
; but the important thing here is fp
; fp ← sp. Keep dynamic link in fp
; code of the function
; sp ← fp. Restore dynamic link in fp
; restore the callee saved registers.
; this will restore fp as well
; return from the function
```

Full Top
Downward Growing



Full Top
Downward Growing



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

Stack frame skeleton (5)

function:

```
push{r4, r5, fp, lr}  
mov fp, sp
```

```
sub sp, sp, #8  
...
```

```
mov sp, fp  
pop {r4, r5, fp, lr}
```

```
bx lr
```

```
; keep callee-saved registers  
; keep the callee saved registers.  
; w added r5 to keep the stack 8-byte aligned  
; but the important thing here is fp  
; fp ← sp. Keep dynamic link in fp
```

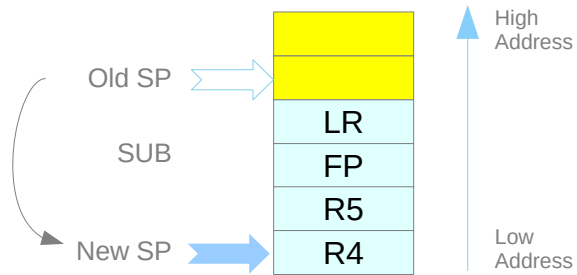
```
; enlarge the stack by 8 bytes  
; code of the function
```

```
; sp ← fp. restore dynamic link in fp  
; restore the callee saved registers.  
; this will restore fp as well  
; return from the function
```

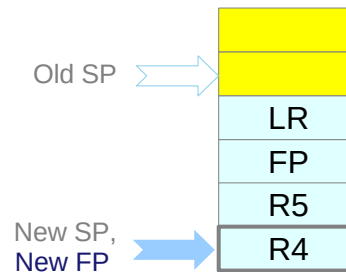
<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Stack frame skeleton (6)

1. push {r4, r5, fp, lr}

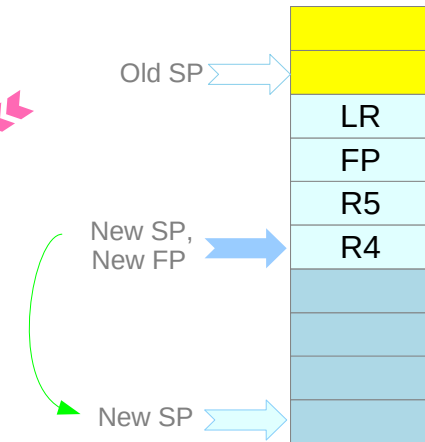


2. mov fp, sp

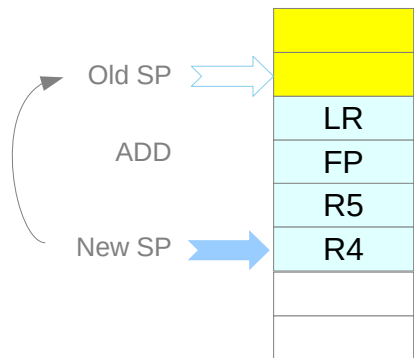


function code

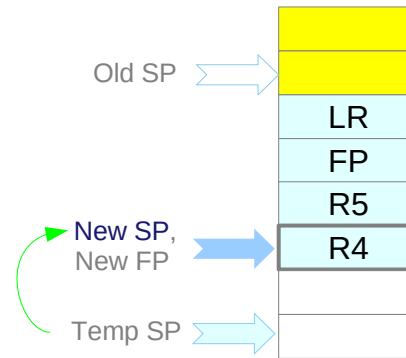
3. sub sp, sp, #8



5. pop {r4, r5, fp, lr}



4. mov sp, fp



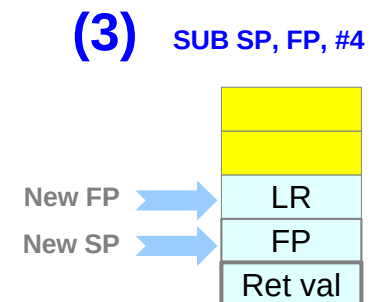
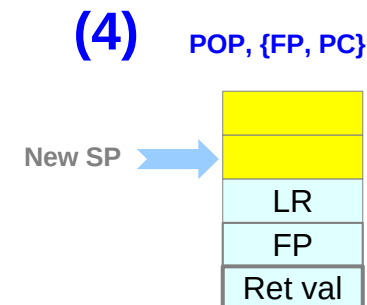
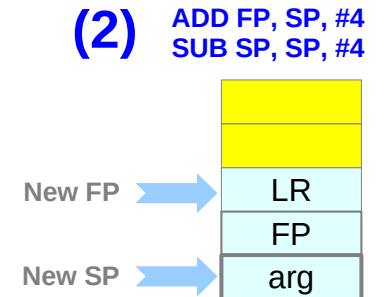
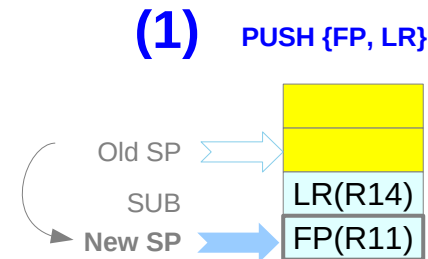
<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

Stack frame example A

```
Int add(int a, int b) {  
    int c;  
    c = a + b;  
    some_func(a,b);  
    return c;  
}
```

```
0x00010414 <+0>  
0x00010418 <+4>  
0x0001041c <+8>  
0x00010420 <+12>  
0x00010424 <+16>  
0x00010428 <+20>  
0x0001042c <+24>  
0x00010430 <+28>  
0x00010434 <+32>
```

```
push    {fp, lr}  
add     fp, sp, #4  
sub     sp, sp, #4  
add     r3, r0, r1  
str     r3, [fp-#8]  
bl    some_func  
str     r0, [fp-#8]  
sub     sp, fp, #4  
pop     {fp, pc}
```



<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame example B

```
int one(int, int);  
int two(int, int);  
int three(int, int);
```

```
Int main(void)  
{  
    int ia, ib, ic;  
  
    ia = 1;  
    ib = 2;  
    ic = one(ia, ib);  
  
    return ic;  
}
```

```
Int one(int a, int b)  
{  
    int c;  
    c = two(a,b);  
    return c;  
}
```

```
Int two(int a, int b)  
{  
    int c;  
    c = three(a,b);  
    return c;  
}
```

```
Int three(int a, int b)  
{  
    int c;  
    c = a+b;  
    return c;  
}
```

<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame for main

```

push  {r11, lr}
add   r11, sp, #4
sub   sp, sp, #24
str   r0, [r11, #-24]
str   r1, [r11, #-28]
mov   r3, #1
str   r3, [r11, #-8]
mov   r3, #2
str   r3, [r11, #-12]
ldr   r1, [r11, #-12]
ldr   r0, [r11, #-8]
bl  <one>
str   r0, [r11, #-16]
ldr   r3, [r11, #-16]
mov   r0, r3
sub   sp, r11, #4
pop   {r11, pc}
    
```

Received arg0
Received arg1

Local var ia

Local var ib

Arg0 for one

Arg1 for one

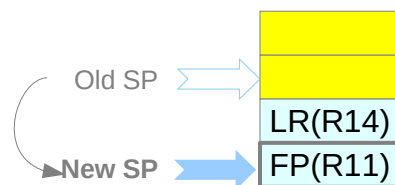
Local var ic

Return value

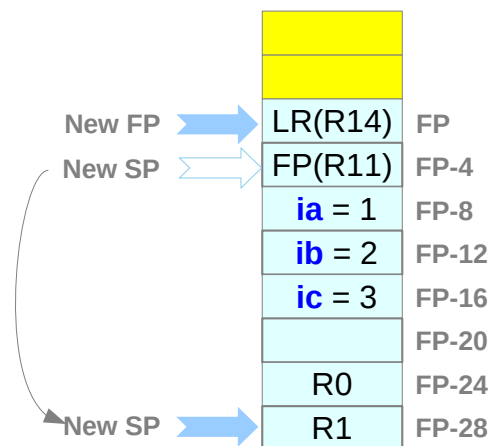
```

int main(void) {
    int ia, ib, ic;
    ia = 1; ib = 2;
    ic = one(ia, ib);
    return ic;
}
    
```

(1) PUSH {FP, LR}



(2) ADD FP, SP, #4
SUB SP, SP, #24

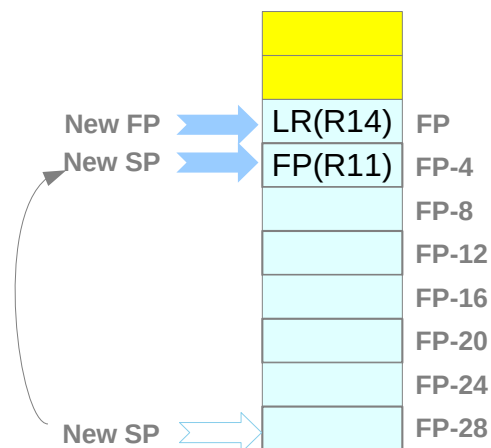


Reg
R0
R1
R3

(4) POP, {FP, PC}



(3) SUB SP, FP, #4

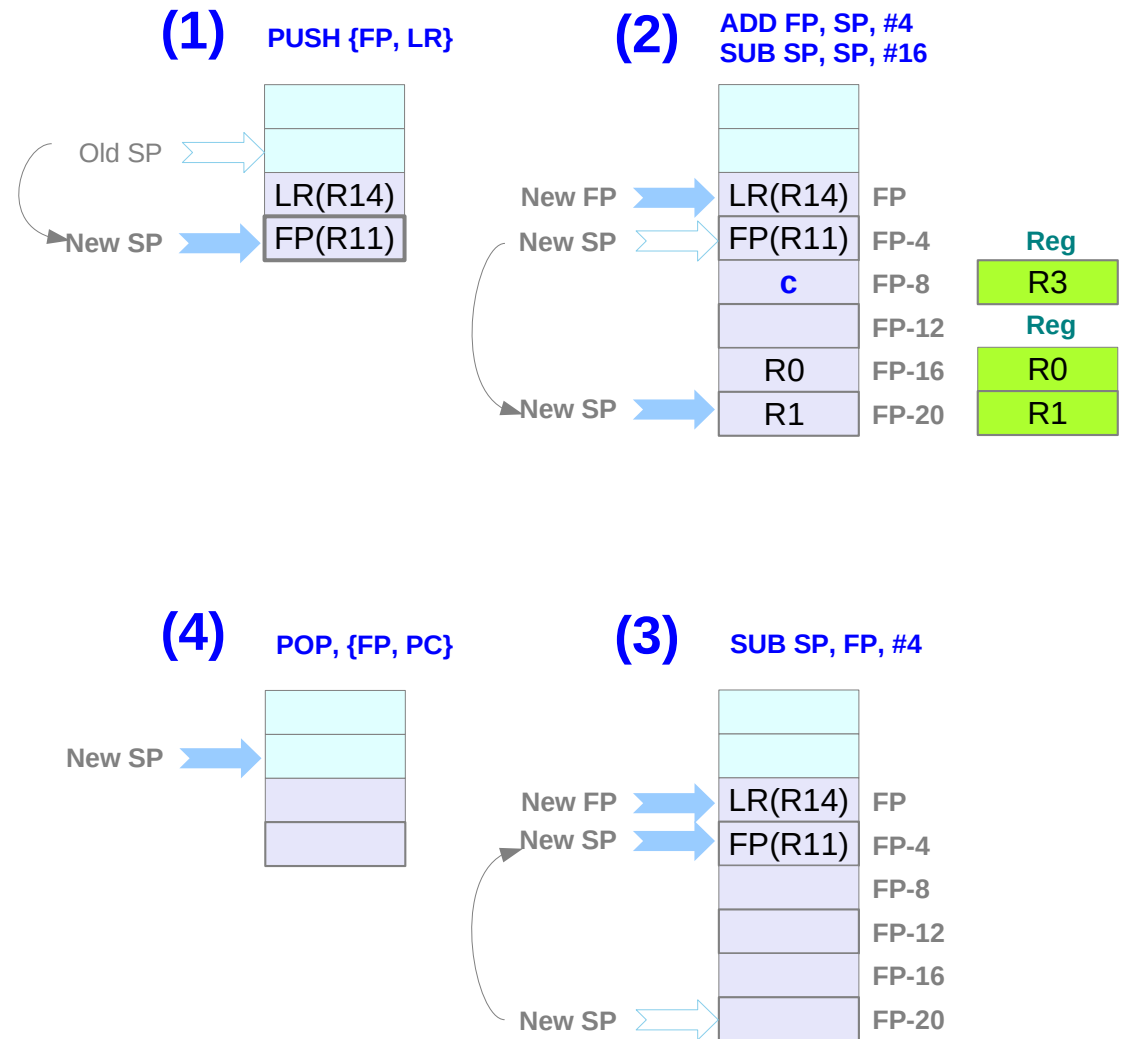


<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame for one

push	{r11, lr}	
add	r11, sp, #4	
sub	sp, sp, #16	
str	r0, [r11, #-16]	Received arg0
str	r1, [r11, #-20]	Received arg1
ldr	r1, [r11, #-20]	Arg0 for two
ldr	r0, [r11, #-16]	Arg1 for two
bl	<two>	
str	r0, [r11, #-8]	Local var c
ldr	r3, [r11, #-8]	
mov	r0, r3	Return value
sub	sp, r11, #4	
pop	{r11, pc}	

```
int one(int a, int b) {
    int c;
    c = two(a,b);
    return c;
}
```

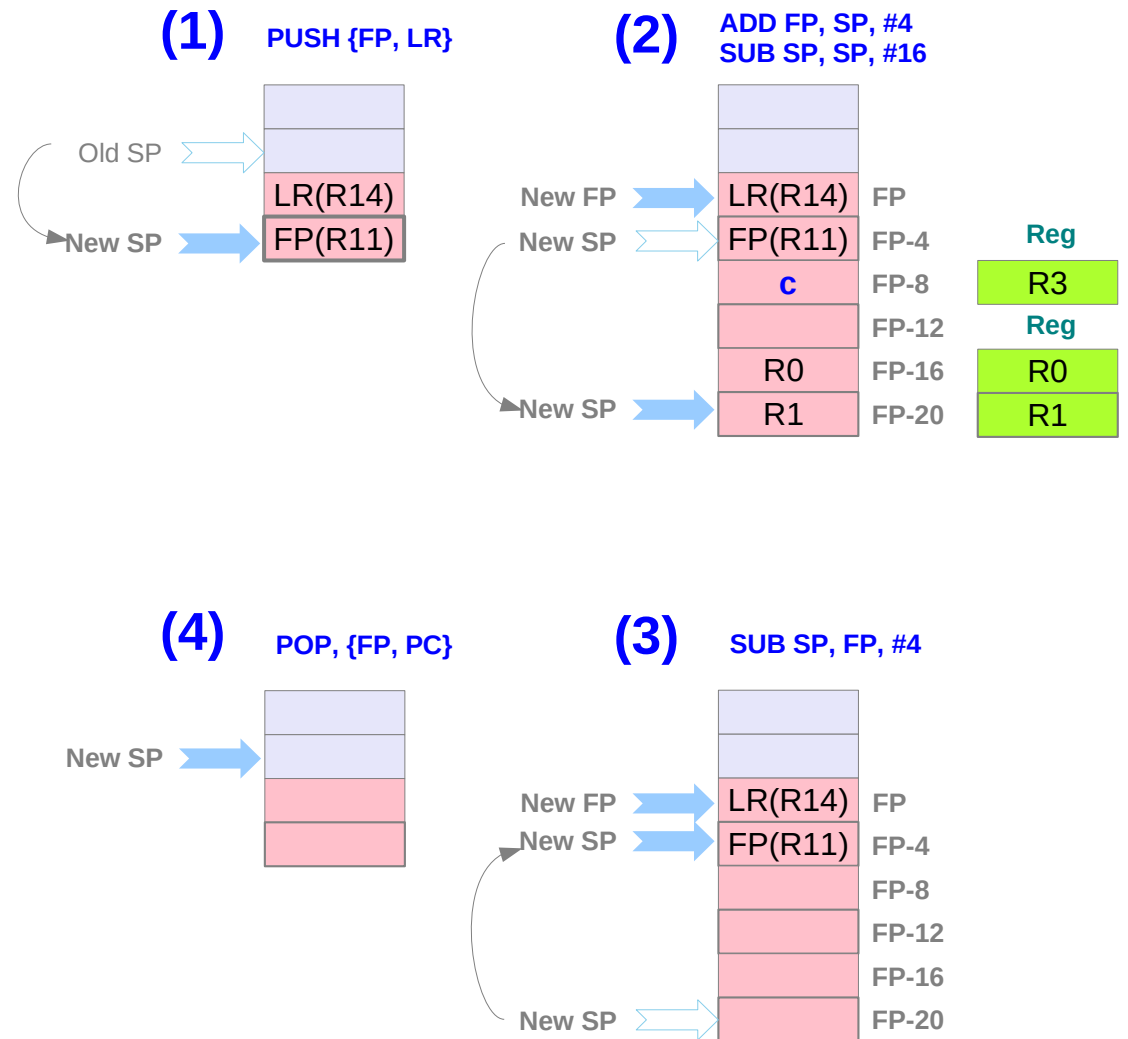


<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame for two

push	{r11, lr}	
add	r11, sp, #4	
sub	sp, sp, #16	
str	r0, [r11, #-16]	Received arg0
str	r1, [r11, #-20]	Received arg1
ldr	r1, [r11, #-20]	Arg0 for three
ldr	r0, [r11, #-16]	Arg1 for three
bl	<three>	
str	r0, [r11, #-8]	Local var c
ldr	r3, [r11, #-8]	
mov	r0, r3	Return value
sub	sp, r11, #4	
pop	{r11, pc}	

```
int two(int a, int b) {
    int c;
    c = three(a,b);
    return c;
}
```



<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame for three

```

push    {r11}
add     r11, sp, #0
sub     sp, sp, #20
str     r0, [r11, #-16]
str     r1, [r11, #-20]
ldr     r2, [r11, #-16]
ldr     r3, [r11, #-20]
add     r3, r2, r3
str     r3, [r11, #-8]
ldr     r3, [r11, #-8]
mov     r0, r3
add     sp, r11, #0
pop     {r11}

```

bx lr

```

int three(int a, int b) {
    int c;
    c = a+b;
    return c;
}

```

Received arg0

Received arg1

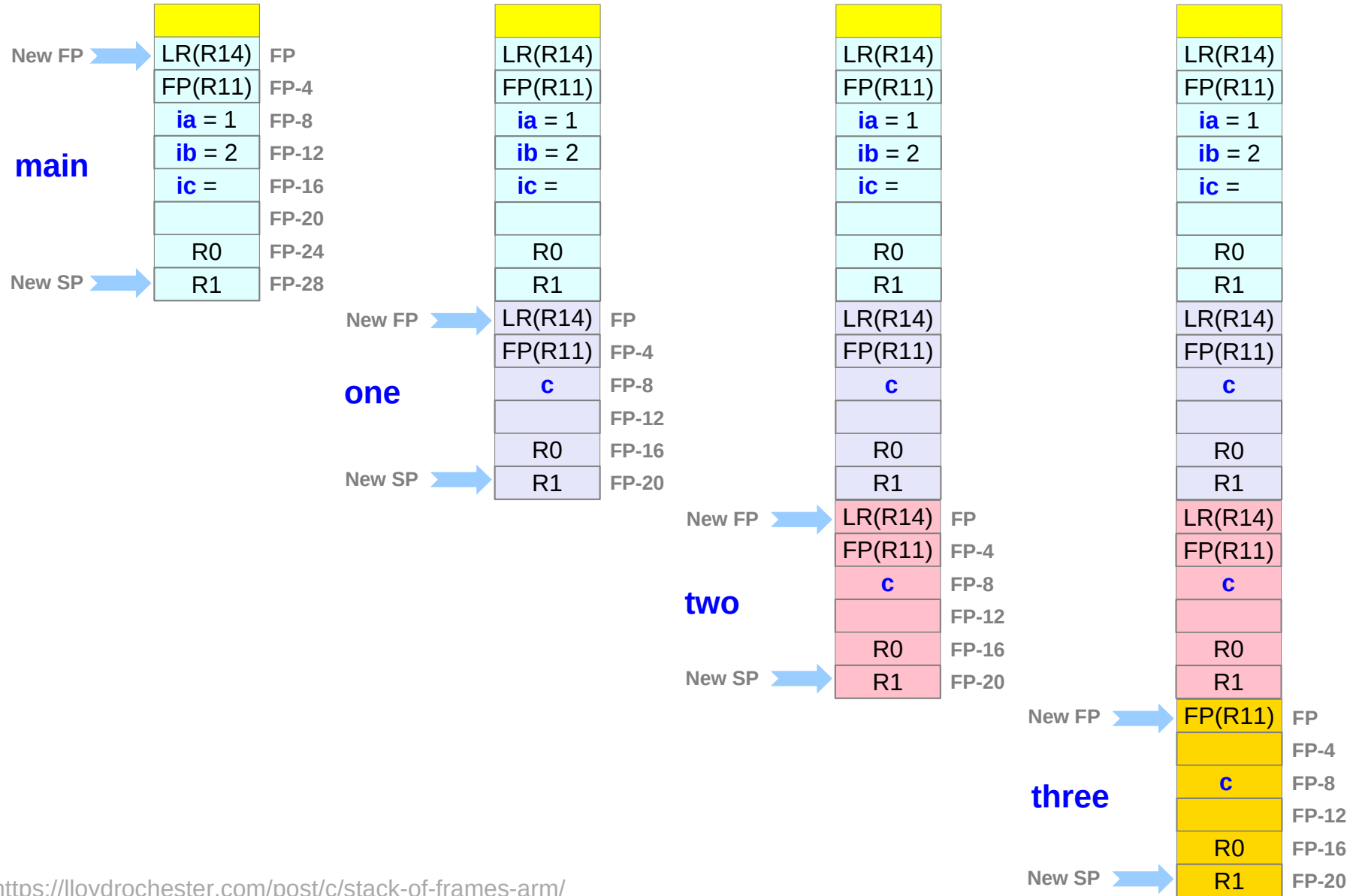
Local var c

Return value



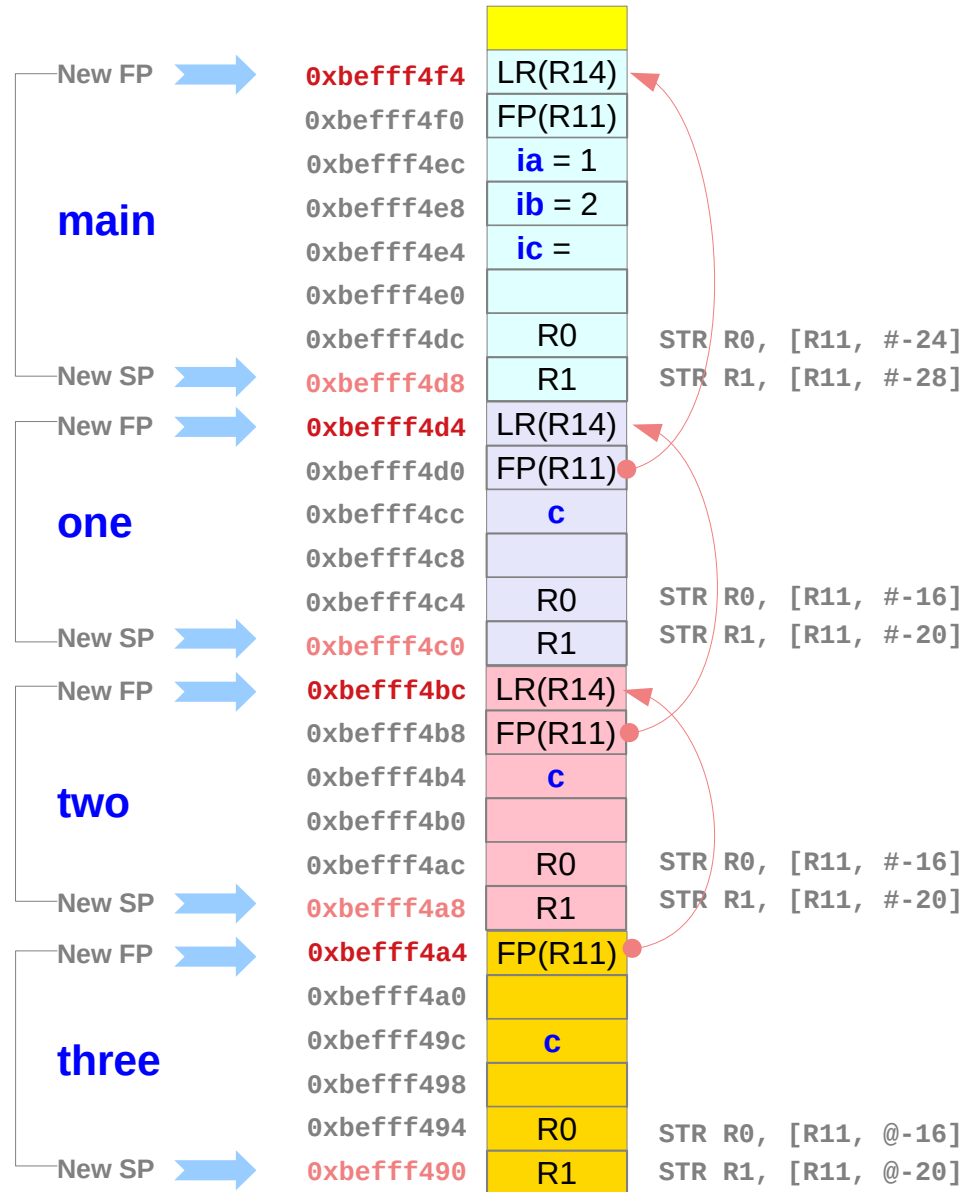
<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame snapshots



<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Stack frame memory map



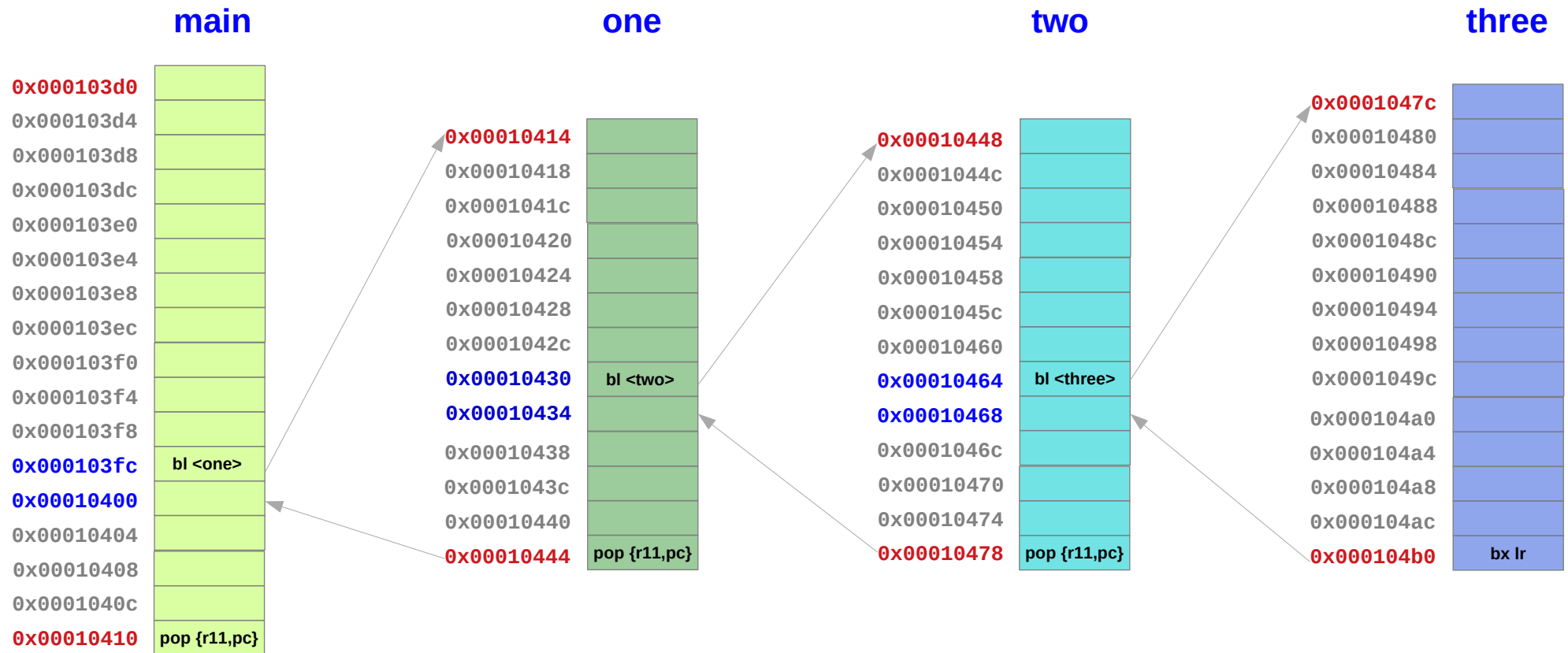
<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Text area memory map



<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Nested procedure calls



<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Disassembly of main

(gdb) disassemble **main**

Dump of assembler code for function main:

0x000103d0 <+0>:	push	{r11, lr}	; lr= 0xbfe84718 r11 at lowest address
0x000103d4 <+4>:	add	r11, sp, #4	; r11=fp= 0xbefff4f4
0x000103d8 <+8>:	sub	sp, sp, #24	; sp= 0xbefff4d8 , frame size 28=24+4
0x000103dc <+12>:	str	r0, [r11, #-24]	; 0xbefff4dc
0x000103e0 <+16>:	str	r1, [r11, #-28]	; 0xbefff4d8
0x000103e4 <+20>:	mov	r3, #1	
0x000103e8 <+24>:	str	r3, [r11, #-8]	
0x000103ec <+28>:	mov	r3, #2	
0x000103f0 <+32>:	str	r3, [r11, #-12]	
0x000103f4 <+36>:	ldr	r1, [r11, #-12]	
0x000103f8 <+40>:	ldr	r0, [r11, #-8]	
0x000103fc <+44>:	bl	0x10414 <one>	; here the lr will be set to 0X00010400
0X00010400 <+48>:	str	r0, [r11, #-16]	; r0 has the return value from function one
0x00010404 <+52>:	ldr	r3, [r11, #-16]	
0x00010408 <+56>:	mov	r0, r3	; r0 will return with the value of int ic
0x0001040c <+60>:	sub	sp, r11, #4	; point sp one word above fp
0x00010410 <+64>:	pop	{r11, pc }	; pc will be restored to 0xbfe84718

End of assembler dump.

<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Disassembly of one

(gdb) disassemble **one**

Dump of assembler code for function one:

0x00010414 <+0>:	push	{r11, lr}	; lr= 0x00010400 r11=fp= 0xbefff4f4
0x00010418 <+4>:	add	r11, sp, #4	; r11=fp= 0xbefff4d4
0x0001041c <+8>:	sub	sp, sp, #16	; sp= 0xbefff4c0 frame is size 20=16+4
0x00010420 <+12>:	str	r0, [r11, #-16]	; 0xbefff4c4
0x00010424 <+16>:	str	r1, [r11, #-20]	; 0xbefff4c0
0x00010428 <+20>:	ldr	r1, [r11, #-20]	
0x0001042c <+24>:	ldr	r0, [r11, #-16]	
0x00010430 <+28>:	bl	0x10448 <two>	; lr will be 0x00010434
0x00010434 <+32>:	str	r0, [r11, #-8]	
0x00010438 <+36>:	ldr	r3, [r11, #-8]	
0x0001043c <+40>:	mov	r0, r3	
0x00010440 <+44>:	sub	sp, r11, #4	; point sp one word above fp
0x00010444 <+48>:	pop	{r11, pc }	; fp=0xbefff4f4, lr= 0x00010400

End of assembler dump.

<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Disassembly of **two**

(gdb) disassemble **two**

Dump of assembler code for function two:

0x00010448 <+0>:	push	{r11, lr}	; lr= 0x00010434 , r11=fp= 0xbefff4d4
0x0001044c <+4>:	add	r11, sp, #4	; fp= 0xbefff4bc
0x00010450 <+8>:	sub	sp, sp, #16	; sp= 0xbefff4a8 frame is 20=16+4 words
0x00010454 <+12>:	str	r0, [r11, #-16]	; 0xbefff4ac
0x00010458 <+16>:	str	r1, [r11, #-20]	; 0xbefff4a8
0x0001045c <+20>:	ldr	r1, [r11, #-20]	
0x00010460 <+24>:	ldr	r0, [r11, #-16]	
0x00010464 <+28>:	bl	0x1047c <three>	; lr will be set to 0x00010468
0x00010468 <+32>:	str	r0, [r11, #-8]	
0x0001046c <+36>:	ldr	r3, [r11, #-8]	
0x00010470 <+40>:	mov	r0, r3	
0x00010474 <+44>:	sub	sp, r11, #4	
0x00010478 <+48>:	pop	{r11, pc }	

End of assembler dump.

<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Disassembly of three

(gdb) disassemble **three**

Dump of assembler code for function three:

```
0x0001047c <+0>:  push    {r11}
0x00010480 <+4>:  add     r11, sp, #0
0x00010484 <+8>:  sub     sp, sp, #20
0x00010488 <+12>: str     r0, [r11, #-16]
0x0001048c <+16>: str     r1, [r11, #-20]
0x00010490 <+20>: ldr     r2, [r11, #-16]
0x00010494 <+24>: ldr     r3, [r11, #-20]
0x00010498 <+28>: add     r3, r2, r3
0x0001049c <+32>: str     r3, [r11, #-8]
0x000104a0 <+36>: ldr     r3, [r11, #-8]
0x000104a4 <+40>: mov     r0, r3
0x000104a8 <+44>: add     sp, r11, #0
0x000104ac <+48>: pop     {r11}
0x000104b0 <+52>:  bx     lr
```

End of assembler dump.

```
; (str r11, [sp, #-4]!) NOTICE no lr!!
; r11=fp=0xbefff4bc
; dont add #4 here since no frp=0xbefff4a4
; stack is size 20 sp=0xbefff490
; 0xbefff494
; 0xbefff490
; (ldr r11, [sp], #4)
; lr=0x00010468
```

<https://lloydrochester.com/post/c/stack-of-frames-arm/>

Local Data Generating Examples

```
void sq(int *c)
{
    (*c) = (*c) * (*c);
}
```

```
int sq_sum5(int a, int b, int c, int d, int e)
{
    sq(&a);
    sq(&b);
    sq(&c);
    sq(&d);
    sq(&e);
    return a + b + c + d + e;
}
```

```
...
sq_sum5(1, 2, 3, 4, 5);
...
```

callee
function

- **sq** received a reference
- registers do not have an address
- allocate temporary local storage

caller
function

Callee Function Code

sq_sum5:

push { fp, lr }

mov fp, sp

sub sp, sp, #16

str r0, [fp, #-16] *(fp - 16) ← r0

str r1, [fp, #-12] *(fp - 12) ← r1

str r2, [fp, #-8] *(fp - 8) ← r2

str r3, [fp, #-4] *(fp - 4) ← r3

mov sp, fp

pop { fp, lr }

bx lr

sq:

ldr r1, [r0] r1 ← (*r0)

mul r1, r1, r1 r1 ← r1 * r1

str r1, [r0] (*r0) ← r1

bx lr

sub r0, fp, #16 r0 ← fp - 16

bl sq call sq (&a)

sub r0, fp, #12 r0 ← fp - 12

bl sq call sq (&b)

sub r0, fp, #8 r0 ← fp - 8

bl sq call sq (&c)

sub r0, fp, #4 r0 ← fp - 4

bl sq call sq (&d)

add r0, fp, #8 r0 ← fp + 8

bl sq call sq (&e)

ldr r0, [fp, #-16] r0 ← *(fp - 16) :a

ldr r1, [fp, #-12] r1 ← *(fp - 12) :b

add r0, r0, r1 r0 ← r0 + r1

ldr r1, [fp, #-8] r1 ← *(fp - 8) :c

add r0, r0, r1 r0 ← r0 + r1

ldr r1, [fp, #-4] r1 ← *(fp - 4) :d

add r0, r0, r1 r0 ← r0 + r1

ldr r1, [fp, #8] r1 ← *(fp + 8) :e

add r0, r0, r1 r0 ← r0 + r1

Caller Function Code

```
.data
.align 4

message:
.asciz "Sum of 1^2 + 2^2 + 3^2 + 4^2 +
5^2 is %d\n"

.text

sq:      <<defined above>>
sq_sum5:<defined above>>

.globl main
main:

push { r4, lr }

pop { r4, lr }

bx lr
```

```
mov r0, #1    a ← 1
mov r1, #2    b ← 2
mov r2, #3    c ← 3
mov r3, #4    d ← 4

mov r4, #5    r4 ← 5

sub sp, sp, #8
str r4, [sp]  e ← 5

bl sq_sum5 sq_sum5 ( 1, 2, 3, 4, 5 )

add sp, sp, #8

mov r1, r0
ldr r0, address_of_message

bl printf

address_of_message: .word message
```

main (1)

```
/* squares.s */  
.data  
  
.align 4  
message: .asciz "Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is %d\n"  
  
.text  
  
sq:  
    <<defined above>>  
  
sq_sum5:  
    <<defined above>>  
  
.globl main
```

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

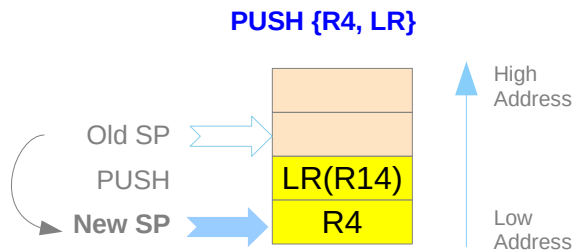
main (2)

main:

```
push    {r4, lr}           ; Keep callee-saved registers
```

```
; Prepare the call to sq_sum5
```

```
mov     r0, #1             ; Parameter r0 ← a=1  
mov     r1, #2             ; Parameter r1 ← b=2  
mov     r2, #3             ; Parameter r2 ← c=3  
mov     r3, #4             ; Parameter r3 ← d=4  
mov     r4, #5             ; Parameter r4 ← e=5
```

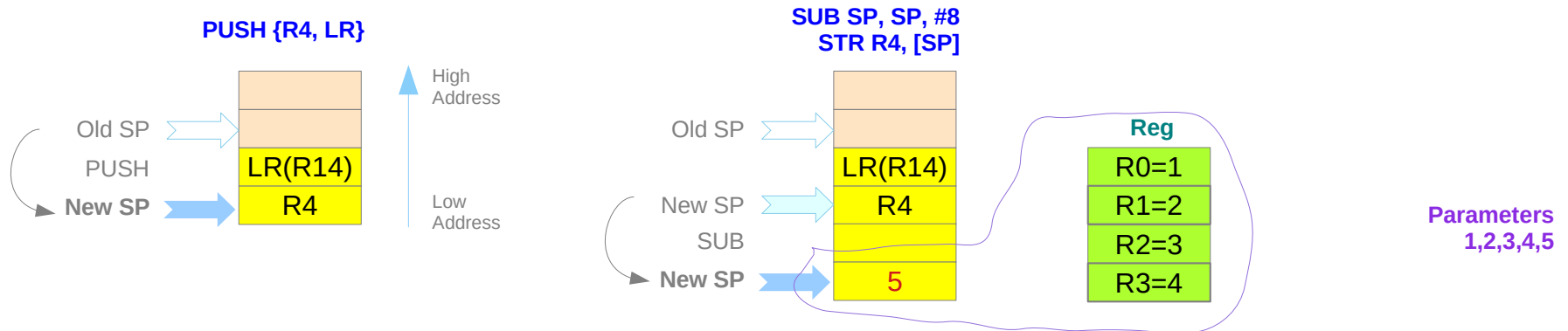


<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

main (3)

; Parameter e goes through the stack, ; Parameter r4 ← e=5
; so it requires enlarging the stack

```
sub    sp, sp, #8    ; Enlarge the stack 8 bytes,  
; we will use only the topmost 4 bytes  
str    r4, [sp]      ; push the parameter e = 5  
bl     sq_sum5       ; call sq_sum5 (1, 2, 3, 4, 5)
```



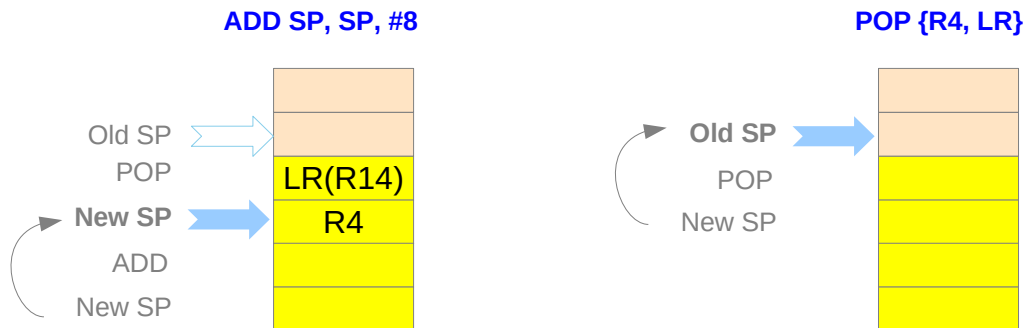
<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

main (4)

```
add    sp, sp, #8           ; Shrink back the stack

; Prepare the call to printf
mov    r1, r0               ; The result of sq_sum5       ; 2nd arg to printf
ldr    r0, addr_of_msg     ; the address of the message ; 1st arg to printf
bl     printf              ; Call printf

pop    {r4, lr}            ; Restore callee-saved registers
bx     lr
```



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

main (5)

```
addr_of_msg: .word msg
```

```
msg: .asciz "Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is %d\n"
```

```
$/square
```

```
Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is 55
```

addr_of_msg

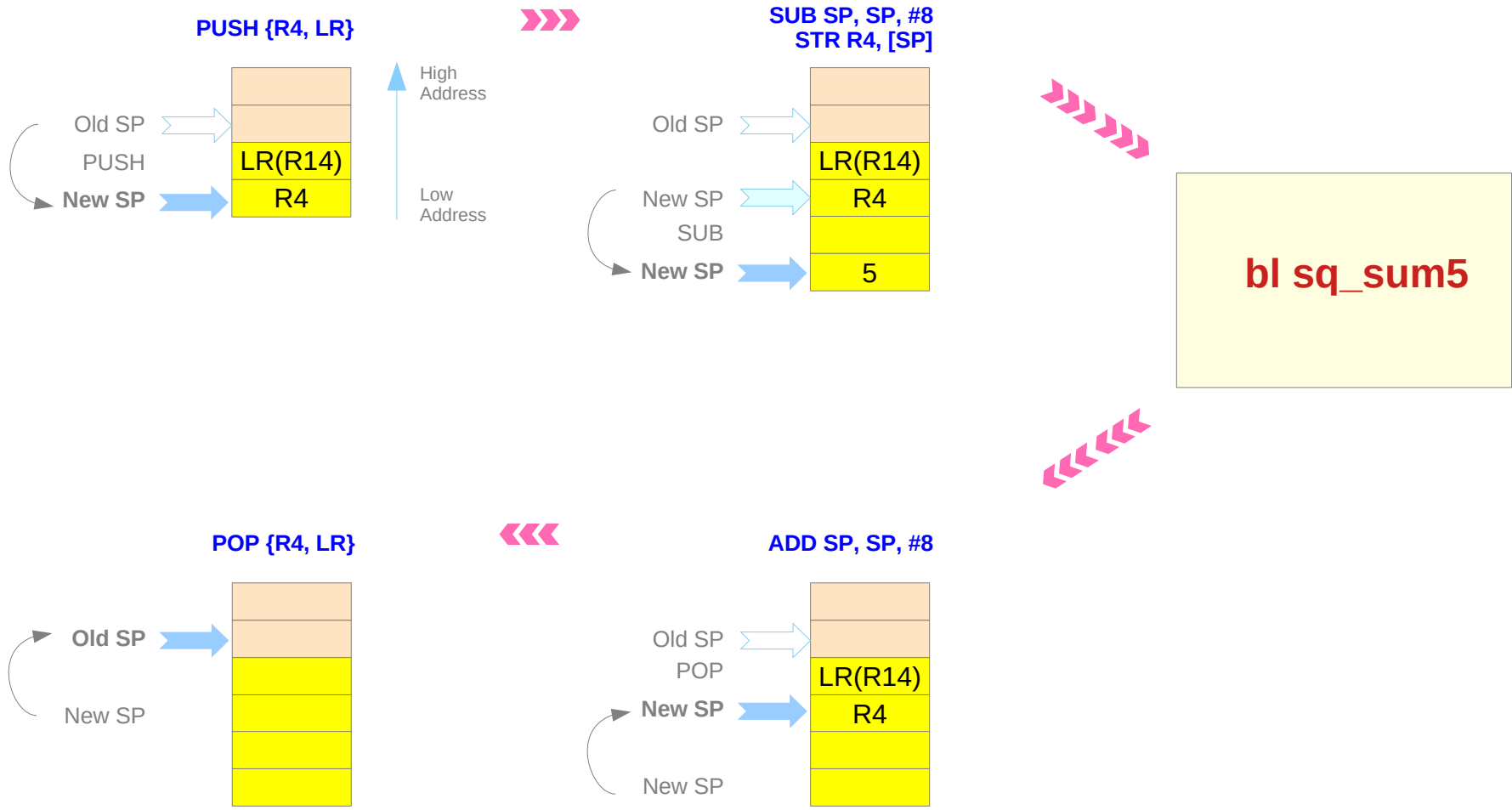
msg

msg

S	u	m	
o	f		1
^	2		+
	2	^	2
	+		3
^	2		+
	4	^	2
	+		5
^	2		i
s		%	d
\n			

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

main's stack frame



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

sq_sum5 (1)

```
int sq_sum5(int a, int b, int c, int d, int e) {  
    sq(&a);  
    sq(&b);  
    sq(&c);  
    sq(&d);  
    sq(&e);  
    return a + b + c + d + e;  
}
```

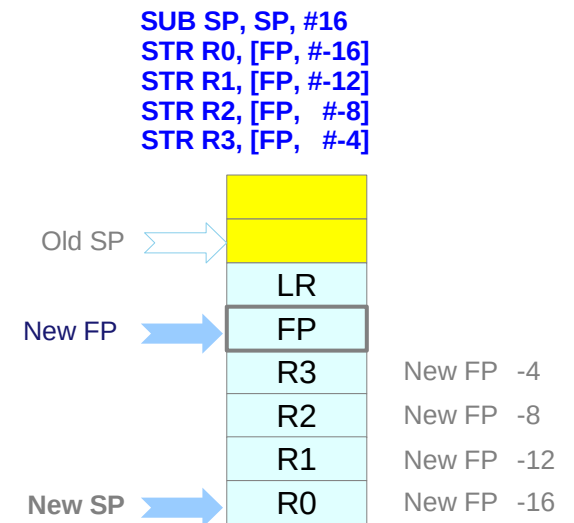
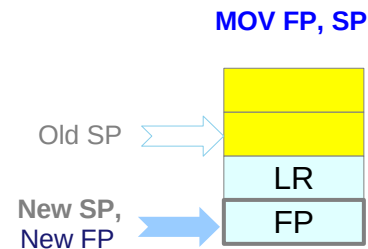
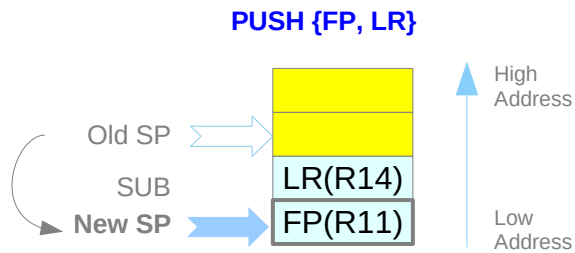
<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

sq_sum5 (2)

sq_sum5:

```
push    {fp, lr}      ; keep fp and all callee-saved registers.
mov     fp, sp        ; set the dynamic link
                           ; allocate space for 4 integers in the stack
                           ; keep parameters in the stack

sub     sp, sp, #16   ; sp ← sp - 16.
str     r0, [fp, #-16] ; *(fp - 16) ← r0
str     r1, [fp, #-12] ; *(fp - 12) ← r1
str     r2, [fp, #- 8] ; *(fp - 8) ← r2
str     r3, [fp, #- 4] ; *(fp - 4) ← r3
```



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

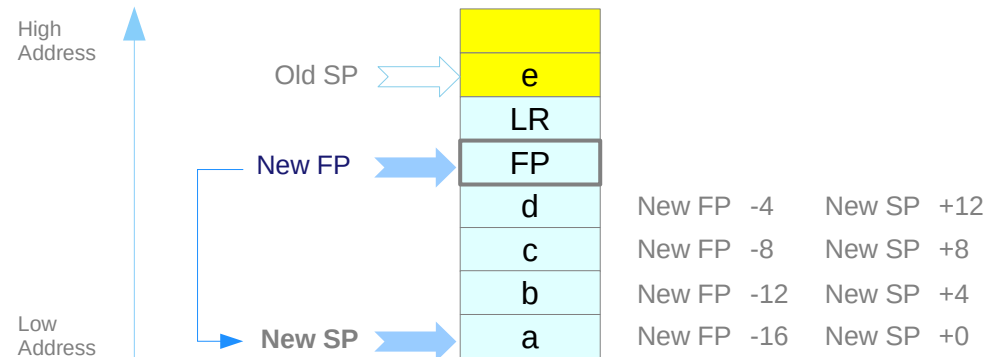
sq_sum5 (3)

Value	Address(es)	
a	[fp, #-16]	[sp]
b	[fp, #-12]	[sp, #4]
c	[fp, #-8]	[sp, #8]
d	[fp, #-4]	[sp, #12]
fp(r11)	[fp]	[sp, #16]
lr(r14)	[fp, #4]	[sp, #20]
e	[fp, #8]	[sp, #24]

High Address

↓

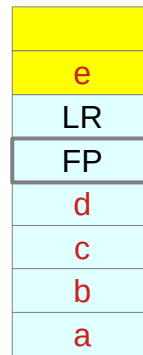
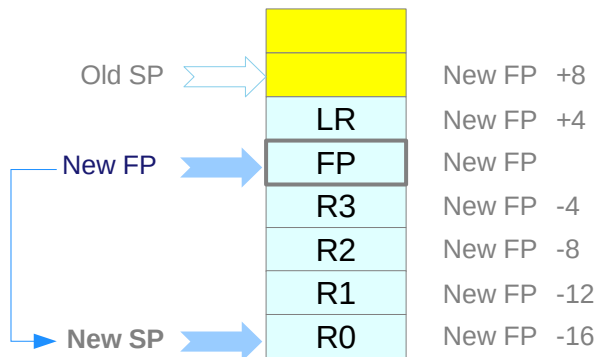
fp[-0] saved pc
fp[-1] saved lr
fp[-2] previous sp
fp[-3] previous fp



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

sq_sum5 (4)

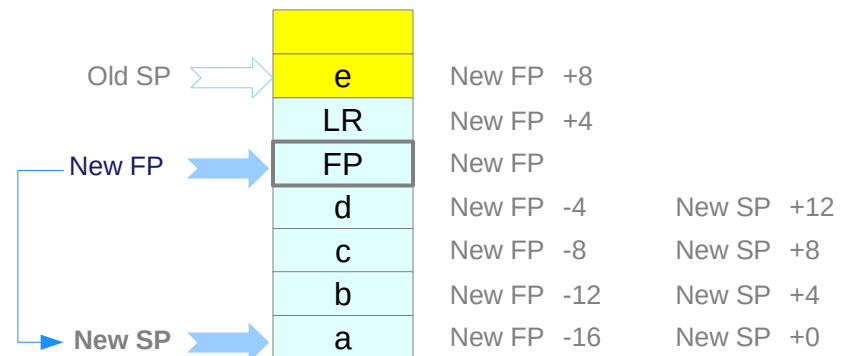
```
sub    r0, fp, #16    ; r0 ← fp - 16    ; address of a on the stack ; 1st parameter
bl   sq            ; call sq(&a);
sub    r0, fp, #12    ; r0 ← fp - 12    ; address of b on the stack ; 1st parameter
bl   sq            ; call sq(&b);
sub    r0, fp, #8     ; r0 ← fp - 8     ; address of c on the stack ; 1st parameter
bl   sq            ; call sq(&c);
sub    r0, fp, #4     ; r0 ← fp - 4     ; address of d on the stack ; 1st parameter
bl   sq            ; call sq(&d);
add    r0, fp, #8     ; r0 ← fp + 8     ; address of e on the stack ; 1st parameter
bl   sq            ; call sq(&e);
```



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

sq_sum5 (5)

```
ldr    r0, [fp, #-16]    ; r0 ← *(fp - 16). ; Loads a into r0
ldr    r1, [fp, #-12]    ; r1 ← *(fp - 12). ; Loads b into r1
add    r0, r0, r1        ; r0 ← r0 + r1      ; (a) + b
ldr    r1, [fp, #-8]     ; r1 ← *(fp - 8).  ; Loads c into r1
add    r0, r0, r1        ; r0 ← r0 + r1      ; (a + b) + c
ldr    r1, [fp, #-4]     ; r1 ← *(fp - 4).   ; Loads d into r1
add    r0, r0, r1        ; r0 ← r0 + r1      ; (a + b + c) + d
ldr    r1, [fp, #8]      ; r1 ← *(fp + 8).   ; Loads e into r1
add    r0, r0, r1        ; r0 ← r0 + r1      ; (a + b + c + d) + e
```

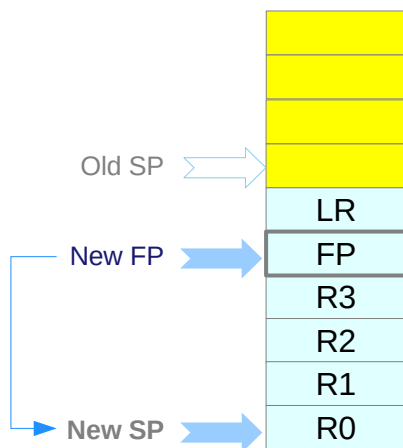


<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

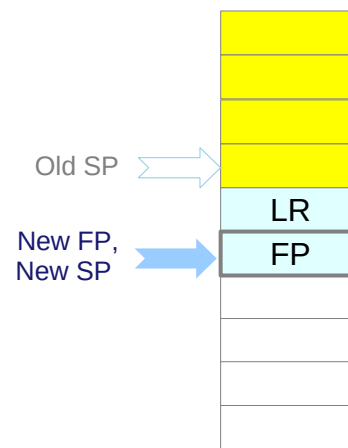
sq_sum5 (6)

```
mov    sp, fp    ; Undo the dynamic link
pop    {fp, lr}  ; Restore fp and callee-saved registers
bx     lr        ; Return from the function
```

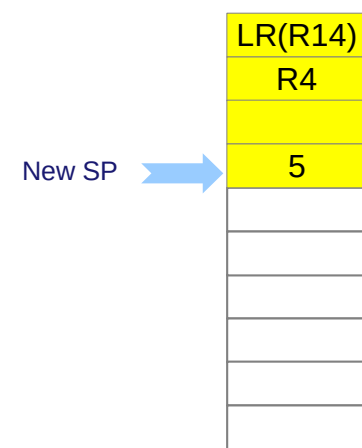
sq_sum5 frame



MOV SP, FP



POP {FP, LR}



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

sq

```
void sq(int *c) {  
    (*c) = (*c) * (*c);  
}
```

```
sq:  
ldr    r1, [r0]    ; r1 ← (*r0)    ; r0 : argument register  
mul    r1, r1, r1  ; r1 ← r1 * r1  
str    r1, [r0]    ; (*r0) ← r1  
bx     lr          ; return from the function
```

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

sq_sum5's stack frame (1)

sq_sum5:

```

push {fp, lr}
mov fp, sp
sub sp, sp, #16
str r0, [fp, #-16]
str r1, [fp, #-12]
str r2, [fp, #-8]
str r3, [fp, #-4]

```

```

sub r0, fp, #16
bl sq
sub r0, fp, #12
bl sq
sub r0, fp, #8
bl sq
sub r0, fp, #4
bl sq
add r0, fp, #8
bl sq

```

```

ldr r0, [fp, #-16]
ldr r1, [fp, #-12]
add r0, r0, r1
ldr r1, [fp, #-8]
add r0, r0, r1
ldr r1, [fp, #-4]
add r0, r0, r1
ldr r1, [fp, #8]
add r0, r0, r1

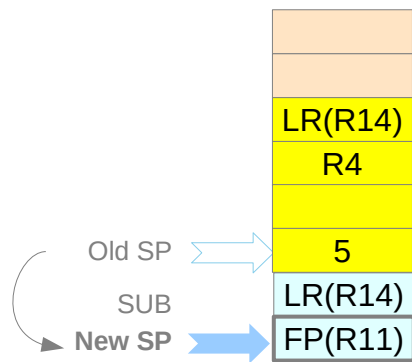
```

```

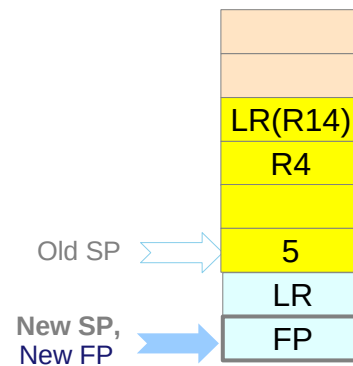
mov sp, fp
pop {fp, lr}
bx lr

```

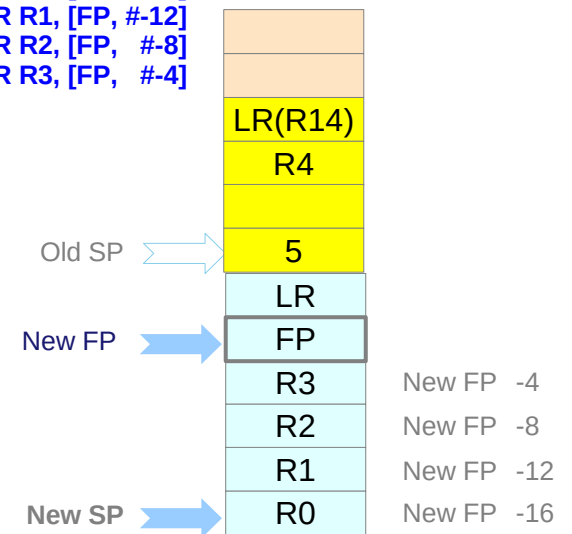
PUSH {FP, LR}



MOV FP, SP



SUB SP, SP, #16
STR R0, [FP, #-16]
STR R1, [FP, #-12]
STR R2, [FP, #-8]
STR R3, [FP, #-4]



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

sq_sum5's stack frame (2)

sq_sum5:

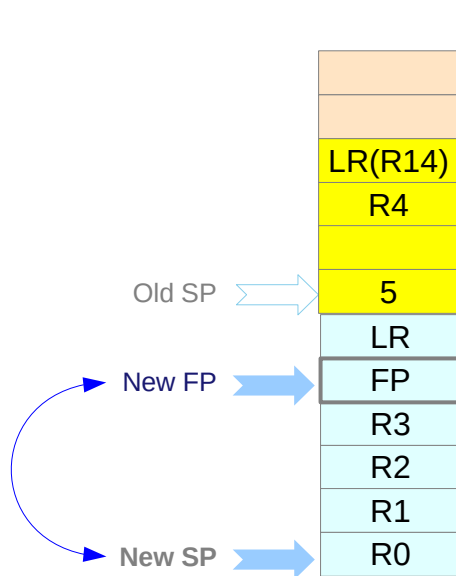
```
push {fp, lr}
mov fp, sp
```

```
sub sp, sp, #16
str r0, [fp, #-16]
str r1, [fp, #-12]
str r2, [fp, #-8]
str r3, [fp, #-4]
```

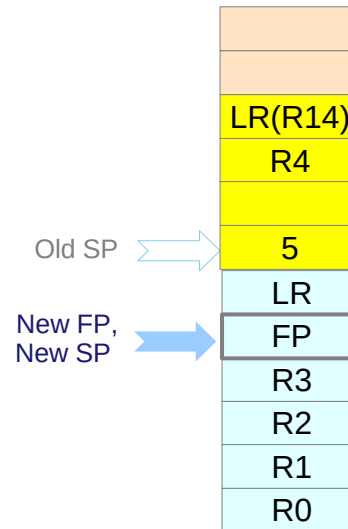
```
sub r0, fp, #16
bl sq
sub r0, fp, #12
bl sq
sub r0, fp, #8
bl sq
sub r0, fp, #4
bl sq
add r0, fp, #8
bl sq
```

```
ldr r0, [fp, #-16]
ldr r1, [fp, #-12]
add r0, r0, r1
ldr r1, [fp, #-8]
add r0, r0, r1
ldr r1, [fp, #-4]
add r0, r0, r1
ldr r1, [fp, #8]
add r0, r0, r1
```

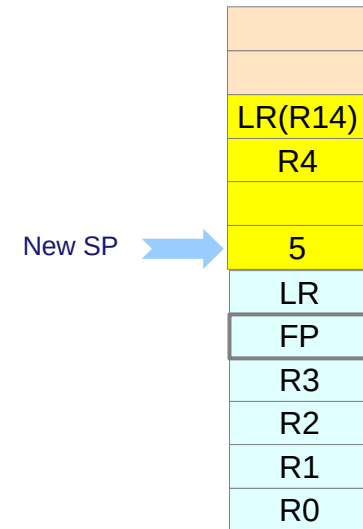
```
mov sp, fp
pop {fp, lr}
bx lr
```



MOV SP, FP



POP {FP, LR}

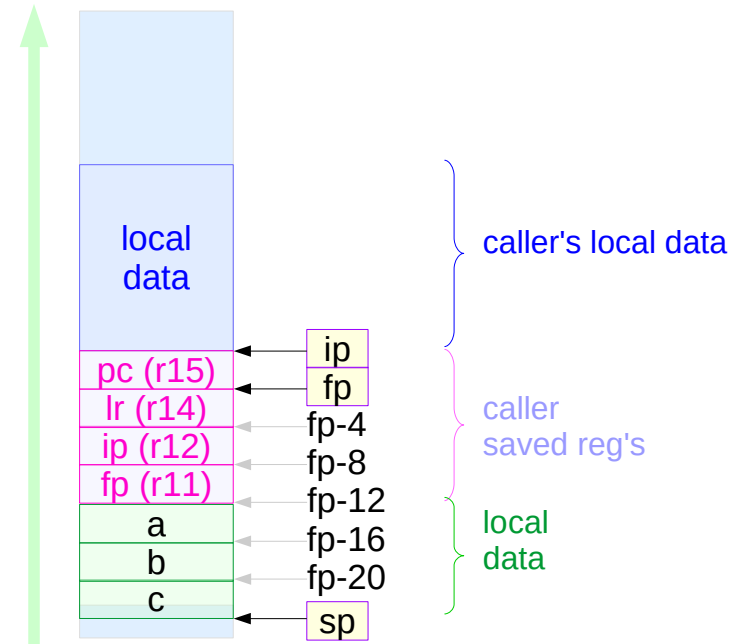


<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

-fno-omit-frame-pointer

```
main:  
mov     ip, sp  
stmfd  sp!, { fp, ip, lr, pc }  
sub     fp, ip, #4  
sub     sp, sp, #12  
ldr     r2, [fp, #-16]  
ldr     r3, [fp, #-20]  
add     r3, r3, r2  
str     r3, [fp, #-24]  
sub     sp, fp, #12  
ldmfd  sp, {fp, sp, pc}
```

```
main()  
{  
    volatile int a, b, c;  
    c = a + b;  
}
```

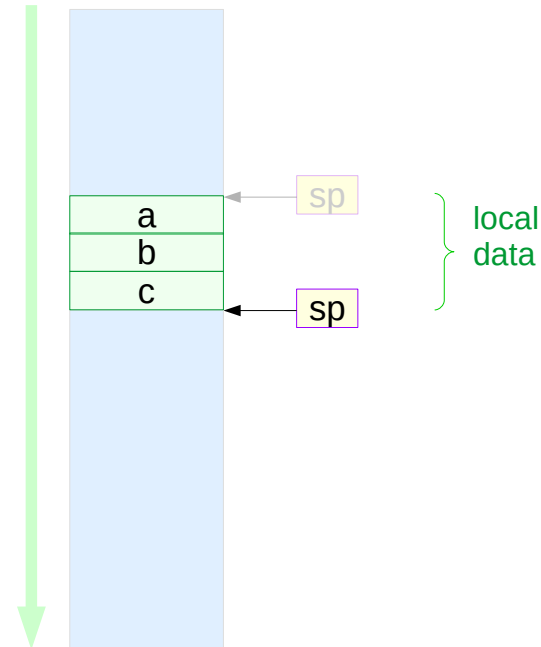


<https://community.arm.com/thread/7092>

-fomit-frame-pointer

```
main:  
sub    sp, sp, #12  
ldr    r2, [sp, #8]  
ldr    r3, [fp, #4]  
add    r3, r3, r2  
str    r3, [sp, #0]  
sub    sp, sp, #12
```

```
main()  
{  
    volatile int a, b, c;  
    c = a + b;  
}
```



<https://community.arm.com/thread/7092>

ARM Register Conventions (1)

Names	Reg No	Usage	preserved
a1-a2	R0-R1	Argument / return result / scratch register	no
a3-a4	R2-R3	Argument / scratch register	no
v1-v8	R4-R11	Variables for local routine	yes
fp	R11	Frame pointer register	no
ip	R12	Intra procedure call scratch register	no
sp	R13	Stack pointer	yes
lr	R14	Link register (Return address)	yes
pc	R15	Program counter	n.a.

ARM Register Conventions (2)

R0	a0	argument	scratch	Return Result
R1	a1			
R2	a2			
R3	a3			
R4	v1	Variable		
R5	v2			
R6	v3			
R7	v4			
R8	v5			
R9	v6			
R10	v7			
R11	v8			
R12			scratch	FP
R13				IP
R14				SP
R15				LR
				PC

R0	not preserved
R1	
R2	
R3	
R4	preserved
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	Not preser.
R13	preserved
R14	
R15	

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

Argument, scratch, variable, return result registers

R0 – R3, R12 :

argument or scratch registers
that are not preserved
by the **callee** on a procedure call

R4 – 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**

R0	Return Result	argument	scratch
R1			
R2			
R3			
R4		Variable	
R5			
R6			
R7			
R8			
R9			
R10			
R11	FP		
R12	IP		scratch
R13	SP		
R14	LR		
R15	PC		

Preserving, non-preserving registers

Registers that is preserved across a procedure

- variable registers **R4 – R11**
- stack pointer register **SP (R13)**
- link register **LR (R14)**
- stack above the stack pointer

Registers that is not preserved across a procedure

- argument registers **R0 – R3**
- intra procedure call scratch register **IP (R12)**
- stack below the stack pointer

R0	not preserved	caller save registers
R1		
R2		
R3		
R4	preserved	callee save registers
R5		
R6		
R7		
R8		
R9		
R10		
R11		
R12	Not preser.	
R13	preserved	
R14		
R15		

ARM Architecture Procedure Call Standard (AAPCS)

Application **B**inary **I**nterface (**ABI**) standard for ARM
allows assembly subroutine
to be callable from C or
callable from someone else's software

parameters passed using registers and stack

- registers **R0**, **R1**, **R2**, and **R3**
to pass the first four input parameters in order
- pass additional parameters via the **stack**
- place the return parameter in register **R0**.

- functions can **freely modify** registers **R0–R3** and **R12**.

- If a function uses **R4–R11**,
 - (a) **push** current register values onto the stack,
 - (b) **use** the registers, and then
 - (c) **pop** the old values off the stack before **returning**.

<https://www.eng.auburn.edu/~nelsovp/courses/elec2220/slides/ARM%20prog%20model%20%20subroutines.pdf>

APCS Register Use Convention

for **non-reentrant, non-variadic functions**

the stack backtrace structure can be created in just 3 instructions, as follows:

```
MOV      ip, sp                ; save current sp, ready to save as old sp
STMFD   sp!, {a1-a4, v1-v5, sb, fp, ip, lr, pc} ; as needed
SUB     fp, ip, #4
```

each argument register **a1-a4** need only be saved
if a memory location is needed
for the corresponding parameter
because it has been spilled by the register allocator or
because its address has been taken

each of the variable registers **v1-v7** need only be saved
if it used by the called function.

the minimum set of registers to be saved is {**fp, old-sp, lr, pc**}.

<https://www.cl.cam.ac.uk/~fms27/teaching/2001-02/arm-project/02-sort/apcs.txt>

Variadic functions

a function of indefinite arity
a variable number of arguments

```
#include <stdarg.h>
#include <stdio.h>

double average(int count, ...) {
    va_list ap;
    int j;
    double sum = 0;

    va_start(ap, count);
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, int);
    }
    va_end(ap);

    return sum / count;
}
```

```
average(3, 1, 2, 3);
```

```
average(4, 1, 2, 3, 4);
```

```
average(5, 1, 2, 3, 4, 5);
```

```
average(6, 1, 2, 3, 4, 5, 6);
```

https://en.wikipedia.org/wiki/Variadic_function

Re-entrant functions

Function is called reentrant if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution

- No **static** or **global** variables
- No **self-modifying** code
- No call to such functions that does not comply with the two rules above

When to use re-entrant function

- executed in interrupt context
- called from multiple threads/tasks

```
int i;
```

```
int fun1()    // non-reentrant  
{  
    return i * 5;  
}
```

```
int fun1(int i) // re-entrant version  
{  
    return i * 5;  
}
```

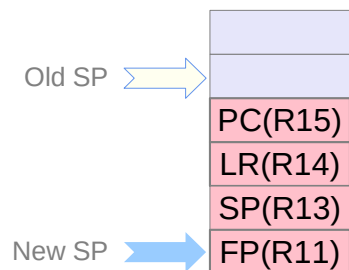
https://en.wikipedia.org/wiki/Variadic_function

APCS Register Use Convention

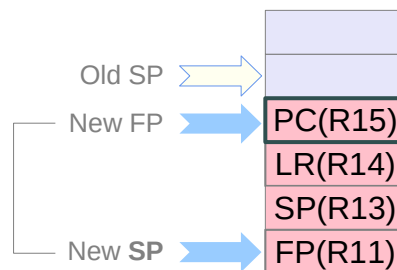
```
MOV      ip, sp      ; ip ← sp
STMFD   sp!, {fp, ip, lr, pc} ; minimum set of registers
SUB     fp, ip, #4
```

R15 **PC** Program counter
R14 **LR** Link address / scratch register
R13 **SP** Lower end of current stack frame
R11 **FP** Frame Pointer

```
MOV      IP, SP
STMFD   SP!, {FP, IP, LR, PC}
```



```
SUB     FP, IP, #4
```



<https://www.cl.cam.ac.uk/~fms27/teaching/2001-02/arm-project/02-sort/apcs.txt>

APCS Register Use Convention

A **reentrant** function must **avoid** using **ip** in its entry sequence:

```
STMFD    sp!, {sp, lr, pc}
STMFD    sp!, {a1-a4, v1-v5, sb, fp}           ; as needed
ADD      fp, sp, #8+4*|{a1-a4, v1-v5, sb, fp}| ; as used above
```

sb (aka **v6**) must be saved by a **reentrant function** if it calls any function from another link unit (which would alter the value in **sb**).

This means that, in general, **sb** must be saved on entry to all **non-leaf, reentrant functions**.

<https://www.cl.cam.ac.uk/~fms27/teaching/2001-02/arm-project/02-sort/apcs.txt>

APCS Register Use Convention

MOV ip, sp
STMFD sp!, {a1-a4, v1-v5, sb, fp, ip, lr, pc}
SUB fp, ip, #4

STMFD sp!, {sp, lr, pc}
STMFD sp!, {a1-a4, v1-v5, sb, fp}
ADD fp, sp, #8+4*|{a1-a4, v1-v5, sb, fp}|

R0	a0	
R1	a1	
R2	a2	
R3	a3	
R4	v1	
R5	v2	
R6	v3	
R7	v4	
R8	v5	
R9	v6	SB
R10	v7	
R11	v8	FP
R12		IP
R13		SP
R14		LR
R15		PC

<https://www.cl.cam.ac.uk/~fms27/teaching/2001-02/arm-project/02-sort/apcs.txt>

IP register

The **ip** register has
a dedicated role only during **function call**;
at other times it may be used as a **scratch register**.

Conventionally, **ip** is used by compiler code generators
as the/a local code generator **temporary register**

<https://www.cl.cam.ac.uk/~fms27/teaching/2001-02/arm-project/02-sort/apcs.txt>

Intra Procedure call scratch register

use **R12** as a temporary within routines.

- not be saved by called routines.
- no guarantee for a unchanged value between a call and entering the target routine.

for instance a **call** to a **library routine**

where the library is only loaded or the call is only fixed up if the call is handled by entering *a little routine* which then goes to the target.

- the routine will use **R12** since it doesn't matter if that is corrupted.
- Having **the register free** makes this much more efficient.

there are also cases in some systems

where the **return** is sometimes monitored as well

- the **exit** doesn't return to the calling routine directly but to *a little routine* which then returns to the caller.

<https://community.arm.com/developer/tools-software/oss-platforms/f/dev-platforms-forum/5436/i-want-to-know-meaning-of-r12-register>

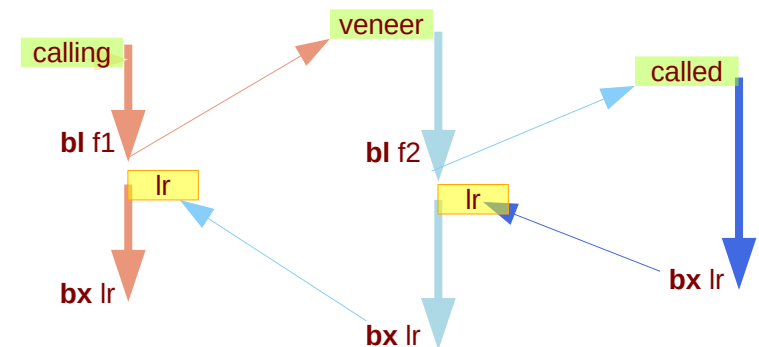
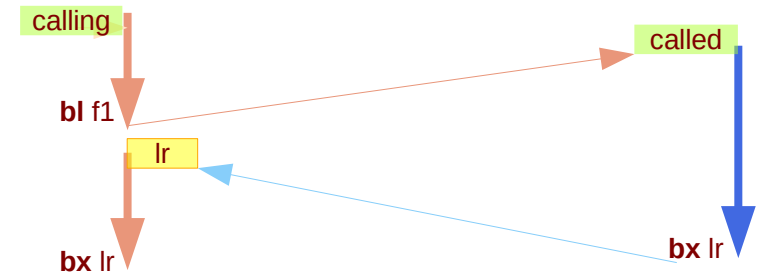
IP Register – used by the linker

both the **ARM-** and **Thumb-state BL** instructions are unable to address the **full 32-bit address space**, so it may be necessary for the **linker** to insert a **vneer** between the calling routine and the called subroutine.

Veneers may also be needed to support **ARM-Thumb inter-working** or **dynamic linking**.

Any **vneer** inserted must preserve the contents of all registers **except IP (R12)** and the condition code flags;

a conforming program must assume that a **vneer** that alters IP may be inserted at any **branch instruction** that is exposed to a **relocation** that supports **inter-working** or **long branches**.



file:///home/young/Descargas/IHI0042E_aapcs.pdf

Caller-saved / Callee-saved registers

R0-R3 are used as **parameters**
R0-R1 are used as **return values**

R0-R3 are **caller-saved** if needed.
R4-R8 are **callee-saved** registers (and maybe more).

R13-R15 are special registers.

R12 allows **groups of registers**
to be accessed efficiently with **LDM/STM**
as you may want to treat groups differently
context save, function call, signal, etc.
all have different requirements

STM and **LDM** instructions store and load value
in a numerically **increasing/decreasing order**.

R0	not preserved	caller save registers
R1		
R2		
R3		
R4	preserved	callee save registers
R5		
R6		
R7		
R8		
R9		
R10		
R11		
R12	Not preser.	
R13	preserved	
R14		
R15		

<https://stackoverflow.com/questions/45359408/r12-in-the-arm-procedure-call-standard>

Why R12 as a scratch register (1)

Why is **R12** designated as a scratch register

- use **consecutive registers** for similar functionality
- easier to remember and does give more flexibility with some ARM instructions.
- easy implementation of **code generation** because you only need to save an upper limit to know callee-saved registers.
- To make function **epilogue/prologue** as fast as possible

R0	not preserved	caller save registers
R1		
R2		
R3		
R4	preserved	callee save registers
R5		
R6		
R7		
R8		
R9		
R10		
R11		
R12	Not preser.	
R13	preserved	
R14		
R15		

The choice of **R12** allows some systems to use **R9-R11** as **callee-saved general purpose** registers without disrupting any sequence of similar registers.

<https://stackoverflow.com/questions/45359408/r12-in-the-arm-procedure-call-standard>

Why R12 as a scratch register (2)

Function epilogue and prologue code may need to run calculations and/or save values.

A scratch register is needed for this.

So, given **LDM/STM** and other ARM addressing modes, the scratch register should NOT break continuous sequences.

You may need to save/restore both caller and callee saved registers depending on context and code generation strategies.

Having a break at **R4** is not a good option.

A natural break with least impact is between callee and upper intrinsic registers (**SP,LR,PC**).

Note that **R9-R11** can be special registers depending on the system (static base, stack extents, and frame pointer) in the prior APCS.

As these are optional, in some systems they maybe saved as per R4-R8.

<https://stackoverflow.com/questions/45359408/r12-in-the-arm-procedure-call-standard>

R0	not preserved	caller save registers	
R1			
R2			
R3			
R4	preserved	callee save registers	
R5			
R6			
R7			
R8			
R9			
R10			
R11			SB
R12	Not preser.		SE
R13	preserved		FP
R14			IP
R15			SP
			LR
			PC

Why a scratch register is needed (1)

Why a scratch register is needed

- **variable sized arrays** based on parameters would be difficult to implement without **multiple stack reservations**.
- some code such as **signals** may rely on **FP** to be atomically set during a **prologue**;
- Ie, you are in the function with **stack** and **frame pointer set** or you are not with no in-between.
- an **IP (R12)** is also useful for **veneers** and other linkage tricks (**PLT,GOT**,etc).

R0	not preserved	caller save registers
R1		
R2		
R3		
R4	preserved	callee save registers
R5		
R6		
R7		
R8		
R9		
R10		
R11		
R12	Not preser.	
R13	preserved	
R14		
R15		

<https://stackoverflow.com/questions/45359408/r12-in-the-arm-procedure-call-standard>

Why a scratch register is needed (2)

the difference between **R12** and **R0-R3**.

at the entry of a function,

R0-R3 may not available to be used
depending on the function **parameters**

Because parameters are passed in **R0-R3**
though, at the later parts of the routine,
R0-R3 can be used as scratch registers

R12 can always be used.

even at the entry of a function
the **R12** can be used

this fact makes **R12** useful for **veneers**, etc.

<https://stackoverflow.com/questions/45359408/r12-in-the-arm-procedure-call-standard>

References

- [1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
- [2] <http://blog.bobuhiro11.net/2014/01-13-baremetal.html>
- [3] <http://www.valvers.com/open-software/raspberry-pi/>
- [4] <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html>