

Version 2.00

# UCSD CSE 30

## Computer Organization and Systems Programming

### Arm Assembly – Part 4

Keith Muller





# Function Calls

Branch with Link (function call) instruction

`bl label`

`bl`

`imm24`

- Function call to the instruction with the address `label` (no local labels for functions)
  - `imm24` number of instructions from pc+8 (24-bits)
  - `label` any function label in the current file, any function label that is defined as `.global` in any file that it is linked to, any C function that is not static

Branch with Link Indirect (function call) instruction

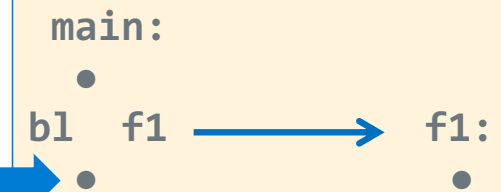
`blx Rm`

`blx`

`Rm`

- Function call to the instruction whose address is stored in Rm (Rm is a function pointer)
- `bl` and `blx` both save the address of the instruction immediately following the `bl` or `blx` instruction in register `lr` (link register is also known as r14)
- The contents of the link register is the return address in the calling function

- (1) Branch to the instruction with the label f1
- (2) copies the address of the instruction AFTER the `bl` in `lr`



# Function Call Return

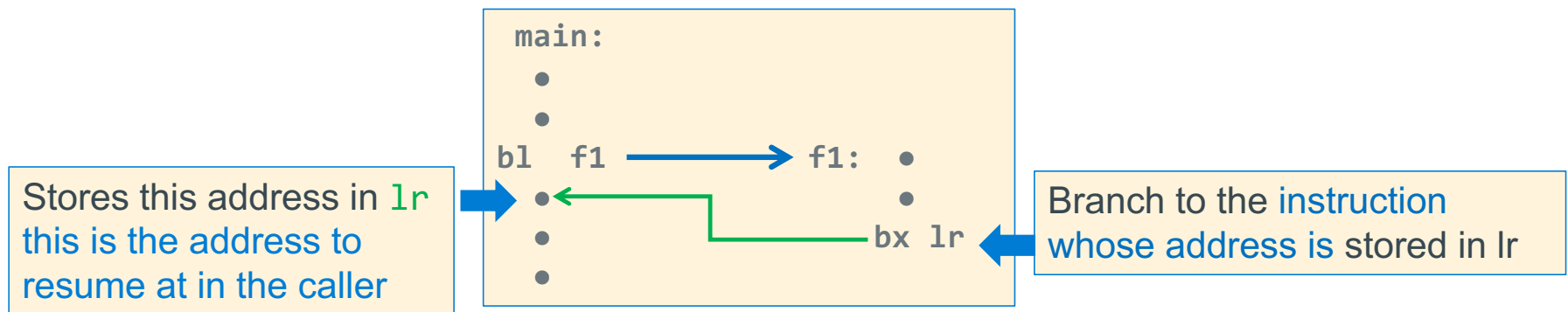
Branch & exchange (function return) instruction

`bx lr`

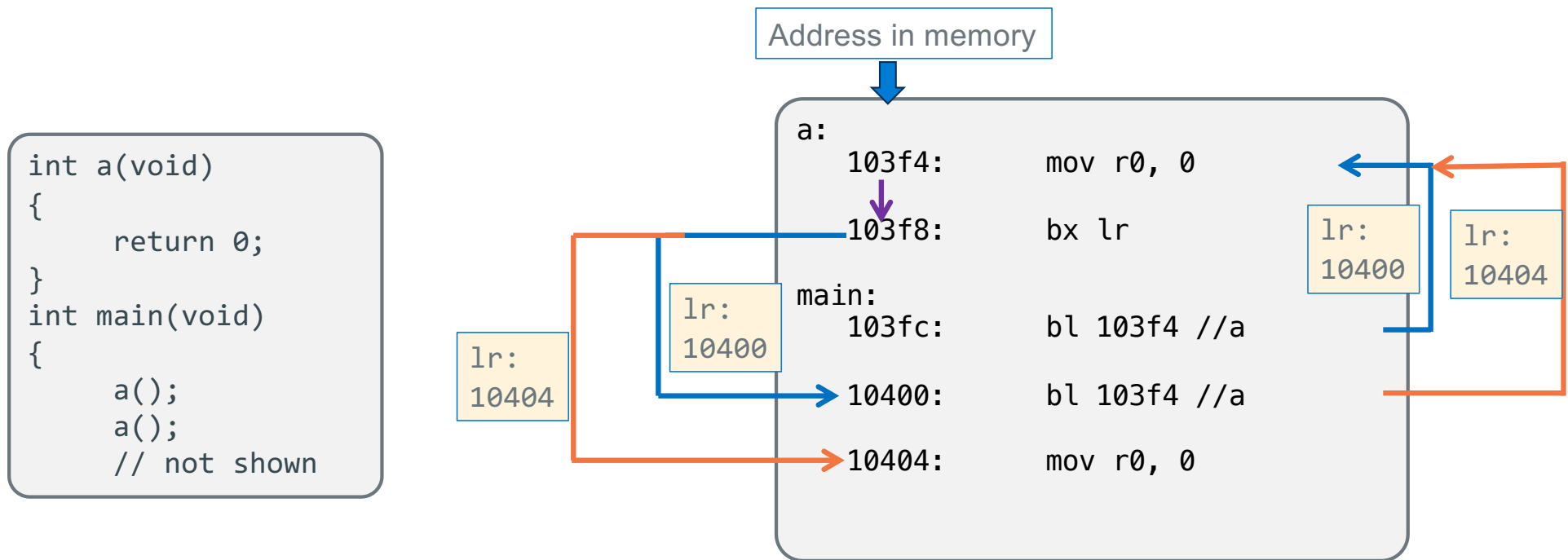


// we will always use lr

- Causes a branch to the instruction whose address is stored in register <lr>
  - It copies lr to the PC
- This is often used to implement a return from a function call (exactly like a C return) when the function is called using either `bl label`, or `blx Rm`



# Understanding bl and bx - 1

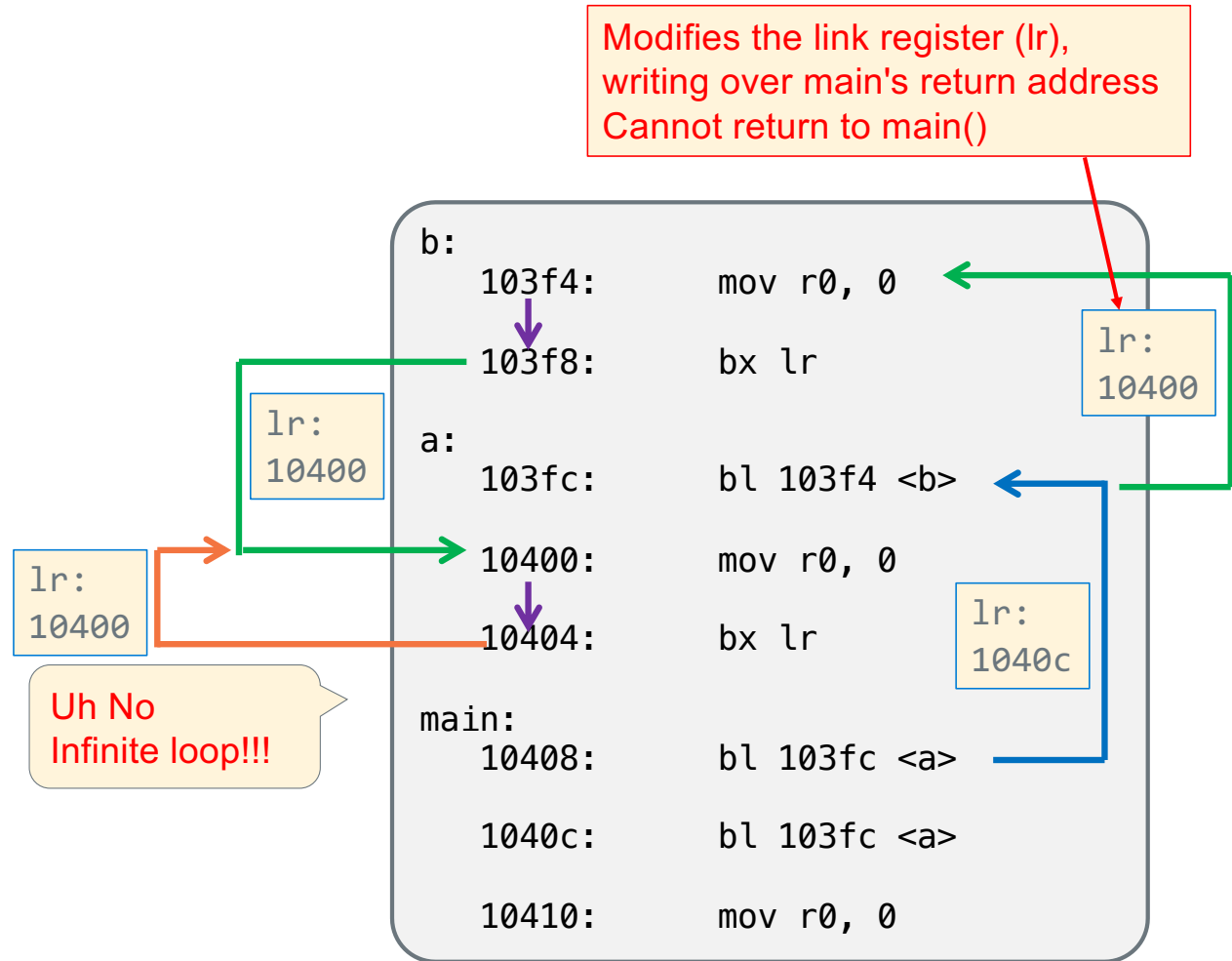


But there is a problem we must address here – next slide

## Understanding bl and bx - 2

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
    // not shown
}
```

We need to preserve the lr!



## Understanding bl and blx - 3

```
int a(void)
{
    return 0;
}

int (*func)() = a;

int main(void)
{
    (*func)();
    // not shown
}
```

But this has the same infinite loop problem when main() returns!

```
.data
func: .word a // func initialized with address of a()

.text
.global a
.type a, %function
.equ FP_OFF, 4

a:
    mov     r0, 0
    bx      lr
    .size a, (. - a)

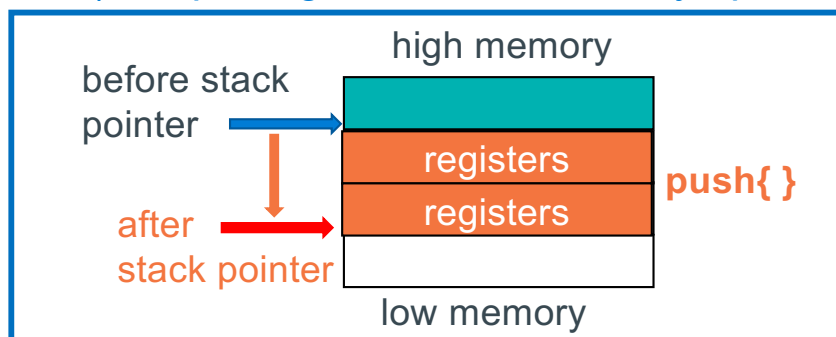
.global main
.type main, %function
.equ FP_OFF, 4

main:
    ldr     r4, =func // load address of func in r4
    ldr     r4, [r4]   // load contents of func in r4
    blx     r4         // we lose the lr for main!
    // not shown
    bx      lr         // infinite loop!
```

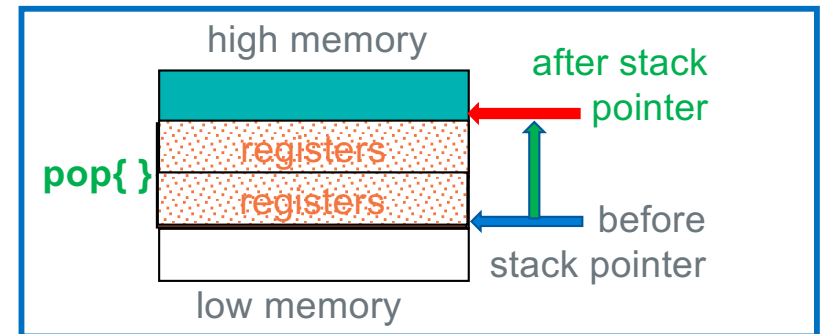
## Preserving and Restoring Registers on the stack - 1

Operation	Pseudo Instruction	Operation
Push registers Function Entry	<code>push {reg list}</code>	$sp = sp - 4 \times \text{\#registers}$ Copy registers to <code>mem[sp]</code>
Pop registers Function Exit	<code>pop {reg list}</code>	Copy <code>mem[sp]</code> to registers, $sp = sp + 4 \times \text{\#registers}$

push (multiple register **str** to memory operation)



push (multiple register **ldr** from memory operation)



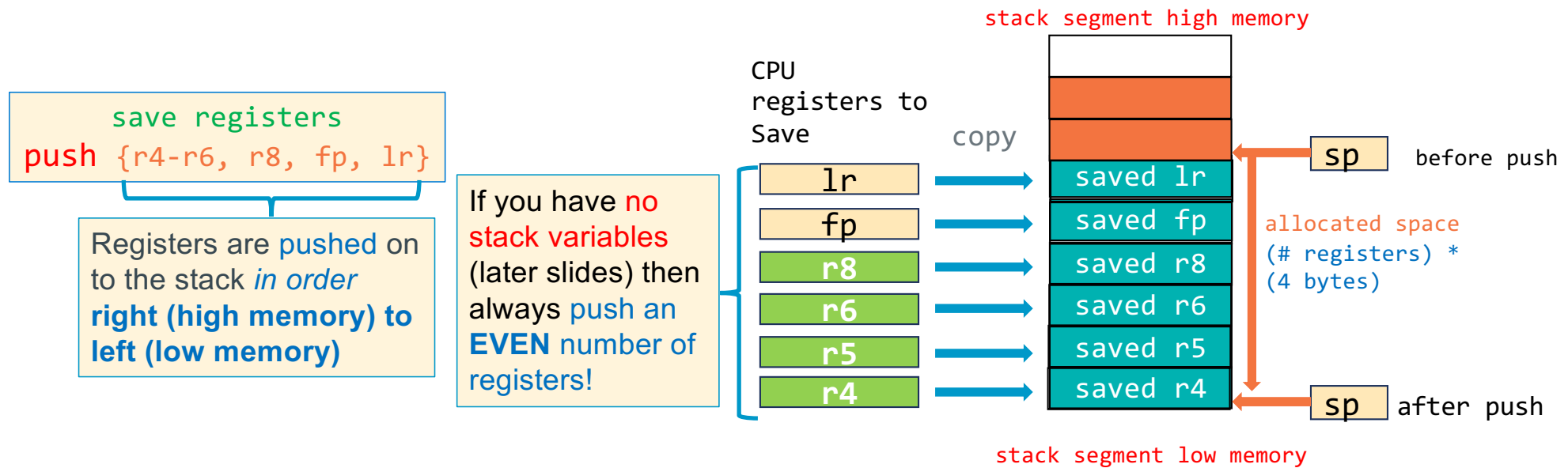
## Preserving and Restoring Registers on the Stack - 2

Operation	Pseudo Instruction	Operation
Push registers Function Entry	<code>push {reg list}</code>	$sp = sp - 4 \times \text{\#registers}$ Copy registers to <code>mem[sp]</code>
Pop registers Function Exit	<code>pop {reg list}</code>	Copy <code>mem[sp]</code> to registers, $sp = sp + 4 \times \text{\#registers}$

- `{reg list}` is a **list of registers in numerically increasing order, left to right**  
`push {r4-r10, fp, lr}` // *fp is r11, lr is r14*
- Registers **cannot be**:
  1. duplicated in the list
  2. listed out of increasing numeric order (left to right)
- Register ranges can be specified `{r4, r5, r8-r10, fp, lr}`
- **Never!** push/pop `r12, r13, or r15`
  - the top two registers on the stack must always be `fp, lr` // ARM function spec – later slides

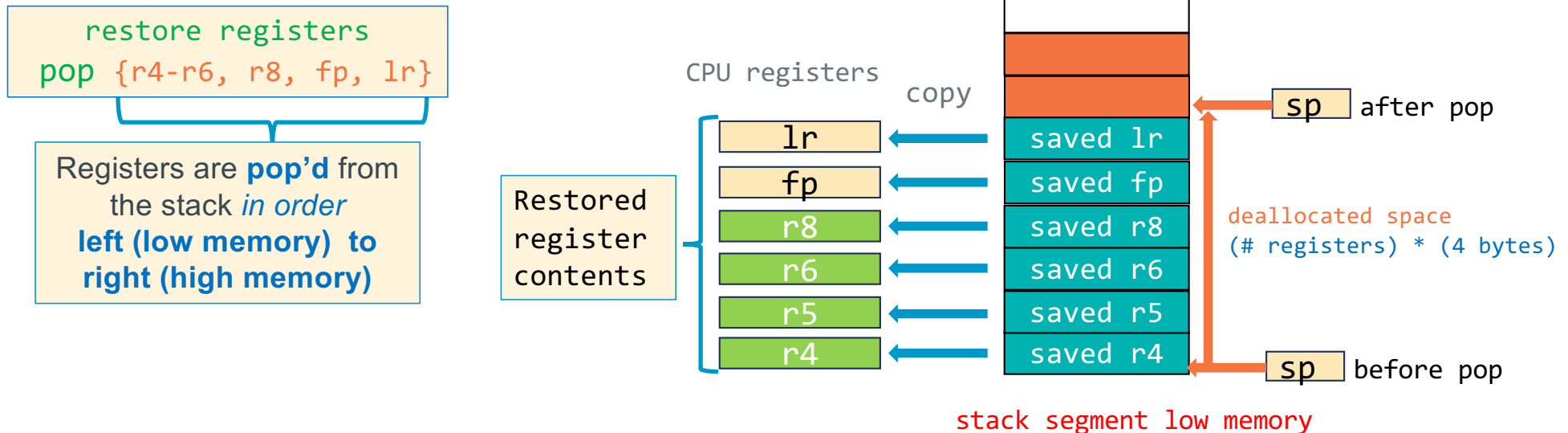


# push: Multiple Register Save to the stack



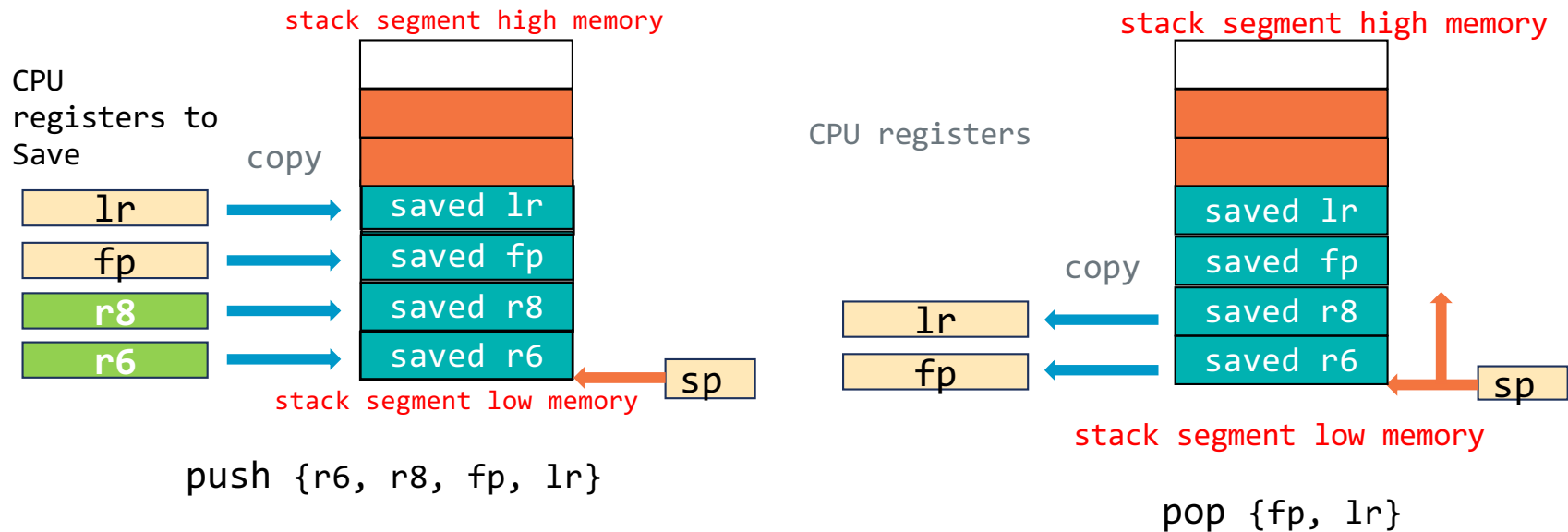
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** subtracts  $(\text{\# of registers saved}) * (4 \text{ bytes})$  from the **sp** to **allocate** space on the stack
  - $sp = sp - (\text{\# registers\_saved} * 4)$
- **this must always be true:  $sp \% 8 == 0$**

# pop: Multiple Register Restore from the stack



- **pop** copies the contents of stack segment memory to the **{reg list}**
- **pop adds:**  $(\# \text{ of registers restored}) * (4 \text{ bytes})$  to **sp** to **deallocate** space on the stack
  - $sp = sp + (\# \text{ registers restored} * 4)$
- **Remember:** **{reg list}** must be the same in both the **push** and the corresponding **pop**

# Consequences of inconsistent push and pop operands



- lr gets contents of saved r8, likely causing a segmentation fault when the `bx lr` is executed at function exit

## Registers: Rules For Use

<i>Register</i>	<i>Function Call Use</i>	<i>Function Body Use</i>	<i>Save before use Restore before return</i>
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No
r4-r10	preserved registers	contents preserved across function calls	Yes
r11 / fp	stack frame pointer	Use to locate variables on the stack	Yes
r12 / ip	may used by assembler with large text file	can be used as a scratch if really needed	No
r13 / sp	stack pointer	stack space allocation	Yes
r14 / lr	link register	contains return address for function calls	Yes
r15	Do not use	Do not use	No

# Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use	Function Body Use	Save before use Restore before return
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No

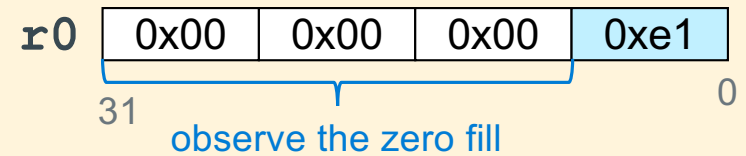
- Where **r0, r1, r2, r3** are arm registers, the function declaration is (first four arguments):  
`r0 = function(r0, r1, r2, r3) // 32-bit return`
- Each **parameter and return value is limited to data that can fit in 4 bytes or less**
- **Calling function:**
  - copy up to the first four parameters into these four registers before calling a function
  - **MUST assume** that the called function will **alter the contents of all four registers: r0-r3**
  - **In terms of C runtime support, these registers contain the copies given to the called function**
  - **C allows the copies to be changed in any way by the called function**
- For **parameters, whose size is larger than 4 bytes**, **pass a pointer to the parameter** (we will cover this later)
- **Called function:**
  - you receive the first four parameters in these four registers (r0 – r3)



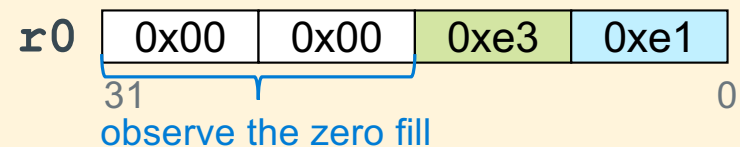
# Argument and Return Value Requirements

- When passing or returning values from a function you must do the following:
  - Make sure that the values in the registers r0-r3 are in their **properly aligned position in the register based on data type**
  - Upper bytes in byte and halfword values in registers r0-r3 when passing arguments and returning values **are zero filled**

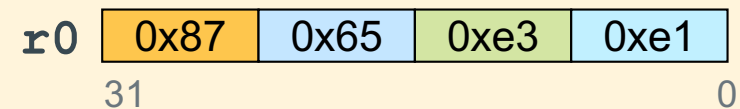
## Single Byte (char)



## Single Halfword (short)



## Full Word (int or pointer)



## What it means to be a Temporary/argument register

```
int a(void)
{
    // not shown
}
int main(void)
{
    int r0 = 0;
    int r1 = 1;
    int r2 = 2;
    int r3 = 3;
    r0 = a();
    // in C r1 and r3 would have the same values
    // after the call
```

```
// main()
// code not shown
mov r0, 0
mov r1, 1
mov r2, 2
mov r3, 3
bl a
// r0 = return value
// r1-r3 values are unknown as a() has right to change them as it wants
```

## Preserved Registers

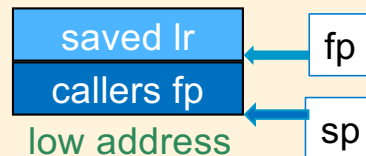
<i>Register</i>	<i>Function Call Use</i>	<i>Function Body Use</i>	<i>Save before use Restore before return</i>
r4-r10	preserved registers	contents preserved across function calls	Yes
r11/fp	stack frame pointer	Use to locate variables on the stack	Yes
r13/sp	stack pointer	stack space allocation	Yes
r14/lr	link register	contains return address for function calls	Yes

- **Any value** you have in a **preserved register before a function call will still be there after the function returns** (Contents are “preserved” across function calls)
- If the function **wants to use a preserved register** it must:
  1. **Save** the **value contained in the register** at function entry
  2. Use the register in the body of the function
  3. **Restore** the **original saved value** to the register at **function exit** (before returning to the caller)
- You use a preserved register when a function makes calls another function and you have:
  1. Local variables allocated to be in registers
  2. Parameters passed to you (in **r0-r3**) that **you need to continue to use after calling another function**

## Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

- **Minimal frame: allocating at function entry: `push {fp, lr}`**

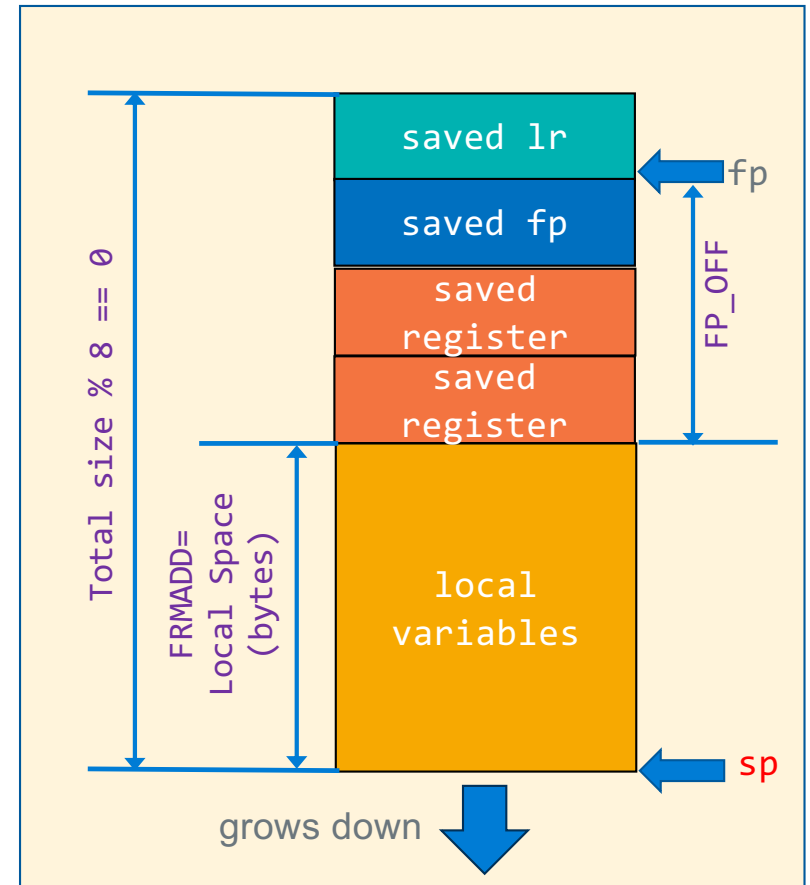
Minimum stack frame



- **sp** always points at top element in the stack (lowest byte address)
- **fp** always points at the bottom element in the stack
  - Bottom element is always the saved **lr** (contains the return address of caller)
  - A saved copy of **callers fp** is always the next element below the **lr**
  - **fp** will be used later when referencing stack variables
- **Minimal frame: deallocating at function exit: `pop {fp, lr}`**
- **On function entry:** **sp** must be 8-byte aligned (`sp % 8 == 0`)

## First Look: A typical Stack Frame

- Saved lr and fp of the caller (so function calls work)
- Save values for any preserved registers this function will change
- Space (FRMADD) for local variables is allocated on the stack right below the lowest pushed register





# Function Prologue and Epilogue

Function  
Prologue  
creates  
stack frame

myfunc:

```
{ push    {fp, lr}           // push (save) fp and lr on stack
  add     fp, sp, FP_OFF     // set fp for this function
  add     sp, sp, -FRMADD    // allocate FRMADD bytes for local vars
                               // by moving sp
  // your code
```

Function  
Epilogue  
removes  
stack frame

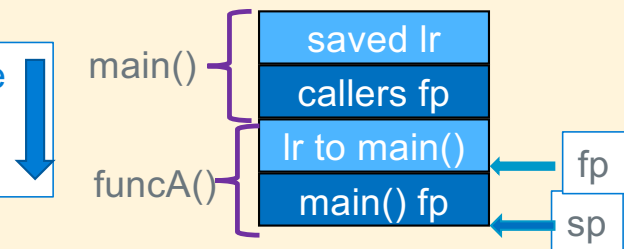
```
{ sub     sp, fp, FP_OFF     // deallocate local variables by moving sp
  pop     {fp, lr}           // pop (restore) fp and lr from stack
  bx      lr                 // return to caller
```

# Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

- **Function entry (Function Prologue):**

1. save lr and fp registers (push)
2. set fp to top entry in stack
3. allocate space for local vars – later slides

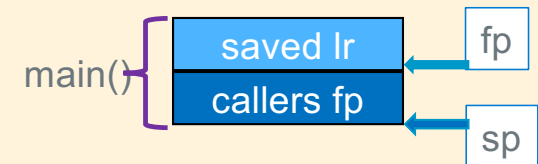
allocate stack space  
 $SP = SP - \text{"space"}$   
grows "down"



- **Function return (Function Epilogue):**

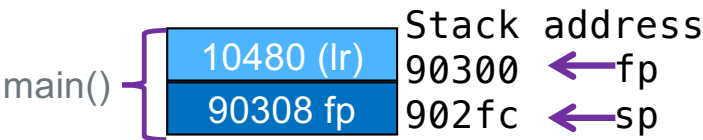
1. deallocate space for locals -later
2. restores lr and fp registers (pop)
3. Return To Caller

deallocate stack space  
 $SP = SP + \text{"space"}$   
shrinks "up"



# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



b:  
103f4: push {fp, lr}  
103f8: add fp, sp, 4  
103fc: mov r0, 0  
10400: sub sp, fp, 4  
10404: pop {fp, lr}  
10408: bx lr

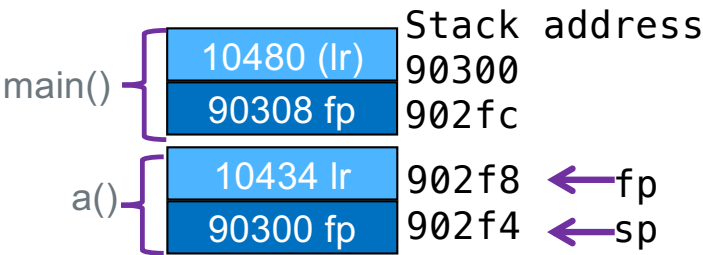
a:  
1040c: push {fp, lr}  
10410: add fp, sp, 4  
10414: bl 103f4 <b>  
10418: mov r0, 0  
1041c: sub sp, fp, 4  
10420: pop {fp, lr}  
10424: bx lr

main:  
10428: push {fp, lr}  
1042c: add fp, sp, 4  
10430: bl 1040c <a>  
10434: bl 1040c <a>  
  
// not shown

Memory address

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



**b:**

```
103f4:    push {fp, lr}
103f8:    add fp, sp, 4
103fc:    mov r0, 0
10400:    sub sp, fp, 4
10404:    pop {fp, lr}
10408:    bx lr
```

**a:**

```
1040c:    push {fp, lr}
10410:    add fp, sp, 4
10414:    bl 103f4 <b>
10418:    mov r0, 0
1041c:    sub sp, fp, 4
10420:    pop {fp, lr}
10424:    bx lr
```

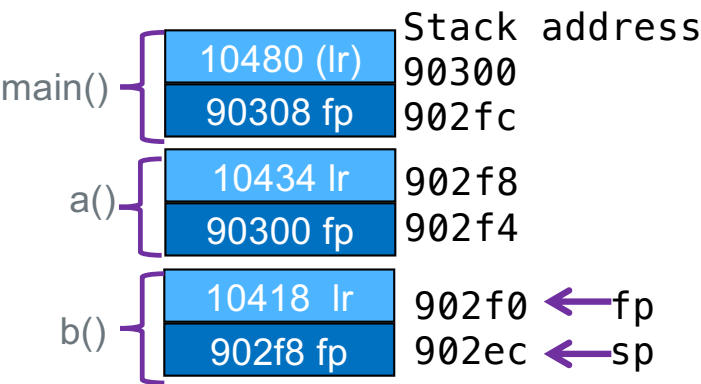
**main:**

```
10428:    push {fp, lr}
1042c:    add fp, sp, 4
10430:    bl 1040c <a>
10434:    bl 1040c <a>
// not shown
```

lr:  
10434

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



```
b:
103f4:  push {fp, lr}
103f8:  add fp, sp, 4
103fc:  mov r0, 0
10400:  sub sp, fp, 4
10404:  pop {fp, lr}
10408:  bx lr

a:
1040c:  push {fp, lr}
10410:  add fp, sp, 4
10414:  bl 103f4 <b>
10418:  mov r0, 0
1041c:  sub sp, fp, 4
10420:  pop {fp, lr}
10424:  bx lr

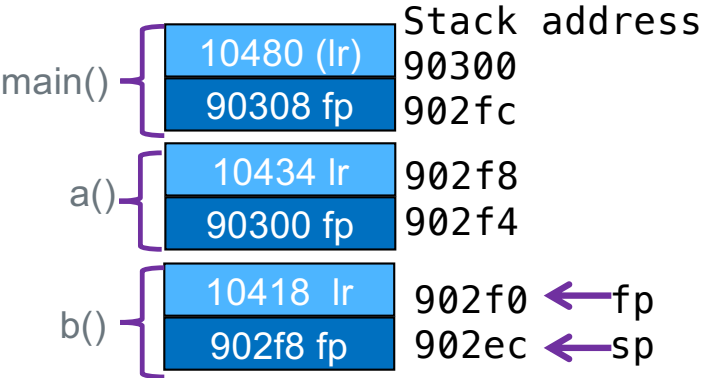
main:
10428:  push {fp, lr}
1042c:  add fp, sp, 4
10430:  bl 1040c <a>
10434:  bl 1040c <a>
// not shown
```

Diagram shows lr register values: 10418 for b() and 10434 for a(). Arrows indicate the flow of control and stack pointer updates.



# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



lr:  
10418

b:

103f4: push {fp, lr}

103f8: add fp, sp, 4

103fc: mov r0, 0

10400: sub sp, fp, 4

10404: **pop** {fp, lr}

10408: bx lr

lr:  
10418

a:

1040c: push {fp, lr}

10410: add fp, sp, 4

10414: bl 103f4 <b>

10418: mov r0, 0

1041c: sub sp, fp, 4

10420: **pop** {fp, lr}

10424: bx lr

lr:  
10434

main:

10428: push {fp, lr}

1042c: add fp, sp, 4

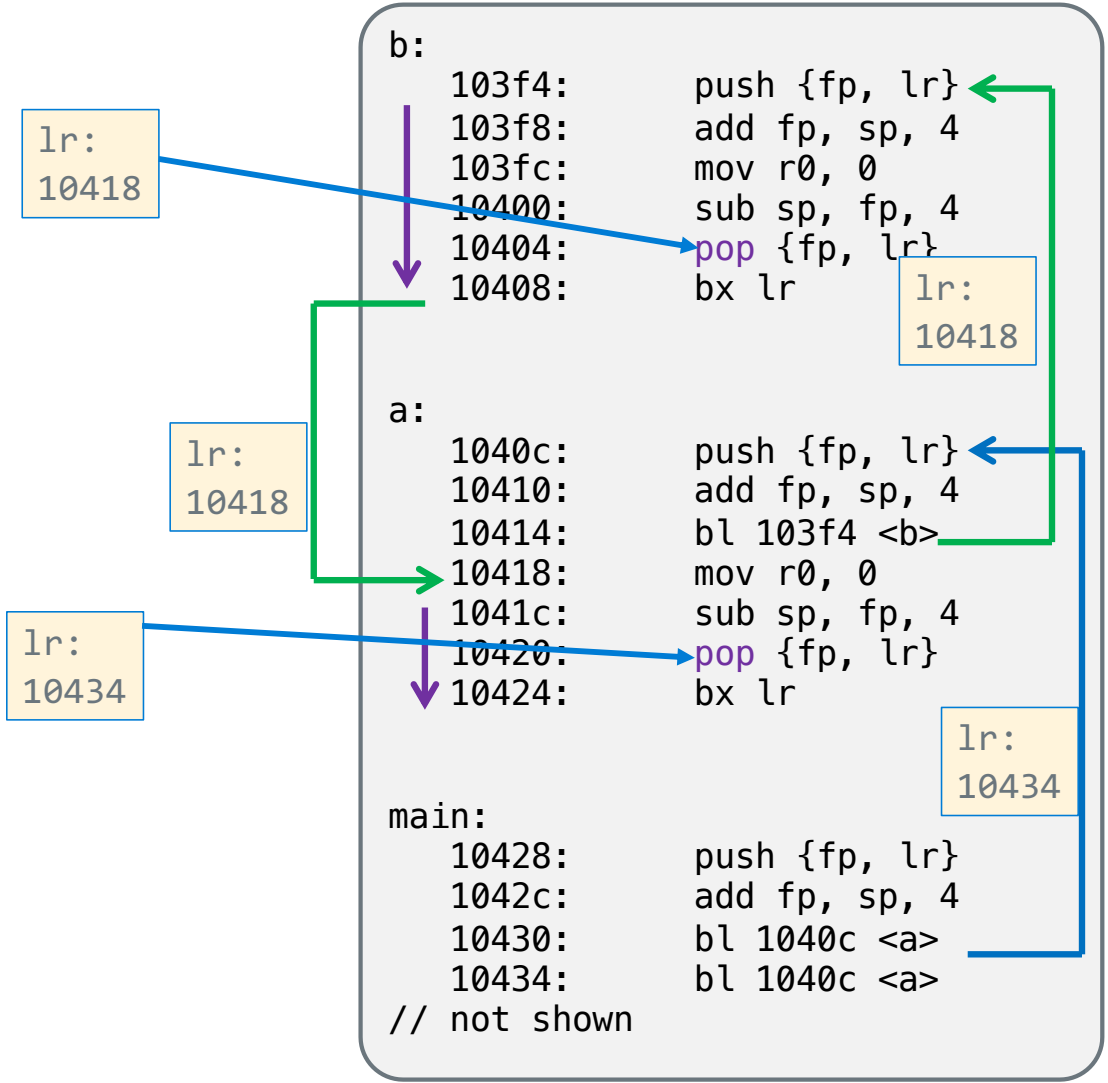
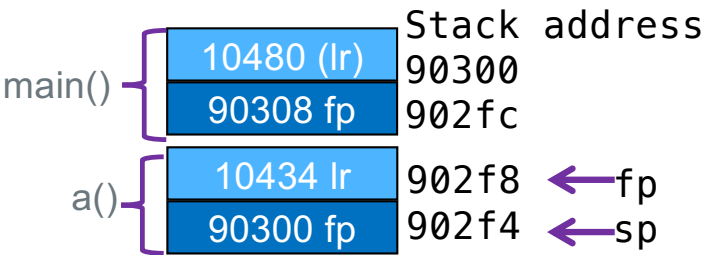
10430: bl 1040c <a>

10434: bl 1040c <a>

// not shown

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```

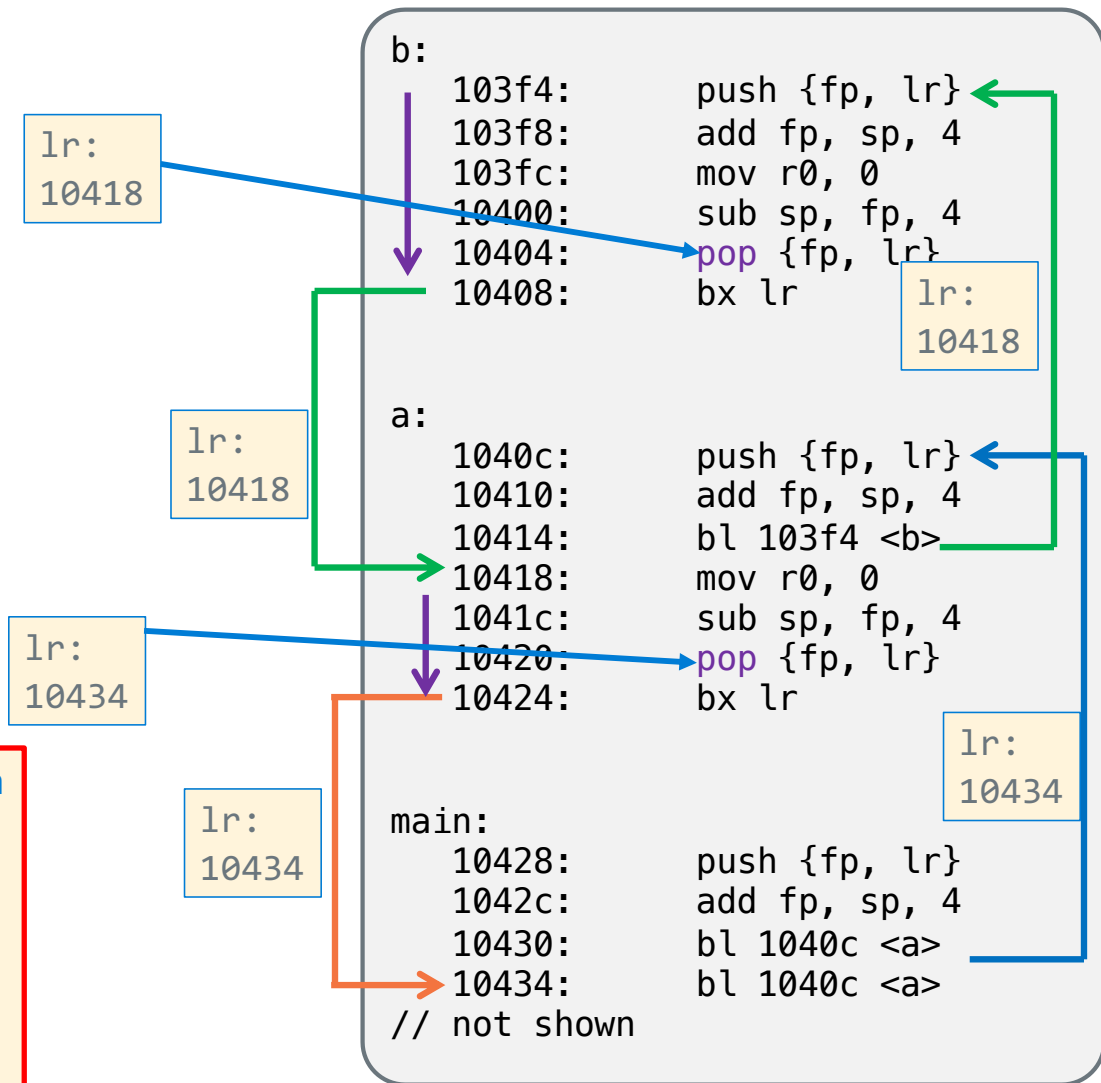
main() {

	Stack address
10480 (lr)	90300 ← fp
90308 fp	902fc ← sp

}

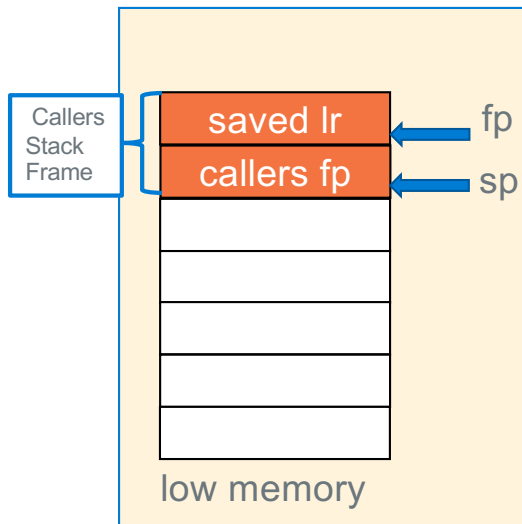
We are saving the lr on the stack on each function call and restoring it before returning.

**Result:** NO infinite loop and we return to the correct instruction in the caller no matter how many functions we call.  
Even recursion will work!



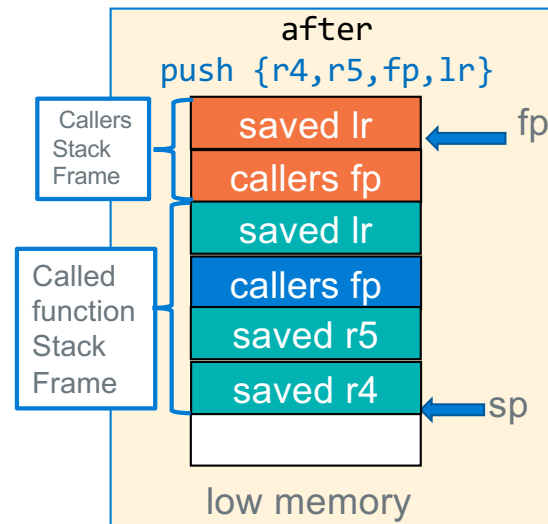
# Function Prologue: Allocating the Stack Frame -1

at function entry



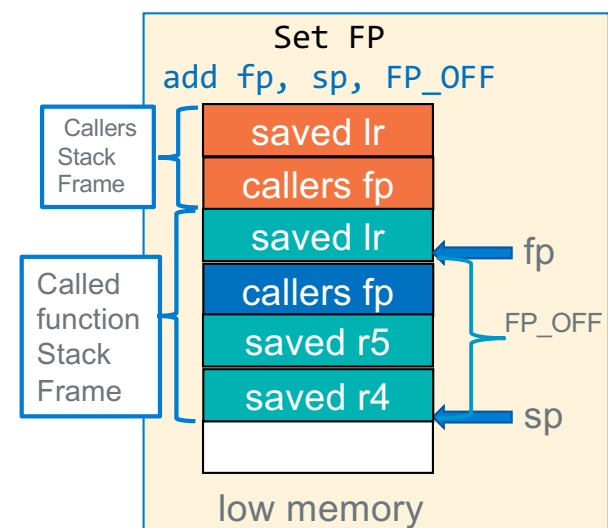
Function was just called this  
how the stack looks  
The orange blocks are part  
of the caller's stack frame

Prologue Step 1 of 3



using a push, save lr, fp and  
those preserved registers it  
wants to use on the stack

Prologue Step 2 of 3



move the fp to point at the  
saved lr as required by the  
Aarch32 spec

**myfunc:**

Function  
Prologue

```
push    {fp, lr}
add     fp, sp, FP_OFF
add     sp, sp, -FRMADD
```

```
// push (save) fp and lr on stack
// set fp for this function
// allocate FRMADD bytes for local vars
// by moving sp
```

## Function Prologue: Allocating the Stack Frame - 2 Prologue Step 3 of 3

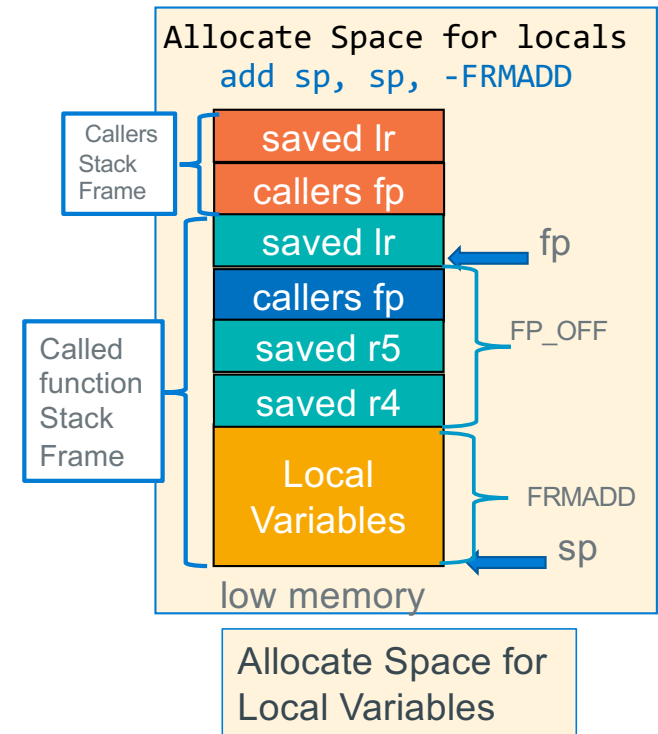
- Space for local variables is allocated on the stack right below the lowest pushed register
- Add memory to the stack frame for local variables** by **moving the sp towards low memory**
- The amount moved is the total size of all local variables in bytes **plus memory alignment padding**

$FRMADD = \text{total local var space (bytes)} + \text{padding}$

- Allocate the space after the register push by

```
add    sp, sp, -FRMADD
```

- fp (frame pointer) is used as a **pointer (base register)** to access all stack variables – later slides



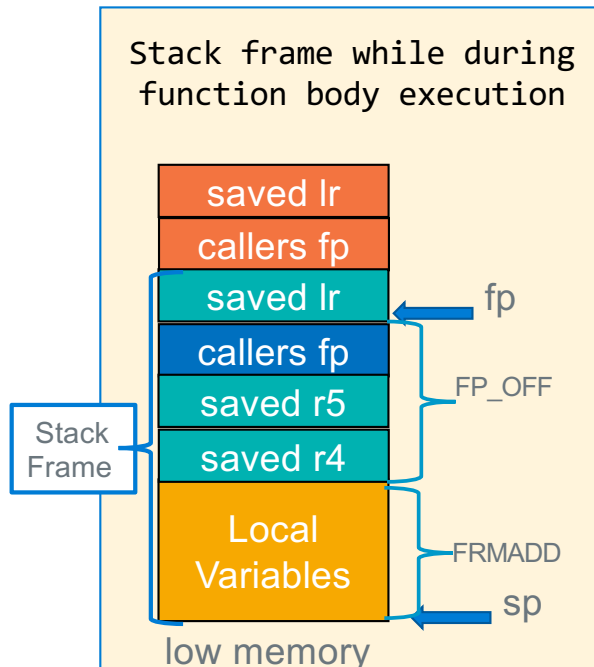
**myfunc:**

Function Prologue

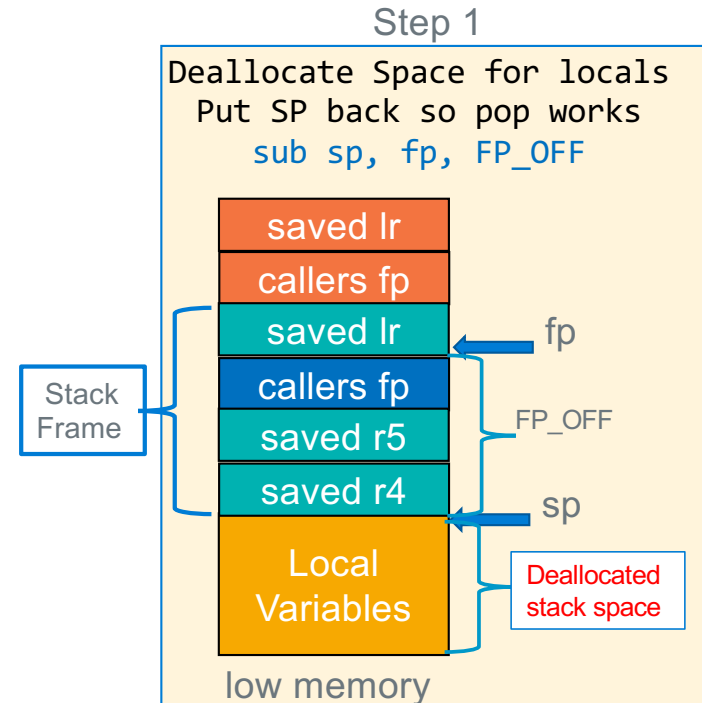
```
push    {fp, lr}           // push (save) fp and lr on stack
add     fp, sp, FP_OFF      // set fp for this function
add     sp, sp, -FRMADD     // allocate FRMADD bytes for local vars
                                // by moving sp
```



# Function Epilogue: Deallocating the Stack Frame - 1



Use fp as a pointer to find local variables on the stack



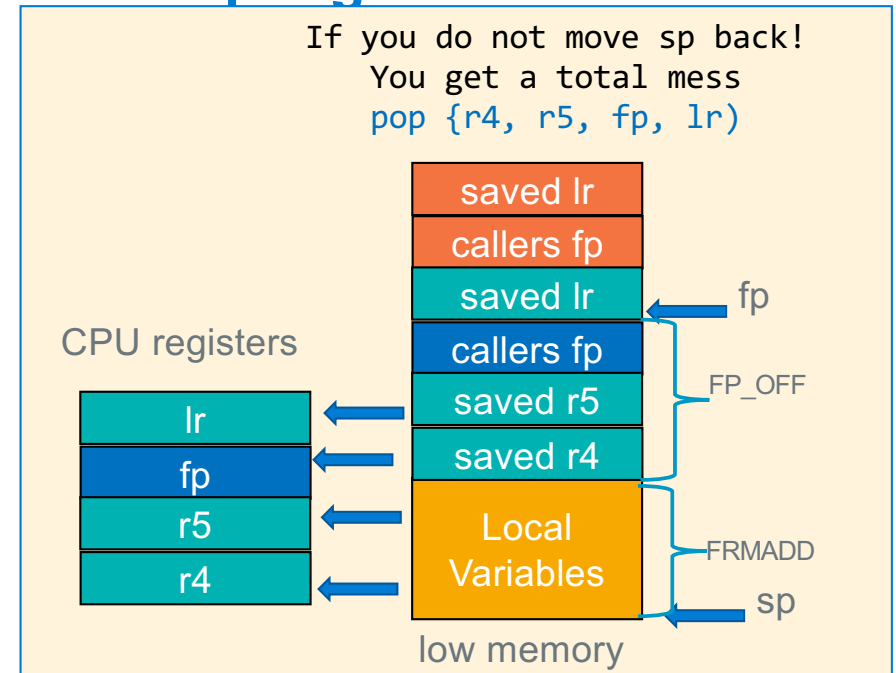
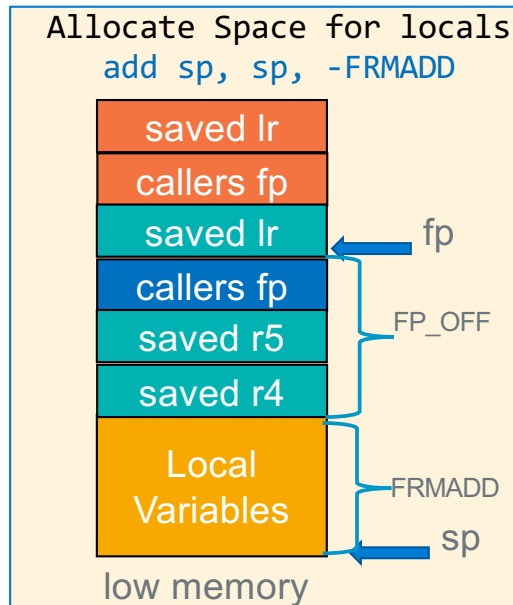
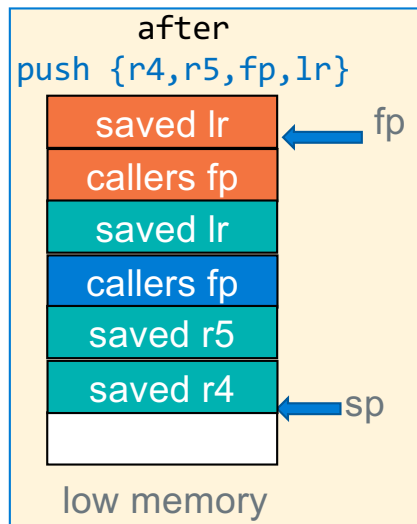
Move SP back to where it was after the push in the prologue.  
So, pop works properly (this also deallocates the local variables)

function  
Epilogue

```
sub    sp, fp, FP_OFF
pop    {fp, lr}
bx     lr
```

```
// deallocate local variables by moving sp
// pop (restore) fp and lr from stack
// return to caller
```

# Why You must move SP before POP in the Epilogue



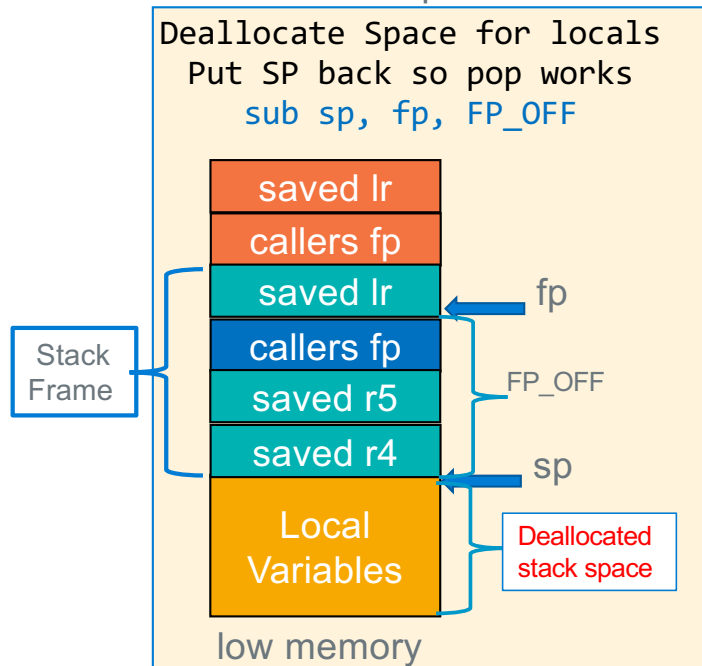
function  
Epilogue

```

sub    sp, fp, FP_OFF    // deallocate local variables by moving sp
pop    {fp, lr}          // pop (restore) fp and lr from stack
bx     lr                // return to caller
    
```

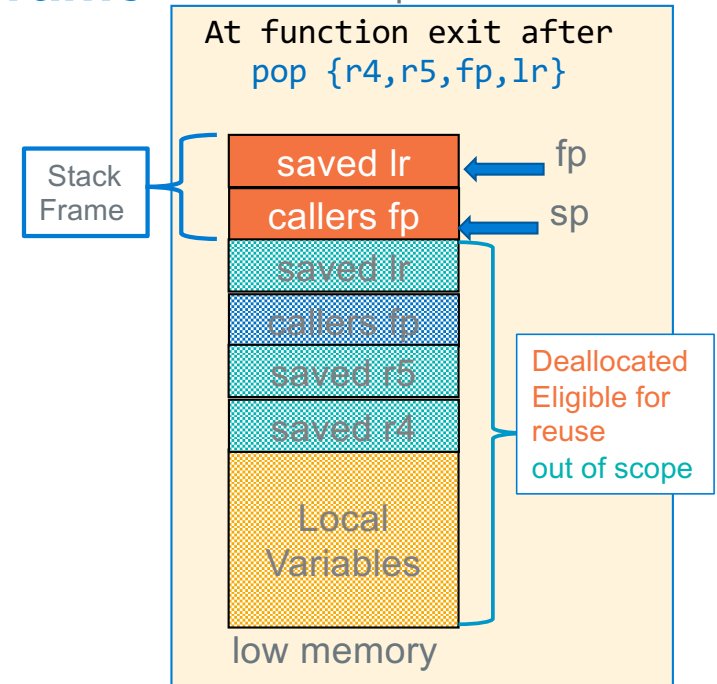
# Function Epilogue: Deallocating the Stack Frame

## Step 1



Move SP back to where it was after the push in the prologue.  
So, pop works properly (this also deallocates the local variables)

## Step 2



Use `pop` to restore the registers to the values they had at function entry

function  
Epilogue

```
sub    sp, fp, FP_OFF    // deallocate local variables by moving sp
pop    {fp, lr}          // pop (restore) fp and lr from stack
bx     lr                // return to caller
```

# How to Set FP

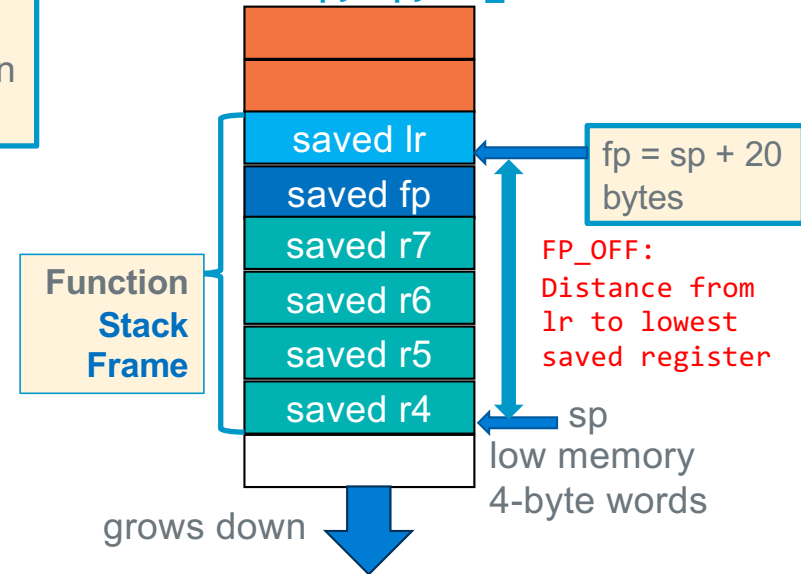
```
// other code etc
.equ    FP_OFF, 20

main:
  push   {r4-r7, fp, lr}
  add    fp, sp, FP_OFF
  .....
  sub    sp, fp, FP_OFF
  pop    {r4-r7, fp, lr}
  bx     lr
```

**Function Prologue**  
always at top of function  
saves regs and **sets fp**

**Function Epilogue**  
always at bottom of function  
**restores** regs including the sp

after push {r4-r7, fp, lr}  
add fp, sp, FP\_OFF



$$FP\_OFF = (\#regs\ saved - 1) * 4$$



**Means Caution, odd number of saved regs!**  
If odd number pushed, make sure frame is 8-byte aligned (later)  
this must always be true: **sp % 8 == 0**

# regs saved	FP_OFF in Bytes Distance from lr to lowest saved register
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

## Reference Table: Global Variable access

var	global variable address into r0 (lside)	global variable contents into r0 (rside)	contents of r0 into global variable
x	ldr r0, =x	ldr r0, =x ldr r0, [r0]	ldr r1, =x str r0, [r1]
*x	ldr r0, =x ldr r0, [r0]	ldr r0, =x ldr r0, [r0] ldr r0, [r0]	ldr r1, =x ldr r1, [r1] str r0, [r1]
**x	ldr r0, =x ldr r0, [r0] ldr r0, [r0]	ldr r0, =x ldr r0, [r0] ldr r0, [r0] ldr r0, [r0]	ldr r1, =x ldr r1, [r1] ldr r1, [r1] str r0, [r1]
stderr	ldr r0, =stderr	ldr r0, =stderr ldr r0, [r0]	<do not write unless you really know what you are doing>
.Lstr	ldr r0, =.Lstr	ldr r0, =.Lstr ldrb r0, [r0]	<read only>

```
.bss // from libc
stderr:.space 4 // FILE *
```

```
.data
x: .data y //x = &y
```

```
.section .rodata
.Lstr: .string "HI\n"
```

stdin, stdout and stderr are global variables

## Assembler Directives: Label Scope Control (Normal Labels only)

```
.extern printf
.extern fgets
.extern strcpy
.global fbuf
```

**.extern** <label>

- **Imports** label (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

**.global** <label>

- **Exports** label (or symbol) to be visible outside the source file boundary (other assembly or c source)
  - label is either a function name or a global variable name
  - Only use with function names or static variables
- **Without** .global, labels are usually (depends on the assembler) **local to the file**

## Passing global variables as a parameter: fprintf()

- `r0 = function(r0, r1, r2, r3)`  
`fprintf(stderr, "arg2", arg3, arg4)`
- create a literal string for `arg2` which tells `fprintf()` how to interpret the remaining arguments
- `stdin`, `stdout`, `stderr` are all **global variable** and are **part of libc**
  - these **names are their lside (label names)**
  - **get their contents** and pass that to `fprintf()`, `fread()`, `fwrite()`

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
```

```
    int a = 2;
    int b = 3;
    int c;
```

We are going to  
put these  
variables in  
temporary  
registers

```
    c = a + b;
    fprintf(stderr, "c=%d\n", c);
```

`r0, r1, r2`

```
    return EXIT_SUCCESS;
}
```

three passed  
args in this  
use of `fprintf`

```
.extern fprintf           //declare fprintf
.section .rodata         // note the dots "."
.Lfst: .string "c=%d\n"
```

// part of the **text segment** below

```
mov    r2, 2              // int a = 2;
mov    r3, 3              // int b = 3;
add    r2, r2, r3         // arg 3: int c = a + b;

ldr    r0, =stderr        // get stderr address
ldr    r0, [r0]           // arg 1: get stderr contents
ldr    r1, =.Lfst         // arg 2: =literal address
bl     fprintf
```

## Example: using preserved registers for local variables

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
```

**You must assume that**  
both getchar() and  
putchar() alter r0-r3

```
    int c; // use r0
    int count = 0; // use r4
```

r0

```
    while ((c = getchar()) != EOF) {
```

```
        putchar(c);
        count++;
```

```
    }
    printf("Echo count: %d\n", count);
    return EXIT_SUCCESS;
}
```



```
.extern getchar
.extern putchar
.section .rodata
.Lst: .string "Echo count: %d\n"
```

```
.text
.type main, %function
.global main
.equ EOF, -1
.equ FP_OFF, 12
.equ EXIT_SUCCESS, 0
```

main:

```
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    mov     r4, 0 //r4 = count
```

/\* while loop code will go here \*/

```
    mov     r0, EXIT_SUCCESS
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
    .size main, (. - main)
```



## Putchar/getchar: The while loop

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n", count);
    return EXIT_SUCCESS;
}
```

initialize count

pre loop test with a call to getchar()  
if it returns EOF in r0 we are done

echo the character read with getchar and  
then read another and increment count

did getchar() return EOF if not loop

saw EOF, print count

```
mov    r4, 0    //count
bl     getchar
cmp    r0, EOF
beq    .Ldone
```

.Lloop:

```
bl     putchar
bl     getchar
add    r4, r4, 1
cmp    r0, EOF
bne    .Lloop
```

.Ldone:

```
mov    r1, r4    //arg2
ldr    r0, =.Lst //arg1
bl     printf
```

address of string literal variable

.Lst: .string "Echo count: %d\n"

**File header and footers are not shown**

# Accessing Pointers (argv) in ARM assembly

```

.extern printf
.extern stderr
.section .rodata
.Lstr: .string "argv[%d] = %s\n"
.text
.global main // main(r0=argc, r1=argv)
.type main, %function
.equ FP_OFF, 20
main:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF
    mov     r7, r1 // save argv!
    ldr     r4, =stderr // get the address of stderr
    ldr     r4, [r4] // get the contents of stderr
    ldr     r5, =.Lstr // get the address of .Lstr
    mov     r6, 0 // set indx = 0;

.Lloop:
    // fprintf(stderr, "argv[%d] = %s\n", indx, *argv)
    ldr     r3, [r7] // arg 4: *argv
    cmp     r3, 0 // check *argv == NULL
    beq     .Ldone // if so done
    mov     r2, r6 // arg 3: indx
    mov     r1, r5 // arg 2: "argv[%d] = %s\n"
    mov     r0, r4 // arg 1: stderr
    bl      fprintf
    add     r6, r6, 1 // indx++ for printing
    add     r7, r7, 4 // argv++ pointer
    b       .Lloop

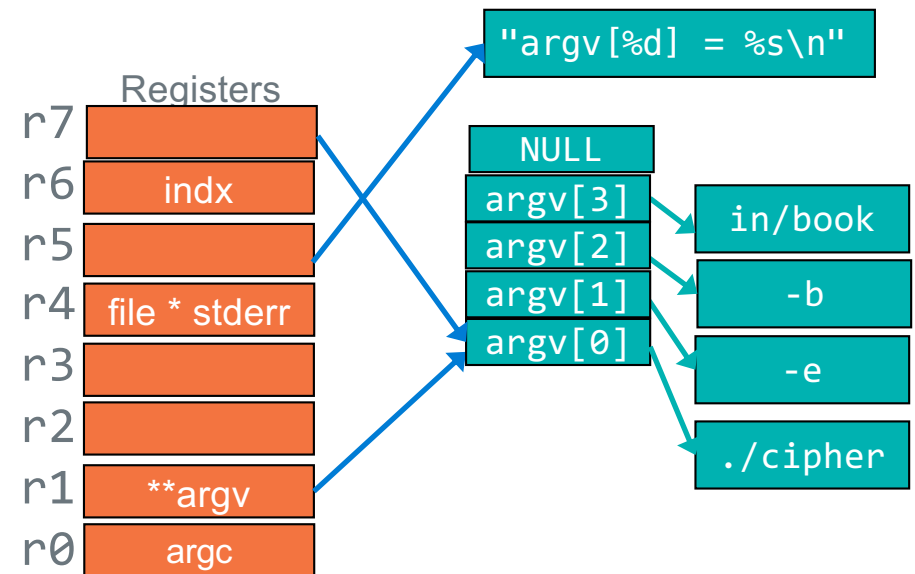
.Ldone:
    mov     r0, 0
    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx      lr
    
```

need to save r1 as  
we are calling a  
function - fprintf

```

% ./cipher -e -b in/B00K
argv[0] = ./cipher
argv[1] = -e
argv[2] = -b
argv[3] = in/B00K
    
```

r0-r3 lost due to fprintf call



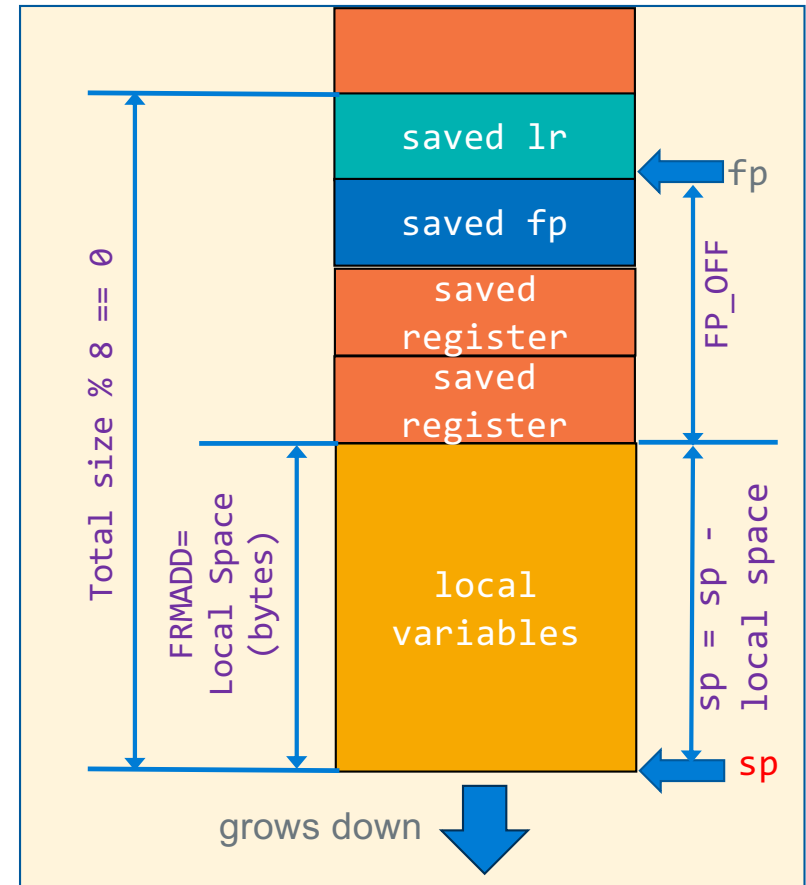
```
fprintf(stderr, "argv[%d] = %s\n", indx, *argv);
```

# Allocating Space For Locals on the Stack

- Space for local variables is allocated on the stack right below the lowest pushed register
  - Move the **sp** towards low memory by the total size of all local variables in bytes **plus padding**

$FRMADD = \text{total local var space (bytes)} + \text{padding}$

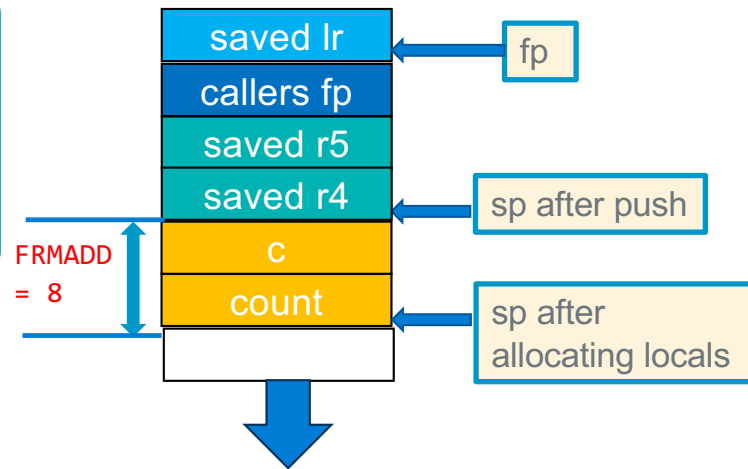
- Allocate the space after the register push by  
`add sp, sp, -FRMADD`
- Requirement:** on function entry, **sp** is always 8-byte aligned  
 $sp \% 8 == 0$
- Padding (as required):**
  - Additional space between variables on the stack to meet memory alignment requirements
  - Additional space so the frame size is evenly divisible by 8
- fp** (frame pointer) is used as a **pointer (base register)** to access all stack variables – later slides



# Local Variables on the stack

after push {r4-r5,fp,lr}  
add fp, sp, FP\_OFF

```
int main(void)
{
    int c;
    int count = 0;
    // rest of code
}
```



```
.text
.type    main, %function
.global  main
.equ     FP_OFF,    12
.equ     FRMADD,    8
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // but we are not done yet!
```

```
// when FRMADD values fail to assemble
ldr r3, =-FRMADD
add sp, sp, r3
```

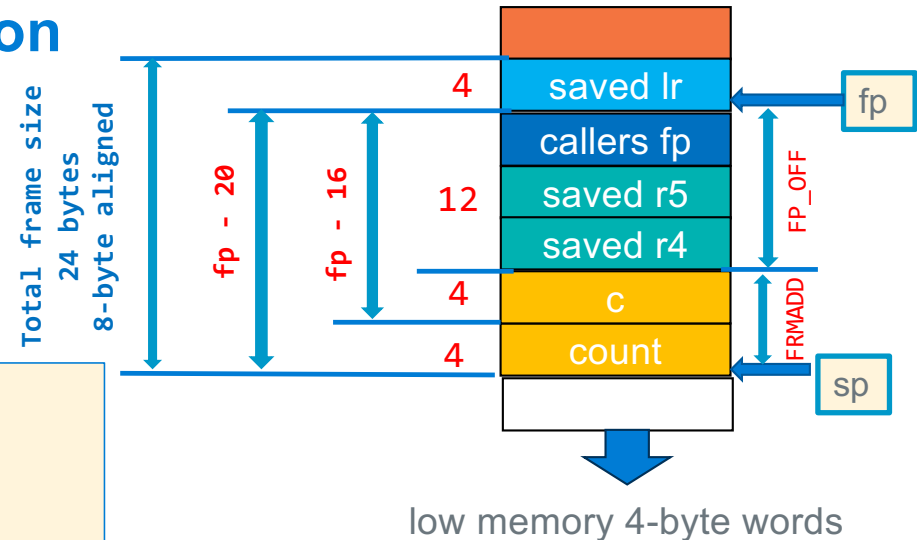
- In this example we are **allocating two variables on the stack**
- When writing assembly functions, in many situations **you may choose allocate these to registers instead**

- Add space on the stack for each local
  - we will allocate space in same order the locals are listed the C function shown from high to low stack address
  - gcc compiler allocates from low to high stack addresses
  - Order does not matter for our use

# Accessing Stack Variables: Introduction

```
int main(void)
{
    int c;
    int count = 0;
    // rest of code
}
```

- To Access data stored in the stack
  - use the `ldr/str` instructions
- Use register `fp` with offset (**distance in bytes**) addressing (use either register offset or immediate offset)
- No matter what address the stack frame is at**, `fp` always points at saved `lr`, so you can find a local stack variable by using an offset address from the contents of `fp`

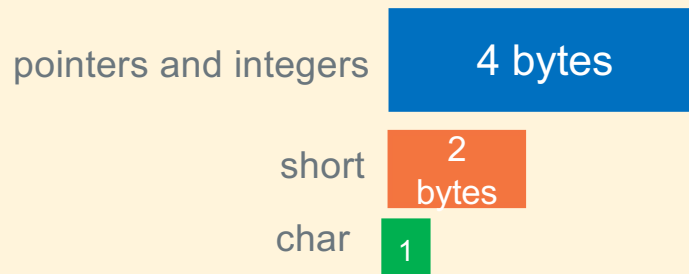


```
.text
.type    main, %function
.global  main
.equ     FP_OFF,    12
.equ     FRMADD,    8
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // but we are not done yet!
```

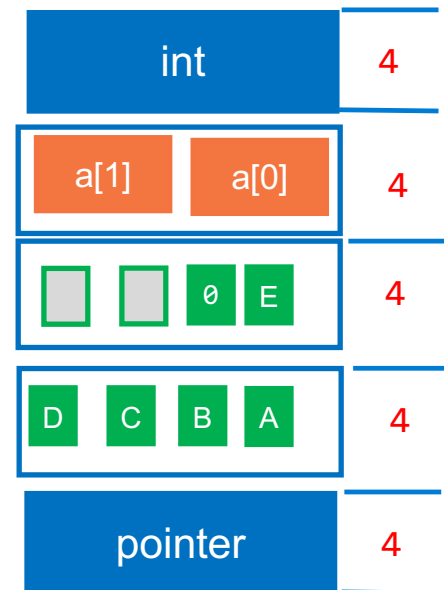
Variable	distance from fp	Read variable	Write Variable
int c	-16	ldr r0, [fp, -16]	str r0, [fp, -16]
int count	-20	ldr r0, [fp, -20]	str r0, [fp, -20]

# Stack Frame Design – Local Variables

- When writing an ARM equivalent for a C program, for CSE30 we will not re-arrange the order of the variables to optimize space (covered in the compiler course)
- **Arrays** start at a 4-byte boundary (even arrays with only 1 element)
  - Exception: double arrays [ ] start at an 8-byte boundary
  - **struct** arrays are **aligned to the requirements of largest member**
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- Padding may be required (see next slide)

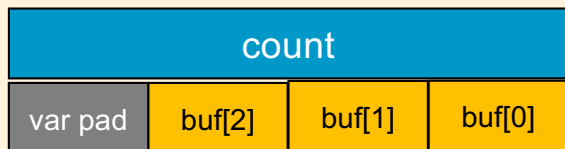


**Rule:** When the function is entered the stack is already 8-byte aligned

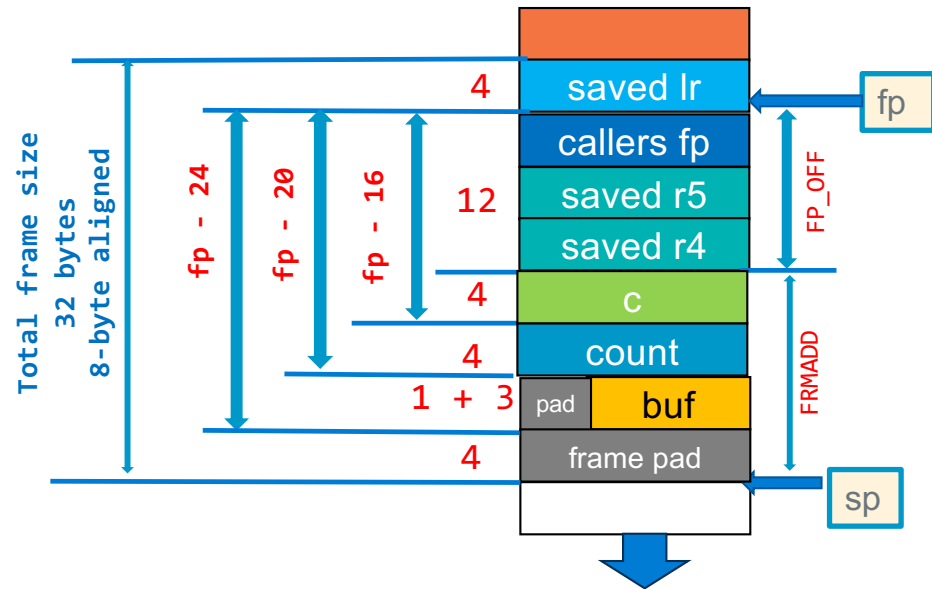
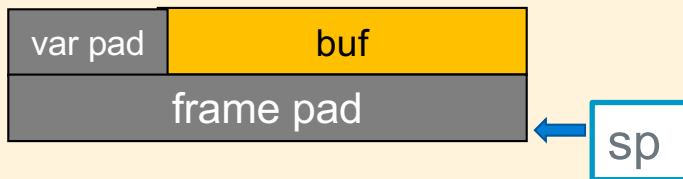


# Stack Variables: Padding

- **Variable padding** – start arrays at 4-byte boundary and **leave unused space at end** (high side address) before the variable higher on the stack



- **Frame padding** – add space below the last local variable to keep 8-byte alignment



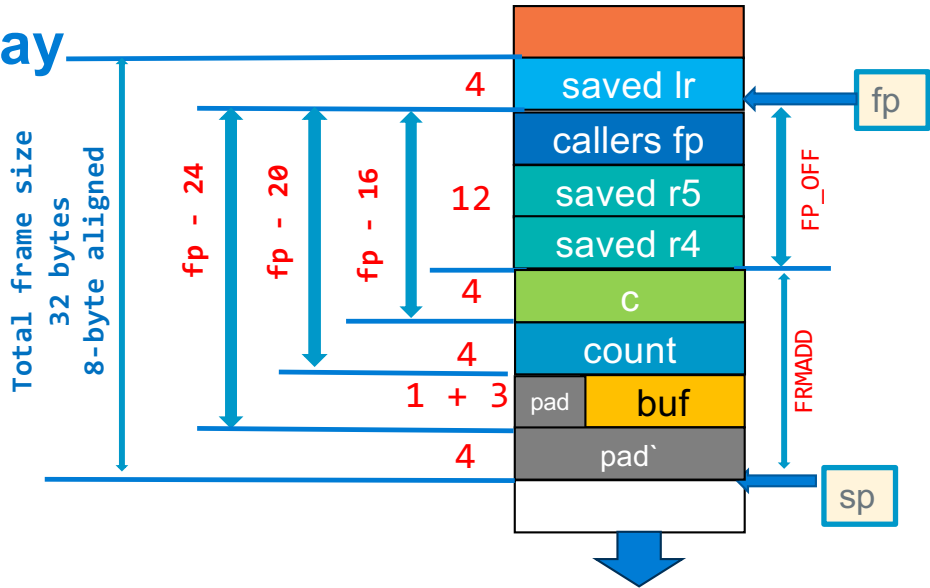
```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
.text
.type    main, %function
.global main
.equ     FP_OFF,    12
.equ     FRMADD,    16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // but we are not done yet!
```

# Accessing Stack Variables, the hard way

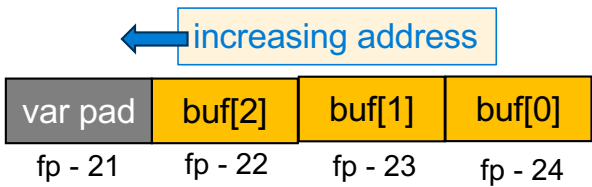
```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
.text
.type    main, %function
.global  main
.equ     FP_OFF,    12
.equ     FRMADD,    16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // but we are not done yet!
```



char buf[ ] by usage with ASCII chars we will use strb (or make it unsigned char)

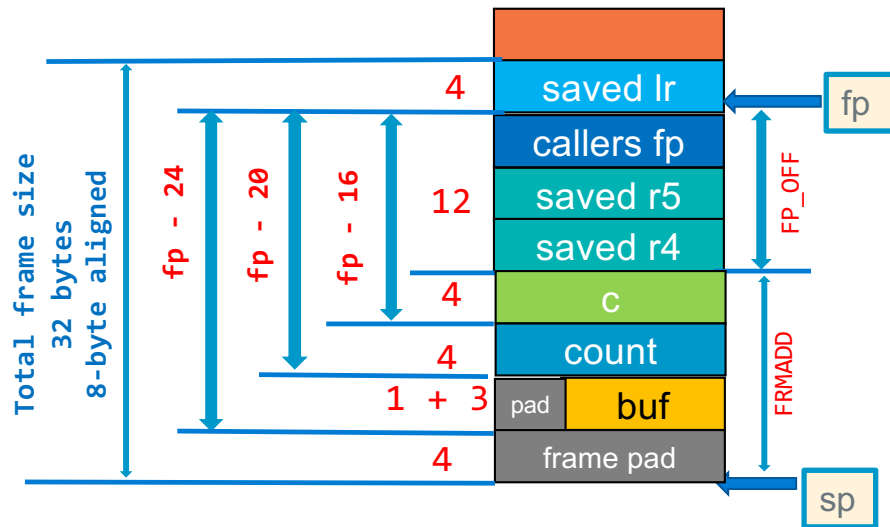
Variable	distance from fp	Read variable	Write Variable
int c	16	ldr r0, [fp, -16]	str r0, [fp, -16]
int count	20	ldr r0, [fp, -20]	str r0, [fp, -20]
char buf[0]	24	ldrb r0, [fp, -24]	strb r0, [fp, -24]
char buf[1]	23	ldrb r0, [fp, -23]	strb r0, [fp, -23]
char buf[2]	22	ldrb r0, [fp, -22]	strb r0, [fp, -22]



- Calculating offsets is a lot of work to get it correct
- It is also hard to debug
- There is a better way!



## Best Practice: Assembler Generated FP Distance Table



### FP Distance Table one For each function

```
.type    main, %function
.global  main
.equ     FP_OFF, 12

.equ     C, 4 + FP_OFF
.equ     COUNT, 4 + C
.equ     BUF, 4 + COUNT
.equ     PAD, 4 + BUF
.equ     FRMADD, PAD - FP_OFF

// FRMADD = 28 - 12 = 16
```

Annotations in the diagram:

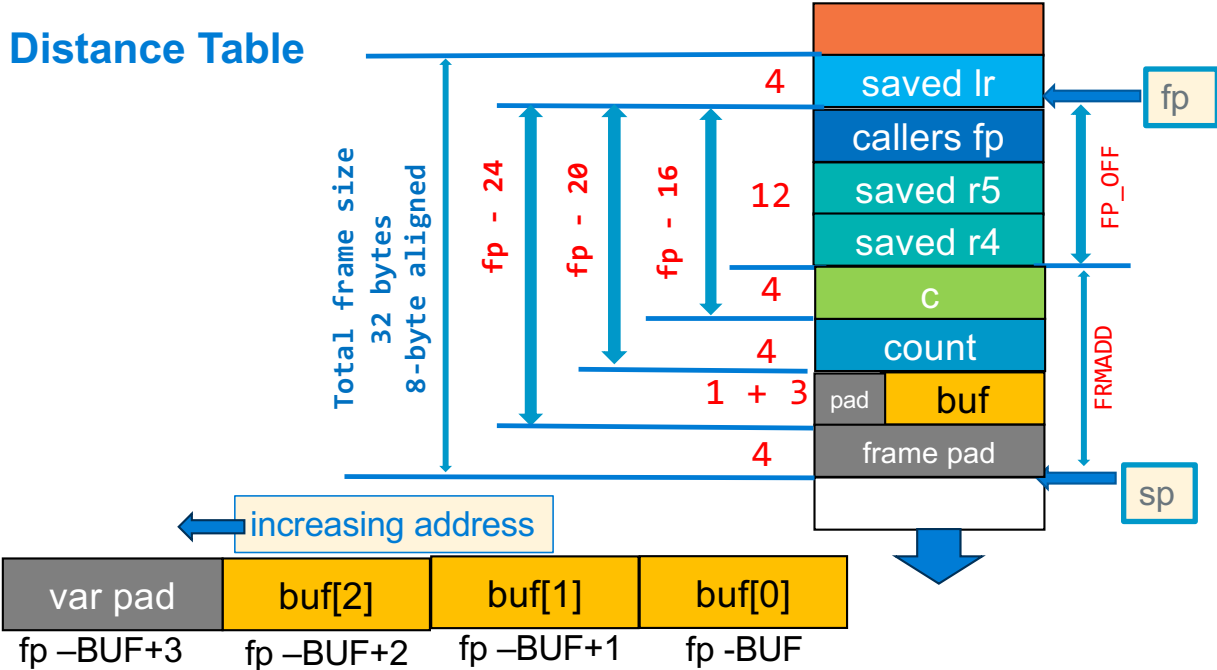
- pushed reg fp distance**: Points to the `FP_OFF` symbol.
- variable size in bytes**: Points to the `4` in the `.equ C, 4 + FP_OFF` line.
- Prior allocation distance**: Points to the `FP_OFF` symbol.

1. For each stack variable create a `.equ` symbol whose value is the distance in bytes from the FP after the prologue push
2. After the last variable add a name PAD for the size of the frame padding (if any) if no padding, PAD will be set to the same value as the variable above it
3. The value of the symbol is an expression that calculates the distance from the FP based on the distance of the variable above it on the stack. The first variable will use `SP_OFF` as the starting distance  
`.equ VAR, size_of var + variable_padding + previous_var_symbol` // previous\_var\_symbol distance of the var above
4. Calculate the size of the local variable area that needs to be added to the sp in bytes  
`FRMADD = distance PAD minus distance of the SP to the FP (FP_OFF) after the prologue push`

Best Practice: Assembler Generated FP Distance Table

FP Distance Table For each function

```
.type    main, %function
.global  main
.equ     FP_OFF,      12
.equ     C,           4 + FP_OFF
.equ     COUNT,       4 + C
.equ     BUF,         4 + COUNT
.equ     PAD,         4 + BUF
.equ     FRMADD,      PAD - FP_OFF
// FRMADD = 28 - 12 = 16
```



Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add r0, fp, -BUF+1	ldrb r0, [fp, -BUF+1]	strb r0, [fp, -BUF+1]
char buf[2]	BUF-2	add r0, fp, -BUF+2	ldrb r0, [fp, -BUF+2]	strb r0, [fp, -BUF+2]

# Initializing and Accessing Stack variables

```
.section .rodata
.Lmess: .string "%d %d %s\n"
.extern printf
.text
.type main, %function
.global main
.equ FP_OFF, 12
.equ C, 4 + FP_OFF
.equ COUNT, 4 + C
.equ BUF, 4 + COUNT
.equ PAD, 4 + BUF
.equ FRMADD, PAD - FP_OFF

main:
push {r4, r5, fp, lr}
add fp, sp, FP_OFF
add sp, sp, -FRMADD
// nothing to do for C
mov r2, 0
str r2, [fp, -COUNT]
strb r2, [fp, -BUF+2]
mov r2, 'h'
strb r2, [fp, -BUF]
mov r2, 'i'
strb r2, [fp, -BUF+1]

ldr r0, =.Lmess // arg1
ldr r1, [fp, -C] // arg2
ldr r2, [fp, -COUNT] // arg3
add r3, fp, -BUF // arg4
bl printf
```

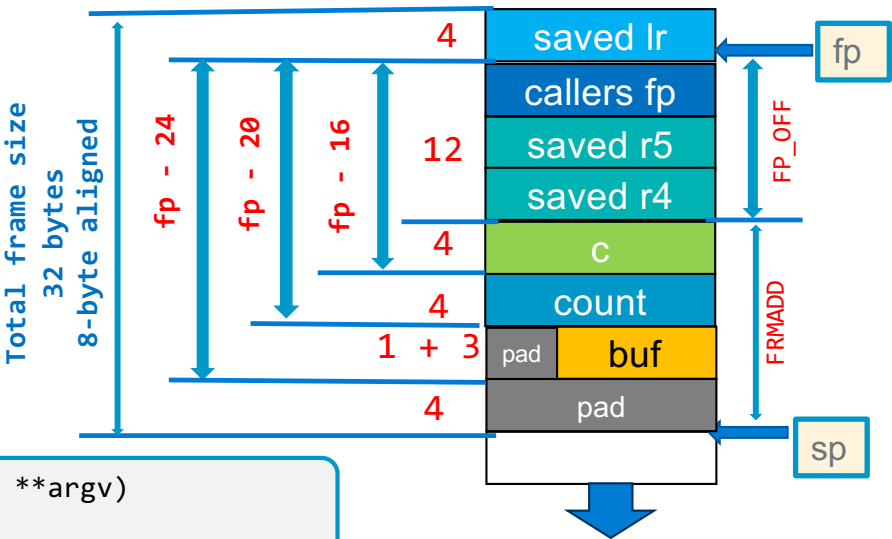
passes contents of stack var C

passes address of a stack variable buf

```
int main(int argc, char **argv)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    printf("%d %d %s\n", c, count, buf);
    // rest of code
}
```

pass stack address

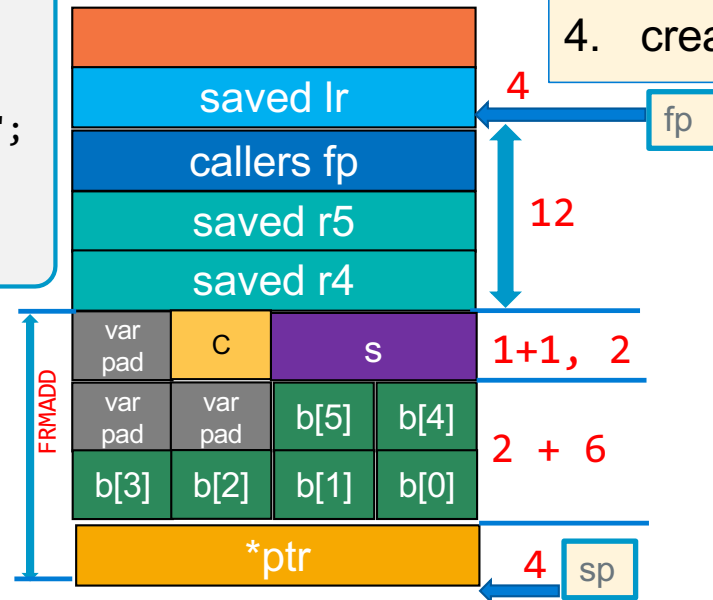
./a.out  
-136572160 0 hi



Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add r0, fp, -BUF+1	ldrb r0, [fp, -BUF+1]	strb r0, [fp, -BUF+1]
char buf[2]	BUF-2	add r0, fp, -BUF+2	ldrb r0, [fp, -BUF+2]	strb r0, [fp, -BUF+2]

# Stack Frame Design Practice

```
void func(void)
{
    signed char c;
    signed short s;
    unsigned char b[] = "Stack";
    unsigned char *ptr = &buf1
    // rest of code
}
```



1. Write the variables in C
2. Draw a picture of the stack frame
3. Write the code to generate the offsets
4. create the table to access the variables

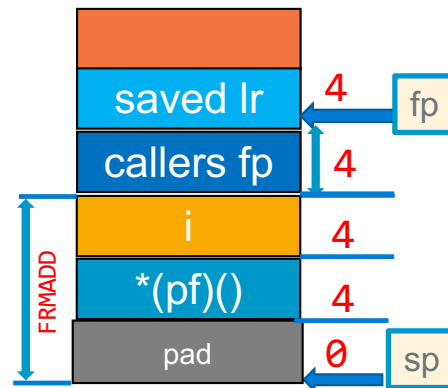
```
.equ    FP_OFF,      12
.equ    C,           2 + FP_OFF
.equ    S,           2 + C
.equ    B,           8 + S
.equ    PTR,         4 + BUF
.equ    PAD,         0 + PTR
.equ    FRMADD,      PAD - FP_OFF
// FRMADD = 28 - 12 = 16
```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
signed char c	C	add r0, fp, -C	ldrsb r0, [fp, -C]	strsb r0, [fp, -C]
signed short s	S	add r0, fp, -S	ldrsh r0, [fp, -S]	strsh r0, [fp, -S]
unsigned char b[0]	BUF	add r0, fp, -B	ldrb r0, [fp, -B]	strb r0, [fp, -B]
unsigned char *ptr	PTR	add r0, fp, -PTR	ldr r0, [fp, -PTR]	str r0, [fp, -PTR]

# Working with Pointers on the stack

```
int sum(int j, int k)
{
    return j + k;
}
void
testp(int j, int k, int (*func)(int, int),
      int *i)
{
    *i = func(j,k);
    return;
}
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```



```
.section .rodata
.Lmess: .string "%d\n"
.extern printf
.text
.global main
.type    main, %function
.equ    FP_OFF, 4
.equ    I,      4 + FP_OFF
.equ    PF,     4 + I
.equ    PAD,    0 + PF
.equ    FRMADD, PAD-FP_OFF
// FRMADD = 12 - 4 = 8
```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]

# Working with Pointers on the stack

```
int
sum(int j, int k)
{
    return j + k;
}

void
testp(int j, int k, int (*func)(int, int),
      int *i)
{
    *i = func(j,k);
    return;
}

int
main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

r0,r1,r2  
already set

```
.global sum
.type sum, %function
sum:
    push    {fp, lr}
    add     fp, sp, FP_OFF

    add     r0, r0, r1

    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
.size sum, (. - sum)
```

```
.global testp
.type testp, %function
.equ FP_OFF, 12
testp:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF

    mov     r4, r3          // save i
    blx     r2              // r0=func(r0,r1)
    str     r0, [r4]        // *i =r0

    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
.size testp, (. - testp)
```

```
.global main
.type main, %function
.equ FP_OFF, 4
.equ I,      4 + FP_OFF
.equ PF,     4 + I
.equ PAD,    0 + PF
.equ FRMADD, PAD-FP_OFF

main:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD

    ldr     r2, =sum        // func address
    add     r1, fp, -PF     // PF address
    str     r2, [r1]       // store in pf

    mov     r0, 1           // arg 1: 1
    mov     r1, 2           // arg 2: 2
    ldr     r2, [fp, -PF]   // arg 3: (*pf)()
    add     r3, fp, -I      // arg 4: &i
    bl      testp

    ldr     r0, =.Lmess     // arg 1: "%d\n"
    ldr     r1, [fp, -I]    // arg 2: i
    bl      printf

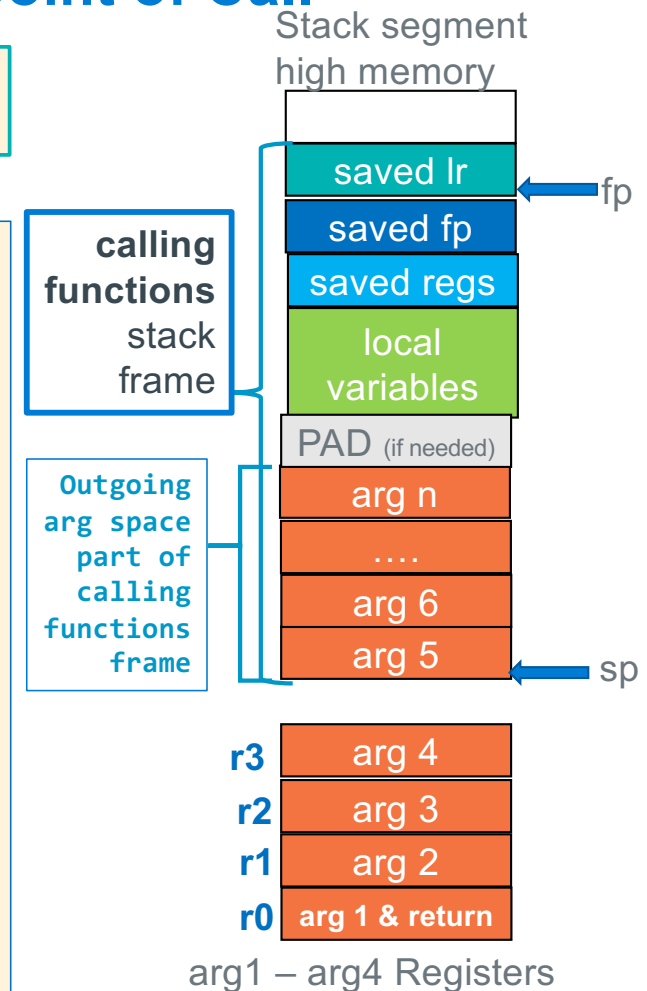
    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
.size main, (. - main)
```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]

# Passing More Than Four Arguments – At the point of Call

```
r0 = function(r0, r1, r2, r3, arg5, arg6, ... argn)
      arg1, arg2, arg3, arg4, ...
```

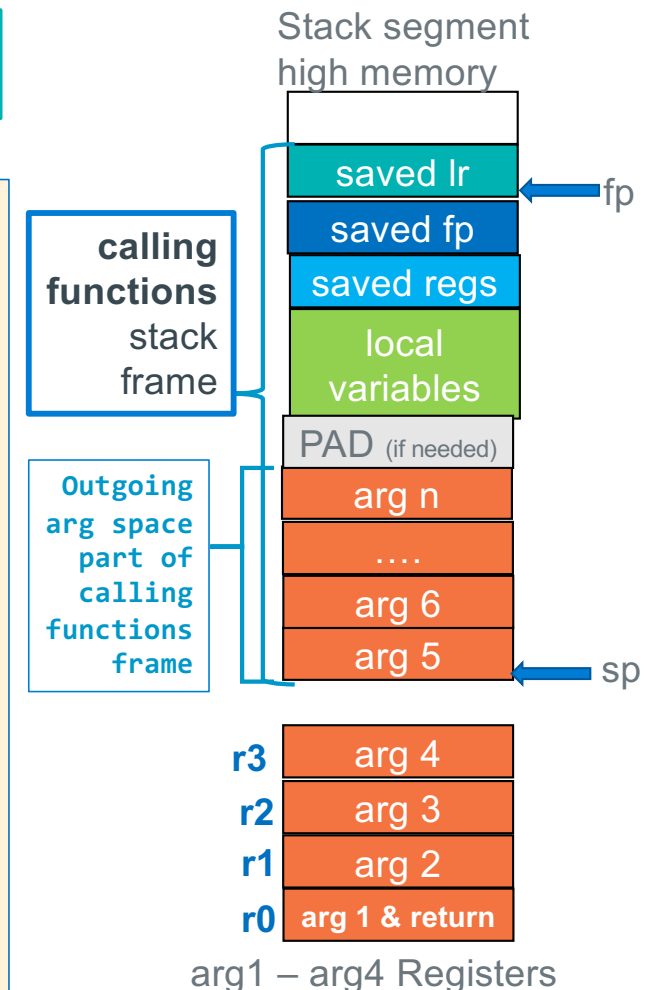
- **Approach: Increase stack frame size to include space for args# > 4**
  - Arg5 and above are in caller's stack frame at the **bottom of the stack**
- **Arg5** is always at the **bottom (at sp)**, arg6 and greater are above it
- **One arg value per slot!** – NO arrays across multiple slots
  - chars, shorts and ints are directly stored
  - Structs (not always), and arrays (always) are passed via a pointer
  - **Pointers** passed as **output parameters** contain an **address that points at** the **stack, BSS, data, or heap**
- Prior to any function call (and obviously at the start of the called function):
  1. sp must point at arg5
  2. sp and therefore **arg5** must be at an 8-byte boundary,
    - **Add padding** to force arg5 alignment if needed is **placed above** the last **argument the called function is expecting**



## Passing More Than Four Arguments – At the point of Call

```
r0 = function(r0, r1, r2, r3, arg5, arg6, ... argn)
      arg1, arg2, arg3, arg4, ...
```

- Called functions have the **right to change stack args** just like they can change the register args!
  - Caller **must always assume all args including ones on the stack are changed by the caller**
- Approach:** Extend the stack frame to include enough space for stack arguments for the called function that has the greatest number of args
  - Examine every function call in the body of a function
  - Find the function call with greatest arg count, this determines space needed for outgoing args
  - Add the greatest arg count space as needed to the frame layout
  - Adjust PAD as required to keep the sp 8-byte aligned
- In the calling function prior to making the call you must
  - Evaluate **first four args**: place the resulting values in r0-r3
  - Evaluate Arg 5 and greater and place the resulting values on the stack





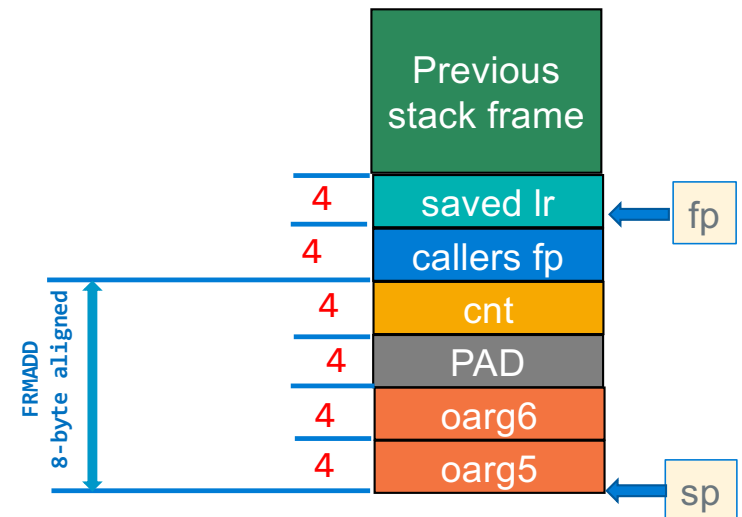
## Calling Function Stack Frame: Pass ARG 5 and higher

### Rules: At point of call

1. OARG5 must be pointed at by sp
2. SP must be 8-byte aligned

```
int cnt;
r0 = func(r0, r1, r2, r3, OARG5, OARG6);
```

```
.equ    FP_OFF, 4
.equ    CNT,      4 + FP_OFF    // int cnt
.equ    PAD,      4 + CNT      // added as needed
.equ    OARG6,    4 + PAD      // int
.equ    OARG5,    4 + OARG6    // int
.equ    FRMADD,   OARG5 - FP_OFF
// FRMADD = 20 - 4 = 16
```

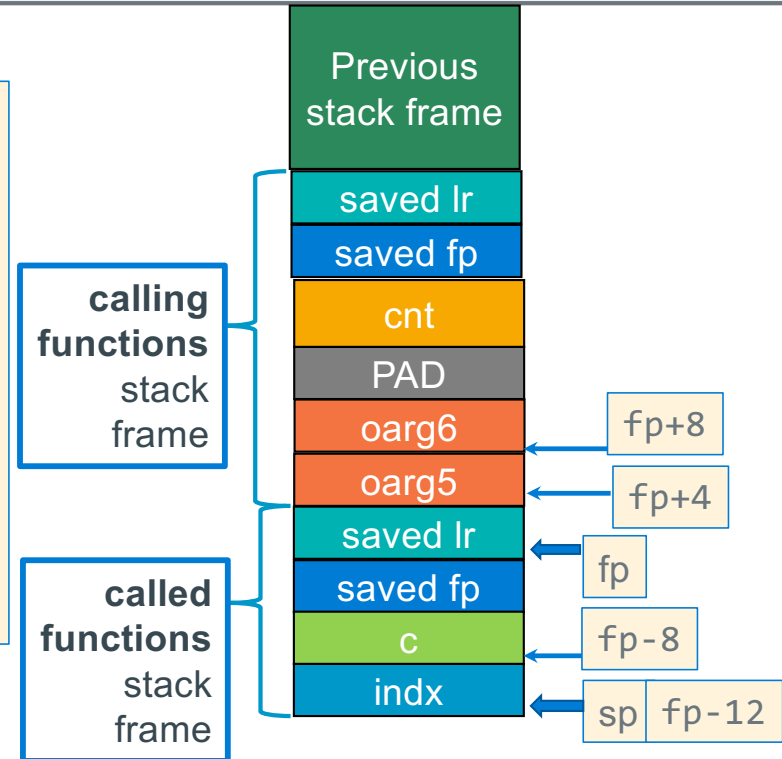


Variable	distance from fp	Address on Stack	Read variable	Write Variable
int cnt	CNT	add r0, fp, -CNT	ldr r0, [fp, -CNT]	str r0, [fp, -CNT]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

# Called Function: Retrieving Args From the Stack

```
r0 = func(r0, r1, r2, r3, r4, ARG5, ARG6);
```

- At function start and before the push{} the **sp** is at an 8-byte boundary
  - Args > 4 in caller's stack frame** and **arg 5** always starts at **fp+4**
    - Additional args are higher up the stack, with one "slot" every 4-bytes
- ```
.equ ARGN, (N-4)*4 // where n must be > 4
```
- This "algorithm" for finding args was designed to enable **variable arg count functions** like `printf("conversion list", arg0, ... argn);`
  - No limit to the number of args (except running out of stack space)



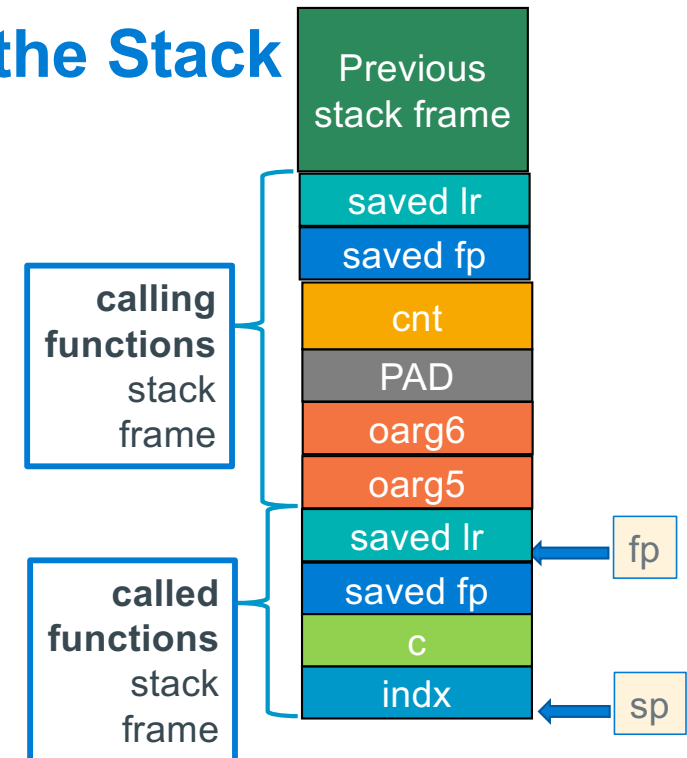
**Rule:**  
**Called functions** always access stack args using a **positive offset to the fp**

## Called Function: Retrieving Args From the Stack

```
.equ    FP_OFF,    4
.equ    C,         4 + FP_OFF
.equ    INDX,      4 + C
.equ    PAD,       4 + INDX
.equ    FRMADD,    PAD - FP_OFF
// below are distances to caller's stack frame
.equ    ARG6,      8
.equ    ARG5,      4
```

```
r0 = func(r0, r1, r2, r3, r4, ARG5, ARG6);
```

**Rule:**  
**Called functions** always access stack args  
 using a **positive offset to the fp**



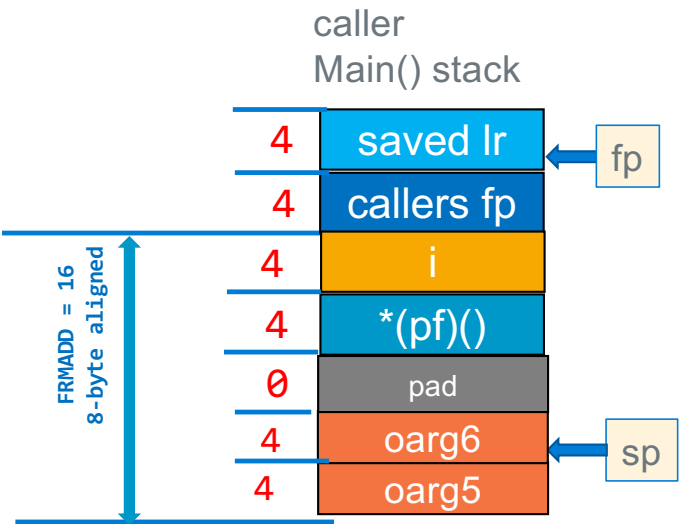
| Variable or Argument | distance from fp | Address on Stack  | Read variable       | Write Variable      |
|----------------------|------------------|-------------------|---------------------|---------------------|
| int arg6             | ARG6             | add r0, fp, ARG6  | ldr r0, [fp, ARG6]  | str r0, [fp, ARG6]  |
| int arg5             | ARG5             | add r0, fp, ARG5  | ldr r0, [fp, ARG5]  | str r0, [fp, ARG5]  |
| int c                | C                | add r0, fp, -C    | ldr r0, [fp, -C]    | str r0, [fp, -C]    |
| int count            | INDX             | add r0, fp, -INDX | ldr r0, [fp, -INDX] | str r0, [fp, -INDX] |

Observe the positive offsets

# Example: Passing Stack Args, Calling Function

```
int sum(int j, int k)
{
    return j + k;
}
void
testp(int j, int k, int l, int m, int (*func)(int, int),
      int *i)
{
    *i = func(j,k) + func(l, m);
    return;
}
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```



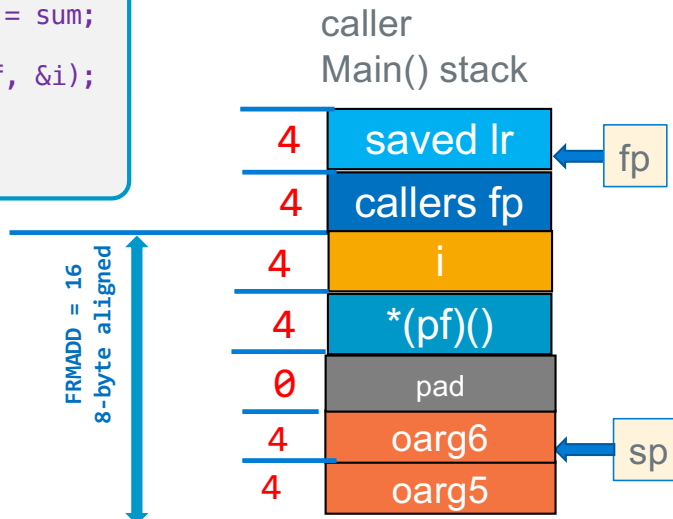
| Variable or Argument | distance from fp | Address on Stack   | Read variable        | Write Variable       |
|----------------------|------------------|--------------------|----------------------|----------------------|
| int i                | I                | add r0, fp, -I     | ldr r0, [fp, -I]     | str r0, [fp, -I]     |
| int (*pf)()          | PF               | add r0, fp, -PF    | ldr r0, [fp, -PF]    | str r0, [fp, -PF]    |
| int oarg6            | OARG6            | add r0, fp, -OARG6 | ldr r0, [fp, -OARG6] | str r0, [fp, -OARG6] |
| int oarg5            | OARG5            | add r0, fp, -OARG5 | ldr r0, [fp, -OARG5] | str r0, [fp, -OARG5] |

```
.equ    FP_OFF, 4
.equ    I,      4 + FP_OFF
.equ    PF,      4 + I
.equ    PAD,     0 + PF
.equ    OARG6,   4 + PAD
.equ    OARG5    4 + OARG6
.equ    FRMADD,  OARG5 - FP_OFF
// FRMADD = 20 - 4 = 16
```

## Example: Passing Stack Args, Calling Function

```
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```



| Variable or Argument | distance from fp | Address on Stack   | Read variable        | Write Variable       |
|----------------------|------------------|--------------------|----------------------|----------------------|
| int i                | I                | add r0, fp, -I     | ldr r0, [fp, -I]     | str r0, [fp, -I]     |
| int (*pf)()          | PF               | add r0, fp, -PF    | ldr r0, [fp, -PF]    | str r0, [fp, -PF]    |
| int oarg6            | OARG6            | add r0, fp, -OARG6 | ldr r0, [fp, -OARG6] | str r0, [fp, -OARG6] |
| int oarg5            | OARG5            | add r0, fp, -OARG5 | ldr r0, [fp, -OARG5] | str r0, [fp, -OARG5] |

```
.equ    FP_OFF, 4
.equ    I,      4 + FP_OFF
.equ    PF,      4 + I
.equ    PAD,     0 + PF
.equ    OARG6,   4 + PAD
.equ    OARG5,   4 + OARG6
.equ    FRMADD,  OARG5 - FP_OFF

main:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD

    ldr     r0, =sum           // get func address
    add     r1, fp, -PF        // PF address on stack
    str     r0, [r1]          // store sum in var pf

    add     r0, fp, -I         // get address of I
    add     r1, fp, -OARG6     // address of OARG6
    str     r0, [r1]          // store address of I in OARG6

    ldr     r0, [fp, -PF]      // get PF from stack
    add     r1, fp, -OARG5     // address of OARG5
    str     r0, [r1]          // store sum address in OARG5

    mov     r0, 1              // arg 1: 1
    mov     r1, 2              // arg 2: 2
    mov     r2, 3              // arg 3: 3
    mov     r3, 4              // arg 4: 4

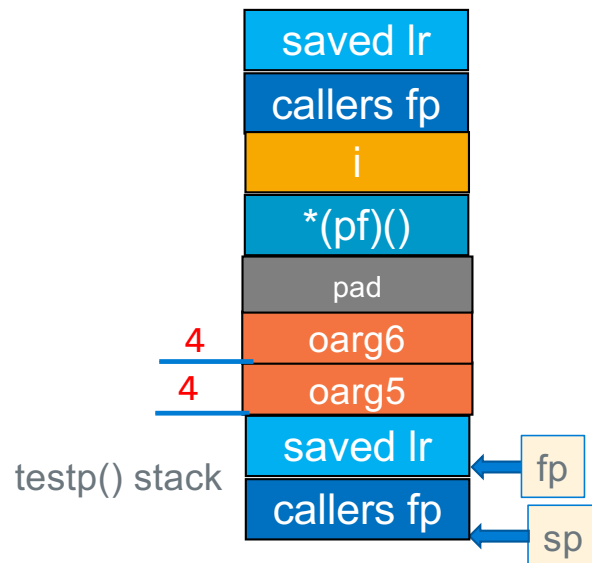
    bl      testp

    ldr     r0, =.Lmess        // arg 1: "%d\n"
    ldr     r1, [fp, -I]       // arg 2: i
    bl      printf

    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
```

# Example: Passing Stack Args, Called Function

```
void
testp(int j, int k, int l, int m, int (*func)(int, int), int *i)
{
    *i = func(j, k) + func(l, m);
    return;
}
```



```
.global testp
.type testp, %function
.equ FP_OFF, 20
.equ ARG6, 8
.equ ARG5, 4

testp:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF

    mov     r4, r2          // save l
    mov     r5, r3          // save m
    ldr     r6, [fp, ARG5]  // load func
    ldr     r7, [fp, ARG6]  // load i
    blx     r6              // r0=func(j, k)

    mov     r1, r5          // arg 2 saved m
    mov     r5, r0          // save func() return value
    mov     r0, r4          // arg 1 saved l
    blx     r6              // r0=func(l,m)
    add     r0, r0, r5      // func(l, m) + func(j, k)
    str     r0, [r7]        // store sum to *i

    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx     lr
```

| Argument    | distance from fp | Address on Stack | Read variable      | Write Variable     |
|-------------|------------------|------------------|--------------------|--------------------|
| int *i      | ARG6             | add r0, fp, ARG6 | ldr r0, [fp, ARG6] | str r0, [fp, ARG6] |
| int (*fp)() | ARG5             | add r0, fp, ARG5 | ldr r0, [fp, ARG5] | str r0, [fp, ARG5] |

## Extra Slides

# Determining Size of the Passed Parameter Area on The Stack

- Find the function called by main with the largest number of parameters
- That function determines the size of the Passed Parameter allocation on the stack

```
int main(void)
{
    /* code not shown */
    a(g, h);

    /* code not shown */
    sixsum(a1, a2, a3, a4, a5, a6);

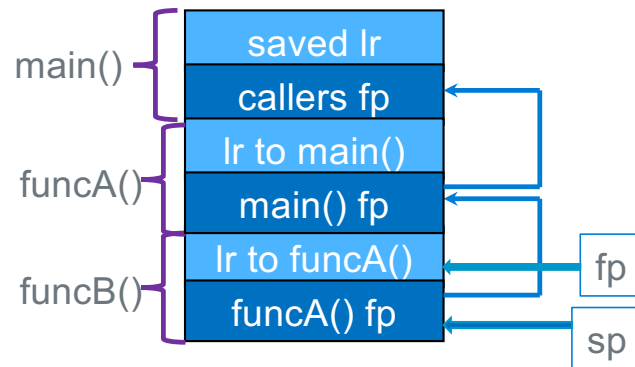
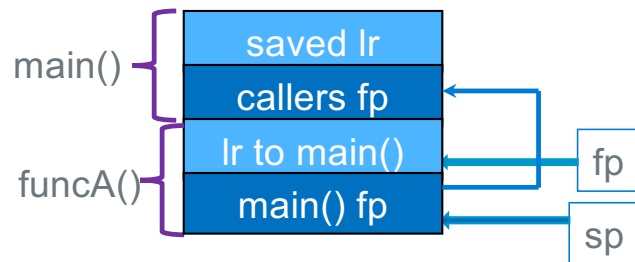
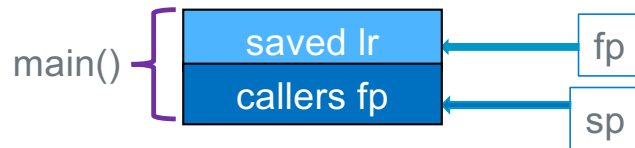
    /* code not shown */

    b(q, w, e, r);
    /* code not shown */
}
```

largest arg count is 6  
allocate space for  $6 - 4 = 2$  arg slots



## By following the saved fp, you can find each stack frame



How gdb finds stack frames

