Version 2.14

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

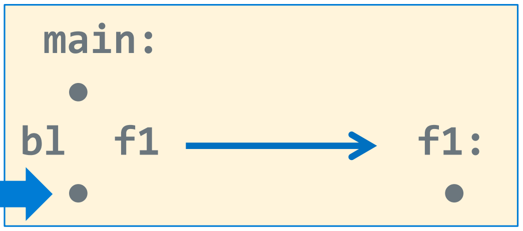
Branch with Link Indirect (function call) instruction

`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



x

Function Call Return

Branch & exchange (function return) instruction

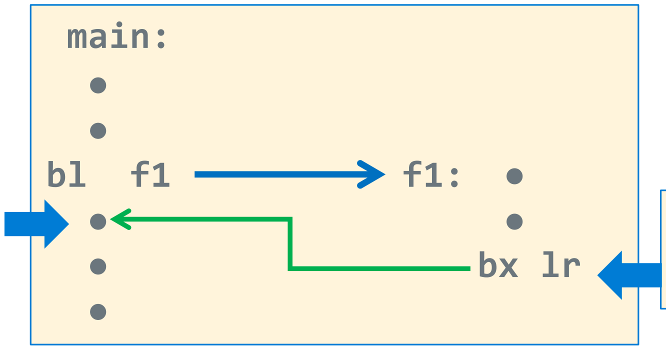
`bx lr`

bx	Rn

`// 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`**

Stores this address in `lr` this is the address to resume at in the caller

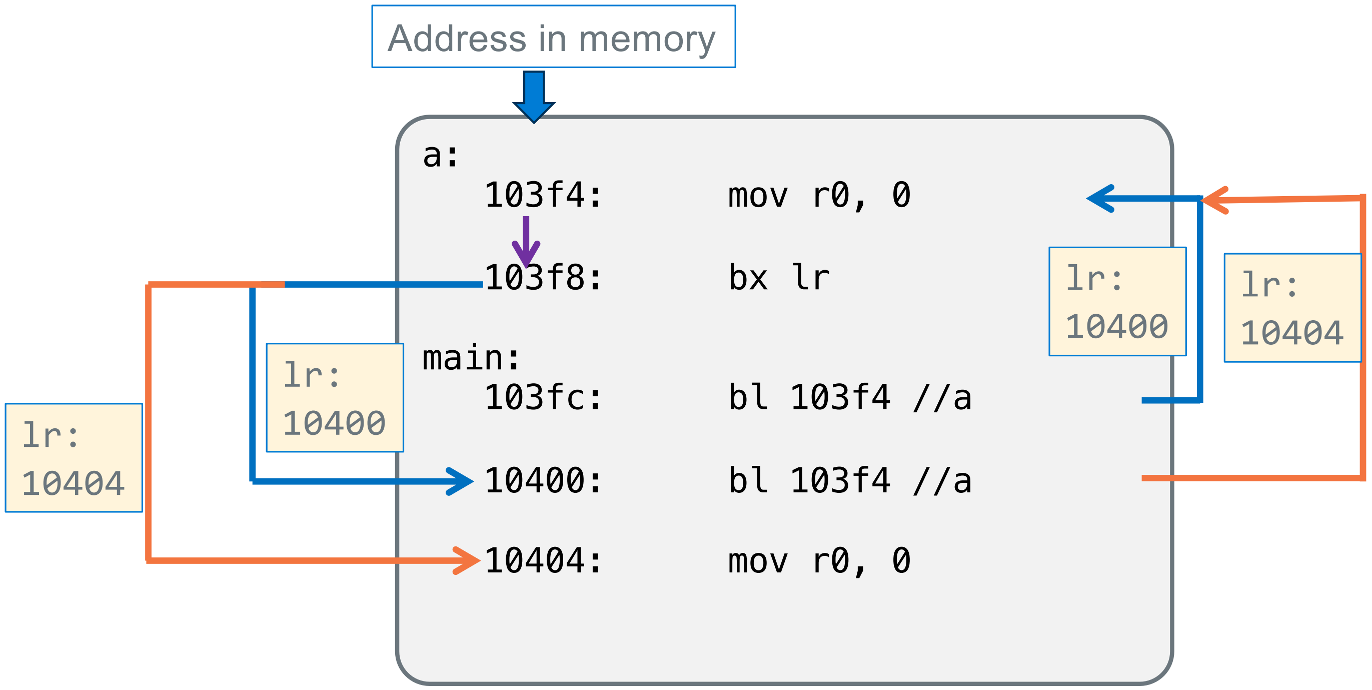


Branch to the instruction whose address is stored in lr

x

Understanding bl and bx - 1

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



```
a:
   103f4:      mov r0, 0
   103f8:      bx lr
main:
   103fc:      bl 103f4 //a
   10400:      bl 103f4 //a
   10404:      mov r0, 0
```

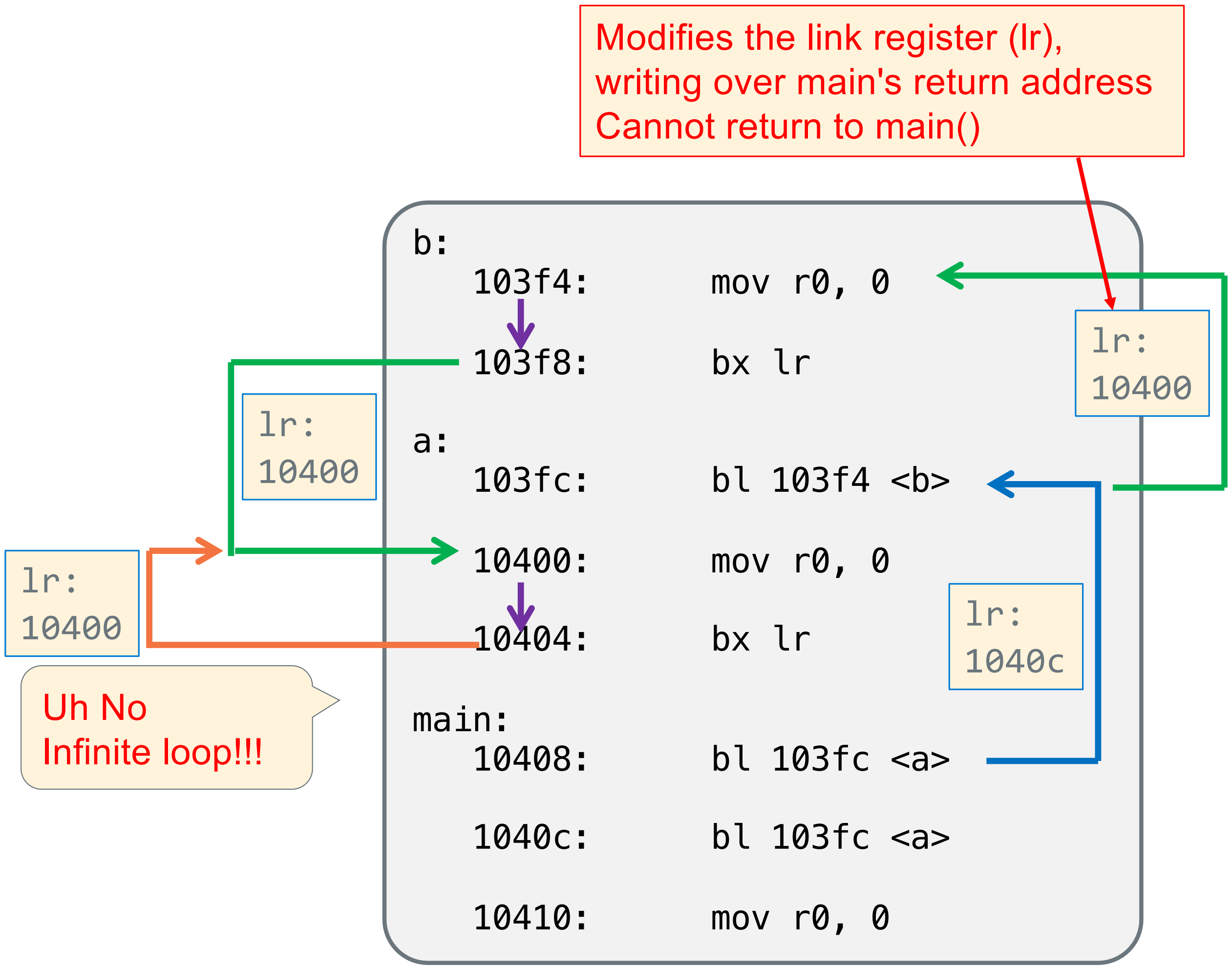
But there is a problem we must address here – next slide

x

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!



```
b:
    103f4:      mov r0, 0
    103f8:      bx lr
a:
    103fc:      bl 103f4 <b>
    10400:      mov r0, 0
    10404:      bx lr
main:
    10408:      bl 103fc <a>
    1040c:      bl 103fc <a>
    10410:      mov r0, 0
```

Modifies the link register (lr), writing over main's return address Cannot return to main()
lr: 10400
lr: 10400
lr: 10400
lr: 1040c
Uh No Infinite loop!!!

x

Understanding bl and blx - 3

```c
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!

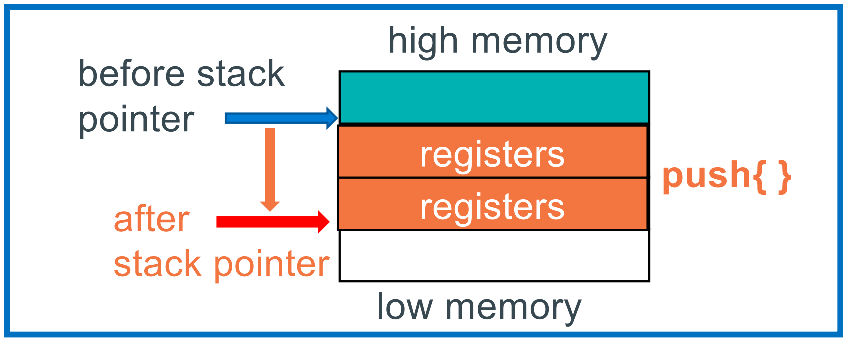
```
     .data
func:.word a // func initialized with address of a()

    .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!
```

x

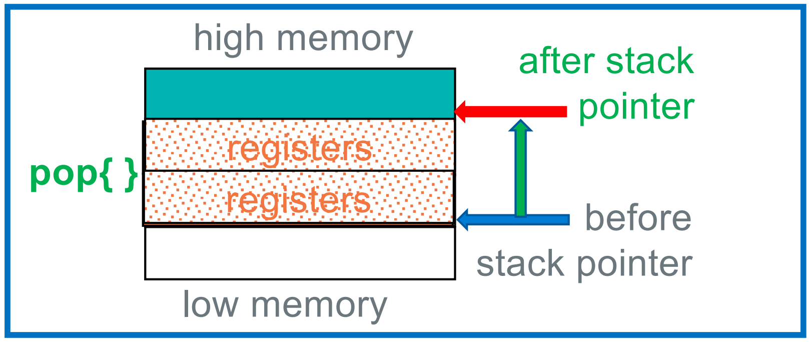
Preserving and Restoring Registers on the stack - 1

Operation	*Pseudo Instruction*	*Operation*
Push registers Function Entry	push *{reg list}*	sp = sp – 4 × #registers Copy registers to mem[sp]
Pop registers Function Exit	pop *{reg list}*	Copy mem[sp] to registers, sp = sp + 4 × #registers

push (multiple register `str` to memory operation)



push (multiple register `ldr` from memory operation)



x

Preserving and Restoring Registers on the Stack - 2

Operation	*Pseudo Instruction*	*Operation*

Push registers
Function Entry
	`push` *{reg list}*	`sp = sp – 4 × #registers`
		`Copy registers to mem[sp]`

Pop registers
Function Exit
	`pop` *{reg list}*	`Copy mem[sp] to registers,`
		`sp = sp + 4 × #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

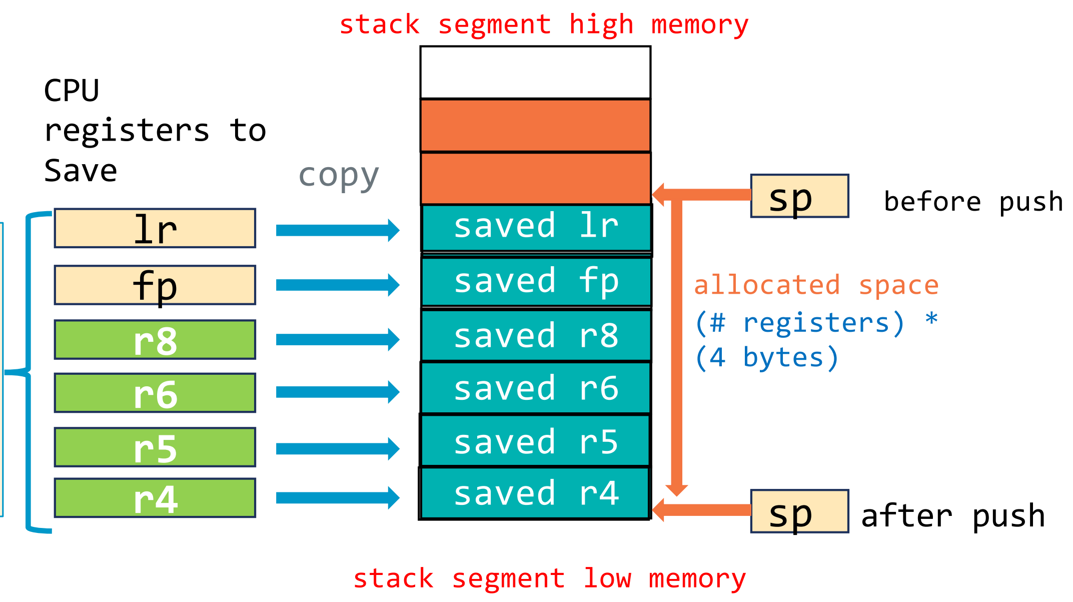
x

push: Multiple Register Save to the stack

```
save registers
push {r4-r6, r8, fp, lr}
```

Registers are pushed on to the stack *in order* **right (high memory) to left (low memory)**

If you have no stack variables (later slides) then always push an **EVEN** number of registers!



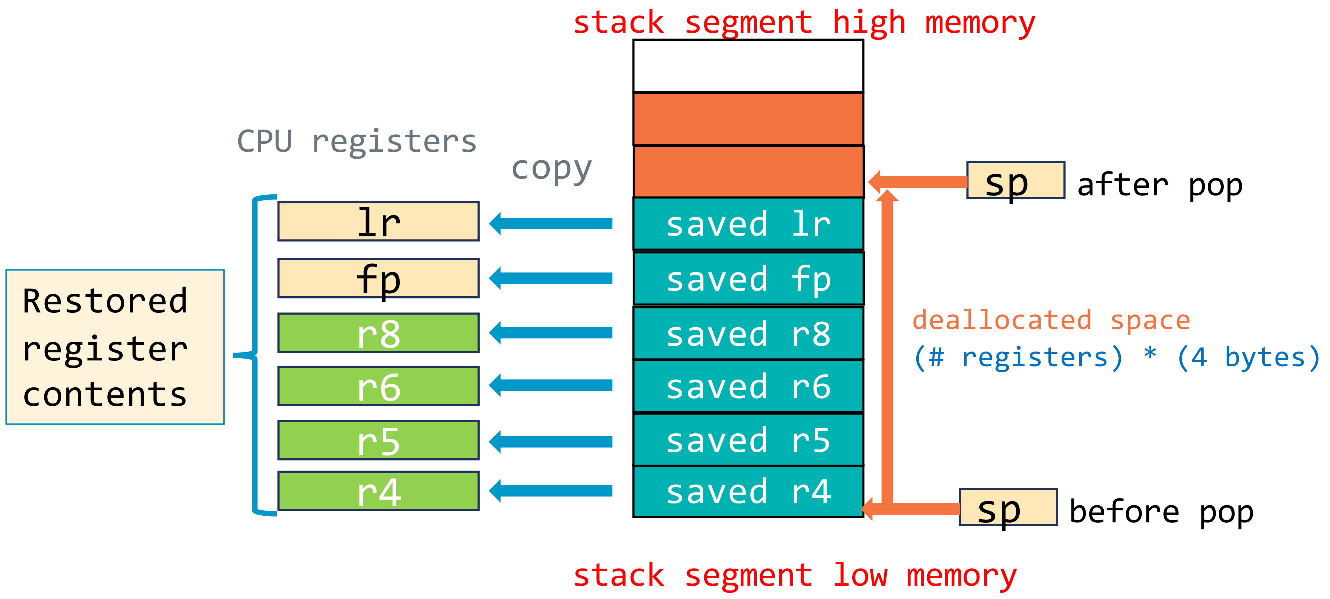
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** subtracts (# of registers saved) * (4 bytes) from the **sp** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

x

pop: Multiple Register Restore from the stack

```
restore registers
pop {r4-r6, r8, fp, lr}
```

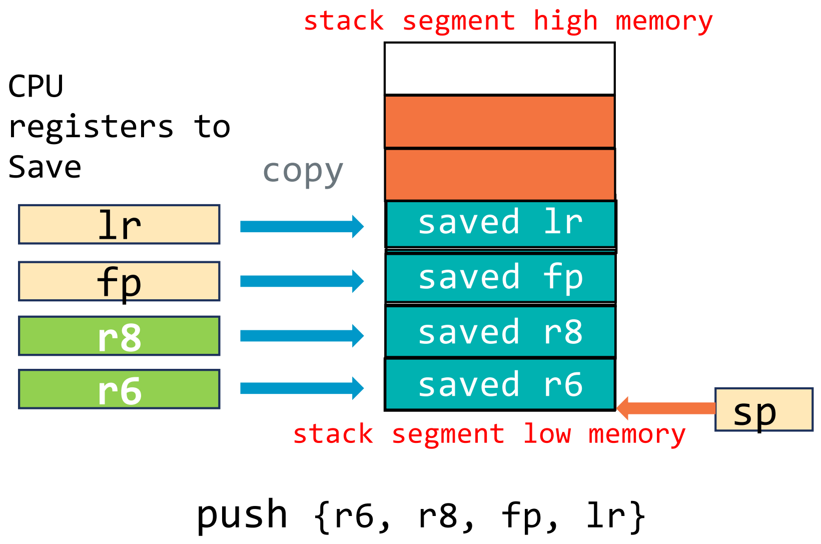
Registers are **pop'd** from the stack *in order* **left (low memory) to right (high memory)**

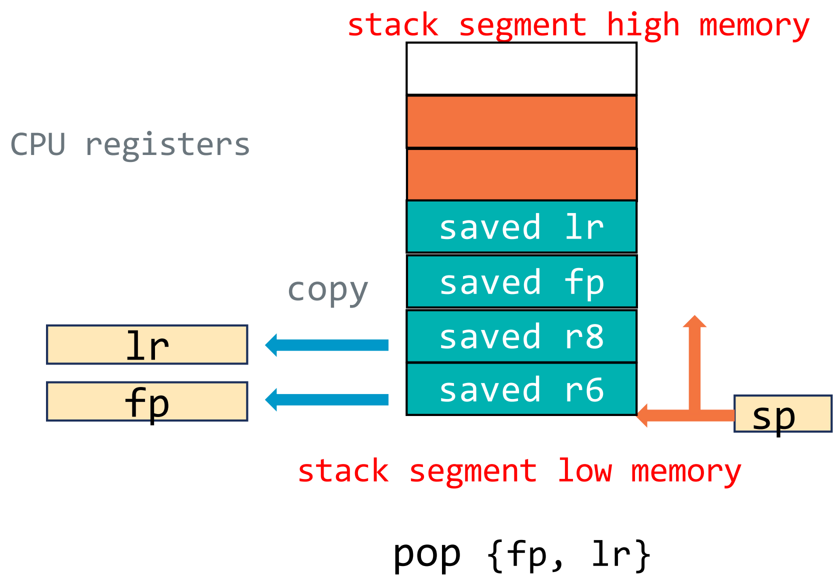


- **pop** copies the contents of stack segment memory to the `{reg list}`
- **pop <u>adds:</u>** (# of registers restored) * (4 bytes) to `sp` to ***deallocate*** space on the stack
  - sp = sp + (# registers restored * 4)
- **`Remember:`** `{reg list}` <u>must be the same</u> in both the `push` and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

x

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

x

Registers: Rules For Use

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
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**
- **Called function:**
  - you receive the first four parameters in these four registers (r0 – r3)

x

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
```

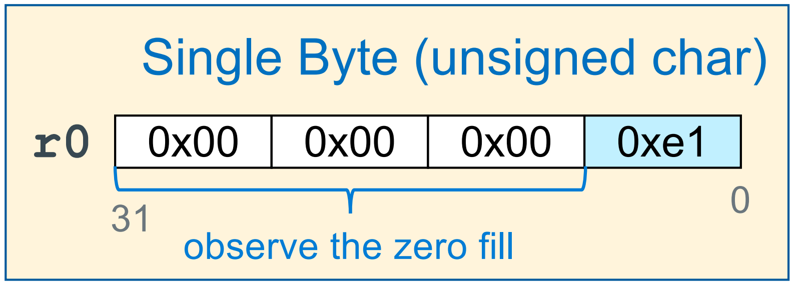
- For parameters, whose size is larger than 4 bytes, pass a pointer to the parameter (we will cover this later)
- **One arg value per register!** – NO arrays across multiple registers
  - chars, shorts and ints are directly stored
  - Structs (not always), and arrays (always) are passed via a pointer
  - **Pointers** passed as output parameters contain an address ***that points at*** the stack, BSS, data, or heap

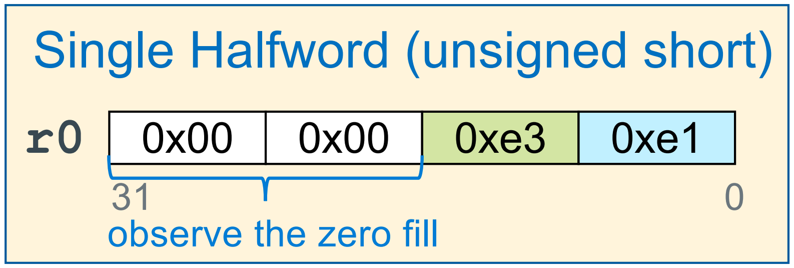
x

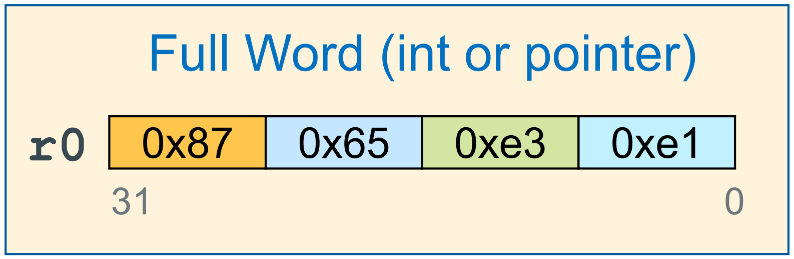
Register Arguments and Return Values

- When passing or returning values from a function you must do the following:

1. Make sure that the values in the registers r0-r3 are in their properly aligned position in the register **based on data type**
2. Upper bytes in byte and halfword values in registers r0-r3 when passing arguments and returning values are
   a. zero filled for unsigned values
   b. sign extended for signed values







x

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

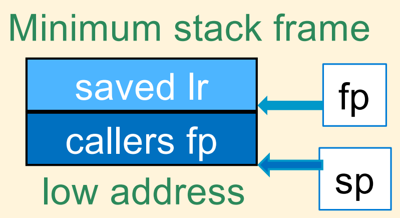
Register	***Function Call Use***	***Function Body Use***	***Save before use Restore before return***
r4-r10	preserved registers	contents preserved across function calls	Yes
r11/fp	stack frame pointer	Use to locate variables on the stack	Yes
r13/sp	stack pointer	stack space allocation	Yes
r14/lr	link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

x

Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

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

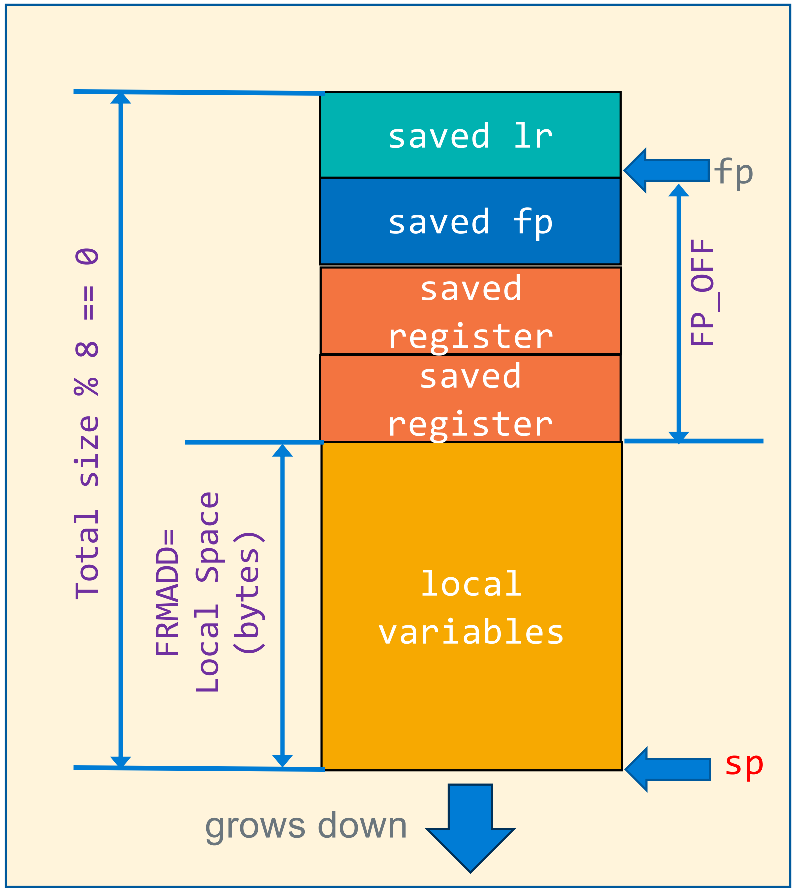


- `sp` always points at top element in the stack (lowest byte address)
- `fp` always points at the bottom element in the stack
    - Bottom element is always the saved `lr` (contains the return address of caller)
    - A saved copy of callers fp is always the next element below the lr
    - fp will be 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`)

x

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



x

Function Prologue and Epilogue

```
    .global myfunc
    .type   myfunc, %function
    .equ    FP_OFF, 4              // fp distance to sp after push
    .equ    FRAMDD, 8              // number of bytes for local stack vars

myfunc:

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

    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

    .size myfunc, (. - myfunc)
```

Function Prologue creates stack frame

Function Epilogue removes stack frame

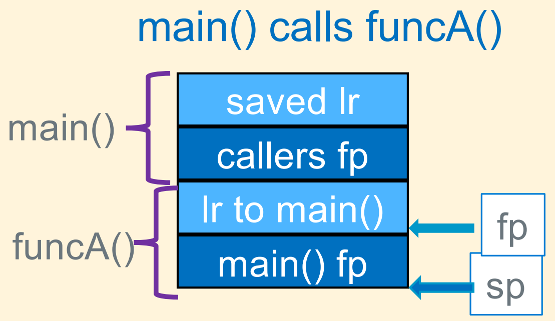
- **Only one prologue** right after the function label (name)
- **Only one epilogue** at the bottom of the function right above the .size directive

x

Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

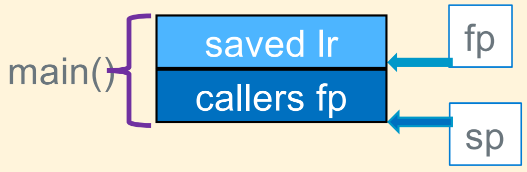
- Function entry (Function **Prologue**):
  1. save lr and fp registers (push)
  2. set fp to top entry in stack
  3. allocate space for local vars – later slides

allocate stack space
SP = SP – "space"
grows "down"



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

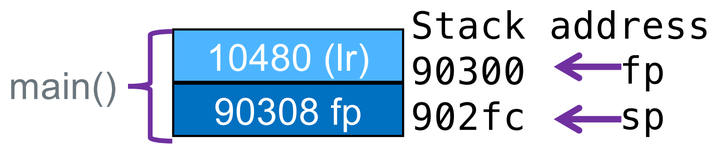
deallocate stack space
SP = SP + "space"
shrinks "up"



x

Using Minimal Stack Frames

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



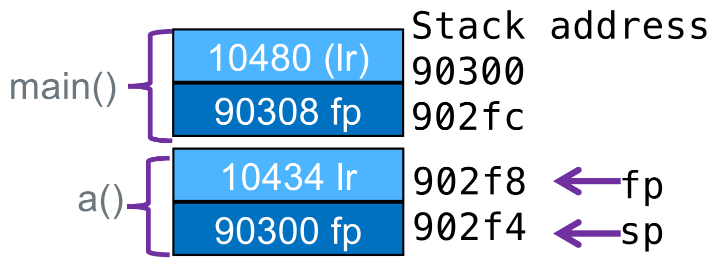
```
b:
    103f4:      push {fp, lr}
    103f8:      add fp, sp, 4
    103fc:      mov r0, 0
    10400:      sub sp, fp, 4
    10404:      pop {fp, lr}
    10408:      bx lr


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

x

Using Minimal Stack Frames

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

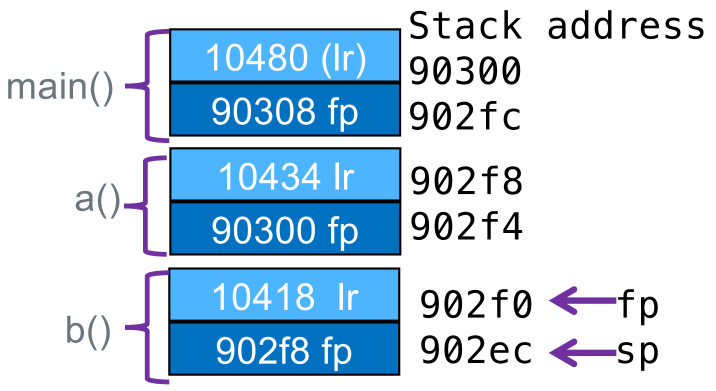


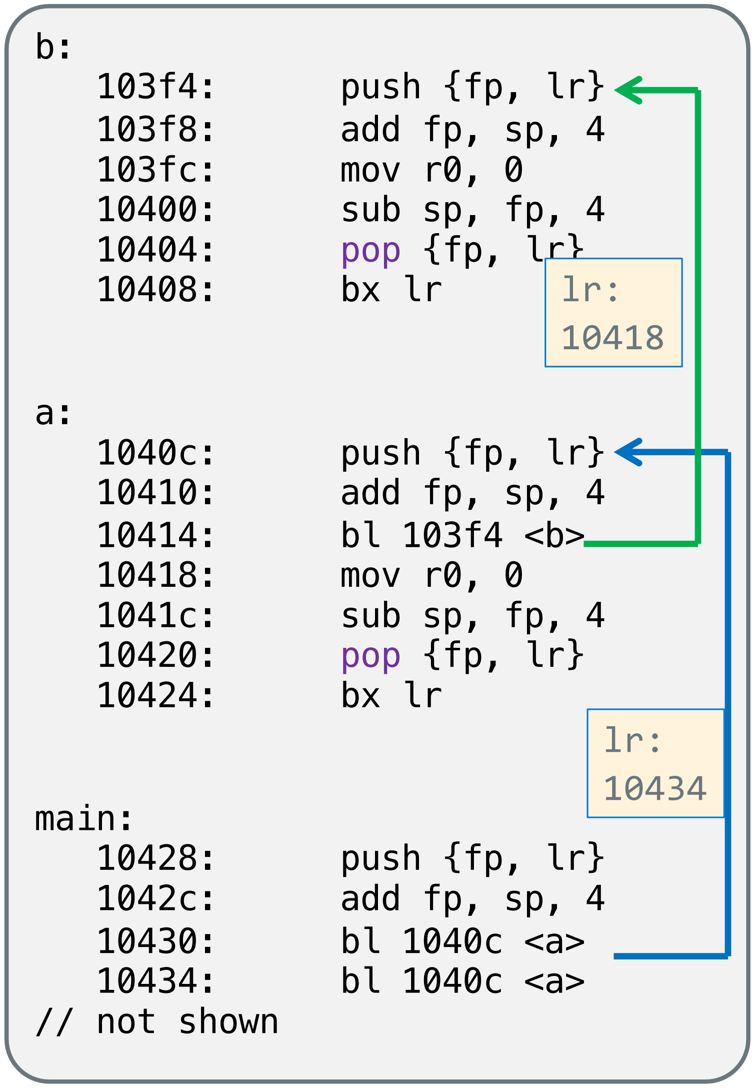
```
b:
    103f4:      push {fp, lr}
    103f8:      add fp, sp, 4
    103fc:      mov r0, 0
    10400:      sub sp, fp, 4
    10404:      pop {fp, lr}
    10408:      bx lr


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

x

Using Minimal Stack Frames

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



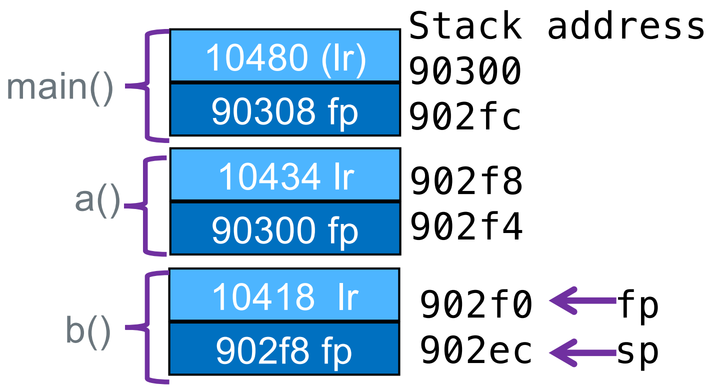


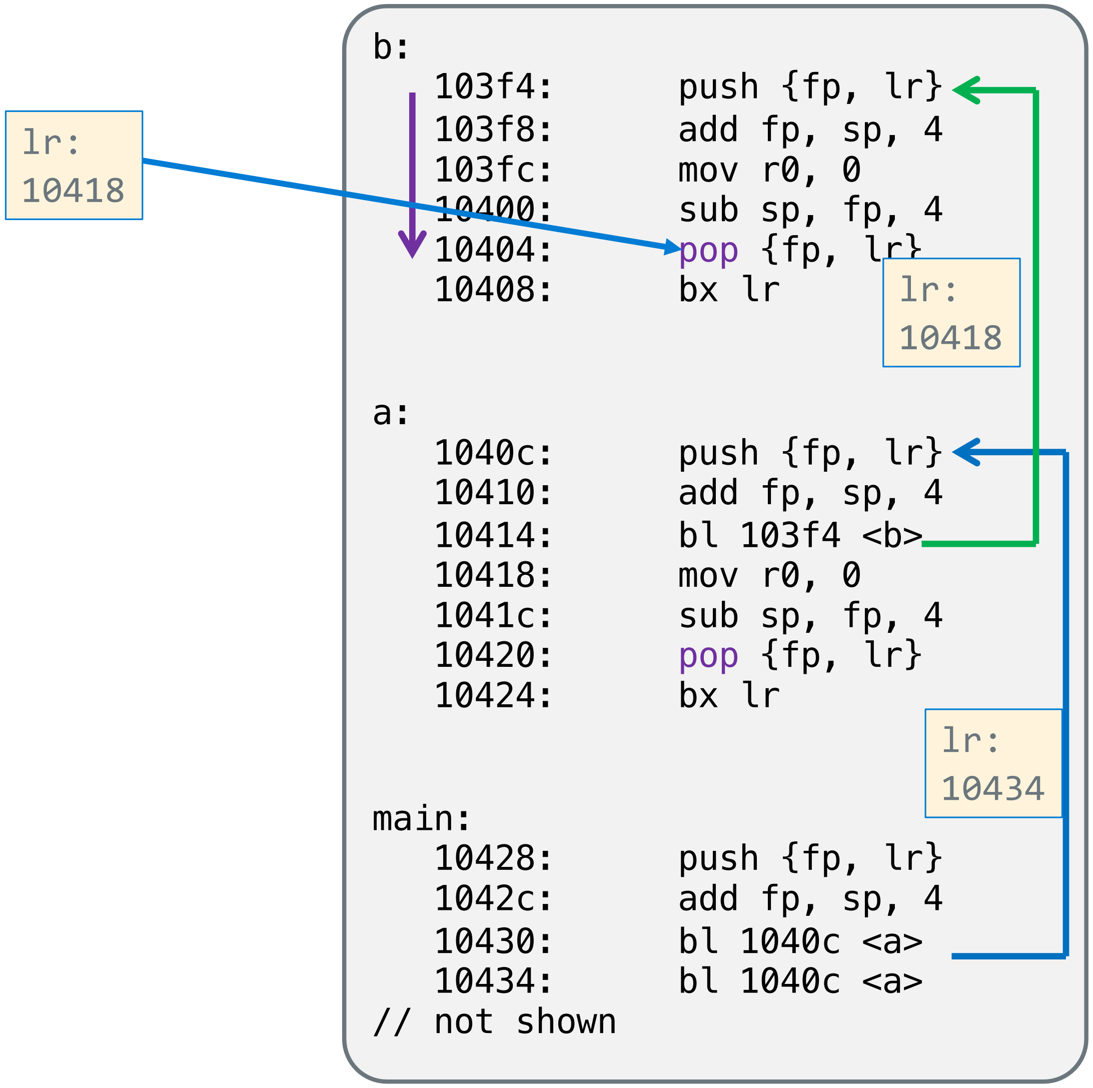
```
b:
    103f4:      push {fp, lr}
    103f8:      add fp, sp, 4
    103fc:      mov r0, 0
    10400:      sub sp, fp, 4
    10404:      pop {fp, lr}
    10408:      bx lr
                              lr:
                              10418

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
```

x

Using Minimal Stack Frames

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



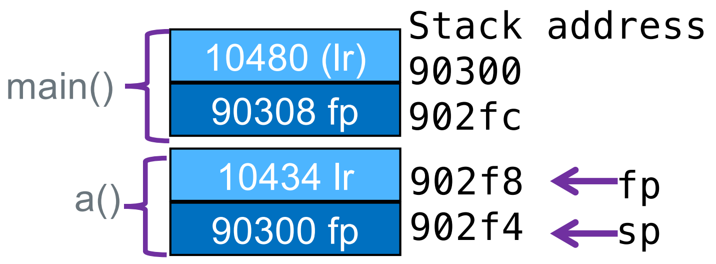


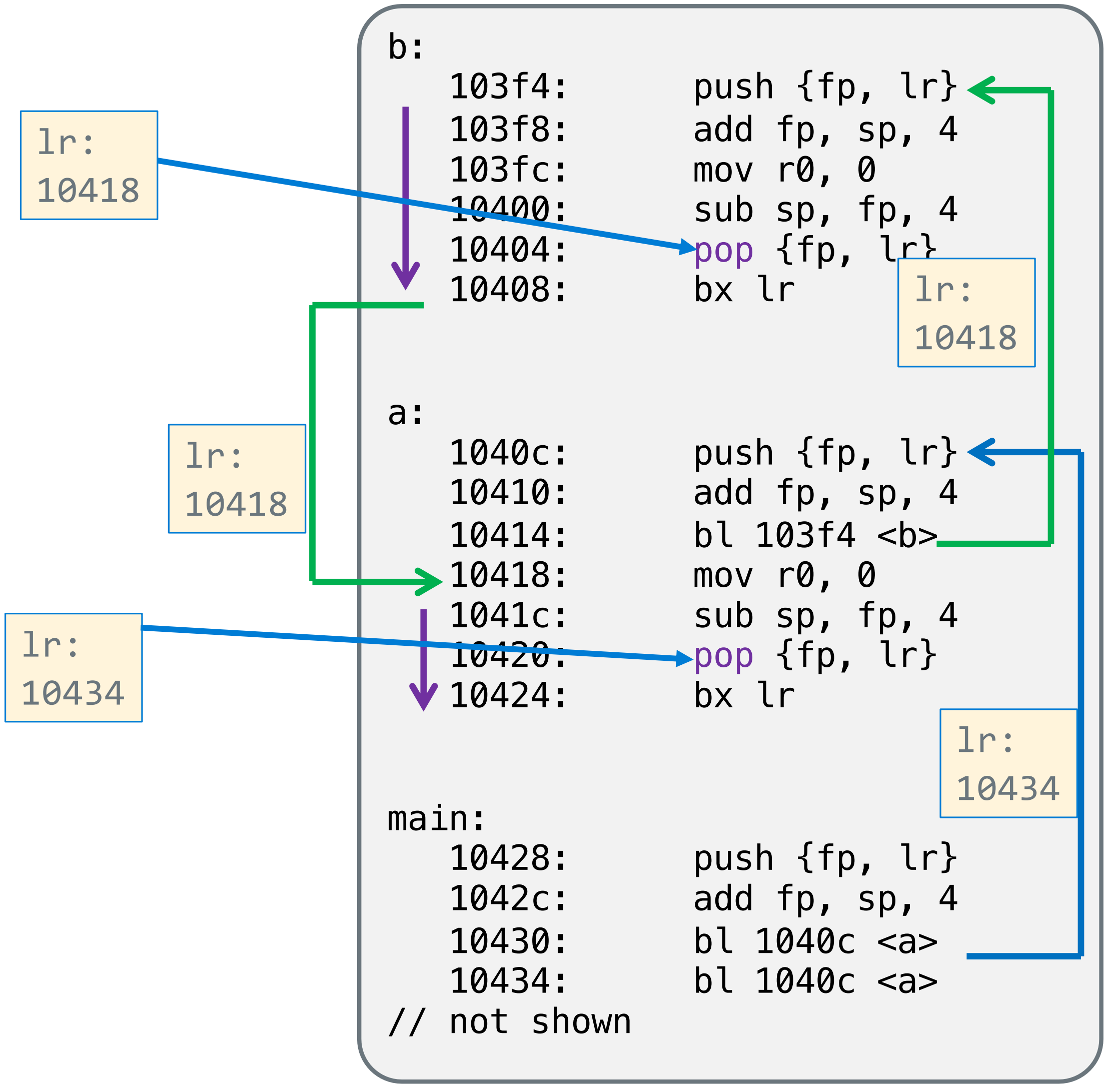
```
b:
   103f4:     push {fp, lr}
   103f8:     add fp, sp, 4
   103fc:     mov r0, 0
   10400:     sub sp, fp, 4
   10404:     pop {fp, lr}
   10408:     bx lr


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
```

x

Using Minimal Stack Frames

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

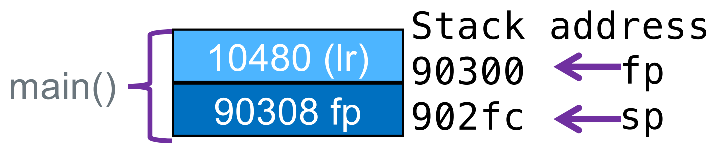



```
b:
    103f4:       push {fp, lr}
    103f8:       add fp, sp, 4
    103fc:       mov r0, 0
    10400:       sub sp, fp, 4
    10404:       pop {fp, lr}
    10408:       bx lr

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
```

x

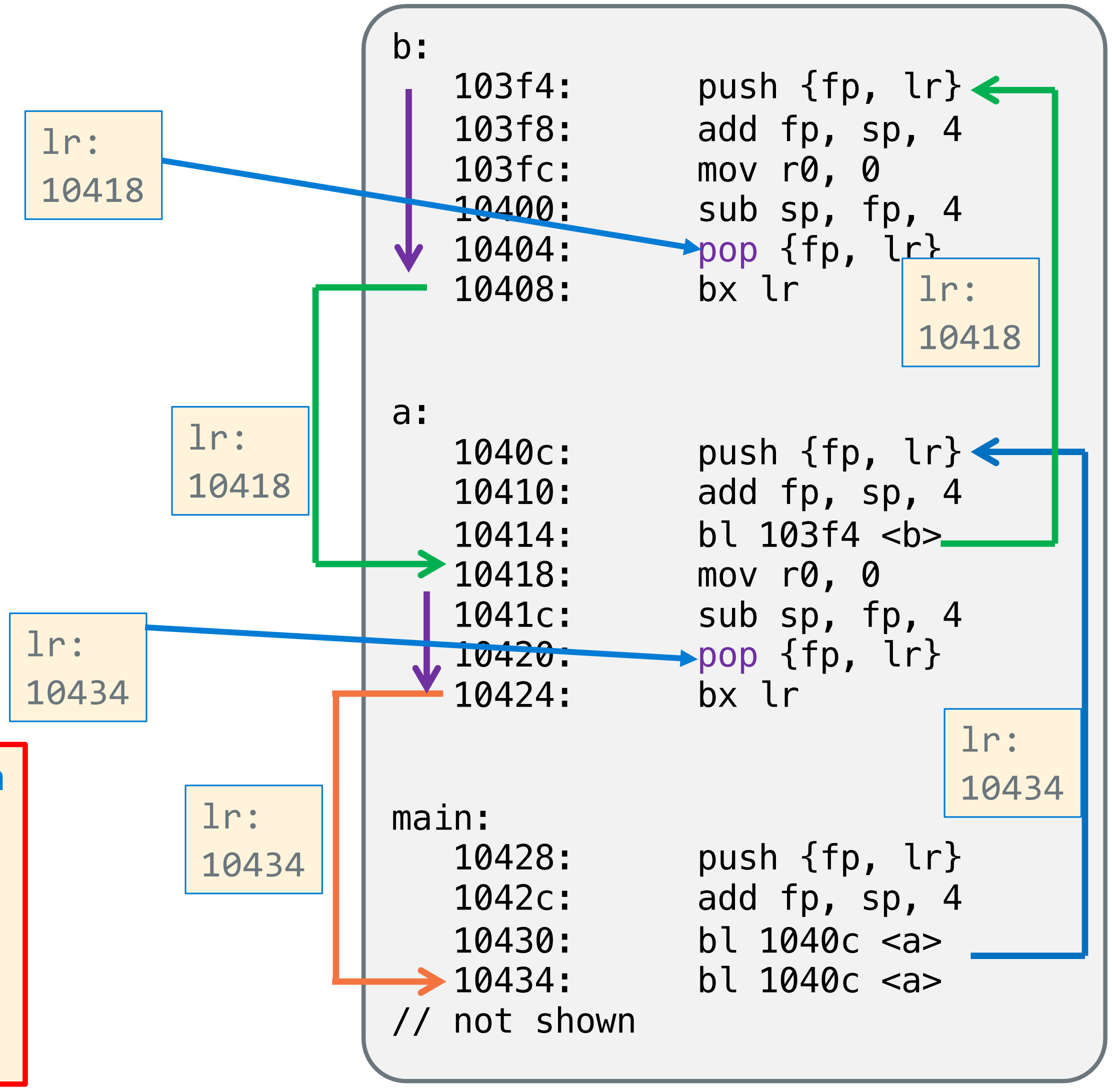
Using Minimal Stack Frames

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



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

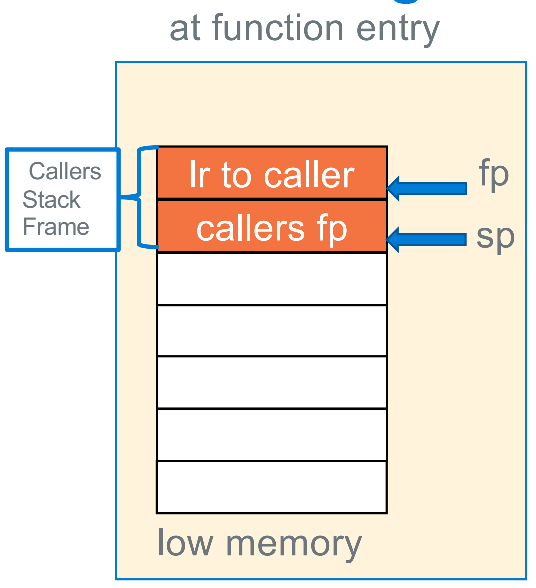
Result: NO infinite loop and we return to the correct instruction in the caller no matter how many functions we call.
Even recursion will work!

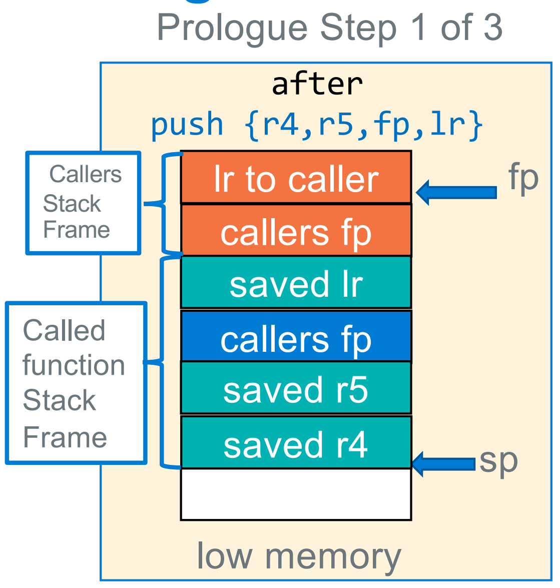


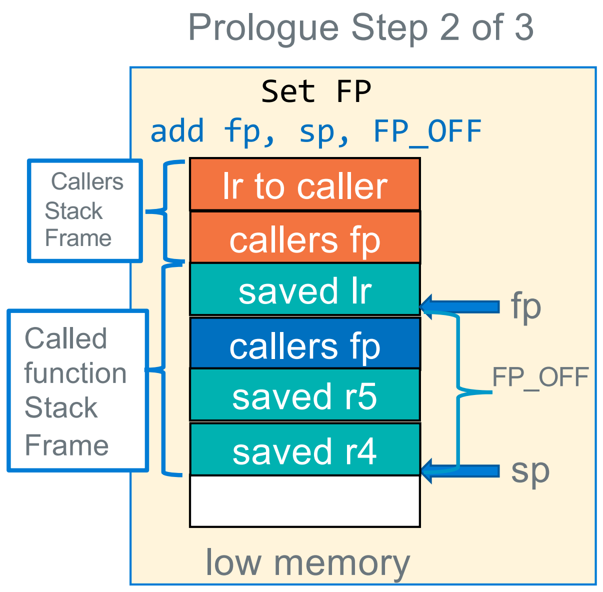
```
b:
    103f4:      push {fp, lr}
    103f8:      add fp, sp, 4
    103fc:      mov r0, 0
    10400:      sub sp, fp, 4
    10404:      pop {fp, lr}
    10408:      bx lr

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
```

x

Function Prologue: Allocating the Stack Frame -1







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

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

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

Function Prologue

```
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
```

x

Function Prologue: Allocating the Stack Frame - 2

Prologue Step 3 of 3

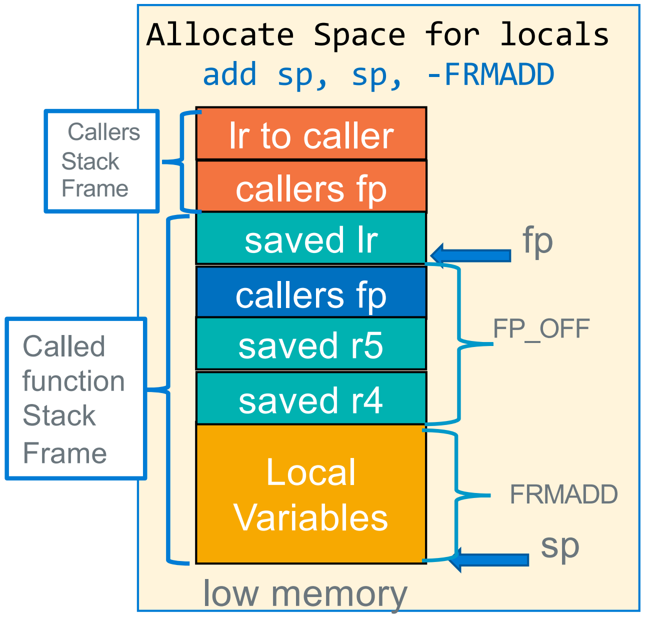
- Space for local variables is allocated on the stack right below the lowest pushed register
- **Add memory to the stack frame for local variables** by **moving** the sp towards low memory
- The amount moved is the total size of all local variables in bytes **plus** memory alignment **padding**

FRMADD = total local var space (bytes) + 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



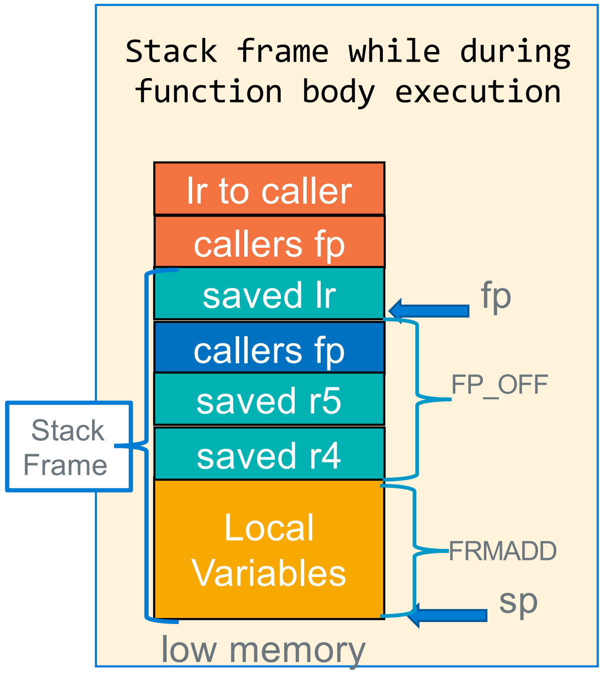
Allocate Space for Local Variables

Function Prologue

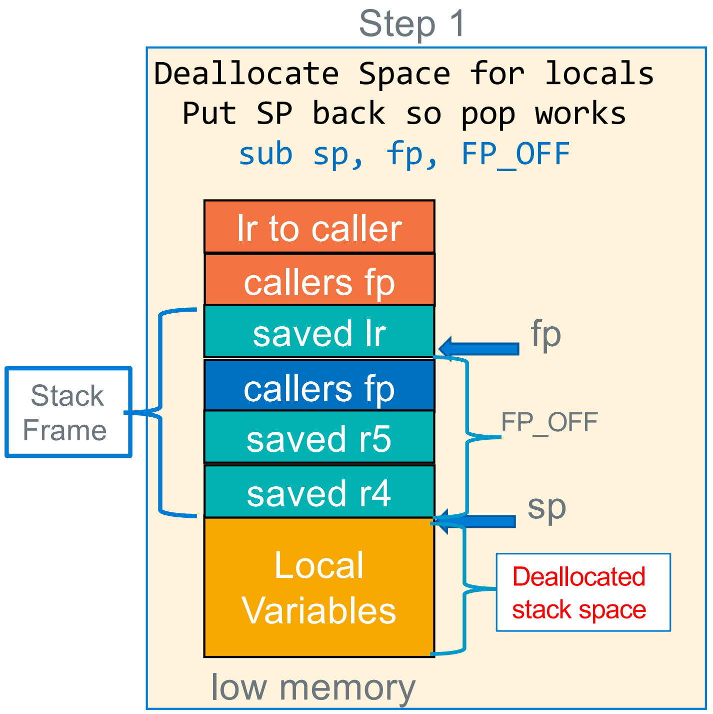
```
myfunc:
    push    {fp, lr}              // push (save) fp and lr on stack
    add     fp, sp, FP_OFF        // set fp for this function
    add     sp, sp, -FRMADD       // allocate FRMADD bytes for local vars
                                  // by moving sp
```

x

Function Epilogue: Deallocating the Stack Frame - 1



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



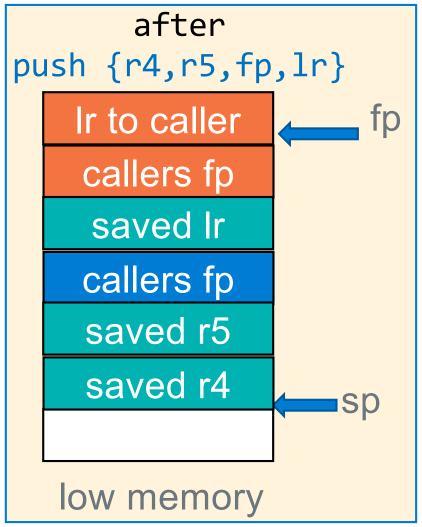
Move SP back to where it was after the push in the prologue.
So, pop works properly (this also deallocates the local variables)

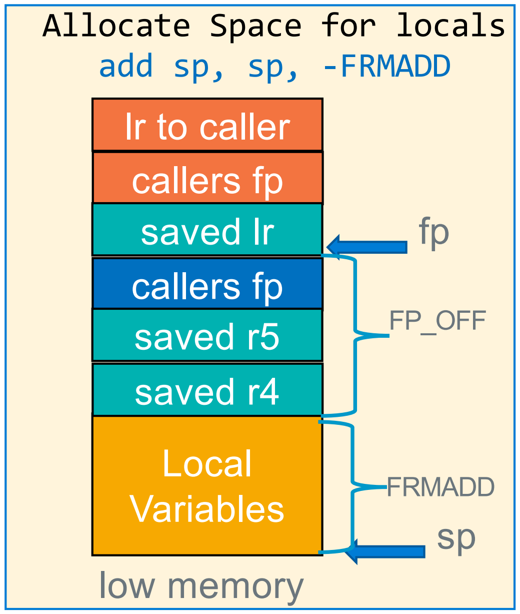
function Epilogue

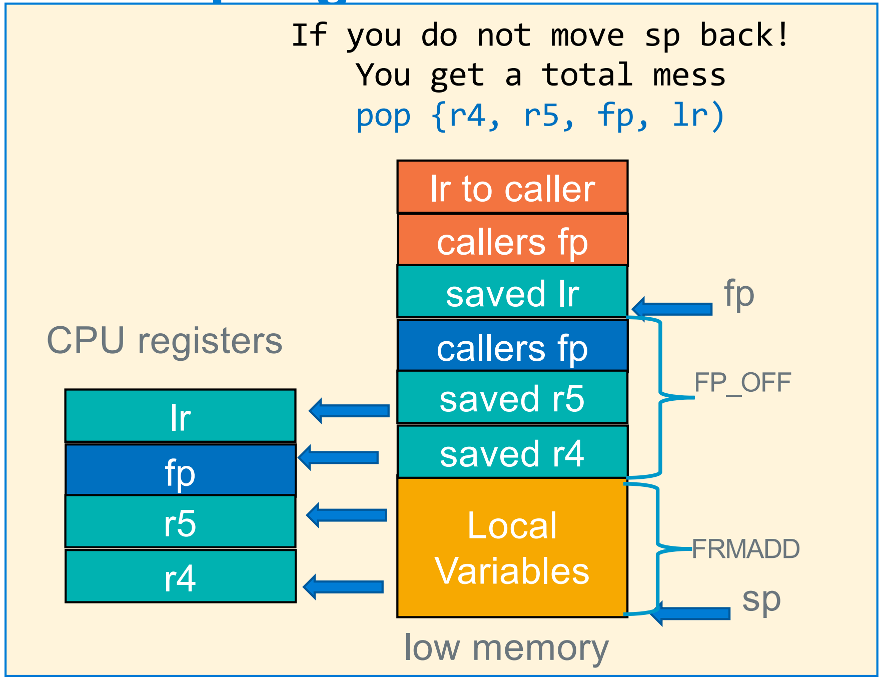
```
sub     sp, fp, FP_OFF          // deallocate local variables by moving sp
pop     {fp, lr}                // pop (restore) fp and lr from stack
bx      lr                      // return to caller
```

x

Why You must move SP before POP in the Epilogue







function Epilogue

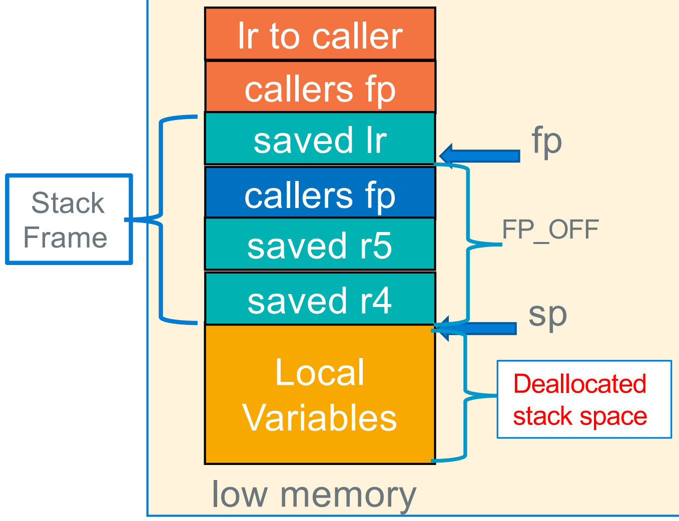
```
sub     sp, fp, FP_OFF          // deallocate local variables by moving sp
pop     {fp, lr}                // pop (restore) fp and lr from stack
bx      lr                      // return to caller
```

x

Function Epilogue: Deallocating the Stack Frame

Step 1

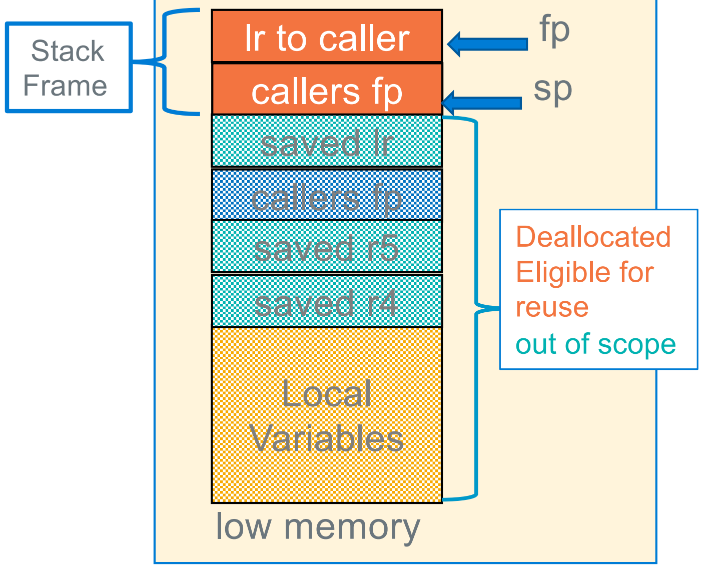
```
Deallocate Space for locals
  Put SP back so pop works
    sub sp, fp, FP_OFF
```



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

```
At function exit after
    pop {r4,r5,fp,lr}
```



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
```

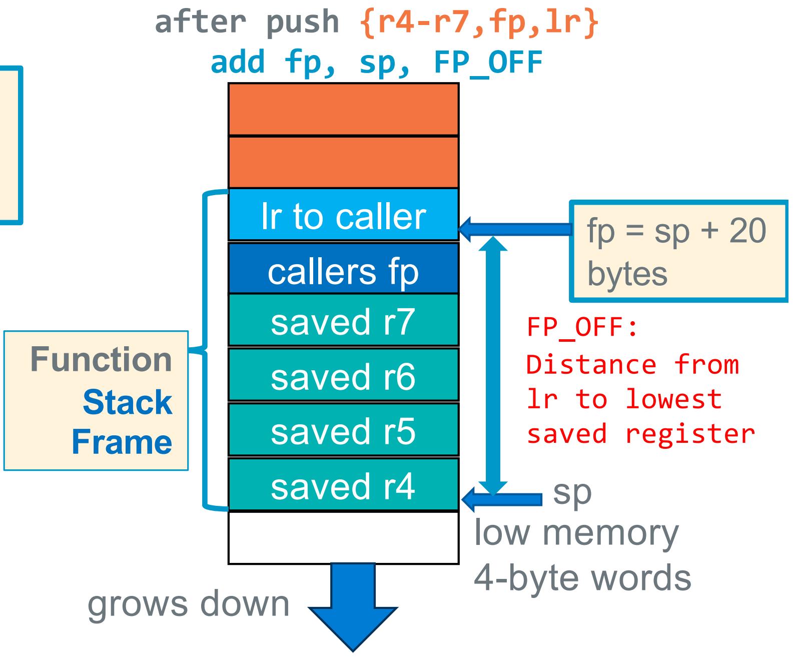
x

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



```
FP_OFF = (#regs saved - 1) * 4
```

# 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

▨ 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

x

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

x

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**

x

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;

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

            r0,    r1,    r2

    return EXIT_SUCCESS;
}
```

We are going to put these variables in temporary registers

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
```

x

Example: using preserved registers for local variables

```c
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c; // use r0
    int count = 0;  // use r4

             r0
    while ((c = getchar()) != EOF) {

        putchar(c);
        count++;
    }        r0        r0   r1

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

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

Push two registers to keep stack 8-byte aligned (sp % 8 == 0)

```
         .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)
```

x

Putchar/getchar: The while loop

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

address of string literal variable

```
        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
```

```
#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;
}
```

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

File header and footers are not shown

x

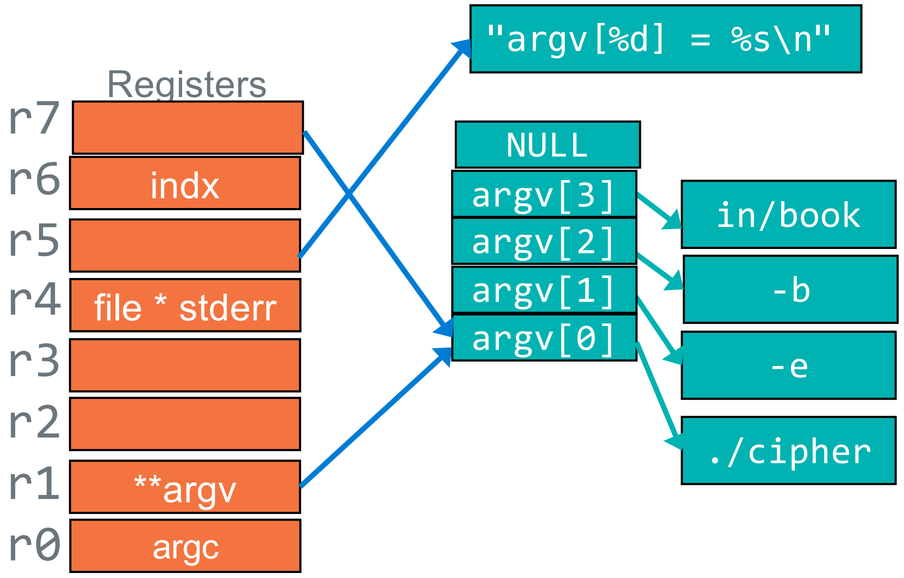
Accessing Pointers (argv) in ARM assembly

```
    .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;

// see next slide

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

r0-r3 lost due to fprintf call



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

x

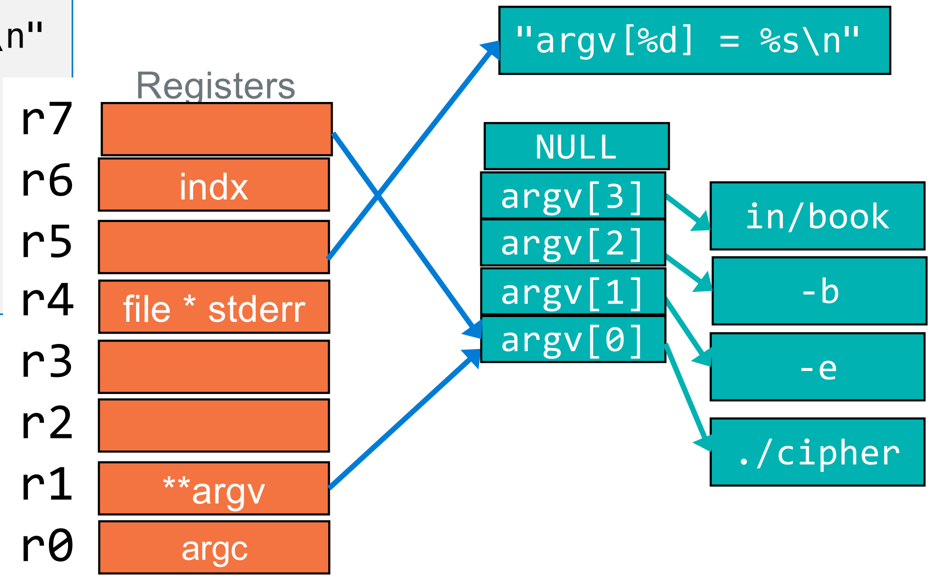
Accessing Pointers (argv) in ARM assembly

```
.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:
```

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

r0-r3 lost due to fprintf call

observe the different increment sizes



x

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 = total local var space (bytes) + padding

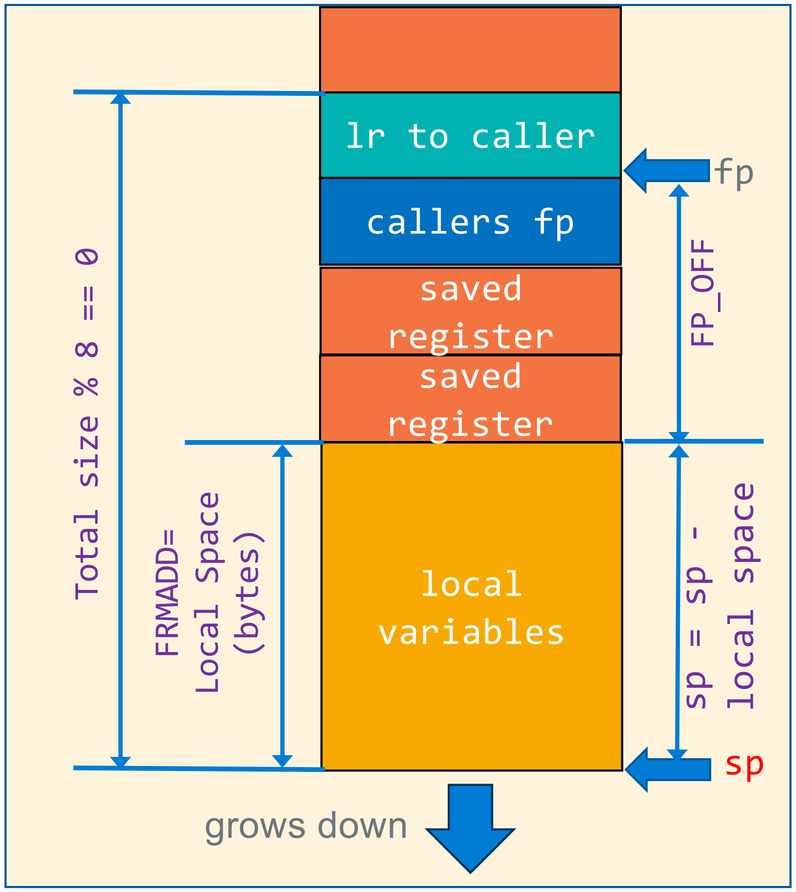
- Allocate the space after the register push by

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

- **Requirement:** on function entry, sp is always 8-byte aligned

  sp % 8 == 0

- **Padding (as required):**
  1. Additional space between variables on the stack to meet memory alignment requirements
  2. 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



x

Review Variables: Size

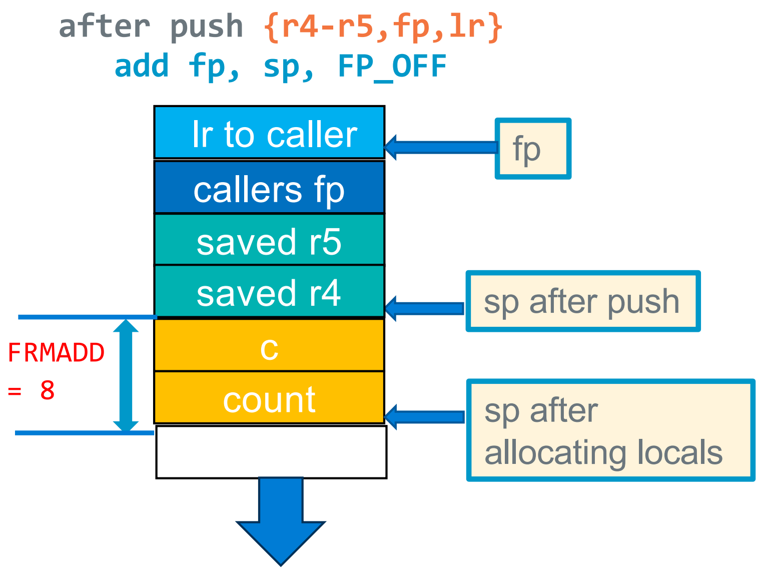
- **Integer types**
  - `char (unspecified default)`
  - `int (signed default)`
- **Floating Point**
  - `float`, `double`
- Optional Modifiers for each base type
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]
  - `const`: variable read only
- **char type**
  - One byte in a byte addressable memory
  - **Be careful** char is unsigned on arm and signed on other HW like intel

C Data Type	AArch-32 contiguous Bytes	printf specification
unsigned char	1	%c
signed char	1	%c
short int	2	%hd
unsigned short int	2	%hu
int	4	%d / %i
unsigned int	4	%u
long int	4	%ld
long long int	8	%lld
float	4	%f
double	8	%lf
long double	8	%Lf
pointer *	4	%p

x

Local Variables on the stack

```
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
```

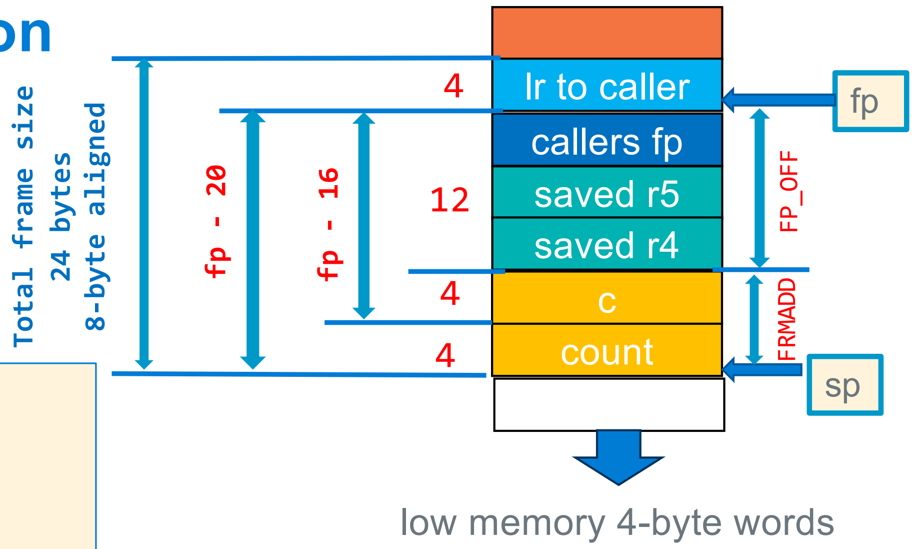
- In this example we are allocating two variables on the stack
- When writing assembly functions, in many situations you may choose allocate these to registers instead

- 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

x

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**

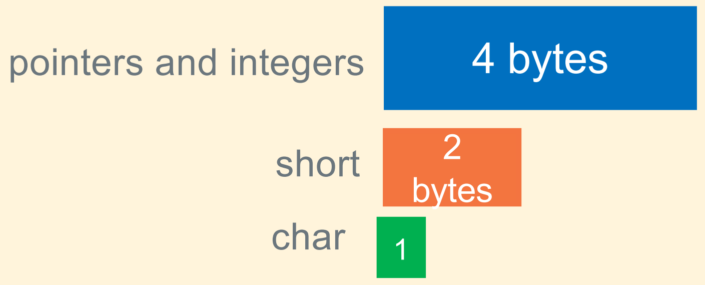
Variable	distance from fp	Read variable	Write Variable
`int c`	-16	`ldr r0, [fp, -16]`	`str r0, [fp, -16]`
`int count`	-20	`ldr r0, [fp, -20]`	`str r0, [fp, -20]`

```
 .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!
```

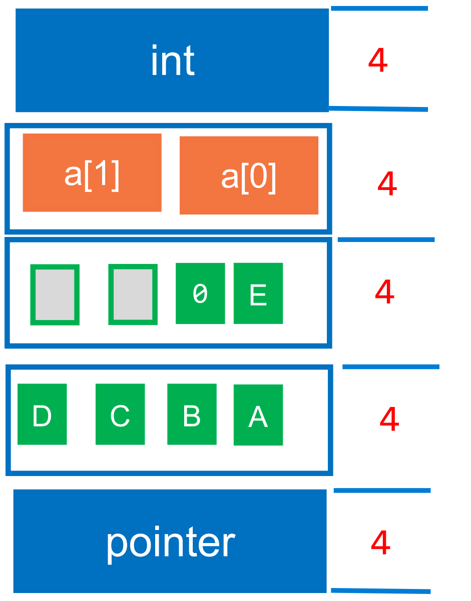
x

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)



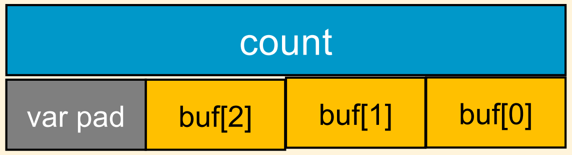
Rule: When the function is entered the stack is already 8-byte aligned



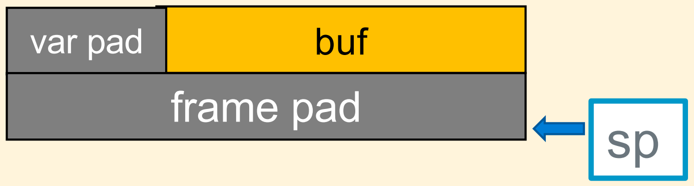
x

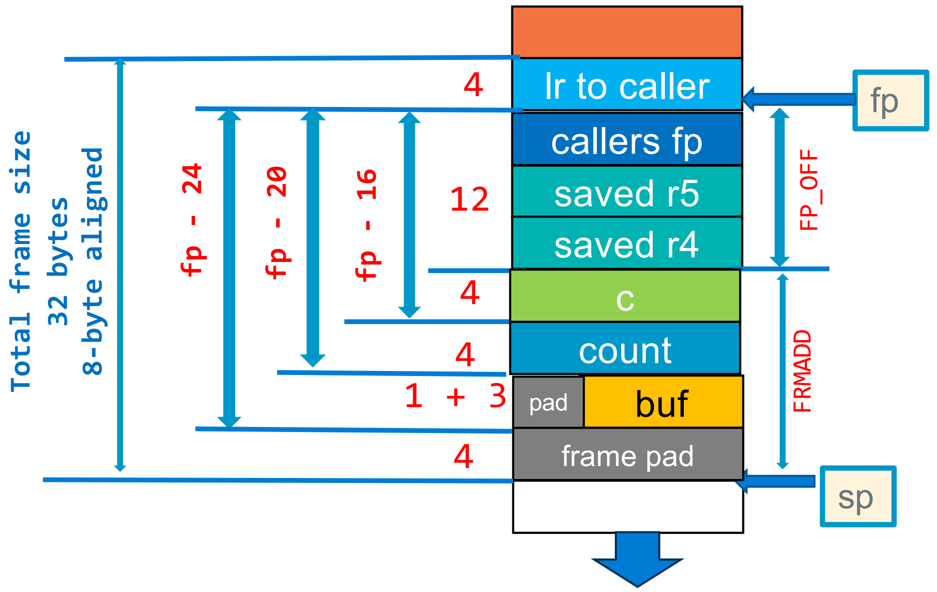
Stack Variables: Padding

- **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
}
```

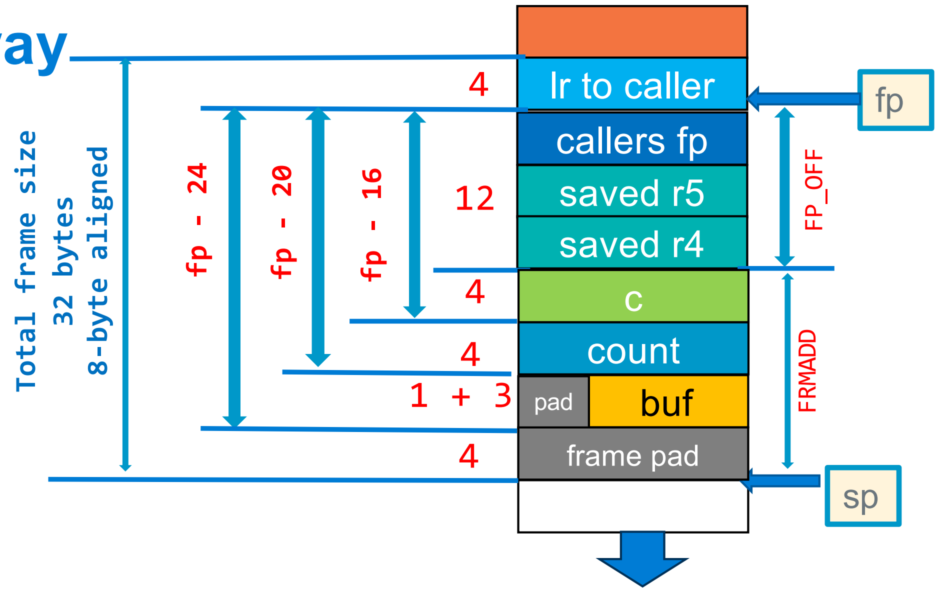
```
 .text
    .type   main, %function
    .global main
   .equ    FP_OFF,    12
   .equ    FRMADD,    16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
// but we are not done yet!
```

x

Accessing Stack Variables, the hard way

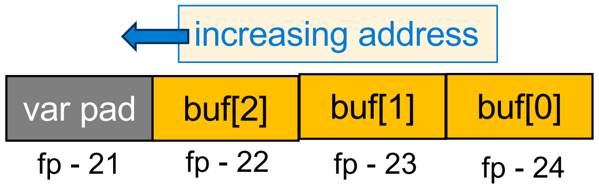
```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
 .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)

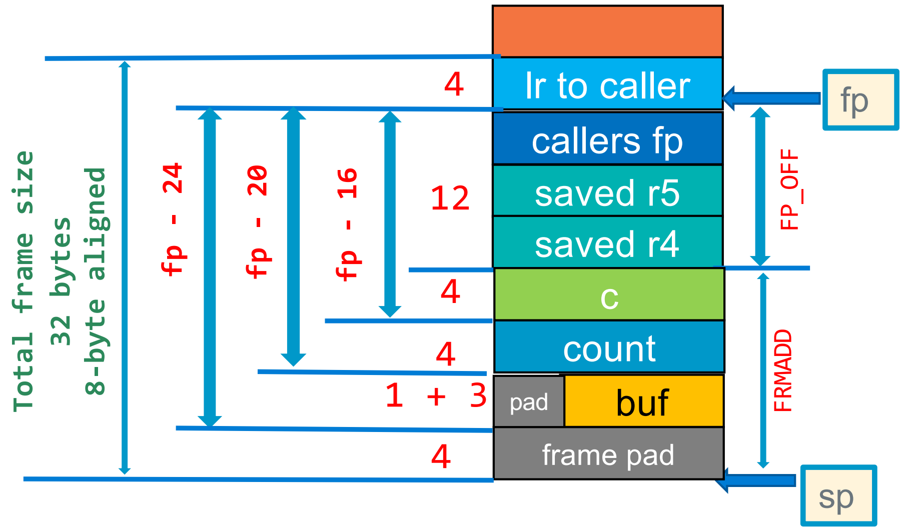
Variable	distance from fp	Read variable	Write Variable
`int c`	16	`ldr r0, [fp, -16]`	`str r0, [fp, -16]`
`int count`	20	`ldr r0, [fp, -20]`	`str r0, [fp, -20]`
`char buf[0]`	24	`ldrb r0, [fp, -24]`	`strb r0, [fp, -24]`
`char buf[1]`	23	`ldrb r0, [fp, -23]`	`strb r0, [fp, -23]`
`char buf[2]`	22	`ldrb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

x

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
```

pushed reg fp distance

variable size in bytes

Prior allocation distance

1. For each stack variable create a .equ symbol whose value is the distance in bytes from the FP after the prologue
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

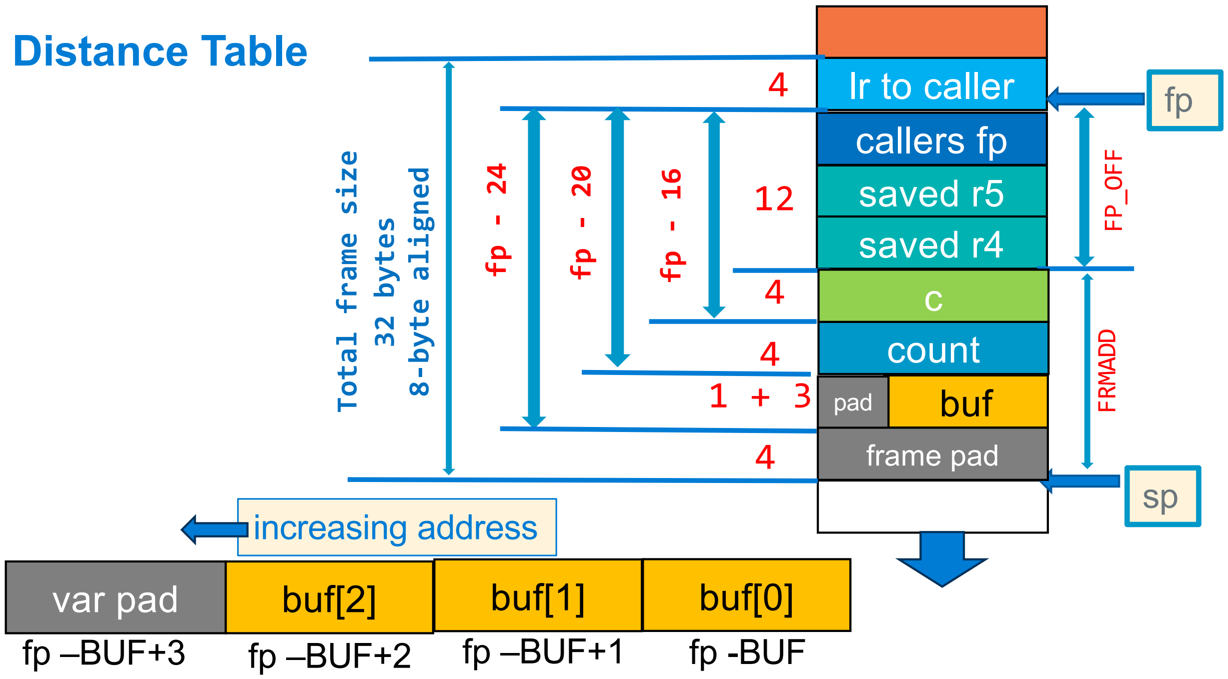
   FRMADD = distance PAD minus distance of the SP to the FP (FP_OFF) after the prologue push

x

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]`

x

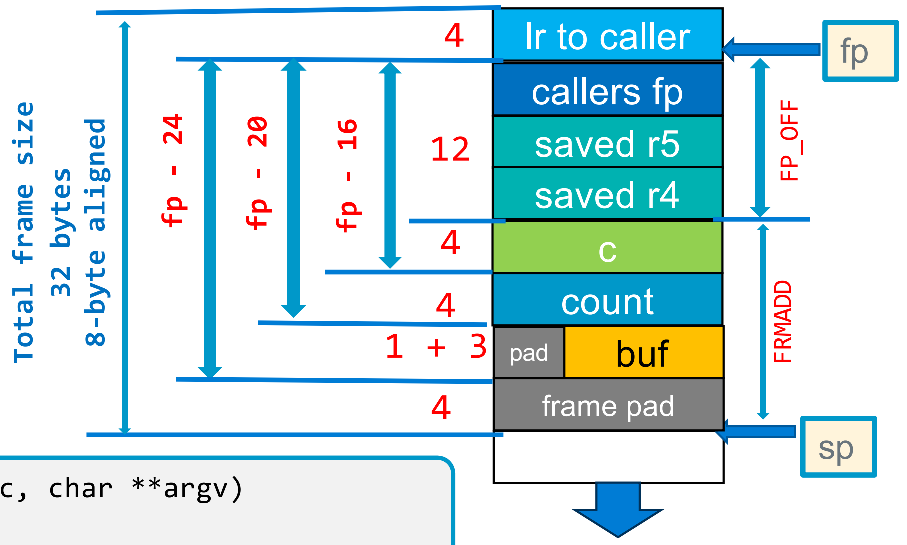
Initializing and Accessing Stack variables

```
    .section .rodata
.Lmess: .string "%d %d %s\n"
    .extern printf
```

```
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 and COUNT

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
```

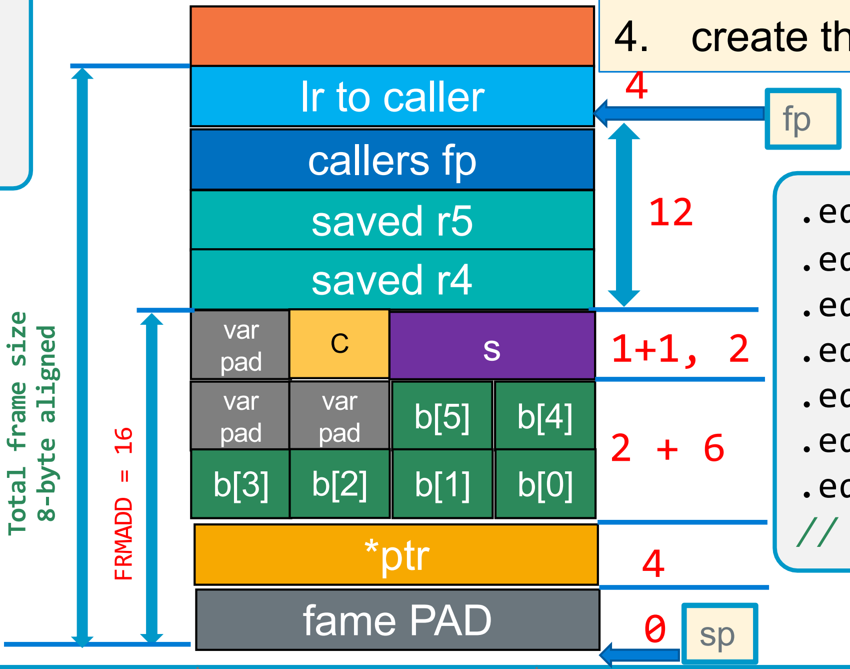
Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add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

x

Stack Frame Design Practice

```
void func(void)
{
  signed char c;
  signed short s;
  unsigned char b[] = "Stack";
  unsigned char *ptr = &b;
    // 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 distance table to the variables



```
.equ    FP_OFF,     12
.equ    C,          2 + FP_OFF
.equ    S,          2 + C
.equ    B,          8 + S
.equ    PTR,        4 + B
.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]	B	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]

x

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;                      // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum;  // pf could be in a register

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

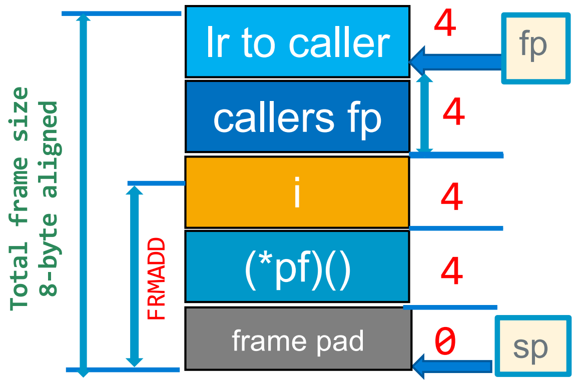
Output Parameters (like i) you pass a pointer to them, **must be on the stack!**

x

Working with Pointers on the stack

```
int main()
{
    int i; // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum;  // pf could be in a register

    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]

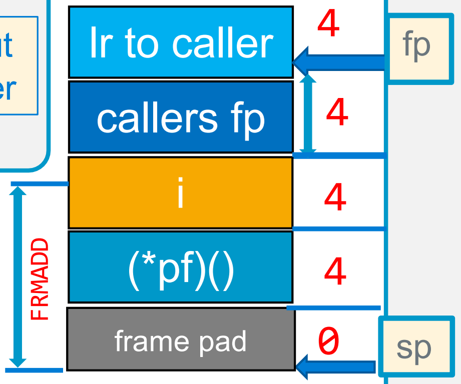
x

Working with Pointers on the stack

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

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

I is Output Parameter

```
    .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
```



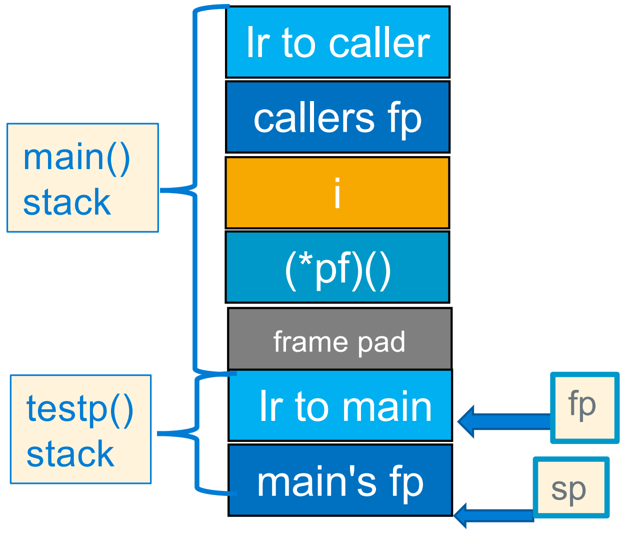
```
main:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp,-FRMADD

    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
```

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]

x

Working with Pointers on the stack

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



r0,r1,r2 already set

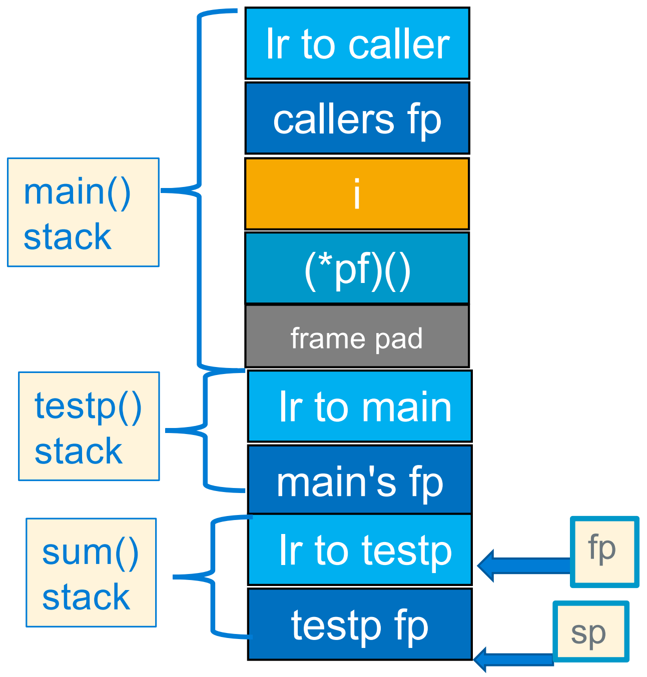
```
    .global testp
    .type   testp, %function
    .equ    FP_OFF, 12
testp:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF

    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)
```

x

Working with Pointers on the stack

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

```
    .global sum
    .type   sum, %function
    .equ    FP_OFF, 4
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)
```

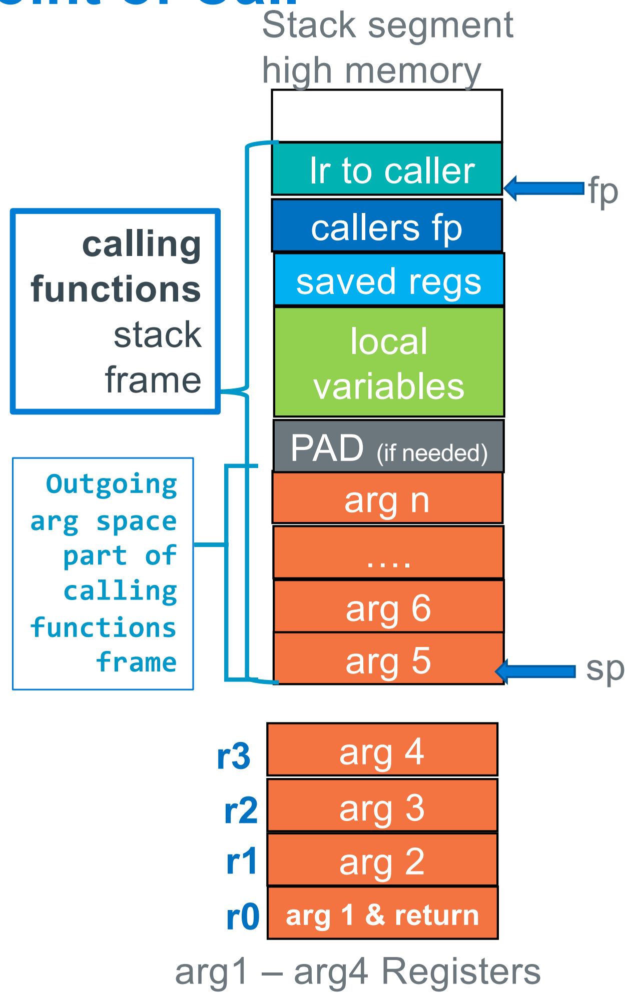


x

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
- 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**,
  3. **Add padding** to force arg5 alignment if needed is **placed above** the last **argument the called function is expecting**

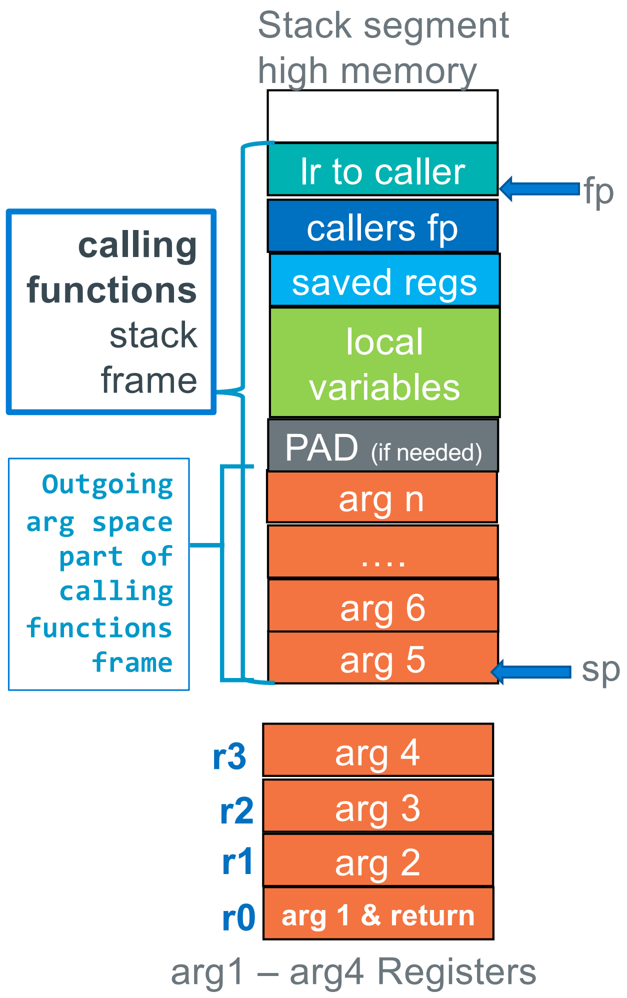


x

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**
- Calling function prior to making the call you must
  1. Evaluate first four args: place the resulting values in r0-r3
  2. Evaluate Arg 5 and greater and place the resulting values on the stack

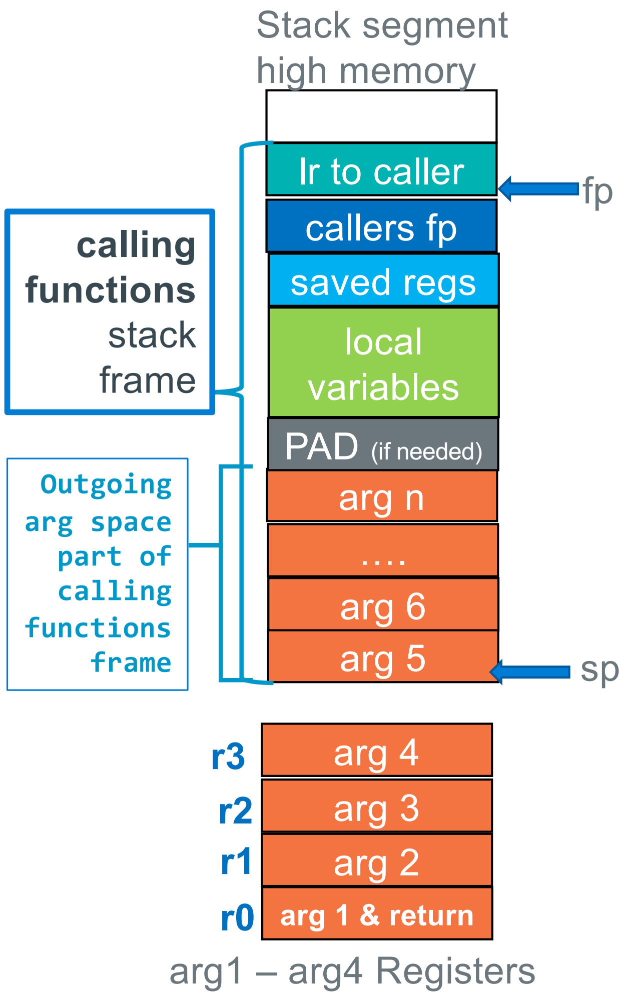


x

Passing More Than Four Arguments – At the point of Call

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

- **Approach**: Extend the stack frame to include enough space for stack arguments for the called function that has the greatest number of args
  1. Examine every function call in the body of a function
  2. Find the function call with greatest arg count, this determines space needed for outgoing args
  3. Add the greatest arg count space as needed to the frame layout
  4. Adjust PAD as required to keep the sp 8-byte aligned



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

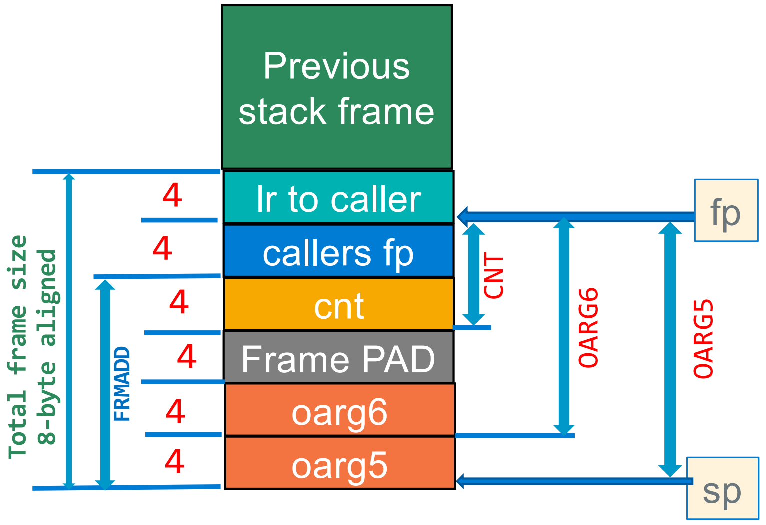
x

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 at function call**

```
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       // 4 bytes
.equ   OARG5,        4 + OARG6     // 4 bytes
.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]`

x

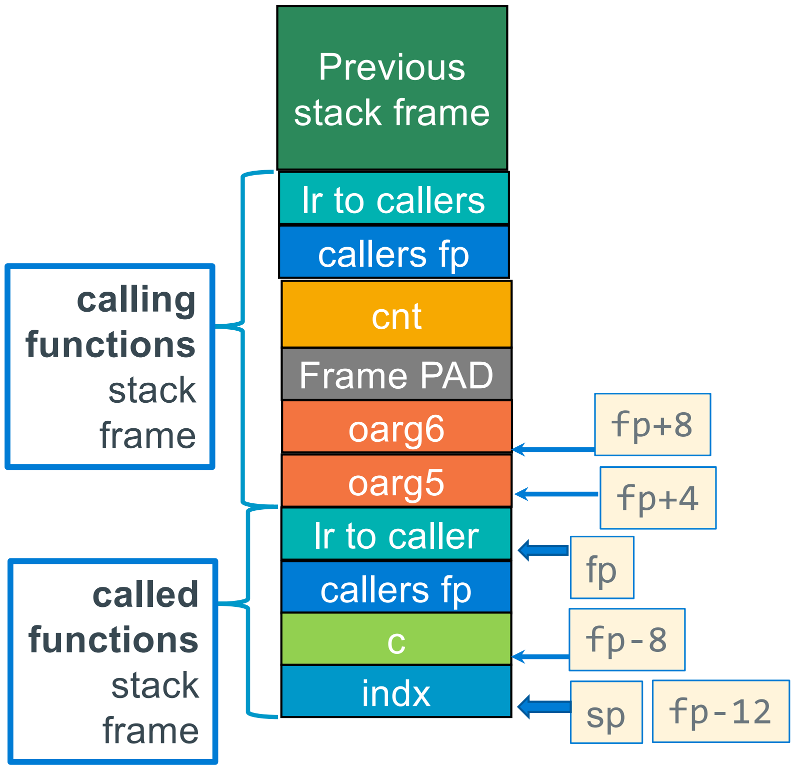
Called Function: Retrieving Args From the Stack

```
r0 = func(r0, r1, r2, r3, 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**

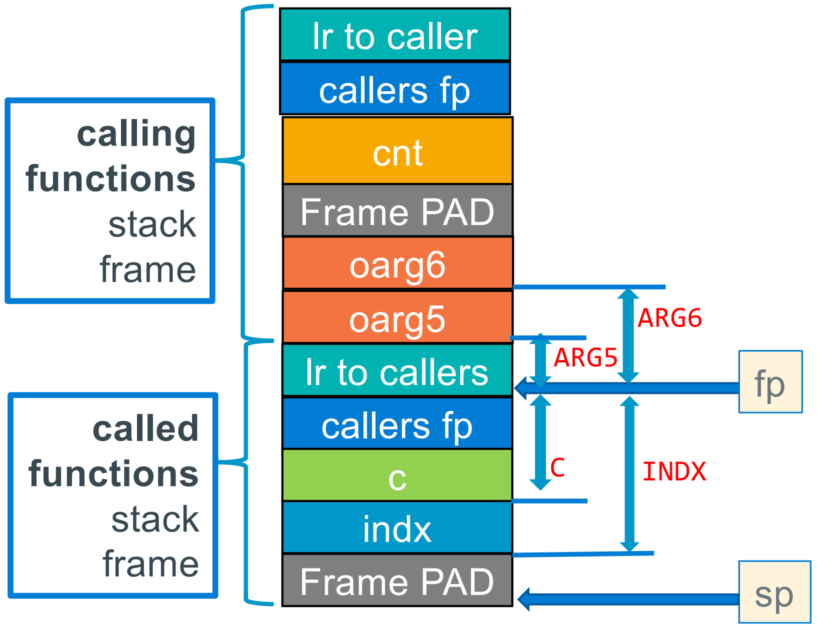
x

Called Function: Retrieving Args From the Stack

```
.equ    FP_OFF,     4
.equ    C,          4 + FP_OFF
.equ    INDX,       4 + C
.equ    PAD,        0 + INDX
.equ    FRMADD,     PAD - FP_OFF
// below are distances into the 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

x

Example: Passing Stack Args, Calling Function

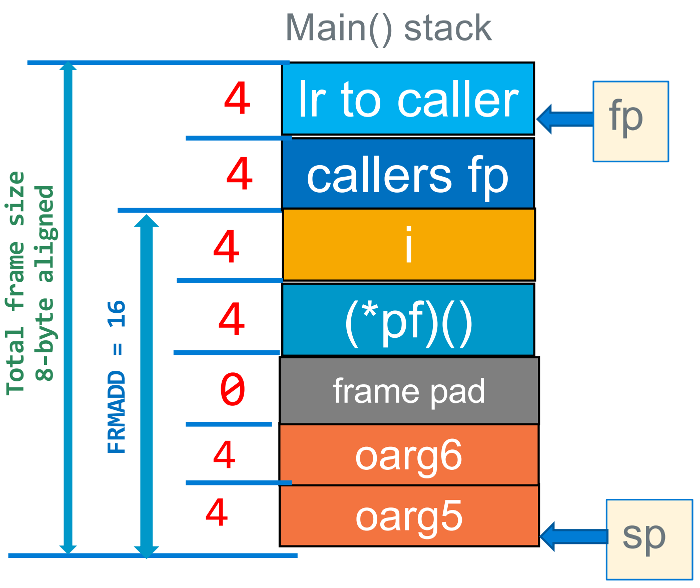
```
int sum(int j, int k)
{
    return j + k;
}

        arg1      arg2      arg3      arg4           arg5                  arg6
void
testp(int j, int k, int l, int m, int (*func)(int, int), int *i)
{
    *i = func(j,k) + func(l, m);  // notice two func() calls

    return;
}

int main()
{
    int i;  // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

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

    return EXIT_SUCCESS;
}
```

x

Example: Passing Stack Args, Calling Function

```
int main()
{
    int i;  // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

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

```
     .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
```

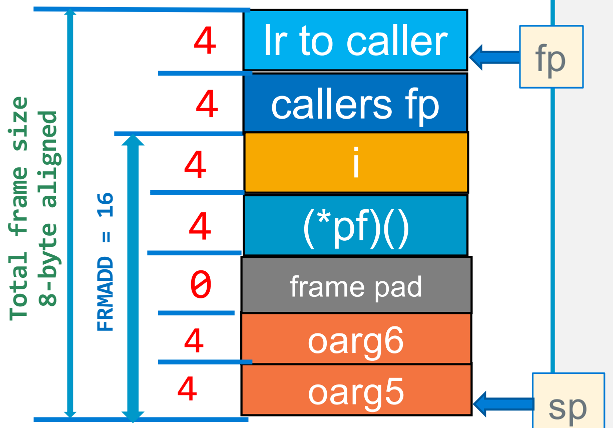


Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
`int i`	I	`add r0, fp, -I`	`ldr r0, [fp, -I]`	`str r0, [fp, -I]`
`int (*pf)()`	PF	`add r0, fp, -PF`	`ldr r0, [fp, -PF]`	`str r0, [fp, -PF]`
`int oarg6`	OARG6	`add r0, fp, -OARG6`	`ldr r0, [fp, -OARG6]`	`str r0, [fp, -OARG6]`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

x

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;
}
```



```
    .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
```

```
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]         // arg 6: store address of I

    ldr     r0, [fp, –PF]    // get PF from stack
    add     r1, fp, –OARG5   // address of OARG5
    str     r0, [r1]         // arg 5: store sum() address

    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
```

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]`

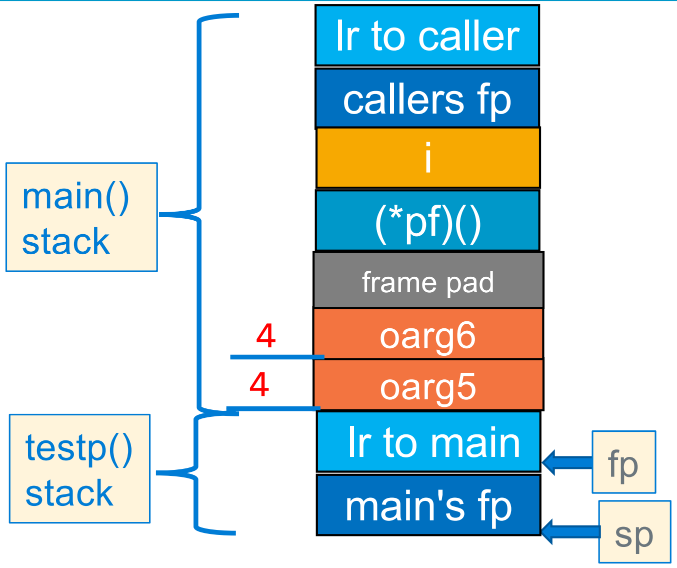
X

Example: Passing Stack Args, Called Function

arg1 arg2 arg3 arg4 arg5 arg6

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

    return;
}
```

short circuit: make this call first



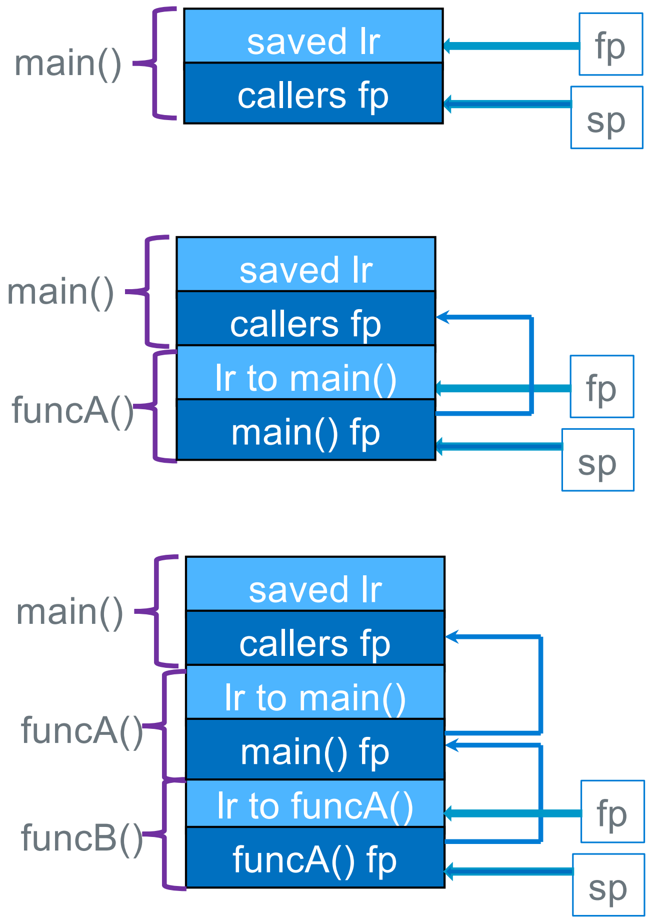
```
        .equ    FP_OFF, 20
        .equ    ARG6,   8
        .equ    ARG5,   4
testp:
        push    {r4-r7, fp, lr}
        add     fp, sp, FP_OFF

        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
```

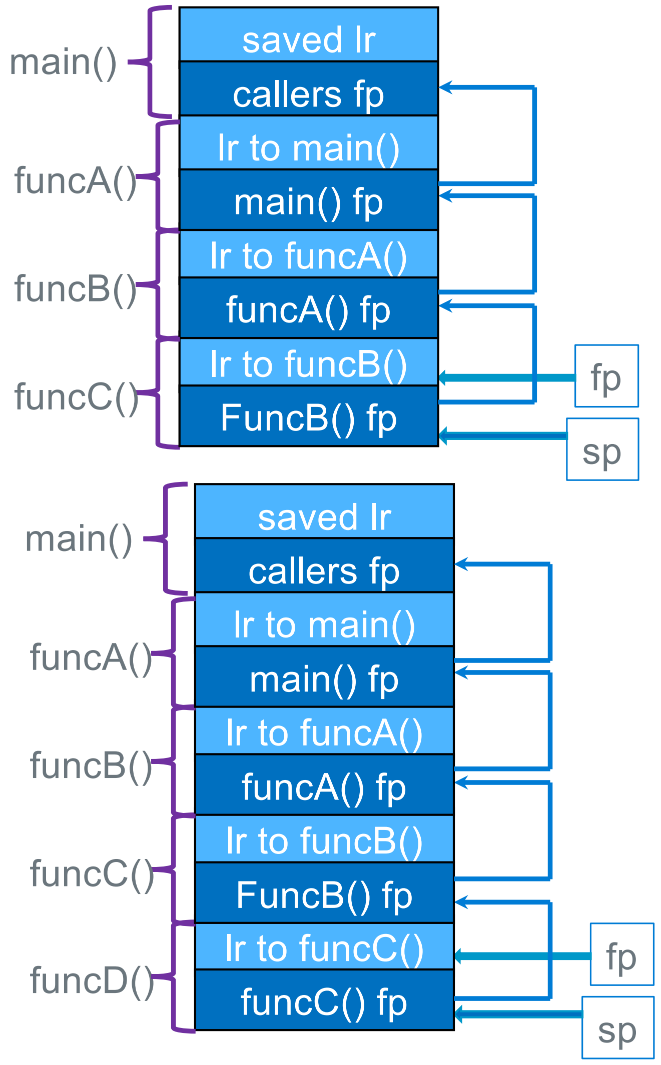
Argument	distance	Address on Stack	Read variable	Write Variable
`int *i`	ARG6	`add r0, fp, ARG6`	`ldr r0, [fp, ARG6]`	`str r0, [fp, ARG6]`
`int (*fp)()`	ARG5	`add r0, fp, ARG5`	`ldr r0, [fp, ARG5]`	`str r0, [fp, ARG5]`

x

Extra Slides

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





How gdb finds stack frames

x