Version 2.04

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 17

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

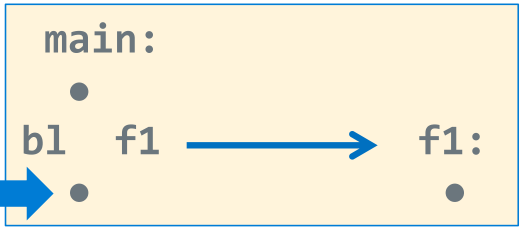
Branch with Link Indirect (function call) instruction

```
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



```
main:
 •
bl  f1        f1:
 •             •
```

x

Function Call Return

Branch & exchange (function return) instruction

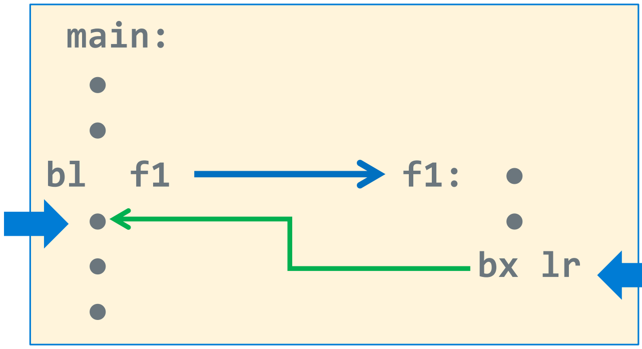
`bx lr`

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



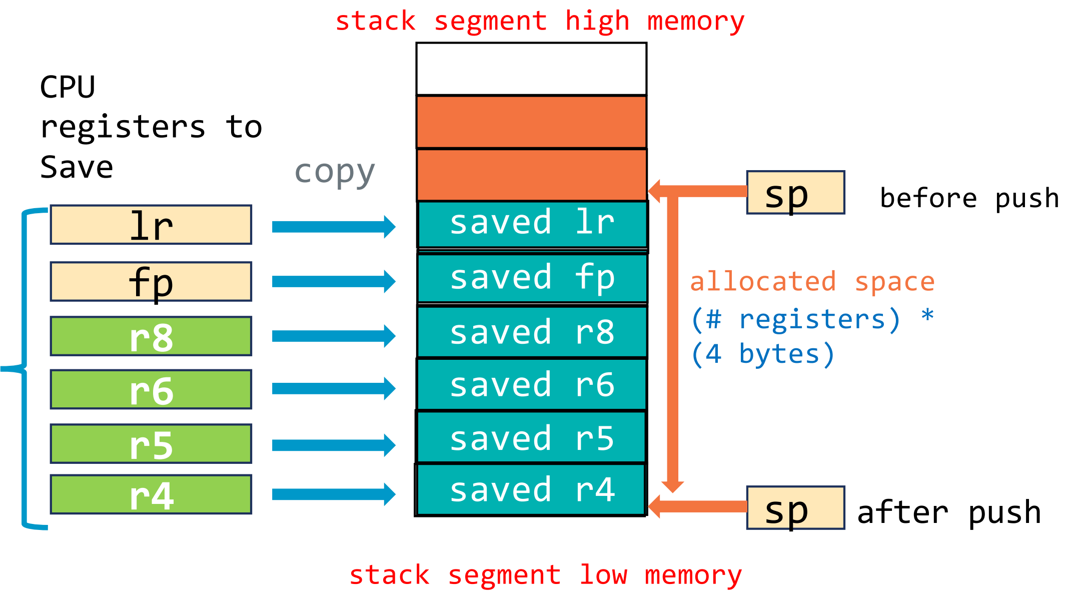
Branch to the instruction whose address is stored in lr

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!



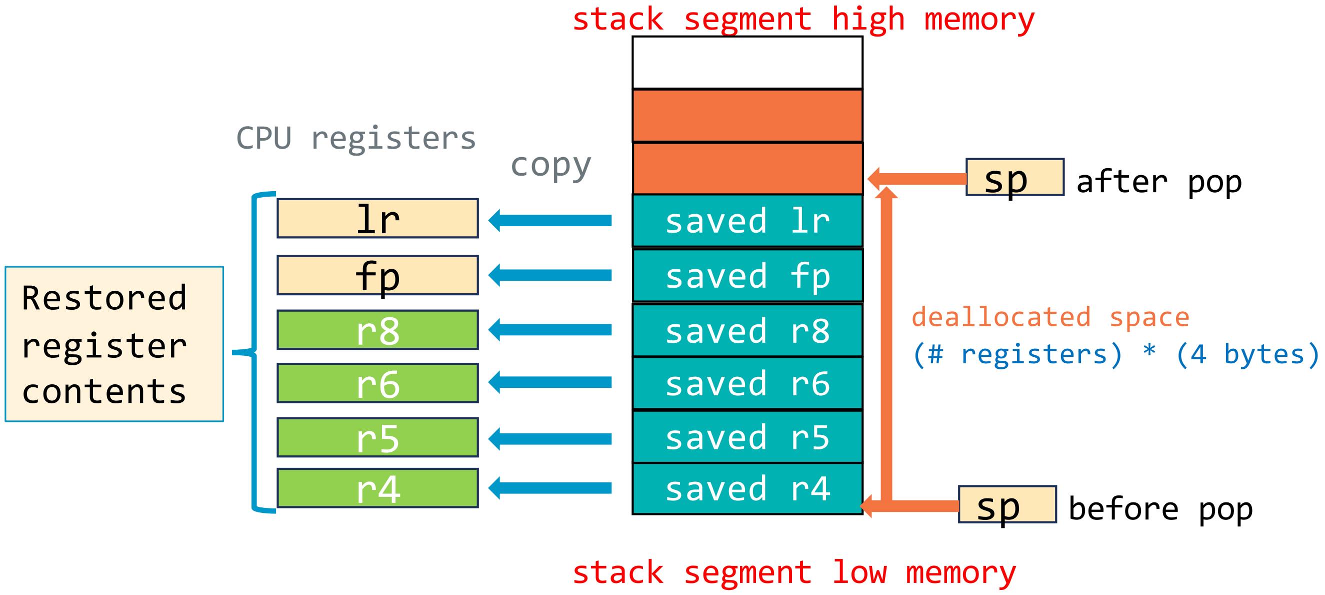
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** subtracts (# of registers saved) * (4 bytes) from the **sp** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

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

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**
- 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)

x

What it means to be a Temporary/argument register

```c
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

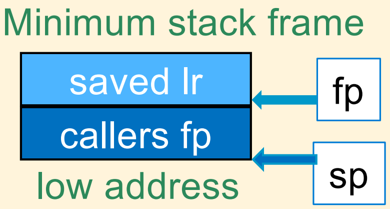
Register	*Function Call Use*	*Function Body Use*	*Save before use* *Restore before return*
r4-r10	preserved registers	contents preserved across function calls	Yes
r11/fp	stack frame pointer	Use to locate variables on the stack	Yes
r13/sp	stack pointer	stack space allocation	Yes
r14/lr	link register	contains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

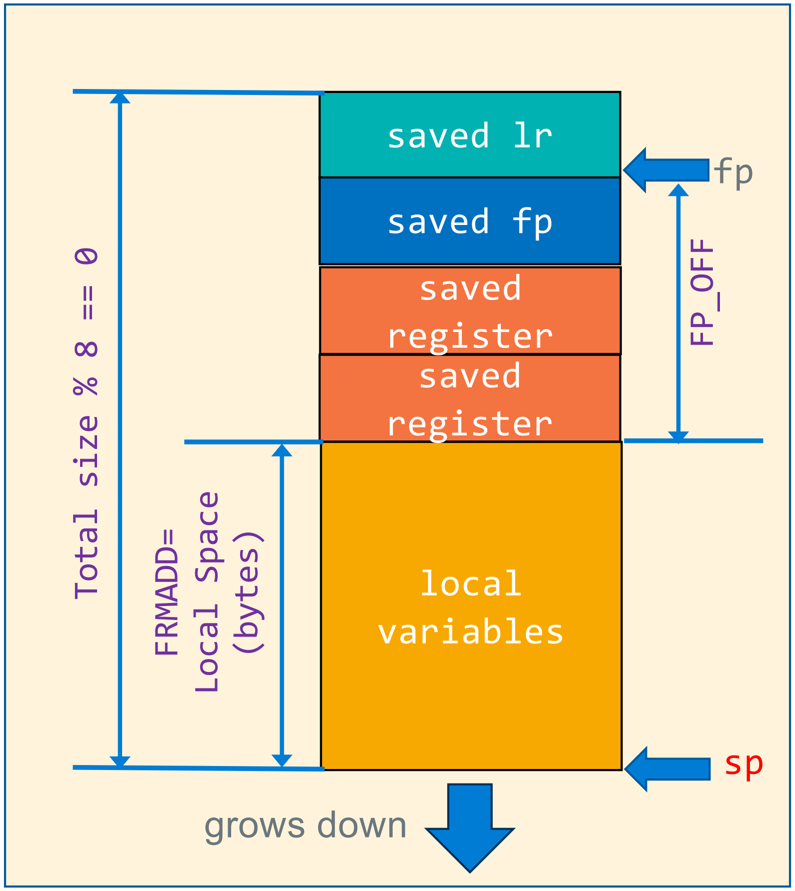


- `sp` always points at top element in the stack (lowest byte address)
- `fp` always points at the bottom element in the stack
  - Bottom element is always the saved `lr` (contains the return address of caller)
  - A saved copy of callers fp is always the next element below the lr
  - fp will be used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

Function Prologue creates stack frame

```
    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
```

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

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

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

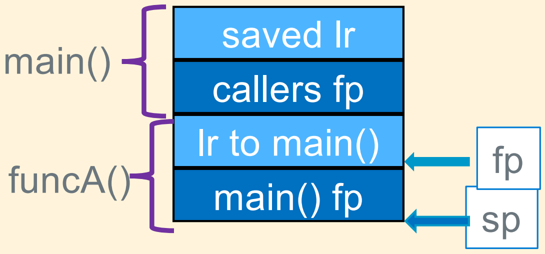
x

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

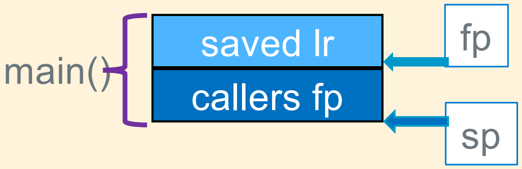
allocate stack space
SP = SP – "space"
grows "down"

main() calls funcA()



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

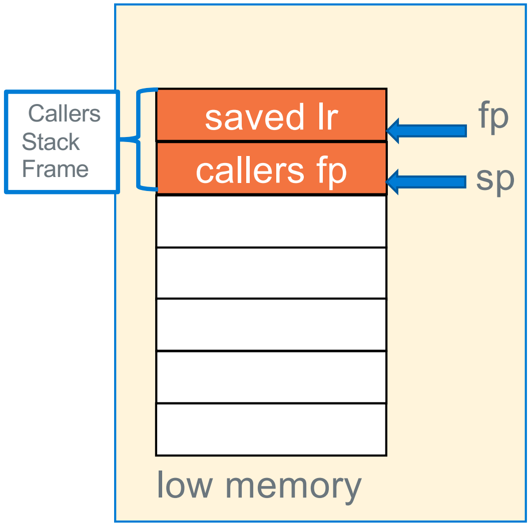
deallocate stack space
SP = SP + "space"
shrinks "up"



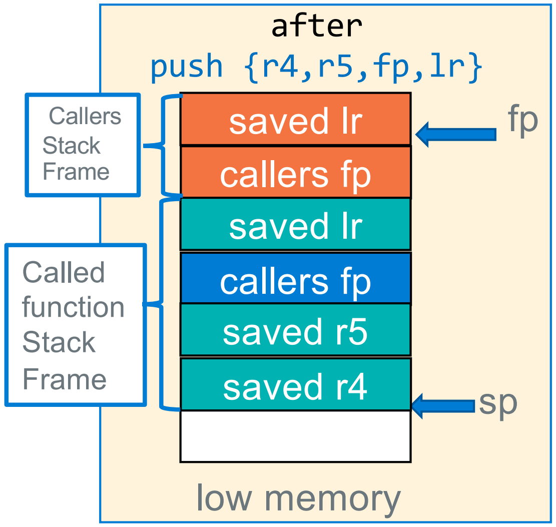
x

Function Prologue: Allocating the Stack Frame -1

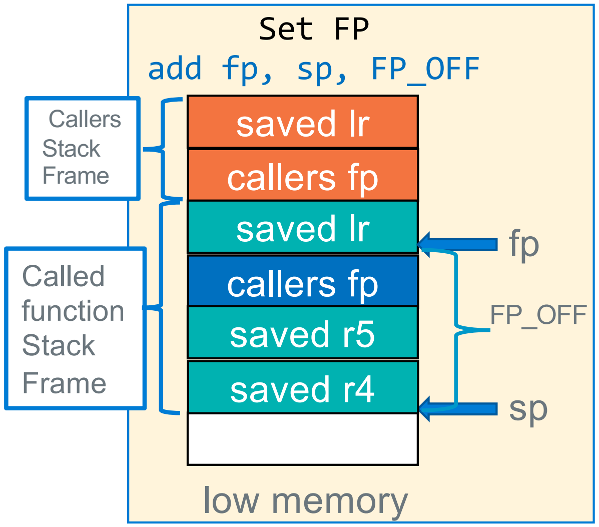
at function entry



Prologue Step 1 of 3



Prologue Step 2 of 3



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

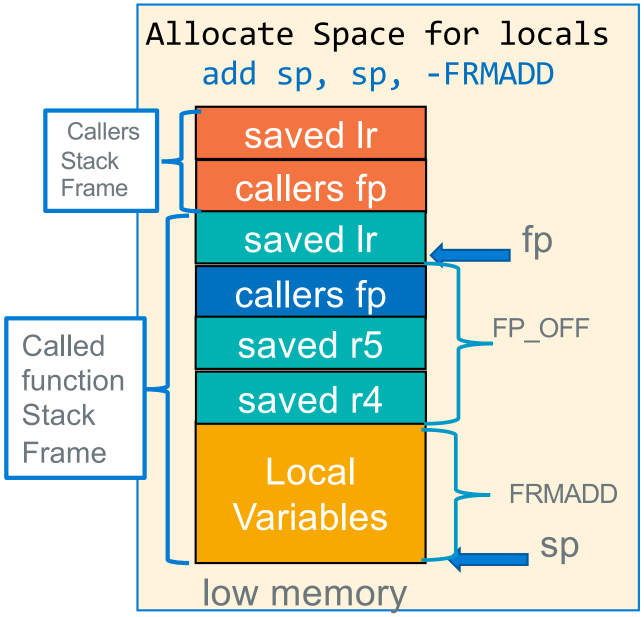
- Space for local variables is allocated on the stack right below the lowest pushed register
- **Add memory to the stack frame for local variables** by **moving** the sp towards low memory
- The amount moved is the total size of all local variables in bytes **plus** memory alignment **padding**

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



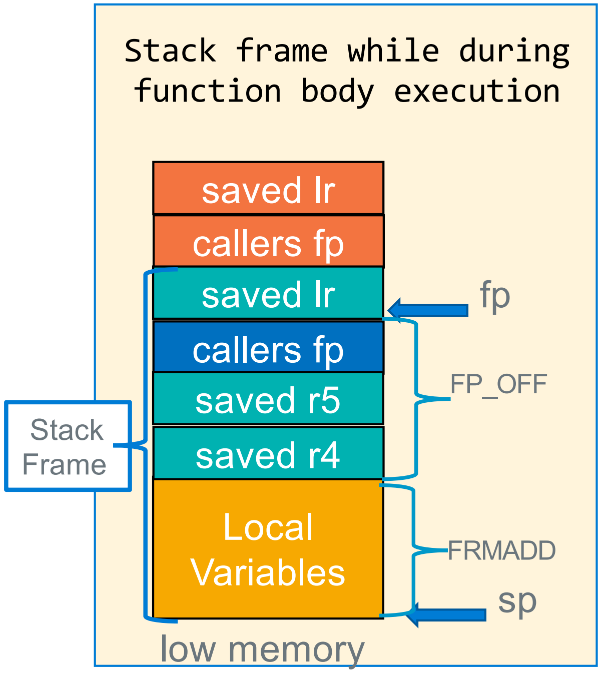
Allocate Space for Local Variables

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
```

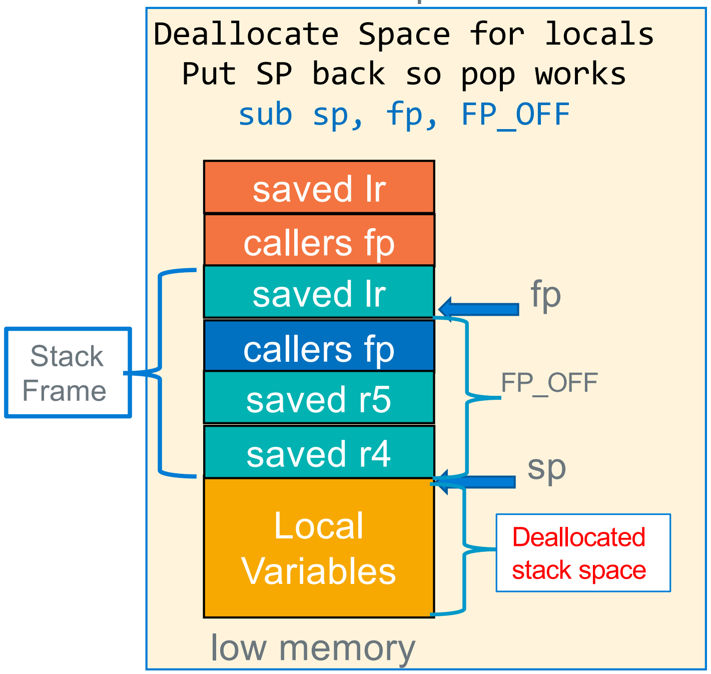
x

Function Epilogue: Deallocating the Stack Frame - 1


Stack frame while during function body execution

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

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)

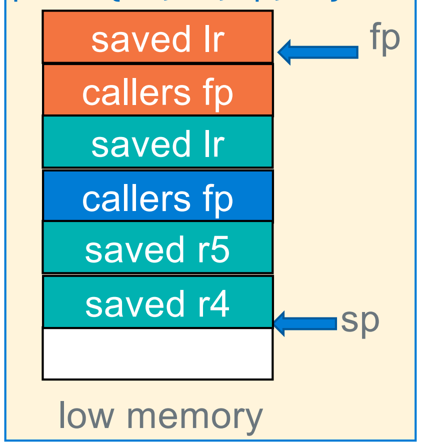
function Epilogue

```
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

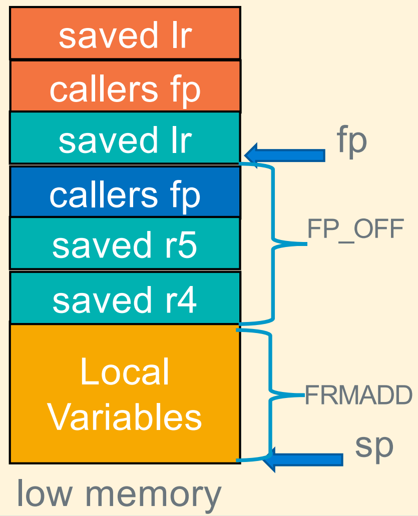
Why You must move SP before POP in the Epilogue

after
push {r4,r5,fp,lr}



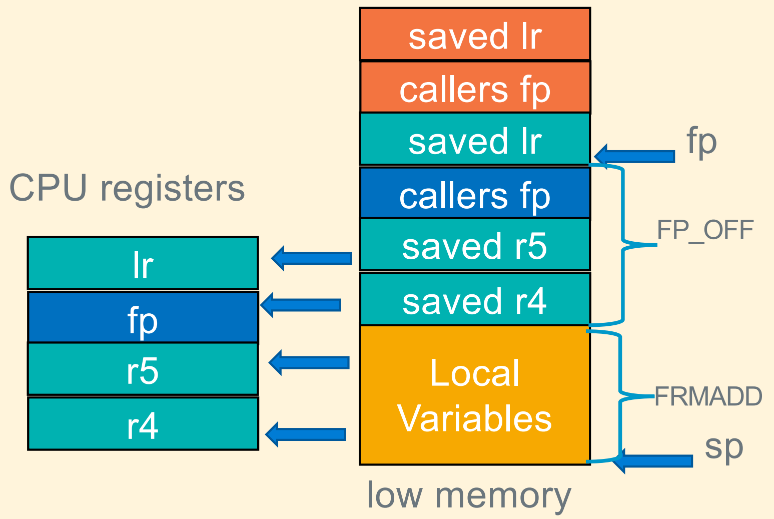
saved lr
callers fp
saved lr
callers fp
saved r5
saved r4
fp
sp
low memory

Allocate Space for locals
add sp, sp, -FRMADD



saved lr
callers fp
saved lr
callers fp
saved r5
saved r4
Local Variables
fp
FP_OFF
FRMADD
sp
low memory

If you do not move sp back!
You get a total mess
pop {r4, r5, fp, lr)



CPU registers
lr
fp
r5
r4
saved lr
callers fp
saved lr
callers fp
saved r5
saved r4
Local Variables
fp
FP_OFF
FRMADD
sp
low memory

function
Epilogue

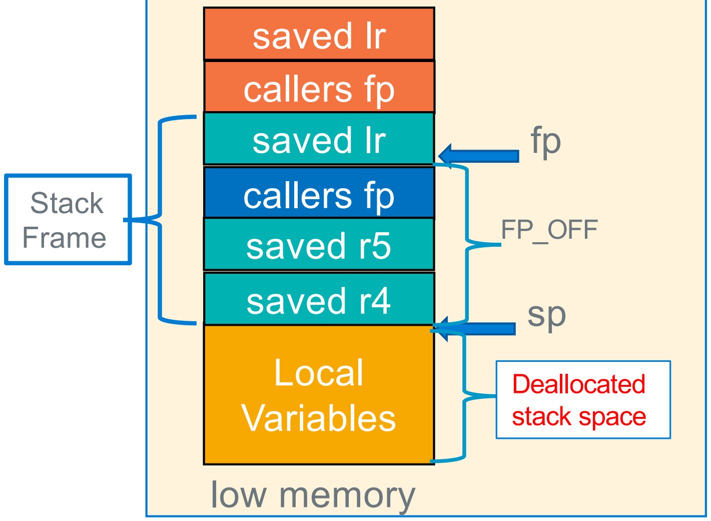
```
sub     sp, fp, FP_OFF         // deallocate local variables by moving sp
pop     {fp, lr}               // pop (restore) fp and lr from stack
bx      lr                     // return to caller
```

x

Function Epilogue: Deallocating the Stack Frame

Step 1

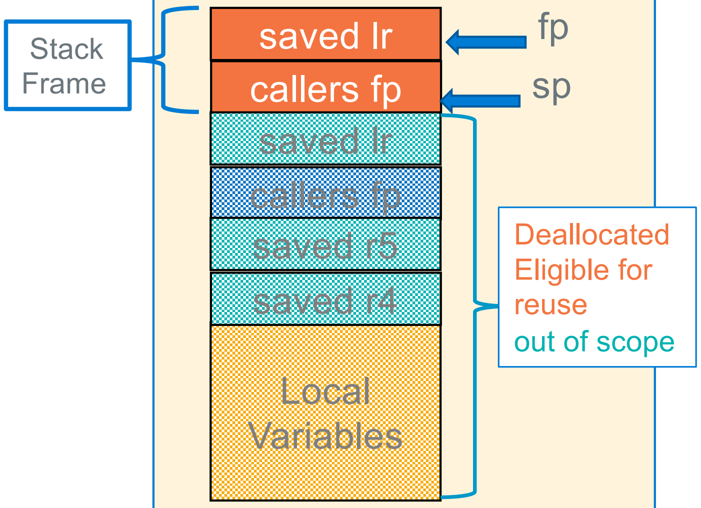
```
Deallocate Space for locals
  Put SP back so pop works
    sub sp, fp, FP_OFF
```



low memory

Step 2

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



low memory

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

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
```

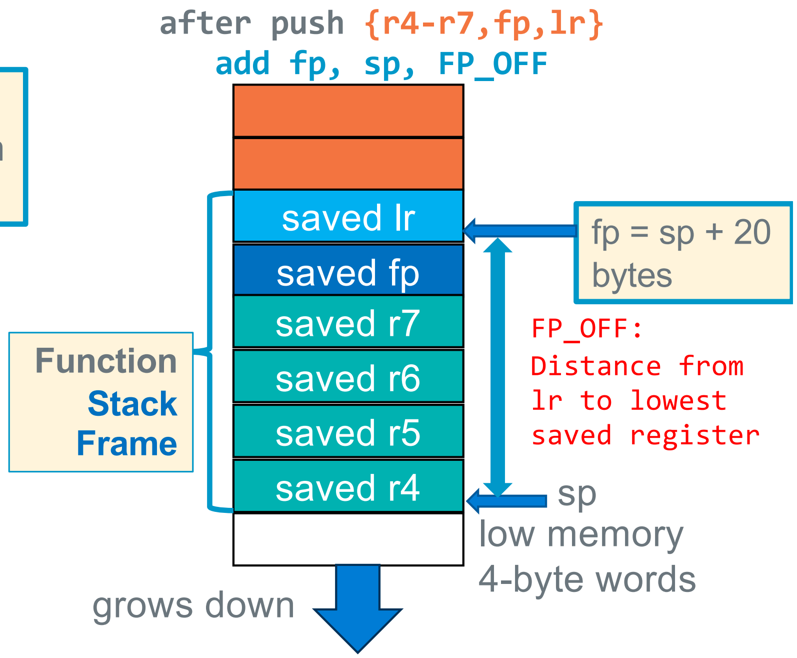
x

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



# 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

```
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

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);

    return EXIT_SUCCESS;
}
```

We are going to put these variables in temporary registers

r0,   r1,   r2

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

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

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

        putchar(c);
        count++;
    }

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

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

r0

r0

r0 r1

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

```c
#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

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
```

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

File header and footers are not shown

x

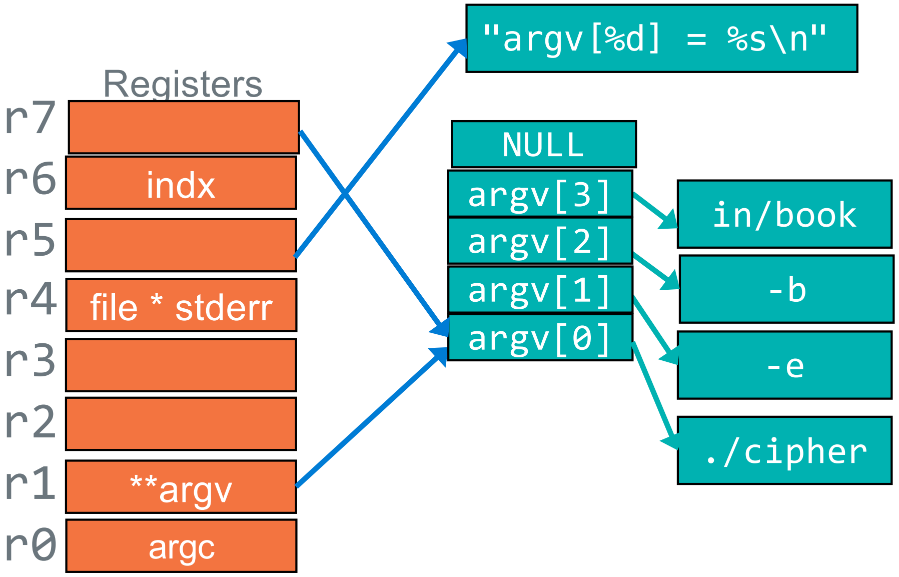
Accessing Pointers (argv) in ARM assembly

```
    .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

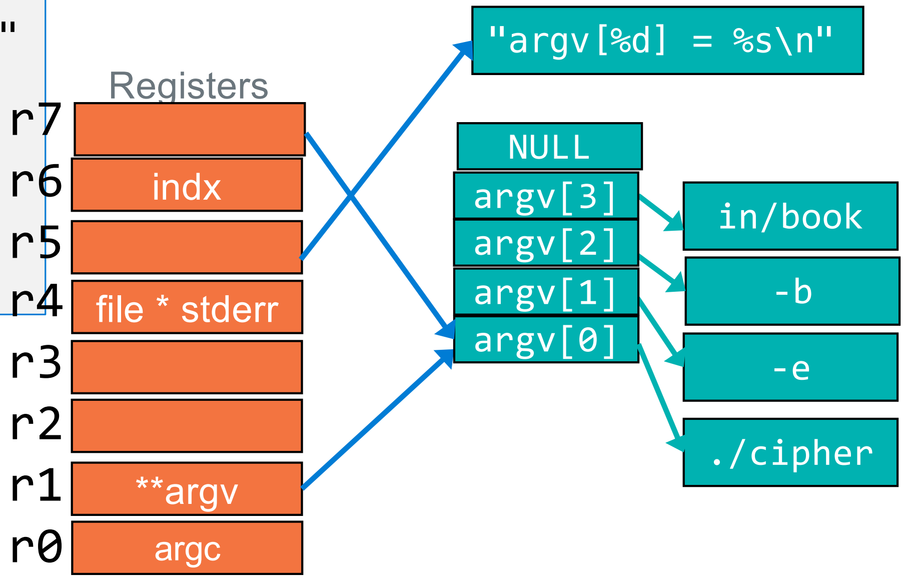
Accessing Pointers (argv) in ARM assembly

```
.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



Registers

	Registers
r7	
r6	indx
r5	
r4	file * stderr
r3	
r2	
r1	**argv
r0	argc

"argv[%d] = %s\n"

NULL
argv[3]
argv[2]
argv[1]
argv[0]

in/book

-b

-e

./cipher

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

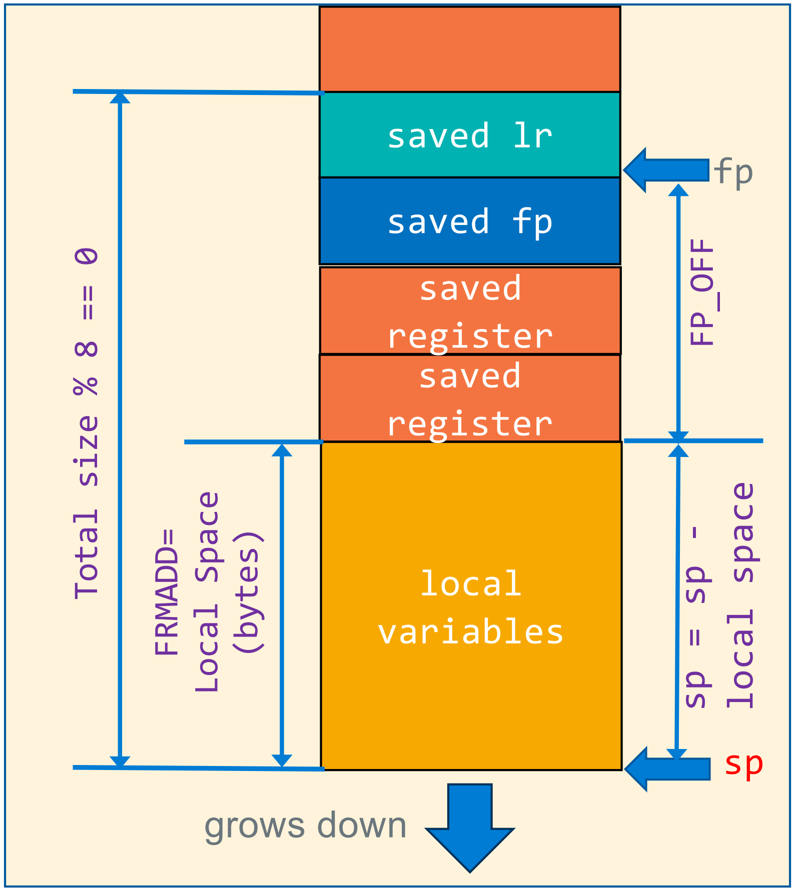
- Allocate the space after the register push by

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

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

  sp % 8 == 0

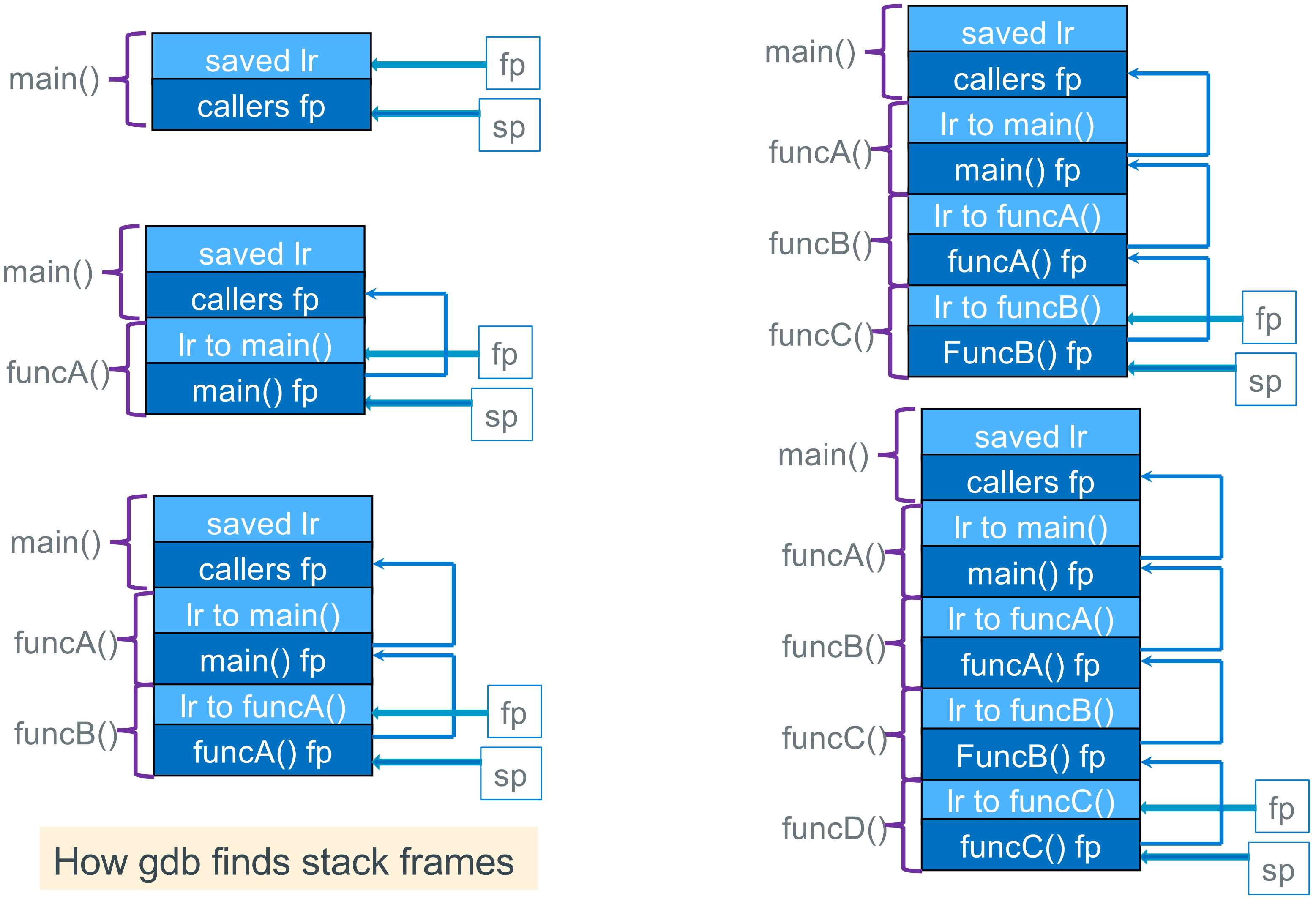
- **Padding (as required):**
  1. Additional space between variables on the stack to meet memory alignment requirements
  2. Additional space so the frame size is evenly divisible by 8
- fp (frame pointer) is used as a **pointer (base register)** to access all stack variables – later slides



x

Extra Slides

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



How gdb finds stack frames

x