Version 1.02

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 4

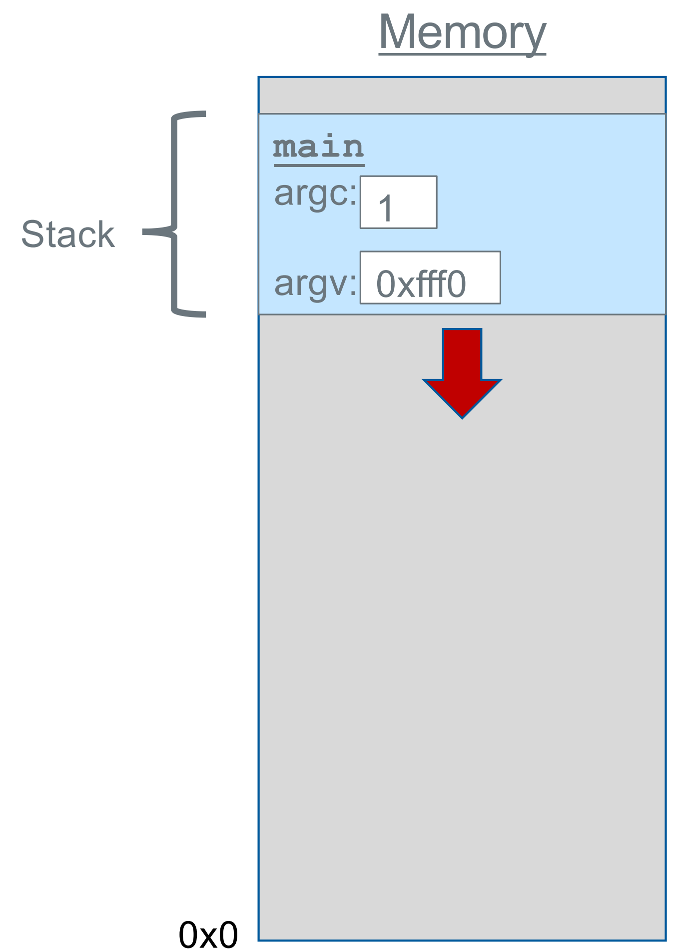
Lecture 19 – November 22, 2022

Keith Muller

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

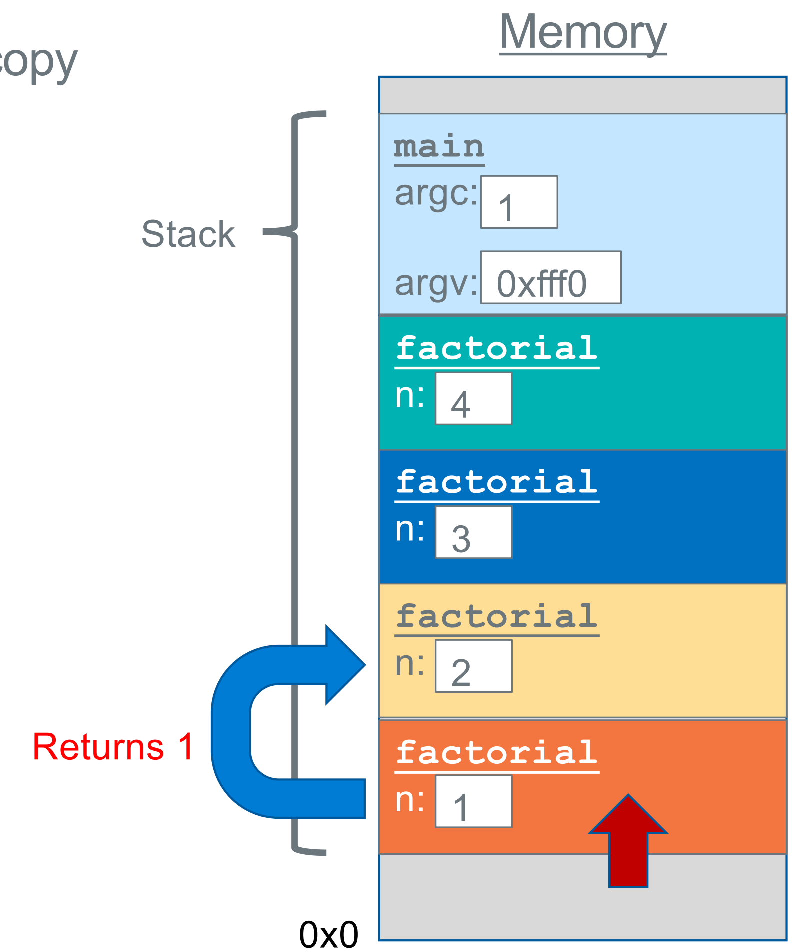
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

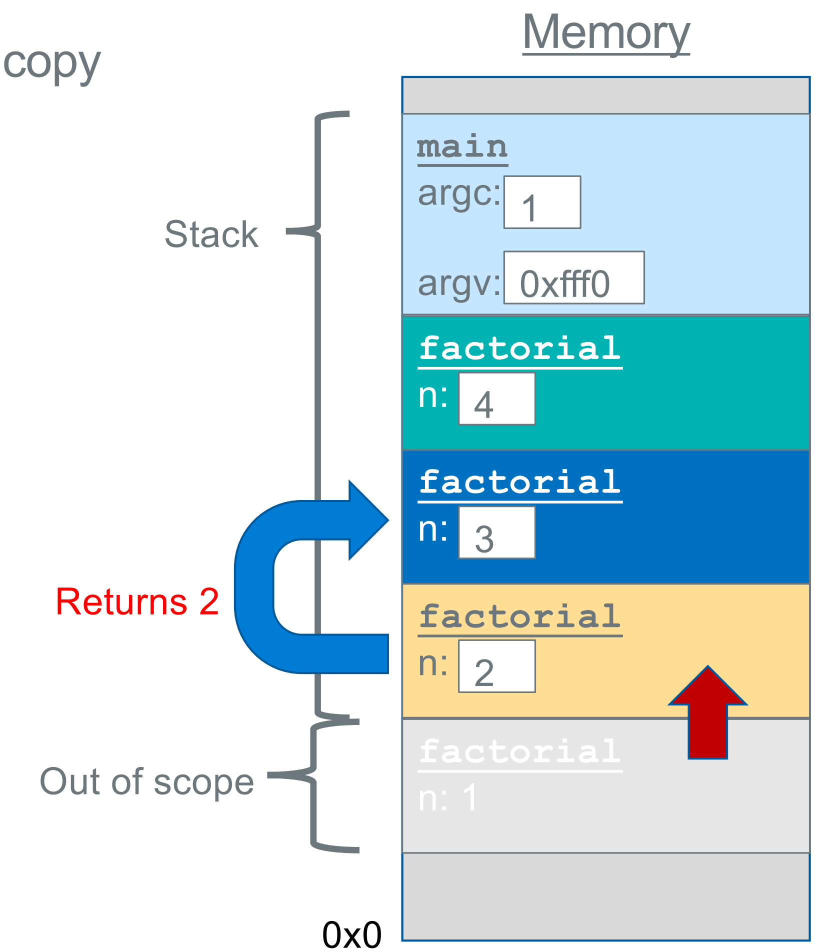
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

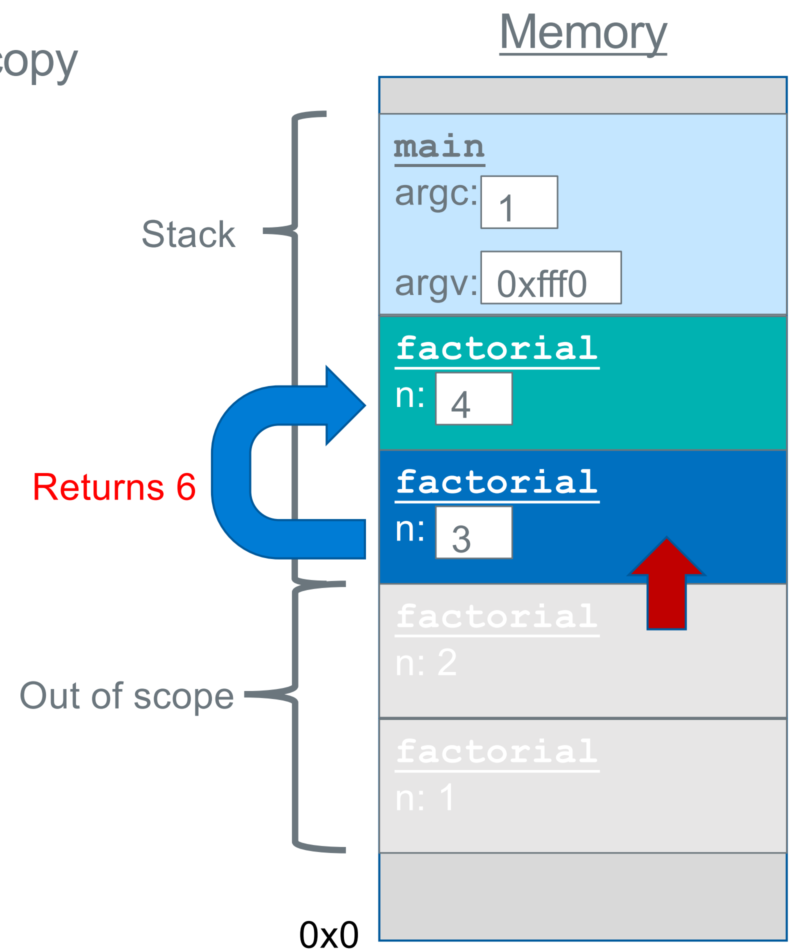
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

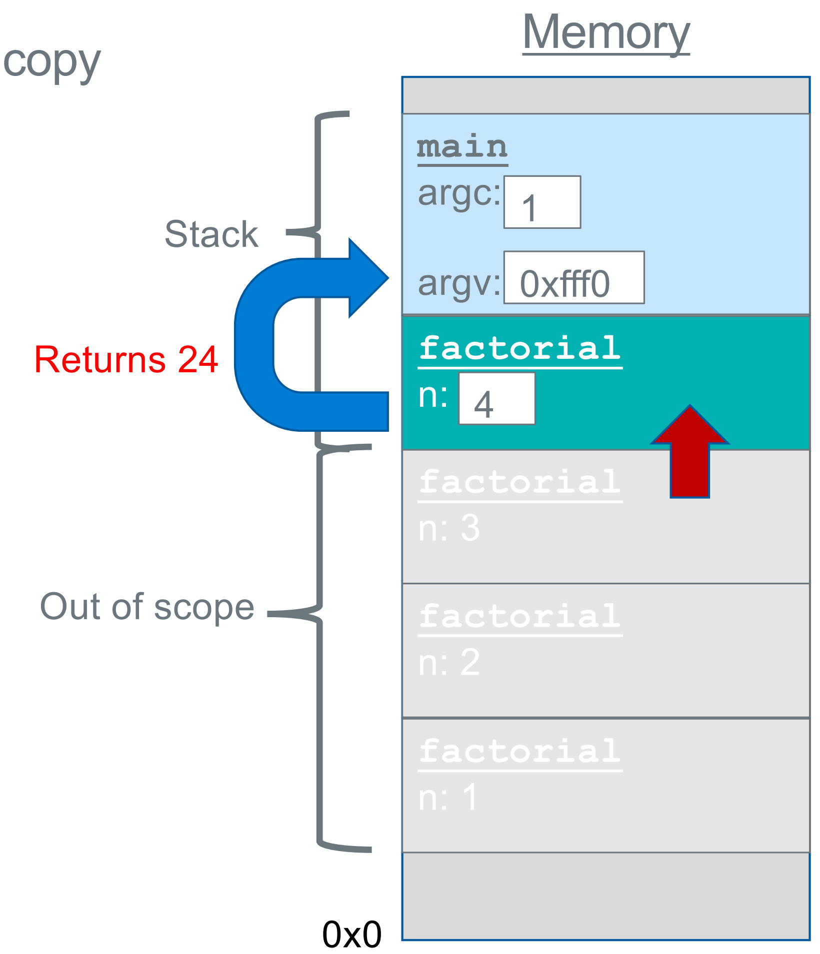
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

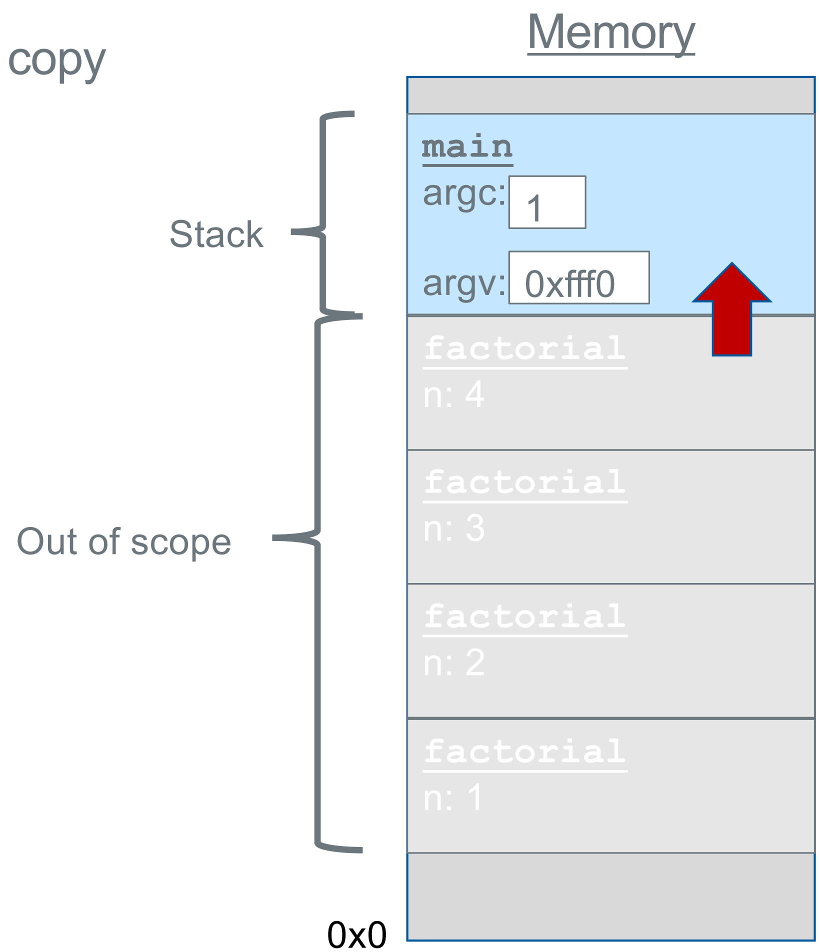
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

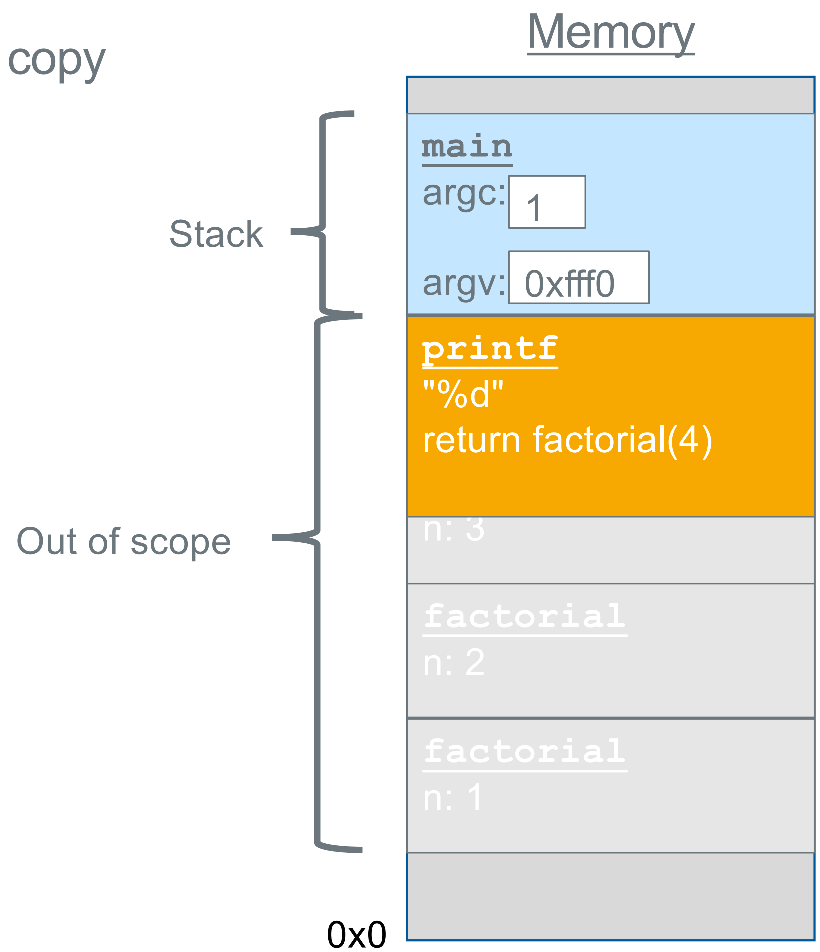
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

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

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



Ghost of Stack Frames Past…..

same stack frame
variable layout

```
% ./a.out
before ghost: 0 66328
after ghost: 30 300
wraith: 30 300
%
```

See how wraith has the old values left over from the prior call to ghost

```c
void ghost(int n)
{
    int x;
    int y;

    printf("before ghost: %d %d\n", x, y);
    x = 10*n;
    y = 100*n;
    printf("after ghost: %d %d\n", x, y);
    return;
}

void wraith (void)
{
    int a;
    int b;

    printf("wraith: %d %d\n", a, b);
    return;
}

int main(void)
{
    ghost(3);
    wraith();
    return EXIT_SUCCESS;
}
```

Function Header and Footer Assembler Directives

function entry point
address of the first instruction in the function
Must not be a local label (does not start with .L)

```
                .text
Function        .global myfunc              // make myfunc global for linking
  Header        .type   myfunc, %function  // define myfunc to be a function
                .equ    FP_OFF,  4          // fp offset in main stack frame
myfunc:
                 // function prologue, stack frame setup
                 // your code
                 // function epilogue, stack frame teardown
Function
  Footer        .size myfunc, (. - myfunc)
```

`.global function_name`

- Exports the function name to other files. **Required for main function,** optional for others

`.type name, %function`

- The `.type` directive sets the **type of a symbol/label name**
- `%function` specifies that **name** is a function (name is the address of the first instruction)

`equ FP_OFF, 4`

- Used for basic stack frame setup; the number 4 will change – later slides

`.size name, bytes`

- The `.size` directive is used to set the size associated with a symbol
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- `bytes` **is best calculated as an expression: (period is the current address in a memory segment)**

  In CSE30 required use: `.size name, (. - name)`

X

Support For Function Calls and Function Call Return - 1

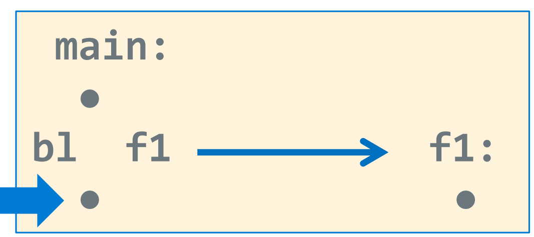
bl	imm24

Branch with Link (function call) instruction

```
bl label
```

- Function call to the instruction with the address `label` (no local labels for functions)
  - imm24 number of instructions from pc+8
- label **any function label** in the current file, or any function label that is defined as .global in any file that it is linked to
- BL **saves** the address of the instruction **immediately** following the **bl** 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

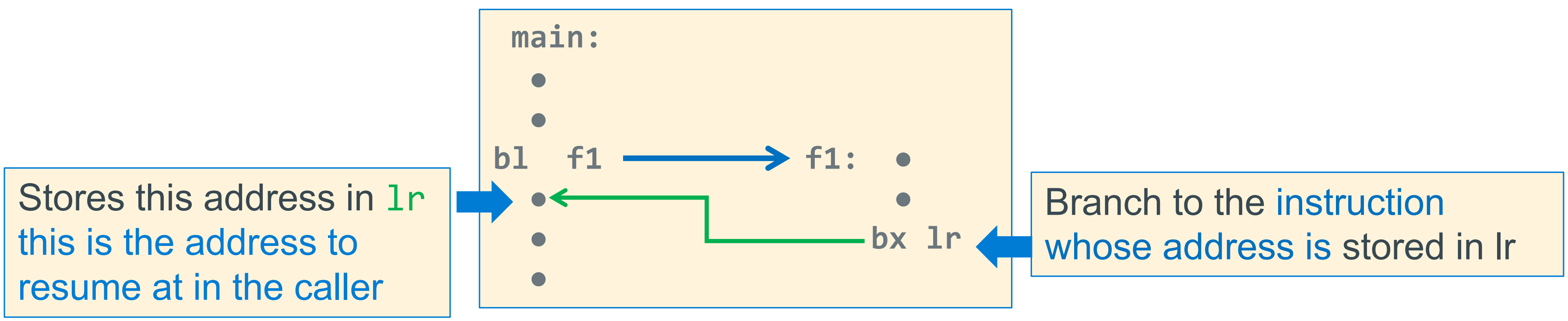
Support For Function Calls and Function Call Return - 2

bx	Rn

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 **bl label**



x

bl and bx operation working together

```c
int main(void)
{
     a();
     // other code
     a();
     return EXIT_SUCCESS;
}
int a(void)
{
    // other code
    return 0;
}
```

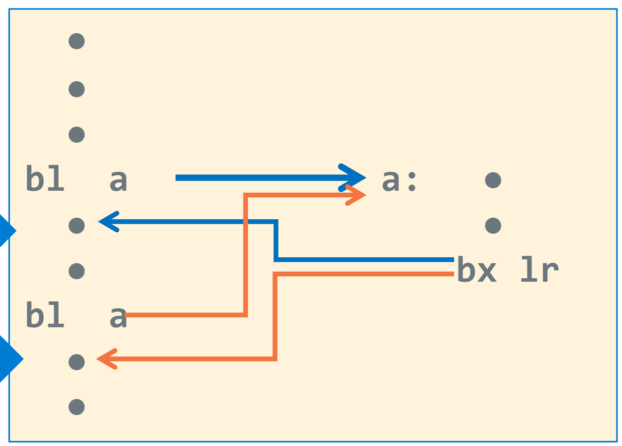
```
        .text
        .type   main, %function
        .global main
         .equ    EXIT_SUCCESS, 0
main:
        // code
        bl      a
ra1     // other code
        bl      a

ra2     mov     r0, EXIT_SUCCESS
        // code
        bx      lr
        .size main, (. - main)


        .type   a, %function
a:
        // code
        mov     r0, 0
        // code
        bx      lr
        .size a, (. - a)
```

ra1 lr

ra2 lr

ra2 lr

address of next instruction is stored in lr



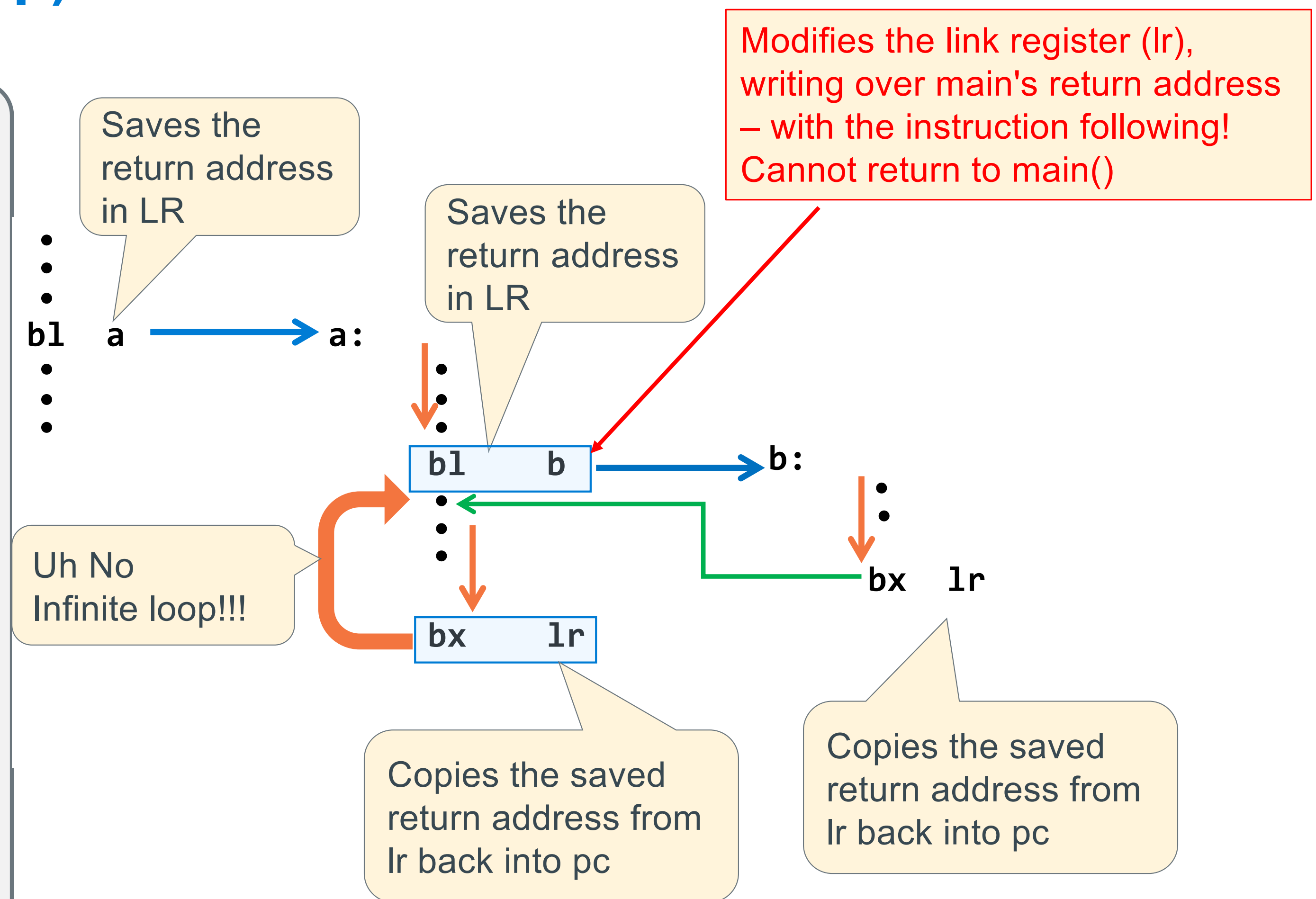
address of next instruction is stored in lr

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

x

Preserving lr (and fp): The Foundation of a stack frame

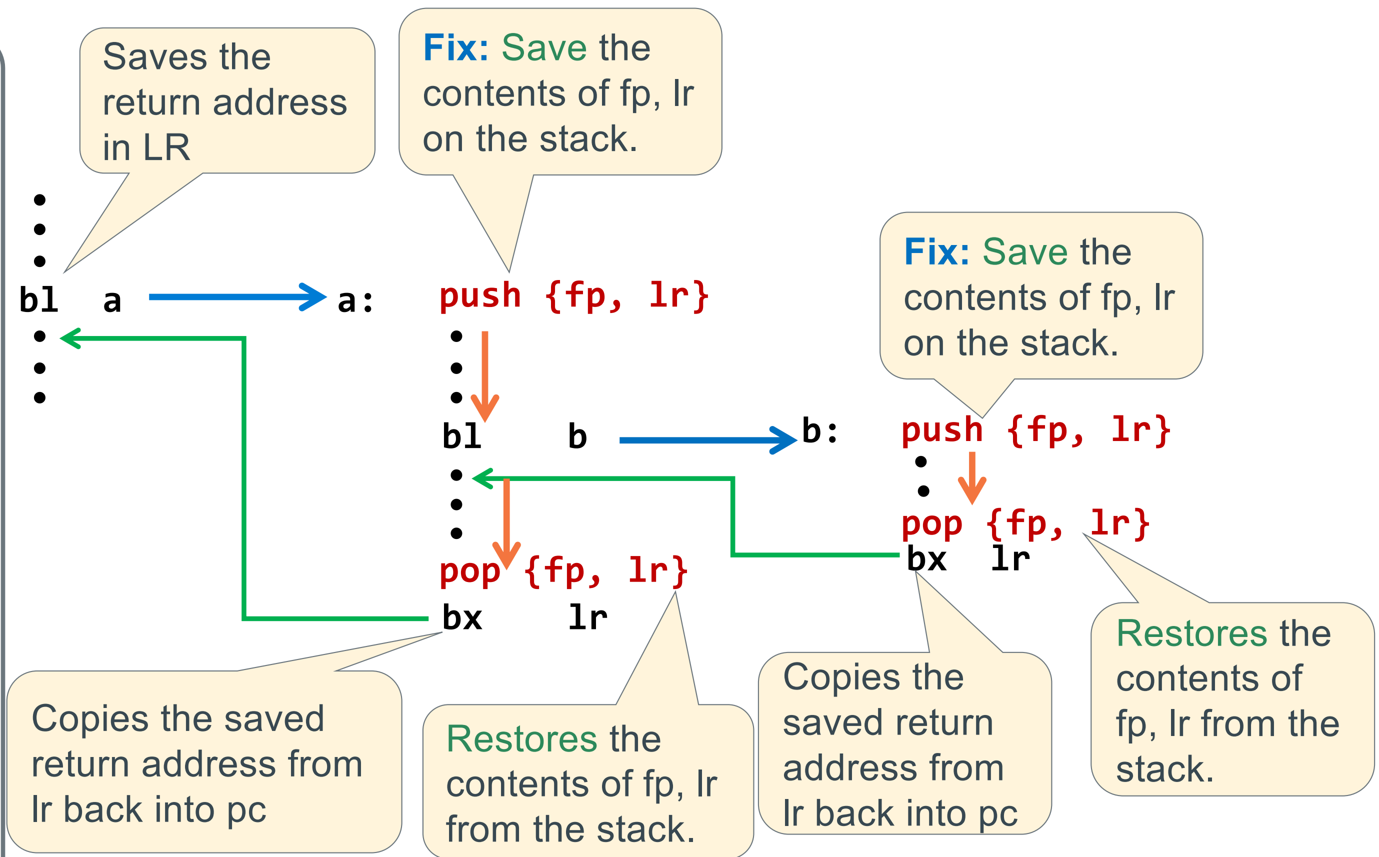
```
int main(void)
{
     a();
     /* other code */
     return EXIT_SUCCESS;
}

int a(void)
{
    b();
    /* other code */
    return 0;
}

int b(void)
{
    /* other code */
    return 0;
}
```



x

Preserving lr (and fp): The Foundation of a stack frame

```
int main(void)
{
     a();
     /* other code */
     return EXIT_SUCCESS;
}

int a(void)
{
    b();
    /* other code */
    return 0;
}

int b(void)
{
    /* other code */
    return 0;
}
```



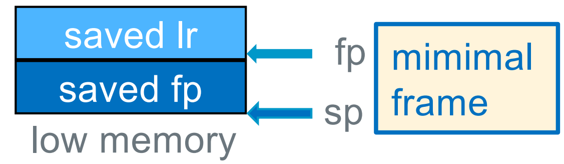
The frame pointer is used to find variables on the stack – later

x

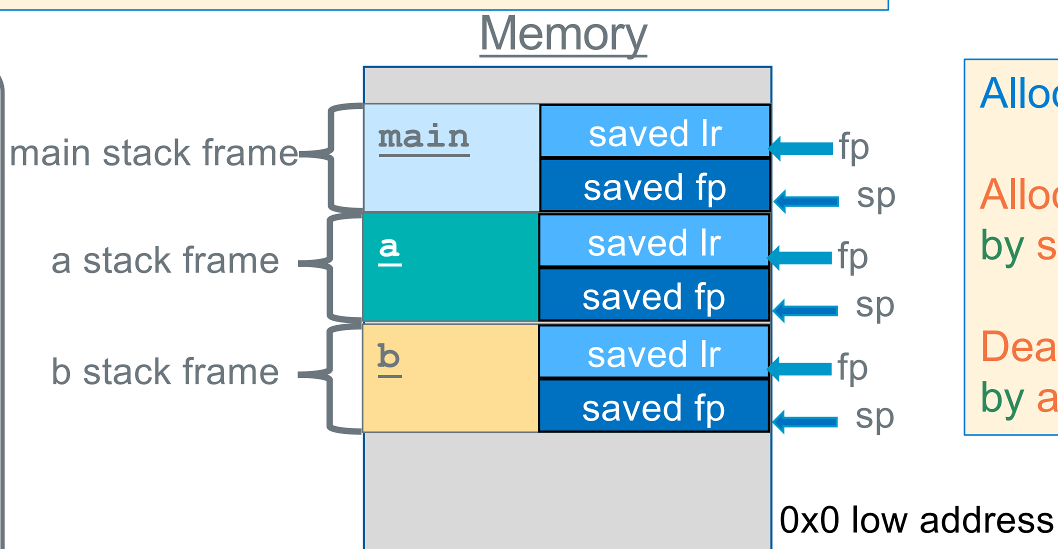
Minimal Stack Frame (Arm Arch32 Procedure Call Standards)

Stack Frame Requirements

- `sp` points at top element in the stack (lowest byte address)
- `fp` points at the `lr` copy stored in the current stack frame
- **Stack frames MUST ALWAYS BE aligned to 8-byte addresses**
  - So, this must always be true: sp % 8 == 0




```
int main(void)
{
     a();
     /* other code */
     return EXIT_SUCCESS;
}
int a(void)
{
    b();
    /* other code */
    return 0;
}
int b(void)
{
    /* other code */
    return 0;
}
```



Allocating stack space

Allocate stack space by subtracting from sp

Deallocate stack space by adding to sp

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

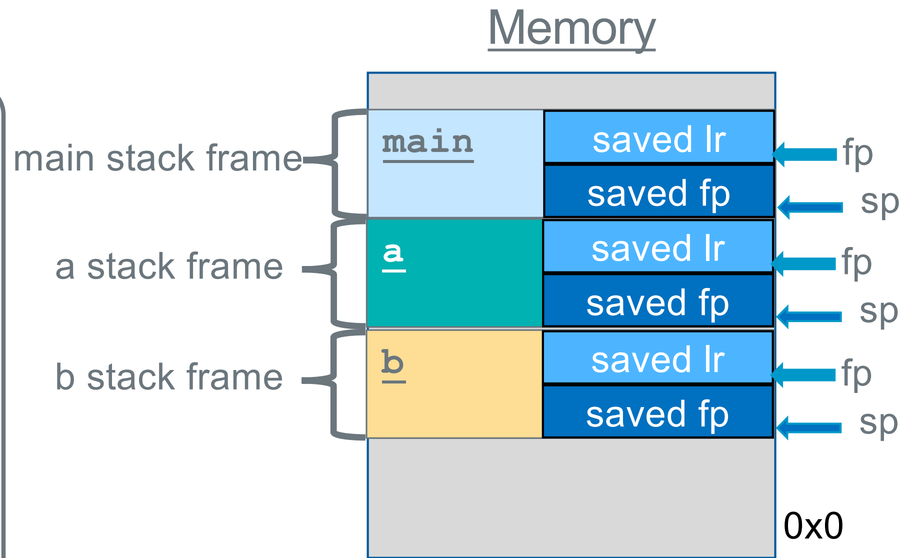
deallocate stack space
SP = SP + "space"
shrinks "up"

We will see how the fp is used in a few slides

x

Minimal Stack Frame (Arm Arch32 Procedure Call Standards)

```
int main(void)
{
     a();
     /* other code */
     return EXIT_SUCCESS;
}
int a(void)
{
    b();
    /* other code */
    return 0;
}
int b(void)
{
    /* other code */
    return 0;
}
```



We will see how the fp is used in a few slides

- Stack Space management Approach
- Function entry (Function **Prologue**):
  1. creates the frame (subtracts from sp)
  2. saves values
- Function return (Function **Epilogue**):
  1. restores values
  2. removes the frame (adds to sp)

x

Review Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	*Function Call Use*
r0	1st parameter
r1	2nd parameter
r2	3rd parameter
r3	4th parameter

Register	*Function Return Value Use*
r0	8, 16 or 32-bit result, 32-bit address or least-significant half of a 64-bit result
r1	most-significant half of a 64-bit result

- Where `r0, r1, r2, r3` are arm registers, the function declaration is (first four arguments):

```
r0 = function(r0, r1, r2, r3)        // 32-bit return
r0, r1 = function(r0, r1, r2, r3)    // 64-bit return - long long
```

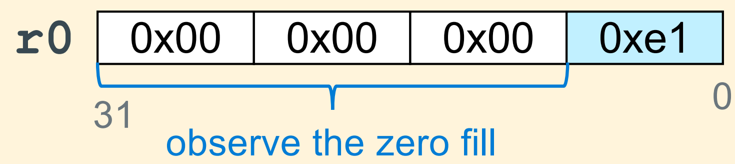
- Each **parameter and return value is limited to data that can fit in 4 bytes or less**
- You receive up to the first four parameters in these four registers
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes, a pointer to the parameter is passed (we will cover this later)
- **You MUST ALWAYS assume** that the called function will **alter the contents of all four registers: r0-r3**
- **Observation: When a function calls another function, the called function has the right to overwrite the first 4 parameters that were passed to it by the calling function**

Argument and Return Value Requirements

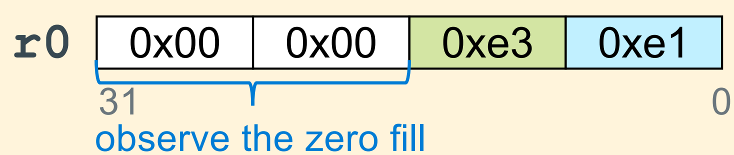
- When passing or returning values from a function you must do the following:

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 zero filled

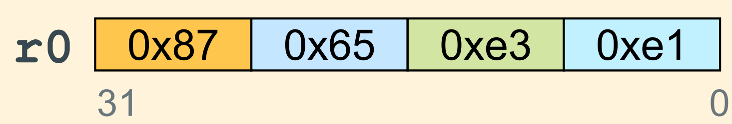
Single Byte



Single Halfword



Full Word



Preserved Registers: Protocols for Use

Register	*Function Call Use*	*Function Body Use*	*Save before use Restore before return*
r4-r10		contents preserved across function calls	Yes
r7	os system call number	contents preserved across function calls	Yes

- **Function Call Spec**:

  Preserved registers **will not be changed** by any function you call

- **Interpretation**: 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)

x

Preserved Registers: When to Use?

Register	*Function Call Use*	*Function Body Use*	*Save before use Restore before return*
r4-r10		contents preserved across function calls	Yes
r7	os system call number	contents preserved across function calls	Yes

- When to use a preserved register in a function you are writing:

1. Values that you want to protect from being changed by a function call
   a) Local variables stored in registers
   b) Parameters passed to you (in `r0-r3`) that you need to continue to use after calling another function
2. Need more than `r0-r3` whether you call another function or not

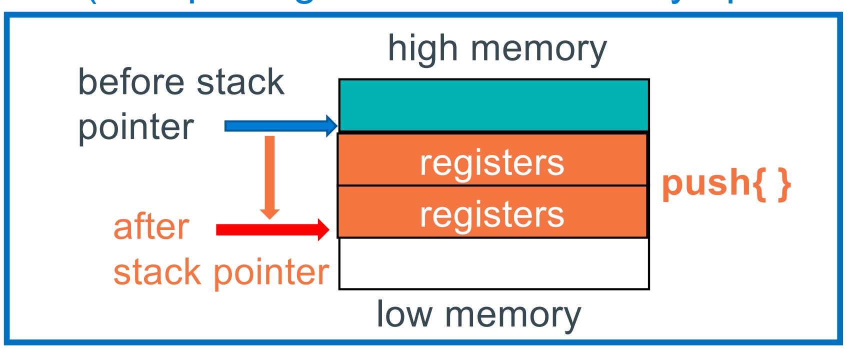
   Options are:
   a) preserved register *or*
   b) stack local variable (later slides)

x

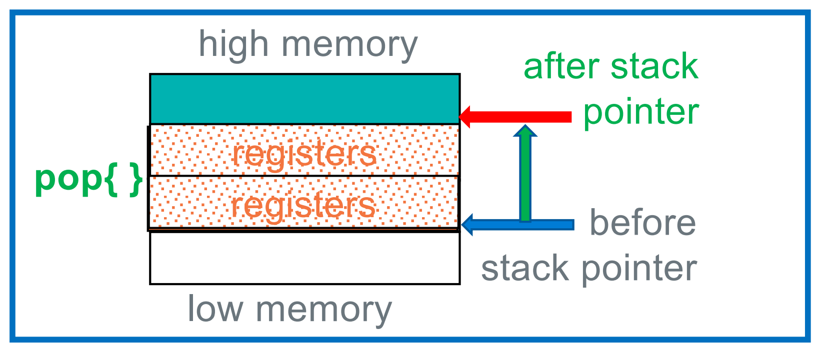
Preserving and Restoring Registers by copying to/from Stack
Moves sp to allocate (Push) or deallocate (pop) stack space

Operation	***Pseudo Instruction (Use in CSE30)***	ARM instruction (reference only)	***Operations***
Push registers onto stack Function Entry	`push {reg list}`	`stmfd sp!, {reg list}`	`sp = sp – 4 × #registers` `Copy registers to mem[sp]`
Pop registers from stack Function Exit	`pop {reg list}`	`ldmfd sp!, {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
Function entry and Function exit

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`

- Where **`{reg list}`** is a **list of registers** in <u>numerically increasing order</u>

  `example: push {r4-r10, fp, lr}`

- Registers cannot be: (1) duplicated in the list, nor be (2) listed out of numeric order
- Register ranges can be specified `{r4, r5, r8-r11, fp, lr}`

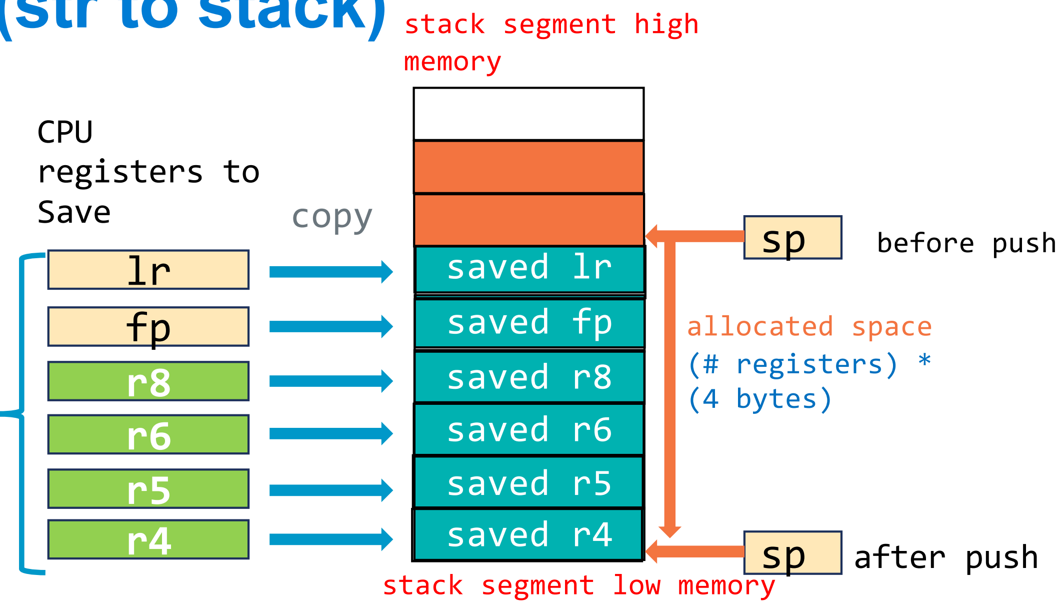
x

push: Multiple Register Save (str to 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)**

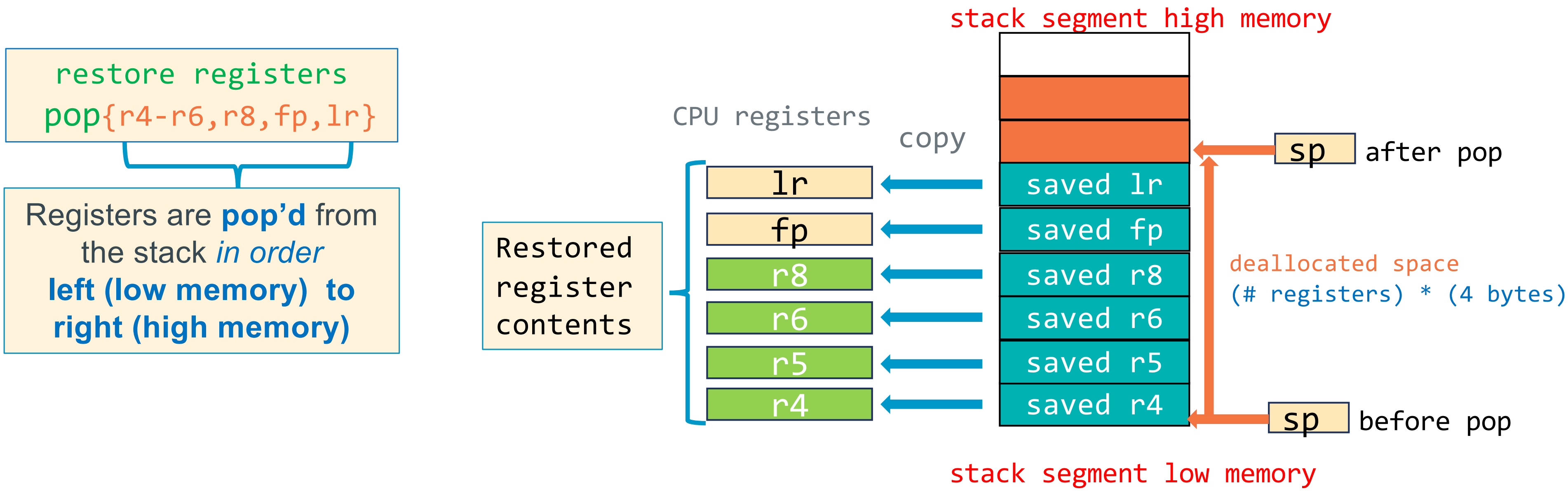
If you have no stack variables (later) then always push an **EVEN** number of registers!



- `push` copies the contents of the `{reg list}` to stack segment memory
- `push` **Also** 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 (ldr from stack)

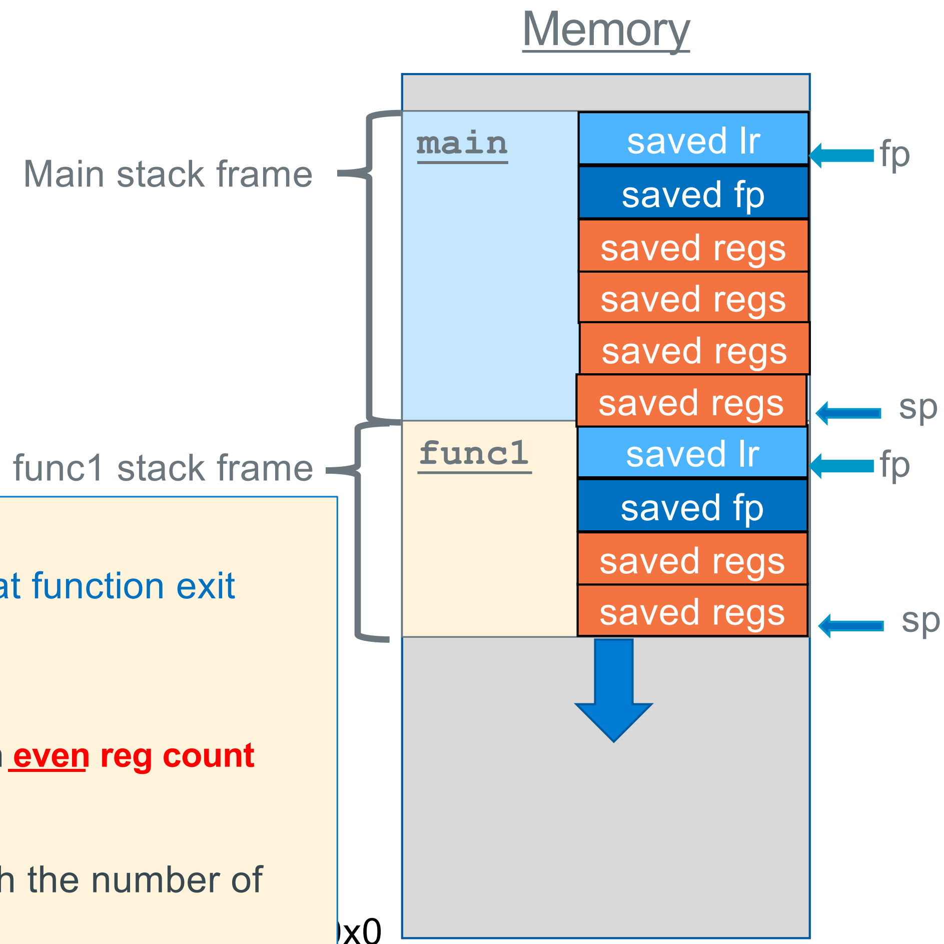


- `pop` copies the contents of stack segment memory to the `{reg list}`
- `pop` **adds:** (# of registers restored) * (4 bytes) to `sp` to ***deallocate*** space on the stack
  - sp = sp + (# registers restored * 4)
- `Remember:` `{reg list}` must be the same in both the `push` and the corresponding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

x

Basic Stack Frames (Arm Arch32 Procedure Call Standards)

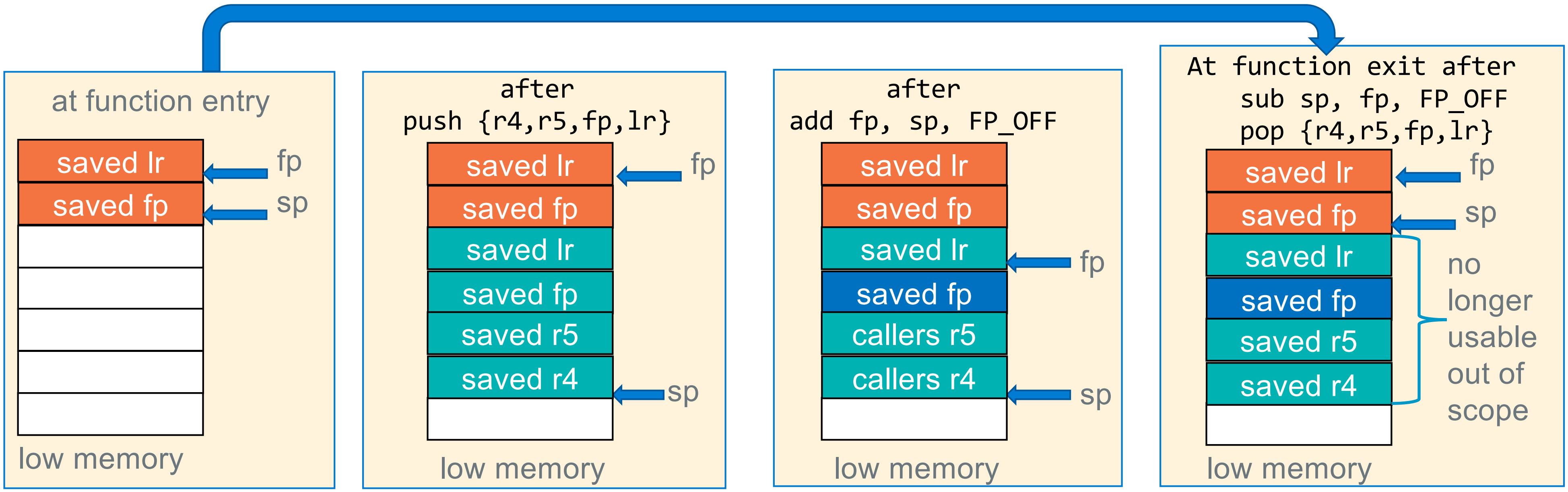
```
void func1() {
    int c = 99;
}
int main(int argc, char **argv)
{
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return EXIT_SUCCESS;
}
```



- **On each function call start (entry)**
  - Preserved registers: push at function entry and pop at function exit
- **Rules**
  - Keep `sp` 8-byte aligned.
    - No stack (local) variables: **round up** `{reg list}` to an **even reg count**
  - **Remember** `fp` must always points at the saved `lr`
- Issue: number of registers saved on the stack varies with the number of registers in the `{reg list}`
  - **In a few slides...**

x

Saving/Restoring Preserved Registers Prologue & Epilogue



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

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

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

At function exit (in the function epilogue) the function uses pop to restore the registers to the values they had at function entry

Part of function prologue

Part of function epilogue

x

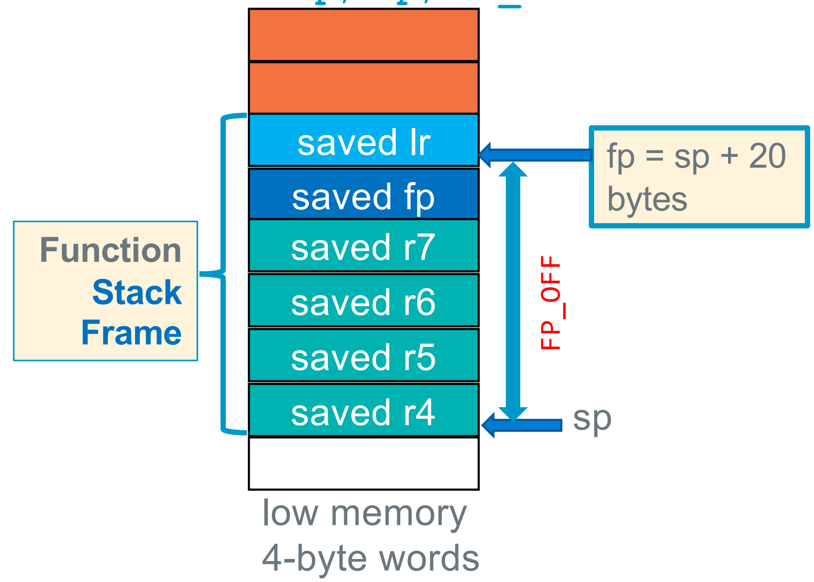
Setting FP_OFF: Distance from FP to SP

```
    // 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
```

always at top of function saves regs and sets fp

always at bottom of function restores regs including the sp

`after push {r4-r7,fp,lr}`
`add fp, sp, FP_OFF`



# regs saved	FP_OFF in Bytes
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

```
FP_OFF = (#regs - 1)*4 // -1 is lr offset from sp

Where # regs = #preserved + lr + fp
```

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

x

Function Prologue and Epilogue: Minimum Stack Frame

- **Function prologue** creates stack frame
  1. push/save registers (**`lr`** & **`fp`** minimum) on stack
  2. set **`fp`** **`(add fp, …)`** to point at the saved lr as required for use by this function (later)
- **Function epilogue** removes stack frame
  1. set **`sp`** to where it was at the push (we may have **moved sp** to allocate space, later slides)
  2. pop/restore registers (**`lr`** & **`fp`** minimum) from stack
- In this example fp is 4 bytes from sp, (FP_OFF) but this will vary…

smallest frame has:
4 bytes between sp & fp

saved lr ← fp
saved fp ← sp

low memory

fp must point at saved lr

```
        .global one
        .type   one, %function
        .equ    FP_OFF,  4

one:
        push    {fp, lr}
        add     fp, sp, FP_OFF

        // your code

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

        bx      lr  // func return

        .size one, (. - one)
```

Function Header
Assembly directives

Position the fp

Function Prologue
always at top of function

Restore the sp

Function Epilogue
always at bottom of function

Function Footer
Assembly directive

x

Allocating Local Variables on the stack

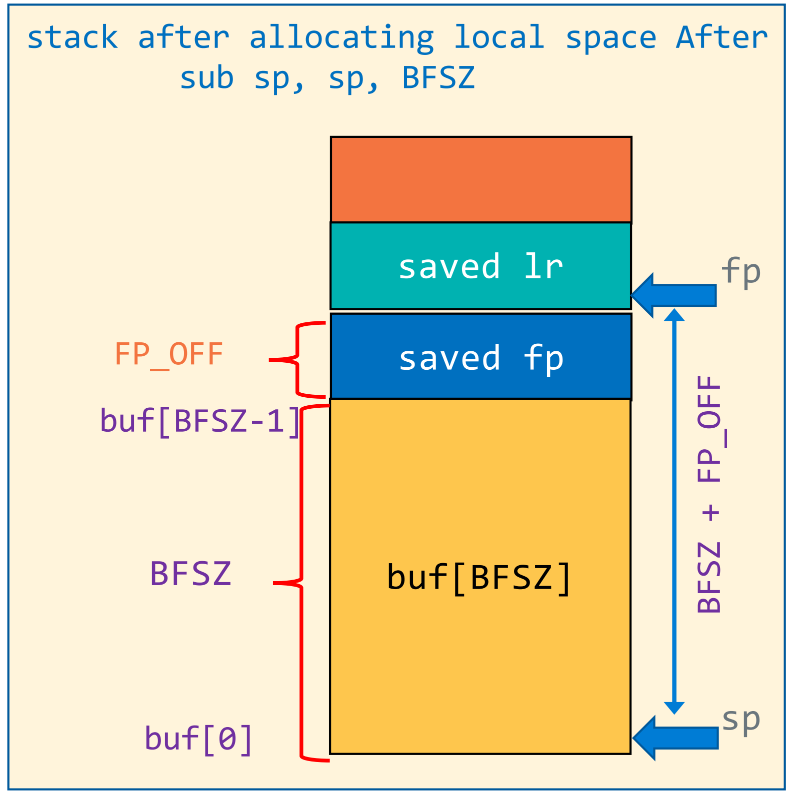
1. Calculate how much additional space is needed by local variables
2. **After the push, Subtract from the sp** the size of the variable in bytes (+ padding - later slides)
3. If the variable has an initial value specified: add code to set the initial value
   a) mov and str are useful for initializing simple variables
   b) **loops** of mov and str to initialize arrays

```
#define BFSZ 256
int main(void)
{
  char buf[BFSZ]; // BFSZ bytes
...
```

```
        .equ    FP_OFF, 4
        .equ    BFSZ, 256

main:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        ldr     r3, =BFSZ
        sub     sp, sp, r3
```

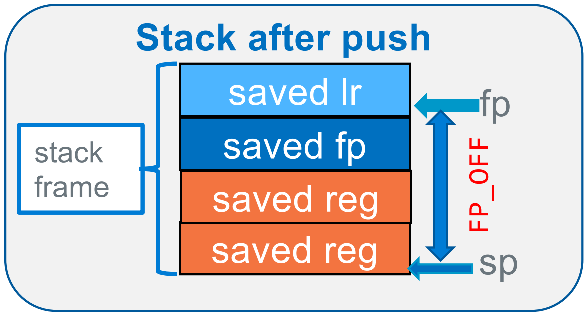
Function Prologue Extended

allocate space for buf[256]

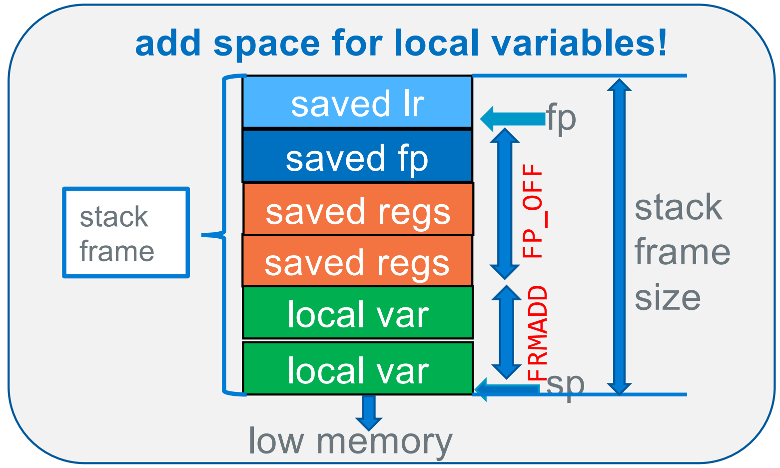


x

Function prologue with local variables




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

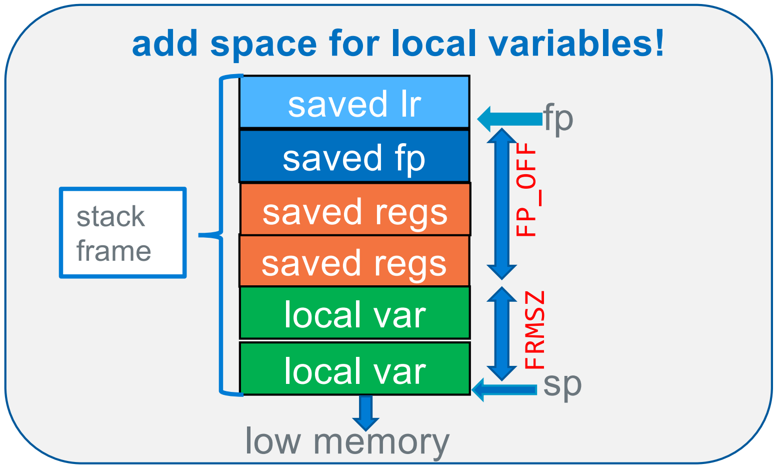


- move the sp to allocate space on the stack for local variables and outgoing parameters (later)

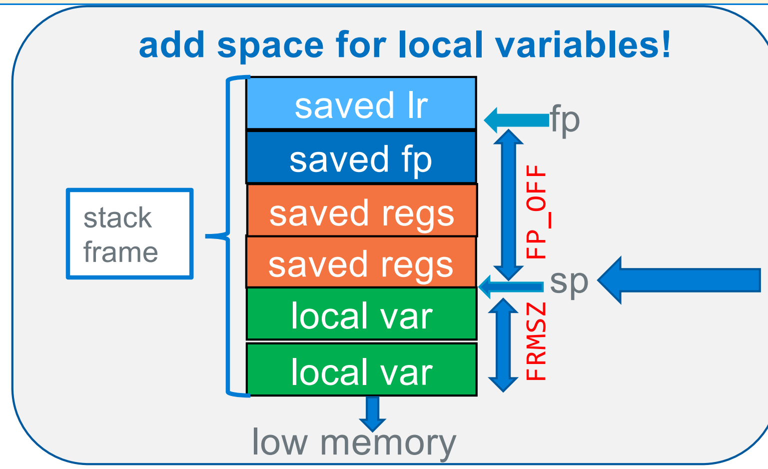
```
    .equ    FRMADD, 8
    push    {fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD // frames may be large
    sub     sp, sp, r3
// your code
```


x

Function epilogue with local variables



- For pop to restore the registers correctly:
  - sp must point at the last saved preserved register put on the stack by the save register operation: the push




```
.equ    FRMADD, 8
push    {fp, lr}
add     fp, sp, FP_OFF
ldr     r3, =FRMADD
sub     sp, sp, r3
  // your code

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

bx      lr  // func return
```

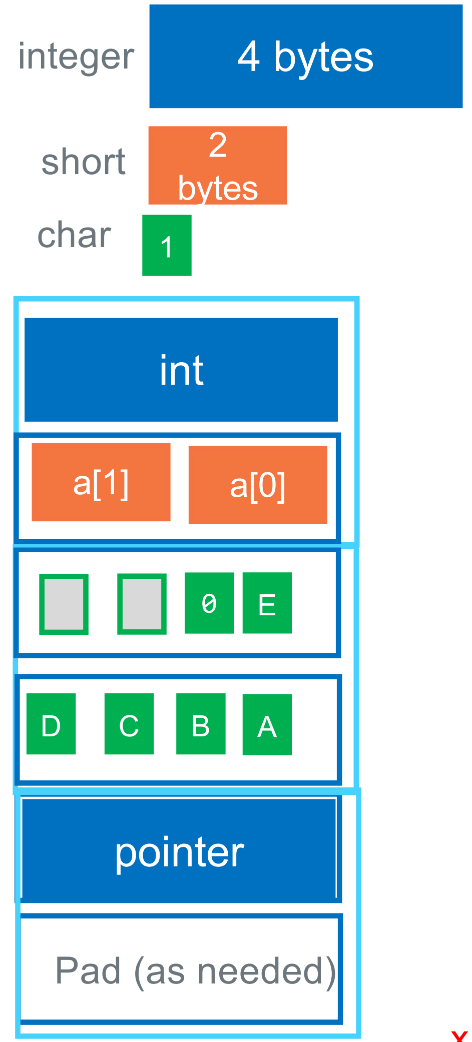
- Return the **sp** (using the **fp**) to the same address it had after the push operation

  sub sp, fp, FP_OFF
- this works no matter how much space was allocated in the prologue

x

Stack Frame Design – Local Variables

- Goal: minimize stack frame size
- 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
- Space padding (0 or 4 bytes) when necessary is added at the high address end of a variables allocated space, based on the variable's alignment and the requirements of variable below it on the stack
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- After all the variables have been allocated, add padding at stack frame bottom (low memory) so the total stack frame size (including all saved registers) is a multiple of 8 when the prologue is finished



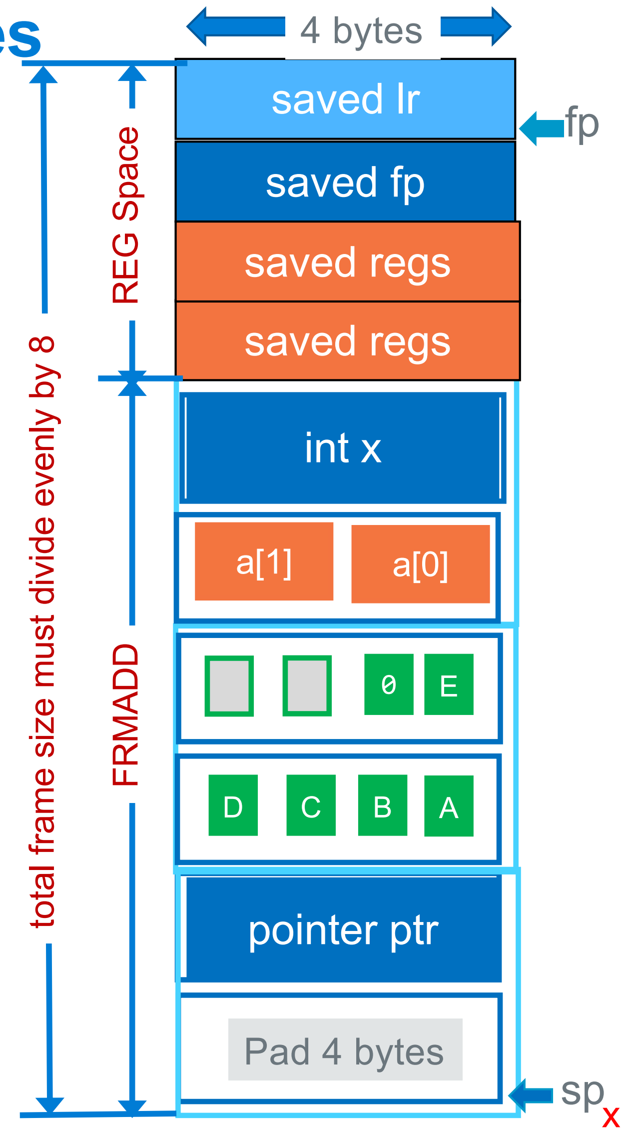
x

Step 1: Stack Frame Design – Local Variables

In this example we are allocating in order of variable definition, **no reordering**

```
int func(void)
{
    int x = 0;
    short st[2];
    char str[] ="ABCDE";
    char *ptr = &array[0];
```

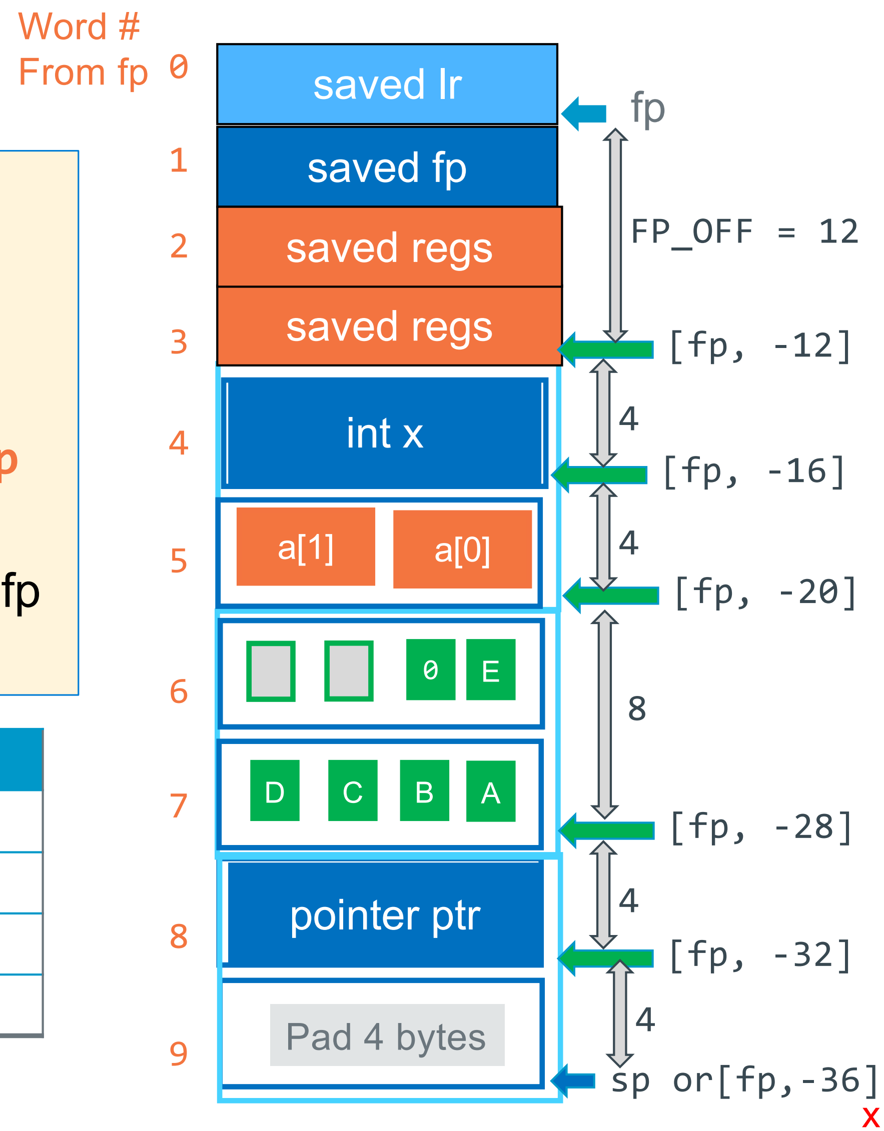
Variable name	Initial Value	Size bytes	Alignment pad to next	Total Size
`int x`	0	4	0	4
`short a[]`	??	2*4	0	4
`char str[]`	"ABCDE"	6	2	8
`char *ptr`	&array[0]	4	0	4
`PAD Added`		4		4
`FRMADD (locals etc)`	-----	-----	-----	24
`Saved Register Space`	-----	4 * 4	---	16
`Total Frame Size`				40



Accessing Stack Variables The Hard Way…..

- Access data stored in the stack
  - use **ldr/str** instructions
- **Use base register fp with offset addressing** (either register offset or immediate offset)
- No matter where in memory the stack is located, **fp** always points at saved **lr**)
- Word offset is a way to visualize the distance from fp for calculating offset values

Variable name	offset from fp	ldr instruction
`int x`	-16	`ldr r0, [fp, -16]`
`short a[]`	-20	`ldrsh r0, [fp, -20]`
`char str[]`	-28	`ldrb r0, [fp, -28]`
`char *ptr`	-32	`ldr r0, [fp, -32]`



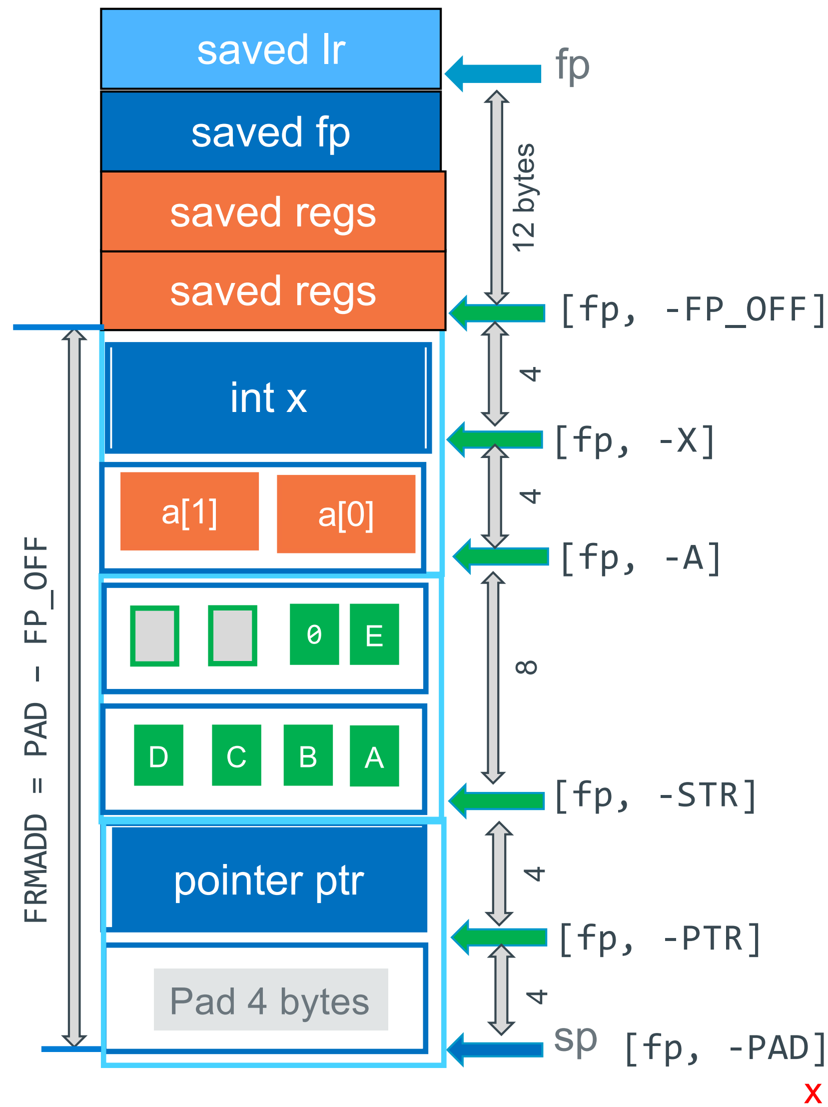
Step 2 Generate offsets from [fp]

- Use the assembler to calculate the offsets from address contained in fp [fp, -offset]

```
.equ FP_OFF, 12
.equ X, 4+FP_OFF // X = 16
.equ A, 4+X      // A = 20
```

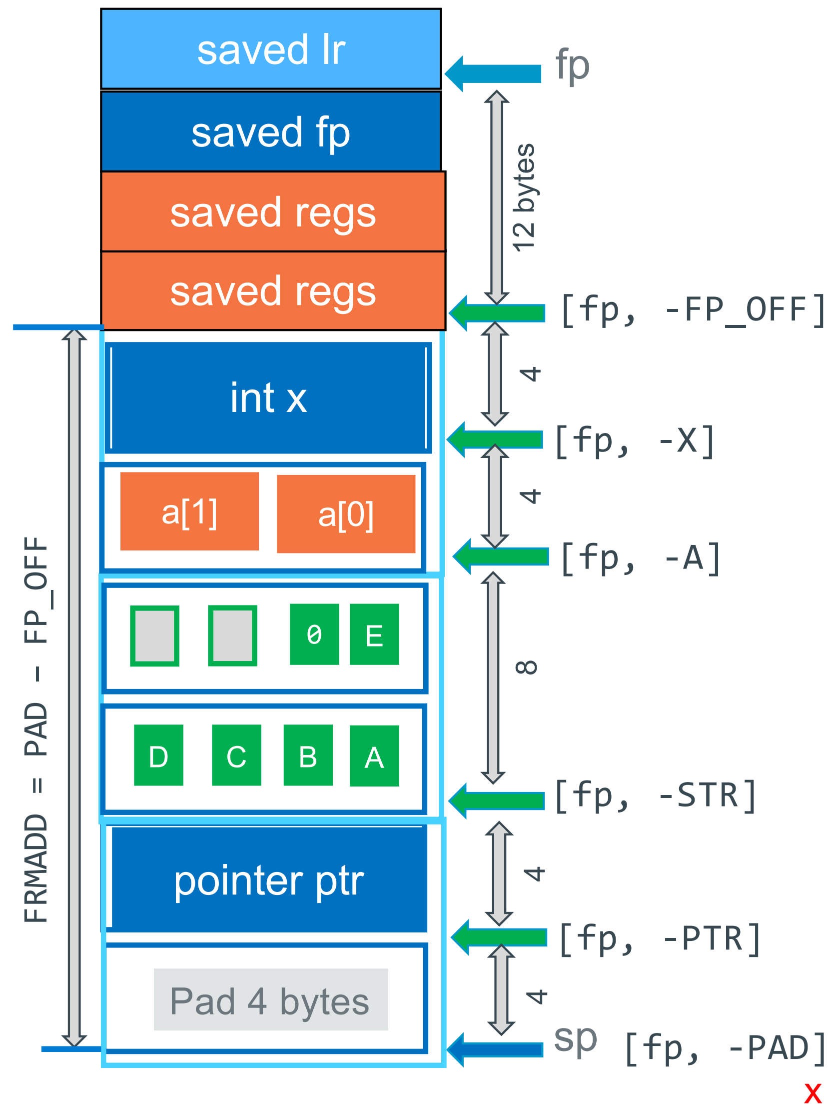
- Assign label names for each local variable
    - Each name is .equ to be the offset from fp

Variable name	Size	Name	expression	Distance from fp
Pushed regs-1	12	FP_OFF		12
int x	4	X	4+ FP_OFF	16
short a[]	4	A	4 + X	20
char str[]	8	STR	4 + A	28
char *ptr	4	PTR	4 + STR	32
PAD Added	4	PAD	4 + PTR	36
FRMADD		FRMADD	PAD-FP_OFF	24



Step 3 Allocate Space in the Prologue

```
    .global func
    .type   func, %function
    .equ      FP_OFF,       12
    .equ      X,            4 + FP_OFF
    .equ      A,            4 + X
    .equ      STR,          8 + A
    .equ      PTR,          4 + STR
    .equ      PAD,          4 + PTR
    .equ      FRMADD        PAD - FP_OFF
func:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD
    sub     sp, sp, r3 // add for locals
    // rest of function code
    // no change to epilogue
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
    .size   func, (. - func)
```



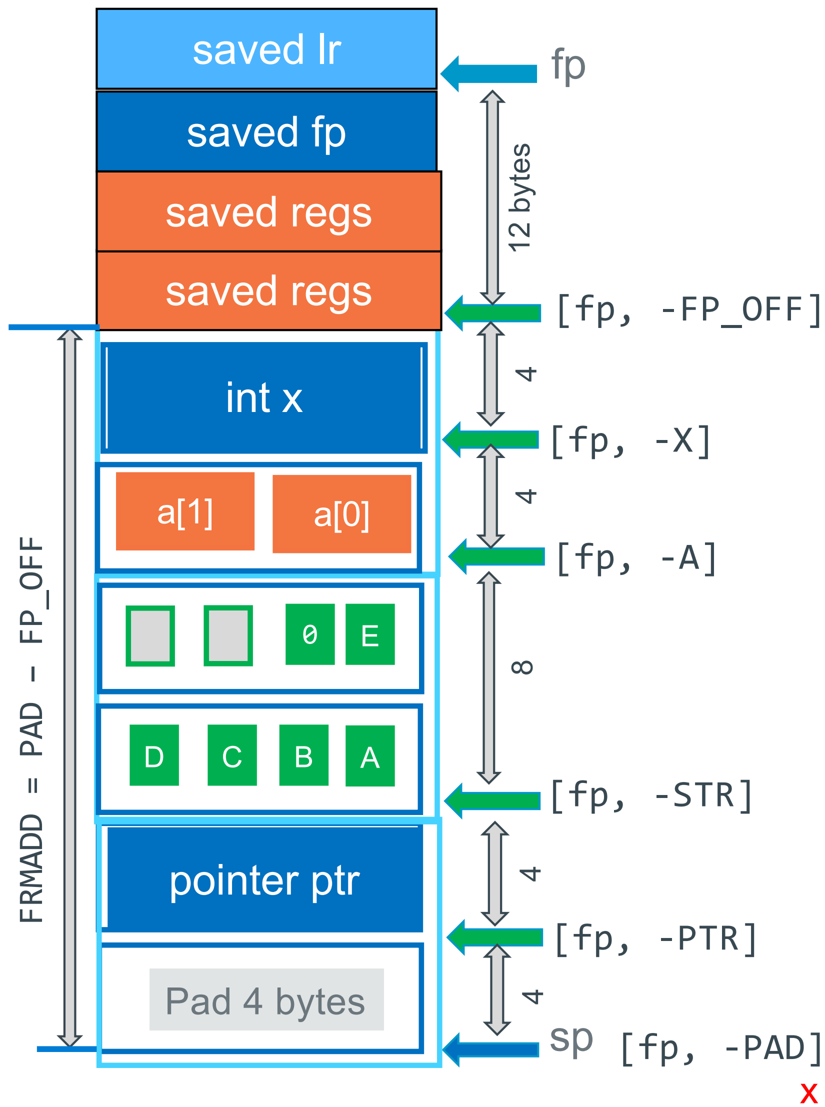
Accessing Stack variables

var	how to get it's address	how to read it's contents
x	`ldr r0, =X` `sub r0, fp, r0`	`ldr r0, =X` `ldr r0, [fp, -r0]`
a[0]	`ldr r0, =A` `sub r0, fp, r0`	`ldr r0, =A` `ldrsh r0, [fp, -r0]`
a[1]	`ldr r0, =A + 2` `sub r0, fp, r0`	`ldr r0, =A + 2` `ldrsh r0, [fp, -r0]`
str[1]	`ldr r0, =STR + 1` `sub r0, fp, r0`	`ldr r0, =STR + 1` `ldrb r0, [fp, -r0]`
ptr	`ldr r0, =PTR` `sub r0, fp, r0`	`ldr r0, =PTR` `ldr r0, [fp, -r0]`

*ptr	`ldr r0, =PTR`
	`sub r0, fp, r0`
	`ldr r0, [r0]`
		`ldr r0, =PTR`
		`ldr r0, [fp, -r0]`
		`ldr r0, [r0]`

var	how to write it's contents
ptr	`ldr r0, =PTR` `str r1, [fp, -r0]`

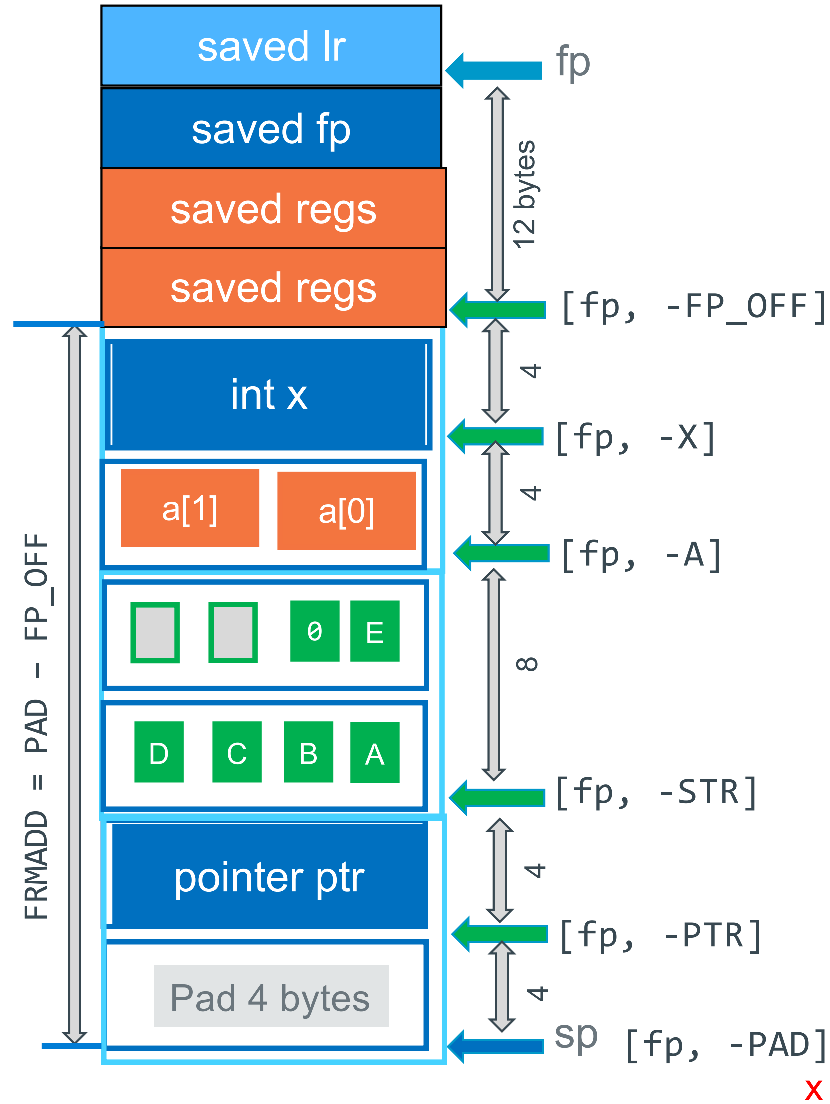
*ptr	`ldr r0, =PTR`
	`ldr r0, [fp, -r0]`
	`str r1, [r0]`



Step 4 Initialize the variables

```
int func(void)
{
    int x = 0;
    short st[2];
    char str[] ="ABCDE";
    char *ptr = &(str[0]);
```

```
    mov    r4, 0
    ldr    r5, =X
    str    r4, [fp, -r5]

    ldr    r5, =STR
    sub    r5, fp, r5      // r5 = addr of STR
    ldr    r4, =PTR
    str    r5, [fp, -r4]  //ptr = &(str[0])
    mov    r4, 'A'
    strb   r4, [r5]
    mov    r4, 'B'
    add    r5, r5, 1
    strb   r4, [r5]
//…
```



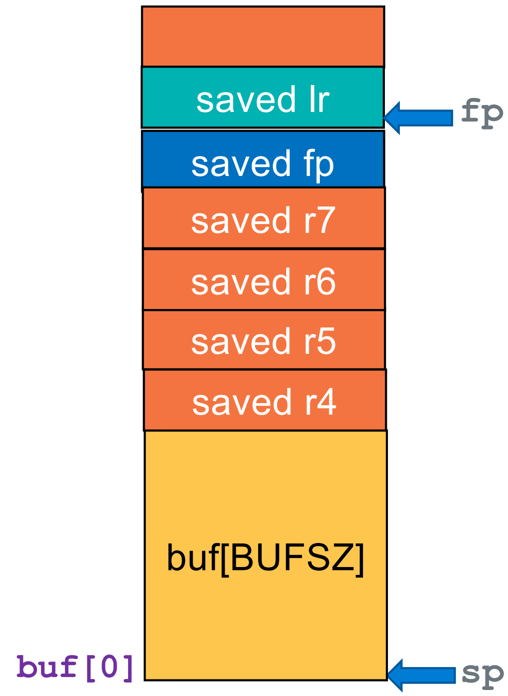
C Stream Functions Array/block read/write

- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
  - Writes an array of *count **elements*** of ***size*** bytes from **stream**
  - *Updates the write file pointer forward by the number of bytes written*
  - returns number of elements written
  - error is short element count or 0
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
  - Reads an array of ***count elements*** of ***size*** bytes from *stream*
  - *Updates the read file pointer forward by the number of bytes read*
  - returns number of elements read, EOF is a return of 0
  - error is short element count or 0
- **I almost always set size to 1 to return bytes read/written**

x

Passing Pointers to Stack Variables

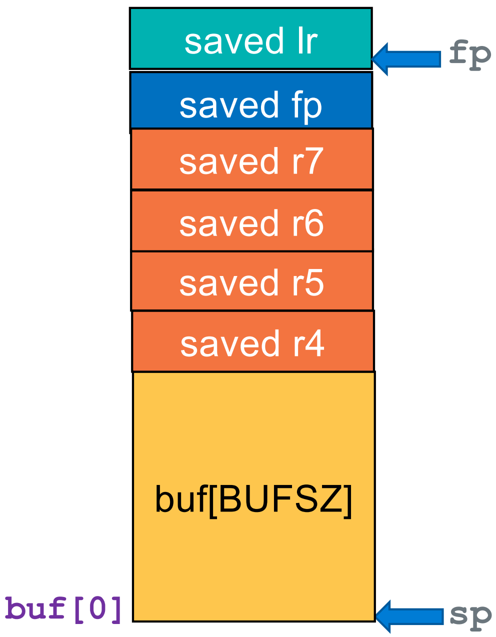
```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BUFSZ 4096

int
main(void) {
    char buf[BUFSZ];
    size_t cnt;   // assign to a register only

    // read from stdin, up to BUFSZ bytes
    // and store them in buf
    // Number of bytes read is in cnt
    while ((cnt = fread(buf, 1, BUFSZ, stdin)) > 0) {
         // write cnt bytes from buf to stdout
        if (fwrite(buf, 1, cnt, stdout) != cnt) {
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```




```
    .text
    .global main
    .type   main, %function    // stack frame below
    .equ    BUFSZ,      4096
    .equ    FP_OFF,     20          // fp offset in main stack frame
    .equ    BUF,        BUFSZ+FP_OFF// buffer
    .equ    PAD,        0+BUF       // Stack frame PAD
    .equ    FRMADD,     PAD-FP_OFF  // space for locals+passed args
```

Reading and Writing bytes using C library routines fread() and fwrite()

```
// save values in preserved registers
ldr     r4, =BUF      // offset in frame
sub     r4, fp, r4    // pointer to buffer
ldr     r5, =stdin    // standard input
ldr     r5, [r5]
ldr     r6, =stdout   // standard output
ldr     r6, [r6]
```




```
// fread(buffer, element_size, number of elements, FILE *)
// fread(r0=buf, r1=1, r2=BUFSZ, r3=stdin)
mov     r0, r4              // buf
mov     r1, 1               // bytes
mov     r2, BUFSZ           // cnt (or ldr r2, =BUFSZ)
mov     r3, r5              // stdin
bl      fread
cmp     r0, 0               // check return value from fread
ble     .Ldone
mov     r7, r0              // save bytes read for compare
```

```
// fwrite(buffer, element_size, number of elements, FILE *)
// fwrite(r0=buf, r1=1, r2=cnt, r3=stdout)
mov     r0, r4              // buf
mov     r1, 1               // bytes
mov     r2, r7              // cnt
mov     r3, r6              // stdout
bl      fwrite
cmp     r0, r7              // check return value from fwrite
beq     .Lloop              // wrote all the bytes, loop
```

Passing Pointers to Stack Variables

```
#define BUFSZ 4096
int main(void) {
    char buf[BUFSZ];
    size_t cnt;   // assign to a register only

    while ((cnt = fread(buf, 1, BUFSZ, stdin)) > 0) {
        if (fwrite(buf, 1, cnt, stdout) != cnt) {
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```

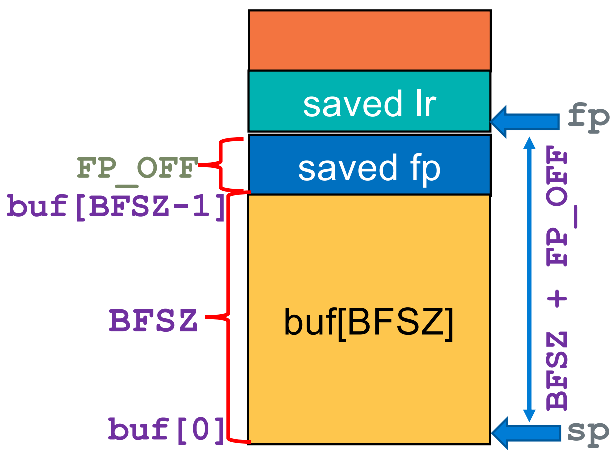
```
    .extern fread
    .extern fwrite
    .extern stdin
    .extern stdout
    .equ EXIT_FAILURE, 1

    .text
    .global main
    .type   main, %function

    .equ    BUFSZ,      4096
    .equ    FP_OFF,     20
    .equ    BUF,        BUFSZ+FP_OFF// buffer
    .equ    PAD,        0+BUF       // Stack frame PAD
    .equ    FRMADD,     PAD-FP_OFF  // locals

// see right -->
.Ldone:
    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx      lr

    .size   main, (. - main)
```

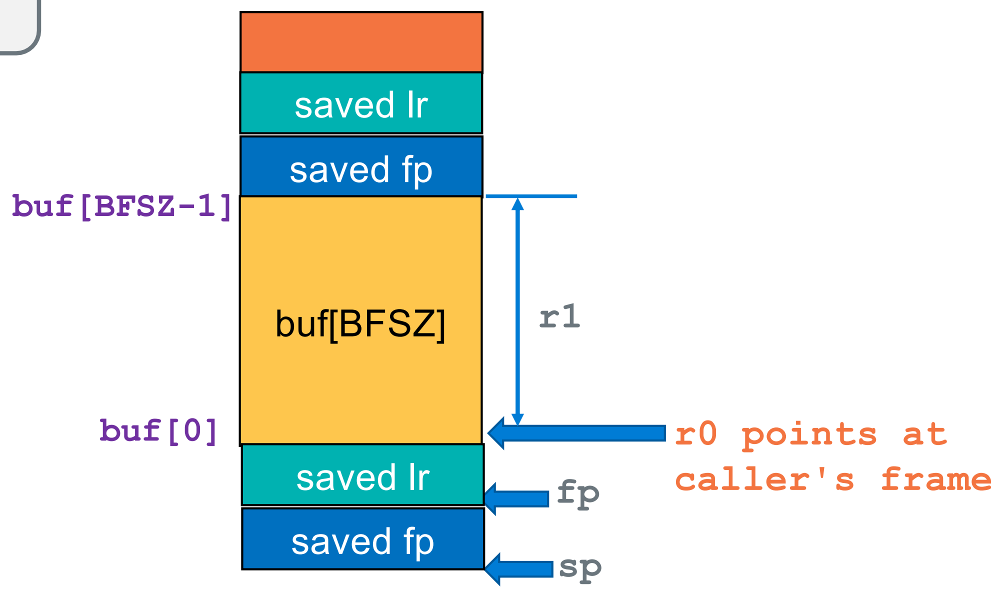
```
main:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF          // set frame pointer
    ldr     r3, =FRMADD             // get frame size
    sub     sp, sp, r3              // allocate space

    // save values in preserved registers
    ldr     r4, =BUF                // offset in frame
    sub     r4, fp, r4              // pointer to buffer
    ldr     r5, =stdin              // standard input
    ldr     r5, [r5]
    ldr     r6, =stdout             // standard output
    ldr     r6, [r6]

.Lloop:
        // fread(r0=buf, r1=1, r2=BUFSZ, r3=stdin)
    mov     r0, r4              // buf
    mov     r1, 1               // bytes
    mov     r2, BUFSZ           // cnt (or ldr r2, =BUFSZ)
    mov     r3, r5              // stdin
    bl      fread
    cmp     r0, 0
    ble     .Ldone
    mov     r7, r0              // save cnt
    // fwrite(r0=buf, r1=1, r2=cnt, r3=stdout)
    mov     r0, r4              // buf
    mov     r1, 1               // bytes
    mov     r2, r7              // cnt
    mov     r3, r6              // stdout
    bl      fwrite
    cmp     r0, r7              // did we write all the bytes?
    beq     .Lloop
    mov     r0, EXIT_FAILURE

.Ldone:
// standard prologue not shown
```

Writing Functions: Receiving a Pointer Parameter - 1

```
#define BFSZ 256
void fillbuf(char *s, int len, char fill);
int main(void)   r0,        r1,        r2
{
  char buf[BFSZ];
  fillbuf(buf, BFSZ, 'A');
  return EXIT_SUCCESS;
}
```

```
void fillbuf(char *s, int len, char fill)
{    r0,         r1,          r2
    char enptr = s + len;
    while (*s < enptr)
       *(s++) = fill;
}
```

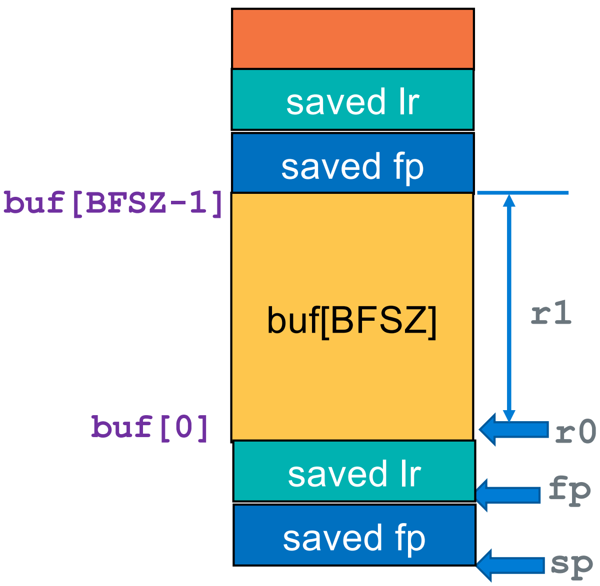




x

Writing Function: Receiving a Pointer Parameter - 2

```
void        r0,        r1,        r2
fillbuf(char *s, int len, char fill)
{
    char enptr = s + len;
    while (s < enptr)
        *(s++) = fill;
}
```



Using r1 for endptr

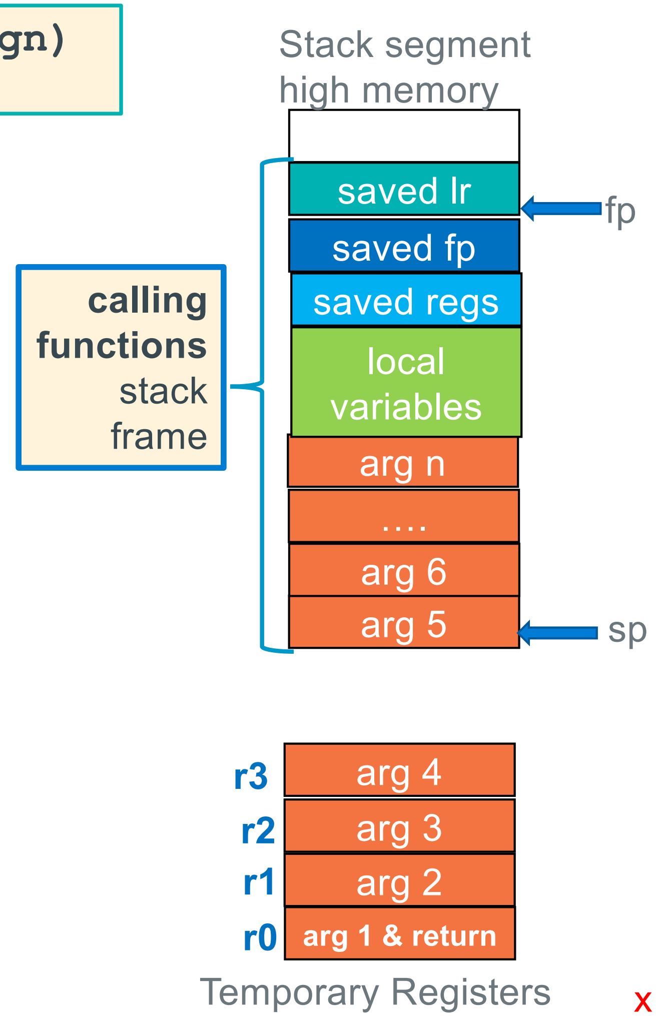
```
fillbuf:
    push    {fp, lr}          // stack frame
    add     fp, sp, FP_OFF    // set fp to base

    add     r1, r1, r0        // copy up to r1 = bufpt + cnt
    cmp     r0, r1            // are there any chars to fill?
    bge     .Ldone            // nope we are done

.Ldowhile:
    strb    r2, [r0]          // store the char in the buffer
    add     r0, 1             // point to next char
    cmp     r0, r1            // have we reached the end?
    blt     .Ldowhile         // if not continue to fill

.Ldone:
    sub     sp, fp, FP_OFF    // restore stack frame top
    pop     {fp, lr}          // restore registers
    bx      lr                // return to caller
```


x

Passing More Than Four Arguments - 1

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

- Each argument is a value that must fit in 32-bits
- **Args > 4 are in the <u>caller's stack frame</u> and arg 5 always starts at fp+4**
  - At the function call (bl) sp points at arg5
  - Additional args are higher up the stack, with one argument "slot" every 4-bytes
- Called functions have the right to change stack args just like they can change the register args!
- Caller must assume all args including ones on the stack are changed by the caller



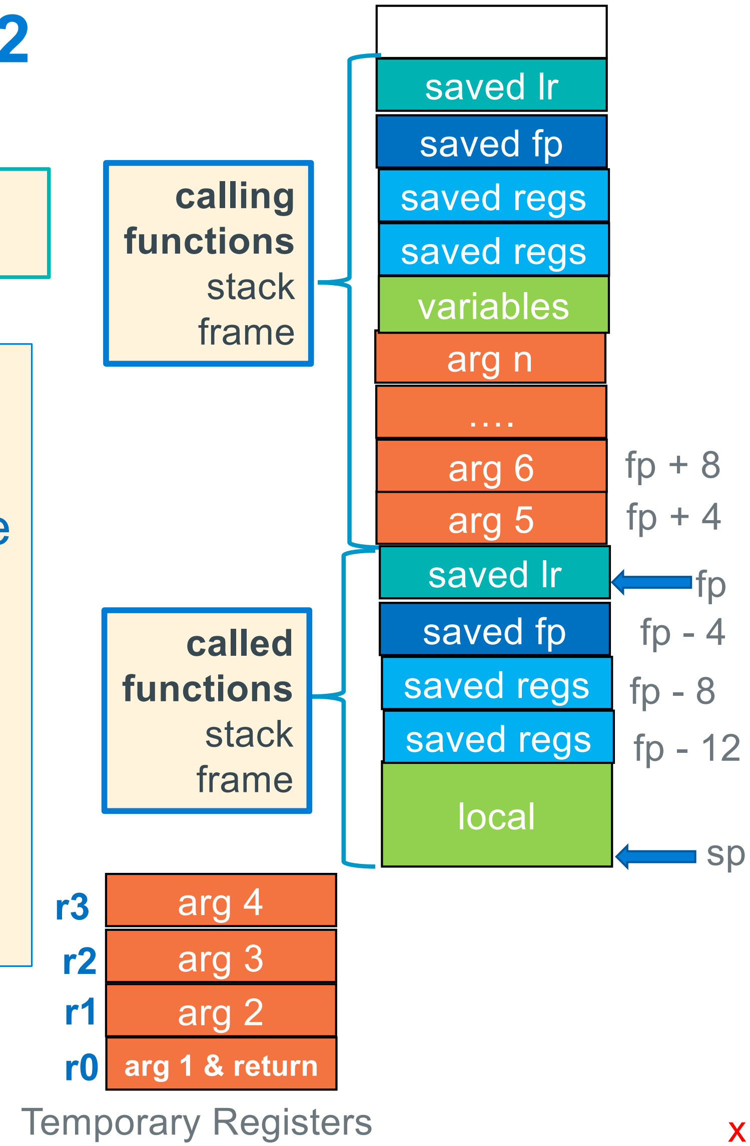
x

Passing More Than Four Arguments - 2

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

- **Addressing rules**
  - Adding to fp to get arg address in caller's frame
  - Subtracting from fp are addresses in called frame
- Why does it work this way?
- This "algorithm" for finding args was designed to enable languages to have variable argument count functions like:

```
printf("conversion list", arg0, … argn);
```

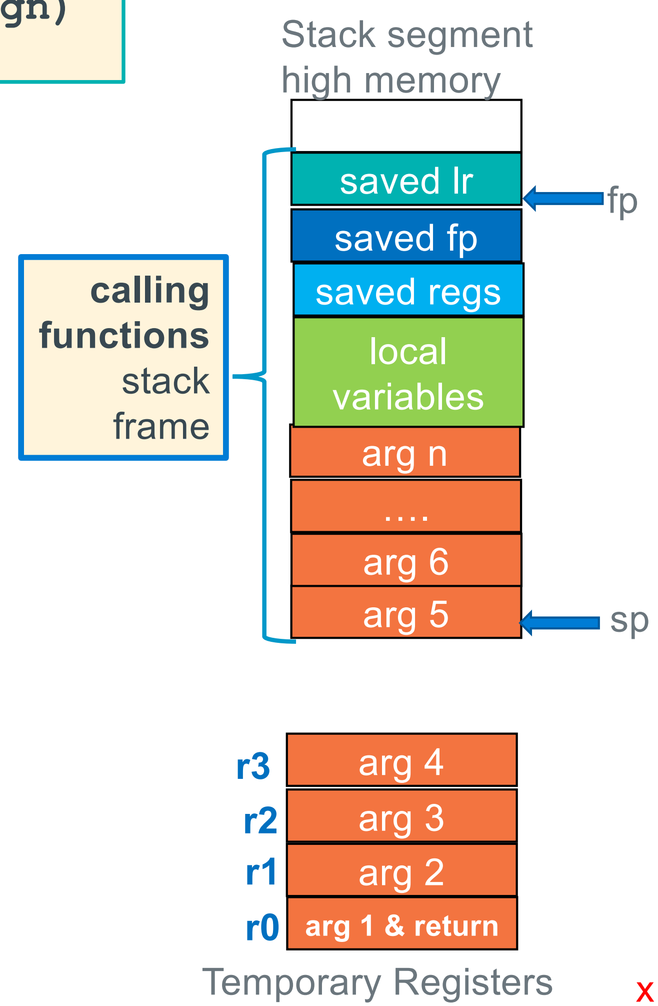


x

Passing More Than Four Arguments – Calling Function

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

- Calling function prior to making the call
  1. Evaluate first four args: place resulting values in r0-r3
  2. Arg 5 and greater are evaluated
  3. Store Arg 5 and greater parameter values on the stack
- **One arg value per slot**! – NO arrays across multiple slots
- chars, shorts and ints are directly stored
- Structs (not always), and arrays are passed via a pointer
- **Pointers** passed as output parameters usually contain an address ***that points at*** the stack, BSS, data, or heap



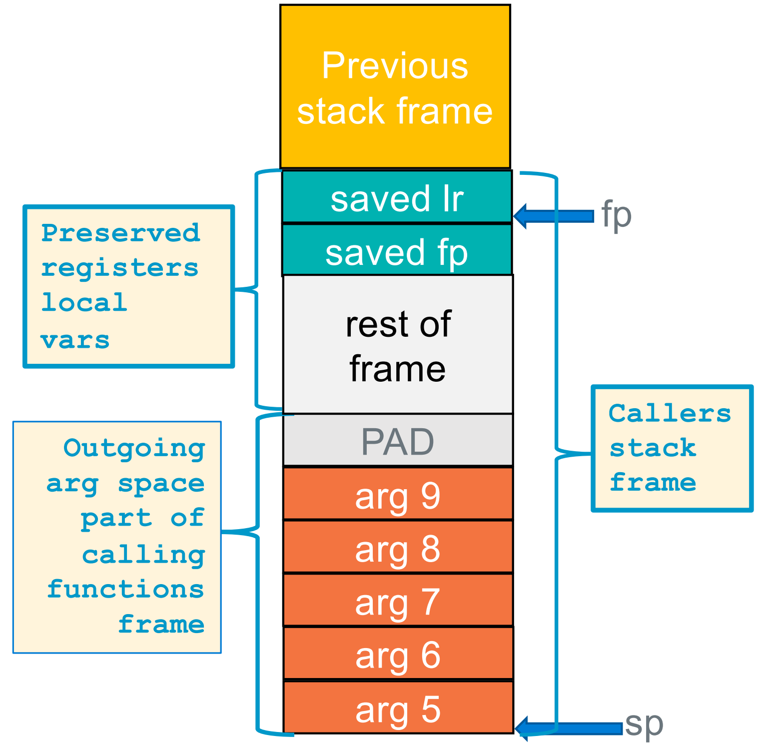
Calling Function: Allocating Stack Parameter Space

At the point of a function call (and obviously at the start of the called function):

1. sp must point at arg5
2. arg5 **must be at an 8-byte boundary**,
   a) **padding** to force arg5 alignment is **placed above** the last **argument the called function is expecting**

Approach: Extend the stack frame to include enough space for stack arguments function with the greatest arg count

1. Examine every function call in the body of a function
2. Find the function call with greatest arg count, Determines space needed for outgoing args
3. Add the space needed to the frame layout



Rules: At point of call
1. **arg5 must be pointed at by sp**
2. **SP must be 8-byte aligned**

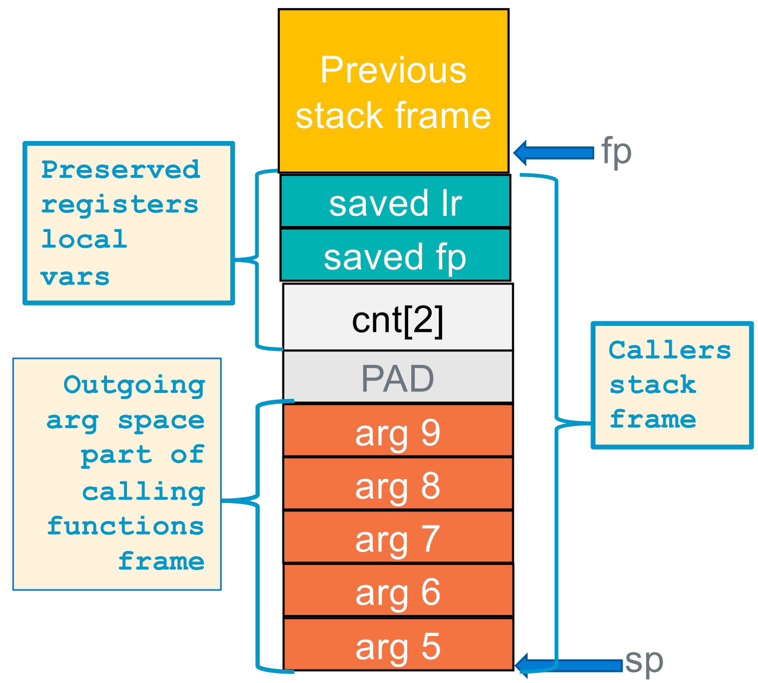
x

Calling Function: Pass ARGS 5 and higher

```
.equ    FP_OFF,4
.equ    CNT,          8 + FP_OFF // int cnt[2];
.equ    PAD,          4 + CNT // added for odd # params
.equ    OARG9,        4 + PAD
.equ    OARG8,        4 + OARG9
.equ    OARG7,        4 + OARG8
.equ    OARG6,        4 + OARG7
.equ    OARG5,        4 + OARG6
.equ    FRMADD        OARG5 – FP_OFF
```

var	write contents
OARG5 = r0	`ldr r0, =OARG5` `str r1, [fp, -r0]`

OARG6 = &cnt	`ldr r1, =CNT`
	`sub r1, fp, r0`
	`ldr r0, =OARG6`
	`str r1, [fp, -r0]`
	`str r1, [r0]`



Rules: At point of call

1. **arg5 must be pointed at by sp**
2. **SP must be 8-byte aligned**

x

Called Function: Retrieving Args From the Stack

- At function start and before the push{} the sp is at an 8-byte boundary
- **Args are in the 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
- This "algorithm" for finding args was designed to enable variable arg count functions like printf("conversion list", arg0, … argn);

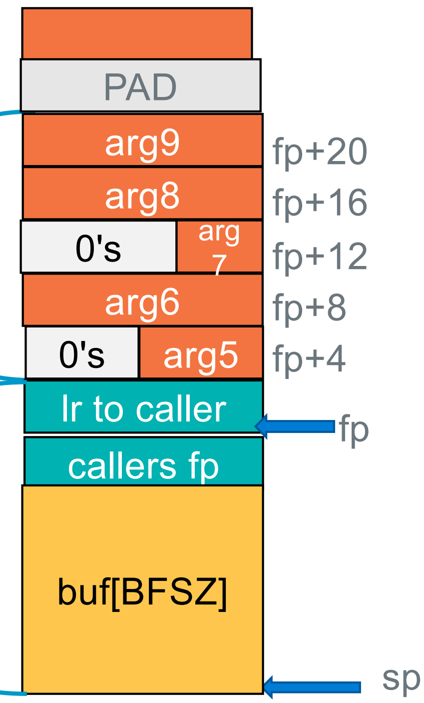
```
int func(int a1, int a2, int a3, int a4,
         short a5, int a6, char a7, int a8, int a9)
```

Constant	Offset	arm ldr /str statement
ARGN	(N-4)*4	ldr r0, [fp, ARGN]
ARG9	20	ldr r0, [fp, ARG9]
ARG8	16	ldr r0, [fp, ARG8]
ARG7	12	ldrb r0, [fp, ARG7]
ARG6	8	ldr r0, [fp, ARG6]
ARG5	4	ldrh r0, [fp, ARG5]

```
.equ ARG9,  20
.equ ARG8,  16
.equ ARG7,  12
.equ ARG6,   8
.equ ARG5,   4
```

Callers Stack frame
no defined limit to number of args, keep going up stack 4 bytes at a time

Current Stack Frame



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

x

Determining 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

Passing More than Four Args – Six Arg Example

- Problem: Write and call a function that receives six integers and returns the sum
- First 4 parameters are in register r0 - r3 and the remaining argument are on the stack
- For this example, we will put all the locals on the stack

```
int main(void)
{
    int cnt = sixsum(1, 2, 3, 4, 5, 6);

    printf("the sum is %d\n", cnt);
    return EXIT_SUCCESS;
}
```

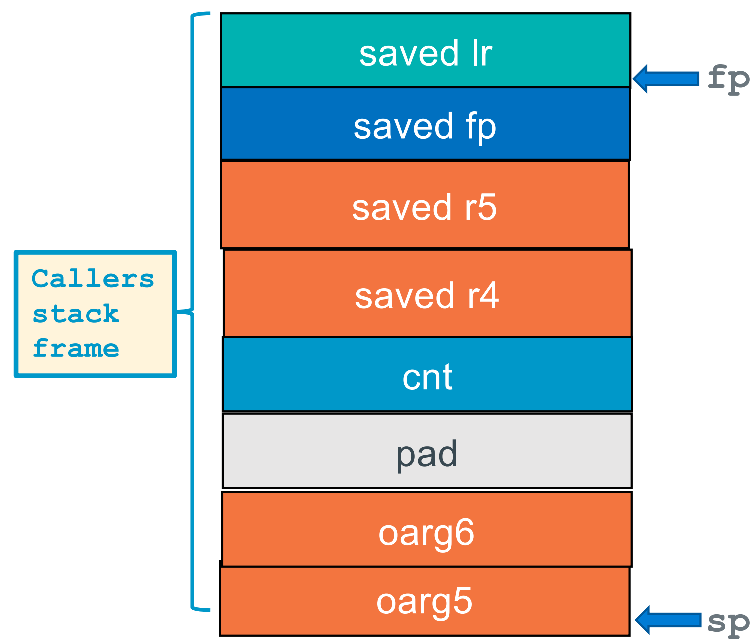
```
int
sixsum(int a1, int a2, int a3, int a4, int a5, int a6)
{
    return a1 + a2 + a3 + a4 + a5 + a6;
}
```


x

Calling Function > 4 Args - 1

```
int cnt = sixsum(1, 2, 3, 4, 5, 6);
```

```
        .equ   FP_OFF,      12  // local base
           // NAME,         SIZE + prev_name
        .equ   CNT,         4 + FP_OFF
        .equ   PAD,         4 + CNT
        .equ   OARG6,       4 + PAD
        .equ   OARG5,       4 + OARG6
        .equ   FRAMESZ      OARG5 - FP_OFF
```

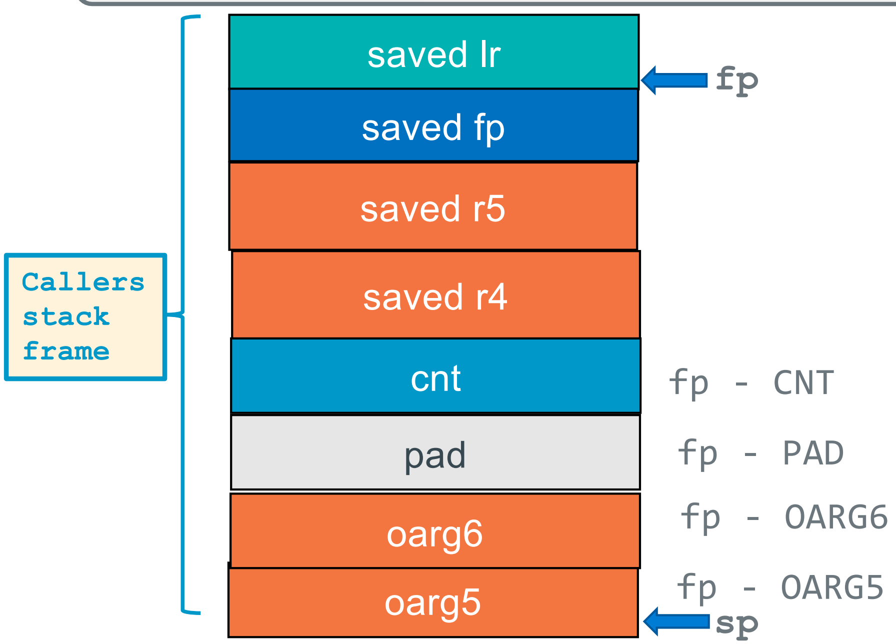


x

Calling Function > 4 Args - 2

```
int cnt = sixsum(1, 2, 3, 4, 5, 6);
```

```
.equu    FP_OFF, 12
.equ     CNT,              4 + FP_OFF
.equ     PAD,              4 + CNT
.equ     OARG6,            4 + PAD
.equ     OARG5,            4 + OARG6
.equ     FRMADD  OARG5 – FP_OFF
```




```
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD
    sub     sp, sp, r3

    mov     r0, 6
    ldr     r5, =OARG6
    str     r0, [fp, -r5]   // arg6
    mov     r0, 5
    ldr     r5, =OARG5
    str     r0, [fp, -r5]   // arg5
    mov     r3, 4           // arg4
    mov     r2, 3           // arg3
    mov     r1, 2           // arg2
    mov     r0, 1           // arg1
    bl      sixsum

    ldr     r5, =CNT
    str     r0, [fp, -r5]  // update cnt on stack
    mov     r1, r0
    ldr     r0, =.Lpfstr
    bl      printf

    mov     r0, EXIT_SUCCESS
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
```

```
        .section .rodata
.Lpfstr: .string "the sum is %d\n"
```

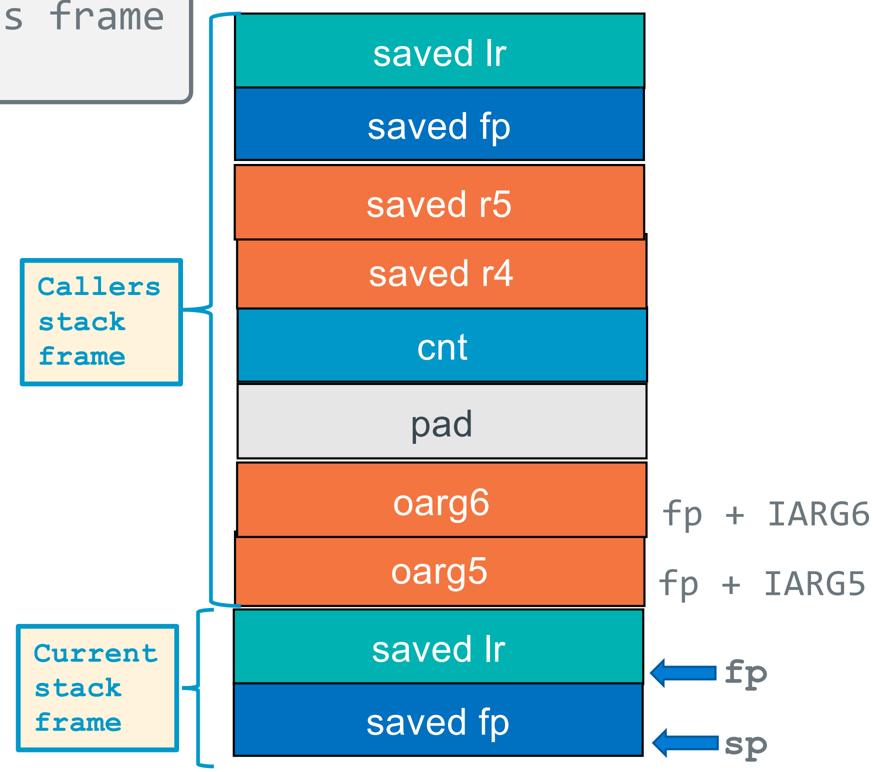
X

Called Function > 4 Args

```
int sixsum(int a1, int a2, int a3, int a4, int a5, int a6)
    return a1 + a2 + a3 + a4 + a5 + a6;
```

```
.equ   IARG6,        8 // offset into caller's frame
.equ   IARG5,        4 // offset into caller's frame
.equ   FP_OFF,       4 // local base
```

```
sixsum:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        add     r0, r0, r1
        add     r0, r0, r2
        add     r0, r0, r3
        ldr     r1, [fp, IARG5]
        add     r0, r0, r1
        ldr     r1, [fp, IARG6]
        add     r0, r0, r1
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
```



x

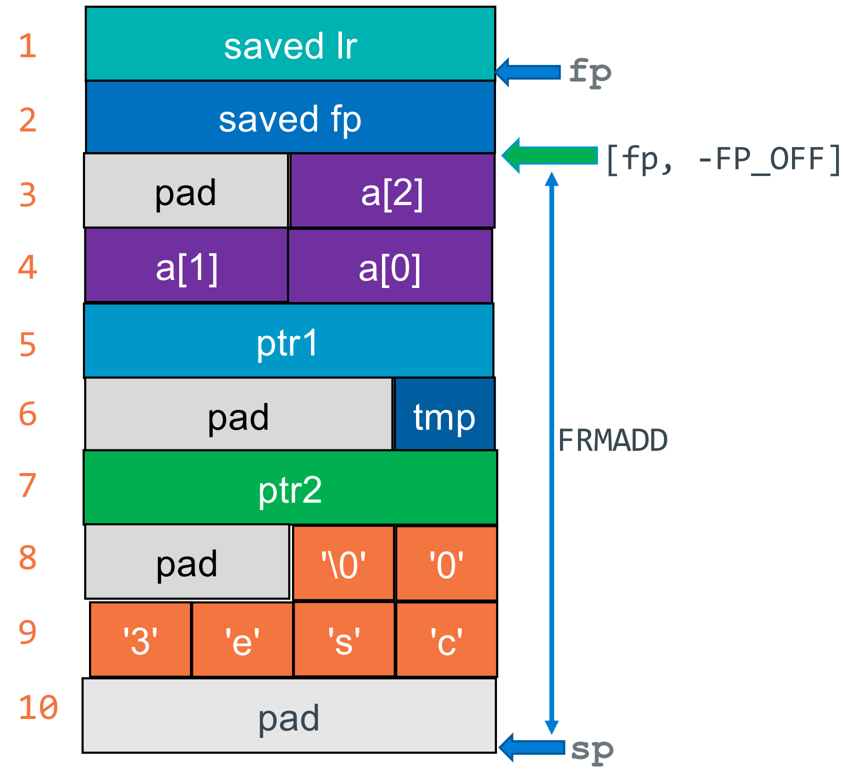
Extra Slides

Local Variables: Stack Frame Design Practice

Example shows allocation **without reordering** variables to optimize space

```
    short a[3];
    short *ptr1;
    char tmp;
    char *ptr2;
    char nm[] = "cse30";
```

```
.equ   FP_OFF,       4  // Local base
// NAME,            SIZE + prev_name
.equ   A,            8 + FP_OFF
.equ   PTR1,         4 + A
.equ   TMP,          4 + PTR1
.equ   PTR2,         4 + TMP
.equ   NM,           8 + PTR2
.equ   PAD,          4 + NM
.equ   FRMADD        PAD – FP_OFF // for locals
```



When writing real code, you do not have to put all locals on the stack

- Place locals in registers if they fit, are accessed often, **and**
- You do not need their address (they are not an output variable in a function call)

x

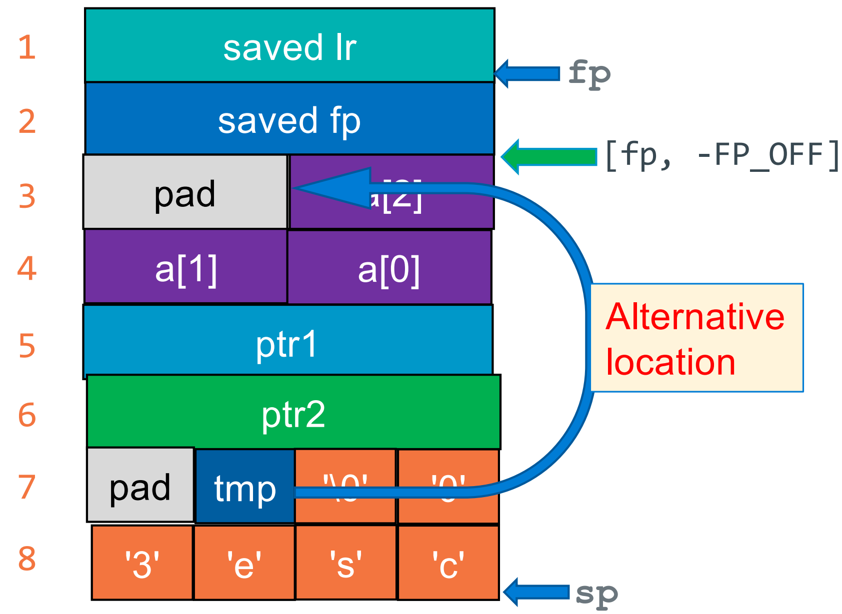
Local Variables: Stack Frame Design Reordering

Example shows allocation **with reordering** variables to optimize space

```
short a[3];
short *ptr1;
char *ptr2;
char tmp;
char nm[] = "cse30";
```

```
.equ   FP_OFF,        4  // Local base
// NAME,             SIZE + prev_name
.equ   A,             8 + FP_OFF
.equ   PTR1,          4 + A
.equ   PTR2,          4 + PTR1
.equ   TMP,           2 + PTR2
.equ   NM,            6 + TMP
.equ   PAD,           0 + NM // not needed
.equ   FRMADD         PAD - FP_OFF
```

size change



When writing real code, you do not have to put all locals on the stack

- Place locals in registers if they fit, are accessed often, **and**
- You do not need their address (they are not an output variable in a function call)

x

ARM Assembly Source File: Header and Footer

File Header
At the top of every ARM source file

```
        .arch   armv6         // armv6 architecture
        .arm                  // arm 32-bit instruction set
        .fpu    vfp           // floating point co-processor
        .syntax unified       // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

File Footer
At the bottom of every ARM source file

```
.section .note.GNU-stack,"",%progbits // set stack/data non-exec
.end
       // everything past the .end is ignored!
       // Debugging notes etc
```

`.syntax unified`

- use the standard ARM assembly language syntax called ***Unified Assembler Language* (*UAL)***

`.section .note.GNU-stack,"",%progbits`

- tells the linker to **make the stack and all data segments not-executable** (no instructions in those sections) – security measure

`.end`

- at the end of the source file, everything written after the .end is ignored

X

Function Header and Footer Assembler Directives

function entry point
address of the first instruction in the function
Must not be a local label (does not start with .L)

Function Header / Function Footer

```
            .text
            .global myfunc              // make myfunc global for linking
            .type   myfunc, %function   // define myfunc to be a function
            .equ    FP_OFF,  4          // fp offset in main stack frame
myfunc:
             // function prologue, stack frame setup
             // your code
             // function epilogue, stack frame teardown
            .size myfunc, (. - myfunc)
```

`.global function_name`

- Exports the function name to other files. **Required for main function,** optional for others

`.type name, %function`

- The `.type` directive sets the **type of a symbol/label name**
- `%function` specifies that **name** is a function (name is the address of the first instruction)

`equ FP_OFF, 4`

- Used for basic stack frame setup; the number 4 will change – later slides

`.size name, bytes`

- The `.size` directive is used to set the size associated with a symbol
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- `bytes` **is best calculated as an expression: (period is the current address in a memory segment)**
  In CSE30 required use: `.size name, (. - name)`

X

Reference For PA8/9: C Stream Functions Opening Files

```
FILE *fopen(char filename[], const char mode[]);
```

- Opens a stream to the specified file in specified file access mode
  - returns NULL on failure – always check the return value; make sure the open succeeded!
- Mode is a string that describes the actions that can be performed on the stream:

"r" Open for reading.

The stream is positioned at the beginning of the file. Fail if the file does not exist.

"w" Open for writing.

The stream is positioned at the beginning of the file. Create the file if it does not exist.

"a" Open for writing.

The stream is positioned at the end of the file. Create the file if it does not exist.

Subsequent writes to the file will always be at current end of file.

- An optional "+" following "r", "w", or "a" opens the file for both reading and writing

x

Reference: C Stream Functions Closing Files and Usage

```
int fclose(FILE *stream);
```

- Closes the specified stream, forcing output to complete (eventually)
  - returns EOF on failure (often ignored as no easy recovery other than a message)
- Usage template for **`fopen()`** and **`fclose()`**
  1. Open a file with **`fopen()`** **always** checking the return value
  2. do i/o – keep calling stdio io routines
  3. close the file with **`fclose()`** when done with that I/O stream

x

C Stream Functions Array/block read/write

- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
  - Writes an array of *count **elements*** of ***size*** bytes from **stream**
  - *Updates the write file pointer forward by the number of bytes written*
  - returns number of elements written
  - error is short element count or 0
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
  - Reads an array of ***count elements*** of ***size*** bytes from *stream*
  - *Updates the read file pointer forward by the number of bytes read*
  - returns number of elements read, EOF is a return of 0
  - error is short element count or 0
- **I almost always set size to 1 to return bytes read/written**

x

C fread/fwrite Example - 1

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BFSZ      8192 /* size of read */
int main(void)
{
  char fbuf[BFSZ];
  FILE *fin, *fout;
  size_t readlen;
  size_t bytes_copied = 0;
  retval = EXIT_SUCCESS;

  if (argc != 3){
    fprintf(stderr, "%s requires two args\n", argv[0]);
    return EXIT_FAILURE;
  }
  /* Open the input file for read */
  if ((fin = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"fopen for read failed\n");
    return EXIT_FAILURE;
  }
  /*  Open the output file for write */
  if ((fout = fopen(argv[2], "w") == NULL) {
    fprintf(stderr, "fopen for write failed\n");
    fclose(fin);
    return EXIT_FAILURE;
  }
```

To handle bytes moved

```
% ls -ls ZZZ
ls: ZZZ: No such file or directory
% ./a.out cp.c ZZZ
bytes copied: 1122
% ls -ls cp.c ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:51 ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:49 cp.c
```


X

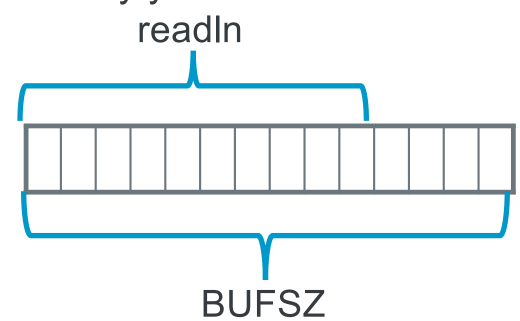
C fread/fwrite Example - 2

```
  /* Read from the file, write to fout */

  while ((readlen = fread(fbuf, 1, BUFSIZ, fin)) > 0) {

    if (fwrite(fbuf, 1, readlen, fout) != readlen) {
        fprintf(stderr, "write failed\n");
         retval =  EXIT_FAILURE;
         break;
    }
    bytes_copied += readlen; //running sum bytes copied
  }

  if (retval == EXIT_FAILURE)
    printf("Failure Copy did not complete only ");
  printf("Bytes copied: %zu\n", bytes_copied);

  fclose(fin);
  fclose(fout);

  return retval;
}
```

By using an element size of 1 with a char buffer, this is byte I/O

Capture the bytes read so you know how many bytes to write

unless file length is an exact multiple of BUFSIZ, the last fread() will always be less than BUFSIZ which is why you write readln



Jargon: the last record is often called the "runt"

x

putchar/getchar
Setting up and Usage

```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;
}
```

r0 r1

```
        .extern getchar
        .extern putchar
        .section .rodata
.Lfstr: .string  "Echo count: %d\n"
        .text
        .equ    EOF,         -1
        .type   main, %function
        .global main
        .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 */
.Ldone:
        mov     r1, r4 // count
        ldr     r0, =.Lfstr
        bl      printf
        mov     r0, EXIT_SUCCESS
        sub     sp, fp, FP_OFF
        pop     {r4, r5, fp, lr}
        bx      lr
        .size main, (. - main)
```

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

```
        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
        ldr     r0, =pfstr
        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;
}
```

File header and footers are not shown

x

printing error messages in assembly

```
.Lmsg0: .string "Read failed\n"
        ldr     r0, =.Lmsg0                     // read failed print error
        bl      errmsg
```

```
        // int errmsg(char *errormsg)
        // writes error messages to stderr
        .type   errmsg, %function               // define to be a function
        .equ    FP_OFF,         4               // fp offset in stack frame
errmsg:
        push    {fp, lr}                        // stack frame register save
        add     fp, sp, FP_OFF                  // set the frame pointer

        mov     r1, r0
        ldr     r0, =stderr
        ldr     r0, [r0]
        bl      fprintf
        mov     r0, EXIT_FAILURE                // Set return value
        sub     sp, fp, FP_OFF                  // restore stack frame top
        pop     {fp, lr}                        // remove frame and restore
        bx      lr                              // return to caller
        // function footer
        .size   errmsg, (. - errmsg)            // set size for function
```


x