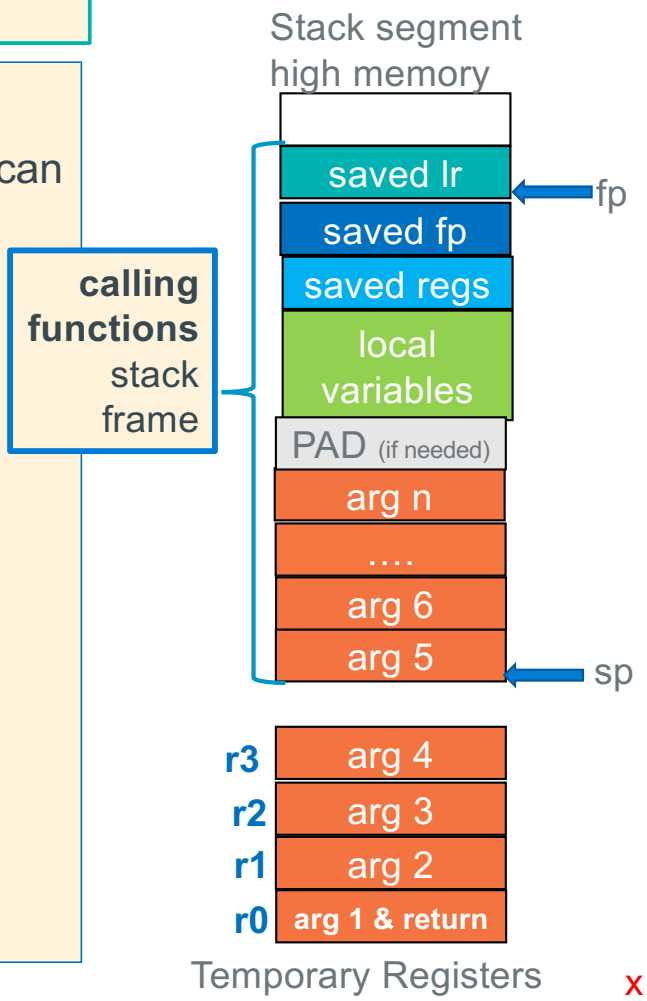

Version 1.00
UCSD CSE 30
Computer Organization and Systems Programming
PA9 – Part 1 writing rdbuf()
Keith Muller

Passing More Than Four Arguments – At the point of Call

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

- **Args > 4 are in the caller's stack frame at SP (argv5), an up**
- 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
- Calling function prior to making the call
  1. Evaluate first four args: place resulting values in r0-r3
  2. 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



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);

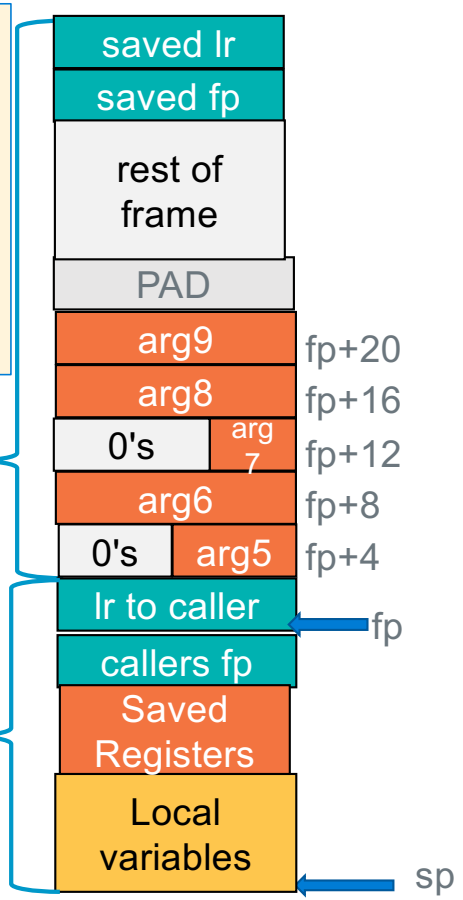
```
int func(int a1, int a2, int a3, int a4,
         short a5, int a6, char a7, int a8, int a9)
```

Constant	Offset	arm ldr /str statement
ARGN	(N-4)*4	ldr r4, [fp, ARGN]
ARG9	20	ldr r4, [fp, ARG9]
ARG8	16	ldr r4, [fp, ARG8]
ARG7	12	ldrb r4, [fp, ARG7]
ARG6	8	ldr r4, [fp, ARG6]
ARG5	4	ldrh r4, [fp, ARG5]

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

Current Stack Frame

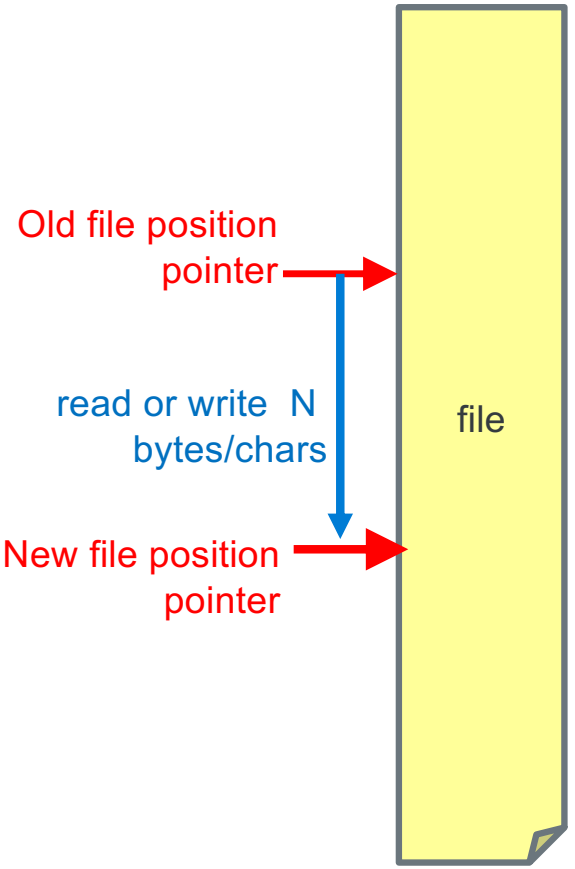
```
.equ ARG9,  20
.equ ARG8,  16
.equ ARG7,  12
.equ ARG6,   8
.equ ARG5,   4
```



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

C Stream Functions Array/block read/write

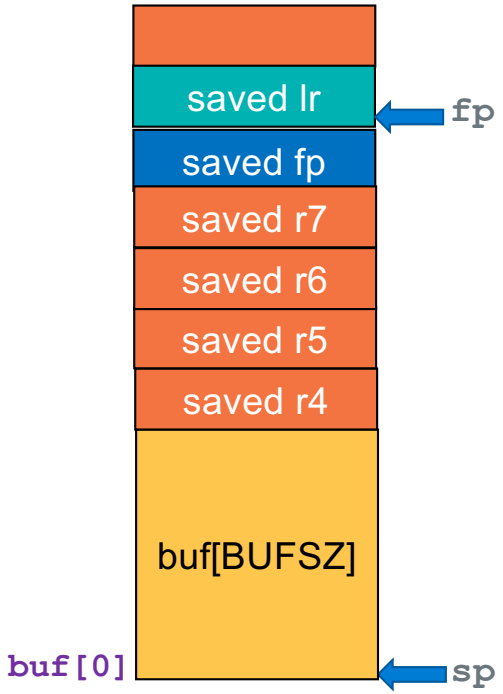
- Read/write ops *advance* the ***file position pointer*** from TOF towards EOF on each I/O
  - Moves towards EOF by number of bytes read/written
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
  - Writes an array *ptr of *count* ***elements*** of ***size*** bytes from **stream**
  - *Updates the write file pointer forward by the number of bytes written*
  - returns number of elements written
    - Treat return != count as an error
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
  - Reads an array *ptr 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,
    - Treat a return of 0 as being in EOF state
- **Set element size to 1 to return bytes read/written**
- EOF is **NOT a character in the file**, but a condition on the stream
- `int feof(FILE *stream)`
  - Returns non-zero at end-of-file for stream
- `int ferror(FILE *stream)`
  - Returns non-zero if error for stream



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;
}
```

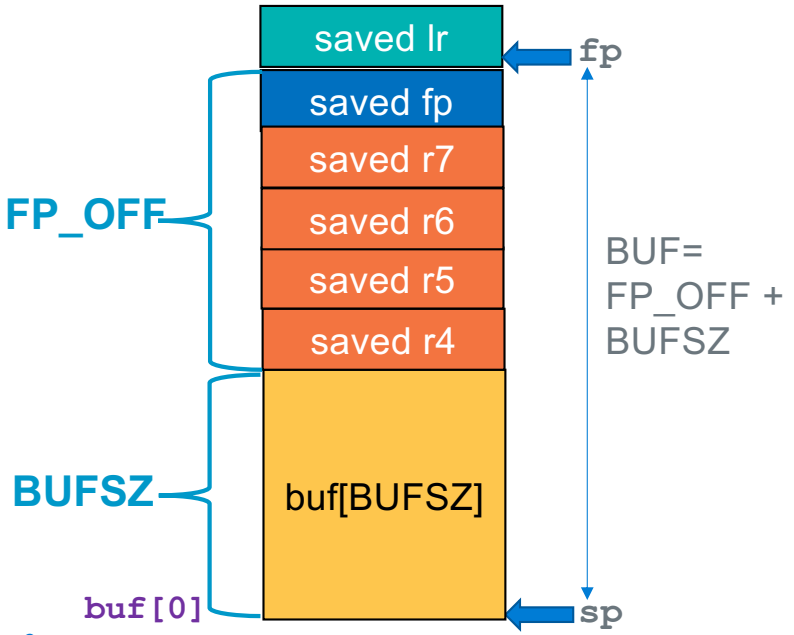



```
    .text
    .global main
    .type   main, %function    // stack frame below
    .equ    BUFSZ,      4096
    .equ    FP_OFF,     20          // fp offset in main stack frame
    .equ    BUF,        BUFSZ+FP_OFF// buffer
    .equ    PAD,        0+BUF       // Stack frame PAD
    .equ    FRMADD,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

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

```
    .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
```

```
    // 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]
```

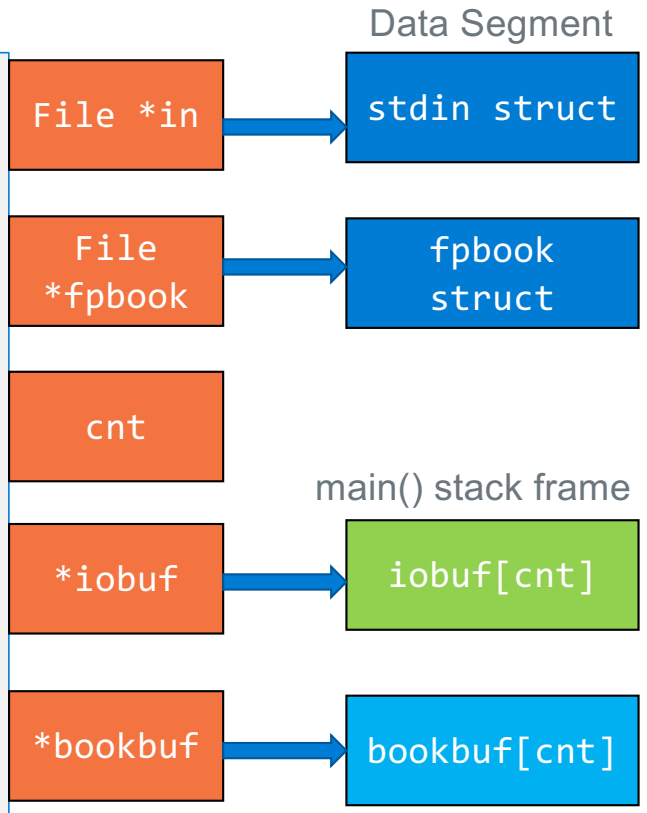



```
    // fread(buffer, element_size, number of elements, FILE *)
    // fread(r0=buf, r1=1, r2=BUFSZ, r3=stdin)
    mov     r0, r4              // buf
    mov     r1, 1               // bytes
    mov     r2, BUFSZ           // cnt (or ldr r2, =BUFSZ)
    mov     r3, r5              // stdin
    bl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

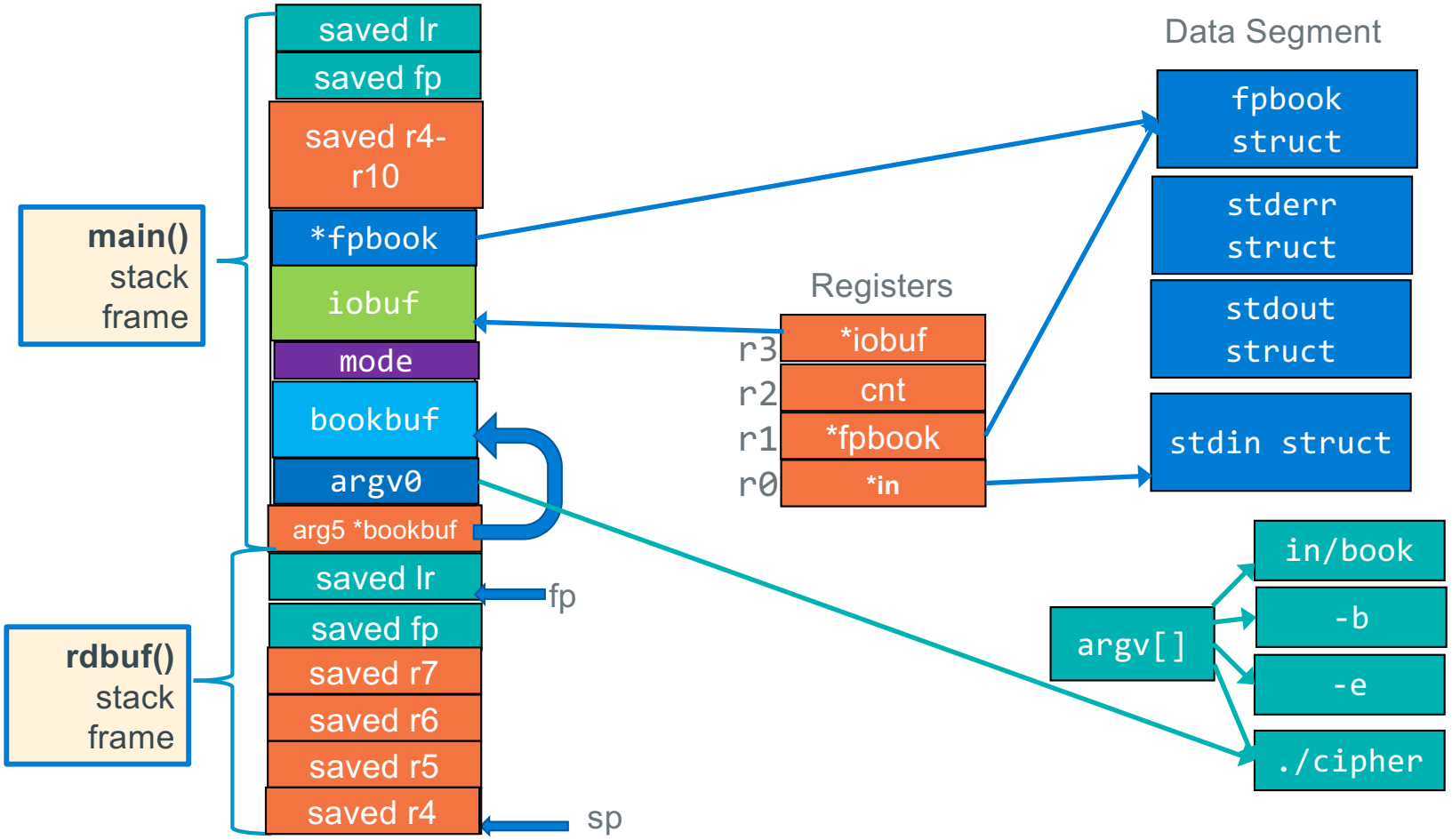
```
    // fwrite(buffer, element_size, number of elements, FILE *)
    // fwrite(r0=buf, r1=1, r2=cnt, r3=stdout)
    mov     r0, r4              // buf
    mov     r1, 1               // bytes
    mov     r2, r7              // cnt
    mov     r3, r6              // stdout
    bl      fwrite
    cmp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

Crdbuf.c

```
int rdbuf(FILE *in, FILE *fpbook, int cnt, char *iobuf, char *bookbuf)
{
    int bytes; /* use in a register */
    /*
     * read the file
     * cnt should be really a size_t but on ARM32 it is an int
     */
    if (feof(in))
        return 0;
    if (ferror(in))
        return EXIT_FAIL;

    if ((bytes = (int)fread(iobuf, 1, cnt, in)) <= 0)
        return 0;
    /*
     * now read the same number of chars from the bookfile
     * as was read from the input file
     */
    if ((int)fread(bookbuf, 1, bytes, fpbook) != bytes)
        return EXIT_FAIL;
    /*
     * return the number of chars read
     */
    return bytes;
}
```



rdbuf() stack frame



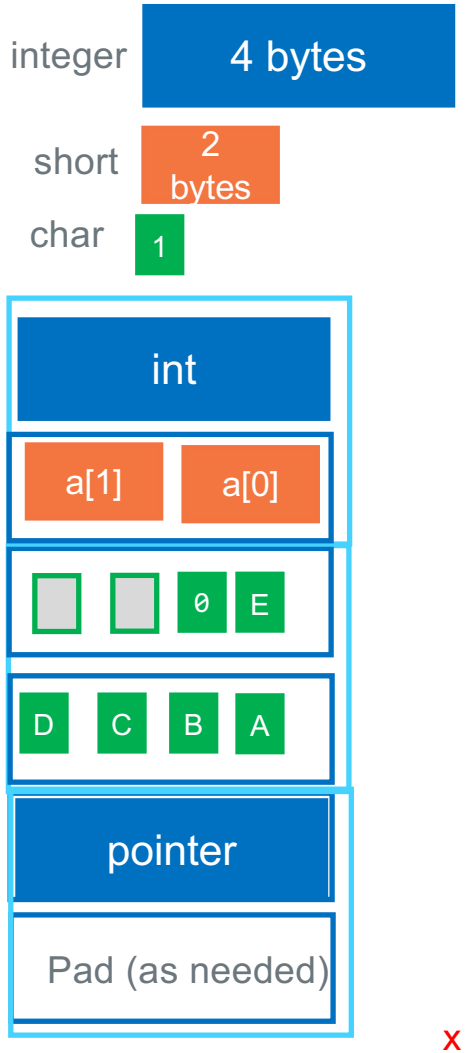

Version 1.00
UCSD CSE 30
Computer Organization and Systems Programming
PA9 – Part 3 main() stack frame
Keith Muller

Cmain.c extract….

```
int main(int argc, char **argv)
{
    /*
     * do not change the definition order of these local variables
     */
    FILE *fpbook;
    char iobuf[BUFSZ];
    int mode;
    char bookbuf[BUFSZ];
    char *argv0;

    int cnt; /* do not put on stack, use a register for this */

// rest of code not shown

    /*
     * read the input and book file until EOF on the input file
     * Either encrypt or decrypt
     * then write it out.
     */
    while ((cnt = rdbuf(stdin, fpbook, BUFSZ, iobuf, bookbuf)) > 0) {
```

Stack Frame Design – Local Variables

- 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

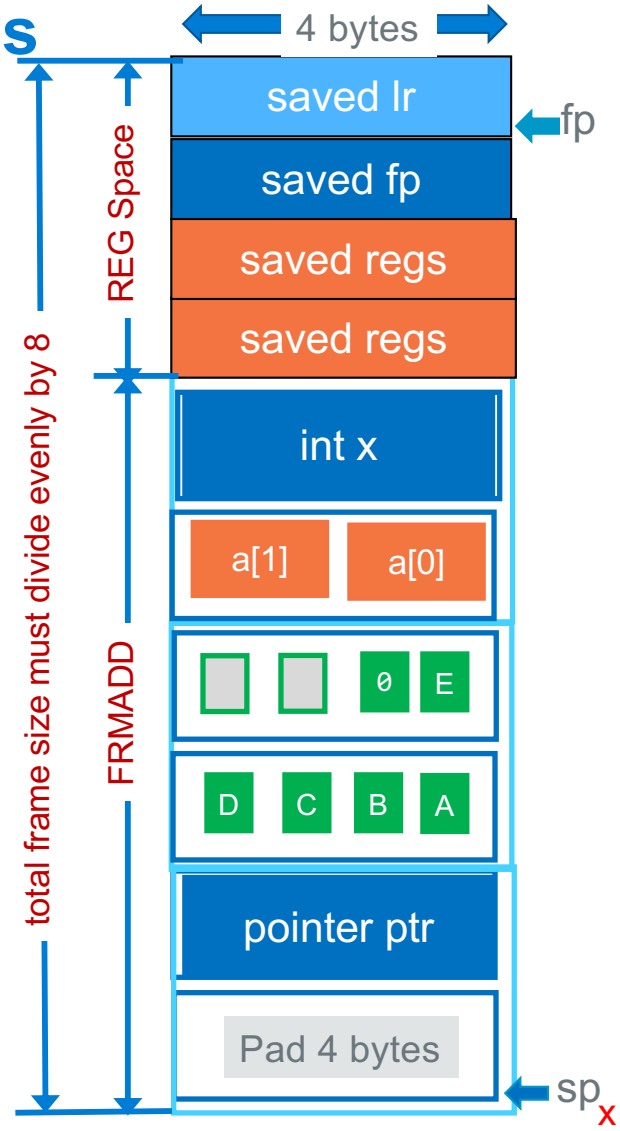


Step 1: Stack Frame Design – Local Variables

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];
```

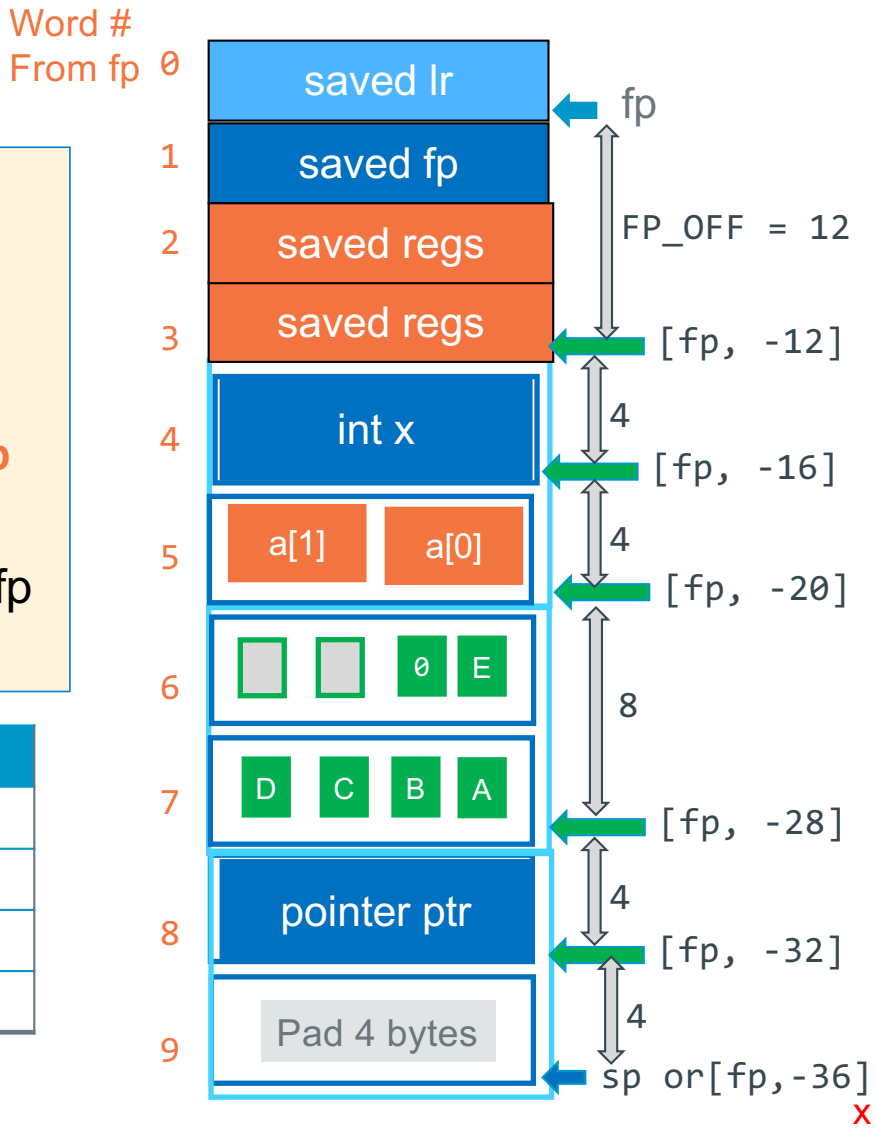
Variable name	Initial Value	Size bytes	Alignment pad to next	Total Size
int x	0	4	0	4
short a[]	??	2*2	0	4
char str[]	"ABCDE"	6	2	8
char *ptr	&array[0]	4	0	4
PAD Added		4		4
FRMADD (locals etc)	-----	-----	-----	24
Saved Register Space	-----	4 * 4	---	16
Total Frame Size				40



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]`



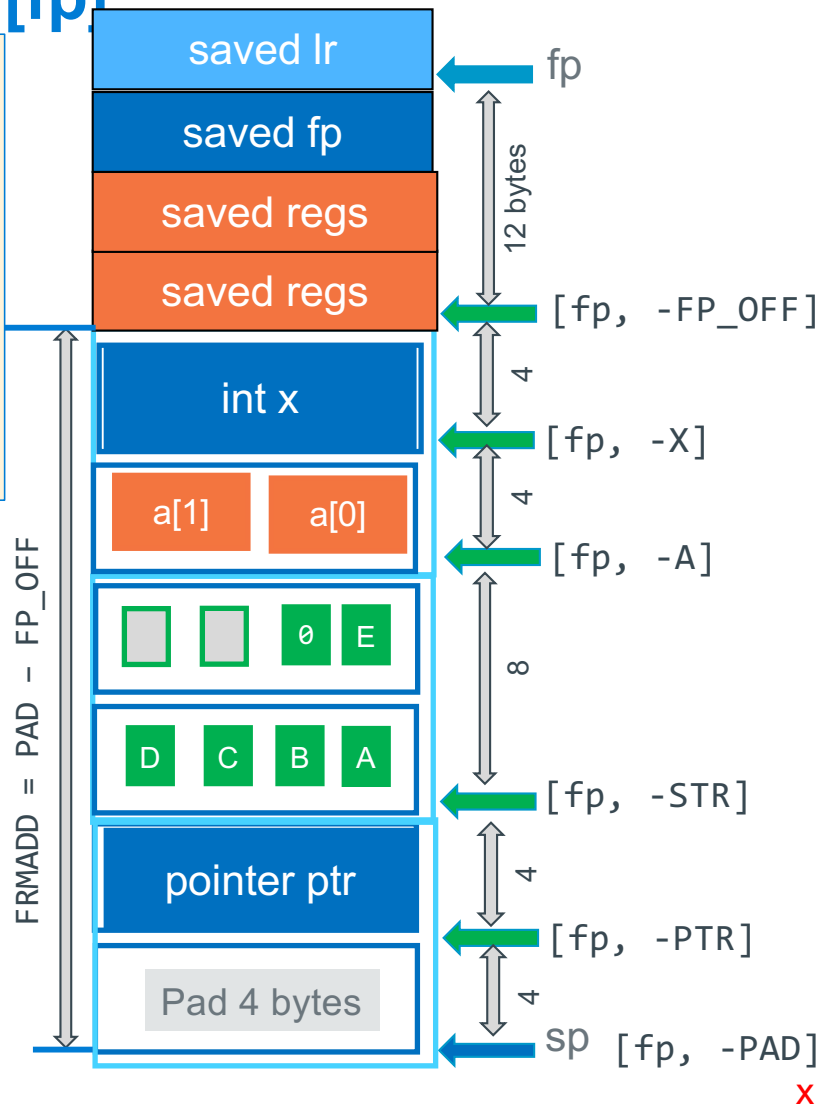
Step 2 Generate Distance offsets from [fp]

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

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

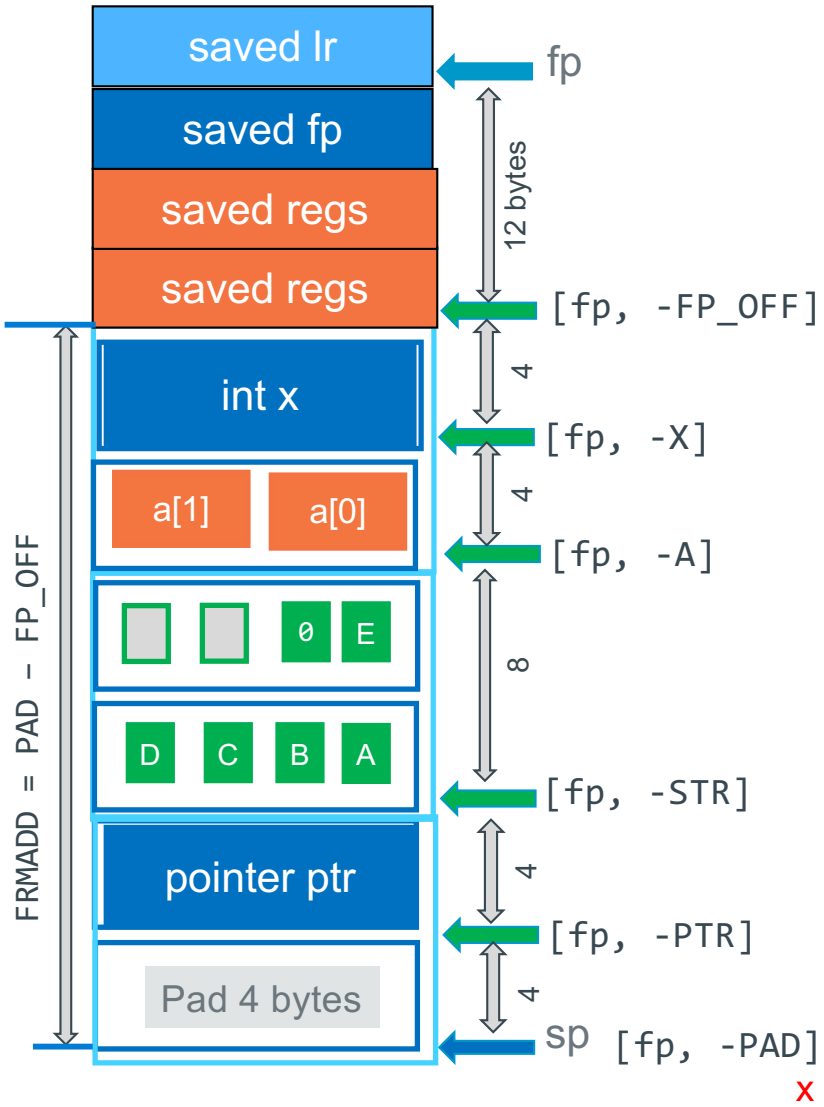
- Assign label names for each local variable
  - Each name is .equ to be the offset from fp

Variable name	Size	Name	expression size+prev	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	8 + 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 //frames can be large
    sub     sp, sp, r3 // add space for locals
    // rest of function code
    // no change to epilogue
    sub     sp, fp, FP_OFF  // deallocate locals
    pop     {r4, r5, fp, lr}
    bx      lr
    .size   func, (. - func)
```



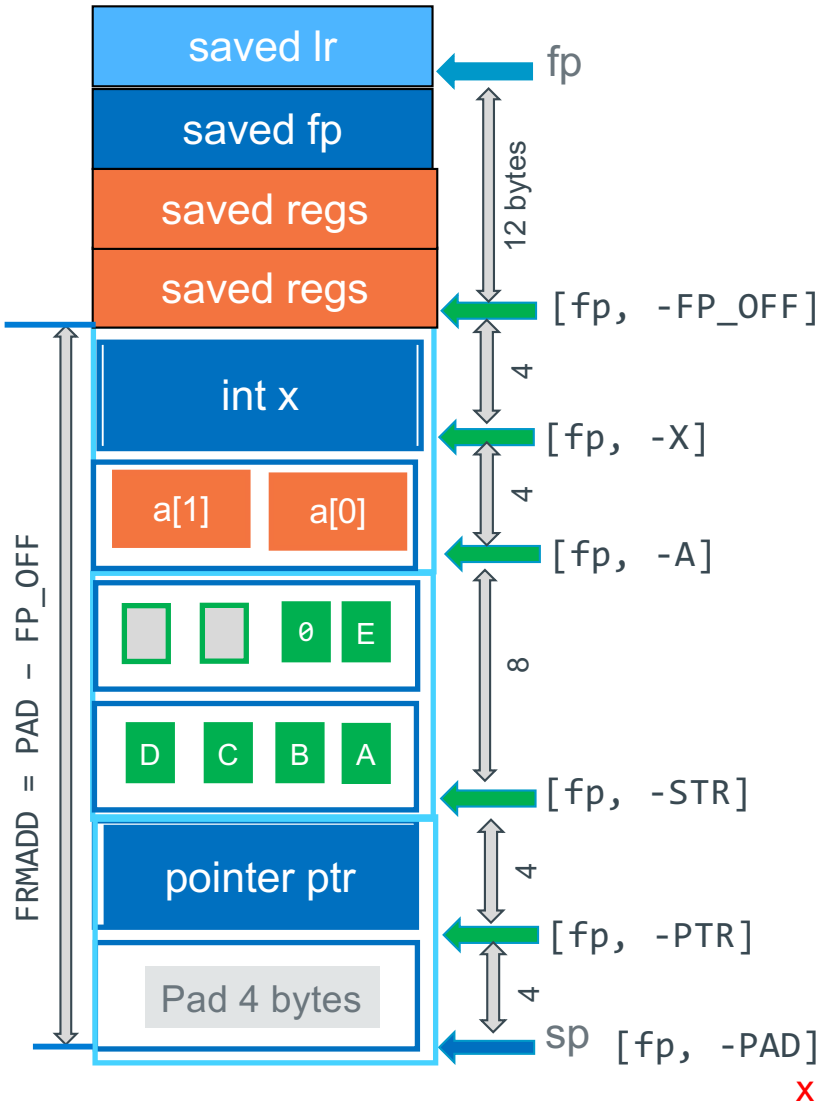
Accessing Stack variables

var	how to get the address	how to read 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 contents
ptr	ldr r0, =PTR str r1, [fp, -r0]

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



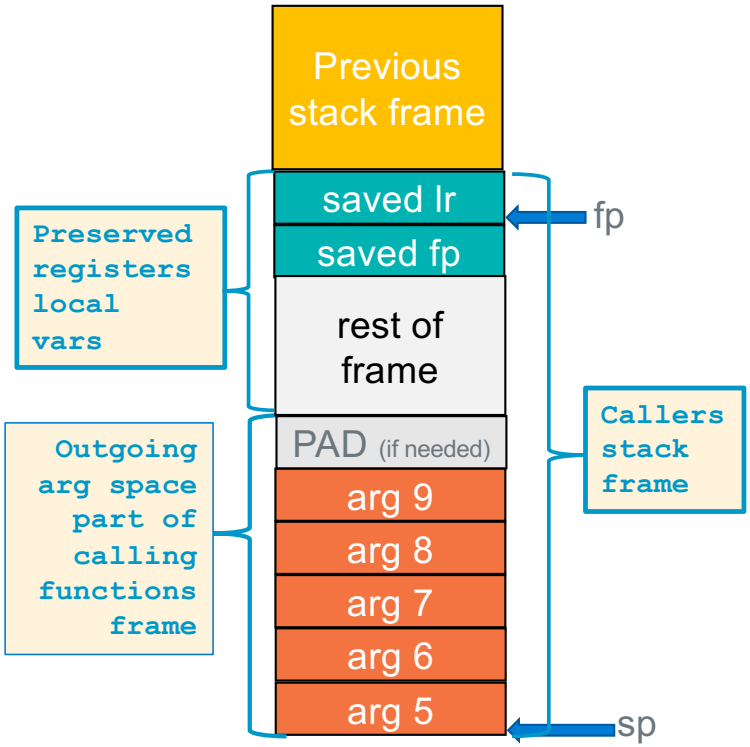
Calling Function: Allocating Stack Parameter Space

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

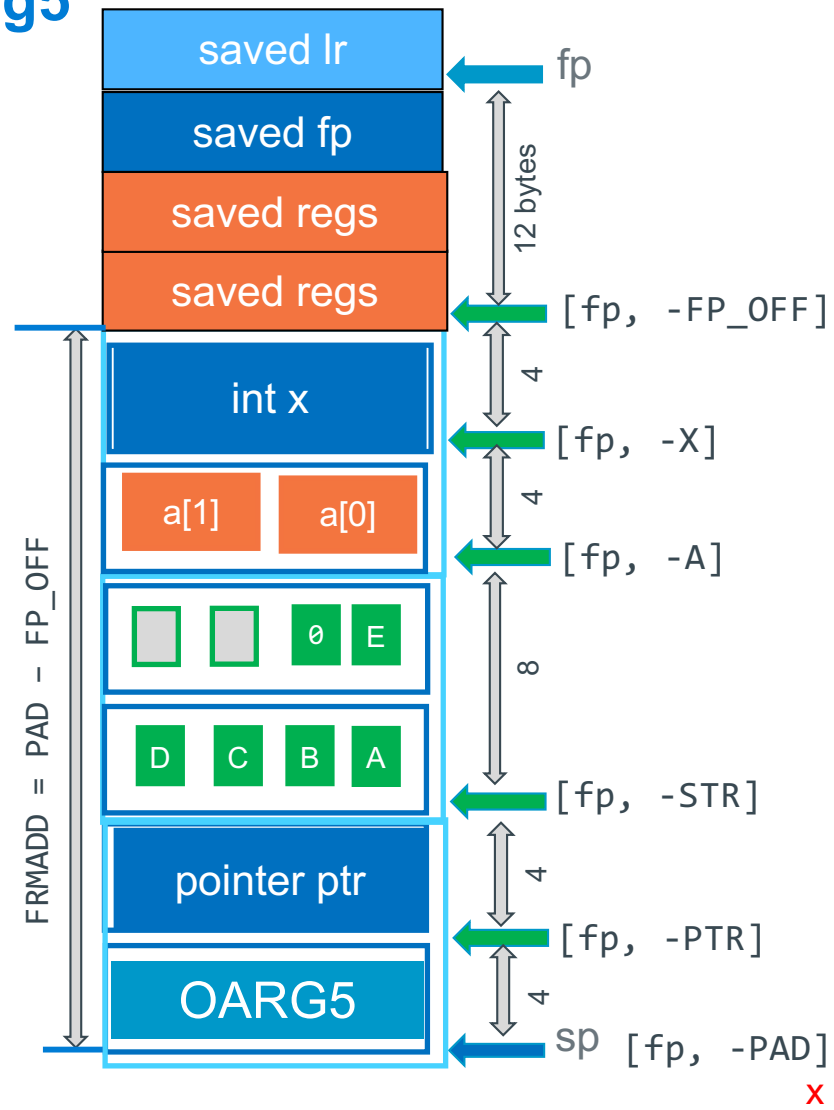
1. **arg5 must be pointed at by sp**
2. **SP must be 8-byte aligned**

x

Step 3 Allocate Space in the Prologue + arg5

Add space to previous example for passing 5 arguments to a function that will be called

```
    .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,          0 + PTR
    .equ      OARG5,        4 + PAD
    .equ      FRMADD        OARG5 - FP_OFF
func:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD //frames can be large
    sub     sp, sp, r3 // add space for locals
```




Version 1.00
UCSD CSE 30
Computer Organization and Systems Programming
PA9 – Part 2 main() argv and calling setup
Keith Muller

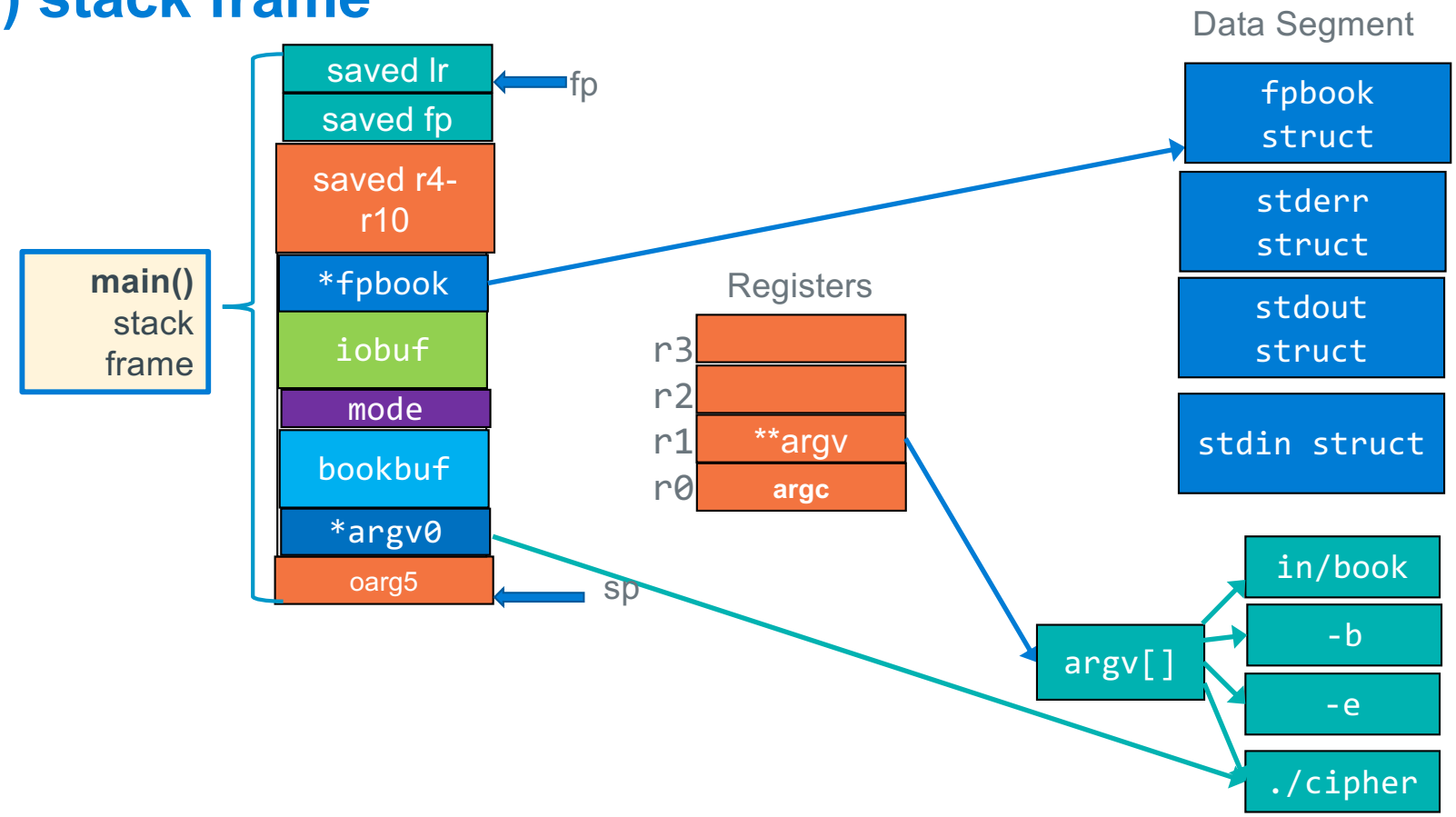
main.c

```
int main(int argc, char **argv)
{
    /*
     * do not change the definition order of these local variables
     */
    FILE *fpbook;
    char iobuf[BUFSZ];
    int mode;
    char bookbuf[BUFSZ];
    char *argv0;
    int cnt; /* do not put on stack, use a register for this */


    /*
     * parse the command line arguments, set mode (encrypt or decrypt)
     * and open the book file
     */
    argv0 = *argv;
    if (setup(argc, argv, &mode, &fpbook) == EXIT_FAIL)
        return EXIT_FAILURE;


    /*
     * read the input and book file until EOF on the input file
     * Either encrypt or decrypt
     * then write it out.
     */
    while ((cnt = rdbuf(stdin, fpbook, BUFSZ, iobuf, bookbuf)) > 0) {
        if (mode == ENCRYPT_MODE)
            cnt = encrypt(iobuf, bookbuf, cnt);
        else
            cnt = decrypt(iobuf, bookbuf, cnt);
        if (fwrite(iobuf, 1, cnt, stdout)  != (size_t)cnt) {
            fprintf(stderr, "%s: write failed\n", argv0);
            fclose(fpbook);
            return EXIT_FAILURE;
        }
    }
    /*
     * close the book file
     */
    fclose(fpbook);
    if (cnt == EXIT_FAIL) {
        fprintf(stderr, "%s: read failed\n", argv0);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```
r0 = setup(r0, r1, r2, r3);
```

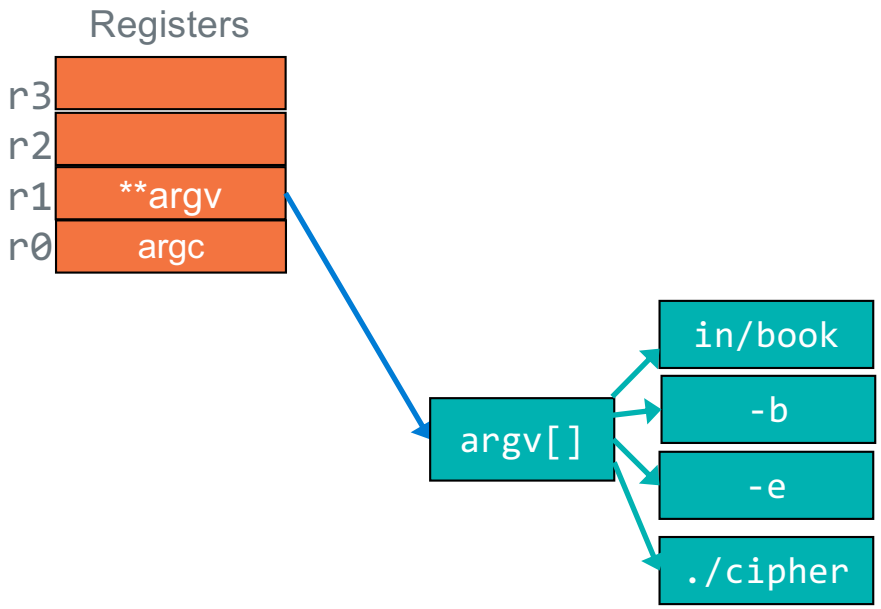
```
r0 = rdbuf(r0, r1, r2, r3, OARG5);
```

main() stack frame



Accessing argv from Assembly

```
    .extern printf
    .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
    ldr     r4, =.Lstr
    mov     r5, 0
    mov     r6, r1
.Lloop:
    // printf("argv[%d] = %s\n", indx, argv[indx])
    ldr     r2, [r6]
    cmp     r2, 0
    beq     .Ldone
    mov     r1, r5
    mov     r0, r4
    bl      printf
    add     r5, r5, 1       //indx++
    add     r6, r6, 4       //argv++
    b       .Lloop
.Ldone:
    mov     r0, 0
    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx      lr
```




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

main.c

```
int main(int argc, char **argv)
{
    /*
     * do not change the definition order of these local variables
     */
    FILE *fpbook;
    char iobuf[BUFSZ];
    int mode;
    char bookbuf[BUFSZ];
    char *argv0;
    int cnt; /* do not put on stack, use a register for this */


    /*
     * parse the command line arguments, set mode (encrypt or decrypt)
     * and open the book file
     */
    argv0 = *argv;
    if (setup(argc, argv, &mode, &fpbook) == EXIT_FAIL)
        return EXIT_FAILURE;


    /*
     * read the input and book file until EOF on the input file
     * Either encrypt or decrypt
     * then write it out.
     */
    while ((cnt = rdbuf(stdin, fpbook, BUFSZ, iobuf, bookbuf)) > 0) {
        if (mode == ENCRYPT_MODE)
            cnt = encrypt(iobuf, bookbuf, cnt);
        else
            cnt = decrypt(iobuf, bookbuf, cnt);
        if (fwrite(iobuf, 1, cnt, stdout)  != (size_t)cnt) {
            fprintf(stderr, "%s: write failed\n", argv0);
            fclose(fpbook);
            return EXIT_FAILURE;
        }
    }
    /*
     * close the book file
     */
    fclose(fpbook);
    if (cnt == EXIT_FAIL) {
        fprintf(stderr, "%s: read failed\n", argv0);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```
r0 = setup(r0, r1, r2, r3);
```

```
r0 = rdbuf(r0, r1, r2, r3, OARG5);
```

setup() stack frame