CSE 30 Fall 2022 Programming Assignment #9 (Vers 1.1c)
Due Sunday December 4 , 2022 @ 11:59PM

Remember: 'Start Early'

Grading (50 points total) What You Need To Do

You must complete all of the following items by the due date to get all points.
The absolute last day to turn in is Wednesday December 7, 2022 at 11:59 PM.

There are **50 points available** for this PA that are distributed as follows:

- ☐ Up to 10 points (at our discretion) if the following files compile/assemble without warnings. This has to be a meaningful attempt to write the code that meets the specifications of the program. Random ARM assembly statements or empty files will not be awarded any points.

  Your submitted code **must only use instructions and addressing modes included on the CSE30 green card.** Any instructions or addressing modes not described in the CSE30 green card may cause your solution to be rejected.

  These are the only files that are part of PA9:

  ```
      main.S (7 points)
      rdbuf.S (3 points)
  ```

- ☐ Up to 30 points for passing the tests in the included test harness
- ☐ Up to 10 points for passing the gradescope tests (these will be run after the late deadline).

Assignment – Stack Frames, Bitwise, and Passing Arguments

PA9 is the second part of a two part project where you will be working on a program called ***cipher***. ***Cipher*** is a file encryption/decryption program that uses a variation of what is called a book cipher. ***Cipher*** will both **encrypt** and **decrypt** **any type of file** **(text and binary)** using a ***combination*** of **per byte bit manipulation** (using bitwise operations) and an **exclusive or** (EOR in arm ^ in C) encoding using sequence of **bytes from a book file** (traditionally a real book, but it can be any text or binary file).

cipher reads the file to be processed from standard input, either encrypts or decrypts the file, and then writes the result to standard output. Cipher has two required arguments:

1. A flag that specifies on of ***either*** encrypt (**-e**) or decrypt (**-d**) operational **mode**
2. A flag **-b** ***<filename>*** that specifies the name of the book file that contains the encoding keys

Synopsis: ./cipher (-d|-e) -b <bookfile>

Flag	Description
-d	Sets the program to decrypt. **<u>Exactly</u> 1 of -d OR -e must be provided, but <u>not</u> both.**
-e	Sets the program to encrypt. **<u>Exactly</u> 1 of -d OR -e must be provided, but <u>not</u> both.**
-b *bookfile*	The file path to the input bookfile. This is required.

Error Messages and Return from main()

All error messages are output in main() (In the below, argv0 is argv[0])

When the write to stdout fails:

```
printf(stderr, "%s: write failed\n", argv0);
```

When either the read of stdin or bookfile fails or the bookfile is too short in length:

```
fprintf(stderr, "%s: read failed\n", argv0);
```

Command line options are handled for you by setup() (in setup.c). Any errors detected by setup are sent to stderr and setup() returns EXIT_FAIL (see cipher.h). **No other errors need to be detected.**

`main() returns EXIT_SUCCESS` when the program completes successfully (no read or write I/O errors), and `EXIT_FAILURE otherwise.`

How the program works

The program is called with a decrypt or encrypt flag, a bookfile flag and a file path for a bookfile. Command line option handling is done for you in the supplied function setup() (written in C in the file setup.c).

Inputs:

- Decrypt flag OR encrypt flag
- Bookfile flag and file path of the bookfile
- Standard input (usually redirected as

```
./cipher -e -b in/BOOK < input_file >output_file
```

Outputs:

- When encrypting, the output goes to **`stdout`**

- When decrypting, the output goes to **`stdout`**
- Options handling and the opening of the bookfile is handled for you in setup(). If setup fails (returns EXIT_FAIL as defined in cipher.h), cipher exits with EXIT_FAILURE.

Getting the starter code

The starter files are obtained from github. **You will be using a fresh git clone from the PA8 starter files** for PA9.

```
cs30fa22@pi-cluster-153:~ % git clone https://github.com/cse30-fa22/PA8_starter.git PA9

cs30fa22@pi-cluster-153:~ % cd PA9

cs30fa22@pi-cluster-153:~ % chmod 0755 runtest in/cmd*

cs30fa22@pi-cluster-153:~ % chmod 0444 in/test* in/BOOK in/SHORTBOOK

cs30fa22@pi-cluster-153:~ % chmod 0444 exp/*
```

You can use the supplied solution versions of encrypt and decrypt for PA9. If you want to use your versions, copy them over from the PA8 directory but you can just use the solution versions for PA9.

It is strongly suggested that you start by writing rdbuf.S first. It is a short program with no loops.

Modifying SELVERS.h to Control Compiling & Assembling

In the starter file there is a file called SELVERS.h, which has four sections of control, labeled (A) through (D) below.

(A) In the section below, select whether you want to use the C version, the assembly version or the library solution version for the function decrypt.

```
// at most one of the following two should be uncommented
// if both are commented out, use the solution code (Suggested for PA9)
//#define MYDECRYPT_C       // when defined will use your decrypt.c for PA8
//#define MYDECRYPT_S       // when defined will use your decrypt.S for PA8
```

(B) In the section below, select whether you want to use the C version, the assembly version or the library solution version for the function encrypt.

```
// at most one of the following two should be uncommented
// if both are commented out, use the solution code (Suggested for PA9)
//#define MYENCRYPT_C       // when defined will use your encrypt.c for PA8
```

```
//#define MYENCRYPT_S       // when defined will use your encrypt.S for PA8
```

(C) In the section below, select whether you want to use the C version (for PA8), the assembly version (for PA9) for main().

```
// only one of the following two must be uncommented
//#define MAIN_C            // Must be uncommented to use Cmain.c for PA8
#define MYMAIN_S          // when defined will use your main.S for PA9
```

(D) In the section below, select whether you want to use the C version (for PA8), the assembly version (for PA9) for rdbuf().

```
// only one of the following two must be uncommented
//#define RDBUF_C           // Must be uncommented to use Crdbuf.c for PA8
#define MYRDBUF_S         // when defined will use your rdbuf.S for PA9
```

Other code in SELVERS.h (not shown above) will cause an compilation/assembly error if the rules described above are not met. Be careful when editing this file as described above.

Compiling and assembling the starter code

Same as PA8.

Running the Tests - What Passing the tests Looks like

Same as PA8.

You should edit SELVERS.h to test your two routines with the test fixture to make sure everything works properly. For example, here are some (but not all) combinations you might want to try:

1. Your rdbuf.S (develop this function first).
2. Your main.S
3. Both your main.S and rdbuf.S

Writing the replacement routines

1. **Write ONLY the following replacement functions for PA9. It is suggested you carefully read the supplied C versions to understand how they work. Also read setup.c as you have to call it from main().**
   ```
   main()          in file: main.S   // arm32 assembly
   rdbuf           in file: rdbuf.S  // arm32 assembly
   ```
2. **These (and SELVERS.h) are the only files that you should modify for PA9**

3. Make sure that you remove the comment for the correct versions you want to test in SELVERS.h

When you are designing your assembly language functions, once you have settled on an algorithm, your next task is to decide which registers are going to be used for what purpose. On entry to a function, r0-r3 are used to pass the first four arguments. It is ok to modify these as you see fit in your code. However, if you are going to make a call to another function, remember that the called function has the right to change these. So if you will still need the parameters after making a function call, either copy them to a protected register or save them on the stack.

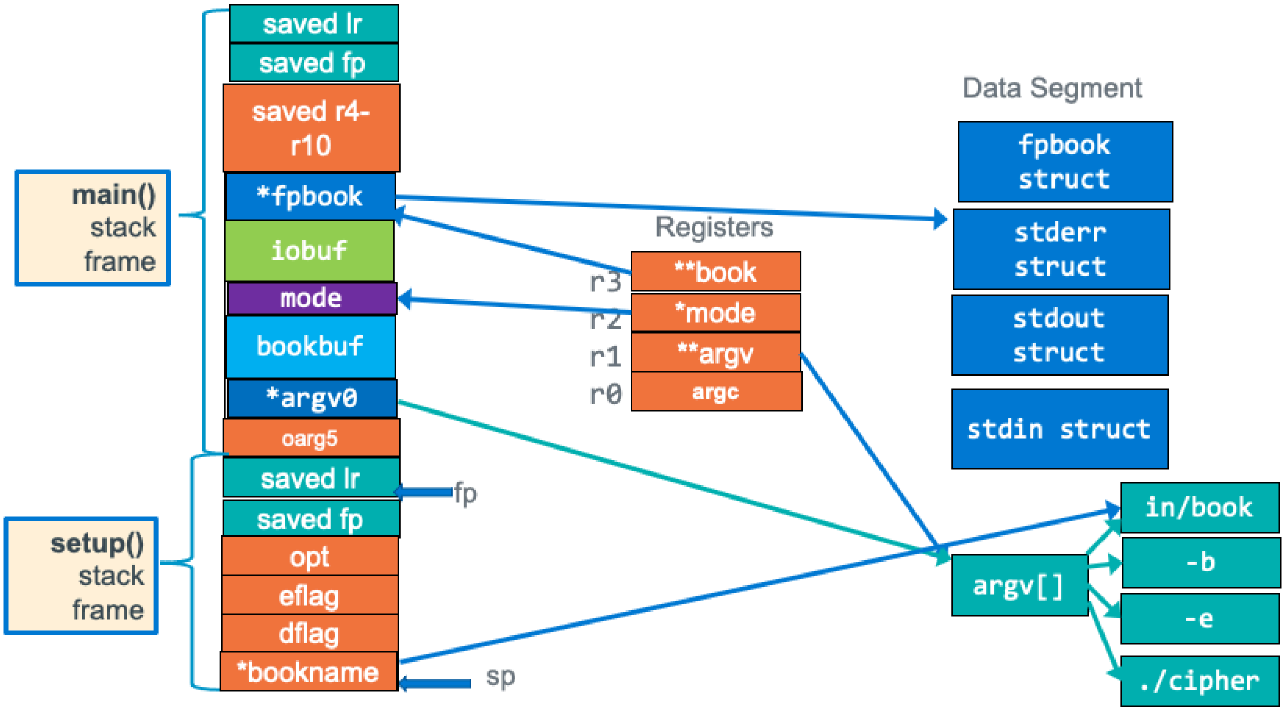
When designing your assembly functions, decide which registers you are going to use for each local variable first. If the function you are working on does not call any function that has output parameters, you can put all the variables in registers (combination of preserved and scratch registers). Output variables require a pointer to be passed (the address) to the function you are calling and it is not possible to get the address of a register. Parameters that do not fit in 32-bits are passed as pointers. Try to use the scratch registers first, remembering that function calls that you make can change the values in r0-r3.

One way to keep track of the registers is to create a table either in your code inside a comment at the top of each function or elsewhere (like a sheet of paper) that describes how you have decided to use each register relative to your algorithm. Example:

```
        // register use
        // r0 on entry contains the address of iobuf (char * pointer)
        // r1 on entry contains the address of bookbuf (char * pointer)
        // r2 on entry contains cnt
        // r3 is loop counter i
        // r4 contains the shifted byte
```

Carefully look at how return values are used from the functions in the C code. Make sure you return the same values in your assembly language equivalent programs.

setup() stack frame



`int setup(int argc, char **argv, int *mode, FILE **fpbook)`

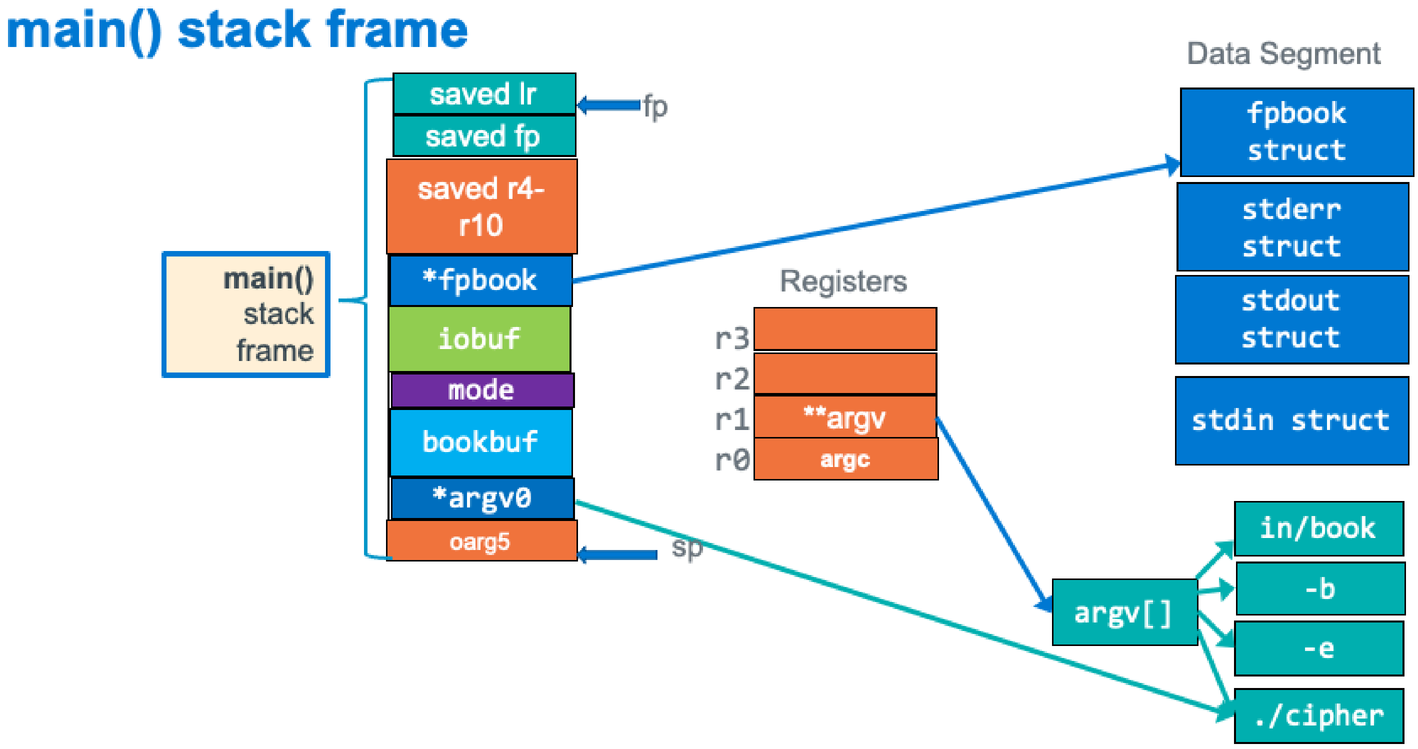
This function is supplied in source form and you do not have to write an assembly language version of it. You do need to call it from your assembly language version of main().

Setup is passed four arguments, using registers r0-r3. Argc (r0) and argv (r1) are input parameters and *mode (r2) and **fpbook (r3) are both output parameters (*mode and **fpbook both point to variables allocated space in main()'s stack frame). **fpbook is a pointer to a memory location in main()'s stack that contains a FILE * pointer (the return value of fopen()). Setup() will use the passed pointers (addresses) to store values into the space allocated for mode and fpbook in main(). Setup() uses getopt() to parse the command line options to determine if the mode of operation is either encrypt or decrypt, setting the value in output variable mode (using the address passed to it by main(). Next setup() open the bookfile using the filename operand to the -b option. It calls the C library function fopen() to do so, then storing the FILE* value returned in the output parameter fpbook.

It is important to note that the first two arguments to setup are argc (in r0) and arg (in r1), which are passed unchanged from main. When writing main.S, registers r0 and r1 at the entry to main are already correct, so all you have to do is pass a pointer to a stack variable mode in register r2 and a pointer to the stack variable fpbook in register r3.

Setup() like all functions, puts its return value in register r0. Setup() returns a 0 (EXIT_SUCCESS) on success and EXIT_FAILURE on any error. When EXIT_FAILURE is returned from setup(), main()

exits returning an EXIT_FAILURE.



```
int main(int argc, char **argv)
```

Study the supplied C source in Cmain.c and understand how the function works. Examine how each function is called, what the parameters are, and what the return value means.
Here is how you read the man pages on the functions used in main() on the pi-cluster

```
% man 3 fwrite
% man 3 fclose
% man 3 fprintf
```

Main() needs to have the layout of its stack frame finished. Use the process that we did in lecture (and the how to video) to complete the stack design. You need to allocate space for each local variable and for the 5th argument passed to rdbuf(). Make sure that the **total size of the stack frame is evenly divisible by 8.** Adjust the space allocated for PAD as needed.

main() is passed two parameters, argc and argv. These two arguments are in r0 and r1.

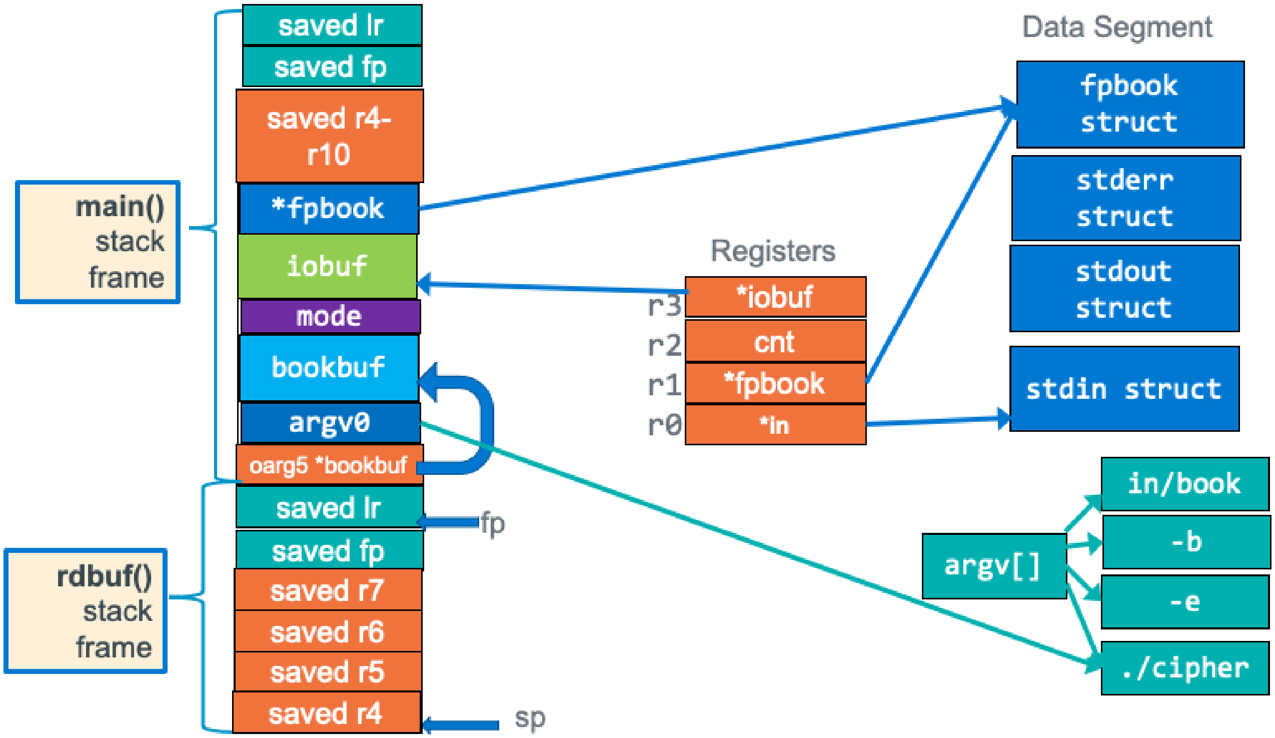
The call to setup() involves two output parameters. You should read the code in setup.c as well as the description above to understand what setup() expects as parameters and what it returns in r0.

After the call to setup(), the body of main() is a large loop that is controlled by a call to rdbuf().

```
.Lloop:
```

```
            cnt = rdbuf()
            If EOF or ERROR,loop is over
            // based on mode either encrypt() or decrypt()
            Frwite iobuf
            If cnt not written, error and loop is over
            Go to.Lloop
    .Ldone:
            fclose(fpbook)
            Return EXIT_SUCCESS or EXIT_FAILURE
```

rdbuf() stack frame



```
int rdbuf(FILE *in, FILE *fpbook, int cnt, char *iobuf, char *bookbuf)
```

Study the supplied C source in Crdbuf.c and understand how the function works. Examine how each function is called, what the parameters are, and what the return value means.

Here is how you read the man pages on the functions used in rdbuf() on the pi-cluster

```
% man 3 feof
% man 3 ferror
% man 3 fread
```

rdbuf() is passed five parameters, three input parameters:in, r0=FILE *in, r1 = FILE *fpbook, and r2=cnt. There are two output parameters: r3 =iobuf and the stack parameter located at [fp,IARG5] = bookbuf.

rdbuf() fills two buffers iobuf and bookbuf with the same number of characters using the C library function fread() to read from the FILE *in and the FILE*fpbook.

On entry to rdbuf() it checks if EOF has been reached on in (which is stdin passed to it by main(). If EOF has been reached, rdbuf() returns 0 signaling to main() that the end of the input has been reached. If EOF has not been reached, it then checks for an IO error condition on FILE *in using ferror(in). If there is an error, it returns EXIT_FAIL.

Then rdbuf reads up to a block of bytes (BUFSZ as defined in main.h) from FILE *in into the array iobuf with a single call to fread(). fread() returns the number of bytes actually read (which is <= BUFSZ, see main.h). When fread() returns 0, there are no characters to read and rdbuf() returns 0 to main().

Next rdbuf() uses fread() on the FILE *book to read exactly the same number of bytes into the bookbuf that it read from FILE *in.

For example if the fread() returns 100 bytes from the input file, this function using fread() then reads exactly 100 bytes from the bookfile.

If rdbuf() does not fill both buffers with the same number of bytes it exits with EXIT_FAIL. Otherwise it returns the number of bytes read.

Coding Requirements

1. You will write your code only in hand-generated arm32 assembly Your submitted code must only use instructions and addressing modes included on the CSE30 green card. Any instructions or addressing modes not in the CSE30 green card may cause your solution to be rejected with 0 points.

2. You are not allowed to use any automated tools that generate assembly language source code.

3. Your code must run in a Linux environment on the pi-cluster.

4. Do not use recursive solutions in your code.

5. In PA9 you should not need to use any helper functions.

6. **You cannot use any downloaded software** (other than what is in the github repository **for this PA**). **If it is not already installed on the pi-cluster by ITS, you cannot use it.**

7. Make sure you pass all the tests before submitting to gradescope.

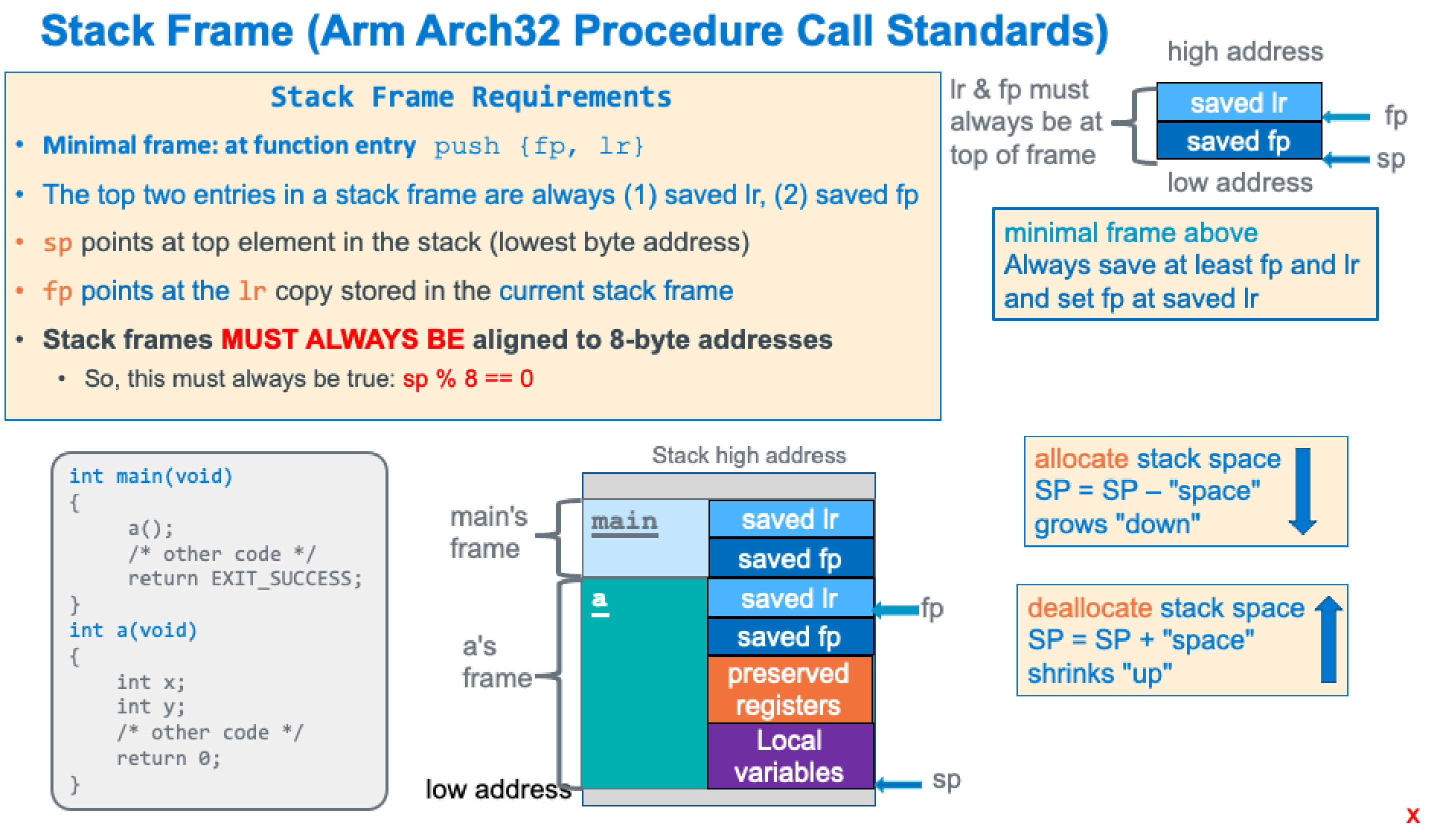
Turning in Files For Your Grade

Before submitting your program to gradescope, <u>make sure</u> at a minimum that your program passes the supplied tests in the test harness.
Submit to gradescope under the **assignment titled PA9**
ONLY submit the following files (total of two (2) files):

```
    main.S
    rdbuf.S
```

You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder.
You can also zip all the files and upload the .zip to the assignment. Ensure that the files you submit are not in a nested folder.

Appendix 1: General Background



Stack Frame (Arm Arch32 Procedure Call Standards)

Stack Frame Requirements

- **Minimal frame: at function entry** `push {fp, lr}`
- The top two entries in a stack frame are always (1) saved lr, (2) saved fp
- 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

lr & fp must always be at top of frame

minimal frame above
Always save at least fp and lr
and set fp at saved lr

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

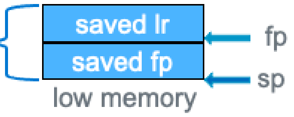
allocate stack space
SP = SP – "space"
grows "down"

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

Function Prologue and Epilogue: Minimum Stack Frame

- Each function has only one Prologue at the top of the function body and only one Epilogue at the bottom of the function body
- When you want to exit the function, set the return value in r0, and then branch to the epilogue
- Function entry (Function **Prologue**):
  1. save preserved registers
  2. set the fp to point at saved lr
  3. allocate space for locals (subtracts from sp)
- Function return (Function **Epilogue**):
  1. deallocate space for locals (adds to sp)
  2. restores preserved registers
  3. return to caller

smallest frame has:
4 bytes between sp & fp



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

one:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        // add space for locals
       // 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

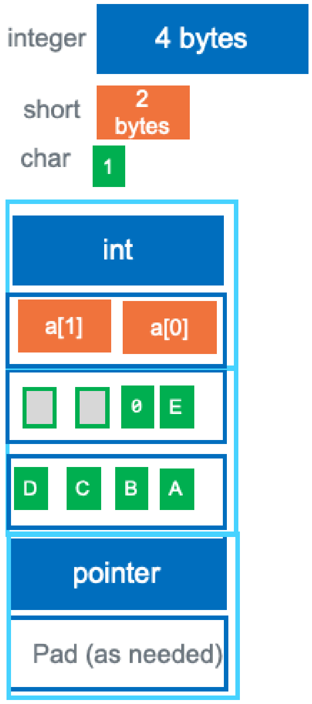
Position the sp

Function Epilogue
always at bottom of function

Function Footer
Assembly directive

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



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

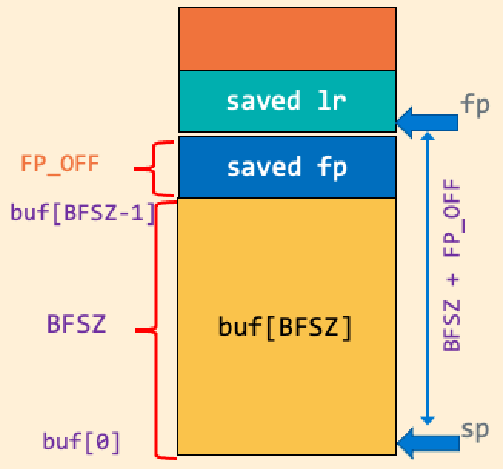
```
#define BFSZ 256
int main(void)
{
   char buf[BFSZ]; // BFSZ bytes
...
```

```
         .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]

stack after allocating local space After sub sp, sp, BFSZ

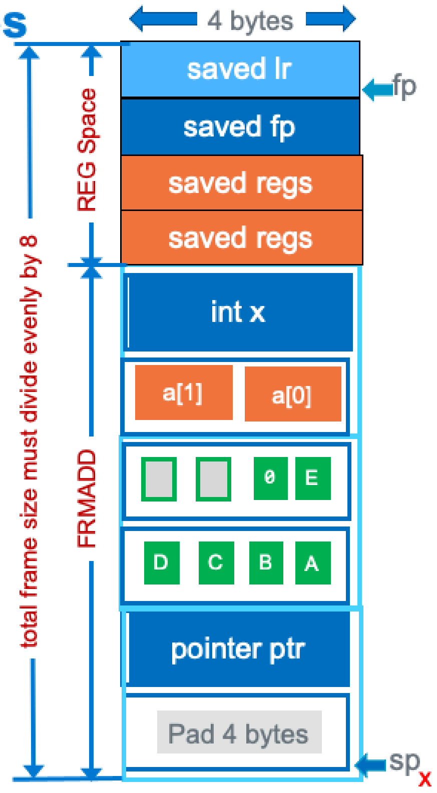


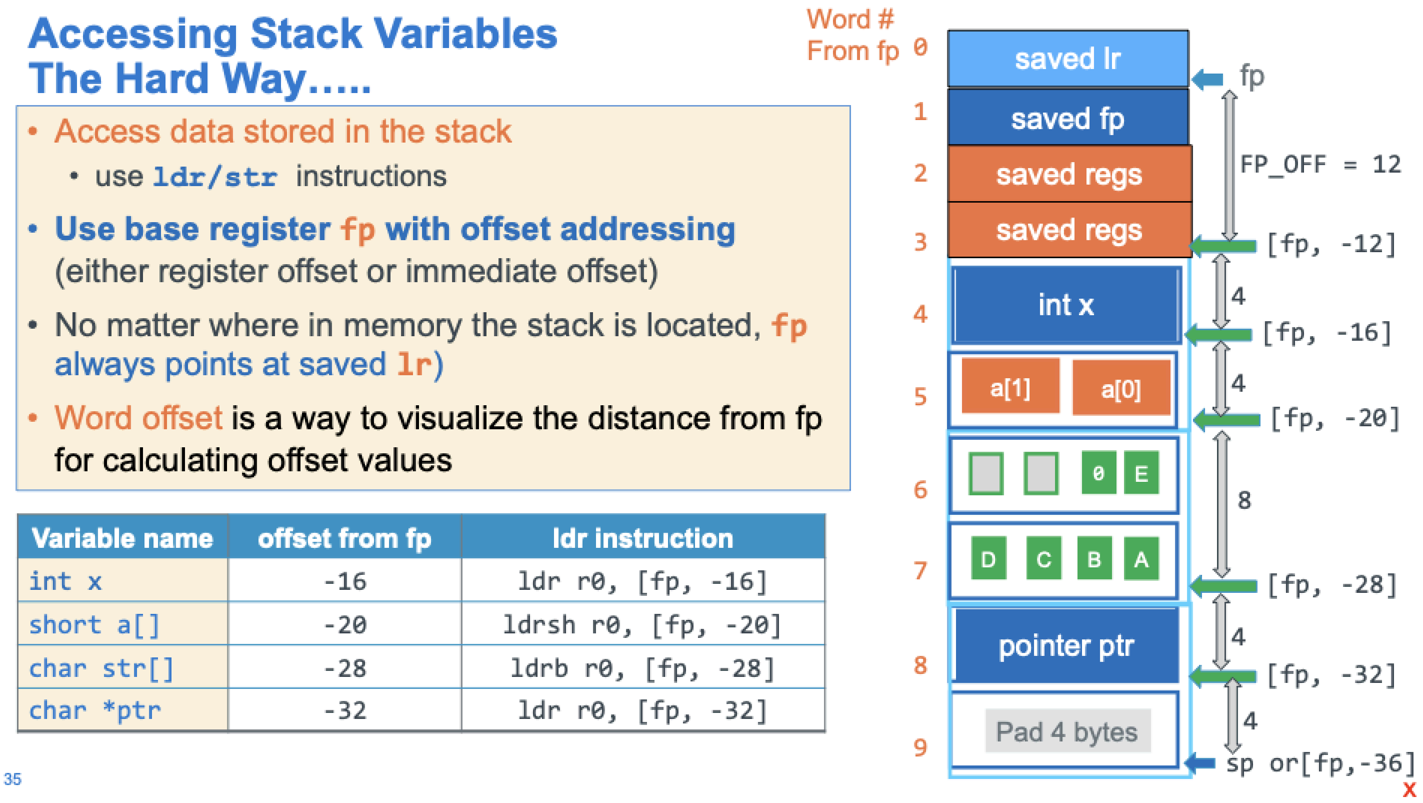
Step 1: Stack Frame Design – Local Variables

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





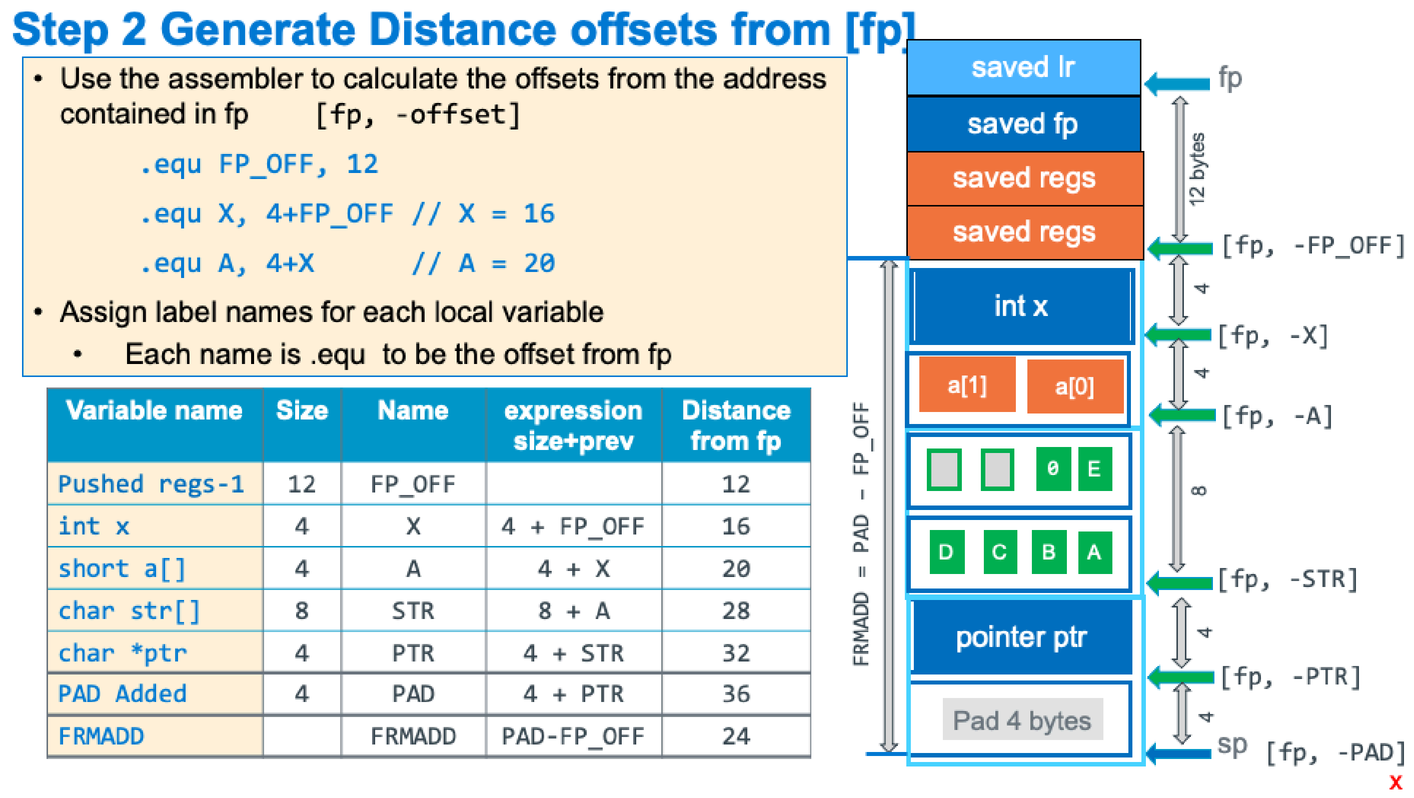
Accessing Stack Variables The Hard Way…..

- 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]

35

Word #
From fp
0
1
2
3
4
5
6
7
8
9
saved lr
saved fp
saved regs
saved regs
int x
a[1]
a[0]
0
E
D
C
B
A
pointer ptr
Pad 4 bytes
fp
FP_OFF = 12
[fp, -12]
4
[fp, -16]
4
[fp, -20]
8
[fp, -28]
4
[fp, -32]
4
sp or[fp,-36]
x



Step 2 Generate Distance offsets from [fp]

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

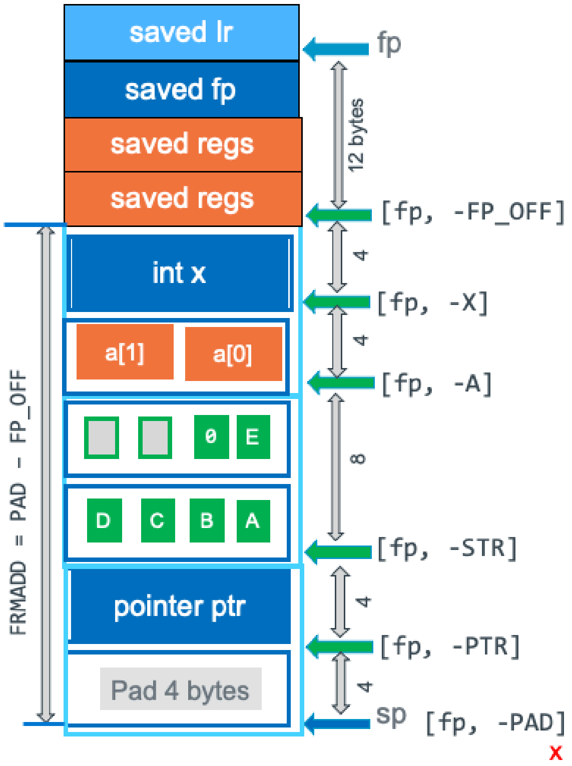
```
.equ FP_OFF, 12
.equ X, 4+FP_OFF // X = 16
.equ A, 4+X      // A = 20
```

- 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

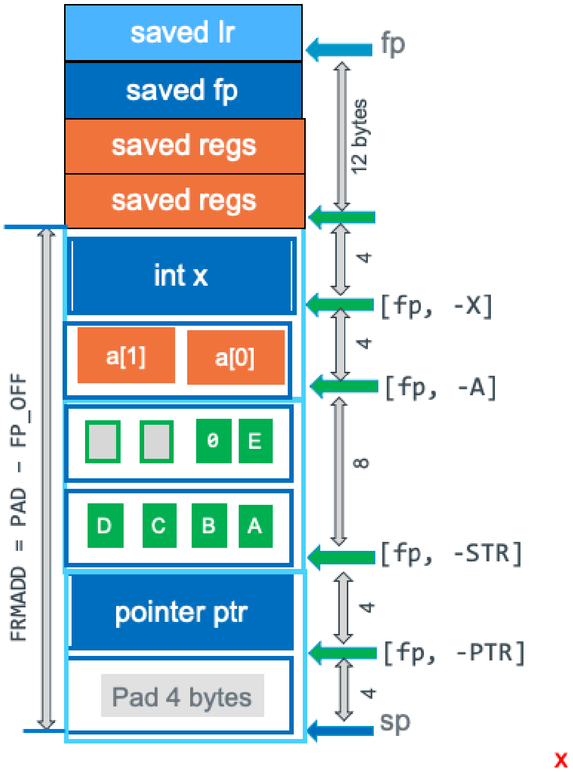
```
    .global func
    .type   func, %function
    .equ      FP_OFF,      12
    .equ      X,           4 + FP_OFF
    .equ      A,           4 + X
    .equ      STR,         8 + A
    .equ      PTR,         4 + STR
    .equ      PAD,         4 + PTR
    .equ      FRMADD       PAD - FP_OFF
func:
    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)
```



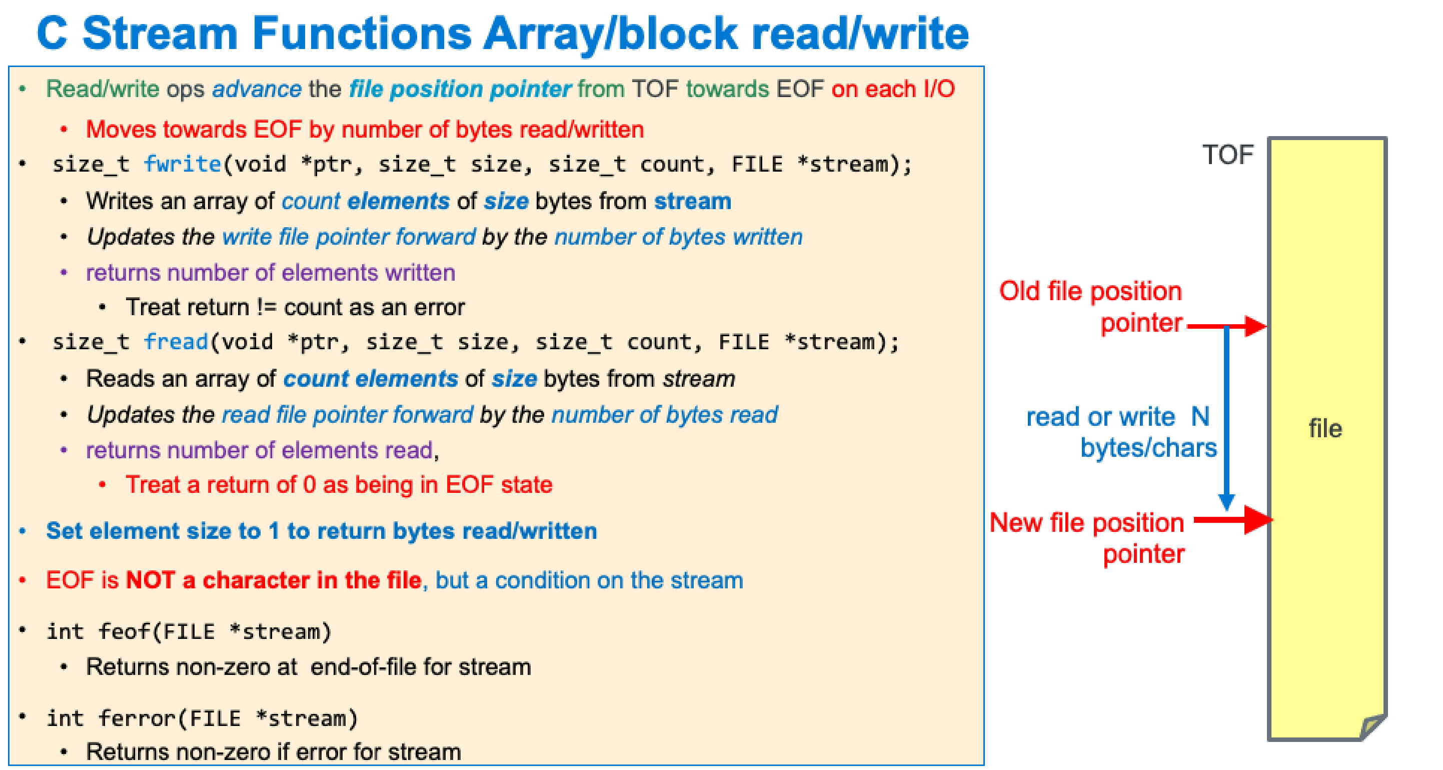
Accessing Stack variables

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]



Appendix 2: fread and fwrite



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 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 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

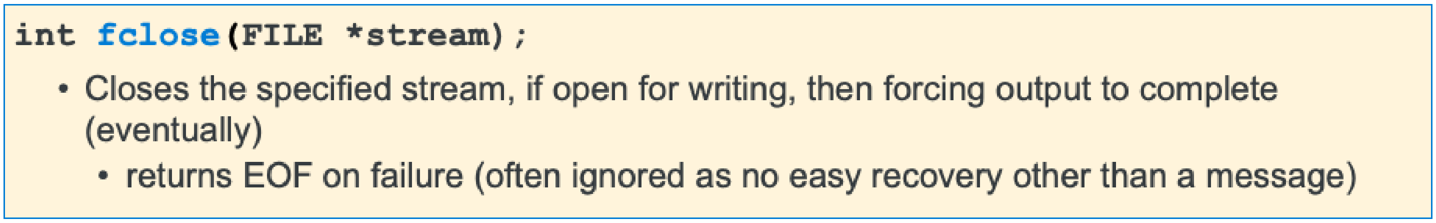
TOF

Old file position pointer

read or write N bytes/chars

New file position pointer

file

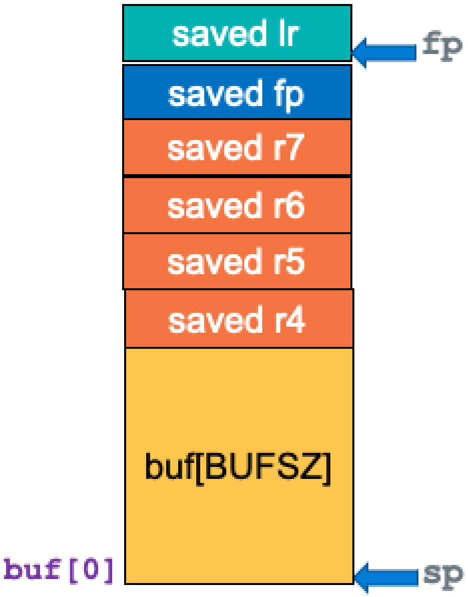


`int fclose(FILE *stream);`

- Closes the specified stream, if open for writing, then forcing output to complete (eventually)
  - returns EOF on failure (often ignored as no easy recovery other than a message)

Example below reads from stdin writes to stdout (copies stdin to stdout with no changes)

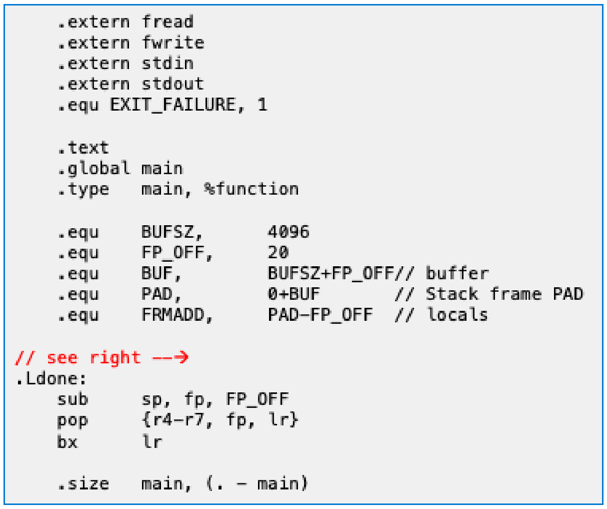
```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BUFSZ 4096

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;
}
```



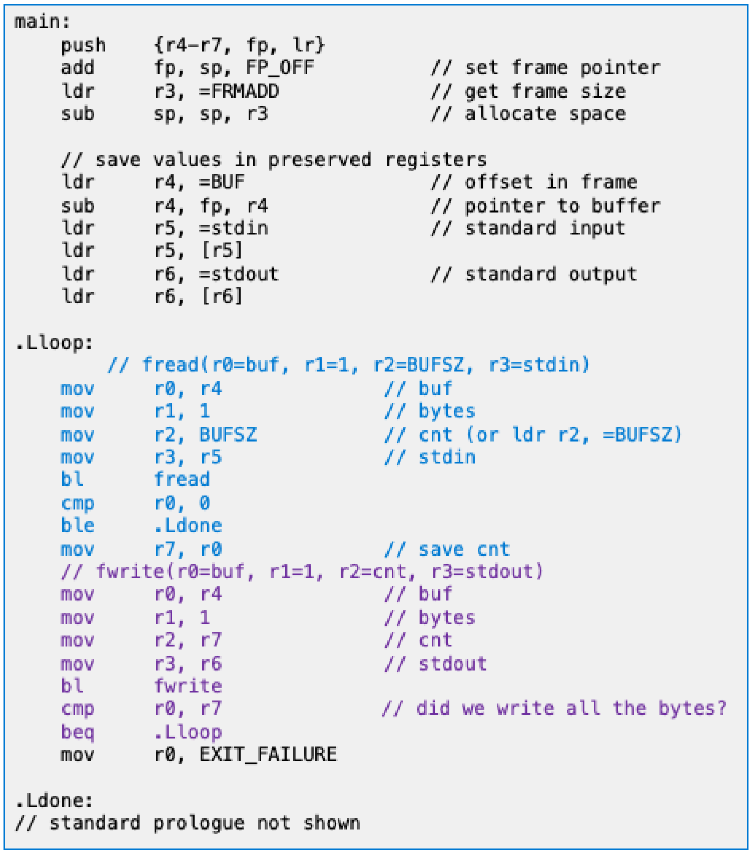
```
    // stack frame
    ldr     r4, =BUF     // offset in frame
    sub     r4, fp, r4   // pointer to buffer
    ldr     r5, =stdin   // standard input
    ldr     r5, [r5]
    ldr     r6, =stdout  // standard output
    ldr     r6, [r6]
```

```
    // 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, r0               // check return value from fwrite
```



```
    .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)
```



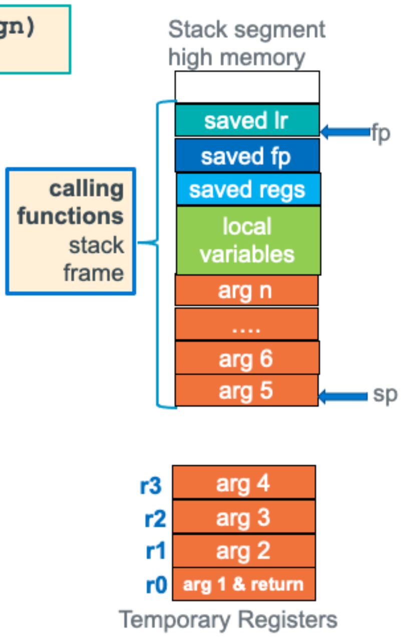
```
main:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF          // set frame pointer
    ldr     r3, =FRMADD             // get frame size
    sub     sp, sp, r3              // allocate space

    // 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
```

Appendix 3: Passing Parameters

Passing More Than Four Arguments - 1

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

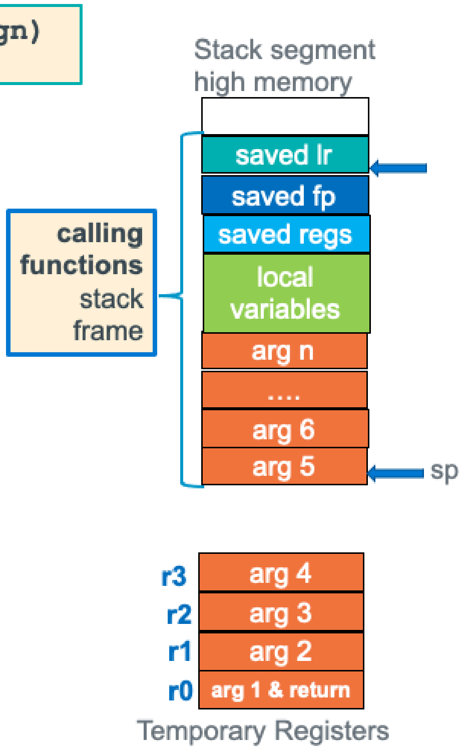
- Each argument is a value that must fit in 32-bits
- **Args > 4 are in the caller's stack frame and arg 5 always starts at fp+4**
  - At the function call (bl) sp points at arg5
  - Additional args are higher up the stack, with one argument "slot" every 4-bytes
- Called functions have the right to change 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



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



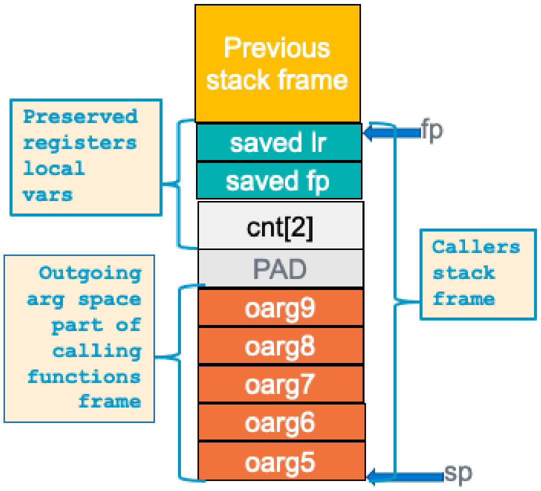
Calling Function: Pass ARGS 5 and higher

```
.equ    FP_OFF,4
.equ    CNT,            8 + FP_OFF       // int cnt[2];
.equ    PAD,            4 + CNT        // added as needed
.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 = r1	`ldr    r0, =OARG5          //distance`
	`str    r1, [fp, -r0]`

OARG6 = &cnt	`ldr    r2, =CNT            //distance`
	`sub    r2, fp, r2          // &cnt`
	
	`ldr    r0, =OARG6          //distance`
	`str    r2, [fp, -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);

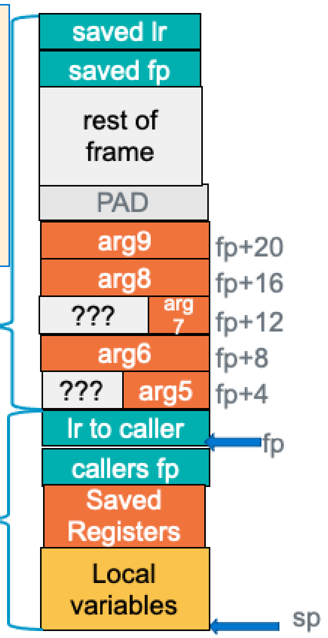
```
int func(int a1, int a2, int a3, int a4,
         short a5, int a6, char a7, int a8, int a9)
```

Constant	Offset	arm ldr /str statement
ARGN	(N-4)*4	`ldr  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

x