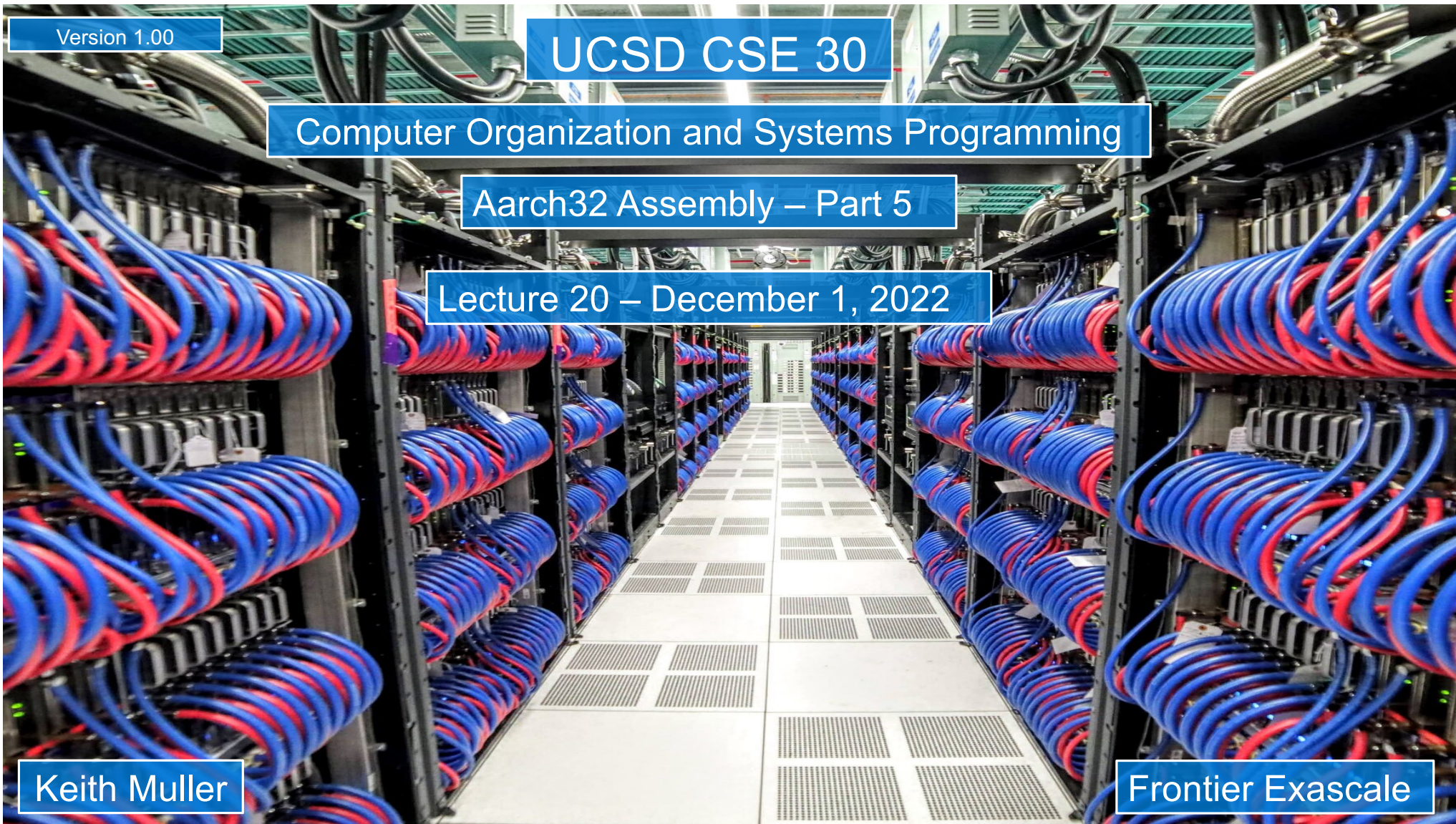Version 1.00

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 5
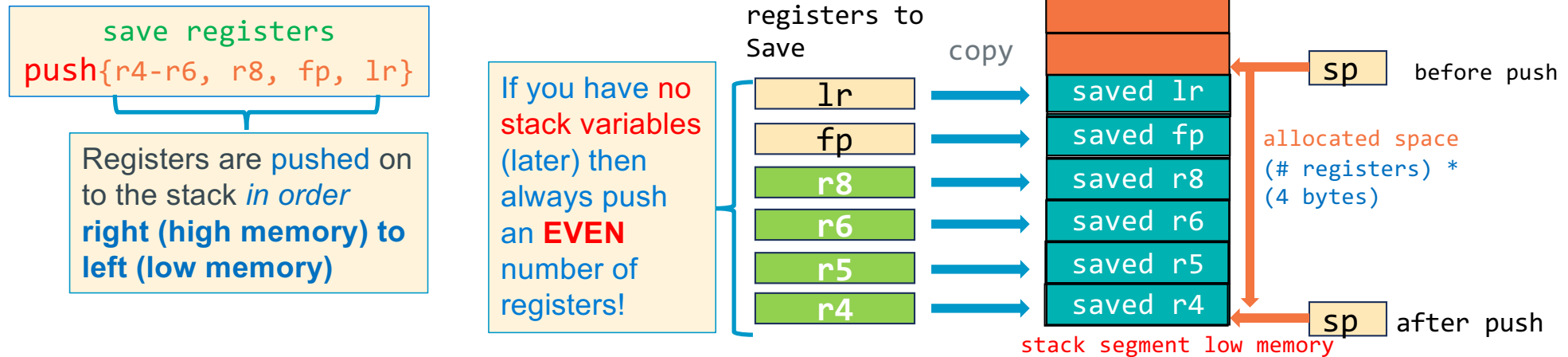
Lecture 20 – December 1, 2022

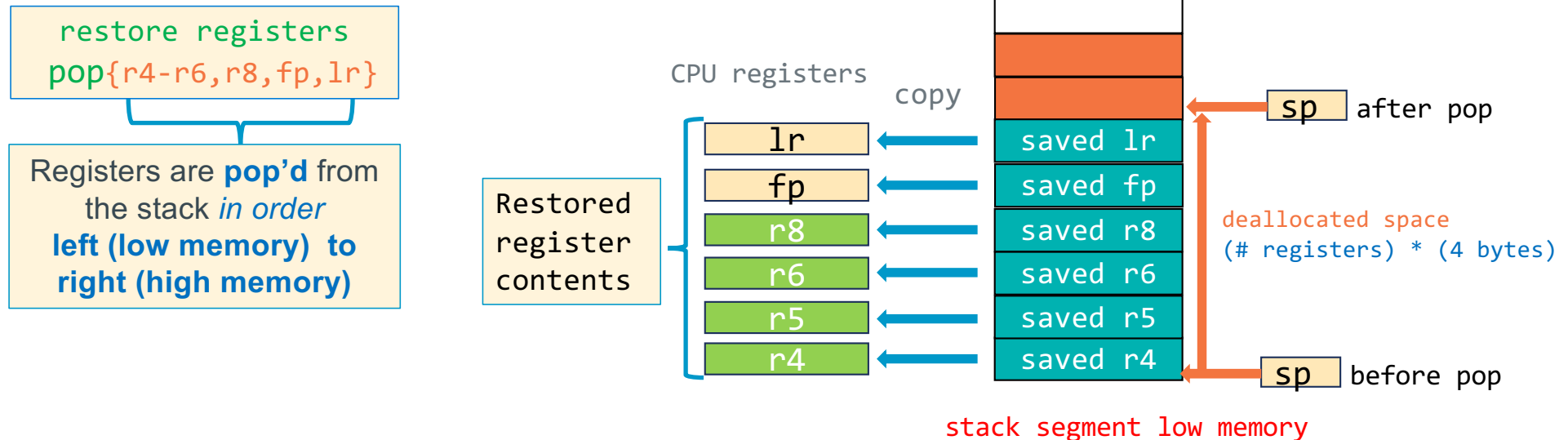Keith Muller

Frontier Exascale

push: Multiple Register Save (str to stack)

stack segment high memory

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

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

CPU registers to Save

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

copy

CPU registers	Stack
lr	saved lr
fp	saved fp
r8	saved r8
r6	saved r6
r5	saved r5
r4	saved r4

sp before push

allocated space
(# registers) *
(4 bytes)

sp after push

stack segment low memory

- **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)

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

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

stack segment high memory

CPU registers

copy

Restored register contents

lr	←	saved lr
fp	←	saved fp
r8	←	saved r8
r6	←	saved r6
r5	←	saved r5
r4	←	saved r4

sp   after pop

deallocated space
(# registers) * (4 bytes)

sp   before pop

stack segment low memory

- **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}** <u>must be the same</u> in both the **push** and the corresponding **pop**

3

X

Local Variables are Part of Each Stack Frame

- Local variables are on the stack below the lowest numbered saved (pushed) register

```
#define BFSZ 4
int main(void)
{
    char buf[BFSZ]; // BFSZ bytes
...
```

X
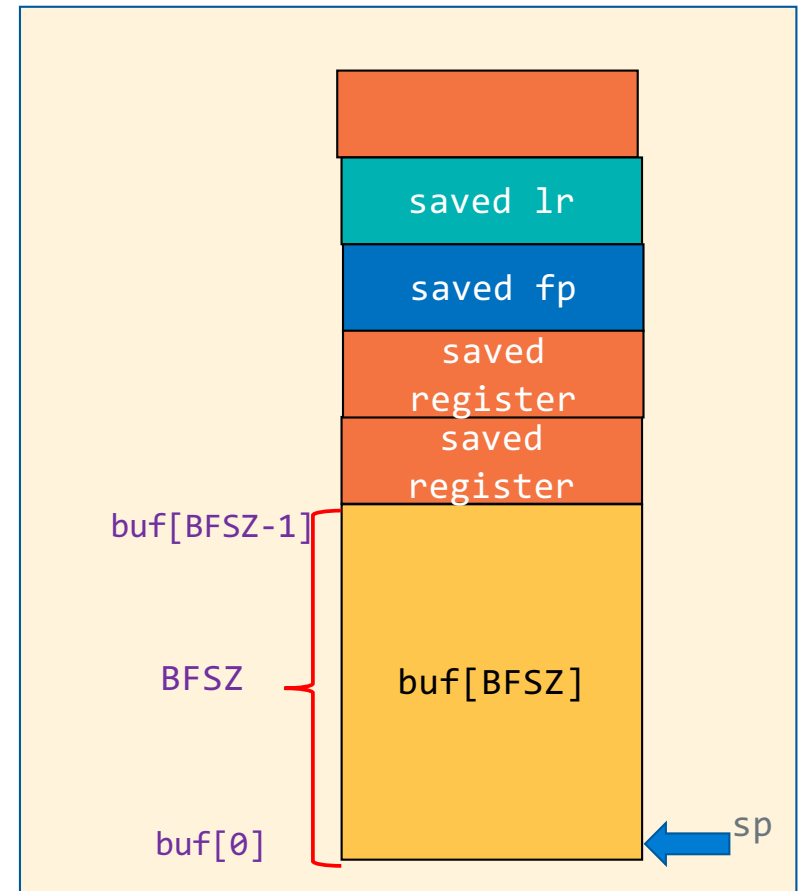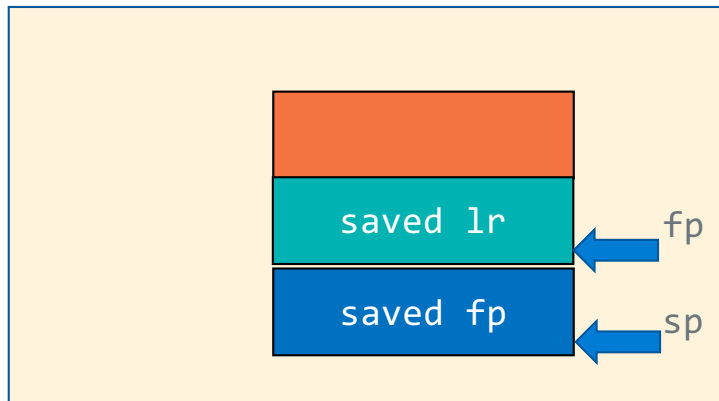
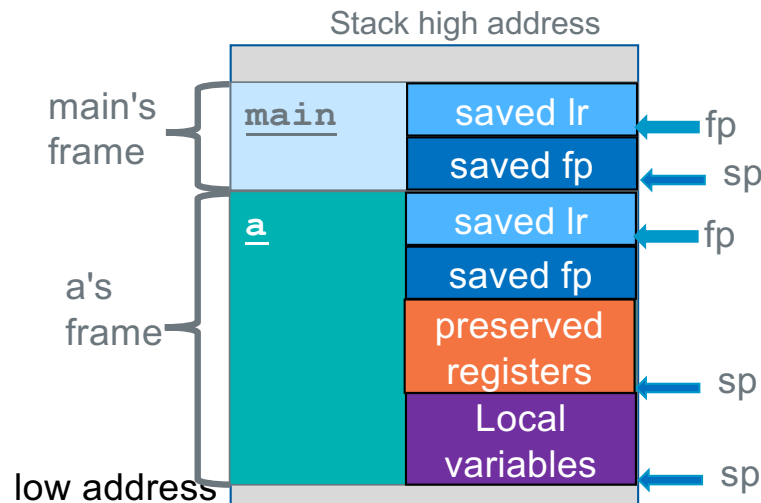# Stack Frame (Arm Arch32 Procedure Call Standards)

Stack Frame Requirements

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

- `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

saved lr ← fp

saved fp ← sp

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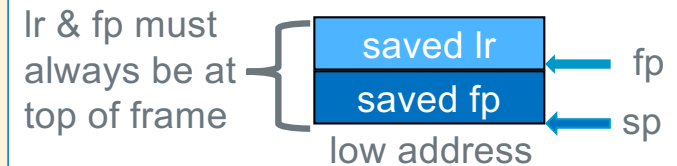
minimal frame above
Always save at least fp and lr
and set fp at saved lr

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

Stack high address

main's frame

main

saved lr ← fp

saved fp ← sp

a's frame

a

saved lr ← fp

saved fp

preserved registers ← sp

Local variables ← sp

low address

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

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

- Function entry (Function **Prologue**):
  1. create frame (subtract
  2. save preserved registers
  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. removes the frame

Note slide has builds

5

X

FP_OFF: Distance from FP to SP
Used to set FP at push and SP before pop

```
        // other code etc
        .equ    FP_OFF,  20
main:
        push    {r4-r7, fp, lr}
        add     fp, sp, FP_OFF
        …….
        sub     sp, fp, FP_OFF
        pop     {r4-r7, fp, lr}
        bx      lr
```
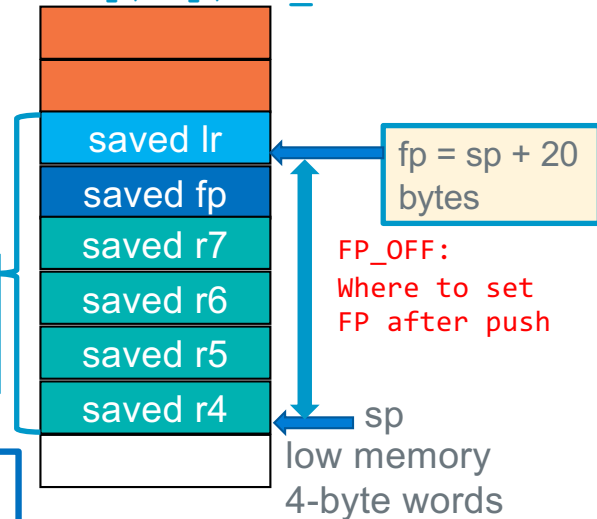
Function Prologue
always at top of function
saves regs and sets fp

Function Epilogue
always at bottom of
function restores
regs including the sp

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

saved lr
saved fp
saved r7
saved r6
saved r5
saved r4

fp = sp + 20
bytes

FP_OFF:
Where to set
FP after push

Function
Stack
Frame

sp
low memory
4-byte words

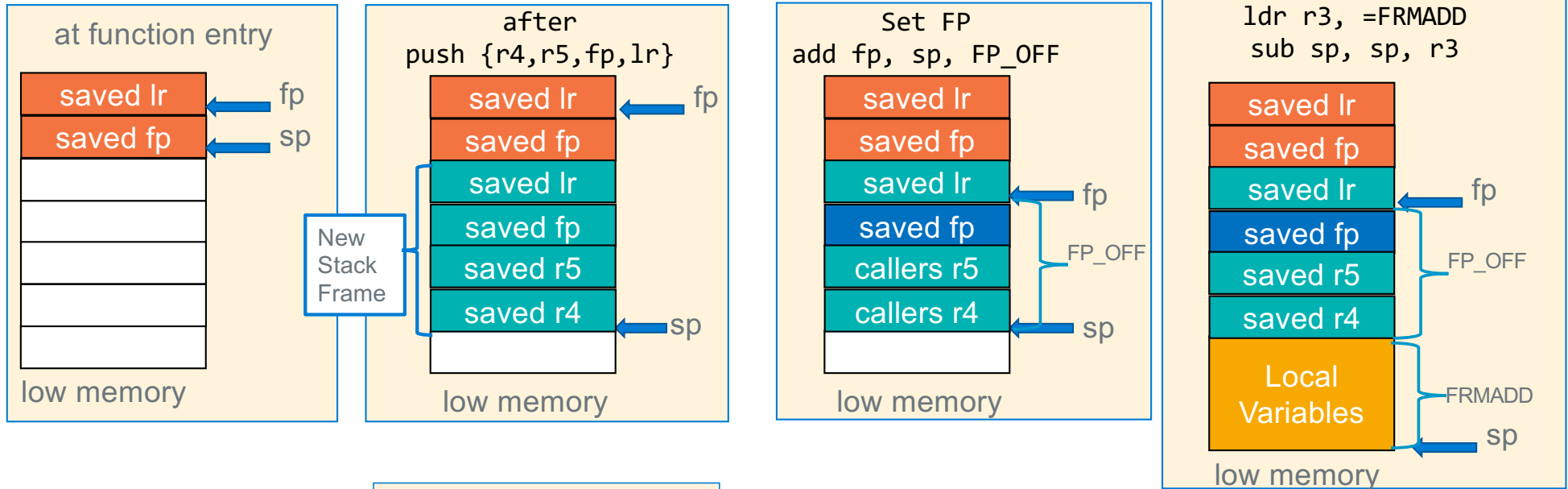| # regs saved	FP_OFF in Bytes
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

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

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

IMPORTANT: FP_OFF has **two** uses:
1. Where to set fp after prologue push (remember sp position)
2. Restore sp (deallocate locals) right before epilogue pop

6

Function Prologue: Allocating the Stack Frame

at function entry

saved lr	← fp
saved fp	← sp

low memory

after push {r4,r5,fp,lr}

saved lr	← fp
saved fp	
saved lr	
saved fp	
saved r5	
saved r4	← sp

New Stack Frame

low memory

Set FP
add fp, sp, FP_OFF

saved lr	
saved fp	
saved lr	← fp
saved fp	
callers r5	FP_OFF
callers r4	← sp

low memory

Allocate Space for locals
ldr r3, =FRMADD
sub sp, sp, r3

saved lr	
saved fp	
saved lr	← fp
saved fp	
saved r5	FP_OFF
saved r4	
Local Variables	FRMADD, ← sp

low memory

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
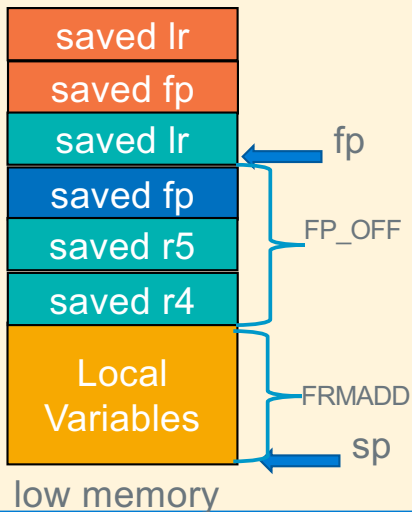Do not push r12 or r13

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

Allocate Space for Local Variables
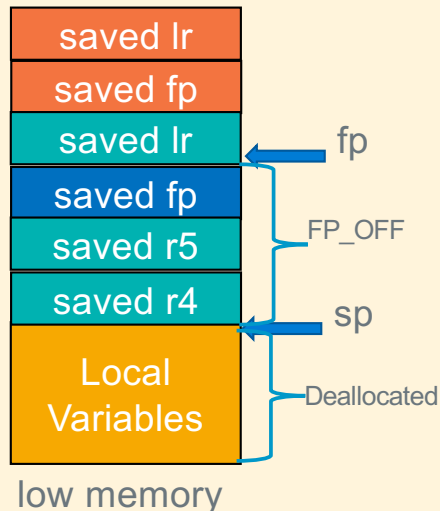
Part of function prologue

x

Function Epilogue: Deallocating the Stack Frame

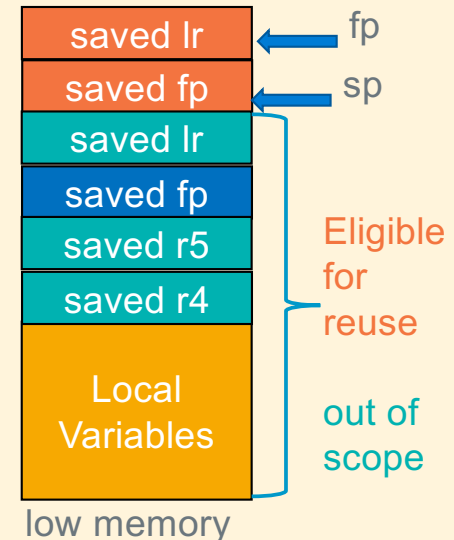Stack frame while during function body execution

saved lr
saved fp
saved lr ← fp
saved fp
saved r5
saved r4
Local Variables

FP_OFF

FRMADD

← sp

low memory

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

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

saved lr
saved fp
saved lr ← fp
saved fp
saved r5
saved r4 ← sp
Local Variables

FP_OFF

Deallocated

low memory

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

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

saved lr ← fp
saved fp ← sp
saved lr
saved fp
saved r5
saved r4
Local Variables

Eligible for reuse

out of scope

low memory

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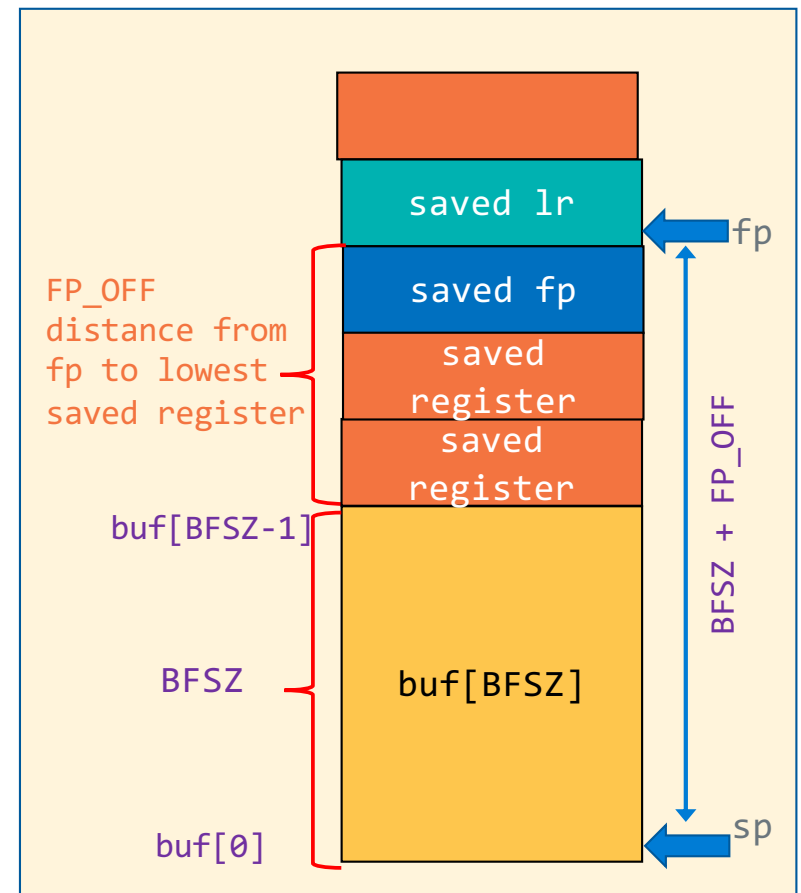
Part of function prologue

x

Local Variables on the Stack

- Local variables are on the stack below the lowest numbered saved register

- frame pointer is used as a **pointer** to stack variables

- fp is the base register in ldr and str instructions
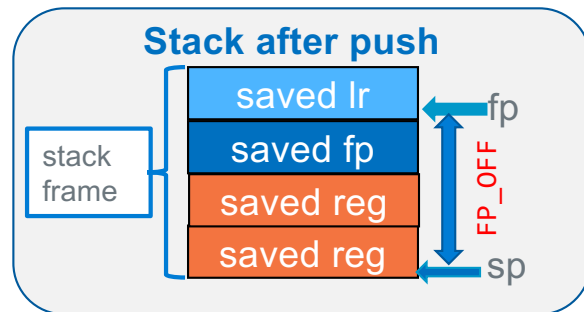
- Example load buf[0] into r4

```
#define BFSZ 4
int main(void)
{
    char buf[BFSZ]; // BFSZ bytes
...
```

- FP_OFF = 12, BUFSZ = 4
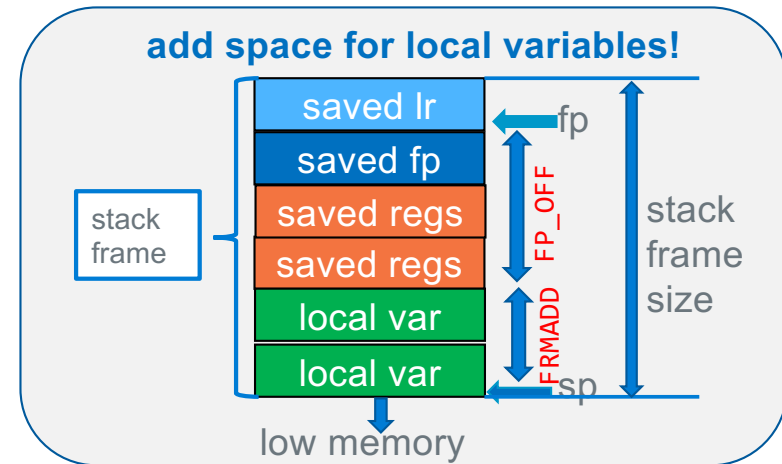- Distance from FP is buf[0] is 12 + 4 = 16

ldrb r4, [fp, -16]

1. Calculate how much additional space is needed by all the local variables

2. **After the register save push, Subtract from the sp** the size of the variable in bytes (+ padding - later slides)

FP_OFF
distance from
fp to lowest
saved register

saved lr

saved fp

saved register

saved register

buf[BFSZ-1]

BFSZ

buf[BFSZ]

buf[0]

fp

sp

BFSZ + FP_OFF

9

X

Function prologue with local variables

Stack after push

stack frame

saved lr ← fp
saved fp
saved reg
saved reg ← sp

FP_OFF

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

add space for local variables!

stack frame

saved lr ← fp
saved fp
saved regs
saved regs
local var
local var ← sp
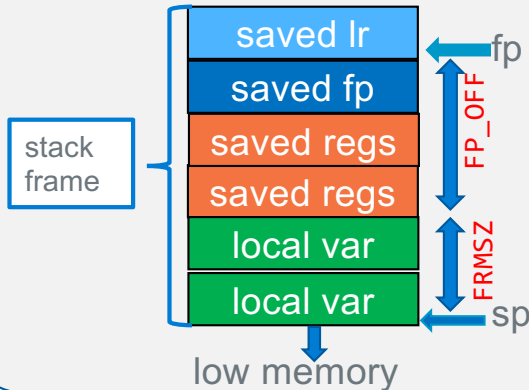
FP_OFF

FRMADD

stack frame size

low memory

- 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
```
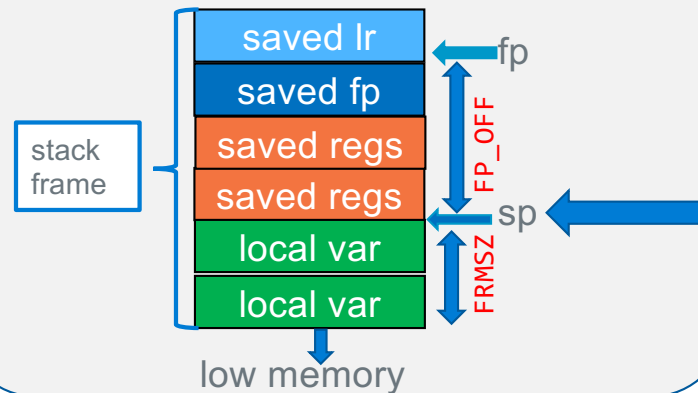
10

X

Function epilogue with local variables

add space for 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

add space for local variables!


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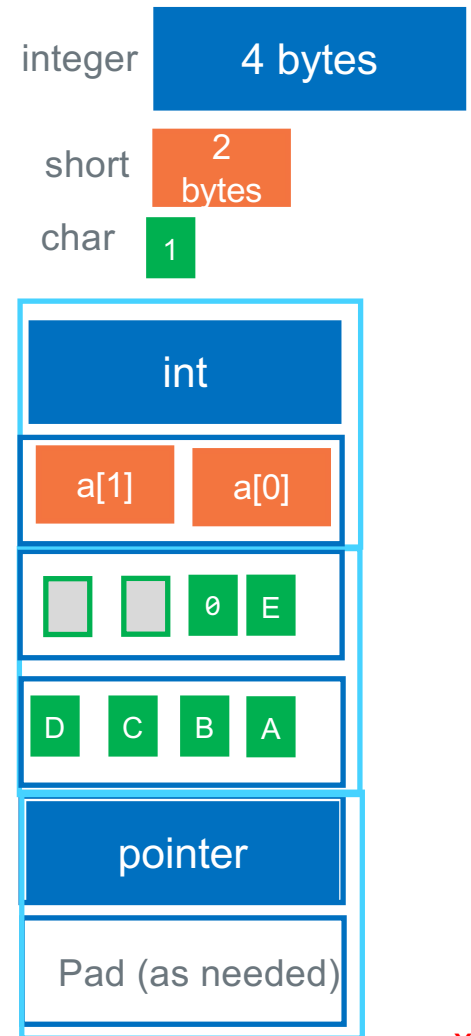
- Return the **sp** (using the **fp**) to the same address it had after the push operation

      sub sp, fp, FP_OFF

- this works no matter how much space was allocated in the prologue

x

Stack Frame Design – Local Variables

- 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

integer | 4 bytes

short | 2 bytes

char | 1

int

a[1] | a[0]

| | | 0 | E |

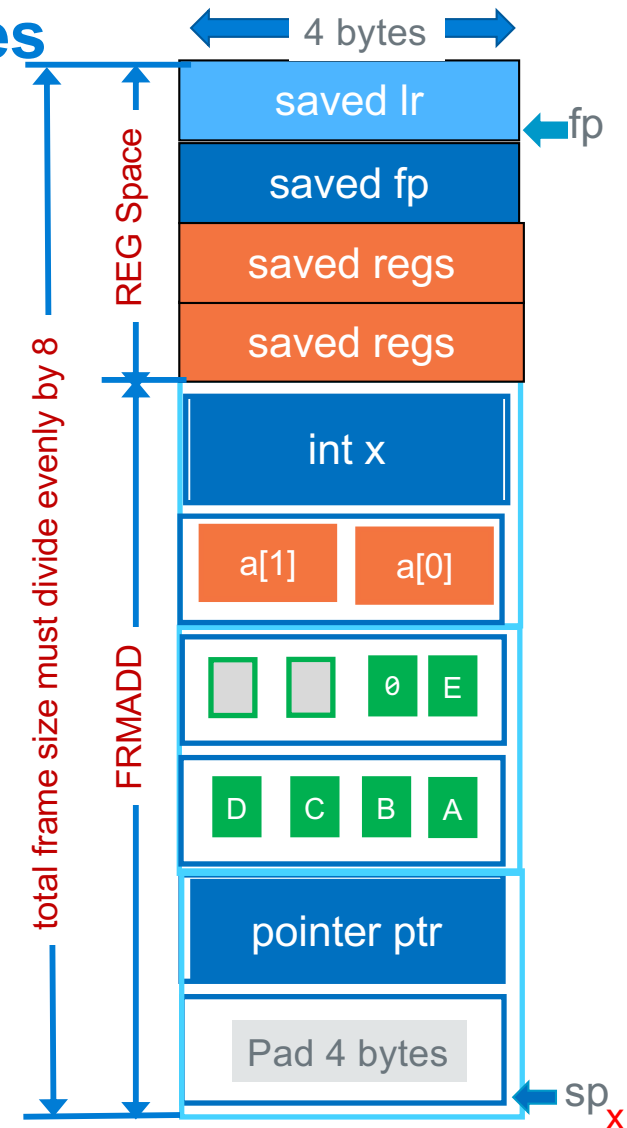| D | C | B | A |

pointer

Pad (as needed)

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

13

4 bytes

saved lr ← fp

saved fp

saved regs

saved regs

REG Space

int x

a[1] a[0]

| | | 0 | E |

| D | C | B | A |

pointer ptr

Pad 4 bytes ← sp

FRMADD

total frame size must divide evenly by 8

x

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]`

Word #
From fp 0
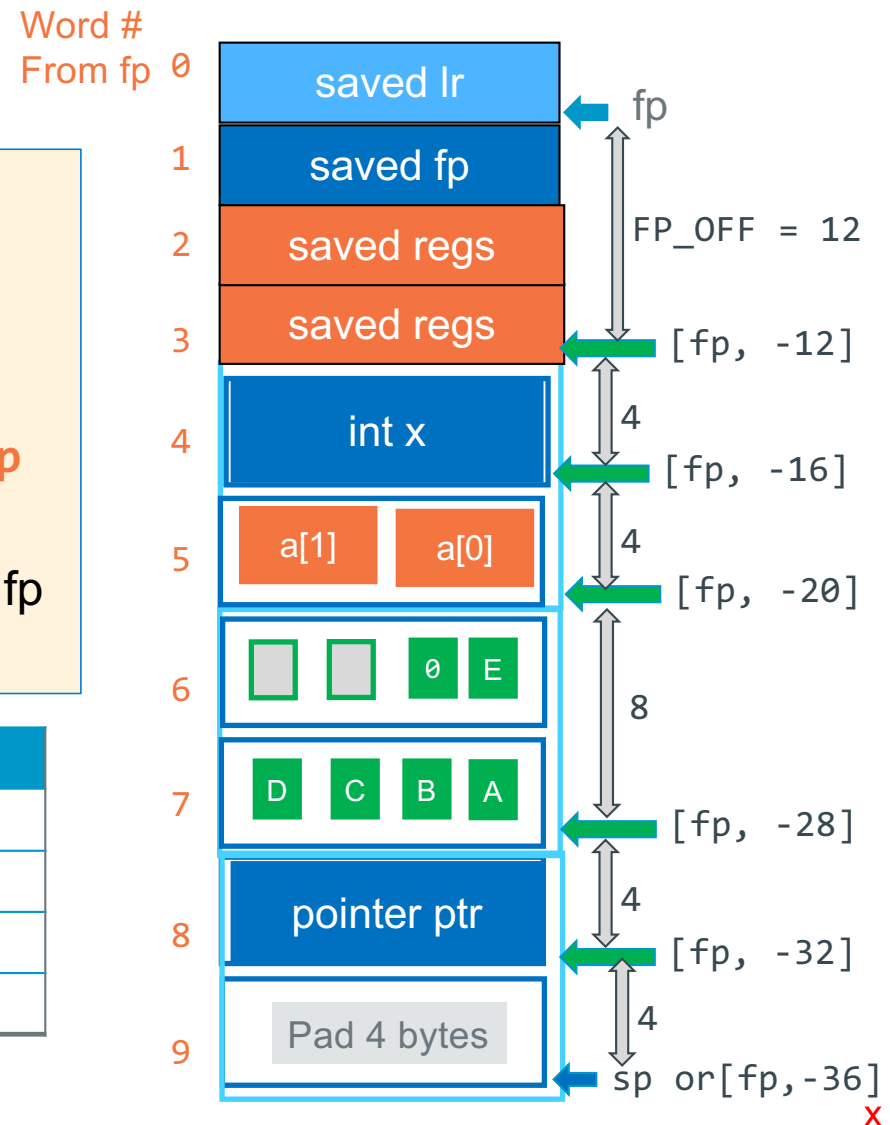
Step 2 Generate Distance offsets from [fp]

- 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

15

X
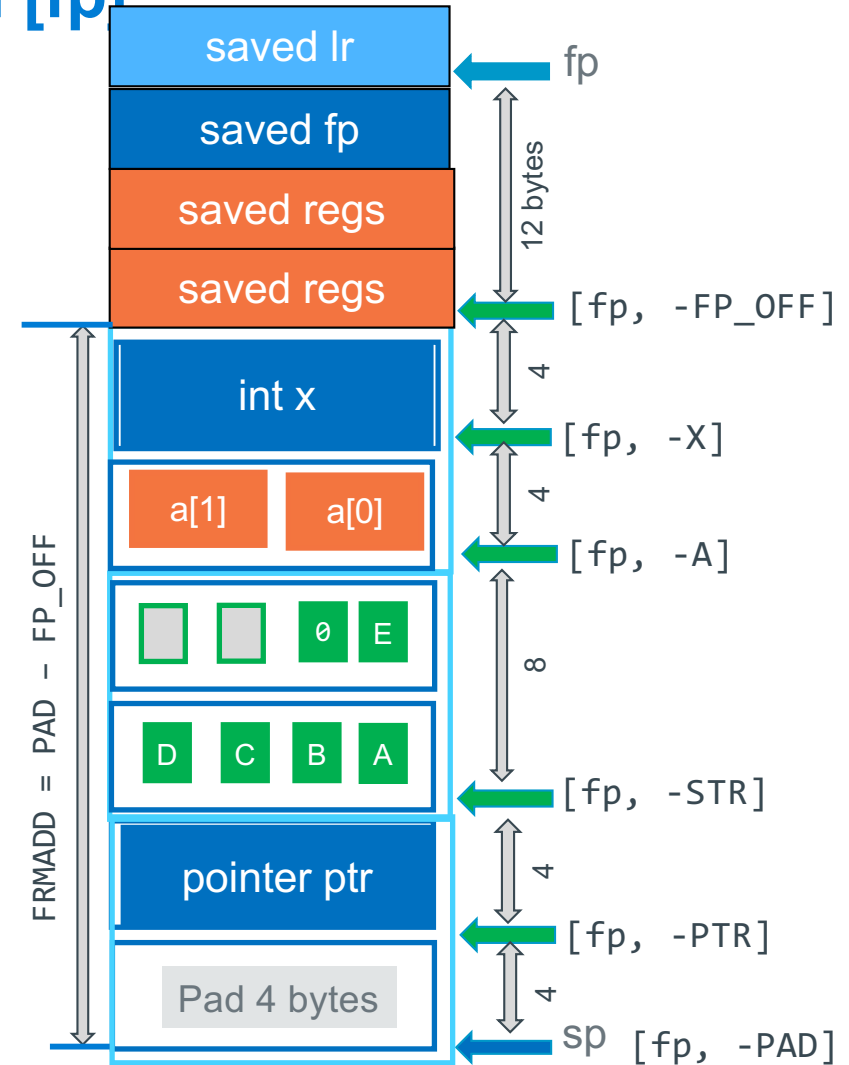
Step 3 Allocate Space in the Prologue

```
        .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)
```

16

saved lr ← fp

saved fp

saved regs

saved regs ← [fp, -FP_OFF]

12 bytes

4

int x ← [fp, -X]

4

a[1] a[0] ← [fp, -A]

0 E

8

D C B A ← [fp, -STR]

pointer ptr

4

← [fp, -PTR]

Pad 4 bytes

4

sp [fp, -PAD]

FRMADD = PAD – FP_OFF

X
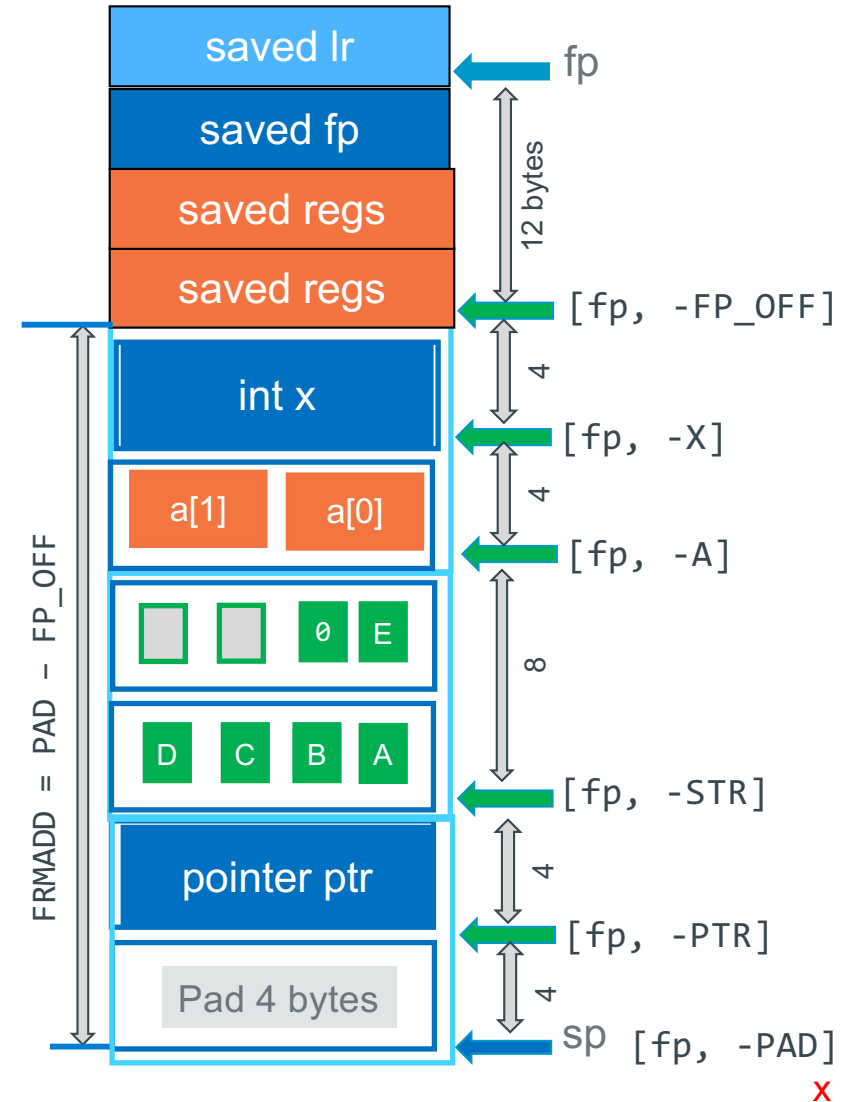
Accessing Stack variables

var	how to get it's 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]

17
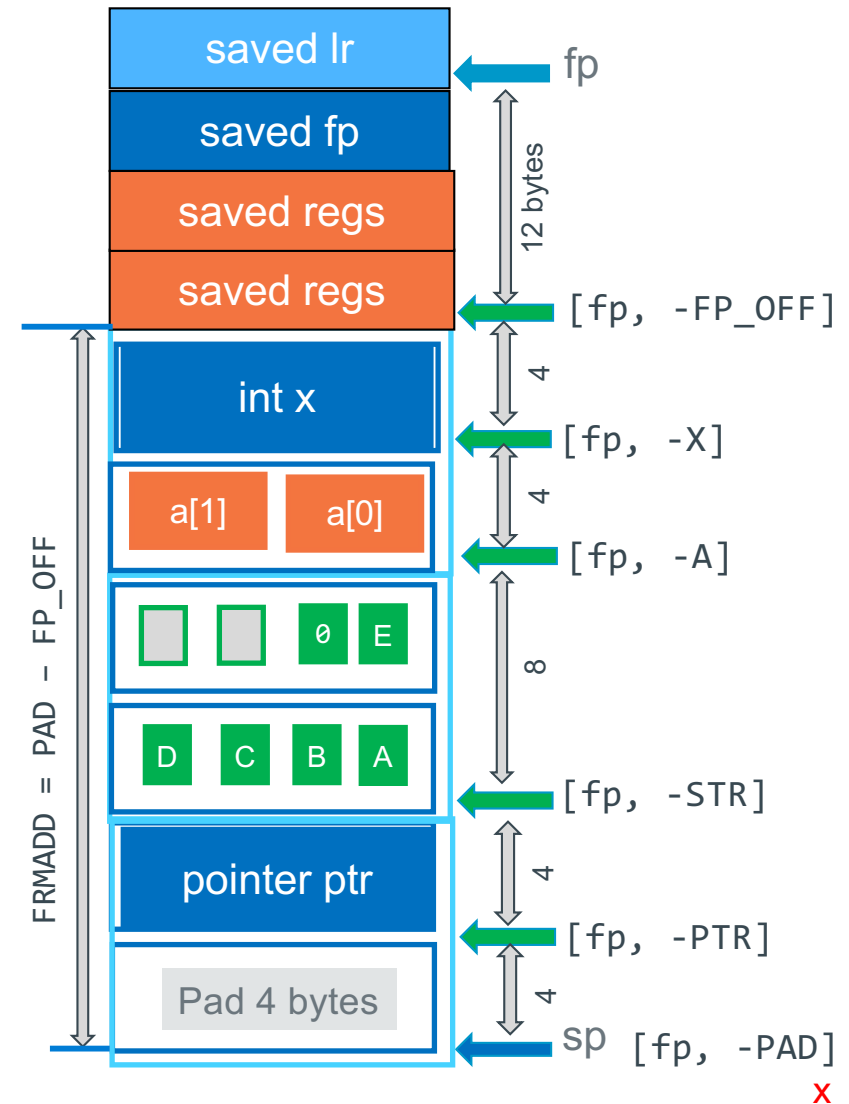
Step 4 Initialize the Local Variables

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

```asm
    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]
//…
```

18

X
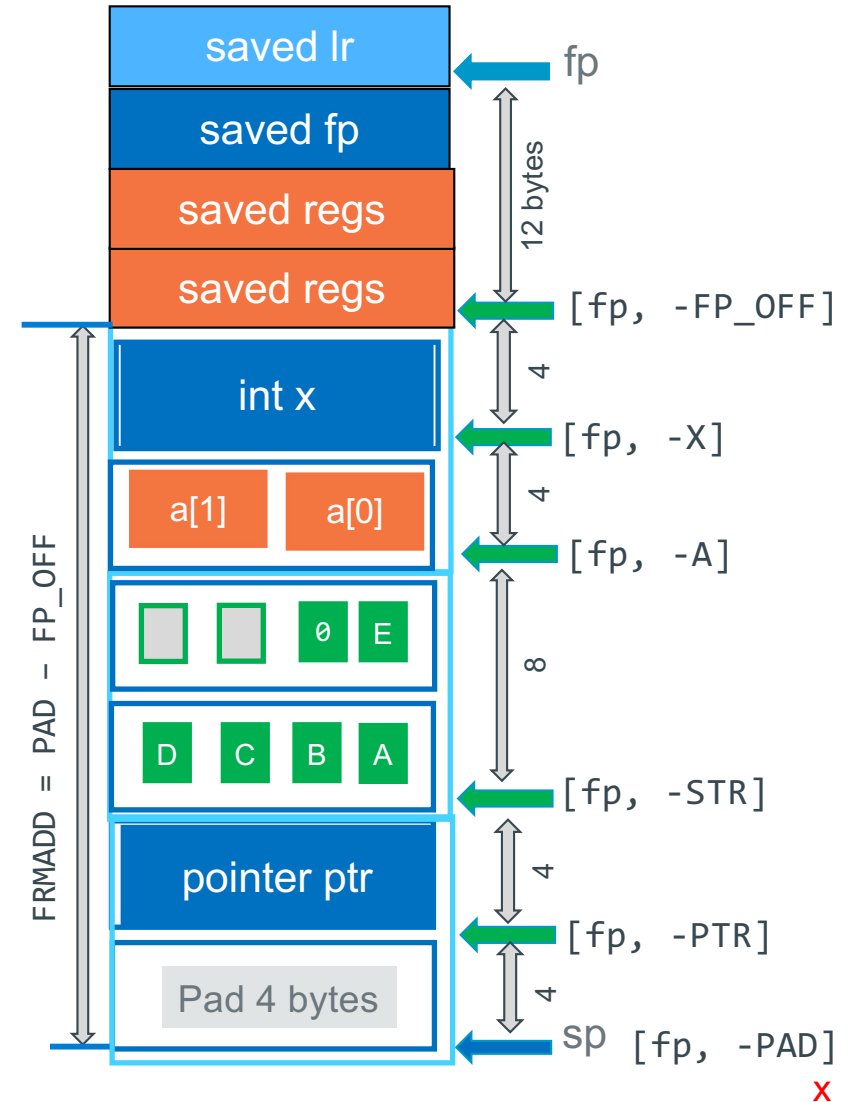
C Stream Functions Array/block read/write

- 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

Passing Pointers to Stack Variables

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BUFSZ 4096
// copies input to output
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;
}
```
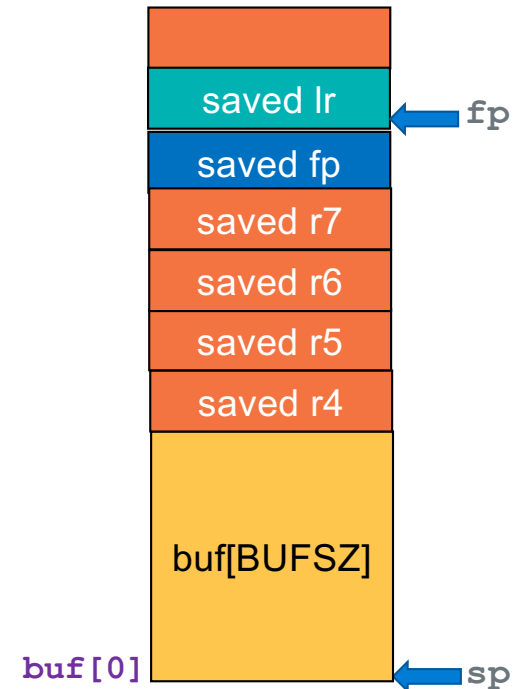
saved lr	← fp
saved fp	
saved r7	
saved r6	
saved r5	
saved r4	
buf[BUFSZ]	
buf[0]	← sp

```asm
        .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]
```


FP_OFF

BUFSZ

BUF=
FP_OFF +
BUFSZ
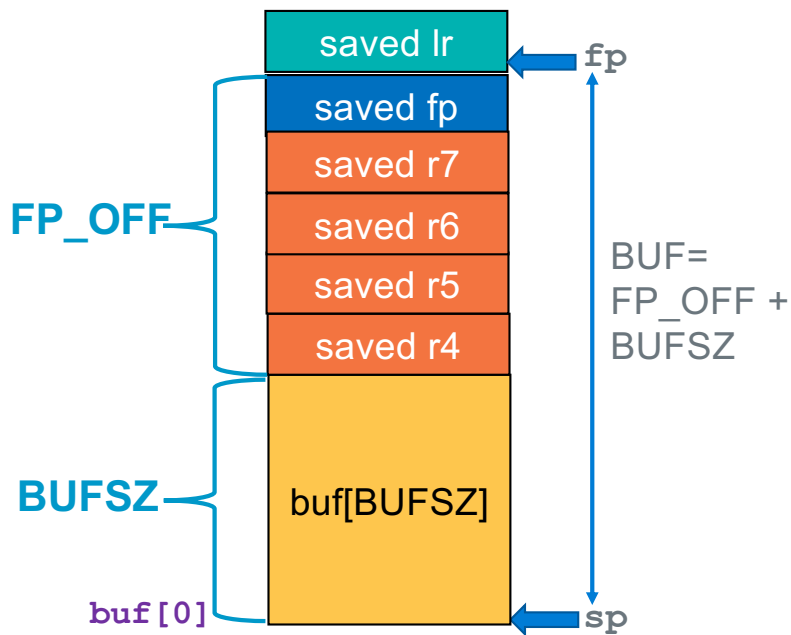
```
        // fread(buffer, element_size, number of elements, FILE *)
        // fread(r0=buf, r1=1, r2=BUFSZ, r3=stdin)
        mov     r0, r4               // buf
        mov     r1, 1                // bytes
        mov     r2, BUFSZ            // cnt (or ldr r2, =BUFSZ)
        mov     r3, r5               // 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
```

21

Passing Pointers to Stack Variables

```c
#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;
}
```

```asm
        .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
        .equ    PAD,        0      + BUF
        .equ    FRMADD,     PAD−FP_OFF

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

        .size   main, (. − main)
```

```asm
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
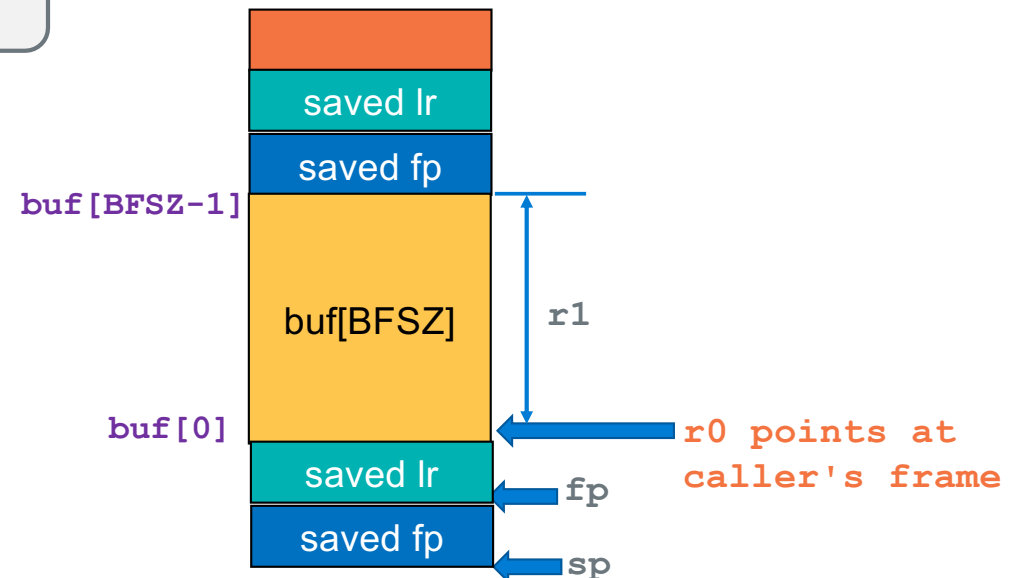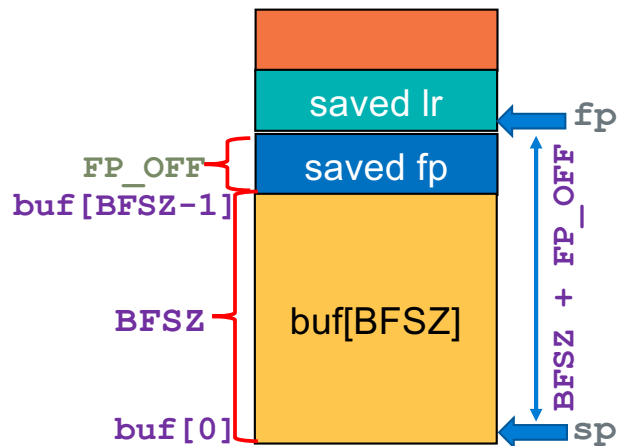
Writing Function: Receiving a Pointer Parameter - 2

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

Using r1 for endptr
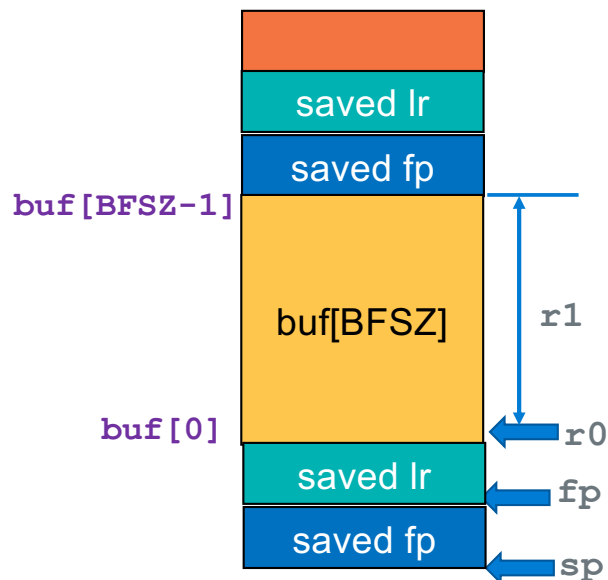
```
fillbuf:
    push    {fp, lr}        // stack frame
    add     fp, sp, FP_OFF  // set fp to base

    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
```

buf[BFSZ-1]

buf[BFSZ]   r1

buf[0]   r0

saved lr   fp

saved fp   sp

saved lr

saved fp

x

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 <u>caller's stack frame</u> 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

- **<u>One arg value per slot</u>!** – 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

25

Stack segment high memory

saved lr	← fp
saved fp	
saved regs	
local variables	
PAD (if needed)	
arg n	
….	
arg 6	
arg 5	← sp

calling functions stack frame

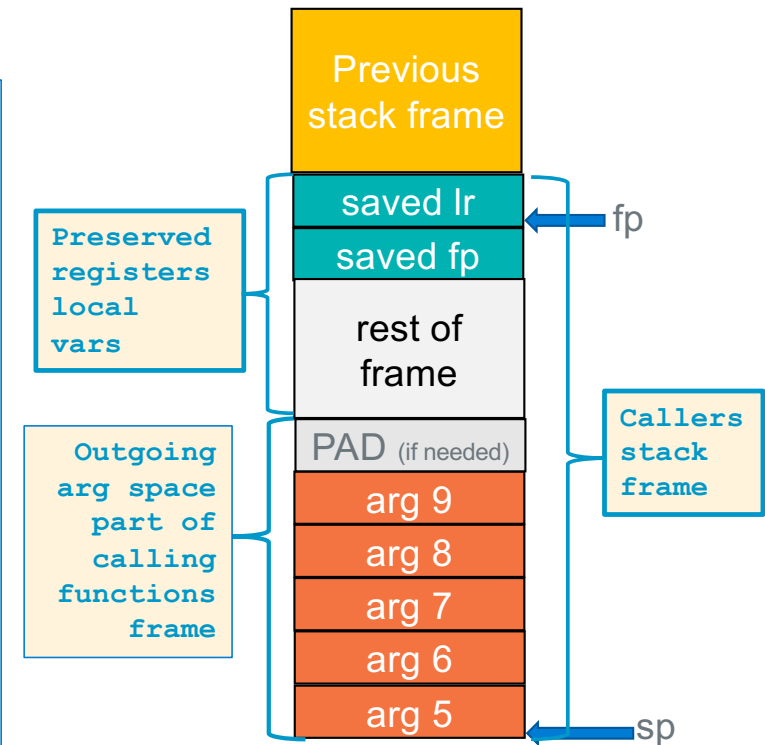| r3 | arg 4 |
r2	arg 3
r1	arg 2
r0	arg 1 & return

Temporary Registers

x
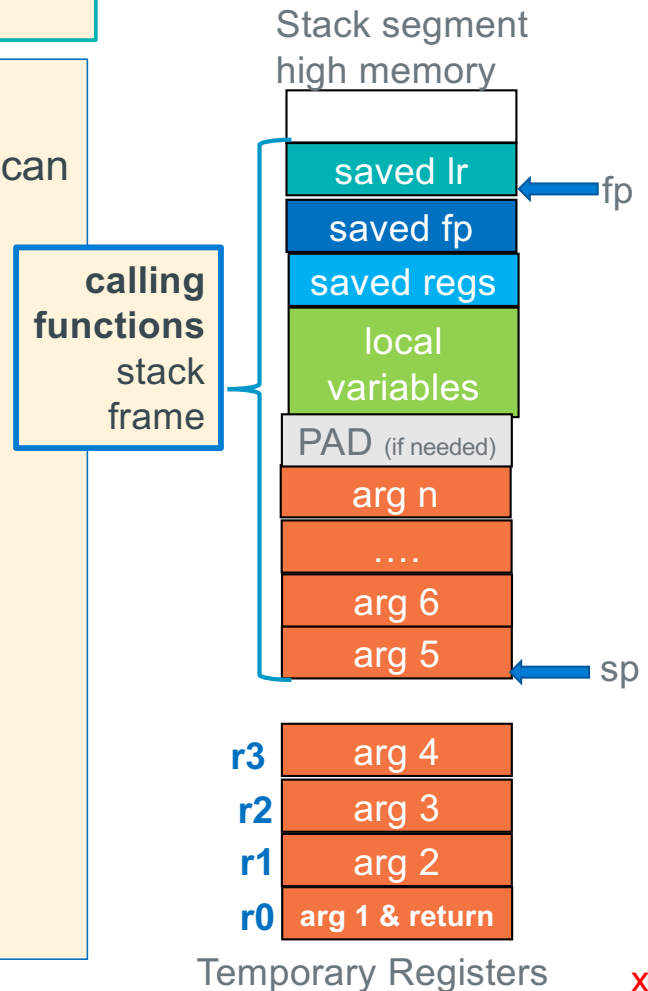
Calling Function: Allocating Stack Parameter Space

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

Previous stack frame

Preserved registers local vars

| saved lr |
| saved fp |

fp

| rest of frame |

Callers stack frame

Outgoing arg space part of calling functions frame

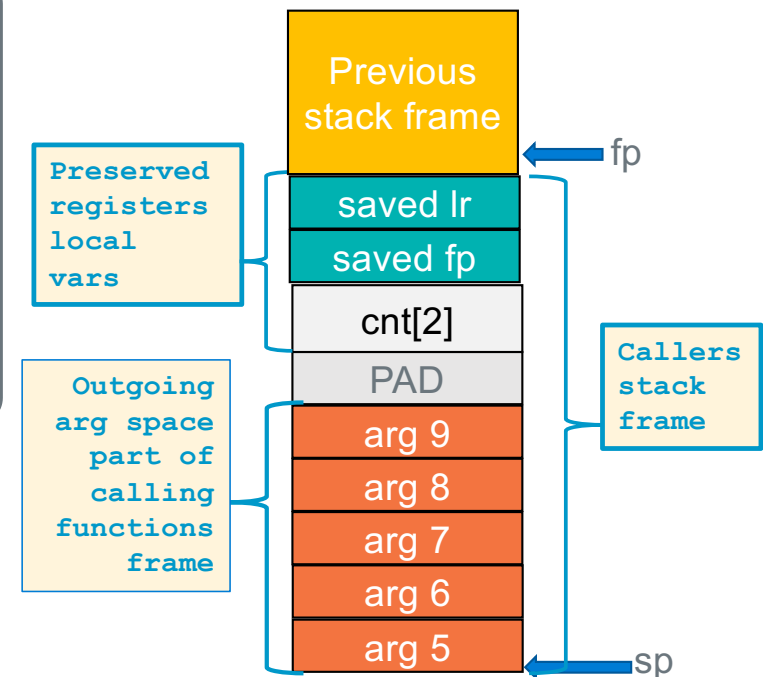| PAD (if needed) |
| arg 9 |
| arg 8 |
| arg 7 |
| arg 6 |
| arg 5 |

sp

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]`

Preserved registers local vars

- Previous stack frame — fp
- saved lr
- saved fp
- cnt[2]
- PAD

Outgoing arg space part of calling functions frame

- arg 9
- arg 8
- arg 7
- arg 6
- arg 5 — sp

Callers stack frame

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]

```
.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

saved lr
saved fp
rest of frame
PAD
arg9 fp+20
arg8 fp+16
0's   arg 7 fp+12
arg6 fp+8
0's   arg5 fp+4
lr to caller fp
callers fp
Saved Registers
Local variables sp
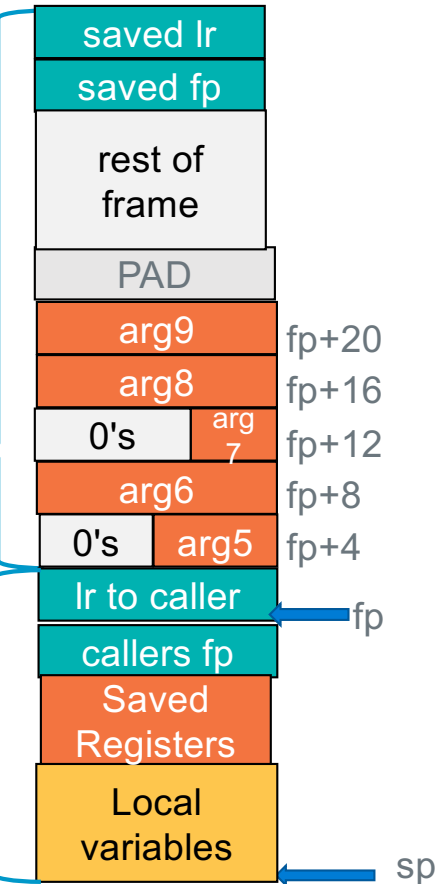
Rule: Called functions always access stack parameters using a positive offset to the fp

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

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

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

```c
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
```

Callers stack frame

saved lr	← fp
saved fp
saved r5
saved r4
cnt
pad
oarg6
oarg5	← sp

X
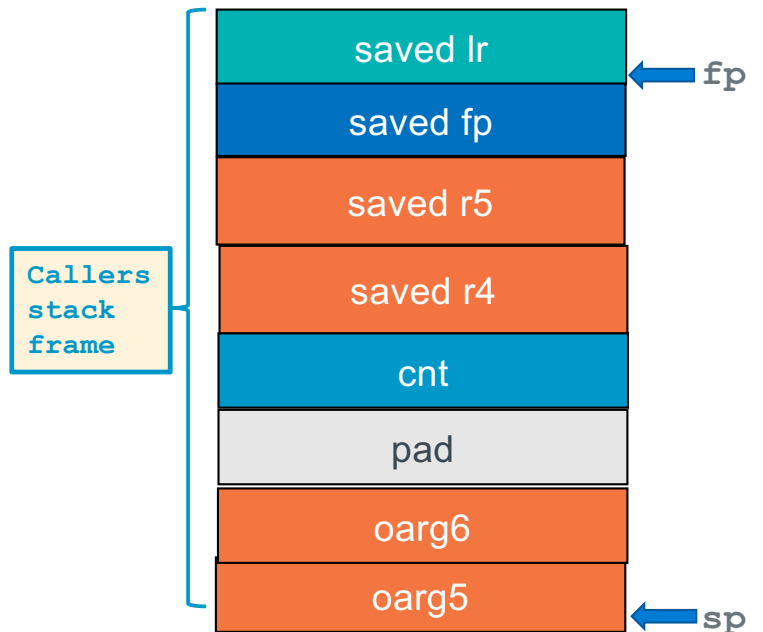
Calling Function > 4 Args - 2

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

```
.equ    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
```

saved lr	← **fp**
saved fp	
saved r5	
saved r4	
cnt	fp - CNT
pad	fp - PAD
oarg6	fp - OARG6
oarg5	fp - OARG5 ← **sp**

Callers stack frame

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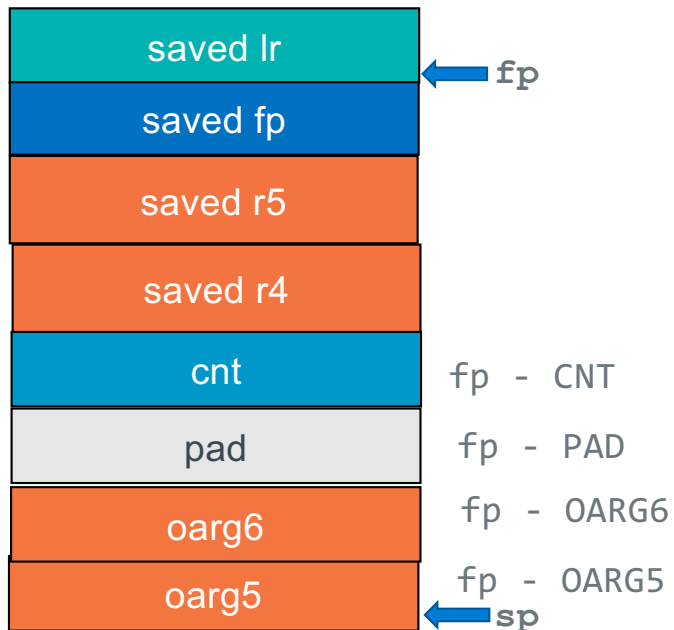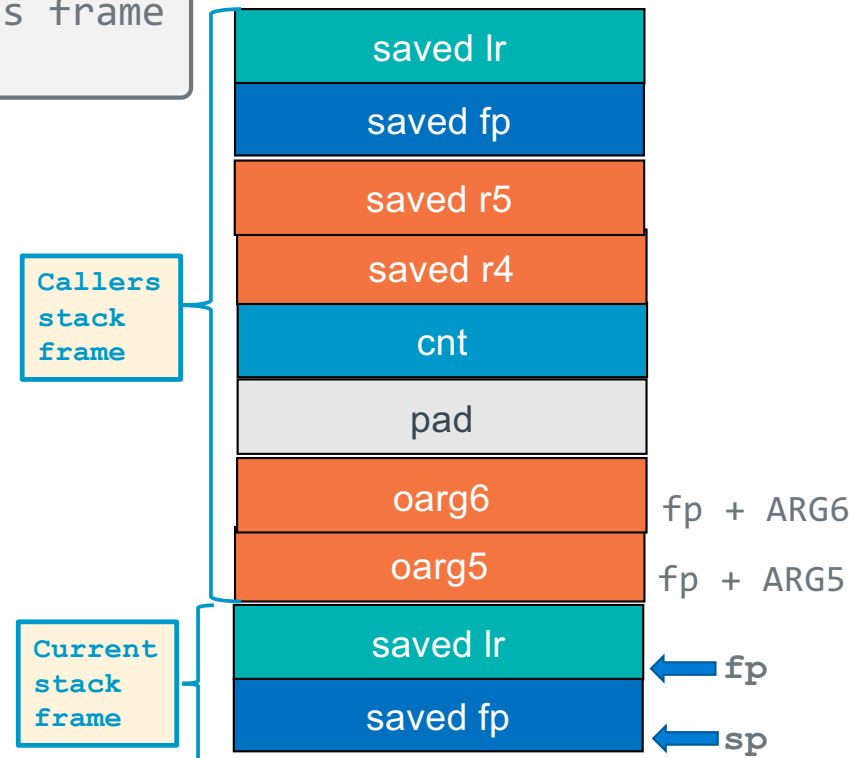
```
        .section .rodata
.Lpfstr: .string "the sum is %d\n"
```

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    ARG6,        8 // offset into caller's frame
.equ    ARG5,        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, ARG5]
        add     r0, r0, r1
        ldr     r1, [fp, ARG6]
        add     r0, r0, r1
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
```

Callers stack frame	saved lr
	saved fp
	saved r5
	saved r4
	cnt
	pad
	oarg6	fp + ARG6
	oarg5	fp + ARG5

| Current stack frame | saved lr | ← fp
| | saved fp | ← sp

X

Accessing argv from Assembly

```
    .extern printf
    .extern stderr
    .section .rodata
.Lstr:  .string "argv[%d] = %s\n"
    .text
    .global main      // main(r0=argc, r1=argv)
    .type    main, %function
    .equ     FP_OFF,      20
main:
    push      {r4-r7, fp, lr}
    add       fp, sp, FP_OFF
    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
```

Registers
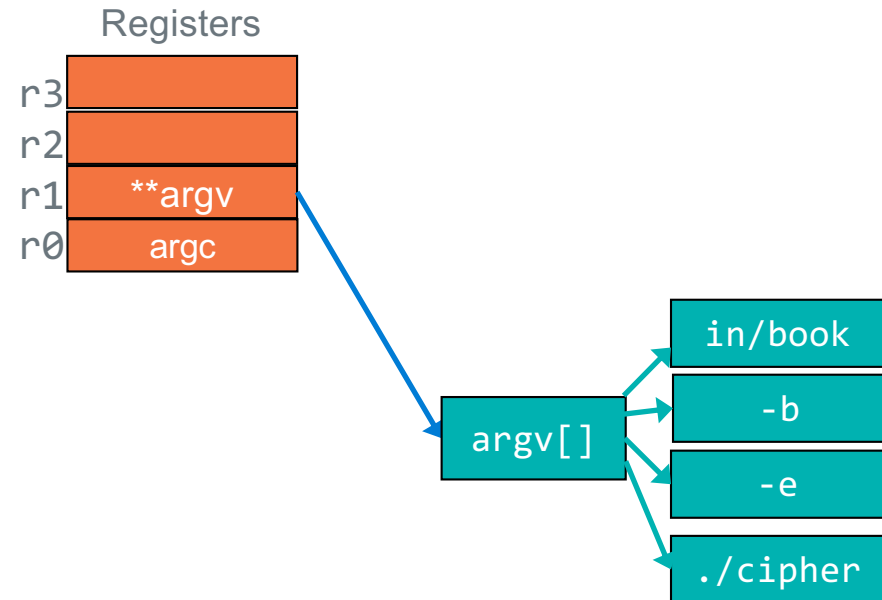

```
 % ./cipher -e -b in/BOOK
argv[0] = ./cipher
argv[1] = -e
argv[2] = -b
argv[3] = in/BOOK
```
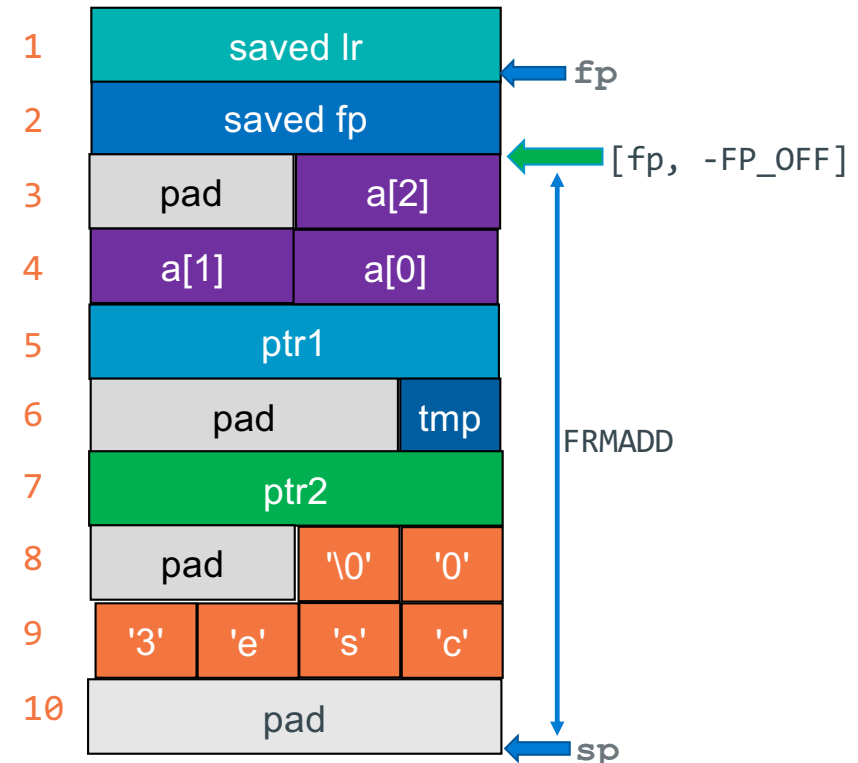
Extra Slides

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)
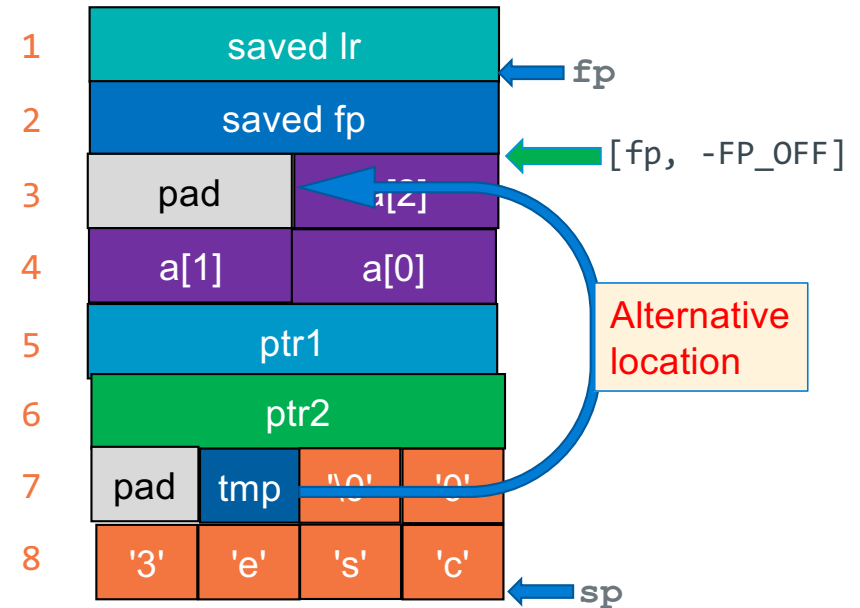
x

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 ➡

1	saved lr	fp
2	saved fp	[fp, -FP_OFF]
3	pad	a[2]
4	a[1]	a[0]
5	ptr1	
6	ptr2	
7	pad	tmp	'\0'	'\0'
8	'3'	'e'	's'	'c'	sp

Alternative location

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)

37

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

38

x

Function Header and Footer Assembler Directives

```
                .text
                .global  myfunc              // make myfunc global for linking
    Function    .type    myfunc, %function  // define myfunc to be a function
    Header      .equ     FP_OFF,  4          // fp offset in main stack frame
          myfunc:

                // function prologue, stack frame setup
                // your code
                // function epilogue, stack frame teardown
    Function    .size myfunc, (. – myfunc)
    Footer
```

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

`.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

- 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

Old file position pointer

read or write N bytes/chars

New file position pointer

file

X
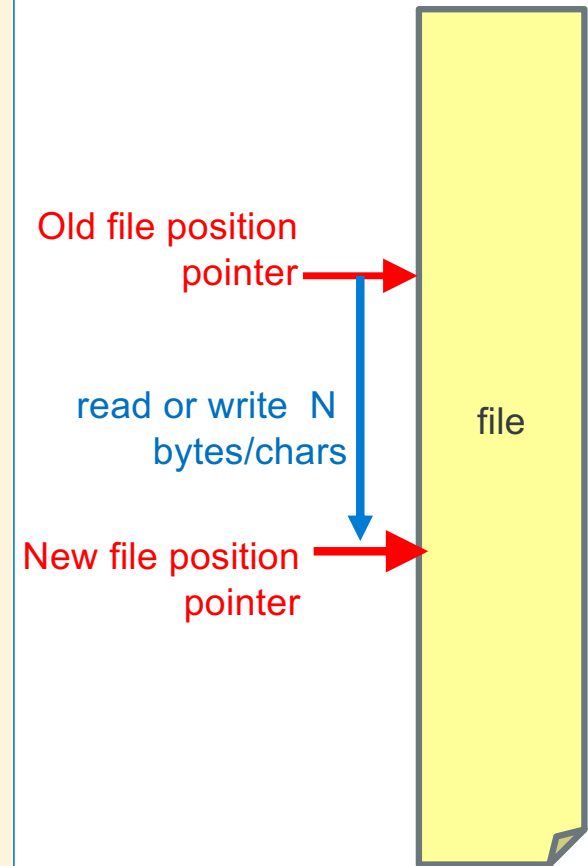
putchar/getchar
Setting up and Usage

```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

```asm
        .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)
```

x

Putchar/getchar:
The while loop

initialize count

pre loop test with a call to getchar()
if it returns EOF in r0 we are done

echo the character read with getchar and
then read another and increment count

did getchar() return EOF if not loop

saw EOF, print count

```
#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;
}
```

```
            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
```

File header and footers are not shown

44

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