Version 2.05

UCSD CSE 30

Computer Organization and Systems Programming

Lecture - 18

Keith Muller

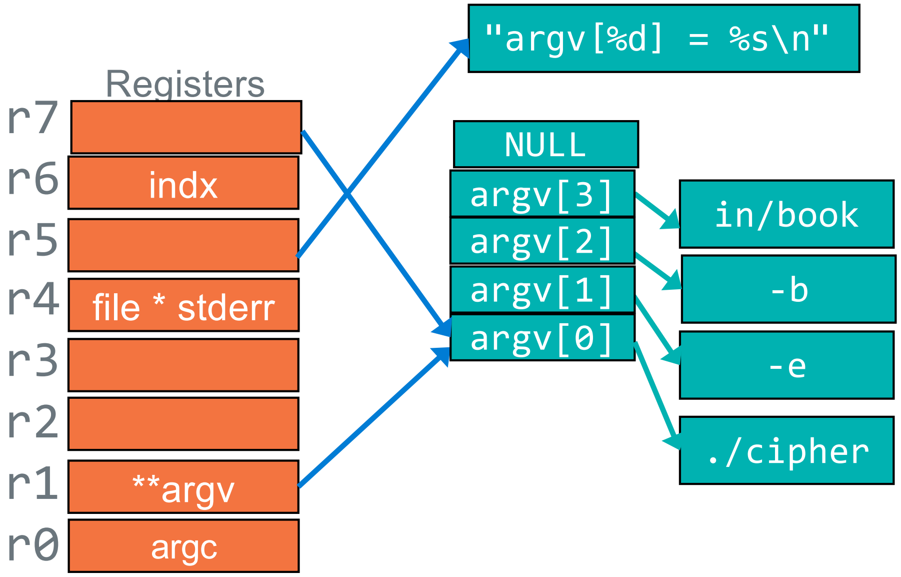
Accessing Pointers (argv) in ARM assembly

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

```
    .extern printf
    .extern stderr
    .section .rodata
.Lstr:  .string "argv[%d] = %s\n"
    .text
    .global main    // main(r0=argc, r1=argv)
    .type   main, %function
    .equ    FP_OFF,     20
main:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF
    mov     r7, r1           // save argv!
    ldr     r4, =stderr      // get the address of stderr
    ldr     r4, [r4]         // get the contents of stderr
    ldr     r5, =.Lstr       // get the address of .Lstr
    mov     r6, 0            // set indx = 0;

// see next slide

.Ldone:
    mov     r0, 0
    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx      lr
```

need to save r1 as we are calling a function - fprintf

r0-r3 lost due to fprintf call



```
fprintf(stderr, "argv[%d] = %s\n", indx, *argv);
```

x

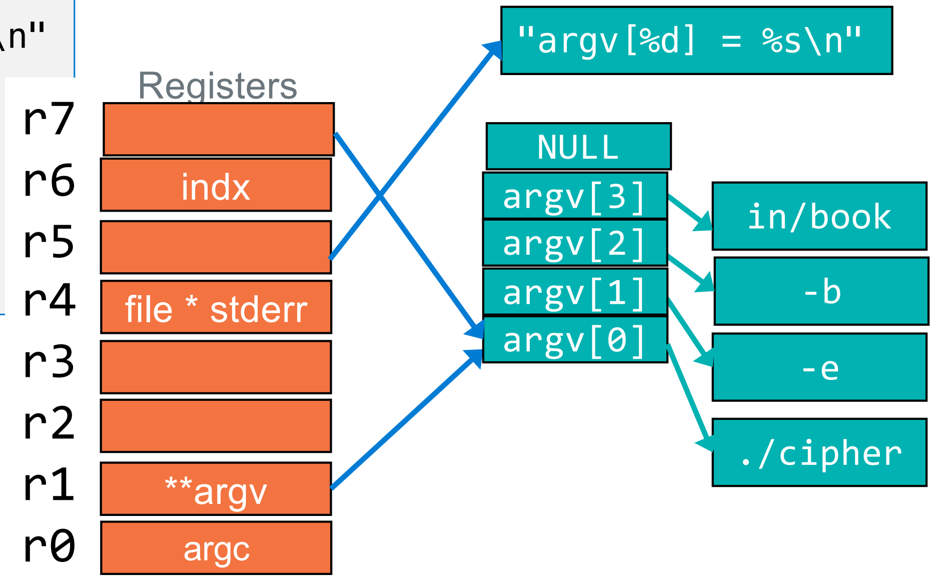
Accessing Pointers (argv) in ARM assembly

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

```
.Lloop:
    // fprintf(stderr, "argv[%d] = %s\n", indx, *argv)
    ldr     r3, [r7]        // arg 4: *argv
    cmp     r3, 0           // check *argv == NULL
    beq     .Ldone          // if so done
    mov     r2, r6          // arg 3: indx
    mov     r1, r5          // arg 2: "argv[%d] = %s\n"
    mov     r0, r4          // arg 1: stderr
    bl      fprintf
    add     r6, r6, 1       // indx++ for printing
    add     r7, r7, 4       // argv++ pointer
    b       .Lloop
.Ldone:
```

observe the different increment sizes

r0-r3 lost due to fprintf call



x

Allocating Space For Locals on the Stack

- Space for local variables is allocated on the stack right below the lowest pushed register
  - Move the sp towards low memory by the total size of all local variables in bytes **plus padding**

    FRMADD = total local var space (bytes) + padding

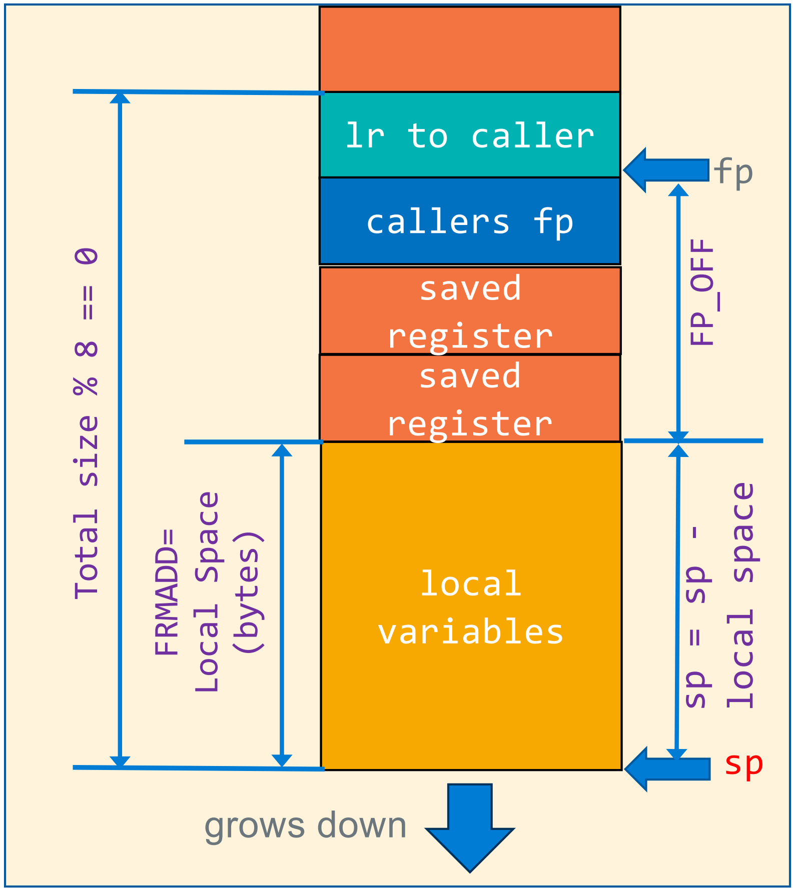
- Allocate the space after the register push by

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

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

  sp % 8 == 0

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

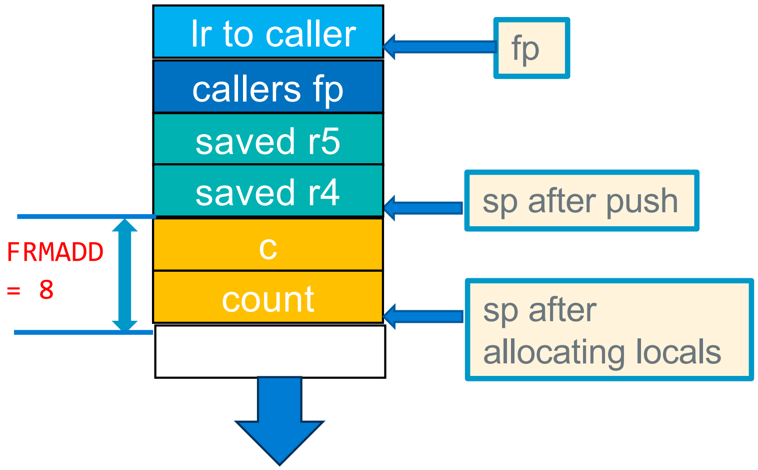


x

Local Variables on the stack

```
int main(void)
{
    int c;
    int count = 0;
    // rest of code
}
```

after push {r4-r5,fp,lr}
add fp, sp, FP_OFF



```
  .text
     .type   main, %function
     .global main
    .equ    FP_OFF,    12
    .equ    FRMADD,     8
main:
     push    {r4, r5, fp, lr}
     add     fp, sp, FP_OFF
     add     sp, sp, -FRMADD
// but we are not done yet!
```

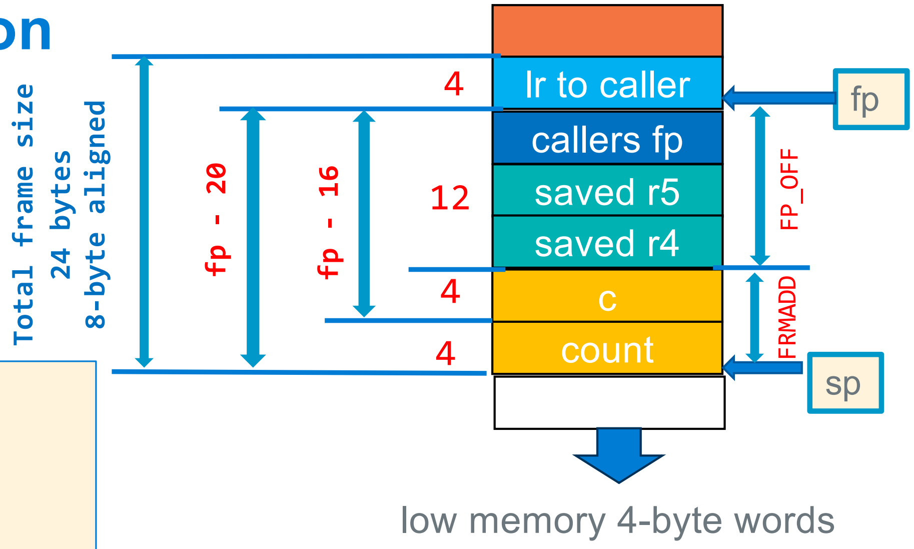
```
// when FRMADD values fail to assemble
        ldr r3, =-FRMADD
        add sp, sp, r3
```

- In this example we are allocating two variables on the stack
- When writing assembly functions, in many situations you may choose allocate these to registers instead

- Add space on the stack for each local
  - we will allocate space in same order the locals are listed the C function shown from high to low stack address
  - gcc compiler allocates from low to high stack addresses
  - Order does not matter for our use

Accessing Stack Variables: Introduction

```
int main(void)
{
    int c;
    int count = 0;
    // rest of code
}
```



- To Access data stored in the stack
  - use the `ldr/str` instructions
- **Use register `fp` with offset (distance in bytes) addressing** (use either register offset or immediate offset)
- ***No matter what address the stack frame is at***, `fp` always points at saved `lr`, so you can find a local stack variable by using an offset address from the contents of `fp`

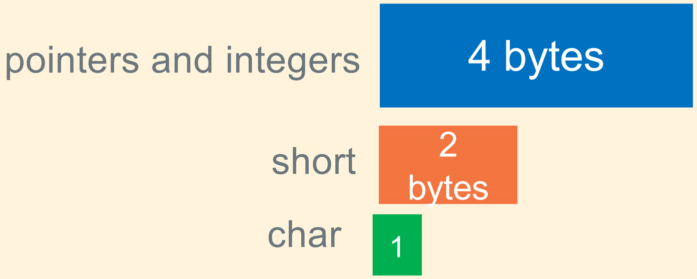
```
 .text
    .type   main, %function
    .global main
   .equ    FP_OFF,    12
   .equ    FRMADD,     8
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
// but we are not done yet!
```

Variable	distance from fp	Read variable	Write Variable
`int c`	-16	`ldr r0, [fp, -16]`	`str r0, [fp, -16]`
`int count`	-20	`ldr r0, [fp, -20]`	`str r0, [fp, -20]`

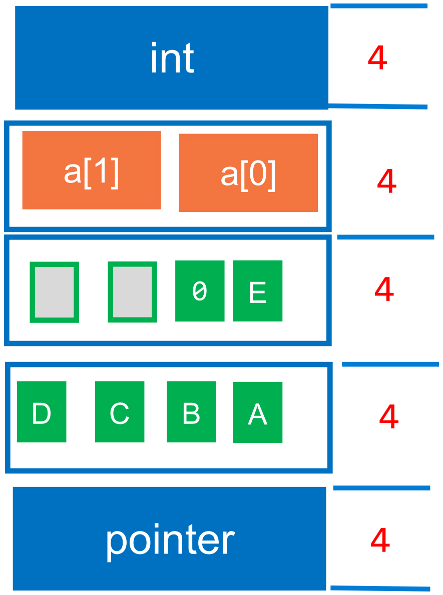
x

Stack Frame Design – Local Variables

- When writing an ARM equivalent for a C program, for CSE30 we will not re-arrange the order of the variables to optimize space (covered in the compiler course)
- 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
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- Padding may be required (see next slide)



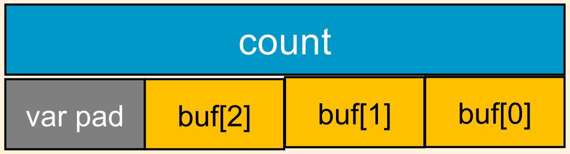
Rule: When the function is entered the stack is already 8-byte aligned



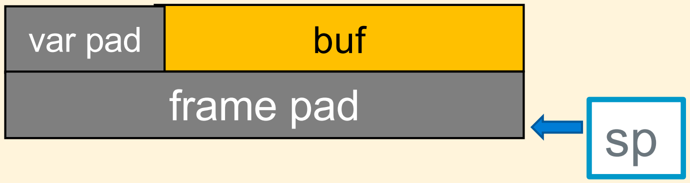
x

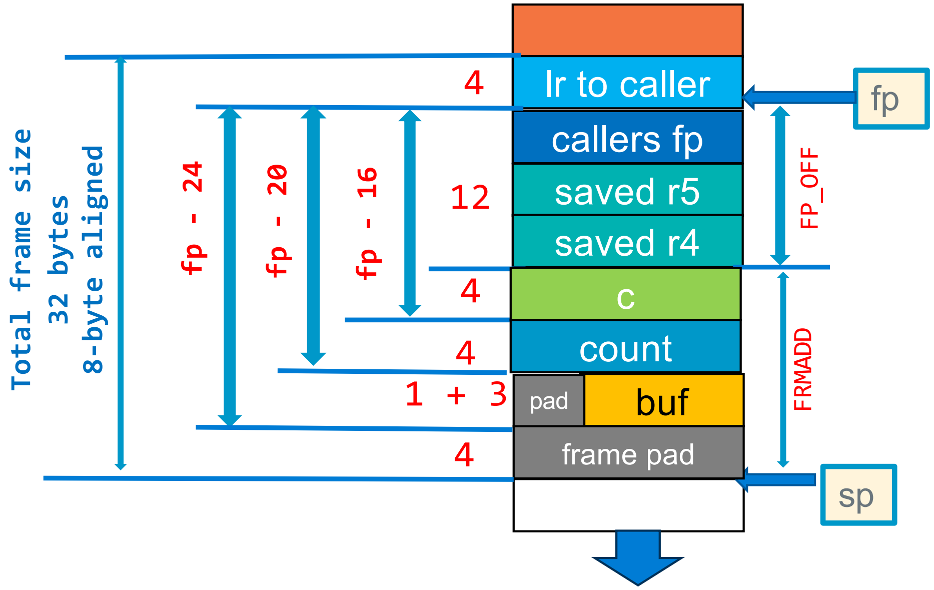
Stack Variables: Padding

- **Variable padding** – start arrays at 4-byte boundary and **leave unused space at end** (high side address) before the variable higher on the stack



- **Frame padding** – **add space below the last local variable** to keep 8-byte alignment





```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

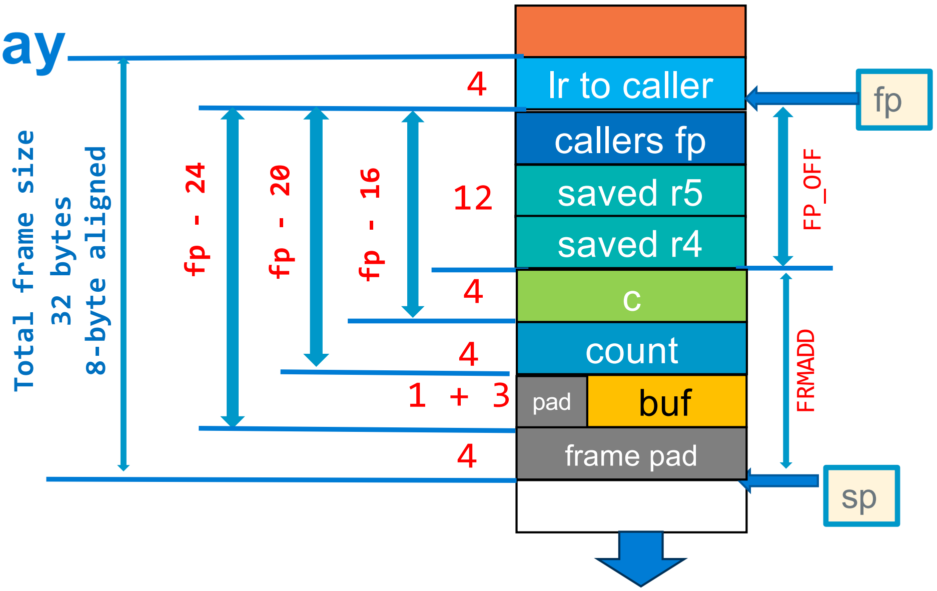
```
  .text
     .type   main, %function
     .global main
   .equ    FP_OFF,    12
   .equ    FRMADD,    16
main:
     push    {r4, r5, fp, lr}
     add     fp, sp, FP_OFF
     add     sp, sp, -FRMADD
// but we are not done yet!
```

x

Accessing Stack Variables, the hard way

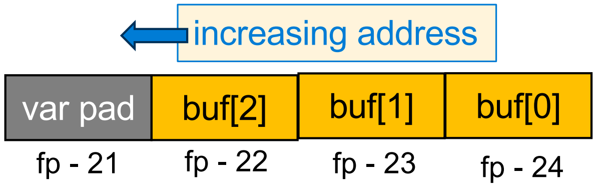
```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
 .text
    .type   main, %function
    .global main
   .equ    FP_OFF,    12
   .equ    FRMADD,    16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
// but we are not done yet!
```



char buf[ ] by usage with ASCII chars we will use strb (or make it unsigned char)

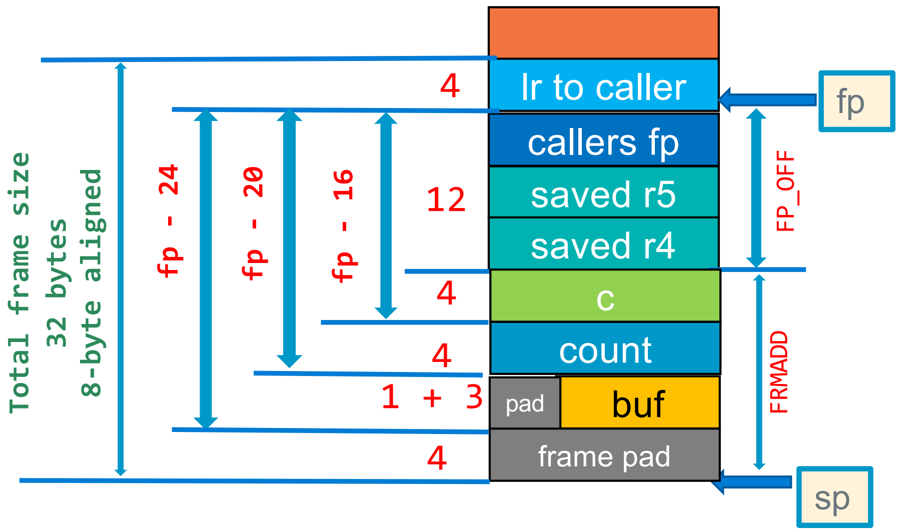
Variable	distance from fp	Read variable	Write Variable
`int c`	16	`ldr r0, [fp, -16]`	`str r0, [fp, -16]`
`int count`	20	`ldr r0, [fp, -20]`	`str r0, [fp, -20]`
`char buf[0]`	24	`ldrb r0, [fp, -24]`	`strb r0, [fp, -24]`
`char buf[1]`	23	`ldrb r0, [fp, -23]`	`strb r0, [fp, -23]`
`char buf[2]`	22	`ldrb r0, [fp, -22]`	`strb r0, [fp, -22]`



- **Calculating offsets is a lot of work to get it correct**
- It is also hard to debug
- There is a better way!

x

Best Practice: Assembler Generated FP Distance Table



FP Distance Table one For each function

```
.type   main, %function
.global main

.equ    FP_OFF,    12

.equ    C,         4 + FP_OFF
.equ    COUNT,     4 + C
.equ    BUF,       4 + COUNT
.equ    PAD,       4 + BUF
.equ    FRMADD,    PAD – FP_OFF
// FRMADD =  28 - 12 = 16
```

pushed reg fp distance

variable size in bytes

Prior allocation distance

1. For each stack variable create a .equ symbol whose value is the distance in bytes from the FP after the prologue push
2. After the last variable add a name PAD for the size of the frame padding (if any) if no padding, PAD will be set to the same value as the variable above it
3. The value of the symbol is an expression that calculates the distance from the FP based on the distance of the variable above it on the stack. The first variable will use SP_OFF as the starting distance

   .equ VAR, size_of var + variable_padding + previous_var_symbol *// previous_var_symbol distance of the var above*
4. Calculate the size of the local variable area that needs to be added to the sp in bytes

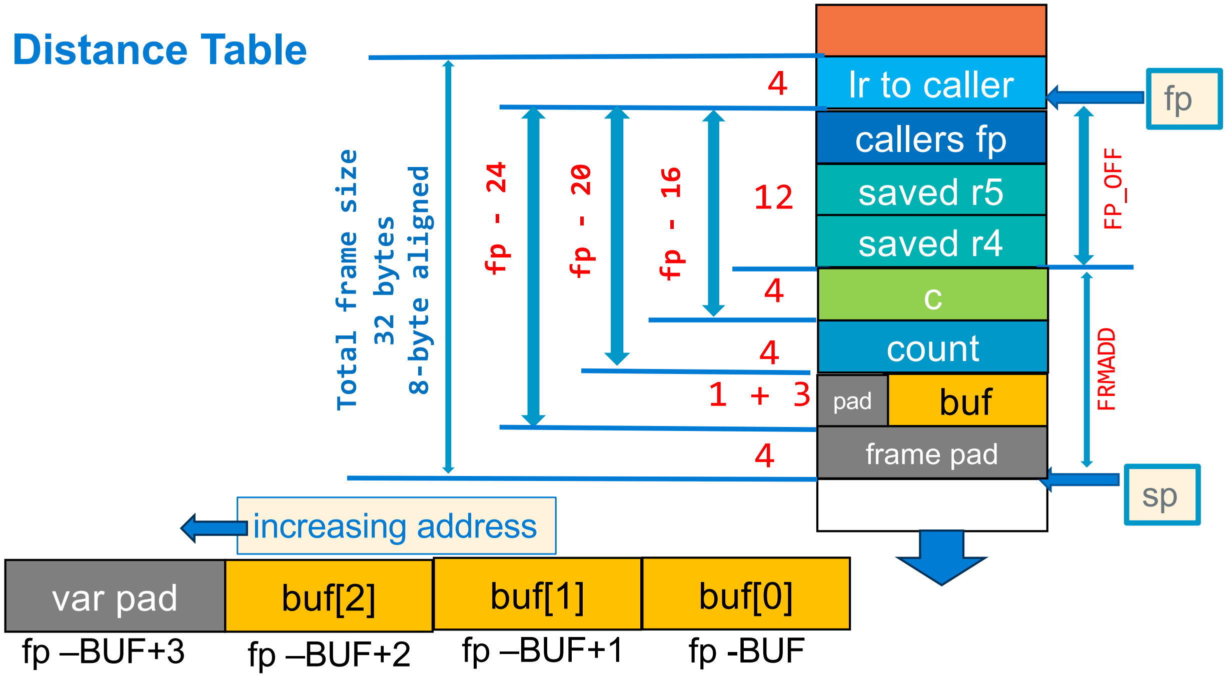
   FRMADD = distance PAD minus distance of the SP to the FP (FP_OFF) after the prologue push

X

Best Practice: Assembler Generated FP Distance Table

FP Distance Table For each function

```
.type   main, %function
.global main
.equ    FP_OFF,     12
.equ    C,          4 + FP_OFF
.equ    COUNT,      4 + C
.equ    BUF,        4 + COUNT
.equ    PAD,        4 + BUF
.equ    FRMADD,     PAD – FP_OFF
// FRMADD =   28 - 12 = 16
```



Variable	distance from fp	Address on Stack	Read variable	Write Variable
`int c`	C	`add r0, fp, -C`	`ldr r0, [fp, -C]`	`str r0, [fp, -C]`
`int count`	COUNT	`add r0, fp, -COUNT`	`ldr r0, [fp, -COUNT]`	`str r0, [fp, -COUNT]`
`char buf[0]`	BUF	`add r0, fp, -BUF`	`ldrb r0, [fp, -BUF]`	`strb r0, [fp, -BUF]`
`char buf[1]`	BUF-1	`add r0, fp, -BUF+1`	`ldrb r0, [fp, -BUF+1]`	`strb r0, [fp, -BUF+1]`
`char buf[2]`	BUF-2	`add r0, fp, -BUF+2`	`ldrb r0, [fp, -BUF+2]`	`strb r0, [fp, -BUF+2]`

x

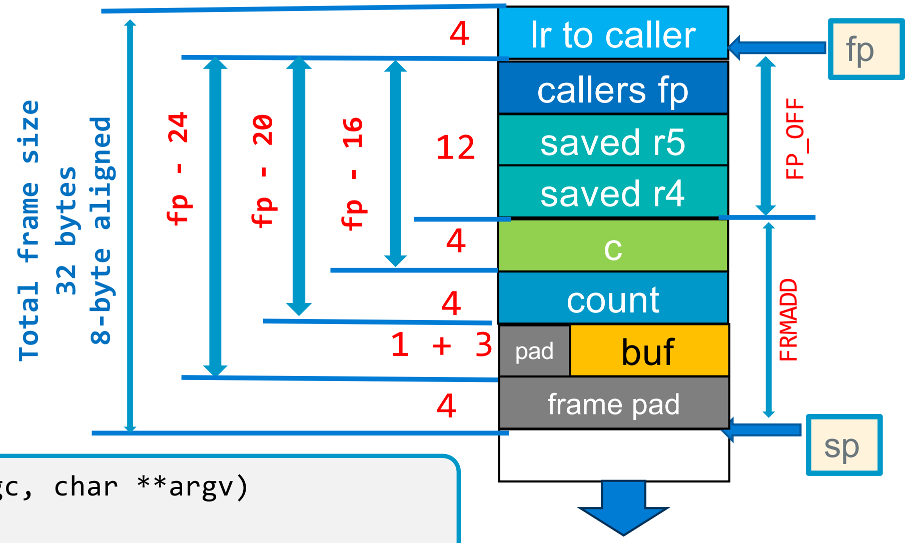
Initializing and Accessing Stack variables

```
    .section .rodata
.Lmess: .string "%d %d %s\n"
    .extern printf
```

```
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // nothing to do for C
    mov     r2, 0
    str     r2, [fp, -COUNT]
    strb    r2, [fp, -BUF+2]
    mov     r2, 'h'
    strb    r2, [fp, -BUF]
    mov     r2, 'i'
    strb    r2, [fp, -BUF+1]

    ldr     r0, =.Lmess       // arg1
    ldr     r1, [fp, -C]      // arg2
    ldr     r2, [fp, -COUNT]  // arg3
    add     r3, fp, -BUF      // arg4
    bl      printf
```

passes contents of stack var C and COUNT

passes address of a stack variable buf



```
int main(int argc, char **argv)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    printf("%d %d %s\n", c, count, buf);
    // rest of code
```

pass stack address

```
./a.out
-136572160 0 hi
```

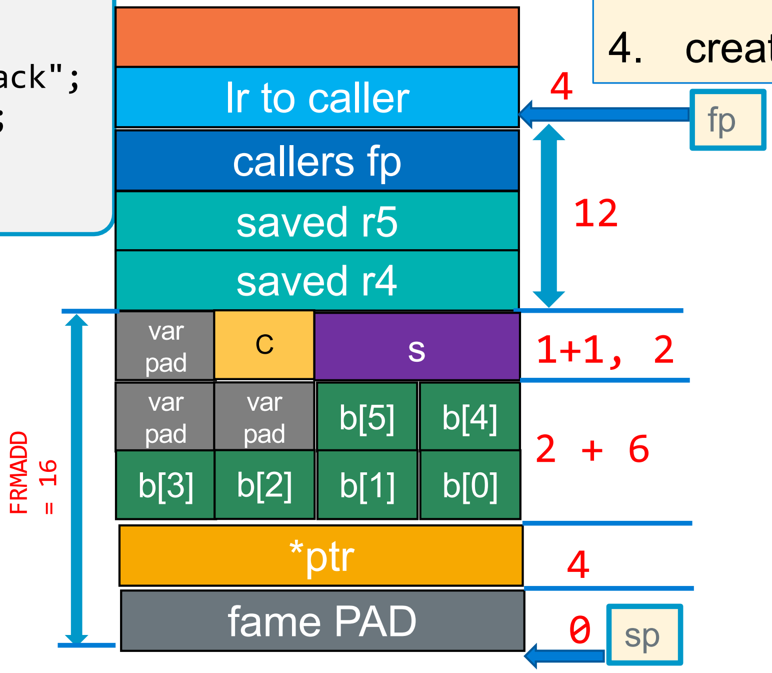
Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add r0, fp, -BUF+1	ldrb r0, [fp, -BUF+1]	strb r0, [fp, -BUF+1]
char buf[2]	BUF-2	add r0, fp, -BUF+2	ldrb r0, [fp, -BUF+2]	strb r0, [fp, -BUF+2]

X

Stack Frame Design Practice

```c
void func(void)
{
  signed char c;
  signed short s;
  unsigned char b[] = "Stack";
  unsigned char *ptr = &b;
    // rest of code
}
```

1. Write the variables in C
2. Draw a picture of the stack frame
3. Write the code to generate the offsets
4. create the distance table to the variables



```
.equ    FP_OFF,     12
.equ    C,          2 + FP_OFF
.equ    S,          2 + C
.equ    B,          8 + S
.equ    PTR,        4 + BUF
.equ    PAD,        0 + PTR
.equ    FRMADD,     PAD – FP_OFF
// FRMADD =  28 - 12 = 16
```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
`signed char c`	C	`add r0, fp, -C`	`ldrsb r0, [fp, -C]`	`strsb r0, [fp, -C]`
`signed short s`	S	`add r0, fp, -S`	`ldrsh r0, [fp, -S]`	`strsh r0, [fp, -S]`
`unsigned char b[0]`	BUF	`add r0, fp, -B`	`ldrb r0, [fp, -B]`	`strb r0, [fp, -B]`
`unsigned char *ptr`	PTR	`add r0, fp, -PTR`	`ldr r0, [fp, -PTR]`	`str r0, [fp, -PTR]`

x

Working with Pointers on the stack

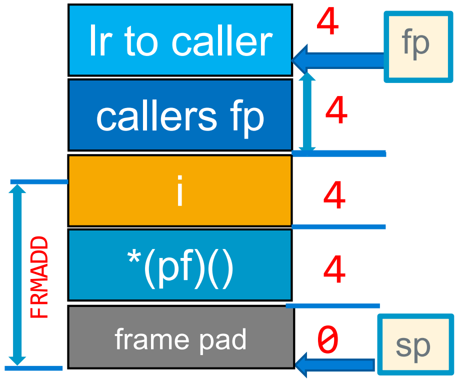
```c
int sum(int j, int k)
{
    return j + k;
}
void testp(int j, int k, int (*func)(int, int), int *i)
{
    *i = func(j,k);
    return;
}
int main()
{
    int i;                    // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum;  // pf could be in a register

    testp(1, 2, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

Output Parameters (like i) you pass a pointer to them, **must be on the stack!**

x

Working with Pointers on the stack

```
int main()
{
    int i; // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum;   // pf could be in a register

    testp(1, 2, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```



```
    .section .rodata
.Lmess: .string "%d\n"
    .extern printf
    .text
    .global main
    .type   main, %function
    .equ    FP_OFF, 4
    .equ    I,      4 + FP_OFF
    .equ    PF,     4 + I
    .equ    PAD,    0 + PF
    .equ    FRMADD, PAD-FP_OFF
// FRMADD =  12 - 4 = 8
```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
`int i`	I	`add r0, fp, -I`	`ldr r0, [fp, -I]`	`str r0, [fp, -I]`
`int (*pf)()`	PF	`add r0, fp, -PF`	`ldr r0, [fp, -PF]`	`str r0, [fp, -PF]`

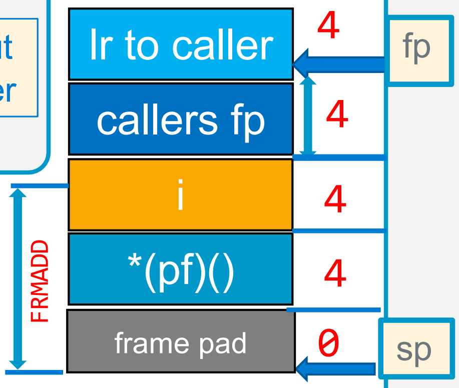
x

Working with Pointers on the stack

```
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

I is Output Parameter

```
    .section .rodata
.Lmess: .string "%d\n"
    .extern printf
    .text
    .global main
    .type   main, %function
    .equ    FP_OFF, 4
    .equ    I,      4 + FP_OFF
    .equ    PF,     4 + I
    .equ    PAD,    0 + PF
    .equ    FRMADD, PAD-FP_OFF
// FRMADD =  12 - 4 = 8
```



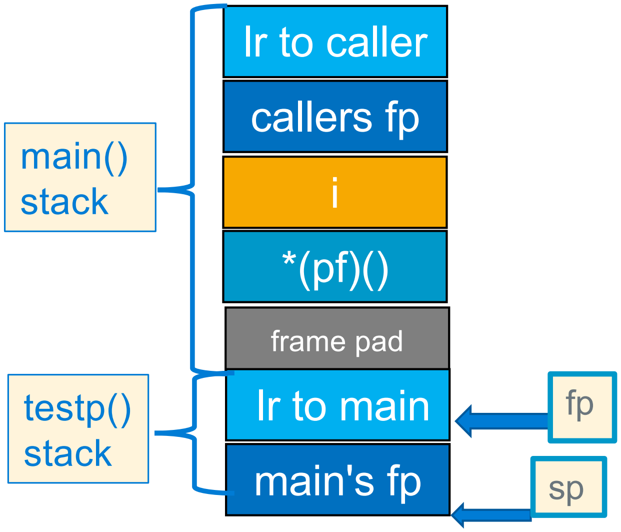
```
main:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp,-FRMADD

    ldr     r2, =sum        // func address
    add     r1, fp, -PF     // PF address
    str     r2, [r1]        // store in pf

    mov     r0, 1           // arg 1: 1
    mov     r1, 2           // arg 2: 2
    ldr     r2, [fp, -PF]   // arg 3: (*pf)()
    add     r3, fp, -I      // arg 4: &I
    bl      testp

    ldr     r0, =.Lmess     // arg 1: "%d\n"
    ldr     r1, [fp, -I]    // arg 2: I
    bl      printf

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

Variable	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]

x

Working with Pointers on the stack

```c
void
testp(int j, int k, int (*func)(int, int), int *i)
{
    *i = func(j,k);
    return;
}
```



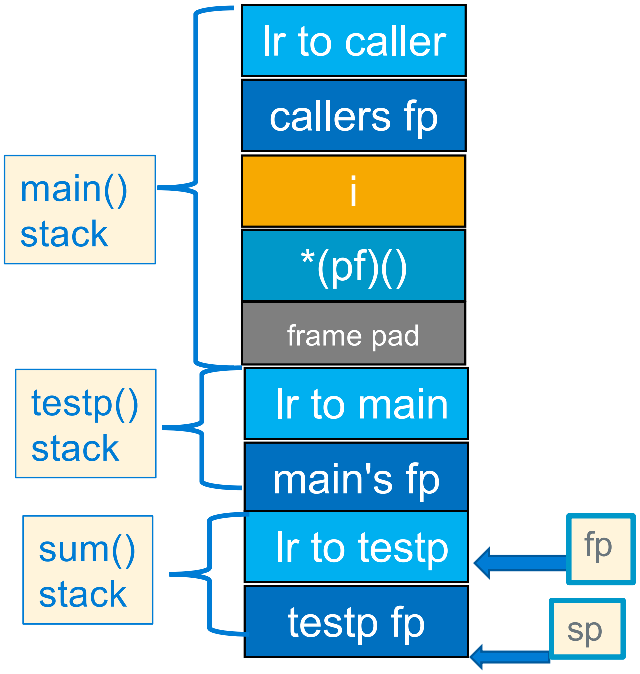
r0,r1,r2 already set

```
    .global testp
    .type   testp, %function
    .equ    FP_OFF, 12
testp:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF

    mov     r4, r3          // save i
    blx     r2              // r0=func(r0,r1)
    str     r0, [r4]        // *i =r0

    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
.size testp, (. - testp)
```

Working with Pointers on the stack

```
int
sum(int j, int k)
{
    return j + k;
}
```

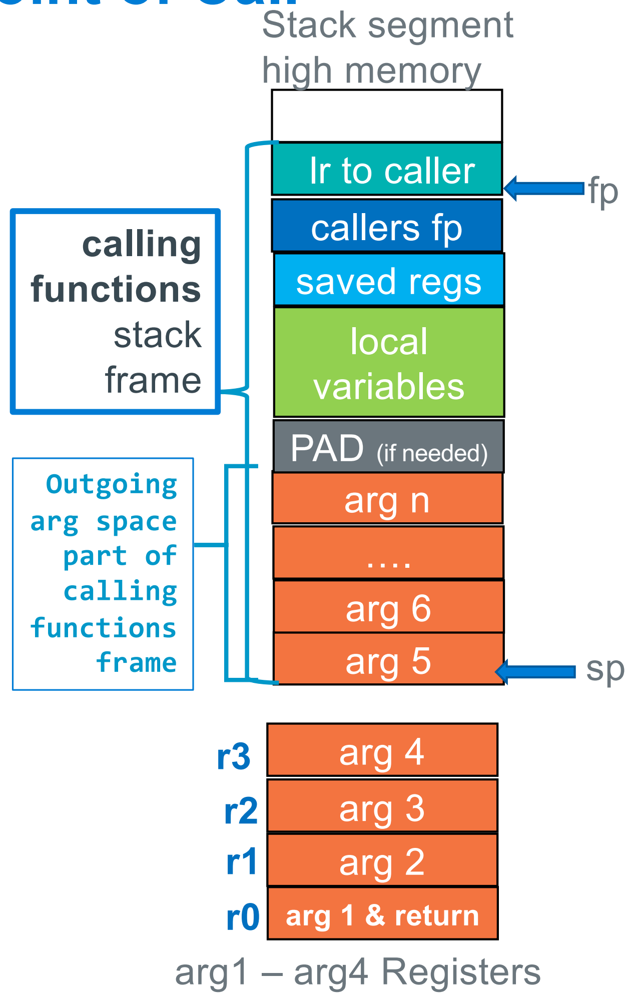
```
    .global sum
    .type   sum, %function
    .equ    FP_OFF, 4
sum:
    push    {fp, lr}
    add     fp, sp, FP_OFF

    add     r0, r0, r1

    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
.size sum, (. - sum)
```



Passing More Than Four Arguments – At the point of Call

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

- **Approach: Increase stack frame size to include space for args# > 4**
  - Arg5 and above are in caller's stack frame at the **bottom of the stack**
- **Arg5** is always at the **bottom (at sp)**, arg6 and greater are above it
- **One arg value per slot!** – NO arrays across multiple slots
  - chars, shorts and ints are directly stored
  - Structs (not always), and arrays (always) are passed via a pointer
- Output parameters contain an address ***that points at*** the stack, BSS, data, or heap
- Prior to any function call (and obviously at the start of the called function):
  1. sp must point at arg5
  2. sp and therefore **arg5 must be at an 8-byte boundary**,
  3. **Add padding** to force arg5 alignment if needed is **placed above** the last **argument the called function is expecting**



Stack segment high memory

calling functions stack frame

Outgoing arg space part of calling functions frame

lr to caller

callers fp

saved regs

local variables

PAD (if needed)

arg n

….

arg 6

arg 5

fp

sp

r3 arg 4

r2 arg 3

r1 arg 2

r0 arg 1 & return

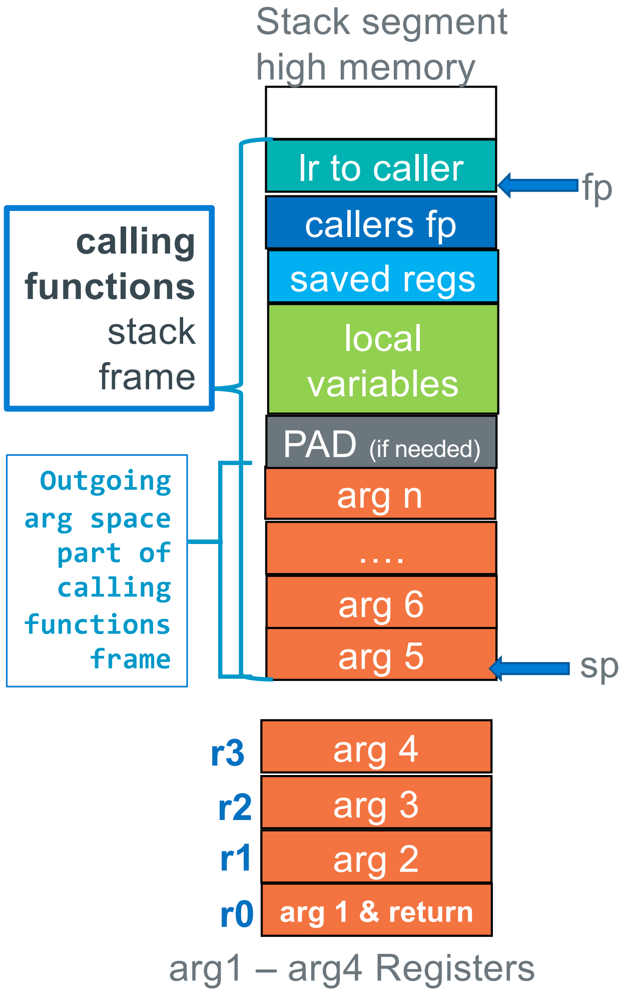
arg1 – arg4 Registers

x

Passing More Than Four Arguments – At the point of Call

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

- **Called functions** have the **right to change stack args** just like they can change the register args!
  - Caller **must always assume all args** including ones on the stack are **changed by the caller**
- Calling function prior to making the call you must
  1. Evaluate first four args: place the resulting values in r0-r3
  2. Evaluate Arg 5 and greater and place the resulting values on the stack

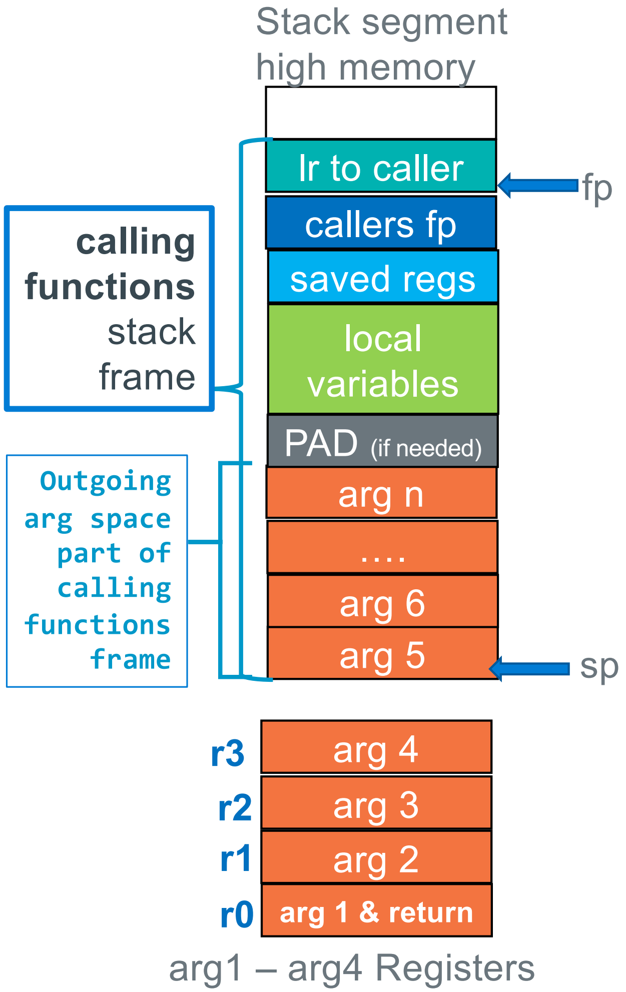


X

Passing More Than Four Arguments – At the point of Call

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

- **Approach**: Extend the stack frame to include enough space for stack arguments for the called function that has the greatest number of args
  1. Examine every function call in the body of a function
  2. Find the function call with greatest arg count, this determines space needed for outgoing args
  3. Add the greatest arg count space as needed to the frame layout
  4. Adjust PAD as required to keep the sp 8-byte aligned



x

Determining Size of 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

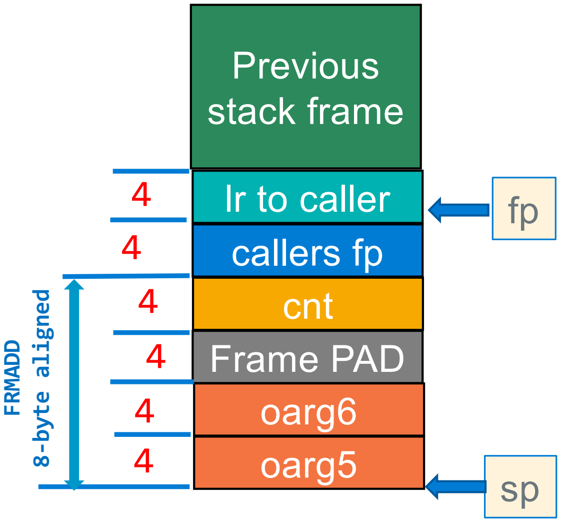
Calling Function Stack Frame: Pass ARG 5 and higher

Rules: At point of call

1. OARG5 must be pointed at by sp
2. SP must be 8-byte aligned at function call

```
int cnt;
r0 = func(r0, r1, r2, r3, OARG5, OARG6);
```

```
.equ   FP_OFF,4
.equ   CNT,           4 + FP_OFF     // int cnt
.equ   PAD,           4 + CNT        // added as needed
.equ   OARG6,         4 + PAD       // 4 bytes
.equ   OARG5,         4 + OARG6     // 4 bytes
.equ   FRMADD        OARG5 - FP_OFF
// FRMADD =  20 - 4 = 16
```



Variable	distance from fp	Address on Stack	Read variable	Write Variable
`int cnt`	CNT	`add r0, fp, -CNT`	`ldr r0, [fp, -CNT]`	`str r0, [fp, -CNT]`
`int oarg6`	OARG6	`add r0, fp, -OARG6`	`ldr r0, [fp, -OARG6]`	`str r0, [fp, -OARG6]`
`int oarg5`	OARG5	`add r0, fp, -OARG5`	`ldr r0, [fp, -OARG5]`	`str r0, [fp, -OARG5]`

x

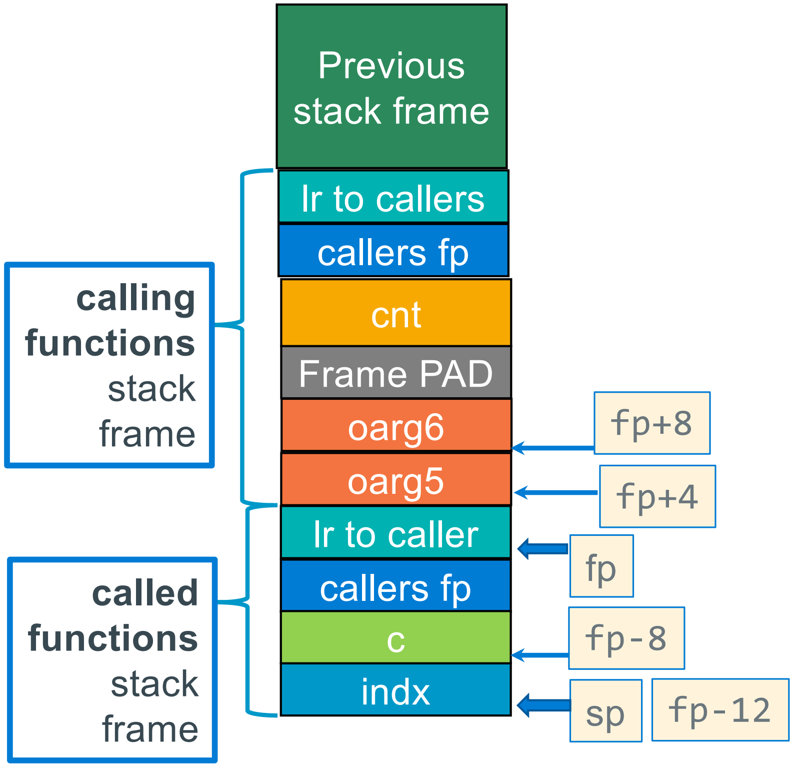
Called Function: Retrieving Args From the Stack

```
r0 = func(r0, r1, r2, r3, r4, ARG5, ARG6);
```

- At function start and before the push{} the sp is at an 8-byte boundary
- **Args > 4 in 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

```
.equ ARGN,  (N-4)*4  // where n must be > 4
```

- This "algorithm" for finding args was designed to enable variable arg count functions like printf("conversion list", arg0, … argn);
- No limit to the number of args (except running out of stack space)

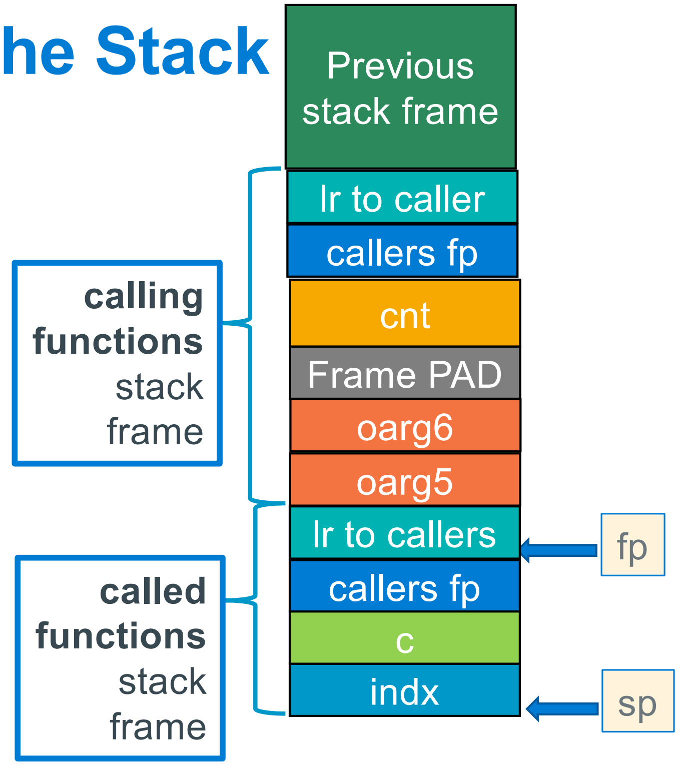


Rule:
Called functions always access stack args using a **positive offset to the fp**

Called Function: Retrieving Args From the Stack

```
.equ    FP_OFF,     4
.equ    C,          4 + FP_OFF
.equ    INDX,       4 + C
.equ    PAD,        4 + INDX
.equ    FRMADD,     PAD - FP_OFF
// below are distances into the caller's stack frame
.equ    ARG6,       8
.equ    ARG5,       4
```

```
r0 = func(r0, r1, r2, r3, r4, ARG5, ARG6);
```



Previous stack frame

calling functions stack frame

lr to caller

callers fp

cnt

Frame PAD

oarg6

oarg5

called functions stack frame

lr to callers

callers fp

c

indx

fp

sp

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

Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
`int arg6`	ARG6	`add r0, fp, ARG6`	`ldr r0, [fp, ARG6]`	`str r0, [fp, ARG6]`
`int arg5`	ARG5	`add r0, fp, ARG5`	`ldr r0, [fp, ARG5]`	`str r0, [fp, ARG5]`
`int c`	C	`add r0, fp, -C`	`ldr r0, [fp, -C]`	`str r0, [fp, -C]`
`int count`	INDX	`add r0, fp, -INDX`	`ldr r0, [fp, -INDX]`	`str r0, [fp, -INDX]`

Observe the positive offsets

x

Example: Passing Stack Args, Calling Function

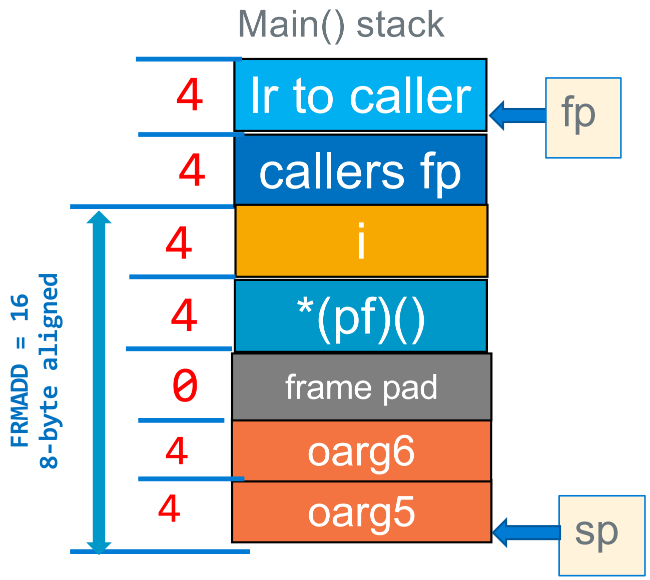
```
int sum(int j, int k)
{
    return j + k;
}

void   arg1    arg2   arg3   arg4       arg5              arg6
testp(int j, int k, int l, int m, int (*func)(int, int), int *i)
{
    *i = func(j,k) + func(l, m);  // notice two func() calls

    return;
}

int main()
{
    int i;  // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);

    return EXIT_SUCCESS;
}
```

x

Example: Passing Stack Args, Calling Function

```
int main()
{
    int i;  // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

```
    .equ    FP_OFF, 4
    .equ    I,      4 + FP_OFF
    .equ    PF,     4 + I
    .equ    PAD,    0 + PF
    .equ    OARG6,  4 + PAD
    .equ    OARG5   4 + OARG6
    .equ    FRMADD, OARG5 – FP_OFF
// FRMADD =  20 - 4 = 16
```

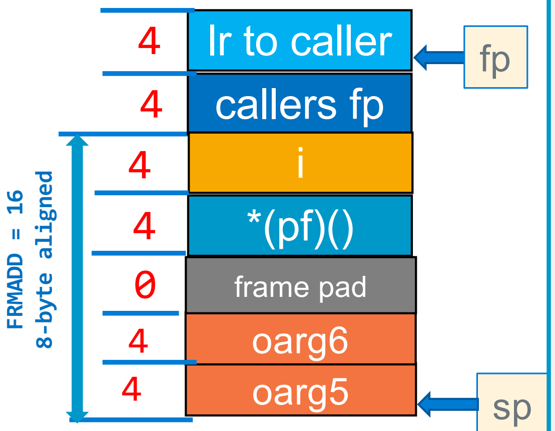


Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

x

Example: Passing Stack Args, Calling Function

```
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```



```
    .equ    FP_OFF, 4
    .equ    I,      4 + FP_OFF
    .equ    PF,     4 + I
    .equ    PAD,    0 + PF
    .equ    OARG6,  4 + PAD
    .equ    OARG5   4 + OARG6
    .equ    FRMADD, OARG5 – FP_OFF
// FRMADD =  20 - 4 = 16
```

```
main:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp,–FRMADD

    ldr     r0, =sum        // get func address
    add     r1, fp, –PF     // PF address on stack
    str     r0, [r1]        // store sum in var pf

    add     r0, fp, –I      // get address of I
    add     r1, fp, –OARG6  // address of OARG6
    str     r0, [r1]        // arg 6: store address of I

    ldr     r0, [fp, –PF]   // get PF from stack
    add     r1, fp, –OARG5  // address of OARG5
    str     r0, [r1]        // arg 5: store sum() address

    mov     r0, 1           // arg 1: 1
    mov     r1, 2           // arg 2: 2
    mov     r2, 3           // arg 3: 3
    mov     r3, 4           // arg 4: 4

    bl      testp

    ldr     r0, =.Lmess     // arg 1: "%d\n"
    ldr     r1, [fp, –I]    // arg 2: i
    bl      printf

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

Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

x

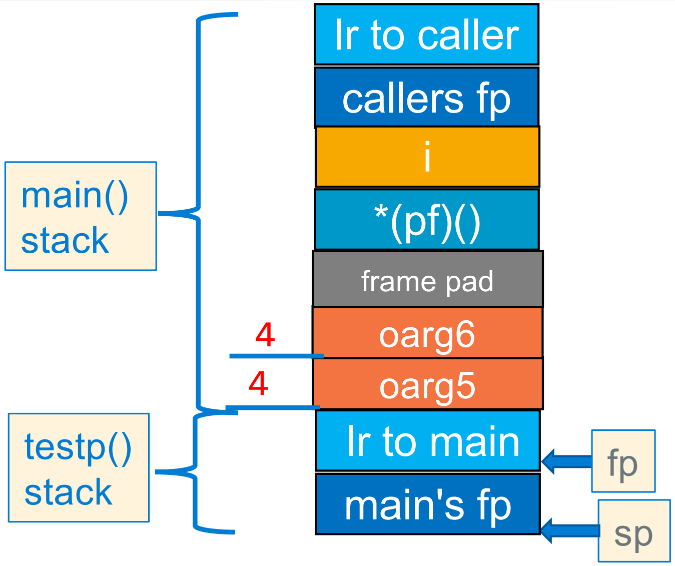
Example: Passing Stack Args, Called Function

arg1 arg2 arg3 arg4 arg5 arg6

```
void
testp(int j, int k, int l, int m, int (*func)(int, int), int *i)
{
    *i = func(j, k) + func(l, m);

    return;
}
```

short circuit: make this call first

```
        .equ     FP_OFF, 20
        .equ     ARG6,    8
        .equ     ARG5,    4
testp:
        push     {r4-r7, fp, lr}
        add      fp, sp, FP_OFF

        mov      r4, r2            // save l
        mov      r5, r3            // save m
        ldr      r6, [fp, ARG5]    // load func
        ldr      r7, [fp, ARG6]    // load i
        blx      r6                // r0 = func(j, k)

        mov      r1, r5            // arg 2 saved m
        mov      r5, r0            // save func return value
        mov      r0, r4            // arg 1 saved l
        blx      r6                // r0 = func(l, m)
        add      r0, r0, r5        // func(l,m) + func(j,k)
        str      r0, [r7]          // store sum to *i

        sub      sp, fp, FP_OFF
        pop      {r4-r7, fp, lr}
        bx       lr
```

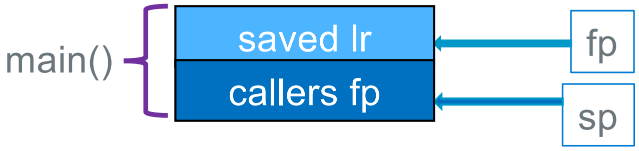


Argument	distance	Address on Stack	Read variable	Write Variable
int *i	ARG6	add r0, fp, ARG6	ldr r0, [fp, ARG6]	str r0, [fp, ARG6]
int (*fp)()	ARG5	add r0, fp, ARG5	ldr r0, [fp, ARG5]	str r0, [fp, ARG5]

x

Extra Slides

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



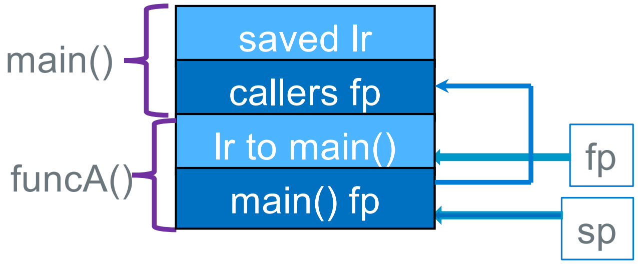
main()

saved lr

callers fp

fp

sp



main()

funcA()

saved lr

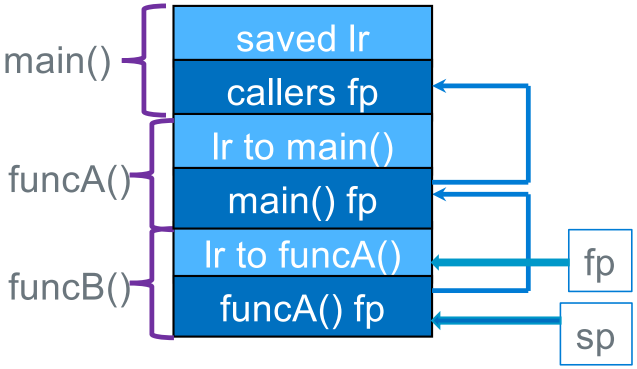
callers fp

lr to main()

main() fp

fp

sp



main()

funcA()

funcB()

saved lr

callers fp

lr to main()

main() fp

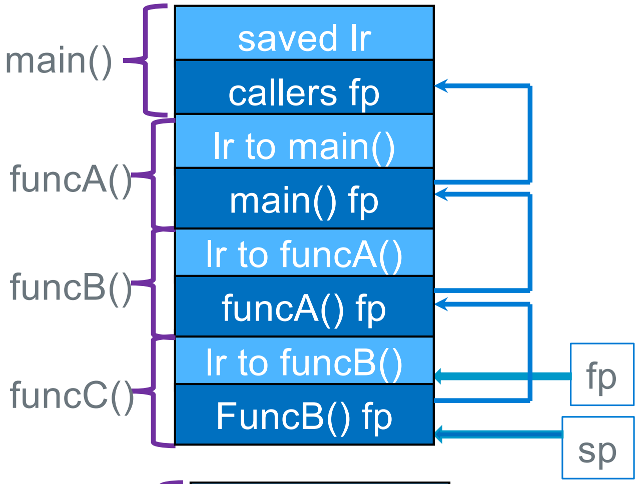
lr to funcA()

funcA() fp

fp

sp

How gdb finds stack frames



main()

funcA()

funcB()

funcC()

saved lr

callers fp

lr to main()

main() fp

lr to funcA()

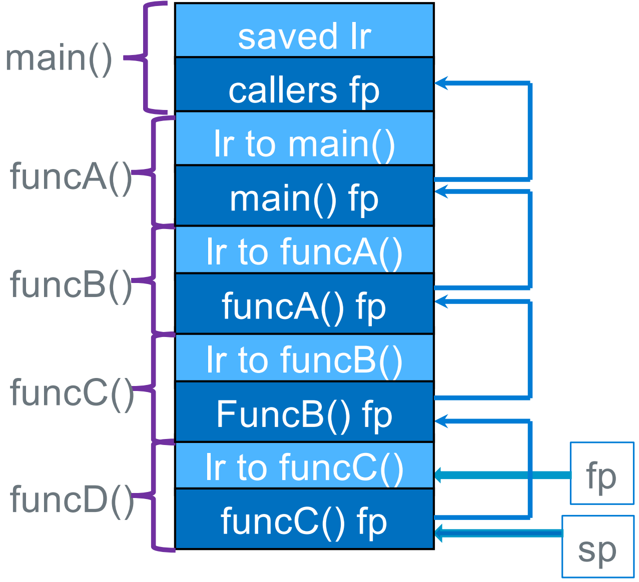
funcA() fp

lr to funcB()

FuncB() fp

fp

sp



main()

funcA()

funcB()

funcC()

funcD()

saved lr

callers fp

lr to main()

main() fp

lr to funcA()

funcA() fp

lr to funcB()

FuncB() fp

lr to funcC()

funcC() fp

fp

sp

x