Version 2.18

UCSD CSE 30

Computer Organization and Systems Programming

Lecture – 19

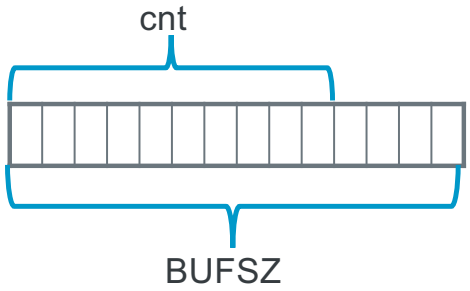
PA8 Review

Keith Muller

C fread() and fwrite()

element size of 1 with a char buffer is byte I/O
Capture bytes read so you know how many bytes to write

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



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

```
size_t
fread(void *ptr, size_t size, size_t count, FILE *stream)
```

- Reads an array of ***count elements*** of ***size*** bytes from *stream*

```
size_t
fwrite(void *ptr, size_t size, size_t count, FILE *stream
```

- Writes an array of ***count elements*** of ***size*** bytes to *stream*

```
int copy(FILE *infp, FILE *outfp, int sz, unsigned char *buf)
{
    size_t cnt;

    while ((cnt = fread(buf, 1, sz, infp)) > 0)
        (void) fwrite(buf, 1, cnt, outfp);
    return 0;
}
```

Note: this is a demo, proper checks are not being made

```
    copy:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF
    cmp     r2, 0
    ble     .Ldone
    mov     r4, r0  // infp
    mov     r5, r1  // outfp
    mov     r6, r2  // sz
    mov     r7, r3  // buf

.Lloop:
      // fread(inbuf, 1, cnt, infp)
    mov     r0, r7
    mov     r1, 1
    mov     r2, r6
    mov     r3, r4
    bl      fread
    cmp     r0, 0
    ble     .Ldone
    // fwrite(inbuf, 1, cnt, outfp)
    //rest not shown
```

x

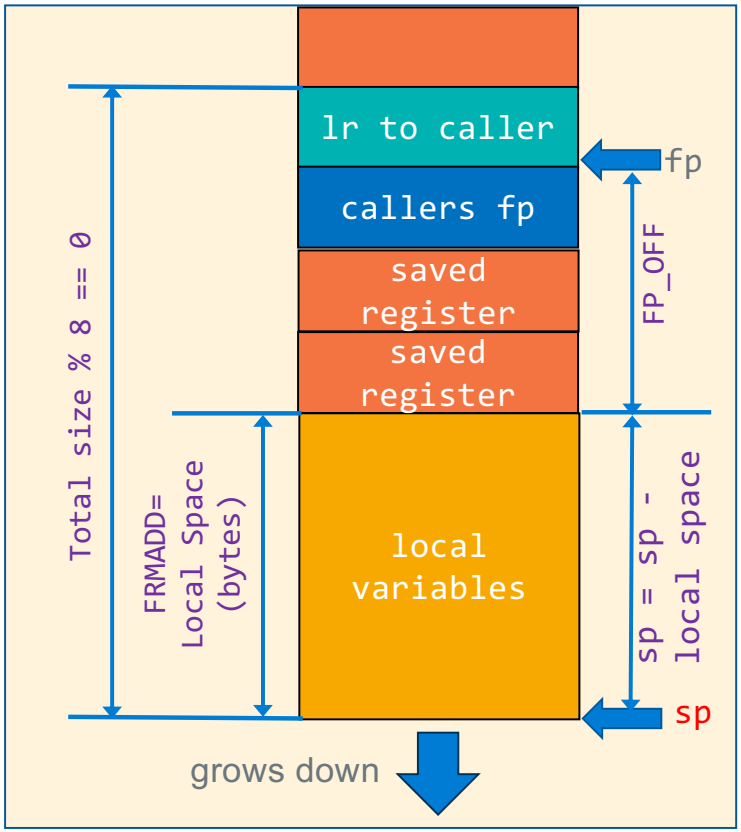
Allocating Space For Locals on the Stack

- 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

Review Variables: Size

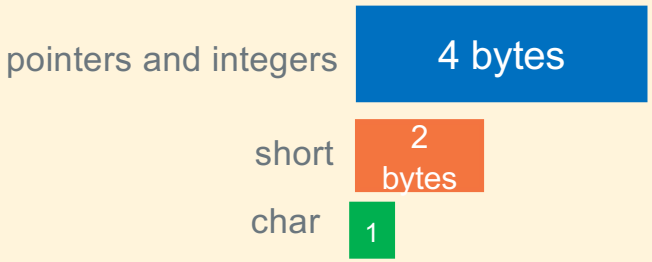
- **Integer types**
  - `char (unspecified default)`
  - `int (signed default)`
- **Floating Point**
  - `float`, `double`
- Optional Modifiers for each base type
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]
  - `const`: variable read only
- **char type**
  - One byte in a byte addressable memory
  - **Be careful** char is unsigned on arm and signed on other HW like intel

C Data Type	AArch-32 contiguous Bytes	printf specification
unsigned char	1	%c
signed char	1	%c
short int	2	%hd
unsigned short int	2	%hu
int	4	%d / %i
unsigned int	4	%u
long int	4	%ld
long long int	8	%lld
float	4	%f
double	8	%lf
long double	8	%Lf
pointer *	4	%p

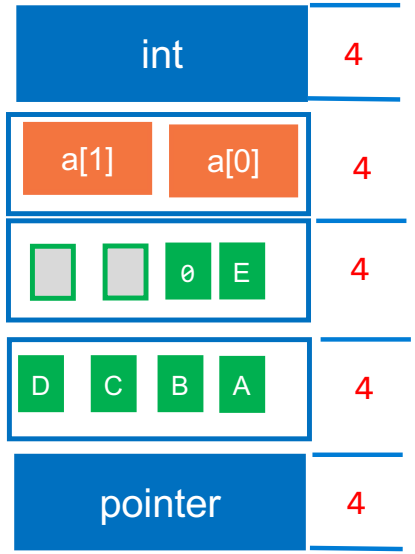
x

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)



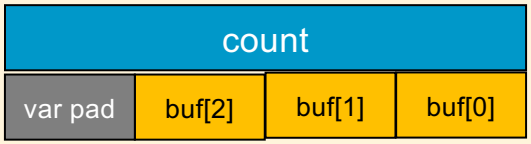
Rule: When the function is entered the stack is already 8-byte aligned



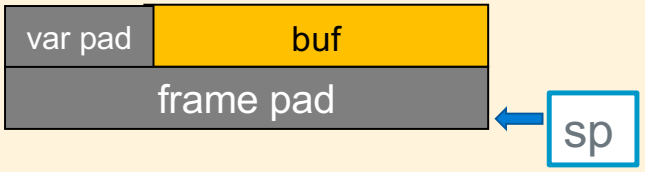
x

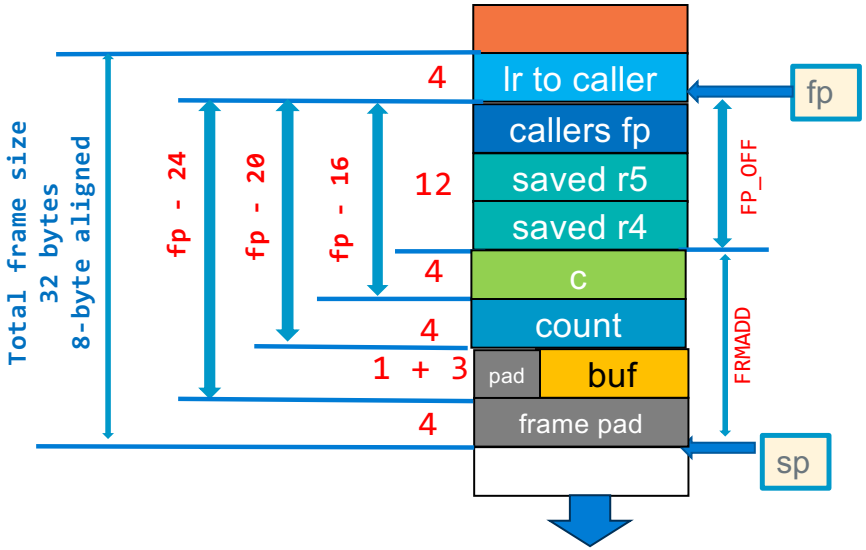
Stack Variables: Padding

- **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



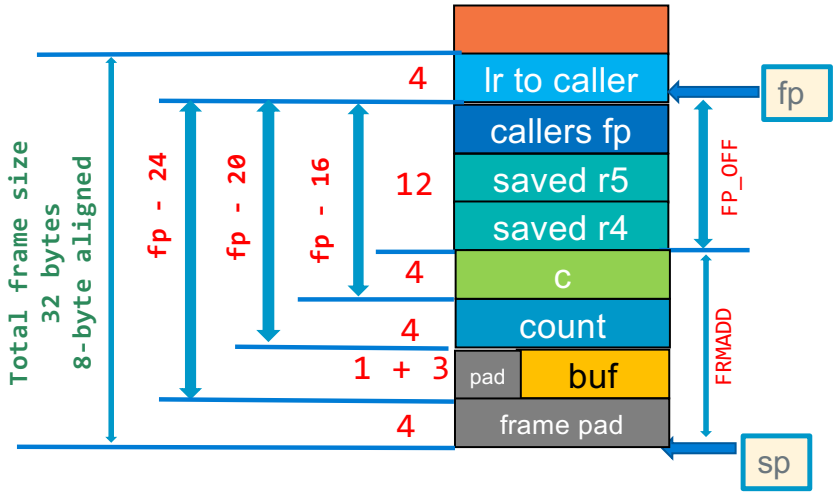



```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
 .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

Best Practice: Assembler Generated FP Distance Table



FP Distance Table one For each function

```
.type   main, %function
.global main
                                pushed reg fp distance
.equ    FP_OFF,     12
                                Prior allocation distance
    variable size in bytes
.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
```

1. For each stack variable create a .equ symbol whose value is the distance in bytes from the FP after the prologue
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

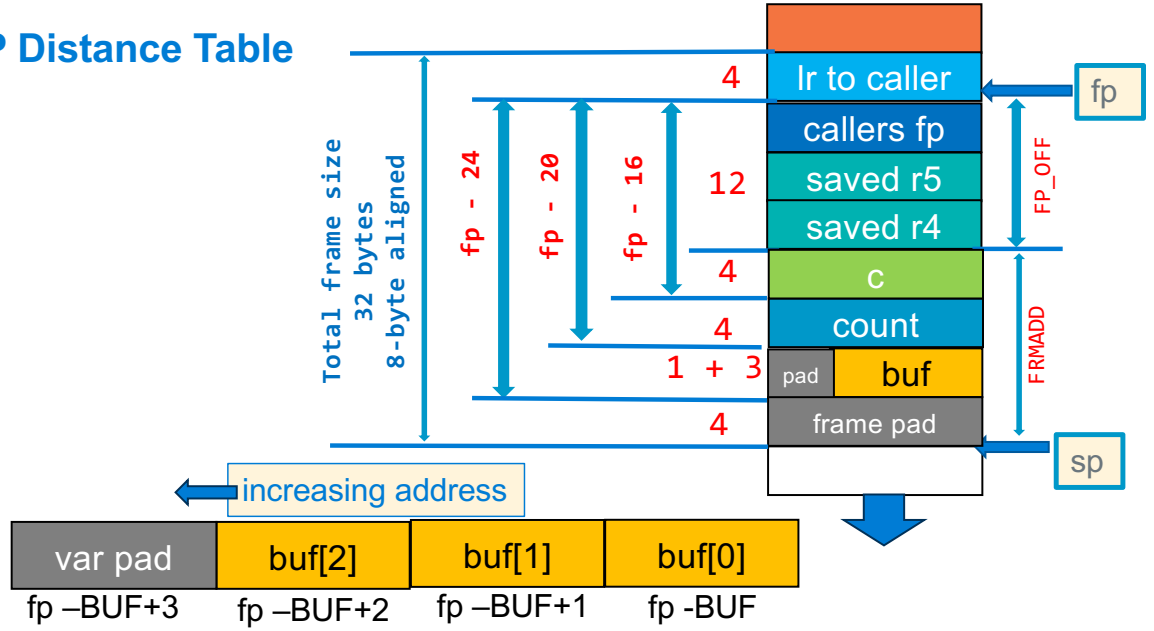
   FRMADD = distance PAD minus distance of the SP to the FP (FP_OFF) after the prologue push

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
```



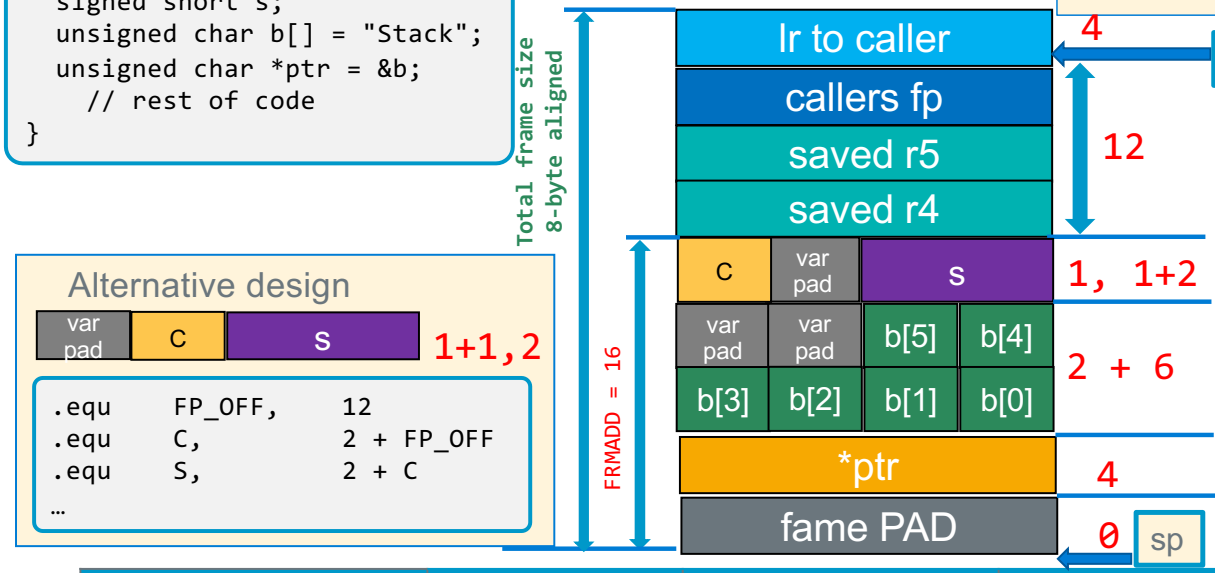
Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add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

Stack Frame Design Practice

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

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



Alternative design


```
.equ    FP_OFF,     12
.equ    C,          2 + FP_OFF
.equ    S,          2 + C
…
```

```
.equ    FP_OFF,     12
.equ    C,          1 + FP_OFF
.equ    S,          3 + C
.equ    B,          8 + S
.equ    PTR,        4 + B
.equ    PAD,        0 + PTR
.equ    FRMADD,     PAD – FP_OFF
// FRMADD =  28 - 12 = 16
```

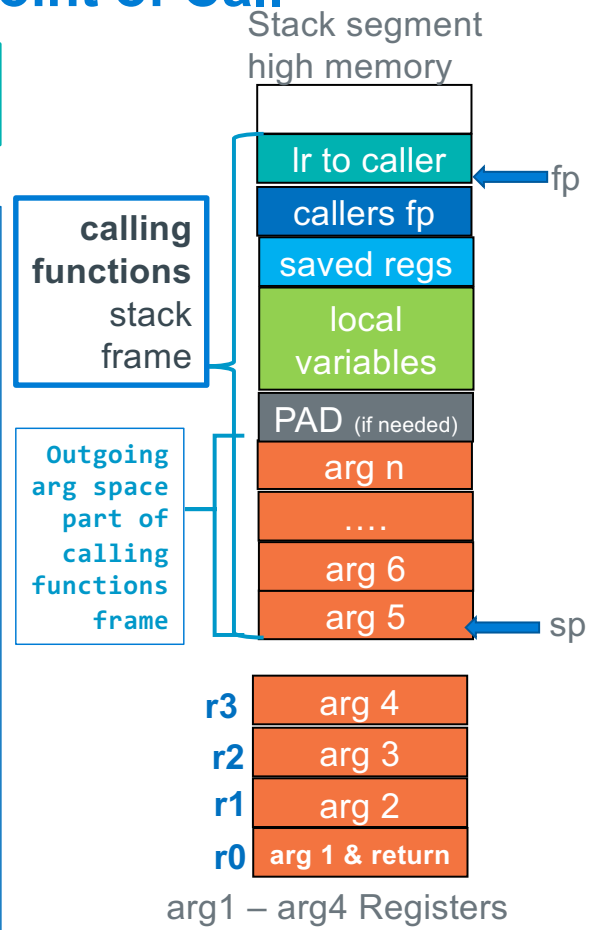
Variable	distance from fp	Address on Stack	Read variable	Write Variable
signed char c	C	add r0, fp, -C	ldrsb r0, [fp, -C]	strsb r0, [fp, -C]
signed short s	S	add r0, fp, -S	ldrsh r0, [fp, -S]	strsh r0, [fp, -S]
unsigned char b[0]	B	add r0, fp, -B	ldrb r0, [fp, -B]	strb r0, [fp, -B]
unsigned char *ptr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

x

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**

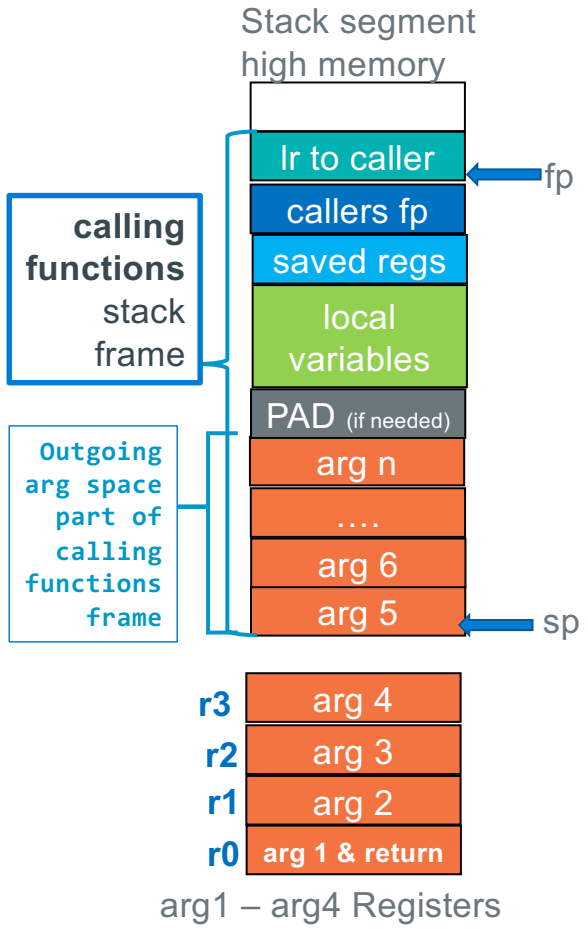


x

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

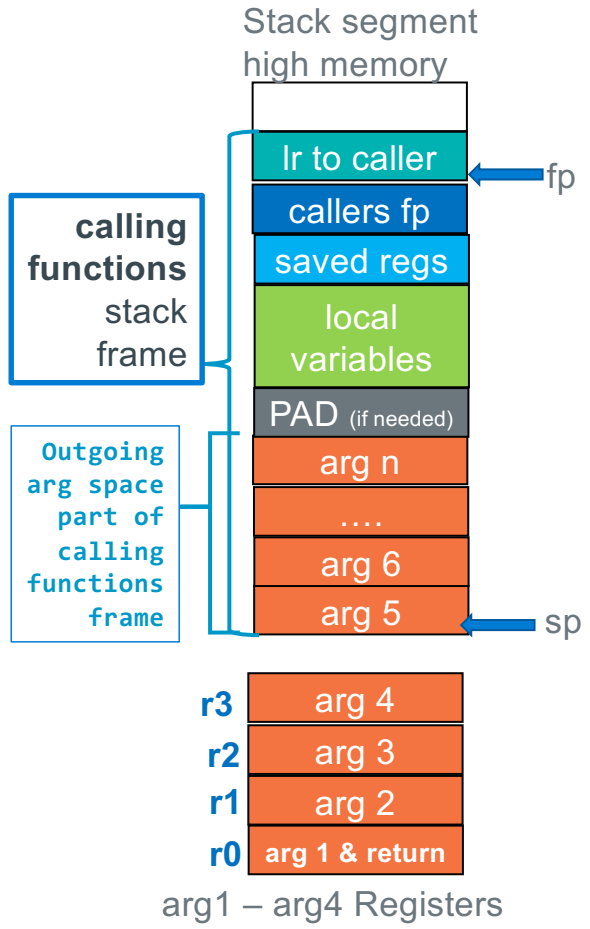


x

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

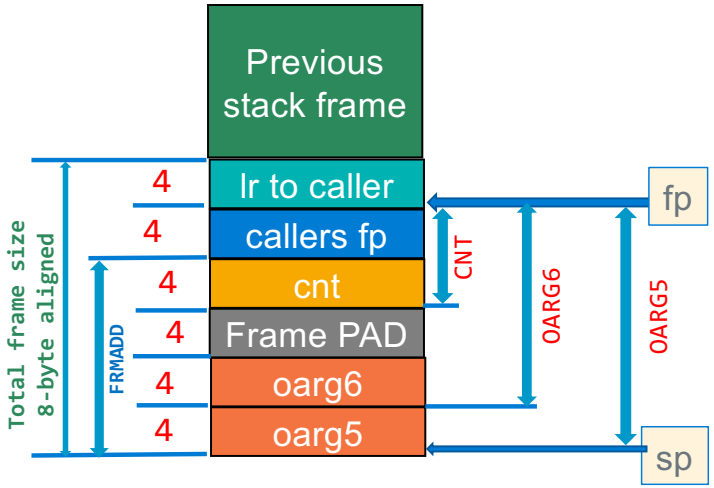
Calling Function Stack Frame: Pass ARG 5 and higher

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

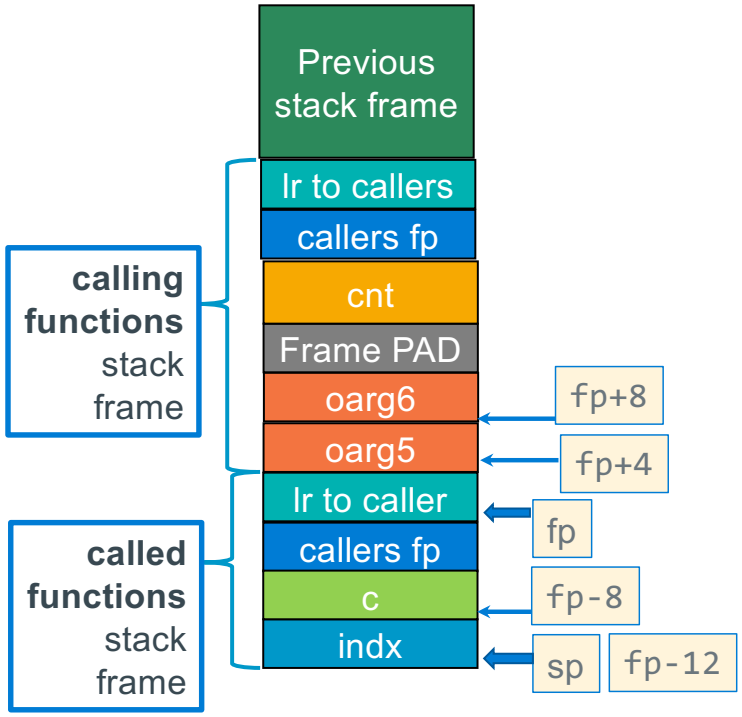
Called Function: Retrieving Args From the Stack

```
r0 = func(r0, r1, r2, r3, 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)



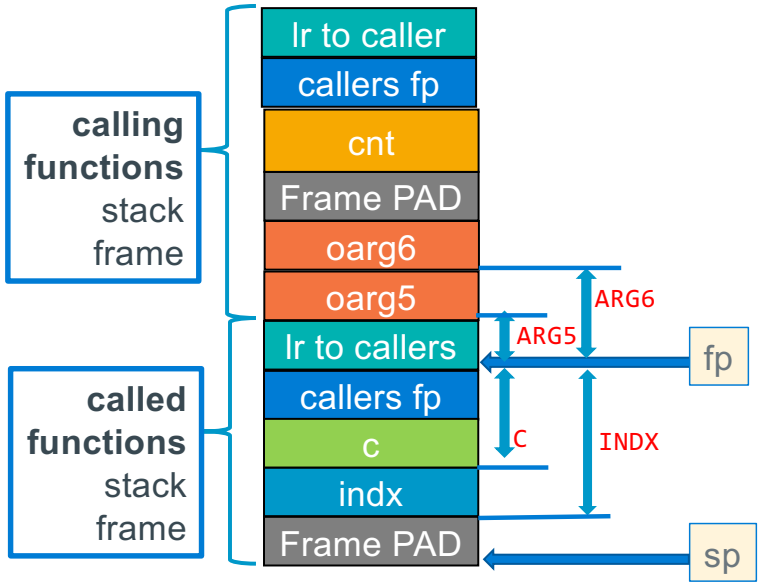
Rule:
Called functions always access stack args using a **positive offset to the fp**

x

Called Function: Retrieving Args From the Stack

```
.equ    FP_OFF,     4
.equ    C,          4 + FP_OFF
.equ    INDX,       4 + C
.equ    PAD,        0 + 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);
```



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

Example: Passing Stack Args, Calling Function

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

void
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;
}
```

arg1 arg2 arg3 arg4 arg5 arg6

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

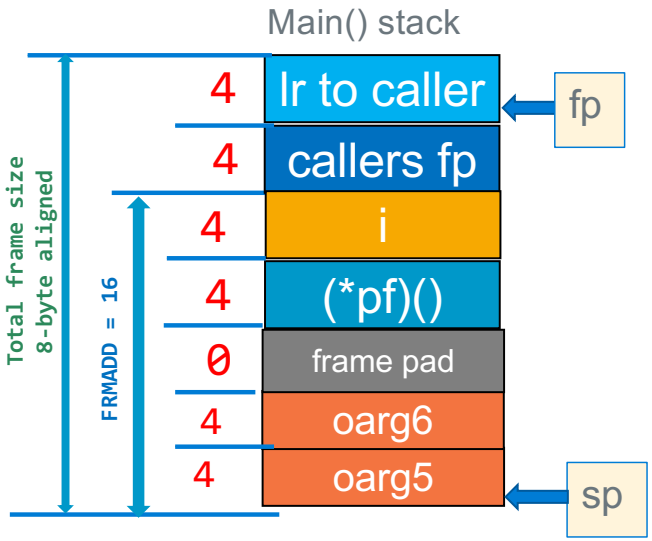
x

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]

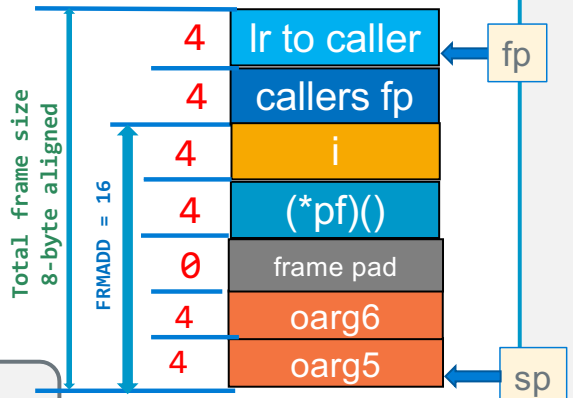
x

Example: Passing Stack Args, Calling Function

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

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

```
    .section .rodata
.Lmess: .string "%d\n"
// other stuff not shown
    .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
    str     r0, [fp, -PF]   // store sum in var pf

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

    ldr     r0, [fp, -PF]   // get PF from stack
    str     r0, [fp,-OARG5] // 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

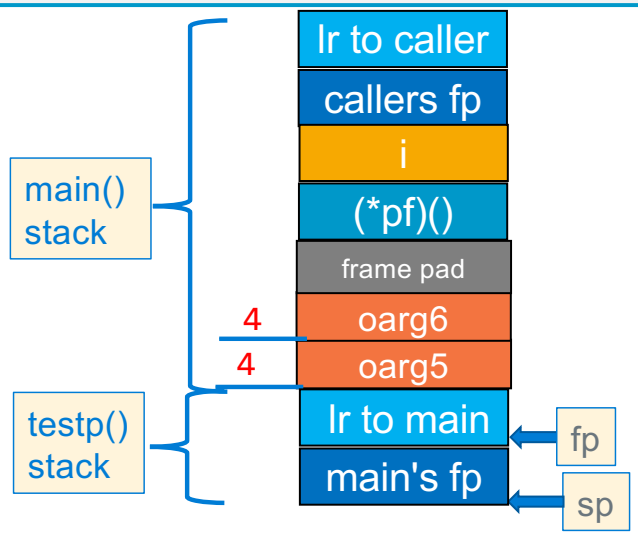
Example: Passing Stack Args, Called Function

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;
}
```

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 (*pf)()`	ARG5	`add r0, fp, ARG5`	`ldr r0, [fp, ARG5]`	`str r0, [fp, ARG5]`

x