

Version 2.12

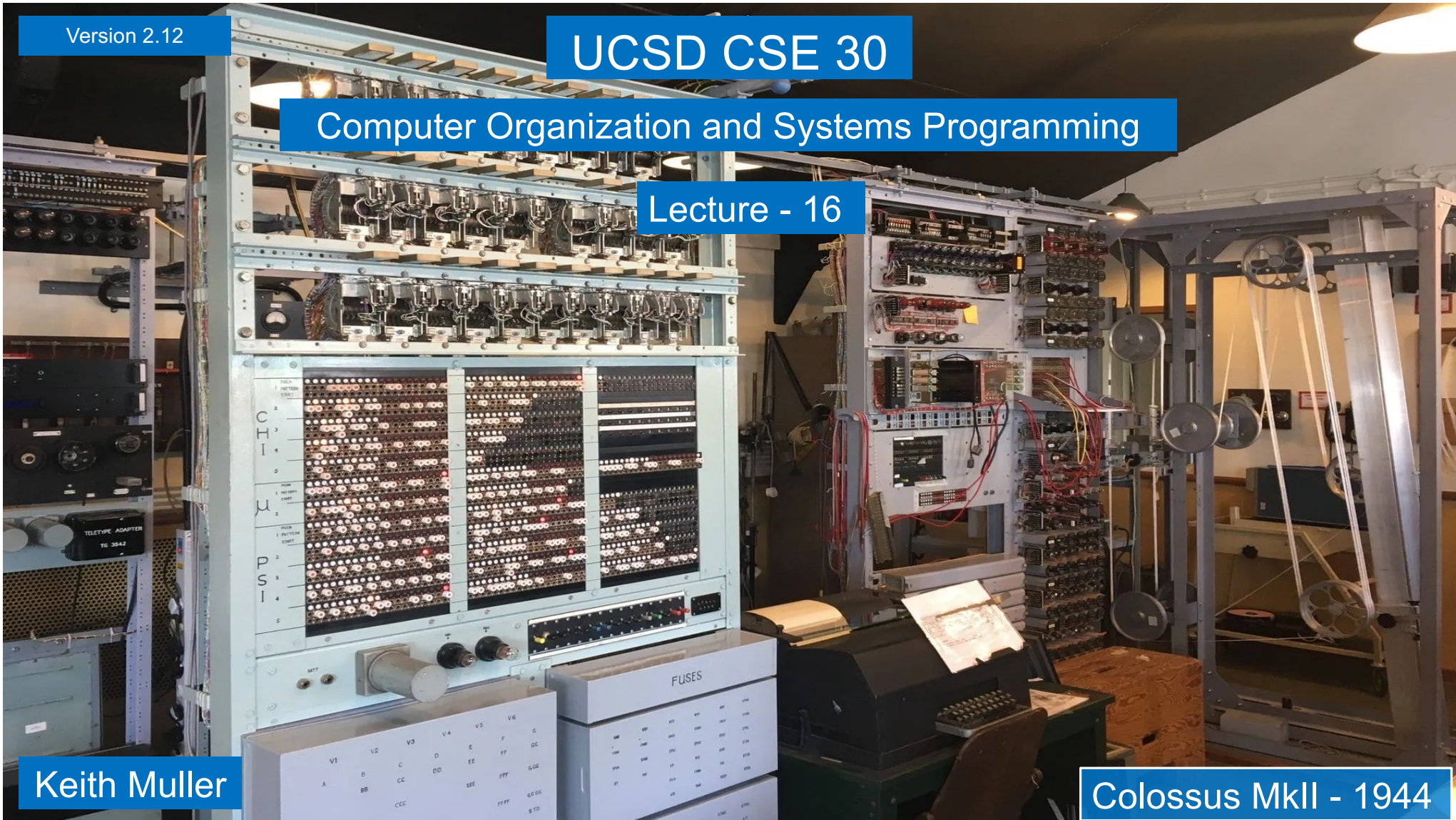
# UCSD CSE 30

## Computer Organization and Systems Programming

### Lecture - 16

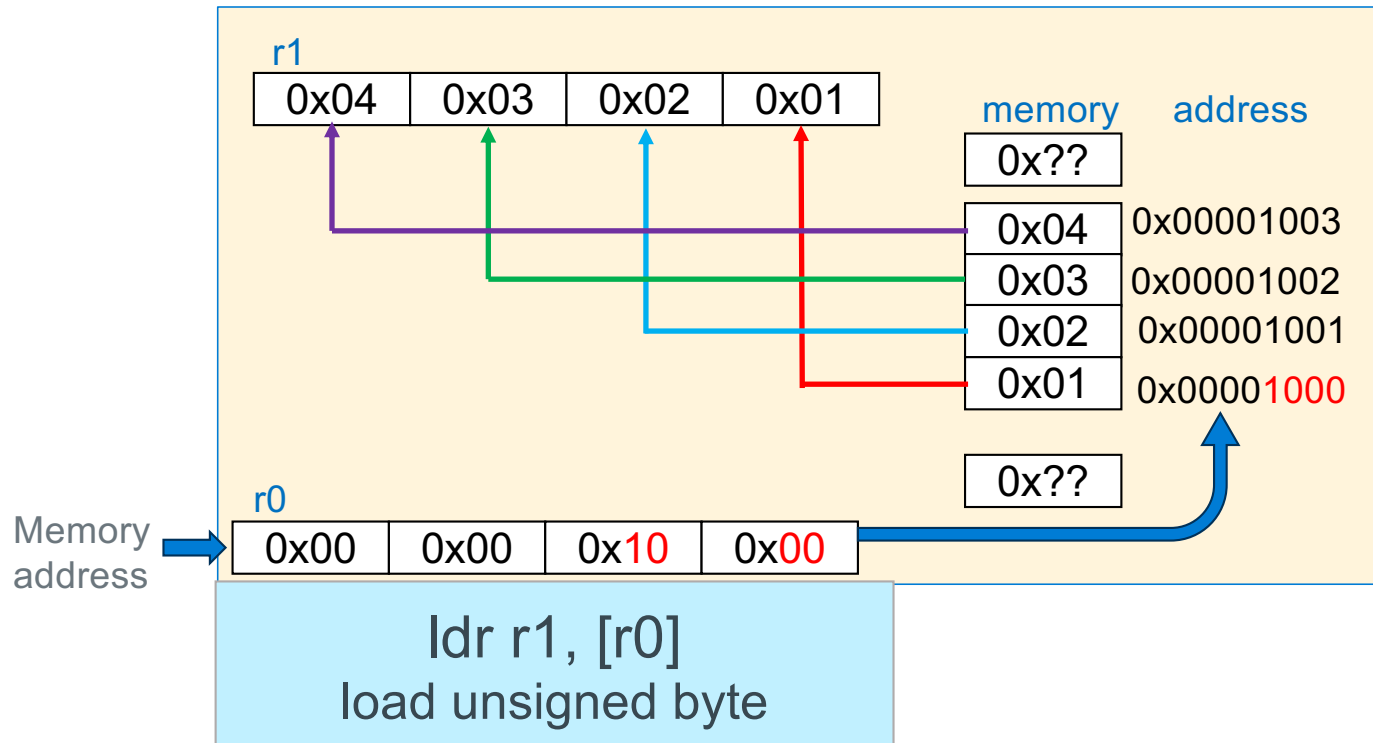
Keith Muller

Colossus MkII - 1944

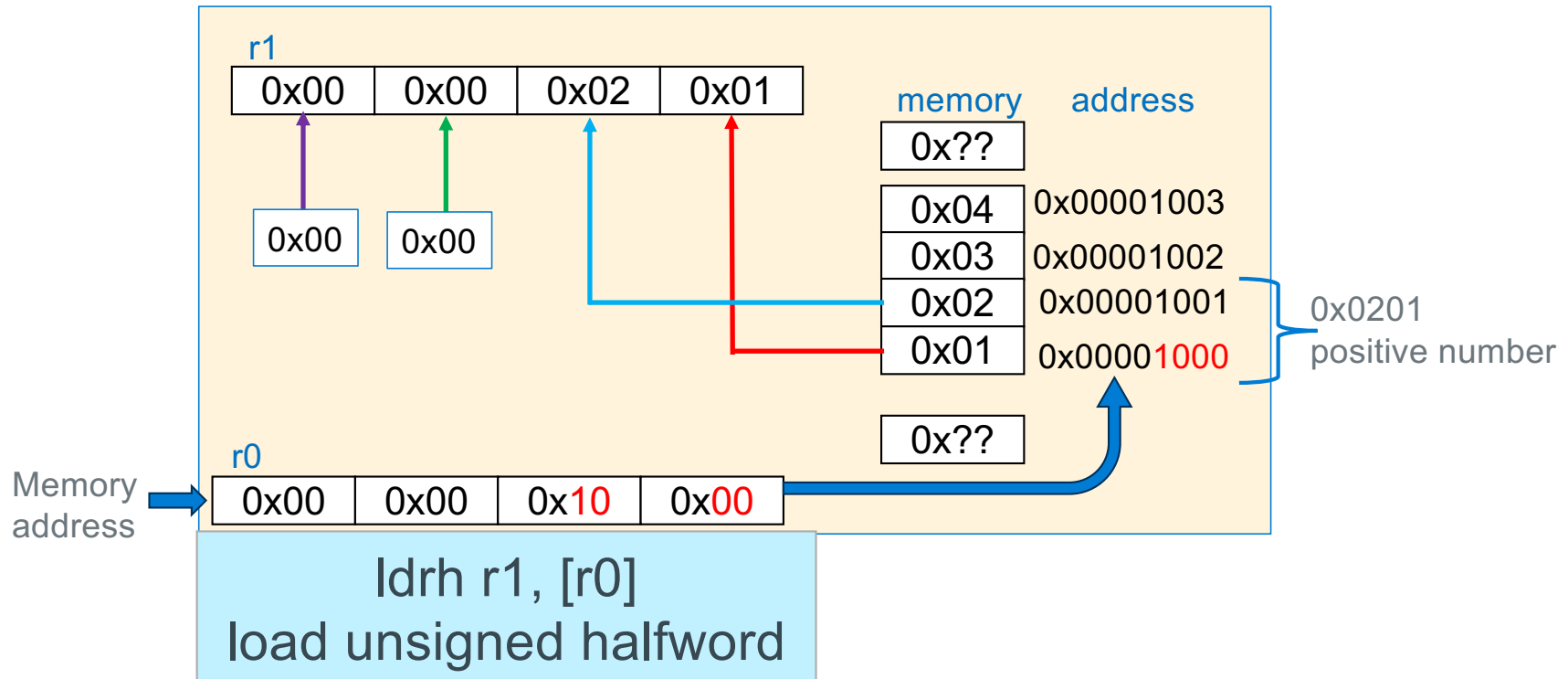




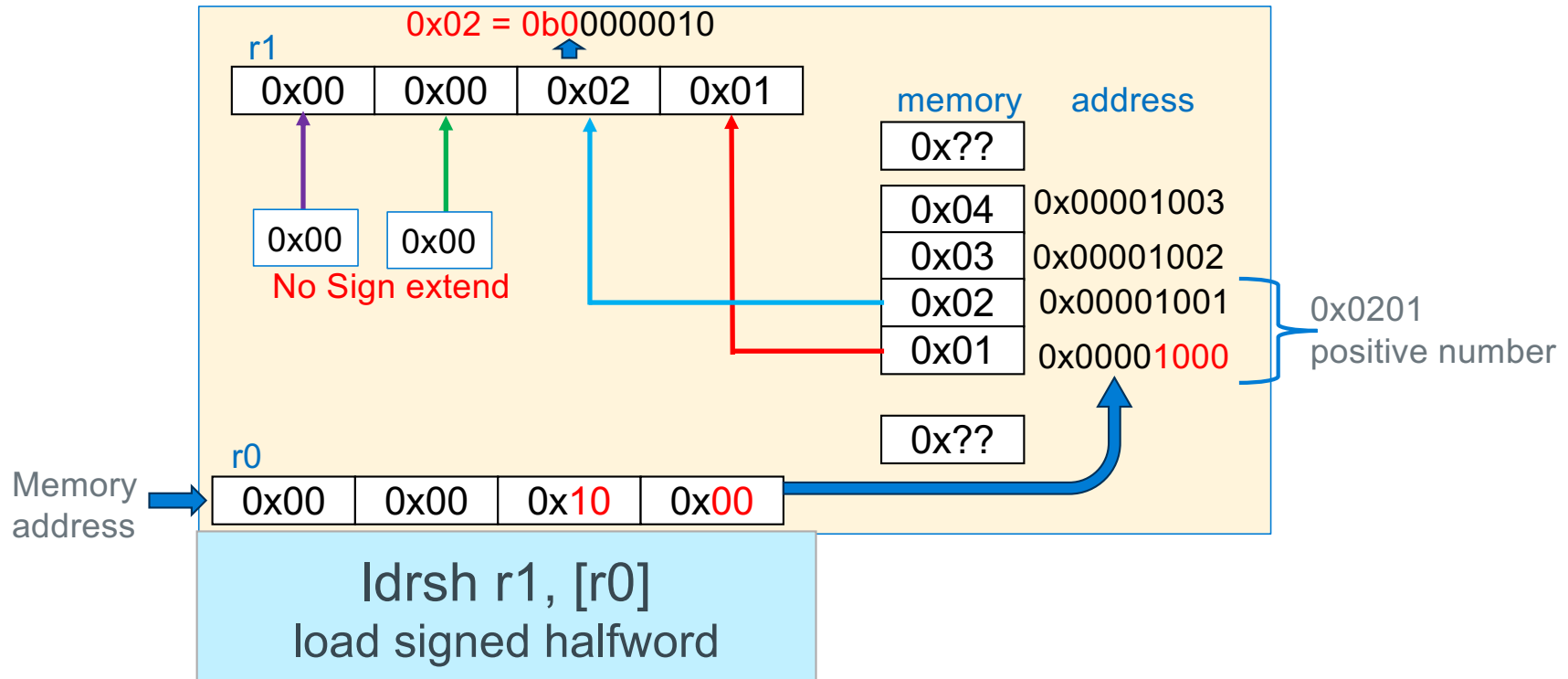
## Loading 32-bit Registers From Memory, 32-bit



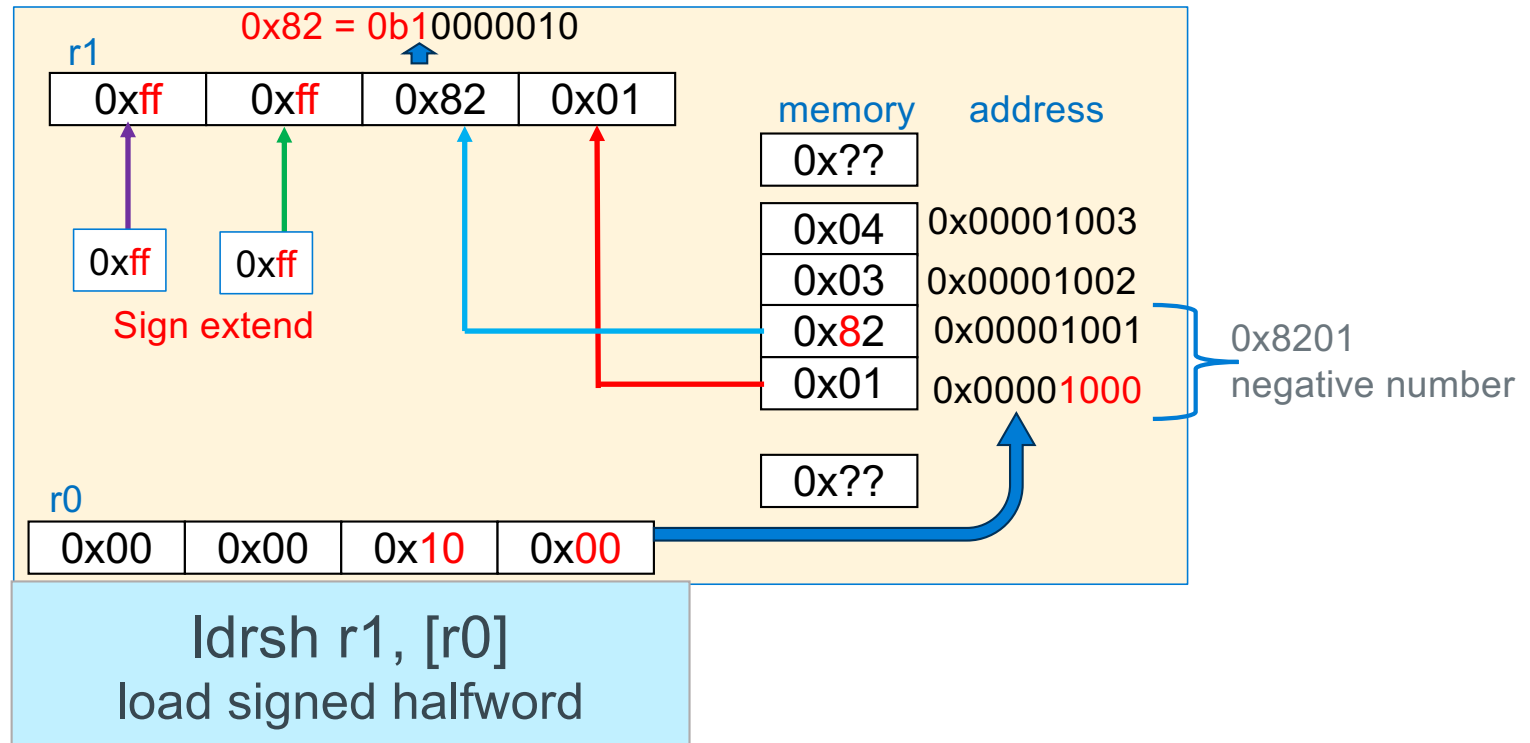
## Loading 32-bit Registers From Memory, 16-bit



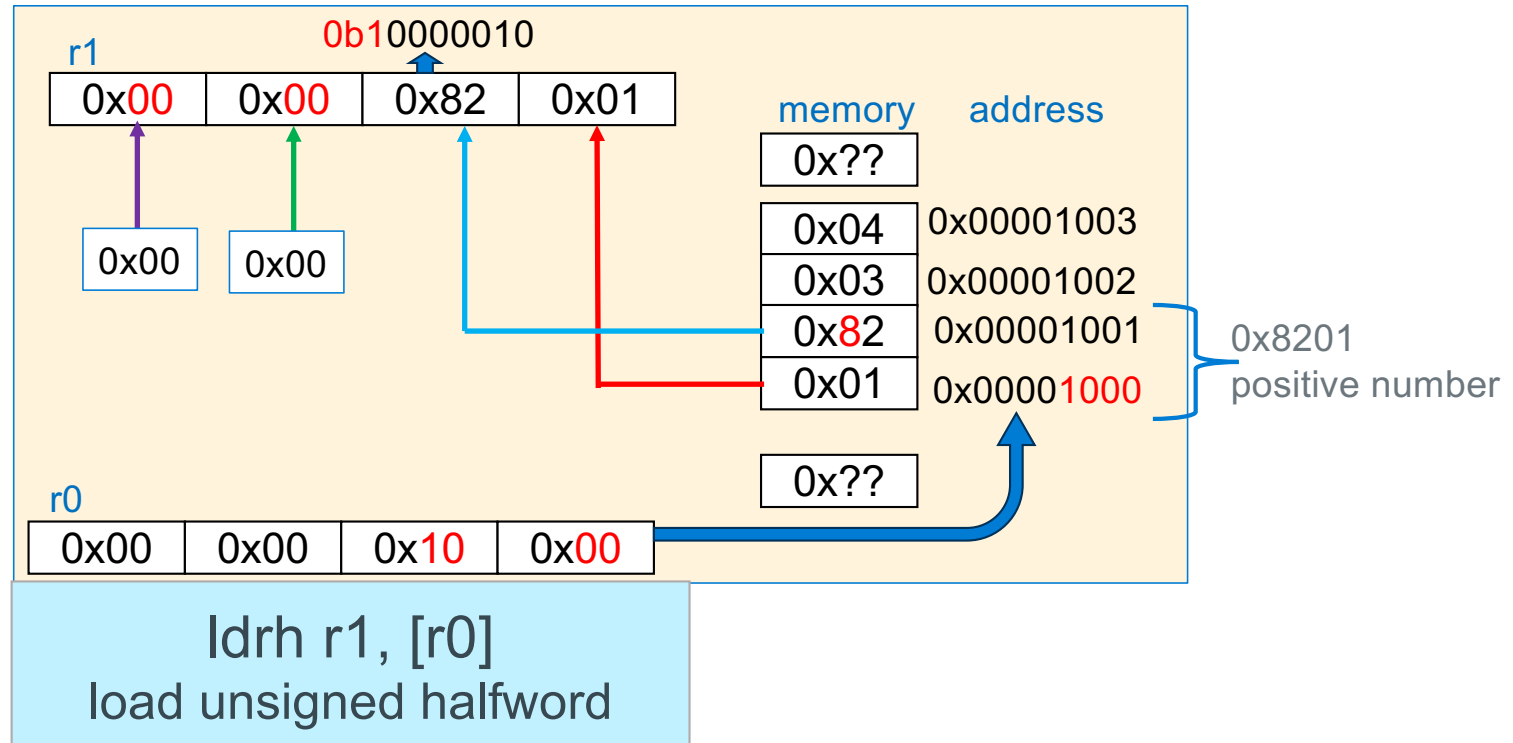
## Loading 32-bit Registers From Memory, 16-bit



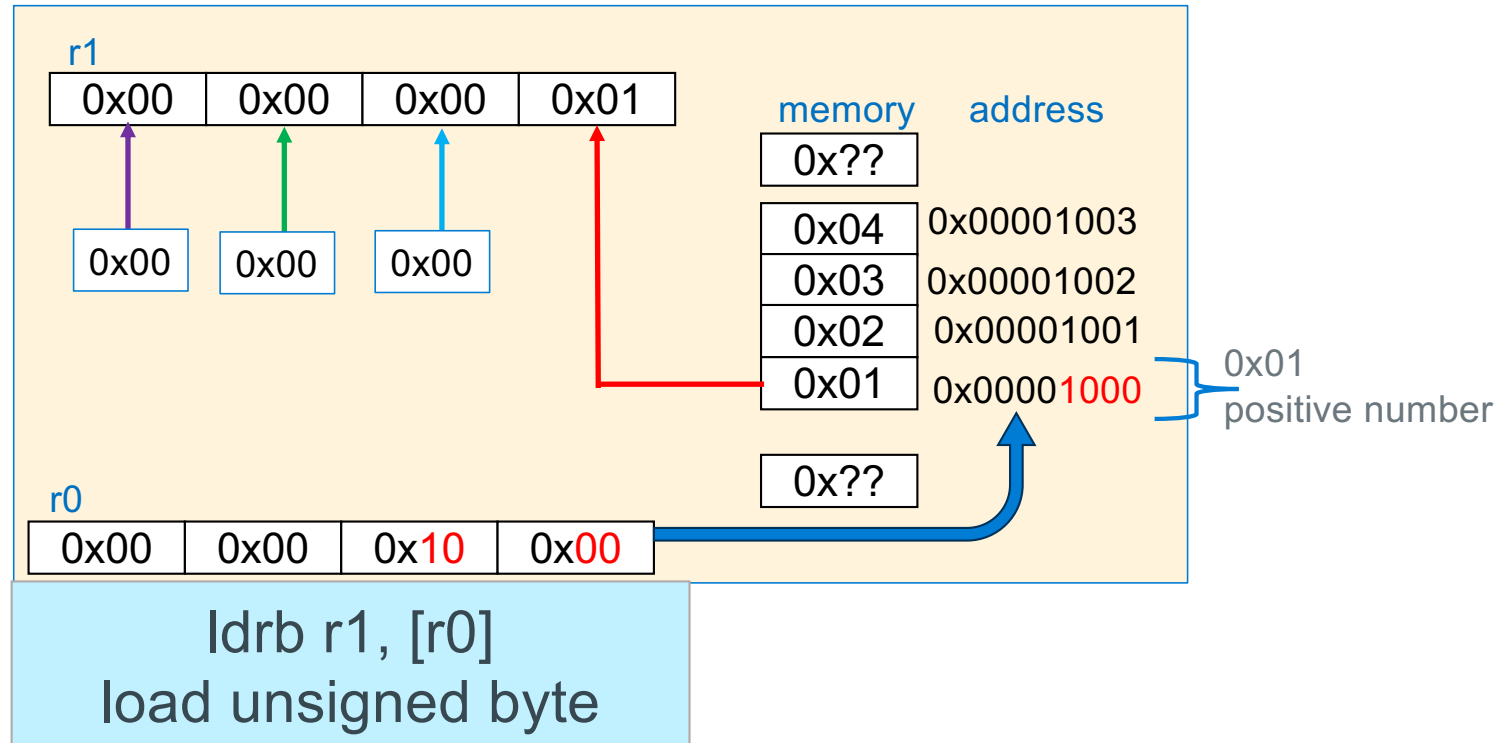
## Loading 32-bit Registers From Memory, 16-bit Signed



## Loading 32-bit Registers From Memory, 16-bit Unsigned

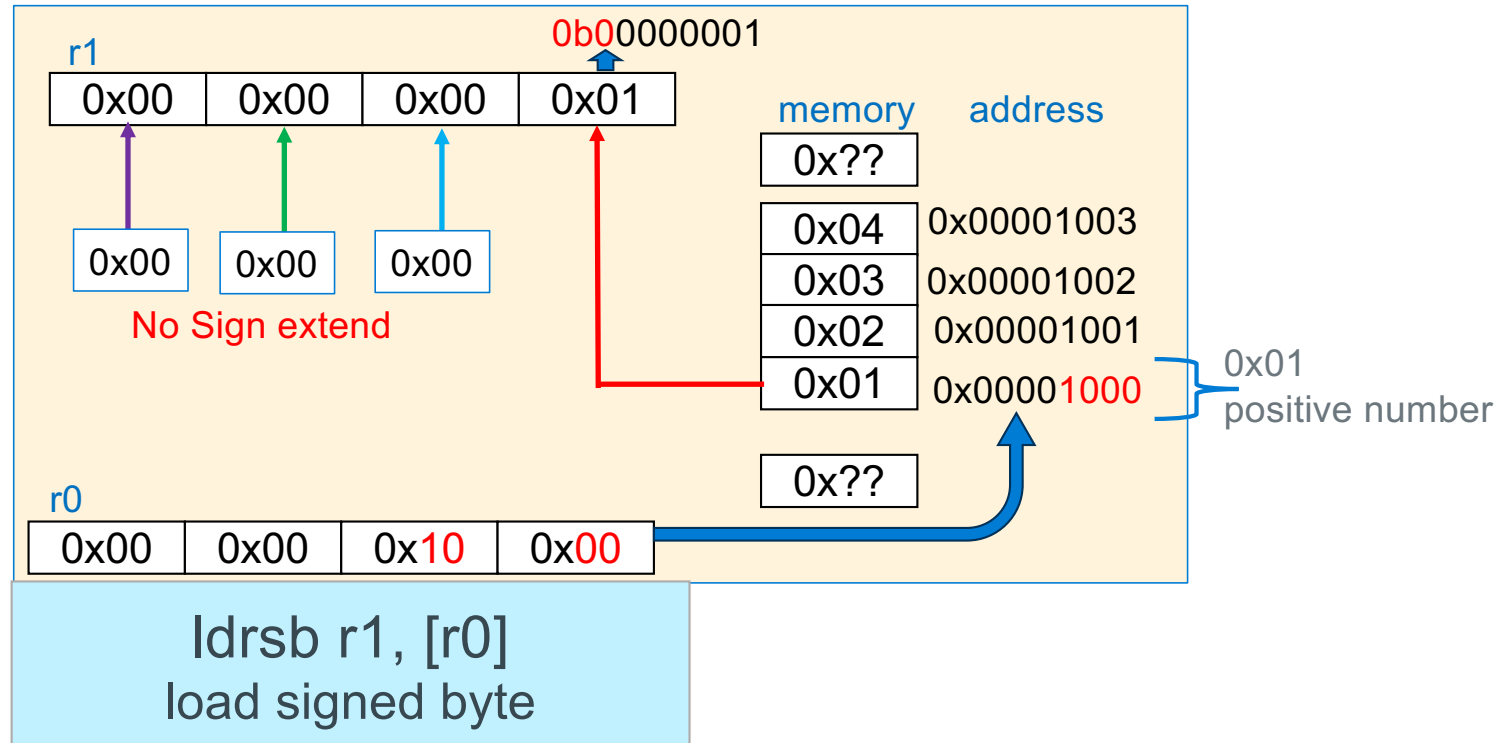


## Loading 32-bit Registers From Memory, 8-bit

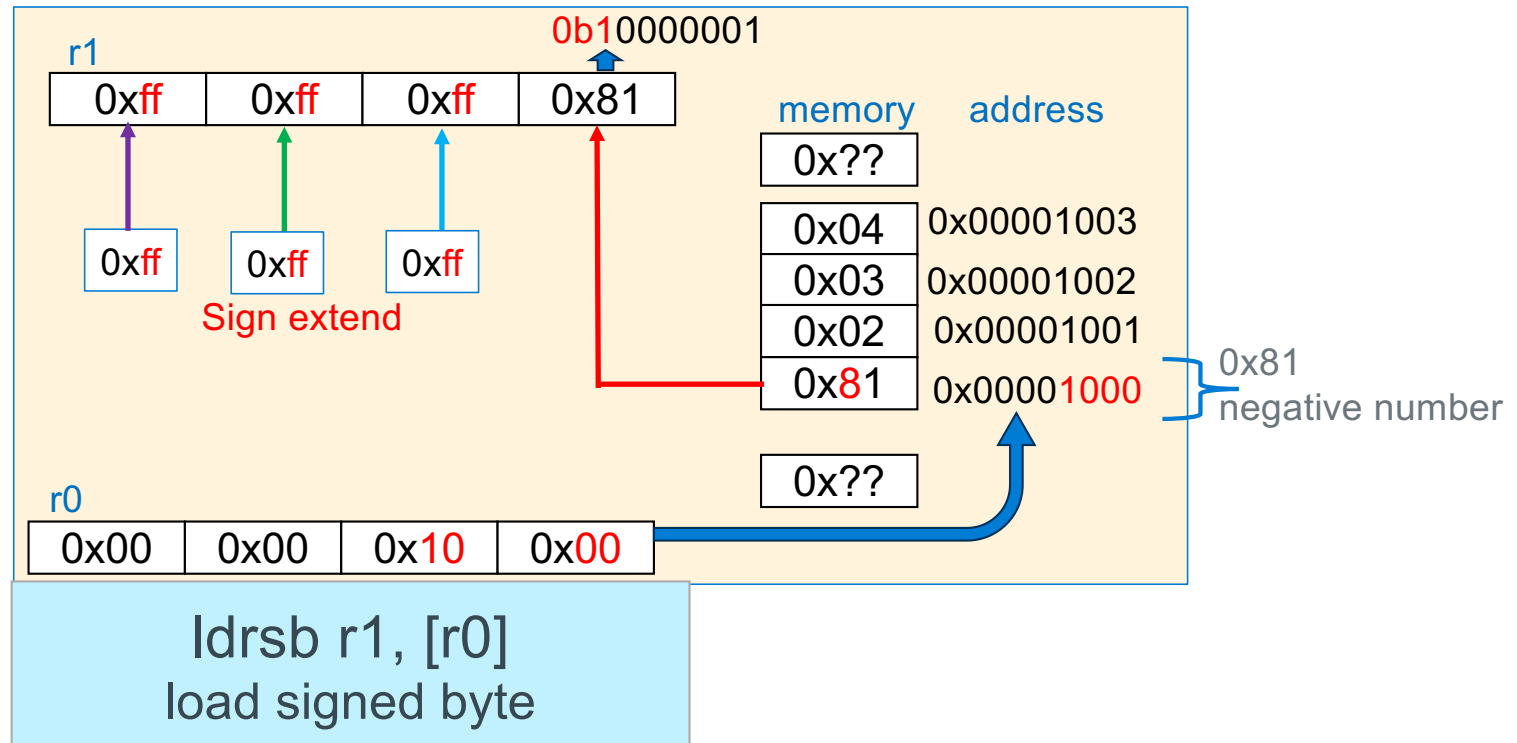




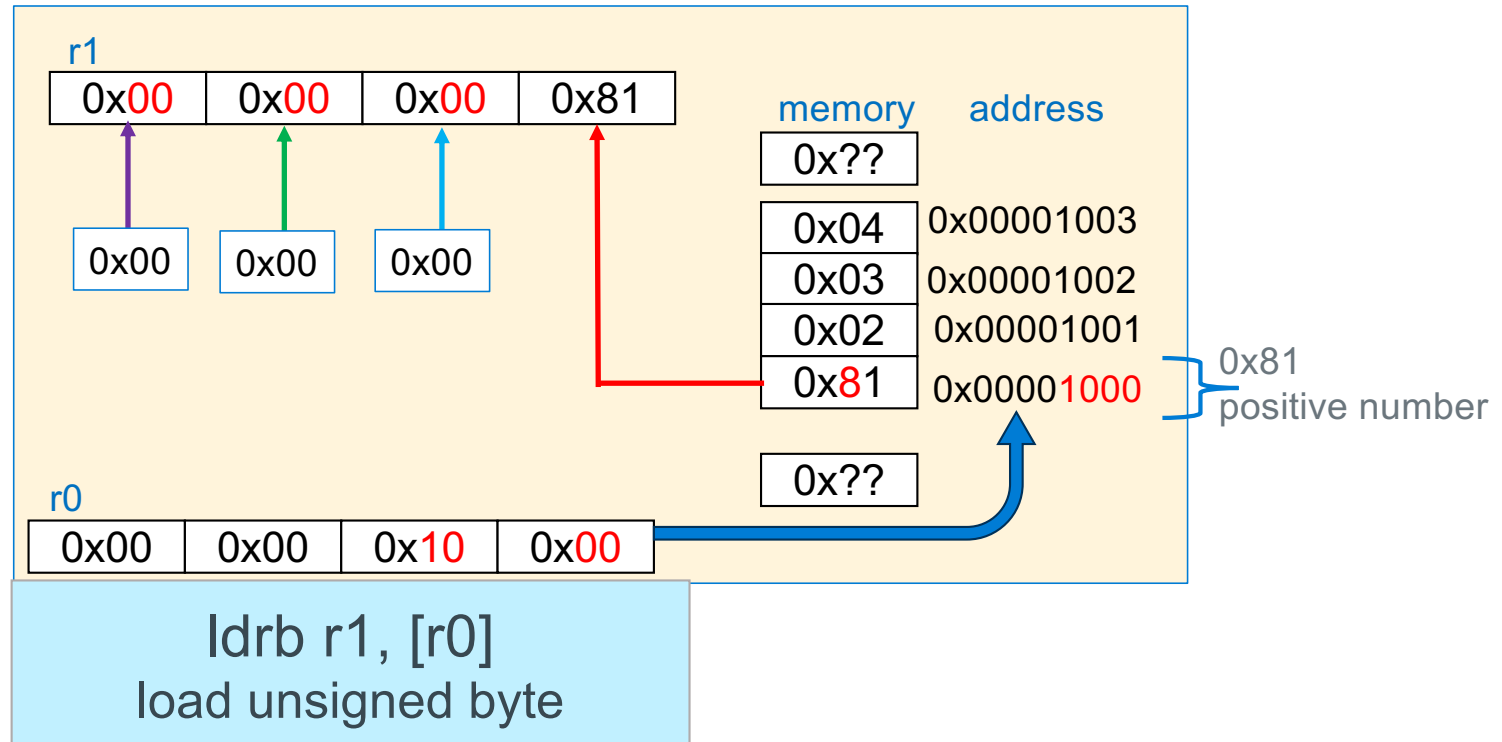
## Loading 32-bit Registers From Memory, 8-bit



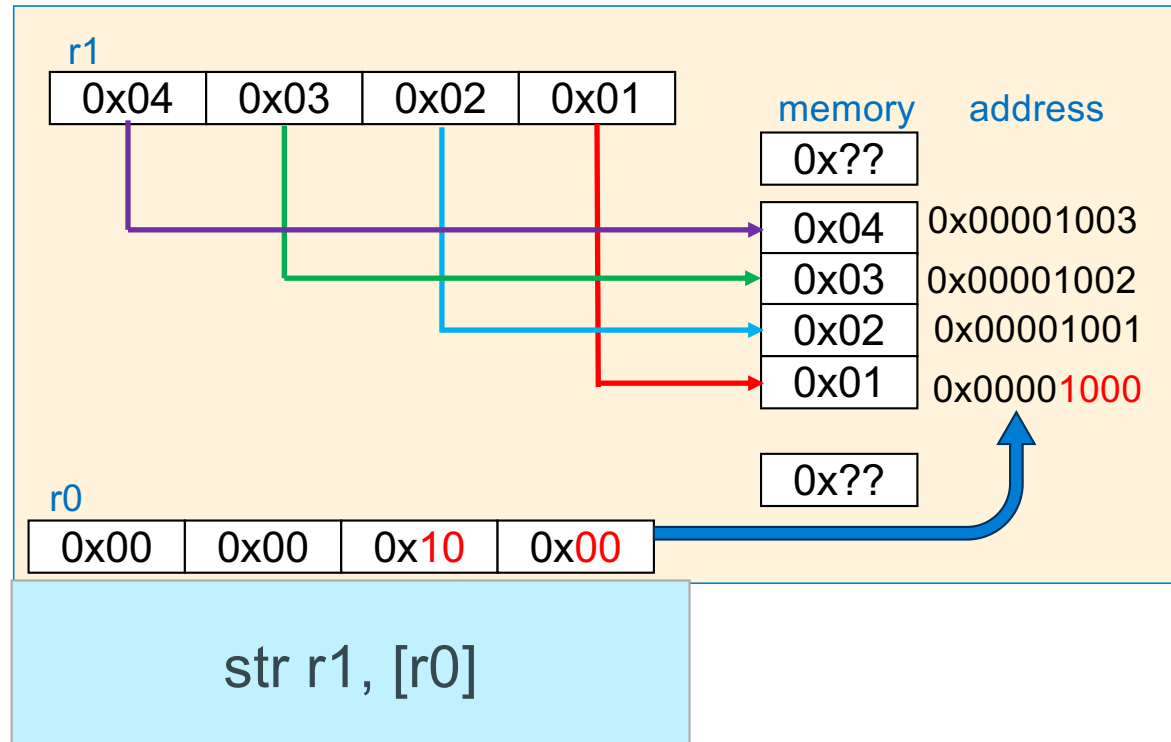
## Loading 32-bit Registers From Memory, 8-bit Signed



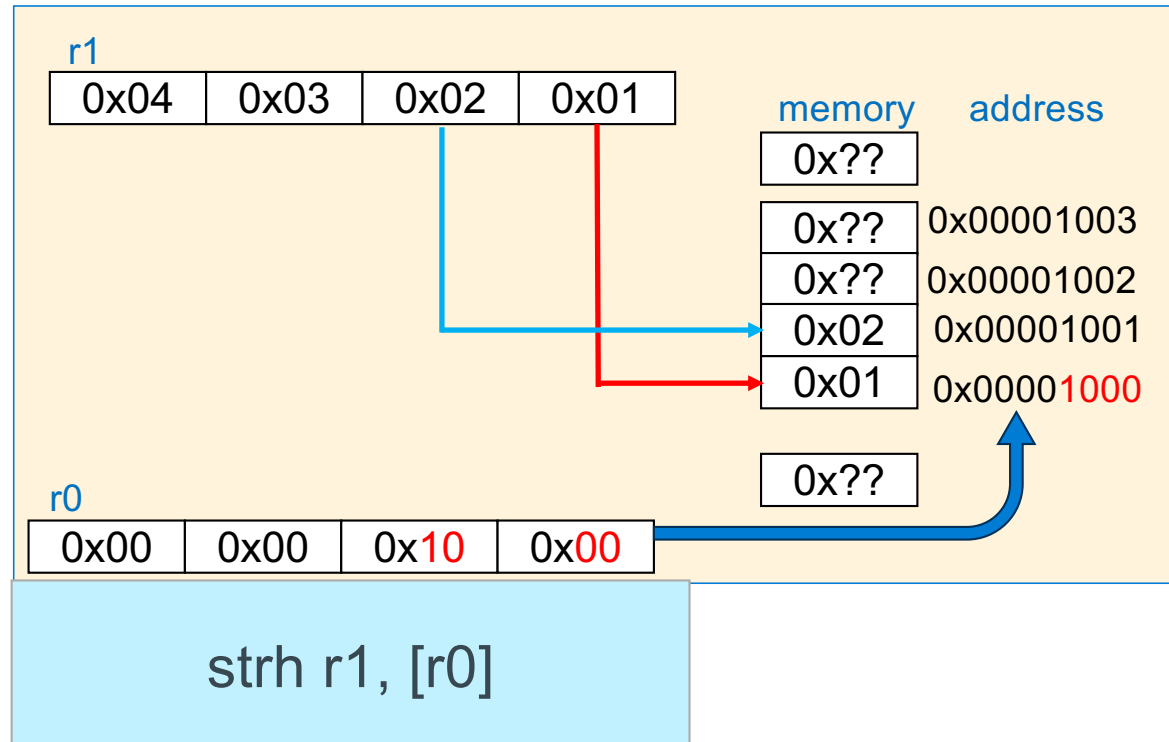
## Loading 32-bit Registers From Memory, 8-bit Signed



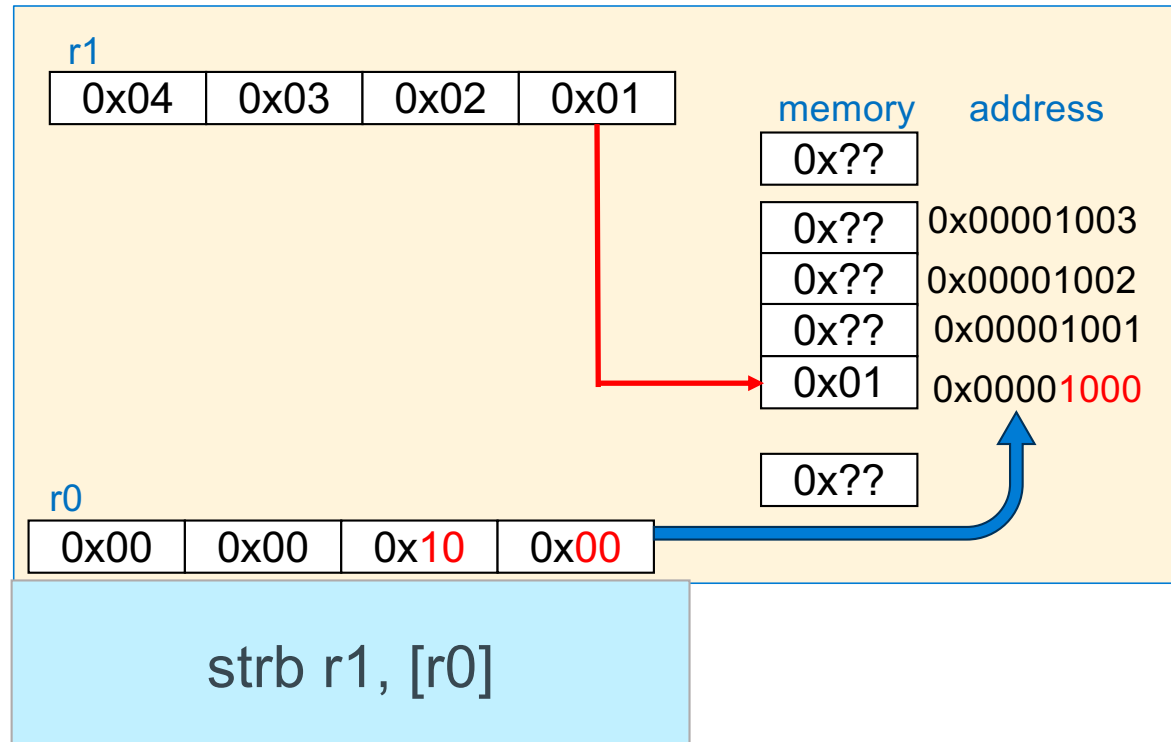
## Storing 32-bit Registers To Memory, 32-bit



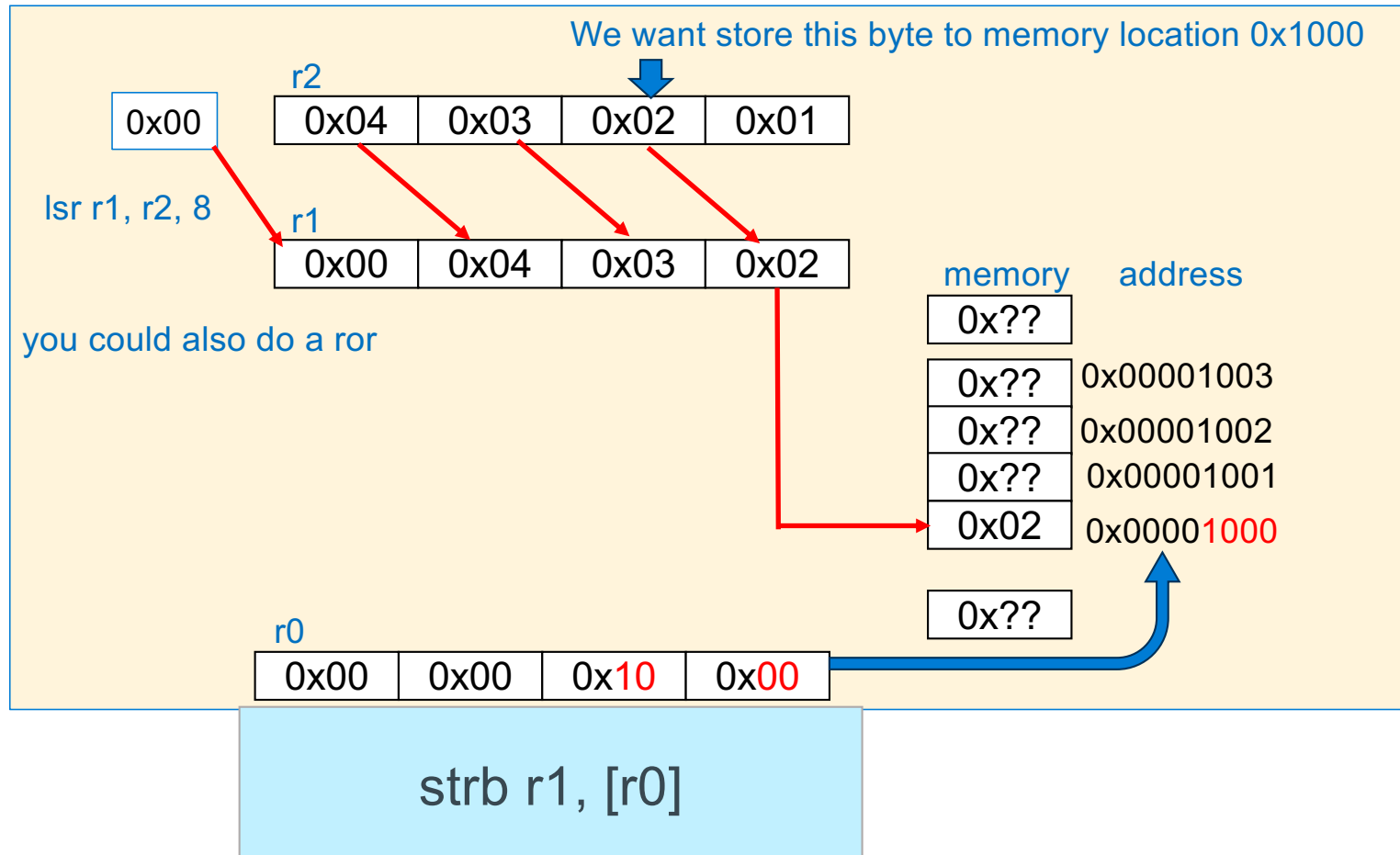
## Storing 32-bit Registers To Memory, 16-bit



## Storing 32-bit Registers To Memory, 8-bit



## Storing 32-bit Registers To Memory, 8-bit – Storing different byte



## using ldr/str: array copy

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6

void icpy(int *, int *, int);

int main(void)
{
    int  src[SZ] = {1, 2, 3, 4, 5, 6};
    int  dst[SZ];

    icpy(src, dst, SZ);
    for (int i = 0; i < SZ; i++)
        printf("%d\n", *(dst + i));

    return EXIT_SUCCESS;
}
```

```
void icpy(int *src, int *dst, int cnt)
{
    int *end = src + cnt;

    if (cnt <= 0)
        return;
    do {
        *dst++ = *src++;
    } while (src < end);
    return;
}
```



## Base Register version

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global icpy
.type icpy, %function
.equ FP_OFF, 12

// r0 contains int *src
// r1 contains int *dst
// r2 contains int cnt
// r3 use as loop term pointer
// r4 use as temp

icpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    // see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
.size icpy, (. - icpy)
.end
```

```
    cmp     r2, 0
    ble     .Ldone
    // pre loop guard

    lsl     r2, r2, 2 //convert cnt to int size
    add     r3, r0, r2 // loop term pointer

.Ldo:
    ldr     r4, [r0] // load from src
    str     r4, [r1] // store to dest

    add     r0, r0, 4 // src++
    add     r1, r1, 4 // dst++

    cmp     r0, r3 // src < term pointer?
    blt     .Ldo
    // loop guard

.Ldone:
```

## Base Register + Register Offset Version

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global icpy
.type icpy, %function
.equ FP_OFF, 12
// r0 contains int *src
// r1 contains int *dst
// r2 contains int cnt
// r3 use as loop counter
// r4 use as temp
```

```
icpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    // see right ->
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx     lr
    .size icpy, (. - cpy)
.end
```

```
    cmp     r2, 0
    ble     .Ldone
    lsl     r2, r2, 2
    mov     r3, 0
.Ldo:
    ldr     r4, [r0, r3]
    str     r4, [r1, r3]
    add     r3, r3, 4
    cmp     r3, r2
    blt     .Ldo
.Ldone:
```

pre loop guard

loop guard

one increment  
covers both arrays

## Base Register + Register Offset With chars

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6
void cpy(char *, char *, int);
int main(void)
{
    char src[SZ] =
        {'a', 'b', 'c', 'd', 'e', '\0'};
    char dst[SZ];

    cpy(src, dst, SZ);
    printf("%s\n", dst);
    return EXIT_SUCCESS;
}
```

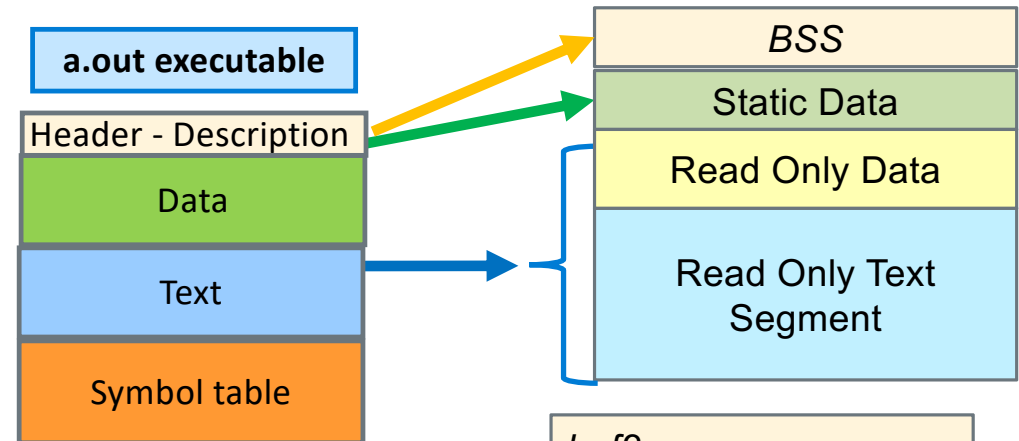
```
    cmp    r2, 0
    ble    .Ldone

    mov    r3, 0           // initialize counter
.Ldo:
    ldrb    r4, [r0, r3]   // load from src
    strb    r4, [r1, r3]   // store to dest
    add     r3, r3, 1       // counter++
    cmp     r3, r2         // count < r3
    blt     .Ldo

.Ldone:
```

# What is the conceptual difference between .bss and .data?

- All static variables that do not specify an initial value default to an initial value of 0 and are placed in .bss segment
- To save file system space in the executable file (the a.out file) the assembler collapses these .bss variables to a location and size "table"
- .data segment variables use the same space in the executable file as they have in memory
- .section .rodata is handled the same as .data



```
// these are .bss variables
int buf1[4096];
int buf2[4096];
```

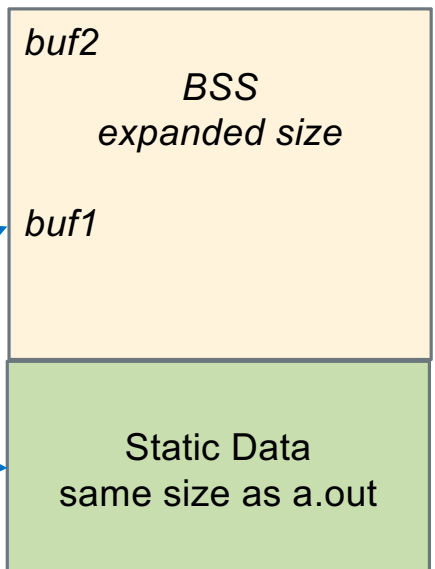
just big enough for address, size

```
// these are .data variables
int table[] = {1,2};
char string[] = "CSE30!!";
```

same size as specified

buf2 address size			
buf1 address size			
2			
1			
'\0'	'!'	'!'	'0'
'3'	'E'	'S'	'c'

executable file



low main mameory

# Variable Alignment In .data, .bss and .section .rodata

Use .align directive to force the assembler to align the address of the next variable defined after the .align

char

1

any address

short

2 bytes

addresses that end in 0b0

integer

4 bytes

addresses that end in 0b00

SIZE Alignment Requirements	Starting Address must end in	Align Directive
8-bit char -1 byte	0b..0 or 0b..1	
16-bit int -2 bytes	0b..0	.align 1
32-bit int -4 bytes pointers, all arrays	0b..00	.align 2



# Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	.align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		char chx = 'A'; char string[] = {'A','B','C', 0};	chx: .byte 'A' string: .byte 'A','B',0x42,0
16-bit int (2 bytes)	.short	.align 1	short length = 0x55aa;	length: .short 0x55aa
32-bit int (4 bytes)	.word .long	.align 2	int dist = 5; int *distptr = &dist; unsigned int mask = 0xaa55; int array[] = {12,~0x1,0xCD,-1};	dist: .word 5 distptr: .word dist mask: .word 0xaa55 array: .word 12,~0x1,0xCD,-3
string with '\0'	.string		char class[] = "cse30";	class: .string "cse30"

Rule: Place the .align above the variable

.align 1

len: .short 0x55aa

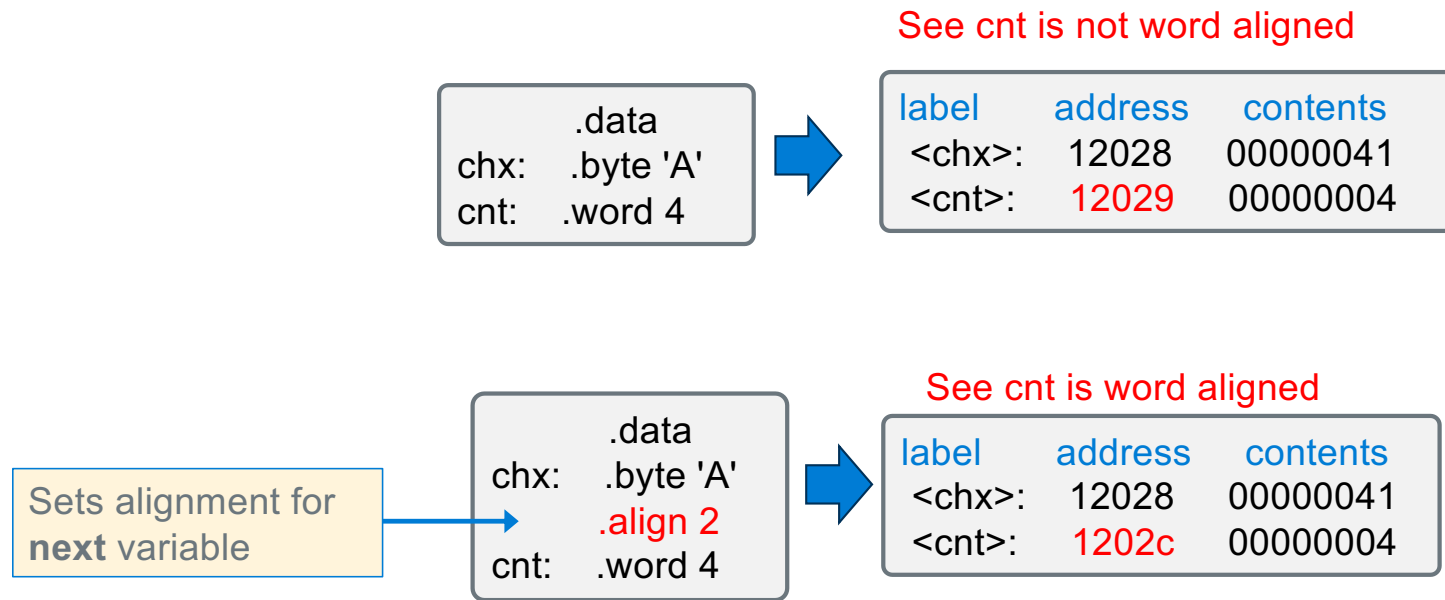
Rule: use .align 2 before every array regardless of type

Rule: place variables with explicit initialized values in a .data segment

Rule: place variables with no explicit initiali value (default to 0) in .bss segment

Rule: place string literals in .section .rodata and use a local label (.Llabel:)

## Defining Static Variables: Why the .align?



## Defining Static variables

```
// format: <var_name> is the address, <value> is the initial value
    var_name: <directive> <value>, <value>, ...

// Use regular labels for all <var_name> if anonymous use local labels .Llabel
```

```
.bss
// put all static variables without an explicit initial value here
// until another section directive is seen everything from this point is in .bss
// format: the value field if specified must be zero in .bss
.align 2
count: .word 0
buf:   .size 400      // int buf[100];

.data
put all static variables with an explicit initial value here
.align 2
array: .word 1, 2, 3, 4 // int array[] = {1, 2, 3, 4};

.section .rodata
// put all immutable string literals here variables
.align 2
.Lmess: .string "count is %d size is %d\n" // for a printf
```



# Defining Static Array Variables (large Arrays)

Label: `.space <size>, <fill>`

`.space size, fill`

- Allocates `size` bytes, each of which contain the value `fill`
- If the comma and `fill` are **omitted**, `fill` is assumed to be **zero**
- if used in `.bss` section: Must be used **without a specified fill**

```
.bss
int_buf: .space 400 // int int_buf[100];
        .align 2
char_buf: .space 100 // char char_buf[100];
        .data
one_buf:  .space 100, 1 // 100 bytes each byte filled with 1
```

## Loading Static variables into a register

- Tell the assembler load the address (Lvalue) of a label into a register:  
`ldr Rd, =Label // Rd = address`
- Tell the assembler load the contents into a register
- `ldr R0, [Rd] // Rd = address`
- *Example to the right:  $y = x$ ;*

**load** a static **memory** variable

1. load the pointer to the memory
2. read (load) from \*pointer

**store** to a static **memory** variable

1. load the pointer to the memory
2. write (store) to \*pointer

```
.bss
y: .space 4

.data
x: .word 200

.text
// function header
main:

// load the address, then contents
// using r2

ldr r2, =x      // int *r2 = &x
ldr r2, [r2]     // r2 = *r2;

// &x was only needed once above
// Note: r2 was a pointer then an int
// no "type" checking in assembly!

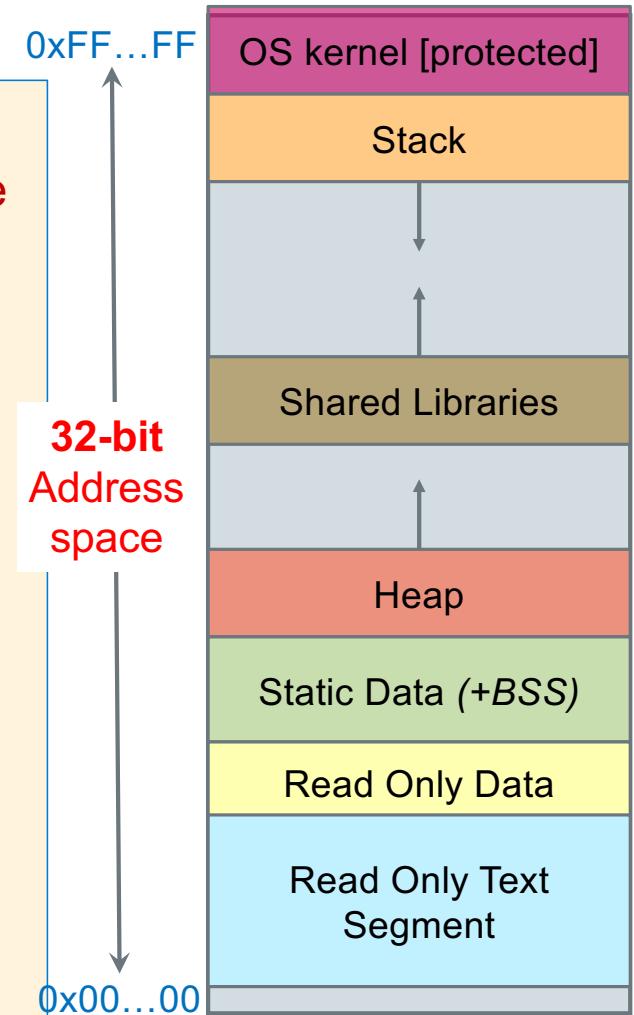
// store the contents of r2

ldr r1, =y      // int *r1 = &y
str r2, [r1]    // *r1 = r2
```

x

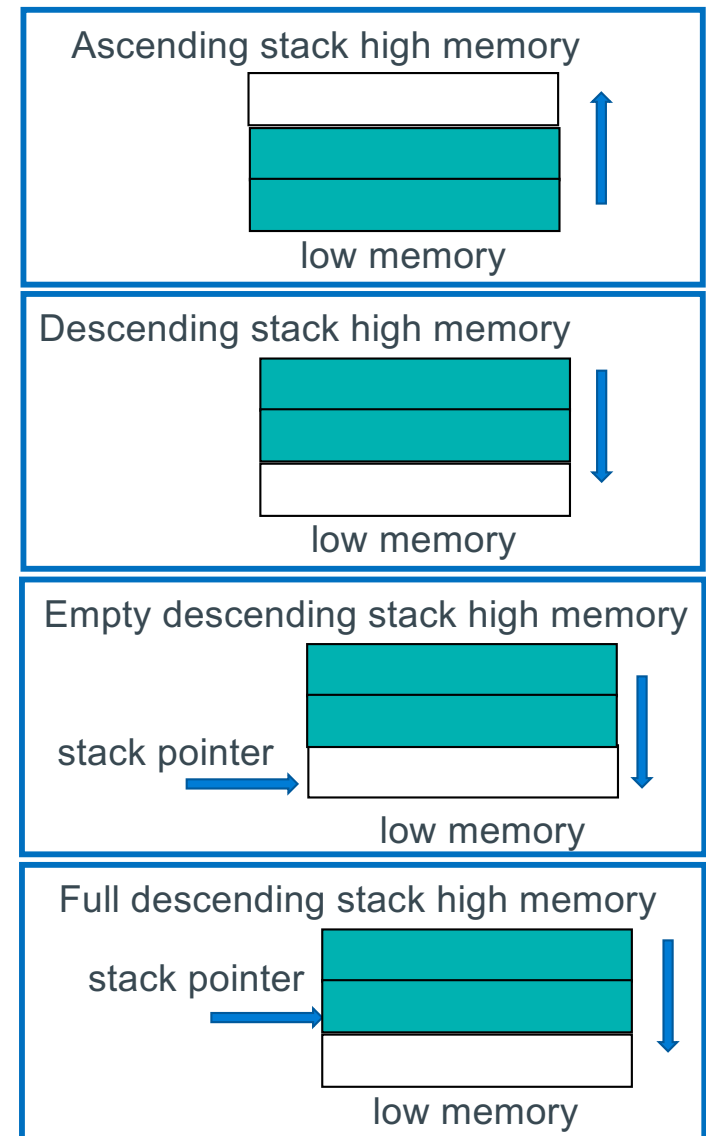
# Stack Segment: Support of Functions

- The stack consists of a series of "*stack frames*" or "*activation frames*", one is **created** each time a function is called at runtime
- Each frame represents a function that is currently being executed and has not yet completed (why activation frame)
- A function's stack "frame" goes away when the function returns
- Specifically, a new stack frame is
  - allocated (**pushed** on the stack) for each function call (**contents are not implicitly zeroed**)
  - deallocated (**popped** from the stack) on function return
- **Stack frame** contains:
  - Local variables, parameters of function called
  - Where to return to which caller when the function completes (the return address)



# Stack types

- A Stack Implements a **last-in first-out (LIFO)** protocol
- Each time a **function is called**, a **stack frame is activated**
  - space is allocated by moving the stack pointer
  - push adds space, pop removes space
- Stack growth direction
  - **Ascending stack**: grows from low memory towards high memory (adding to the sp to allocate memory)
  - **Descending stack**: grows from high memory towards low memory (subtracting from the sp to allocate memory)
- Full versus empty stacks
  - **Empty stack**: **stack pointer (sp)** points at the **next word address** after the last item pushed on the stack
  - **Full stack**: **stack pointer (sp)** points at the **last item pushed on the stack**
- ARM on Linux uses a **full descending stack**



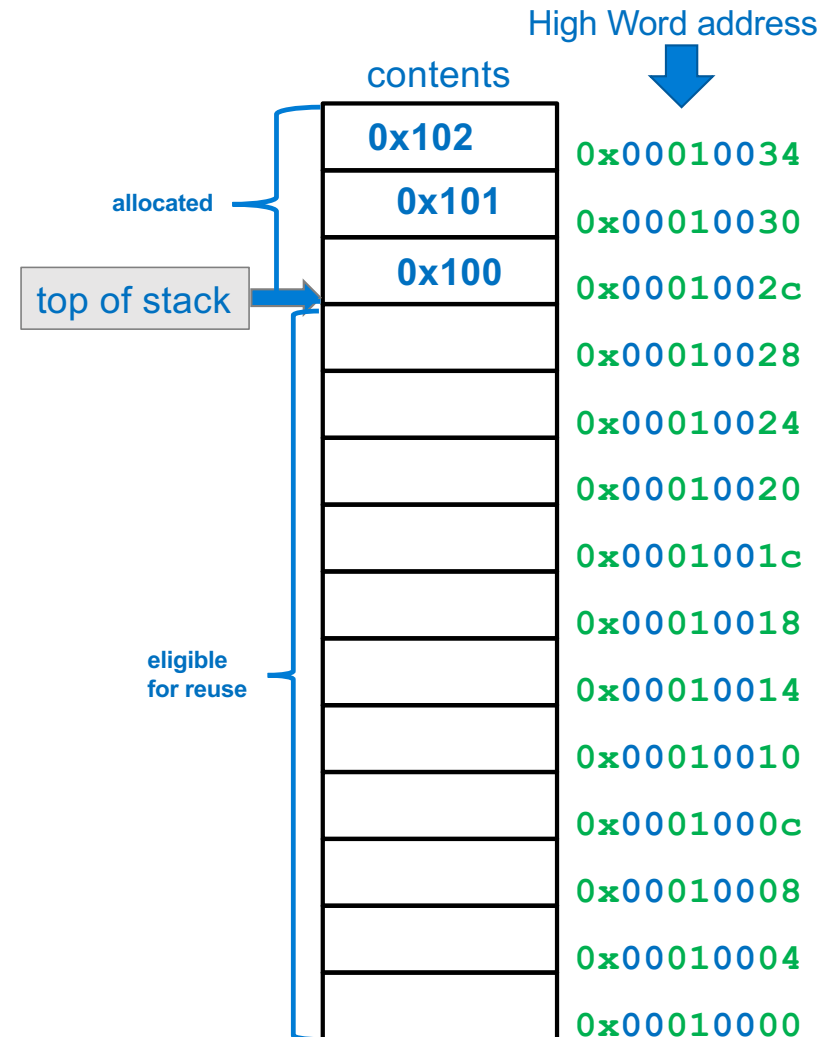
# Arm: Stack Operation

- **Stack** is expandable and grows downward from high memory address towards low memory address
- **Stack pointer (sp)** always points at the **top of stack**
  - contains the starting address of the top element
- New items are **pushed** (*added*) onto the **top of the stack** by **subtracting** from the stack pointer the **size of the element** and then writing the element

push (sp - element size) & write

- Existing items are **popped** (*removed*) from the top of the stack by **adding** to the stack pointer the **size of the element** (leaving the **old contents unchanged**)

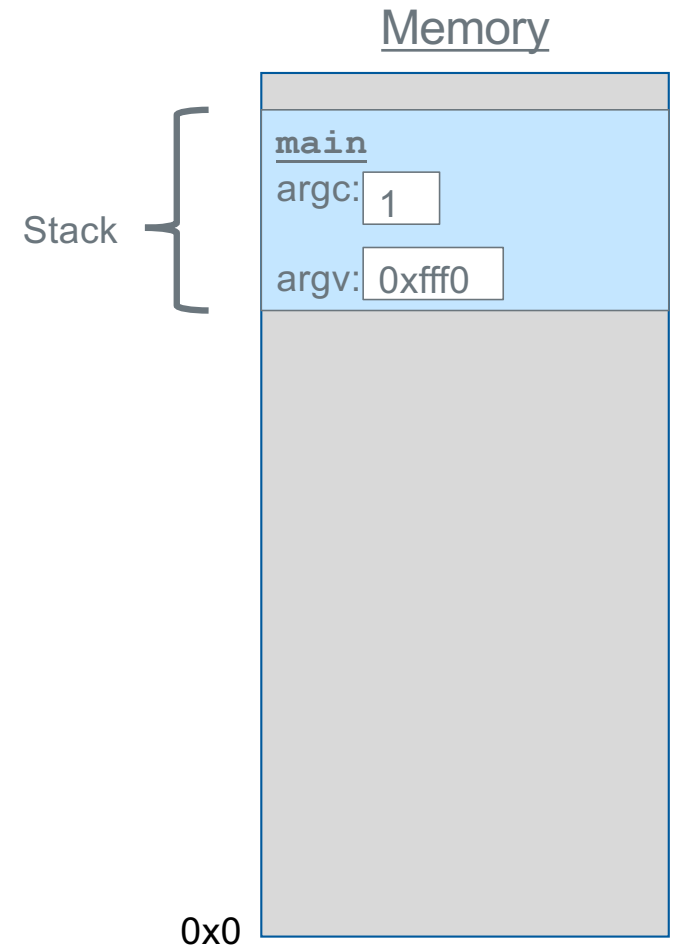
pop (sp + element size)



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables

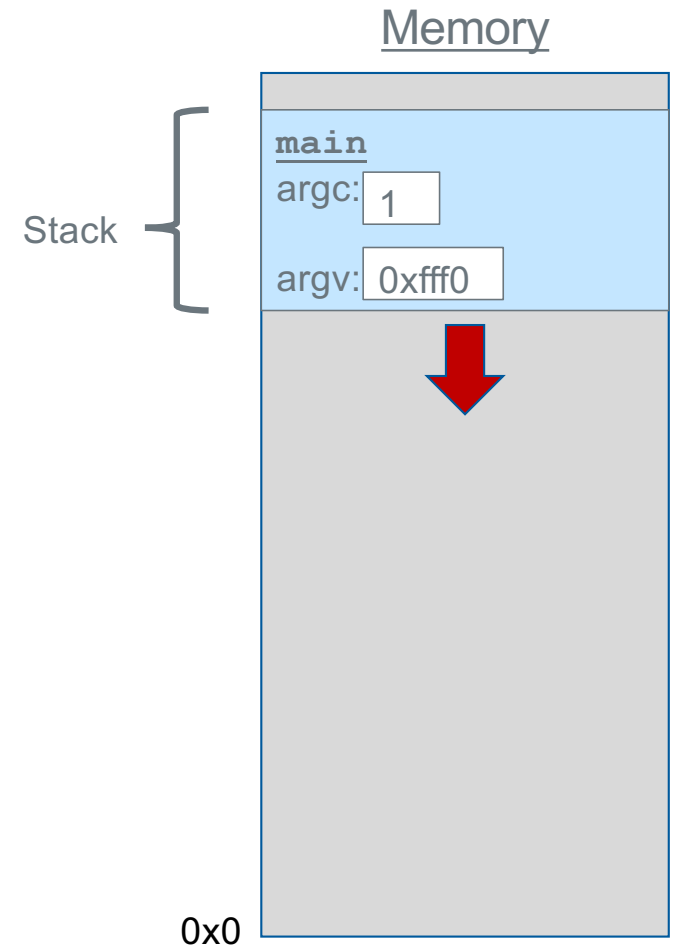
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

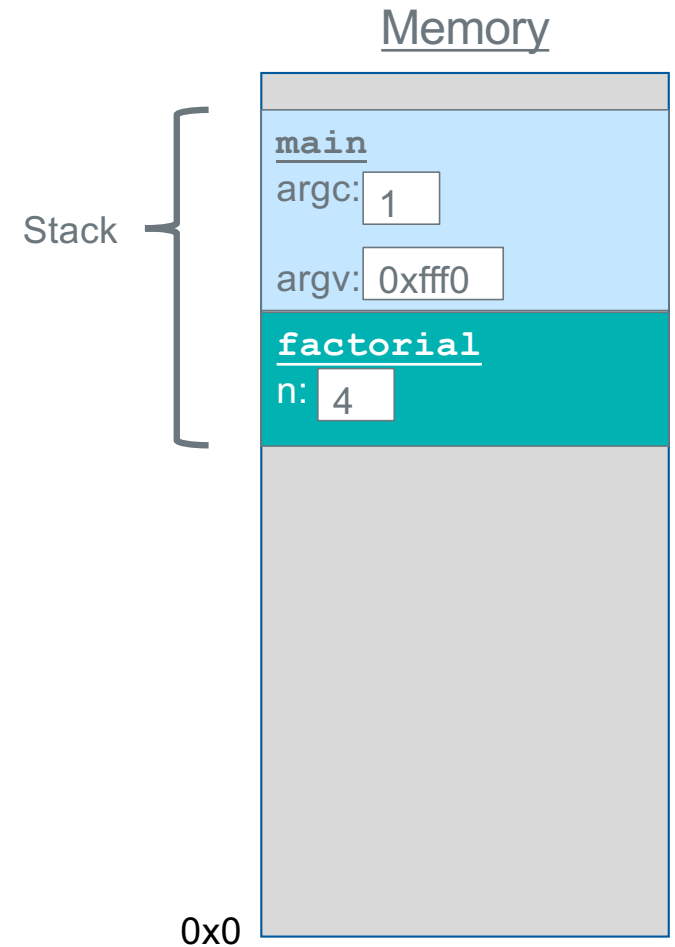
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```

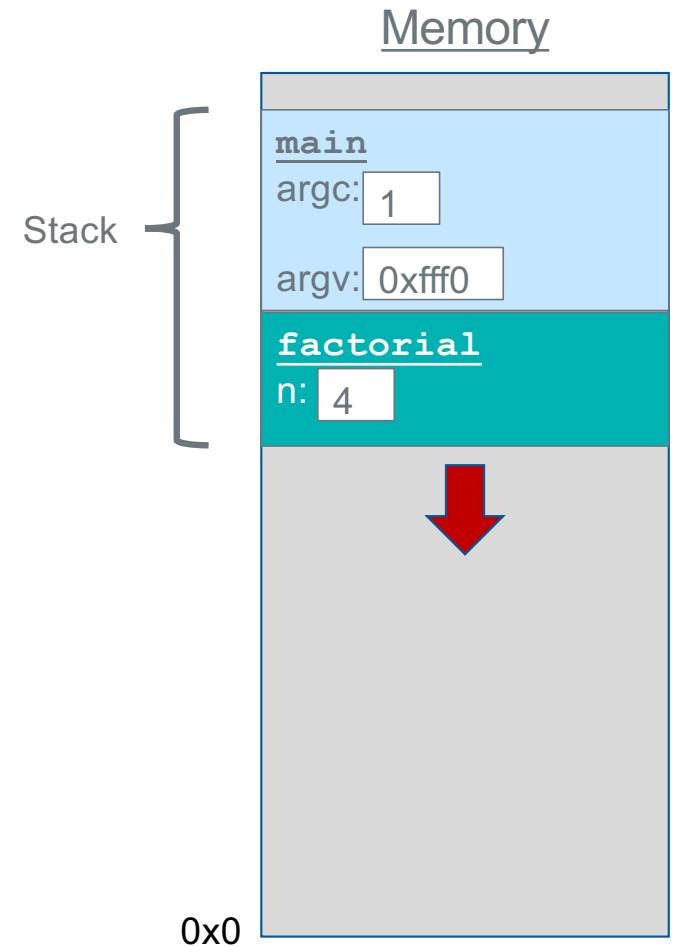




# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

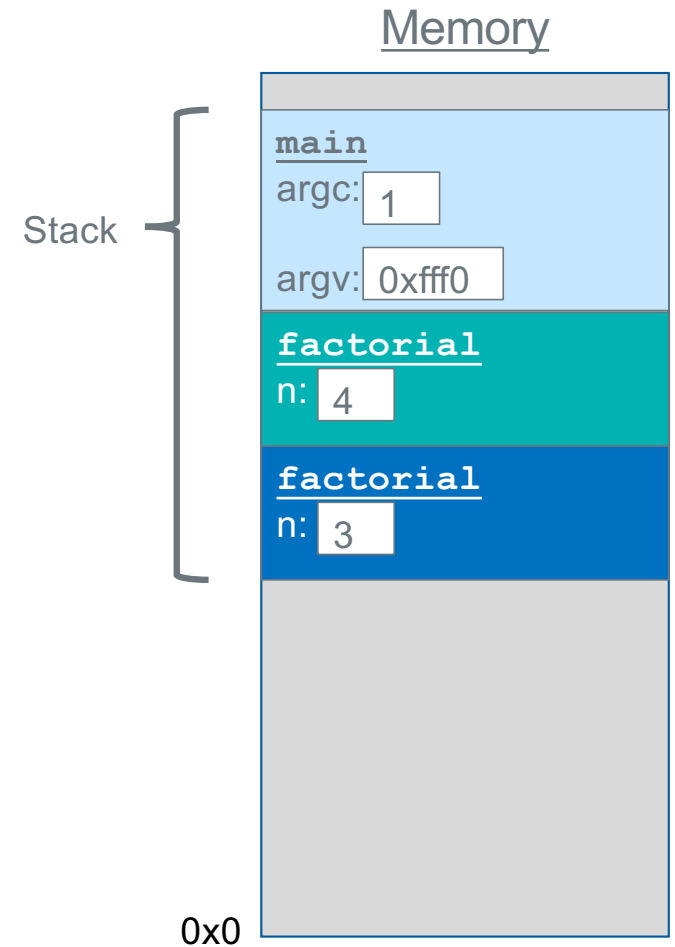
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

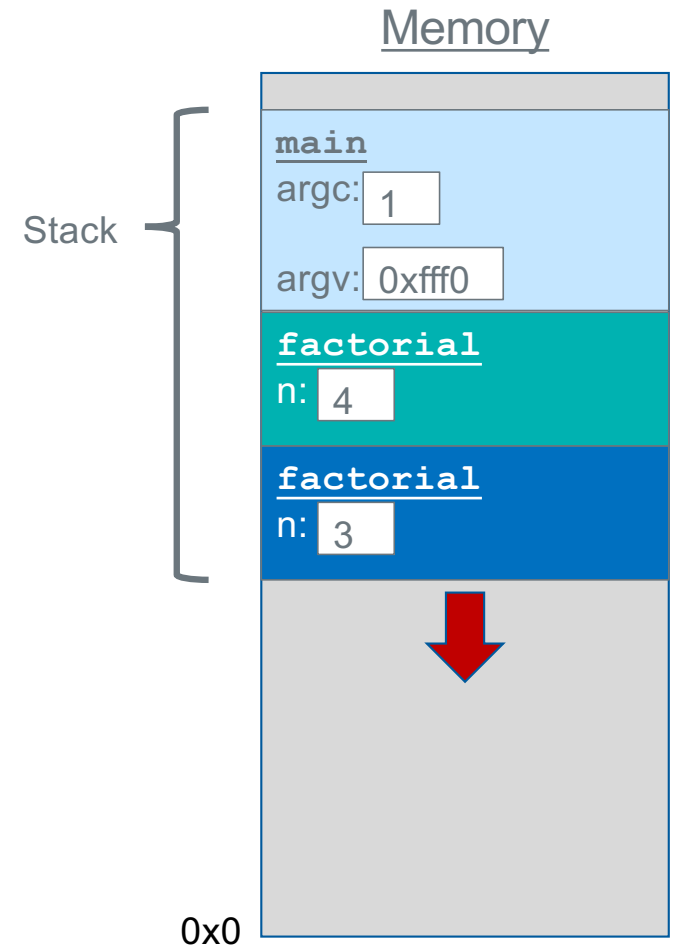
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

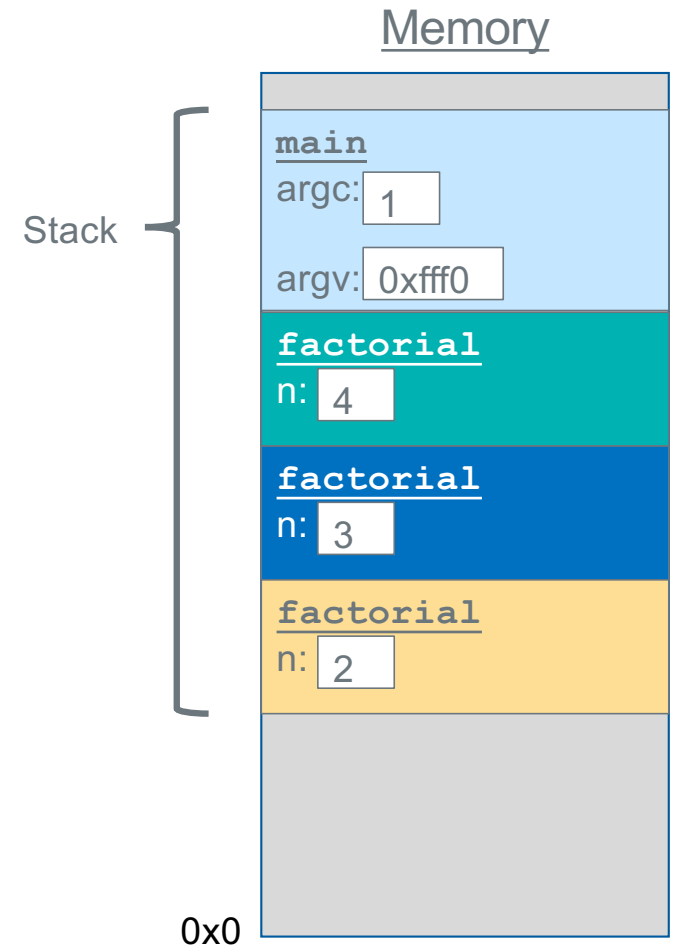
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

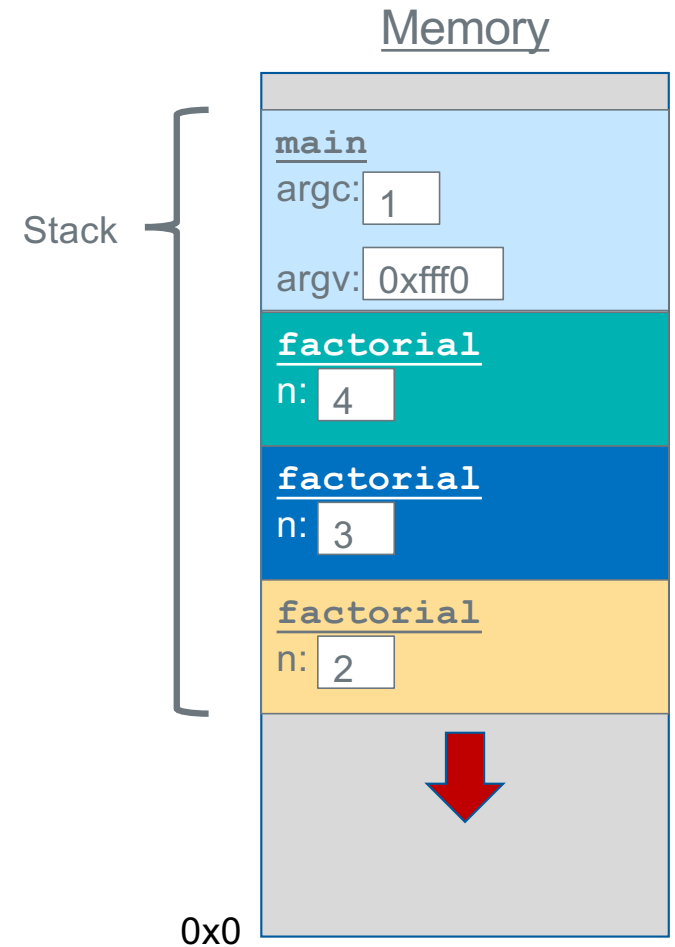
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

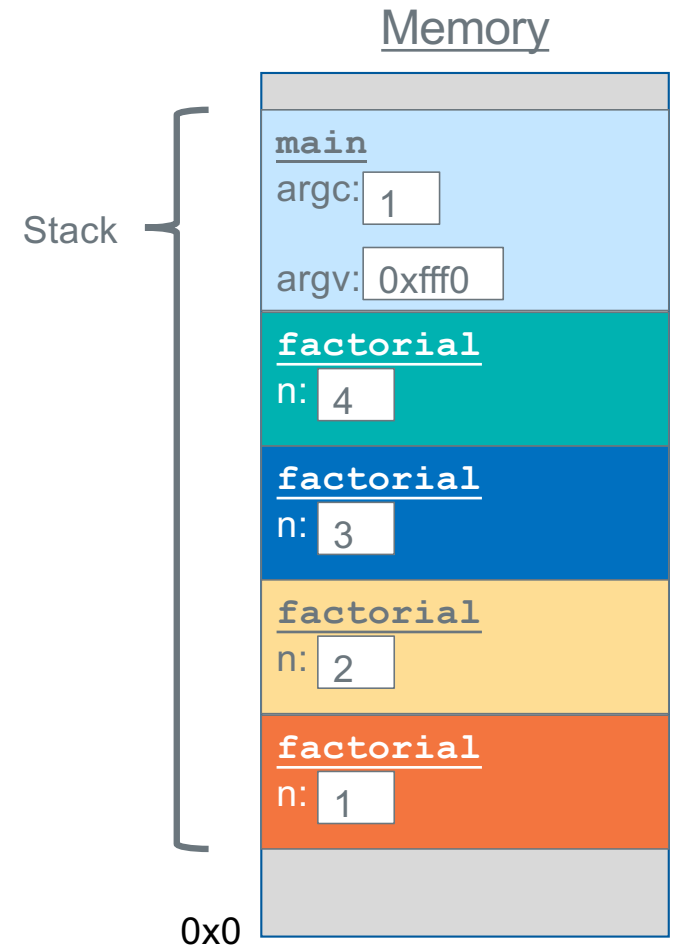
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

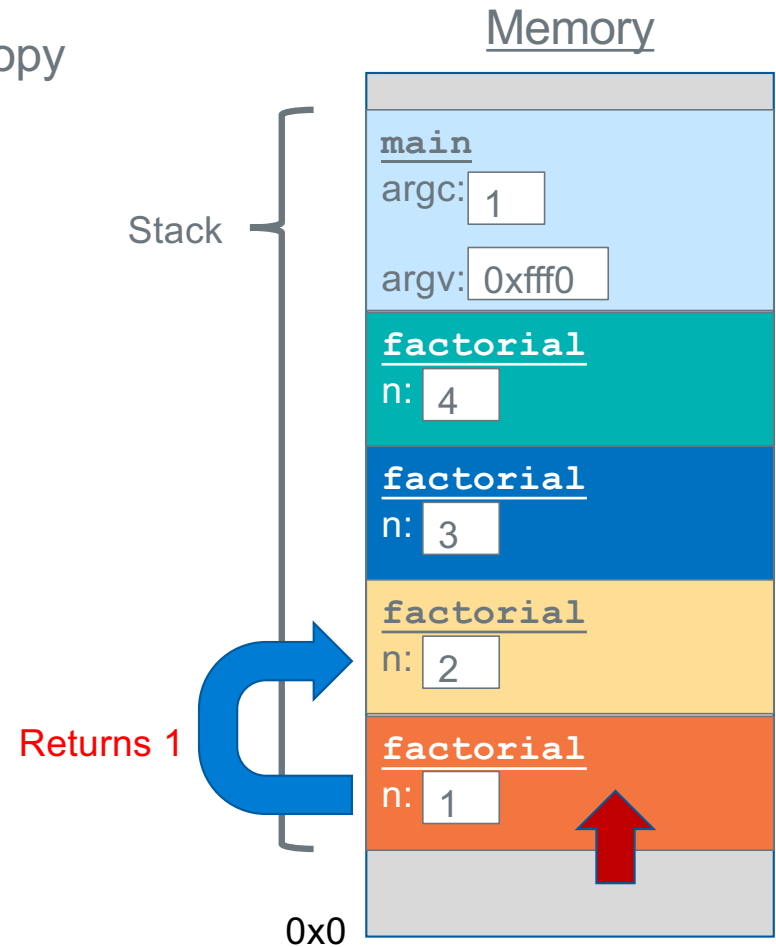
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

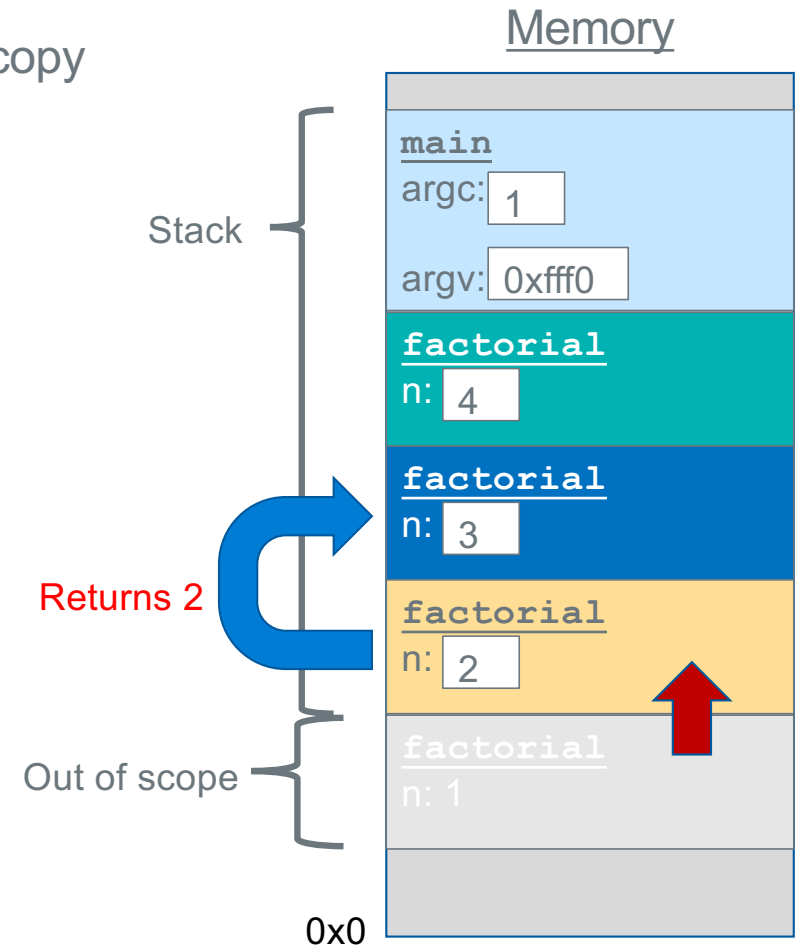
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```

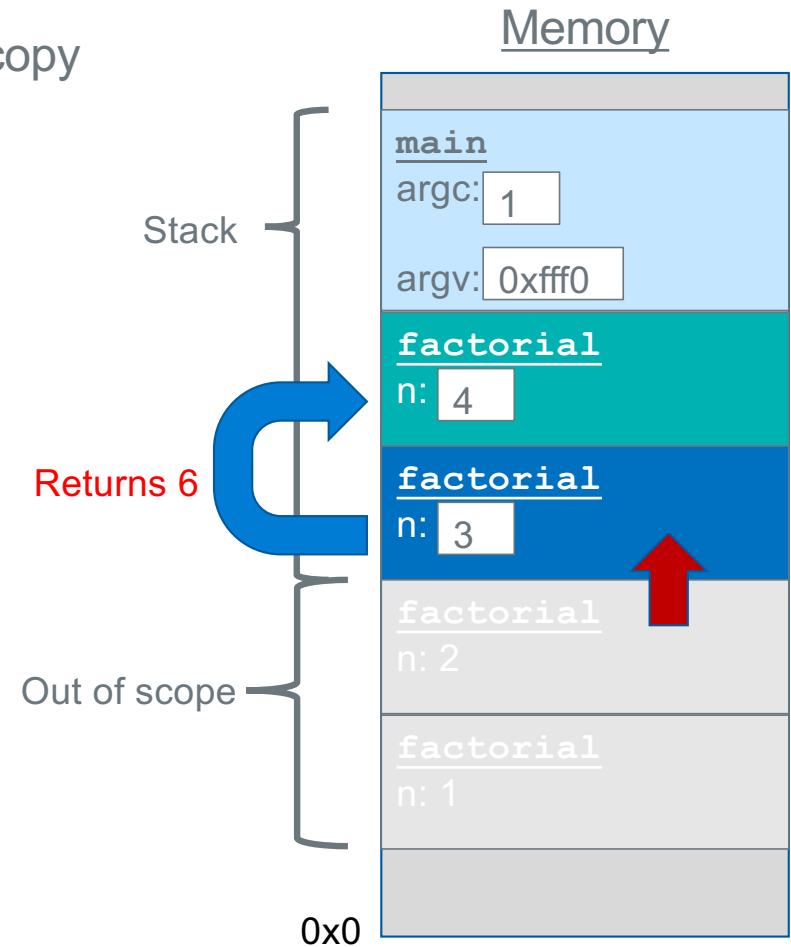




# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

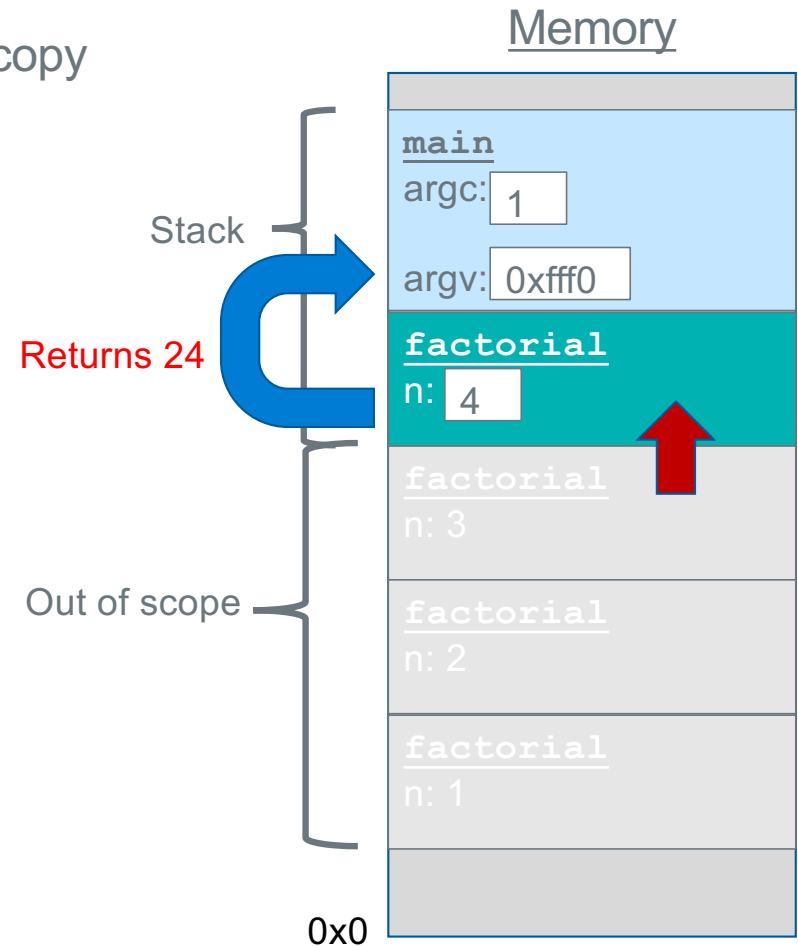
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

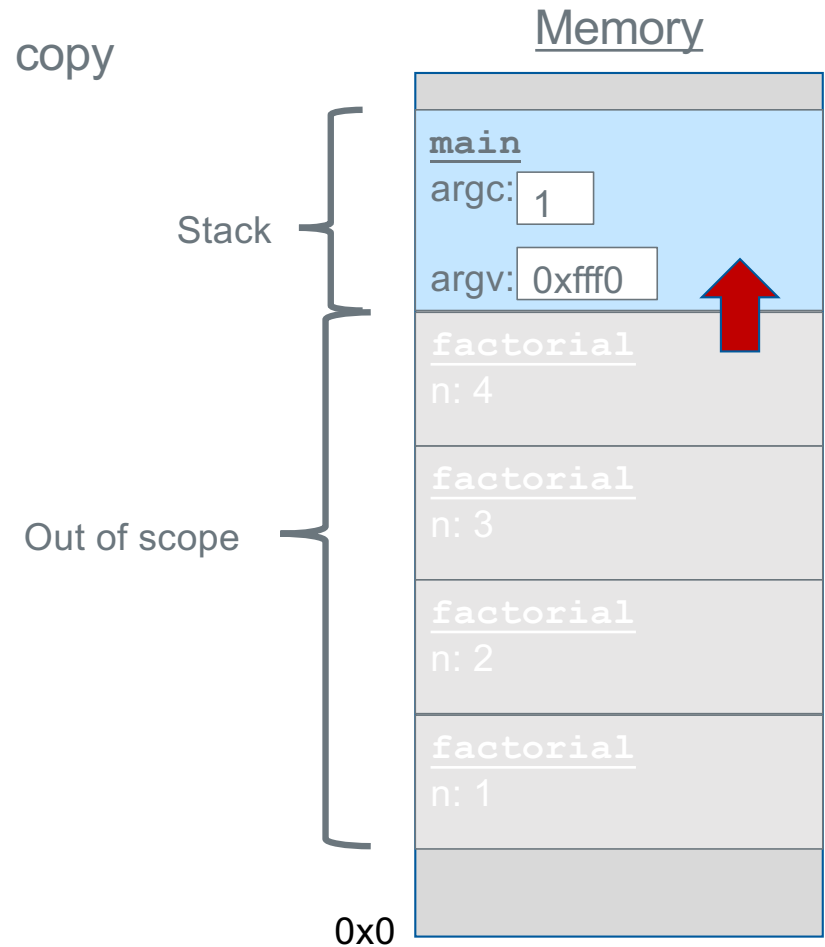
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

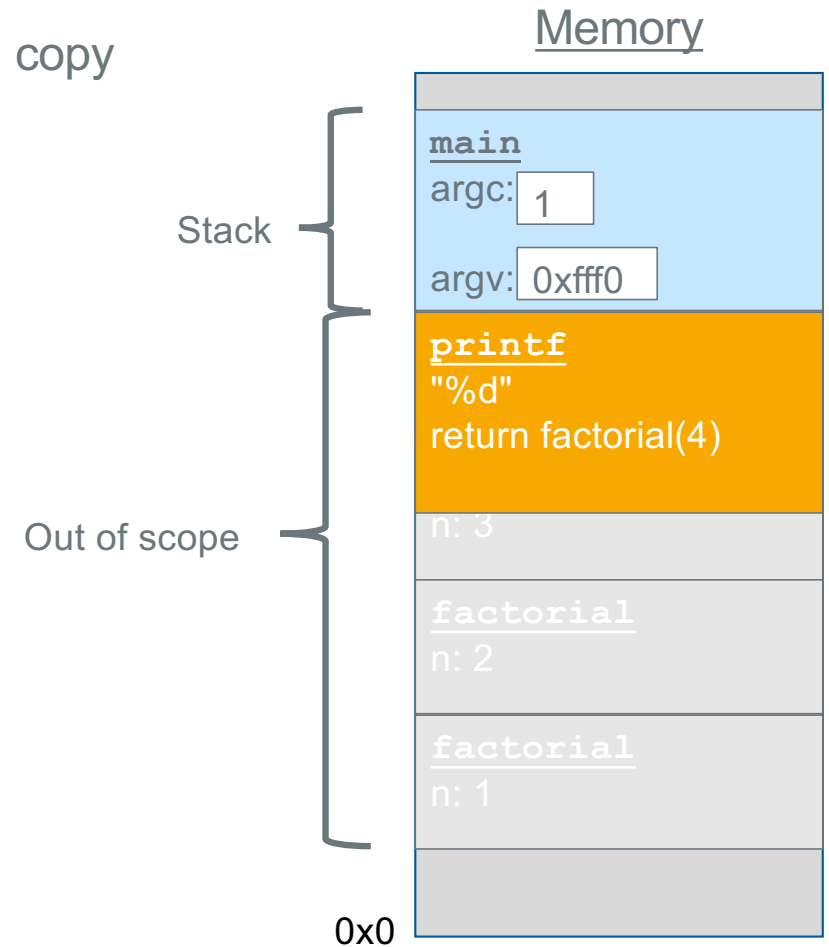
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# Function Calls

Branch with Link (function call) instruction

**bl** *label*

**bl**

**imm24**

- Function call to the instruction with the address *label* (no local labels for functions)
  - *imm24* number of instructions from pc+8 (24-bits)
  - *label* any function label in the current file, any function label that is defined as **.global** in any file that it is linked to, any C function that is not static

Branch with Link Indirect (function call) instruction

**blx** *Rm*

**blx**

**Rm**

- Function call to the instruction whose address is stored in Rm (Rm is a function pointer)
- **bl** and **blx** both save the address of the instruction immediately following the **bl** or **blx** instruction in register **lr** (link register is also known as r14)
- The contents of the link register is the return address in the calling function

- (1) Branch to the instruction with the label f1
- (2) copies the address of the instruction AFTER the bl in lr

main:

•

**bl** f1

→

f1:

•

•

# Function Call Return

Branch & exchange (function return) instruction

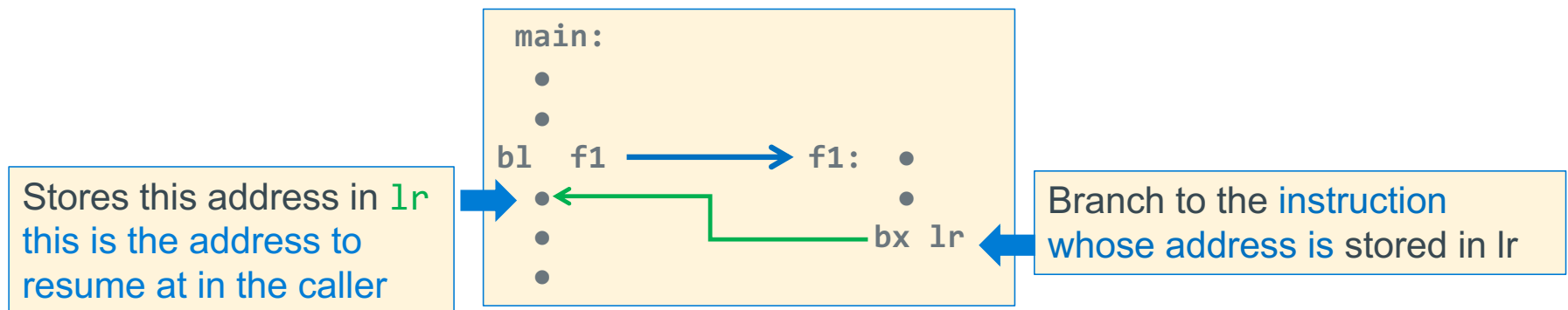
`bx lr`

`bx`

`Rn`

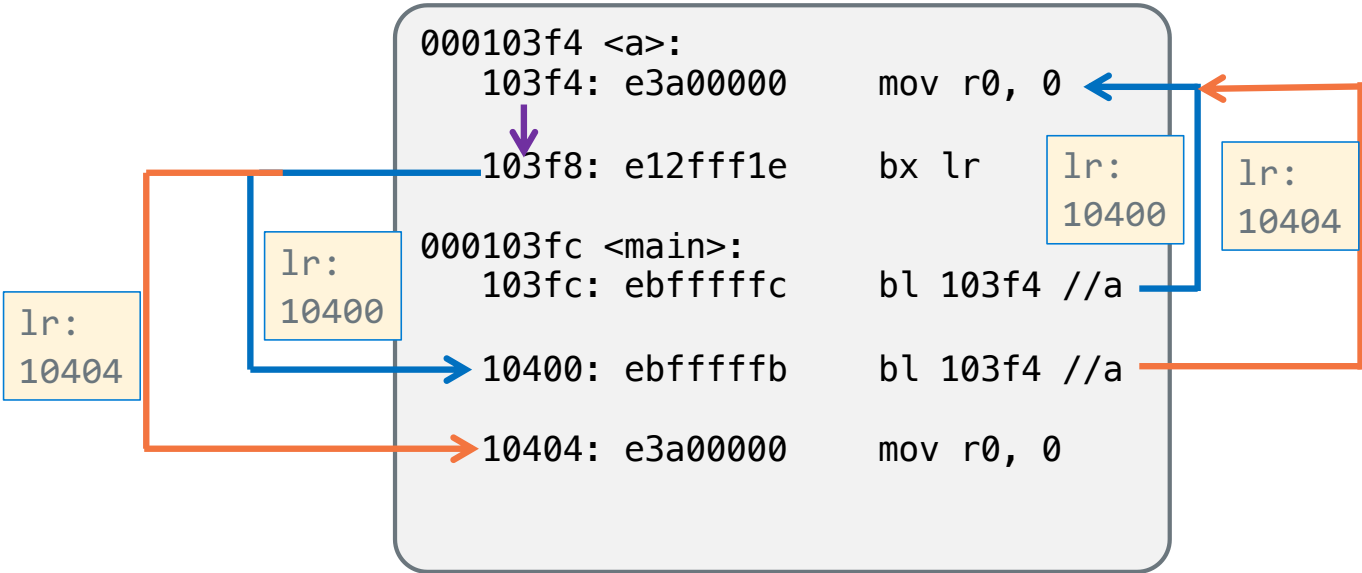
// we will always use `lr`

- Causes a branch to the instruction whose address is stored in register `<lr>`
  - It copies `lr` to the PC
- This is often used to implement a return from a function call (exactly like a C return) when the function is called using either `bl label`, or `blx Rm`



# Understanding bl and bx - 1

```
int a(void)
{
    return 0;
}
int main(void)
{
    a();
    a();
    // not shown
}
```

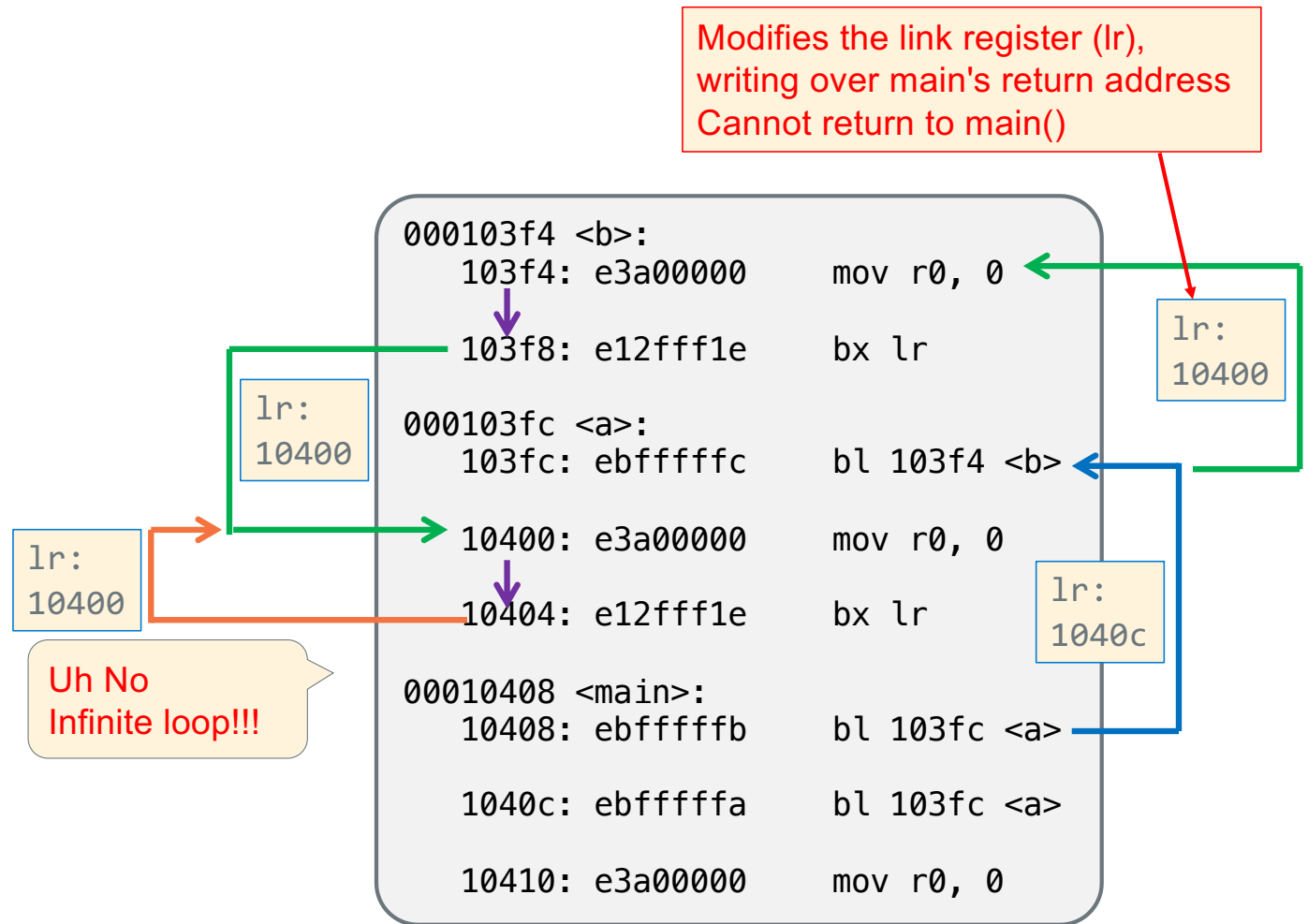


But there is a problem we must address here – next slide

## Understanding bl and bx - 2

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
    // not shown
}
```

We need to preserve the lr!





## Understanding bl and blx - 3

```
int a(void)
{
    return 0;
}

int (*func)() = a;

int main(void)
{
    (*func)();
    // not shown
}
```

But this has the same infinite loop problem when main() returns!

```
.data
func: .word a // func initialized with address of a()

.text
.global a
.type a, %function
.equ FP_OFF, 4

a:
    mov     r0, 0
    bx      lr
    .size a, (. - a)

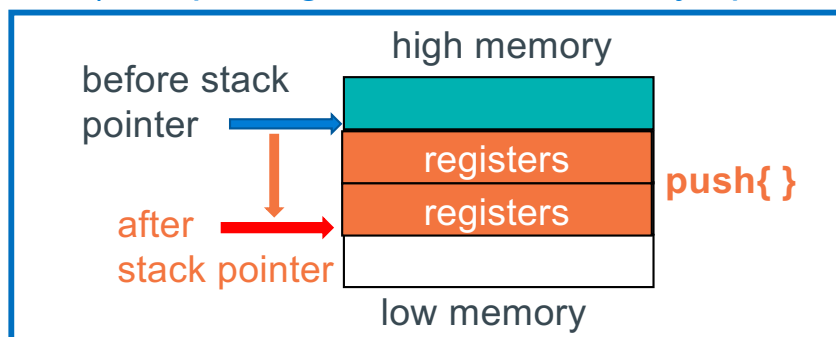
.global main
.type main, %function
.equ FP_OFF, 4

main:
    ldr     r4, =func // load address of func in r4
    ldr     r4, [r4]   // load contents of func in r4
    blx     r4         // we lose the lr for main!
    // not shown
    bx      lr         // infinite loop!
```

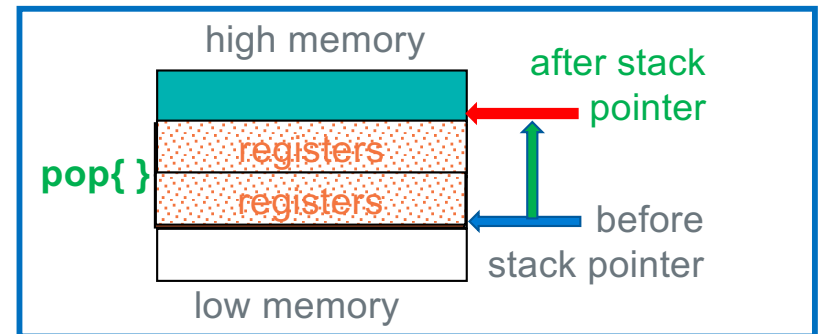
## Preserving and Restoring Registers on the stack - 1

Operation	Pseudo Instruction	Operation
Push registers Function Entry	<code>push {reg list}</code>	$sp = sp - 4 \times \text{\#registers}$ Copy registers to <code>mem[sp]</code>
Pop registers Function Exit	<code>pop {reg list}</code>	Copy <code>mem[sp]</code> to registers, $sp = sp + 4 \times \text{\#registers}$

push (multiple register **str** to memory operation)



push (multiple register **ldr** from memory operation)

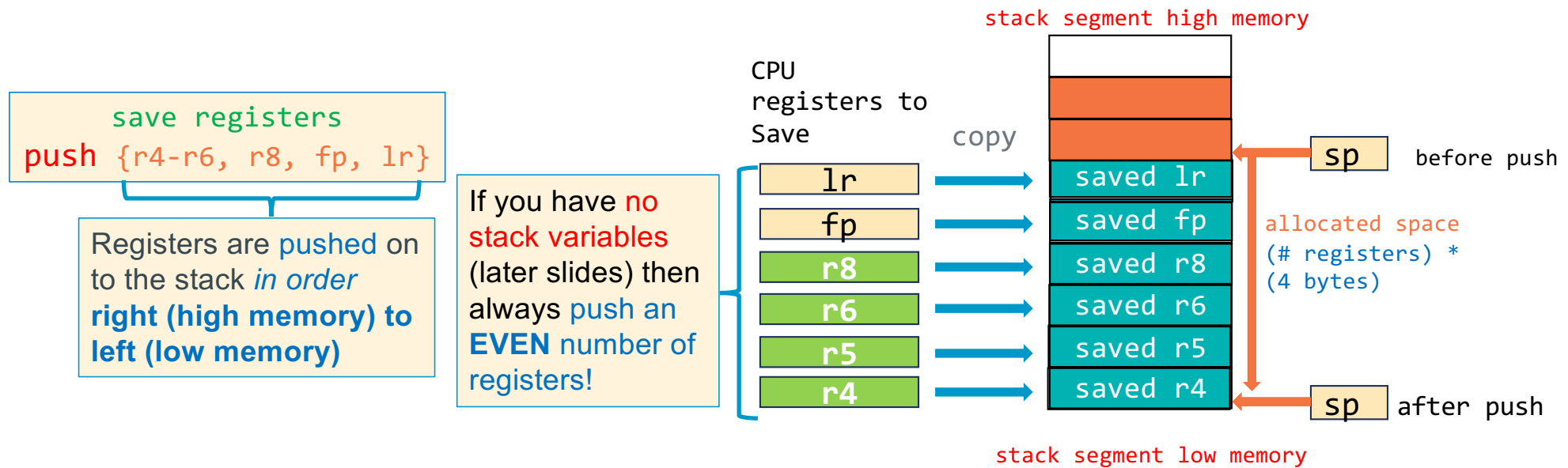


## Preserving and Restoring Registers on the Stack - 2

Operation	Pseudo Instruction	Operation
Push registers Function Entry	<code>push {reg list}</code>	$sp = sp - 4 \times \text{\#registers}$ Copy registers to <code>mem[sp]</code>
Pop registers Function Exit	<code>pop {reg list}</code>	Copy <code>mem[sp]</code> to registers, $sp = sp + 4 \times \text{\#registers}$

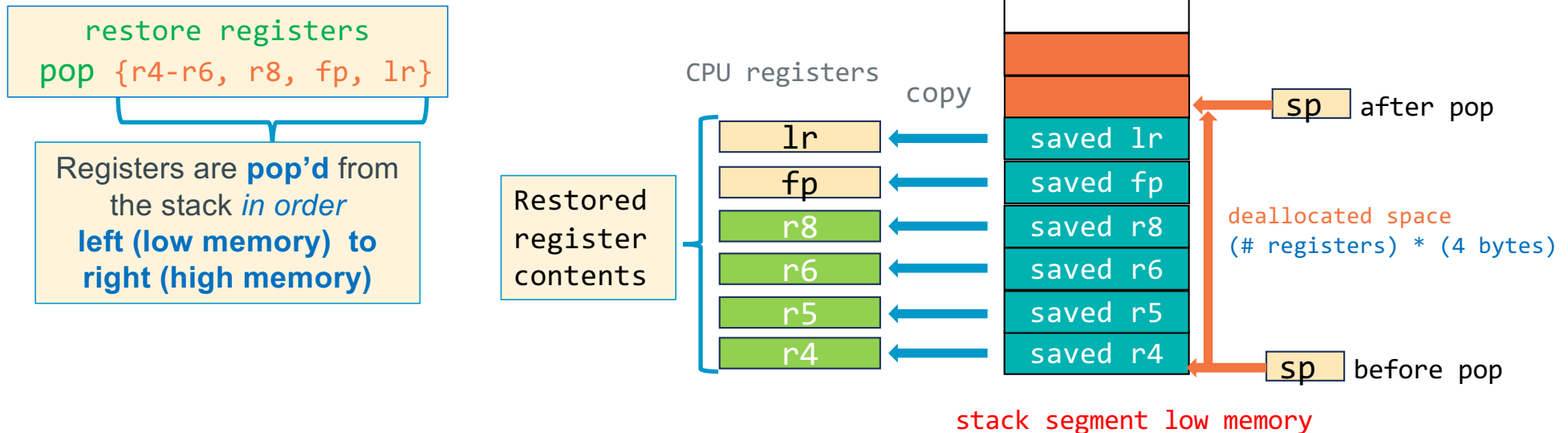
- `{reg list}` is a **list of registers in numerically increasing order, left to right**  
`push {r4-r10, fp, lr}` // *fp is r11, lr is r14*
- Registers **cannot be**:
  1. duplicated in the list
  2. listed out of increasing numeric order (left to right)
- Register ranges can be specified `{r4, r5, r8-r10, fp, lr}`
- **Never!** push/pop `r12, r13, or r15`
  - the top two registers on the stack must always be `fp, lr` // ARM function spec – later slides

# push: Multiple Register Save to the stack



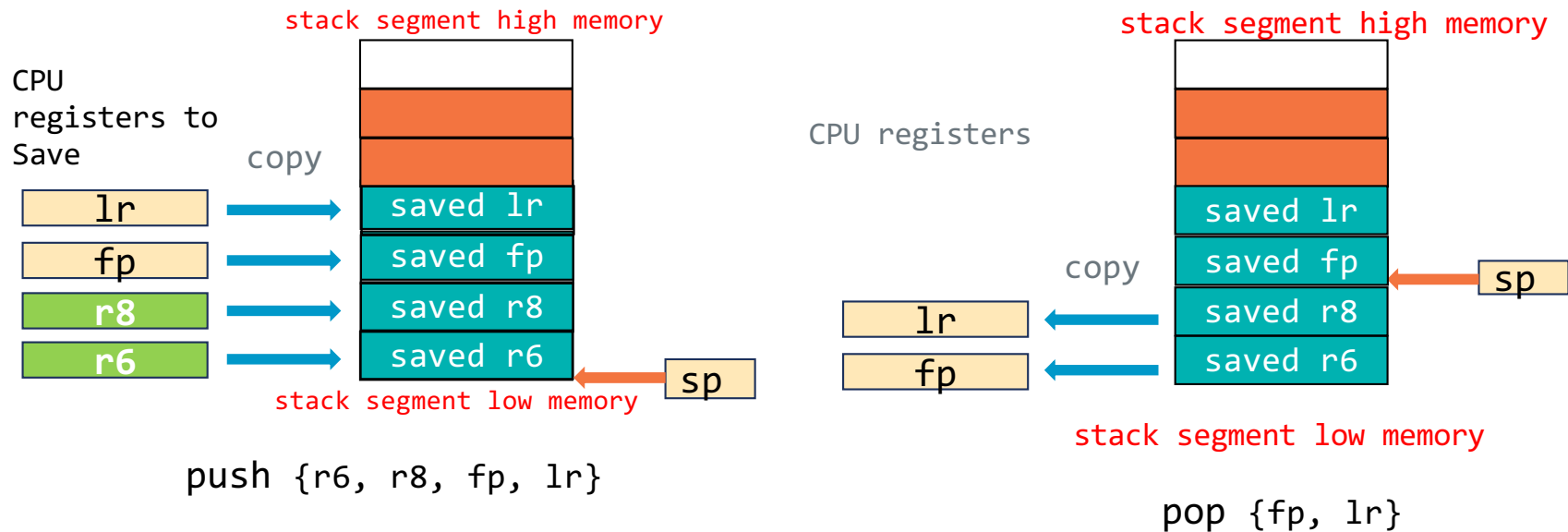
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** subtracts  $(\# \text{ of registers saved}) * (4 \text{ bytes})$  from the **sp** to **allocate** space on the stack
  - $sp = sp - (\# \text{ registers\_saved} * 4)$
- **this must always be true:  $sp \% 8 == 0$**

# pop: Multiple Register Restore from the stack



- **pop** copies the contents of stack segment memory to the **{reg list}**
- **pop adds:**  $(\# \text{ of registers restored}) * (4 \text{ bytes})$  to **sp** to **deallocate** space on the stack
  - $sp = sp + (\# \text{ registers restored} * 4)$
- **Remember:** **{reg list}** must be the same in both the **push** and the corresponding **pop**

# Consequences of inconsistent push and pop operands

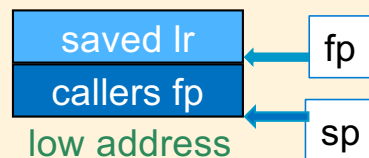


- `lr` gets an address on the stack, likely segmentation fault

## Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

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

Minimum stack frame



- **sp** always points at top element in the stack (lowest byte address)
- **fp** always points at the bottom element in the stack
  - Bottom element is always the saved **lr** (contains the return address of caller)
  - A saved copy of **callers fp** is always the next element below the **lr**
  - **fp** will be used later when referencing stack variables
- Minimal frame: deallocating at function exit: **pop {fp, lr}**
- On function entry: **sp** must be 8-byte aligned (**sp % 8 == 0**)

# Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

- **Function entry (Function Prologue):**

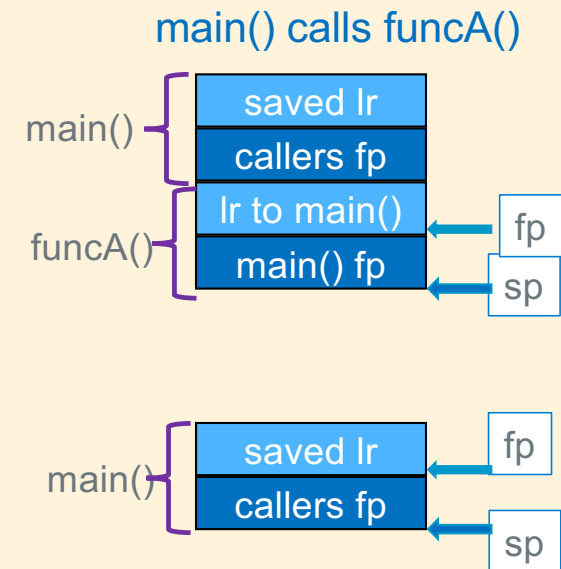
1. create (activate) frame
2. save preserved registers
3. allocate space for locals

allocate stack space  
 $SP = SP - \text{"space"}$   
grows "down"

- **Function return (Function Epilogue):**

1. deallocate space for locals
2. restores preserved registers
3. removes the frame

deallocate stack space  
 $SP = SP + \text{"space"}$   
shrinks "up"





## How to set the FP – Minimum Activation Frame

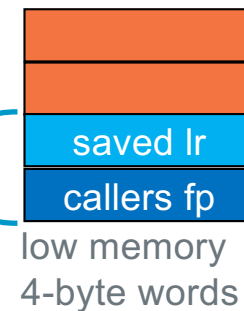
```
// other code
.equ    FP_OFF, 4
main:
  push   {fp, lr}
  add    fp, sp, FP_OFF
  .....
  sub    sp, fp, FP_OFF
  pop    {fp, lr}
  bx     lr
```

**Function Prologue**  
always at top of function  
push saves regs and  
allocates space by  
subtracting from sp and  
sets fp with the add

**Function Epilogue**  
always at bottom of  
function pop restores  
regs fp, lr  
and deallocates space  
by adding to sp

main()  
Stack  
Frame

after push {fp,lr}  
add fp, sp, FP\_OFF



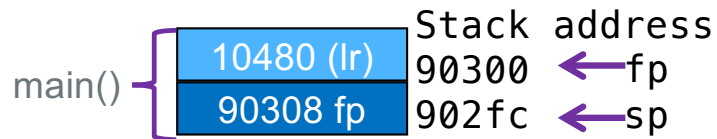
fp = sp + 4 bytes

**IMPORTANT:** FP\_OFF has two uses:

1. Where to set fp after prologue push (remember sp position)
2. Restore sp (deallocates locals) right before epilogue pop

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



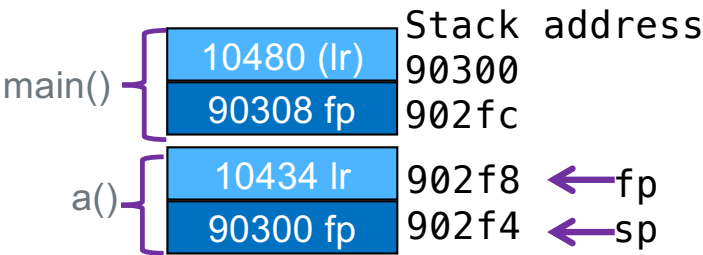
```
000103f4 <b>:
    103f4: e92d4800    push {fp, lr}
    103f8: e28db004    add fp, sp, 4
    103fc: e3a00000    mov r0, 0
    10400: e24bd004    sub sp, fp, 4
    10404: e8bd4800    pop {fp, lr}
    10408: e12fff1e    bx lr
```

```
0001040c <a>:
    1040c: e92d4800    push {fp, lr}
    10410: e28db004    add fp, sp, 4
    10414: ebfffff6    bl 103f4 <b>
    10418: e3a00000    mov r0, 0
    1041c: e24bd004    sub sp, fp, 4
    10420: e8bd4800    pop {fp, lr}
    10424: e12fff1e    bx lr
```

```
00010428 <main>:
    10428: e92d4800    push {fp, lr}
    1042c: e28db004    add fp, sp, 4
    10430: ebfffff5    bl 1040c <a>
    10434: ebfffff4    bl 1040c <a>
    // not shown
```

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



```
000103f4 <b>:
    103f4: e92d4800    push {fp, lr}
    103f8: e28db004    add fp, sp, 4
    103fc: e3a00000    mov r0, 0
    10400: e24bd004    sub sp, fp, 4
    10404: e8bd4800    pop {fp, lr}
    10408: e12fff1e    bx lr

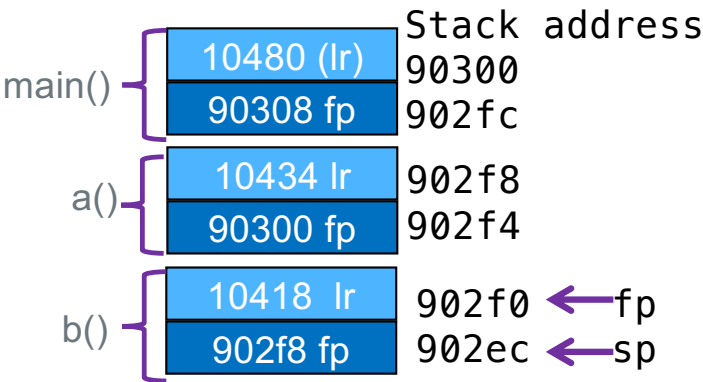
0001040c <a>:
    1040c: e92d4800    push {fp, lr}
    10410: e28db004    add fp, sp, 4
    10414: ebfffff6    bl 103f4 <b>
    10418: e3a00000    mov r0, 0
    1041c: e24bd004    sub sp, fp, 4
    10420: e8bd4800    pop {fp, lr}
    10424: e12fff1e    bx lr

00010428 <main>:
    10428: e92d4800    push {fp, lr}
    1042c: e28db004    add fp, sp, 4
    10430: ebfffff5    bl 1040c <a>
    10434: ebfffff4    bl 1040c <a>
    // not shown
```

lr:  
10434

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



```
000103f4 <b>:
    103f4: e92d4800    push {fp, lr}
    103f8: e28db004    add fp, sp, 4
    103fc: e3a00000    mov r0, 0
    10400: e24bd004    sub sp, fp, 4
    10404: e8bd4800    pop {fp, lr}
    10408: e12fff1e    bx lr
```

lr:  
10418

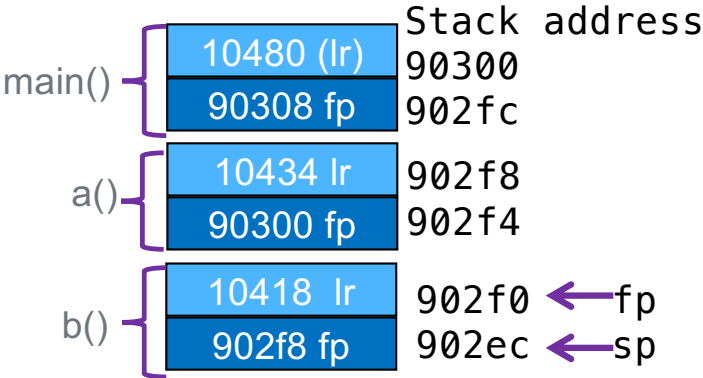
```
0001040c <a>:
    1040c: e92d4800    push {fp, lr}
    10410: e28db004    add fp, sp, 4
    10414: ebfffff6    bl 103f4 <b>
    10418: e3a00000    mov r0, 0
    1041c: e24bd004    sub sp, fp, 4
    10420: e8bd4800    pop {fp, lr}
    10424: e12fff1e    bx lr
```

lr:  
10434

```
00010428 <main>:
    10428: e92d4800    push {fp, lr}
    1042c: e28db004    add fp, sp, 4
    10430: ebfffff5    bl 1040c <a>
    10434: ebfffff4    bl 1040c <a>
    // not shown
```

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



lr: 10418

```
000103f4 <b>:
  103f4: e92d4800  push {fp, lr}
  103f8: e28db004  add fp, sp, 4
  103fc: e3a00000  mov r0, 0
  10400: e24bd004  sub sp, fp, 4
  10404: e8bd4800  pop {fp, lr}
  10408: e12fff1e  bx lr
```

lr: 10418

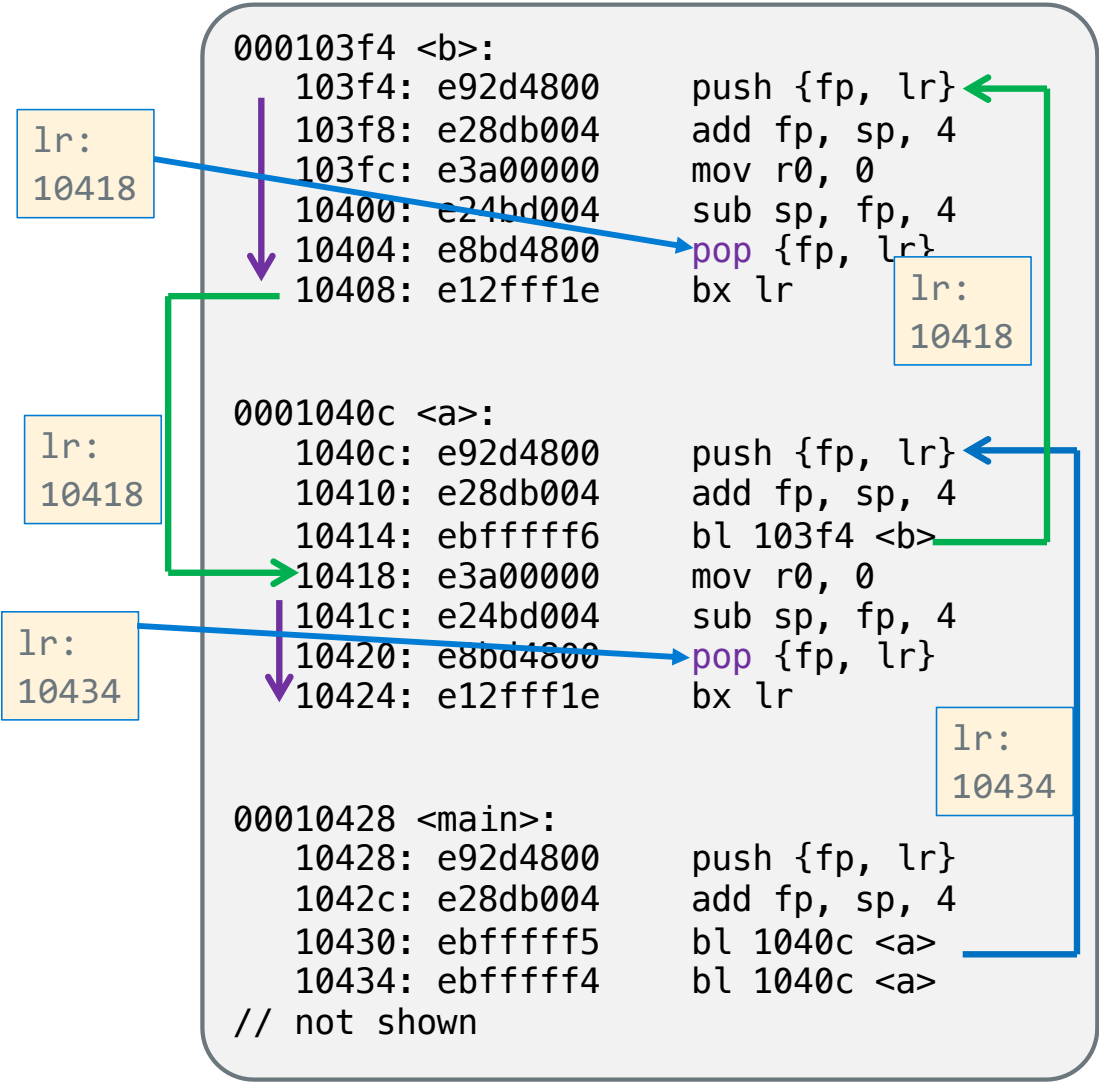
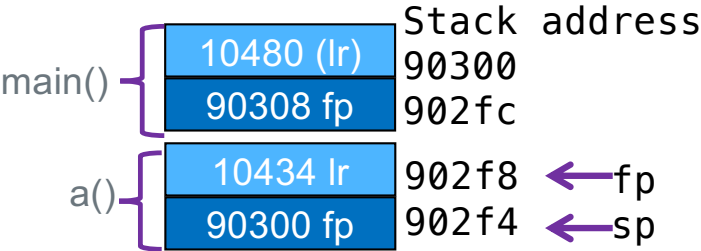
```
0001040c <a>:
  1040c: e92d4800  push {fp, lr}
  10410: e28db004  add fp, sp, 4
  10414: ebfffff6  bl 103f4 <b>
  10418: e3a00000  mov r0, 0
  1041c: e24bd004  sub sp, fp, 4
  10420: e8bd4800  pop {fp, lr}
  10424: e12fff1e  bx lr
```

lr: 10434

```
00010428 <main>:
  10428: e92d4800  push {fp, lr}
  1042c: e28db004  add fp, sp, 4
  10430: ebfffff5  bl 1040c <a>
  10434: ebfffff4  bl 1040c <a>
  // not shown
```

# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```



# Using Minimal Stack Frames

```
int b(void)
{
    return 0;
}
int a(void)
{
    b();
    return 0;
}
int main(void)
{
    a();
    a();
}
```

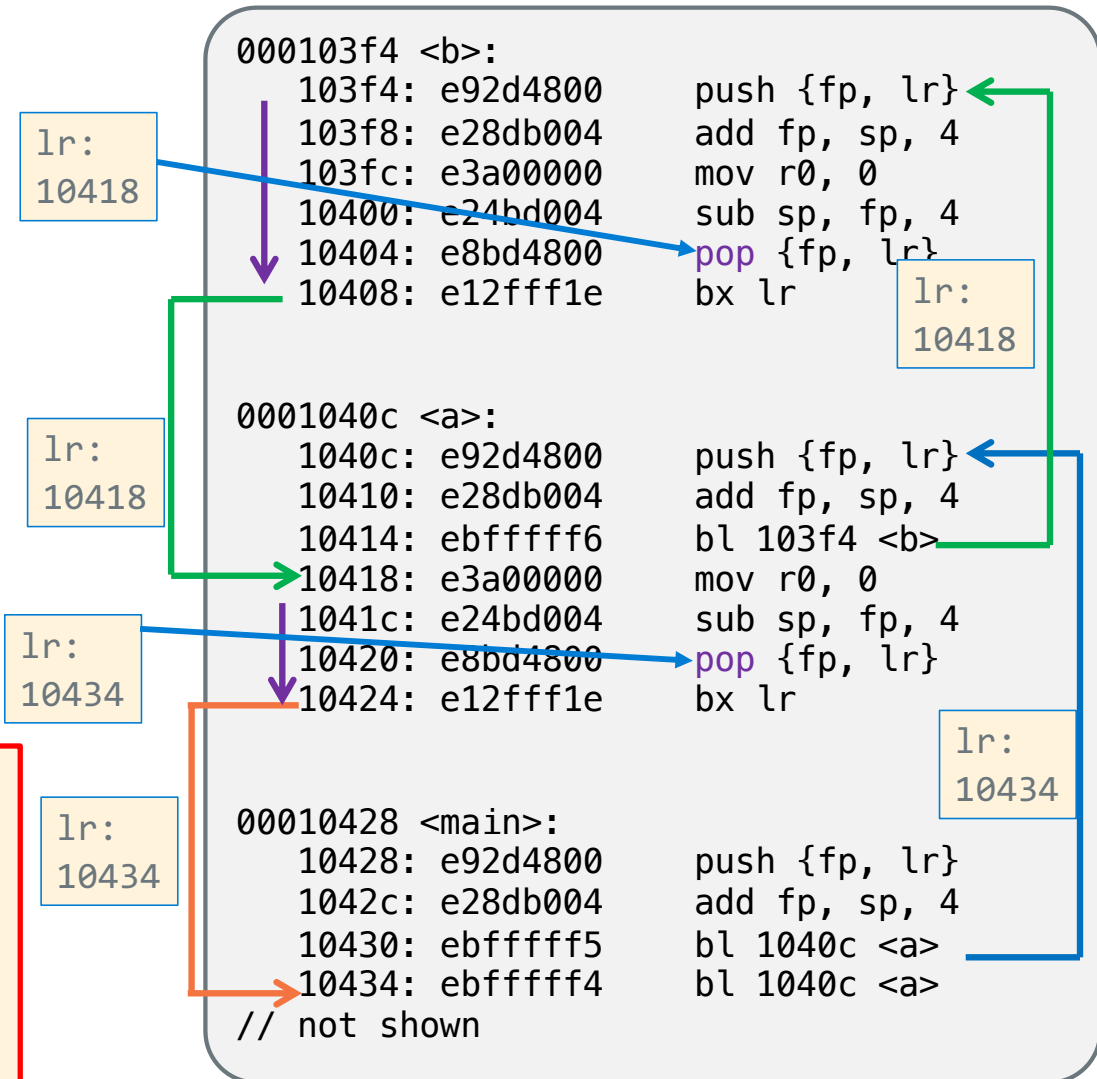
main() {

	Stack address	
10480 (lr)	90300	← fp
90308 fp	902fc	← sp

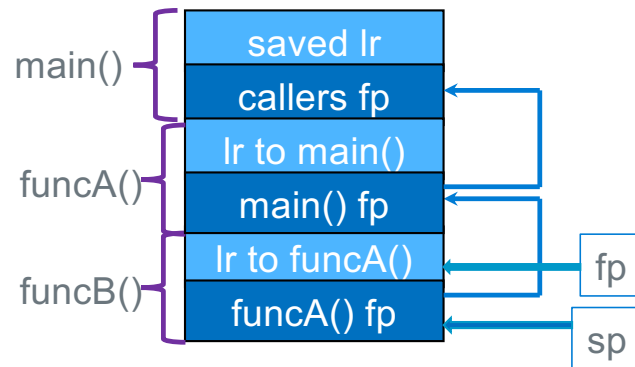
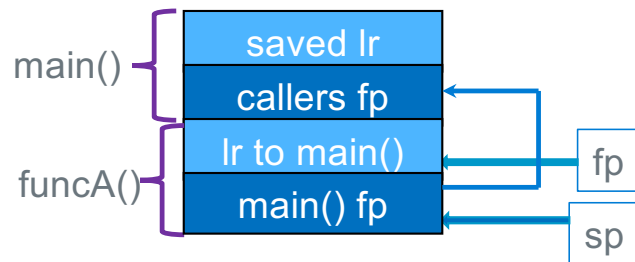
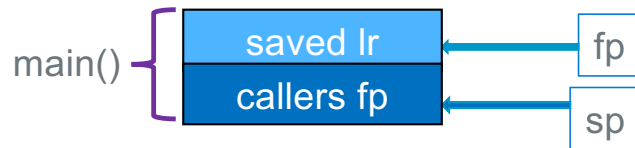
}

We are saving the lr on the stack on each function call and restoring it before returning.

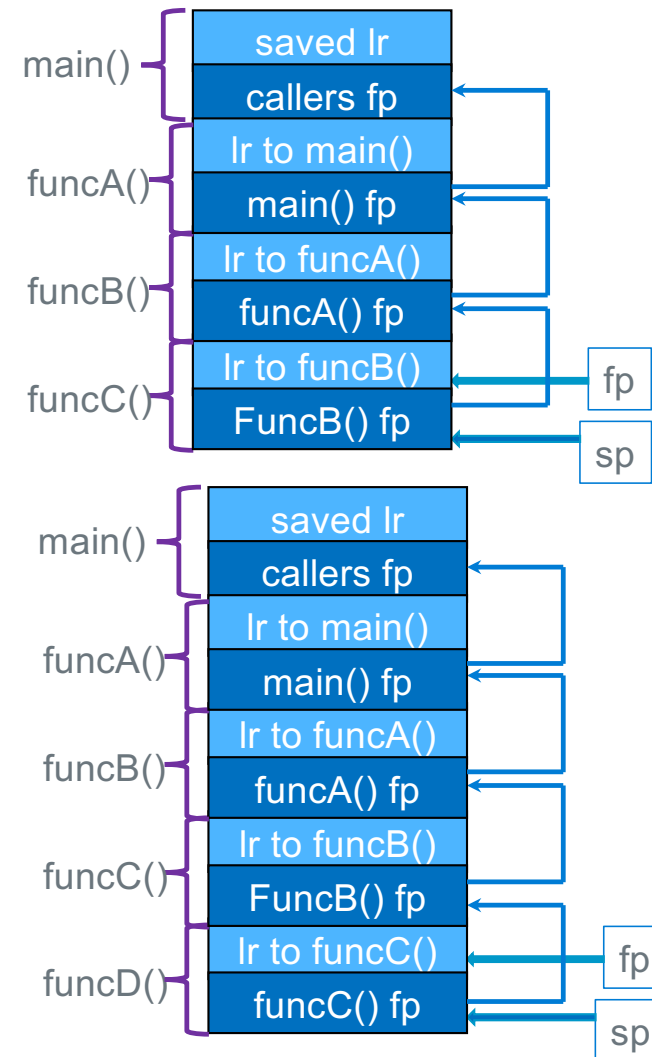
**Result:** NO infinite loop and we return to the correct instruction in the caller no matter how many functions we call.  
Even recursion will work!



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



How gdb finds stack frames





## Registers: Requirements for Use

Register	Function Call Use	Function Body Use	Save before use Restore before return
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No
r4-r10	preserved registers	contents preserved across function calls	Yes
r11/fp	stack frame pointer	Use to locate variables on the stack	Yes
r12/ip	may used by assembler with large text file	can be used as a scratch if really needed	No
r13/sp	stack pointer	stack space allocation	Yes
r14/lr	link register	contains return address for function calls	Yes
r15	Do not use	Do not use	No

- Any value you have in a **preserved register before a function call will still be there after the function returns**
- Contents are “preserved” across function calls

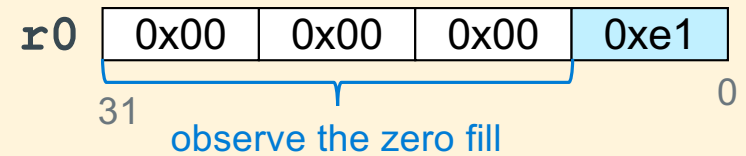
If the function wants to use a preserved register it must:

1. **Save** the **value contained in the register** at **function entry**
2. Use the register in the body of the function
3. **Restore** the **original saved value** to the register at **function exit** (before returning to the caller)

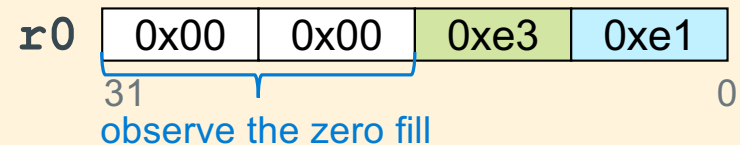
# Argument and Return Value Requirements

- When passing or returning values from a function you must do the following:
  - Make sure that the values in the registers r0-r3 are in their **properly aligned position in the register based on data type**
  - Upper bytes in byte and halfword values in registers r0-r3 when passing arguments and returning values **are zero filled**

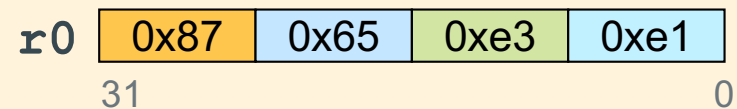
## Single Byte (char)



## Single Halfword (short)



## Full Word (int or pointer)



# Global Variable access

var	global variable address into r0 (lside)	global variable contents into r0 (rside)	contents of r0 into global variable
x	ldr r0, =x	ldr r0, =x ldr r0, [r0]	ldr r1, =x str r0, [r1]
*x	ldr r0, =x ldr r0, [r0]	ldr r0, =x ldr r0, [r0] ldr r0, [r0]	ldr r1, =x ldr r1, [r1] str r0, [r1]
**x	ldr r0, =x ldr r0, [r0] ldr r0, [r0]	ldr r0, =x ldr r0, [r0] ldr r0, [r0] ldr r0, [r0]	ldr r1, =x ldr r1, [r1] ldr r1, [r1] str r0, [r1]
stderr	ldr r0, =stderr	ldr r0, =stderr ldr r0, [r0]	<do not write unless you really know what you are doing>
.Lstr	ldr r0, =.Lstr	ldr r0, =.Lstr ldrb r0, [r0]	<read only>

```
.bss // from libc
stderr:.space 4 // FILE *
```

```
.data
x: .data y //x = &y
```

```
.section .rodata
.Lstr: .string "HI\n"
```

stdin, stdout and stderr are global variables

## Assembler Directives: Label Scope Control (Normal Labels only)

```
.extern printf
.extern fgets
.extern strcpy
.global fbuf
```

**.extern** <label>

- **Imports** label (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

**.global** <label>

- **Exports** label (or symbol) to be visible outside the source file boundary (other assembly or c source)
- label is either a function name or a global variable name
- Only use with function names or static variables
- **Without** .global, labels are usually (depends on the assembler) **local to the file**

# Example calling fprintf()

- `r0 = function(r0, r1, r2, r3)`  
`fprintf(stderr, "arg2", arg3, arg4)`
- create a literal string for arg2 which tells `fprintf()` how to interpret the remaining arguments
- `stdin`, `stdout`, `stderr` are all **global variable** and are **part of libc**
  - these **names are their lside (label names)**
- to use them you must **get their contents** to pass to `fprintf()`, `fread()`, `fwrite()`

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
```

```
    int a = 2;
    int b = 3;
    int c;
```

We are going to  
put these  
variables in  
temporary  
registers

```
    c = a + b;
    fprintf(stderr, "c=%d\n", c);
```

`r0, r1, r2`

```
    return EXIT_SUCCESS;
```

```
}
```

```
.extern fprintf      //declare fprintf
.section .rodata     // note the dots "."
.Lfst: .string "c=%d\n"
```

// part of the **text segment** below

```
mov    r2, 2          // int a = 2;
mov    r3, 3          // int b = 3;
add    r2, r2, r3     // arg 3: int c = a + b;

ldr    r0, =stderr    // get stderr address
ldr    r0, [r0]       // arg 1: get stderr contents
ldr    r1, =.Lfst     // arg 2: =literal address
bl     fprintf
```

three passed  
args in this  
use of fprintf

## Preserved Registers: When to Use?

Register	Function Call Use	Function Body Use	Save before use Restore before return
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No
r4-r10	preserved registers	contents preserved across function calls	Yes
r11/fp	stack frame pointer	Use to locate variables on the stack	Yes
r12/ip	may used by assembler with large text file	can be used as a scratch if really needed	No
r13/sp	stack pointer	stack space allocation	Yes
r14/lr	link register	contains return address for function calls	Yes
r15	Do not use	Do not use	No

- When to use a preserved register in a function you are writing?
- Values that you want to protect from being changed by a function call
  - Local variables stored in registers
  - Parameters passed to you (in **r0-r3**) that you need to continue to use after calling another function

# Saving Preserved registers and setting FP

```
// other code etc
.equ    FP_OFF, 20

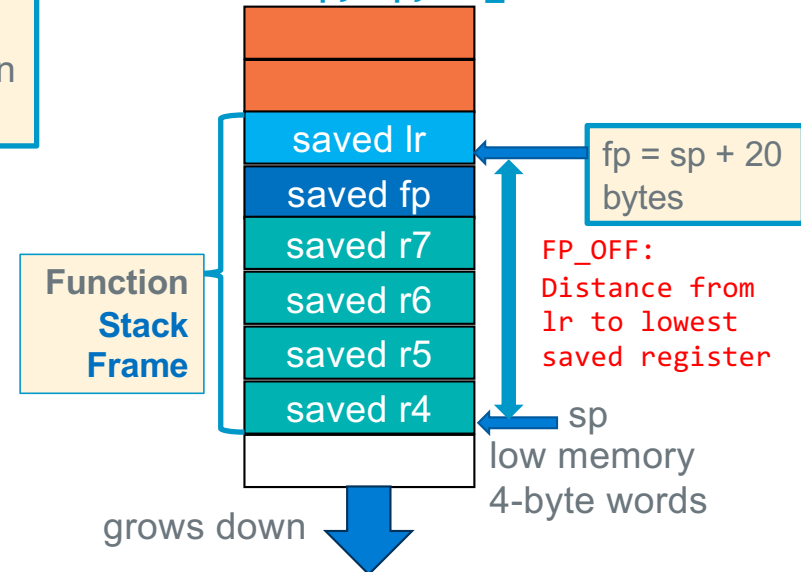
main:
  push   {r4-r7, fp, lr}
  add    fp, sp, FP_OFF
  .....
  sub    sp, fp, FP_OFF
  pop    {r4-r7, fp, lr}
  bx     lr
```

**Function Prologue**  
always at top of function  
saves regs and **sets fp**

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

# regs saved	FP_OFF in Bytes Distance from lr to lowest saved register
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

after push {r4-r7, fp, lr}  
add fp, sp, FP\_OFF



$$FP\_OFF = (\#regs\ saved - 1) * 4$$



**Means Caution, odd number of saved regs!**  
If odd number pushed, make sure frame is 8-byte aligned (later)  
this must always be true: **sp % 8 == 0**

## Example: using preserved registers for local variables

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
```

**You must assume that**  
both getchar() and  
putchar() alter r0-r3

```
    int c; // use r0
    int count = 0; // use r4
```

r0

```
    while ((c = getchar()) != EOF) {
```

```
        putchar(c);
        count++;
```

```
    }
```

```
    printf("Echo count: %d\n", count);
    return EXIT_SUCCESS;
```

```
}
```



```
.extern getchar
.extern putchar
.section .rodata
.Lst: .string "Echo count: %d\n"
```

```
.text
.type main, %function
.global main
.equ EOF, -1
.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 \*/*

```
    mov     r0, EXIT_SUCCESS
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
    .size main, (. - main)
```



# Putchar/getchar: The while loop

```
#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;
}
```

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

```
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    //arg2
ldr    r0, =.Lst  //arg1
bl     printf
```

address of string literal variable

.Lst: .string "Echo count: %d\n"

**File header and footers are not shown**

# Accessing argv from Assembly (stderr version)

```

.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;

.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

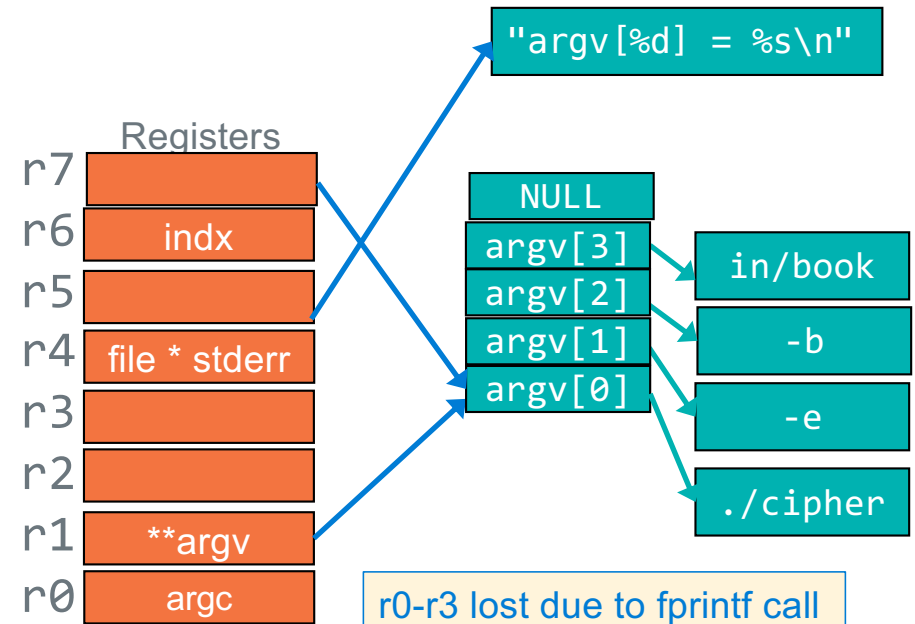
.Ldone:
    mov     r0, 0
    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx      lr
    
```

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

observe the  
different  
increment sizes

```

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



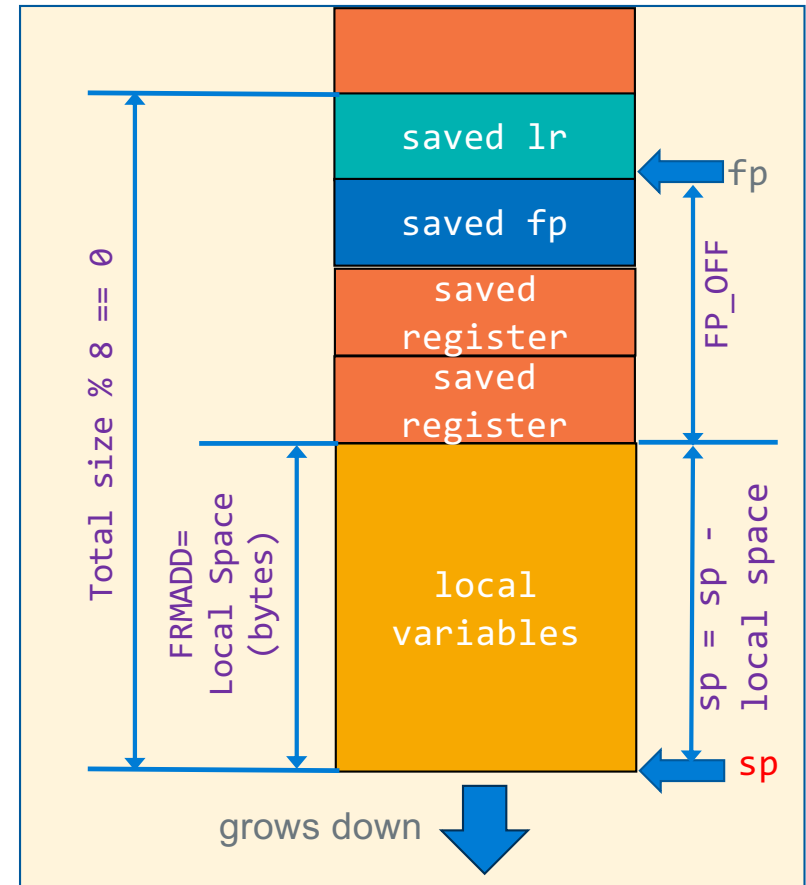
```
fprintf(stderr, "argv[%d] = %s\n", indx, *argv);
```

# Local Variables on the Stack

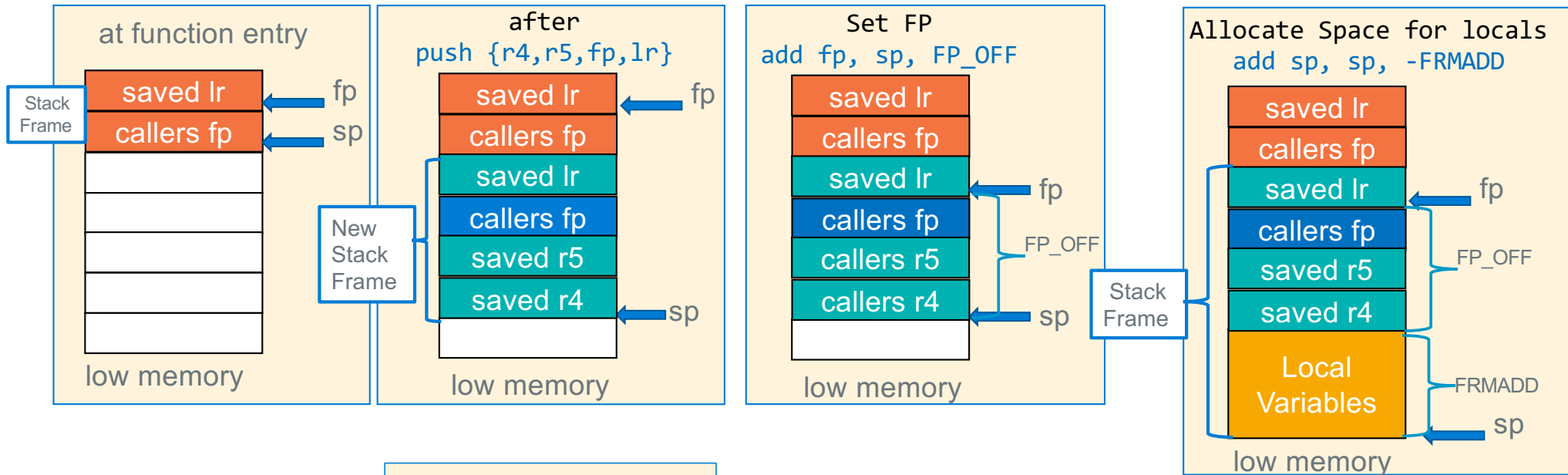
- Space for local variables is allocated on the stack right below the lowest pushed register
  - Move the **sp** towards low memory by the total size of all local variables in bytes **plus padding**

$\text{FRMADD} = \text{total local var space (bytes)} + \text{padding}$

- Allocate the space after the register push by  
`add sp, sp, -FRMADD`
- Requirement:** on function entry, **sp** is always 8-byte aligned  
 $\text{sp} \% 8 == 0$
- Padding (as required):**
  - Additional space between variables on the stack to meet memory alignment requirements
  - 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



# Function Prologue: Allocating the Stack Frame



Function was just called this how the stack looks  
The orange blocks are part of the caller's stack frame

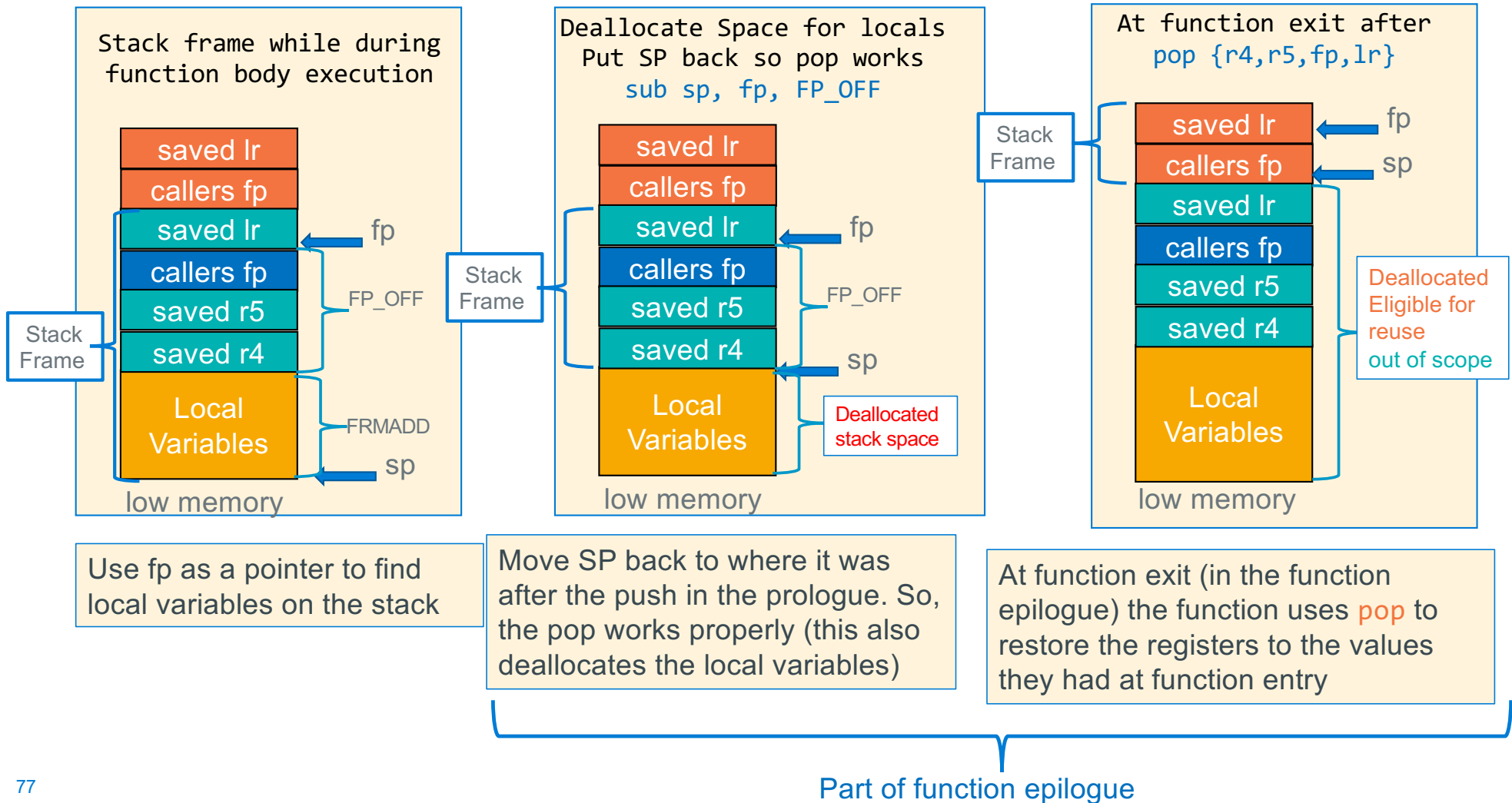
Function saves lr, fp using a push and only those preserved registers it wants to use on the stack  
Do not push r12 or r13

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

# Function Epilogue: Deallocating the Stack Frame



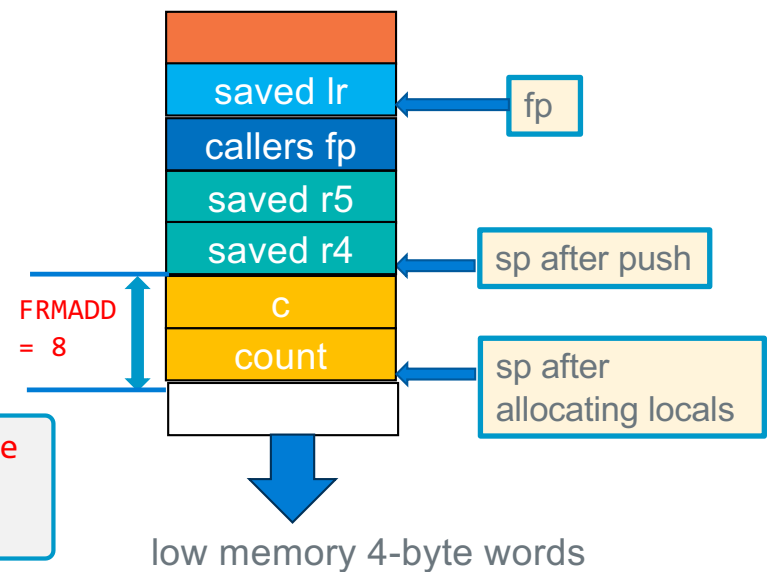
# Local Variables on the stack

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

```
.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
```

after push {r4-r5, fp, lr}  
add fp, sp, FP\_OFF



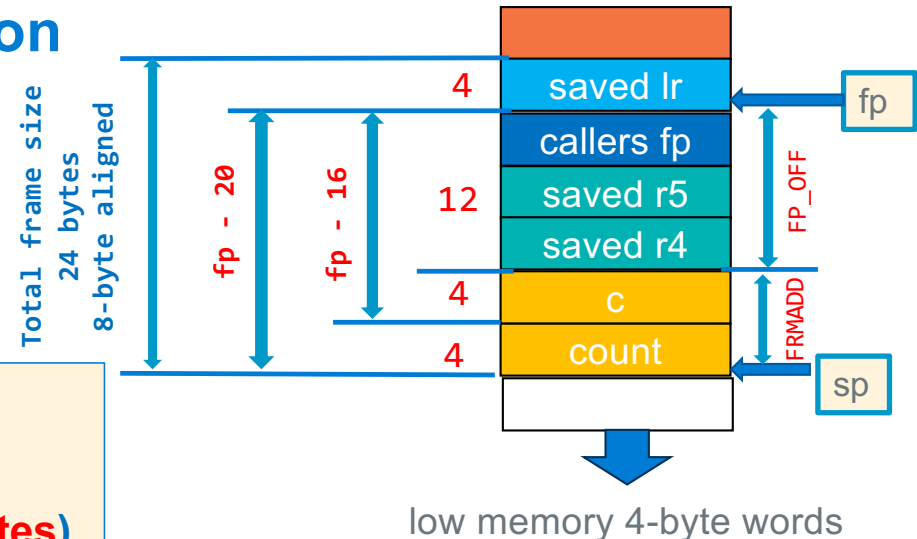
- Add space on the stack for each local
  - we will allocate space in same order the locals are listed the C function shown from high to low stack address
  - gcc compiler allocates from low to high stack addresses
  - Order does not matter for our use

- 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

# 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 (**negative 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`

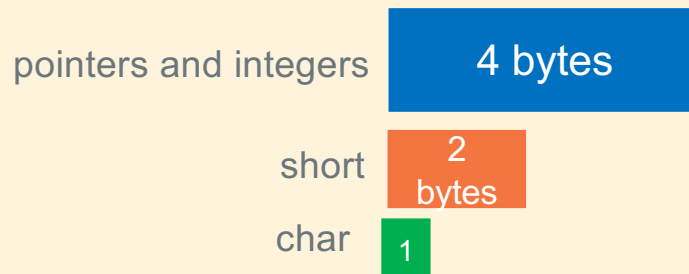


```
.text
.type    main, %function
.global  main
.equ     FP_OFF,    12
.equ     FRMADD,    8
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // but we are not done yet!
```

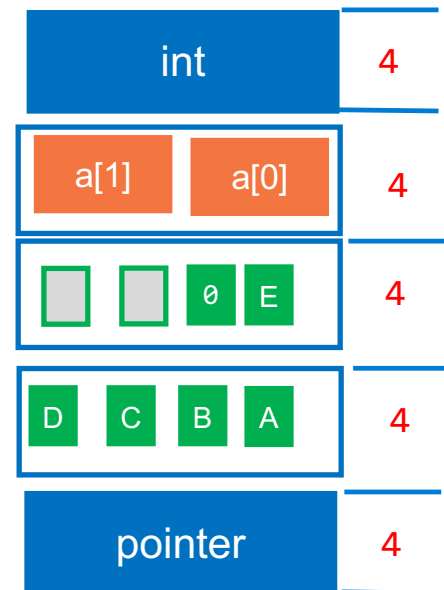
Variable	distance from fp	Read variable	Write Variable
int c	-16	ldr r0, [fp, -16]	str r0, [fp, -16]
int count	-20	ldr r0, [fp, -20]	str r0, [fp, -20]

# 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)



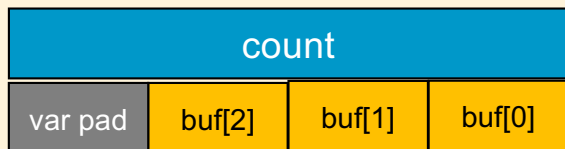
**Rule:** When the function is entered the stack is already 8-byte aligned



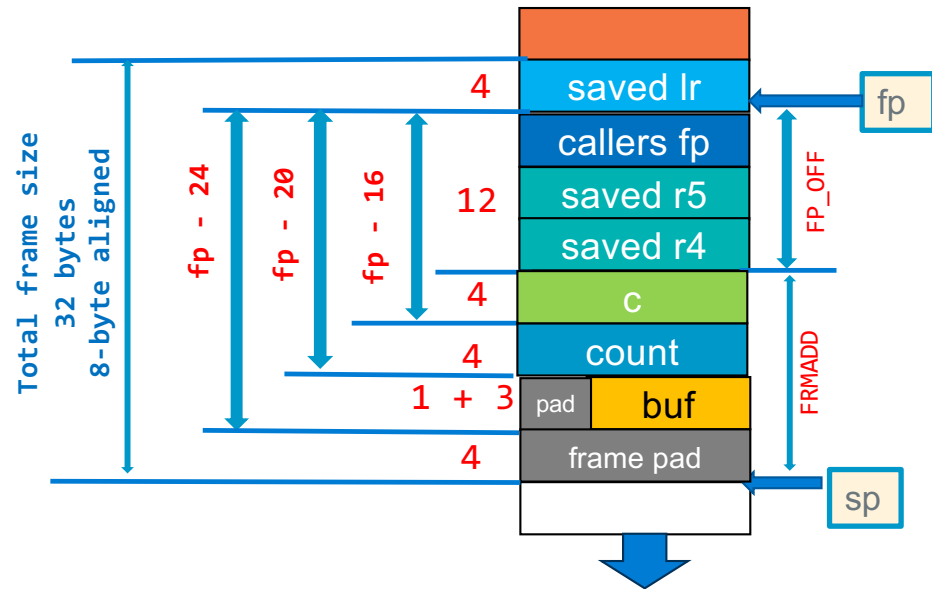
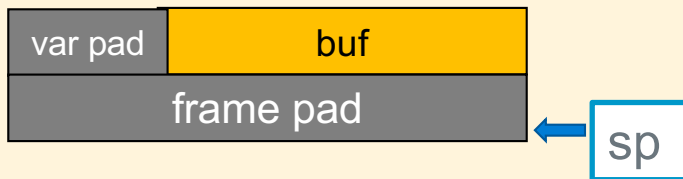


# Stack Variables: Padding

- **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
}
```

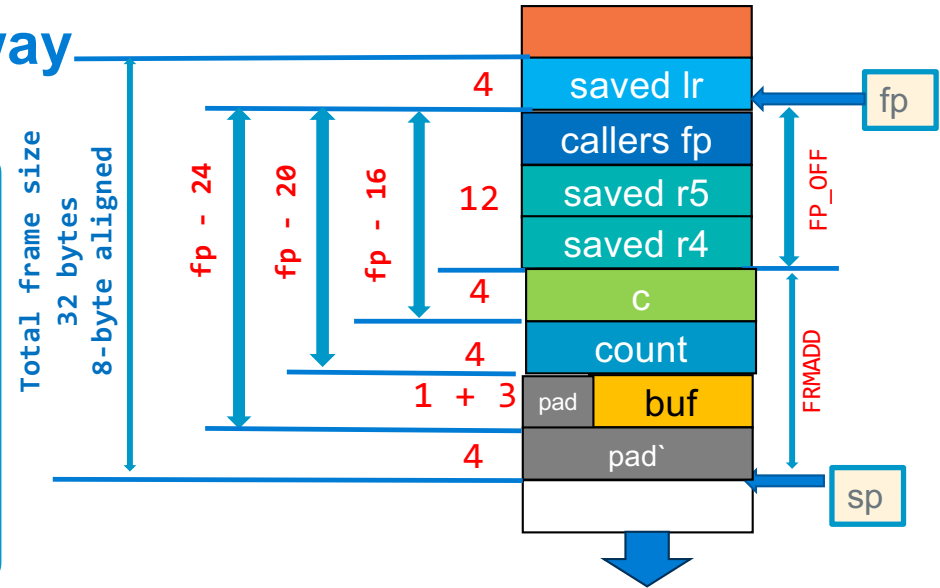
```
.text
.type    main, %function
.global  main
.equ     FP_OFF,    12
.equ     FRMADD,    16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
    // but we are not done yet!
```

## Accessing Stack Variables, the hard way

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

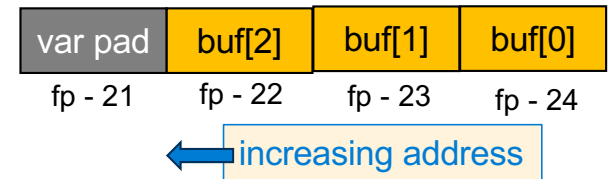
```
.text
.type    main, %function
.global  main
.equ     FP_OFF,      12
.equ     FRMADD,      16

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]



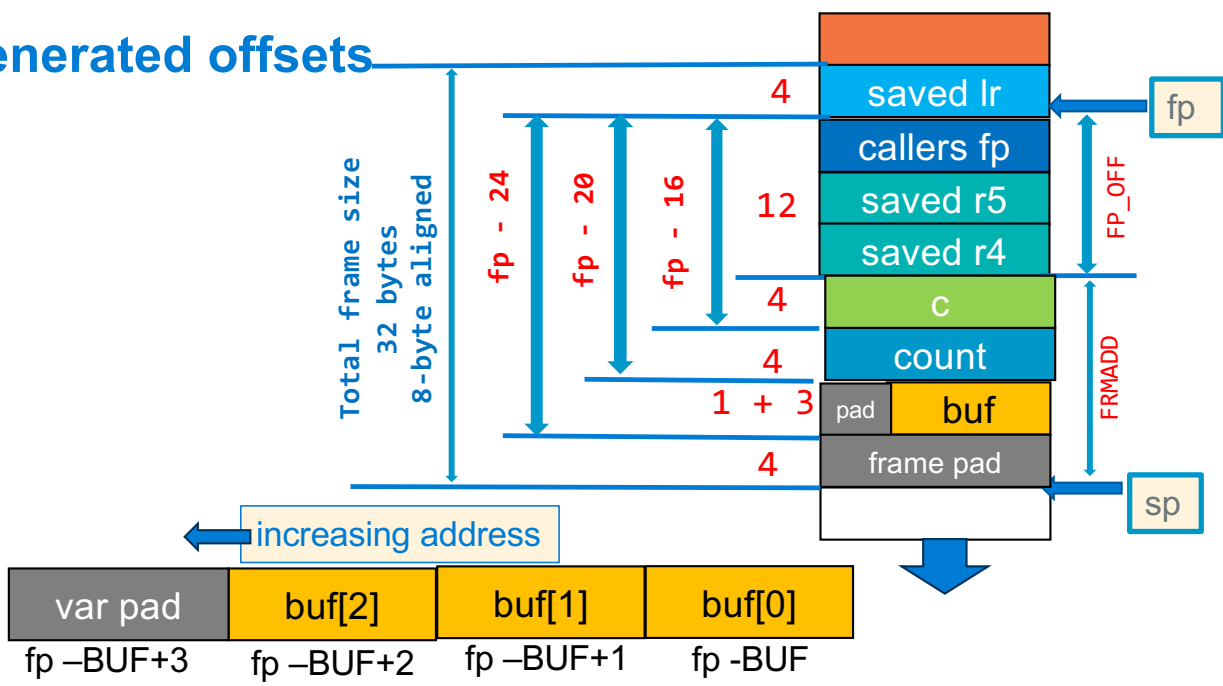
- **Calculating offsets is a lot of work to get it correct**
- It is also hard to debug
- There is a better way!

Best Practice: Use Assembler Generated offsets

```
.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
```

Annotations:   
 - "pushed reg fp distance" points to FP\_OFF.   
 - "variable size in bytes" points to the 4 in the .equ C line.   
 - "Prior allocation distance" points to the FP\_OFF in the .equ C line.



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]

# Initializing and Accessing Stack variables

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

```
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
```

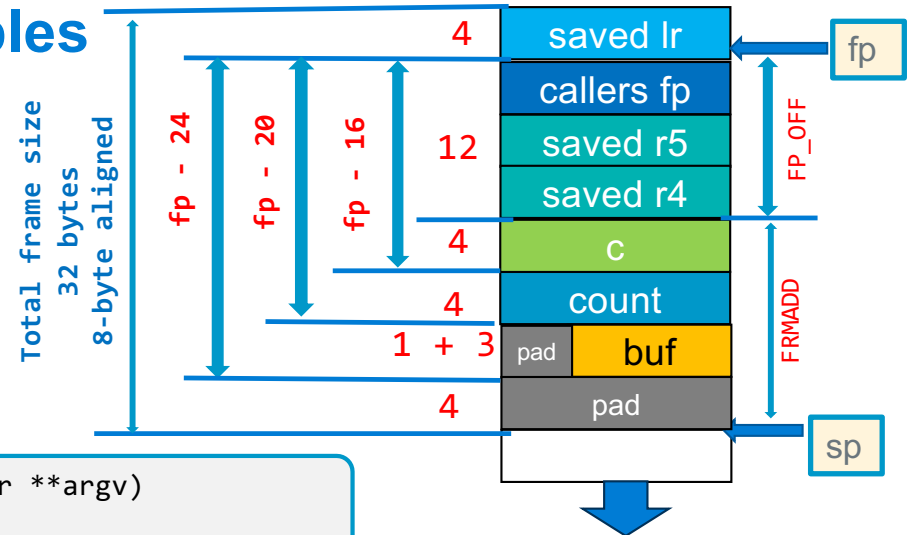
passes  
contents of  
stack var C

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

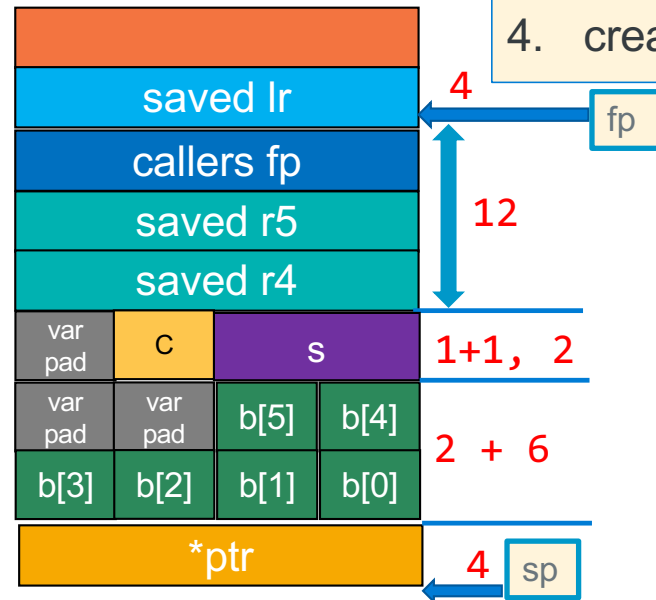


Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add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

# 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 table to access the variables

```
void func(void)
{
    signed char c;
    signed short s;
    unsigned char b[] = "Stack";
    unsigned char *ptr = &buf1
    // rest of code
}
```



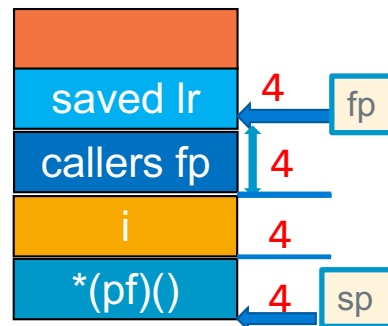
```
.equ    FP_OFF,      12
.equ    C,           2 + FP_OFF
.equ    S,           2 + C
.equ    B,           8 + S
.equ    PTR,         4 + BUF
.equ    PAD,         0 + PTR
.equ    FRMADD,      PAD - FP_OFF
// FRMADD = 28 - 12 = 8
```

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]

# Working with Pointers on the stack

```
int
sum(int j, int k)
{
    return j + k;
}
void
testp(int j, int k, int (*func)(), int *i)
{
    *i = func(j,k);
    return;
}
int
main()
{
    int i;
    int (*pf)() = add;

    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
```

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]

# Working with Pointers on the stack

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

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

int
main()
{
    int i;
    int (*pf)() = add;

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

r0,r1,r2  
already set

```
.global sum
.type sum, %function
add:
    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)
```

```
.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)
```

```
.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

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

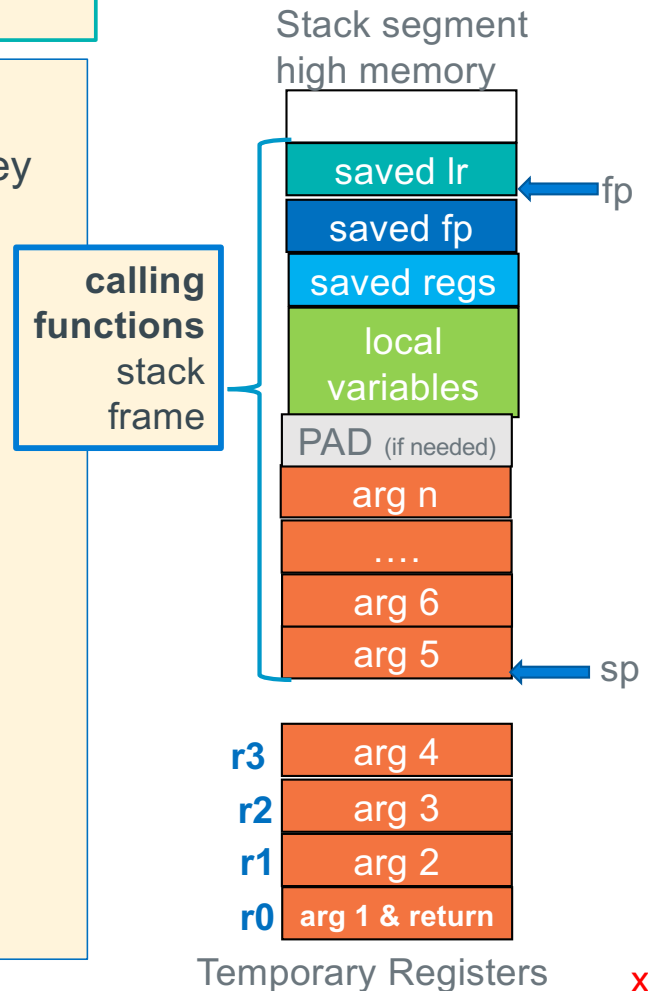
    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx     lr
.size main, (. - main)
```

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]

## Passing More Than Four Arguments – At the point of Call

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

- **Args > 4 are in the caller's stack frame at SP (argv5), an up**
- Called functions have the **right to change stack args** just like they can change the register args!
  - Caller must assume **all args including ones on the stack** are changed by the caller
- Calling function prior to making the call
  1. Evaluate **first four args**: place resulting **values in r0-r3**
  2. Store Arg 5 and greater parameter values on the stack
- **One arg value per slot!** – NO arrays across multiple slots
  - chars, shorts and ints are directly stored
  - Structs (not always), and arrays are passed via a pointer
  - **Pointers** passed as **output parameters** usually contain an **address that points at** the **stack, BSS, data, or heap**





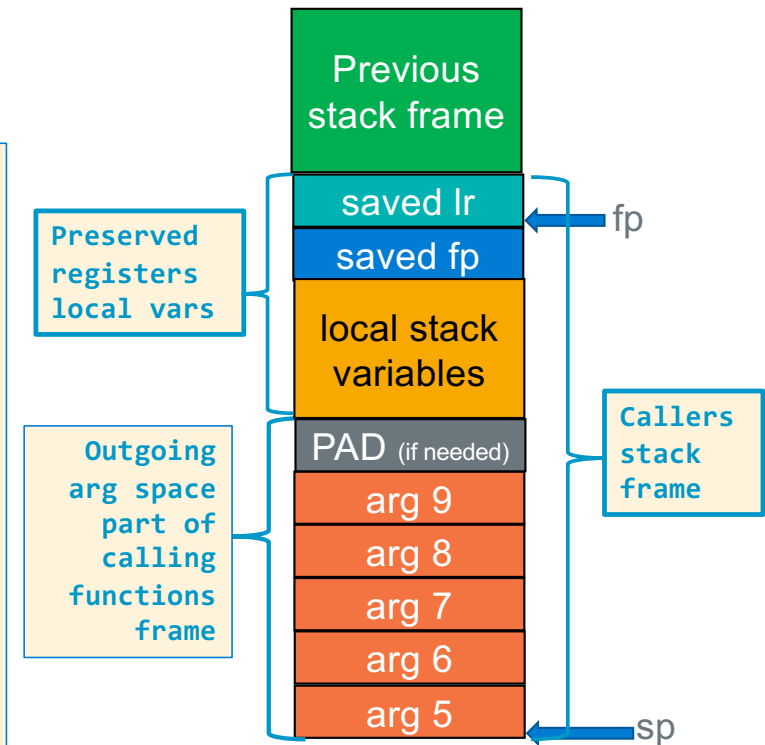
## Calling Function: Allocating Stack Parameter Space

At the point of a function call (and obviously at the start of the called function):

1. sp must point at arg5
2. sp and therefore arg5 **must be at an 8-byte boundary**,
  - a) **padding** to force arg5 alignment if needed is **placed above** the last **argument the called function is expecting**

**Approach:** Extend the stack frame to include enough space for stack arguments function with the greatest arg count

1. Examine every function call in the body of a function
2. Find the function call with greatest arg count, Determines space needed for outgoing args
3. Add the space needed to the frame layout



### Rules: At point of call

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

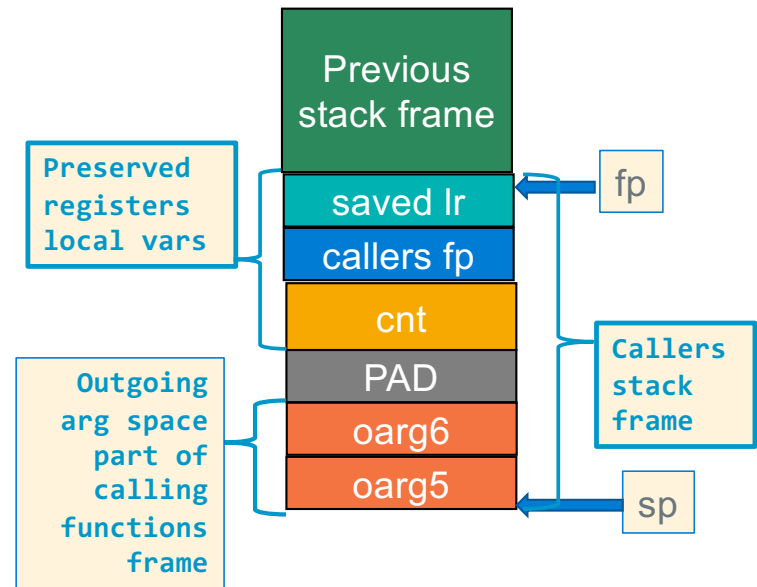
## Calling Function: Pass ARG 5 and higher

```
.equ    FP_OFF, 4
.equ    CNT,      4 + FP_OFF    // int cnt
.equ    PAD,      4 + CNT      // added as needed
.equ    OARG6,    4 + PAD      // int a
.equ    OARG5,    4 + OARG6    // int b
.equ    FRMADD    OARG5 - FP_OFF
```

### Rules: At point of call

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

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

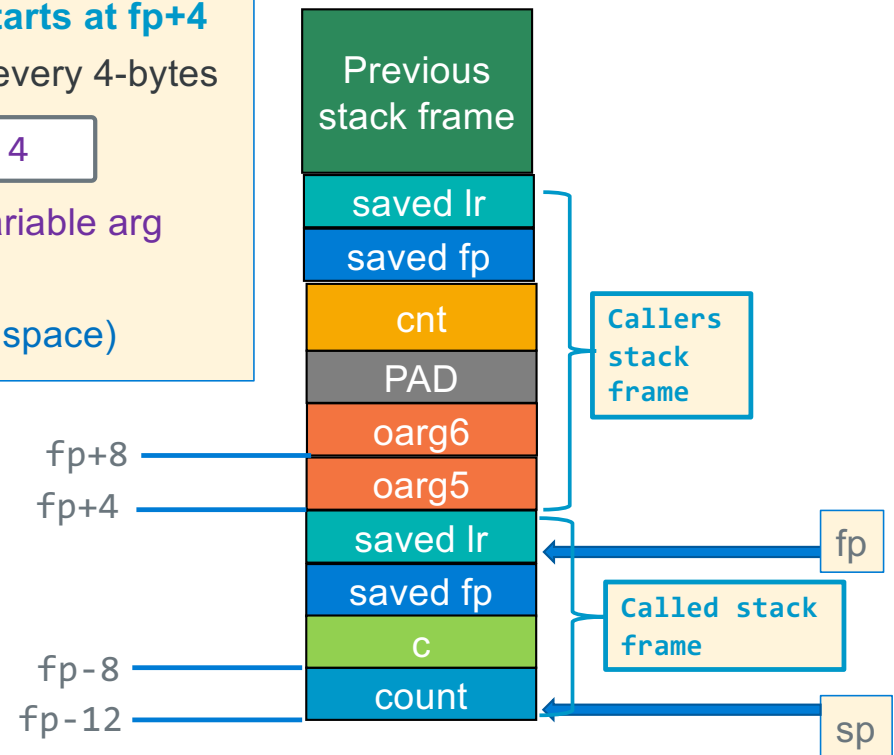


Variable	distance from fp	Address on Stack	Read variable	Write Variable
int cnt	CNT	add r0, fp, -CNT	ldr r0, [fp, -CNT]	str r0, [fp, -CNT]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

# 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
- `.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 parameters using a **positive offset to the fp**



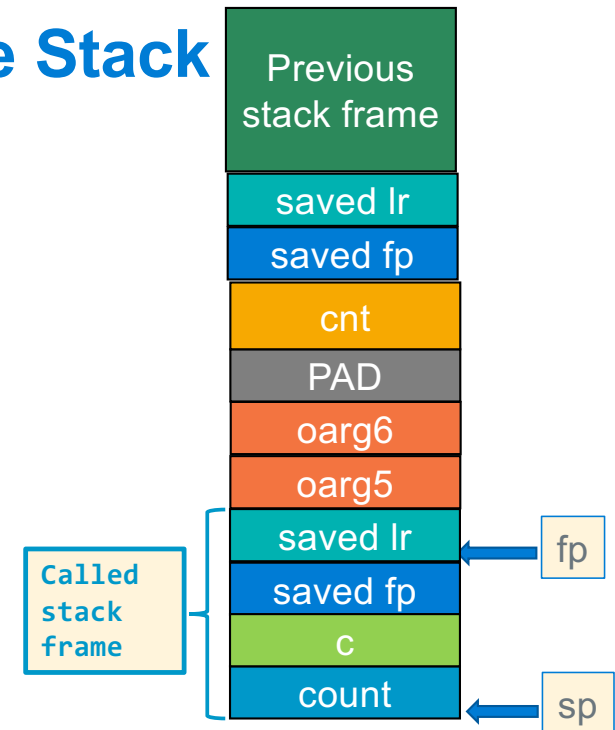
`r0 = func(r0, r1, r2, r3, r4, ARG5, ARG6);`

## Called Function: Retrieving Args From the Stack

```
.equ    FP_OFF,    4
.equ    C,         4 + FP_OFF
.equ    COUNT,     4 + C
.equ    PAD,       4 + COUNT
.equ    FRMADD,    PAD - FP_OFF
.equ    ARG6,      8
.equ    ARG5,      4
```

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

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



Variable	distance from fp	Address on Stack	Read variable	Write Variable
int arg6	OARG6	add r0, fp, OARG6	ldr r0, [fp, OARG6]	str r0, [fp, OARG6]
int arg5	OARG5	add r0, fp, OARG5	ldr r0, [fp, OARG5]	str r0, [fp, OARG5]
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]