Version 1.04

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 4

Lecture 19 – November 22, 2022

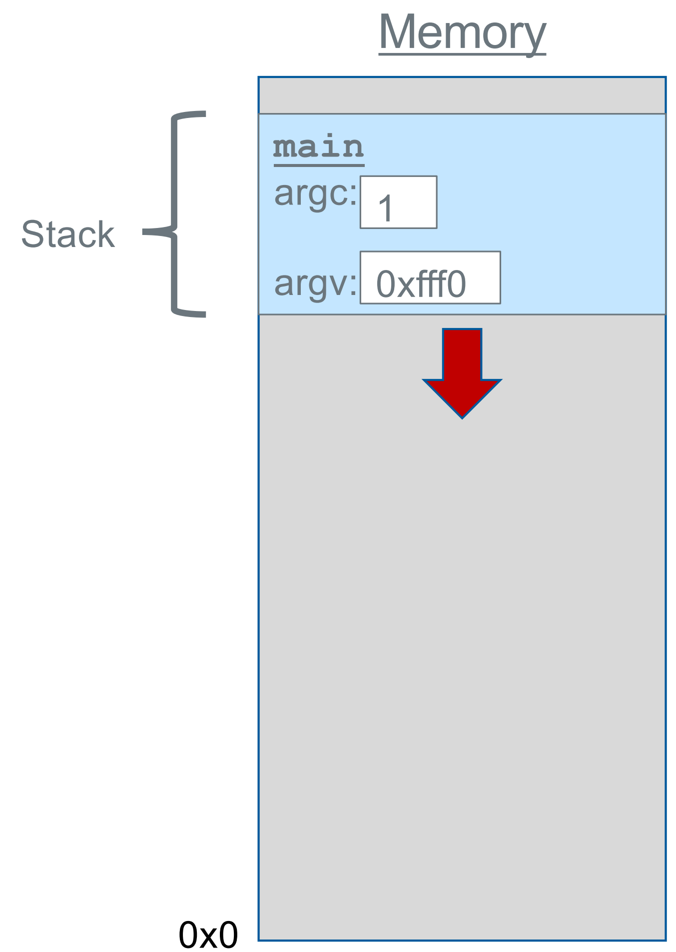
Keith Muller

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;
}
```

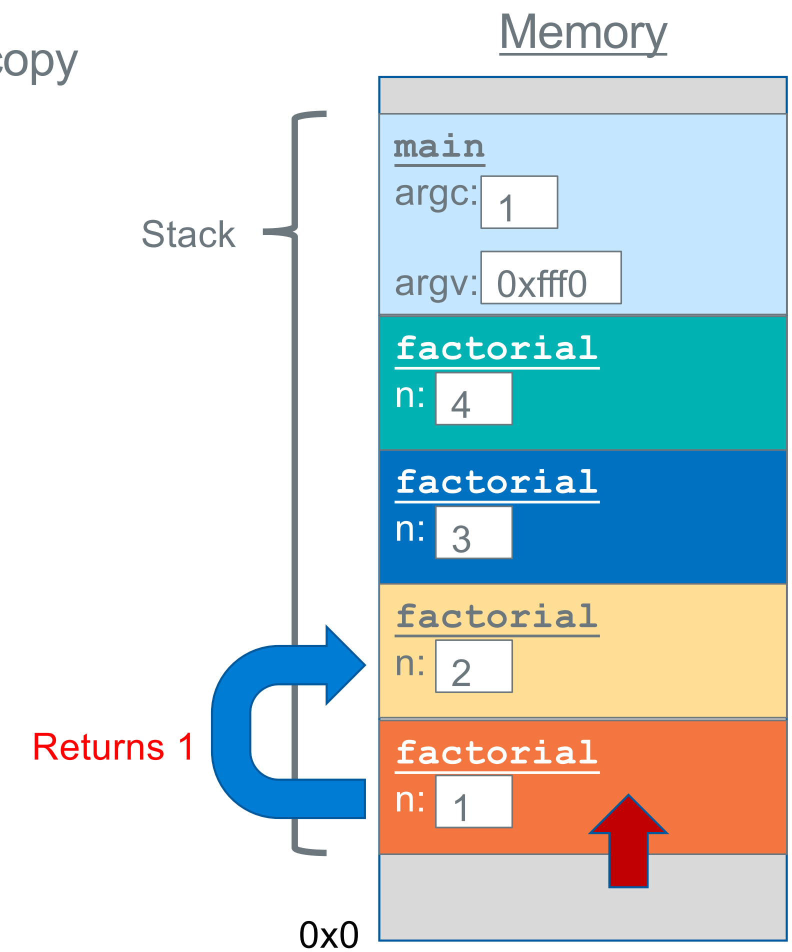


The Stack

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;
}
```

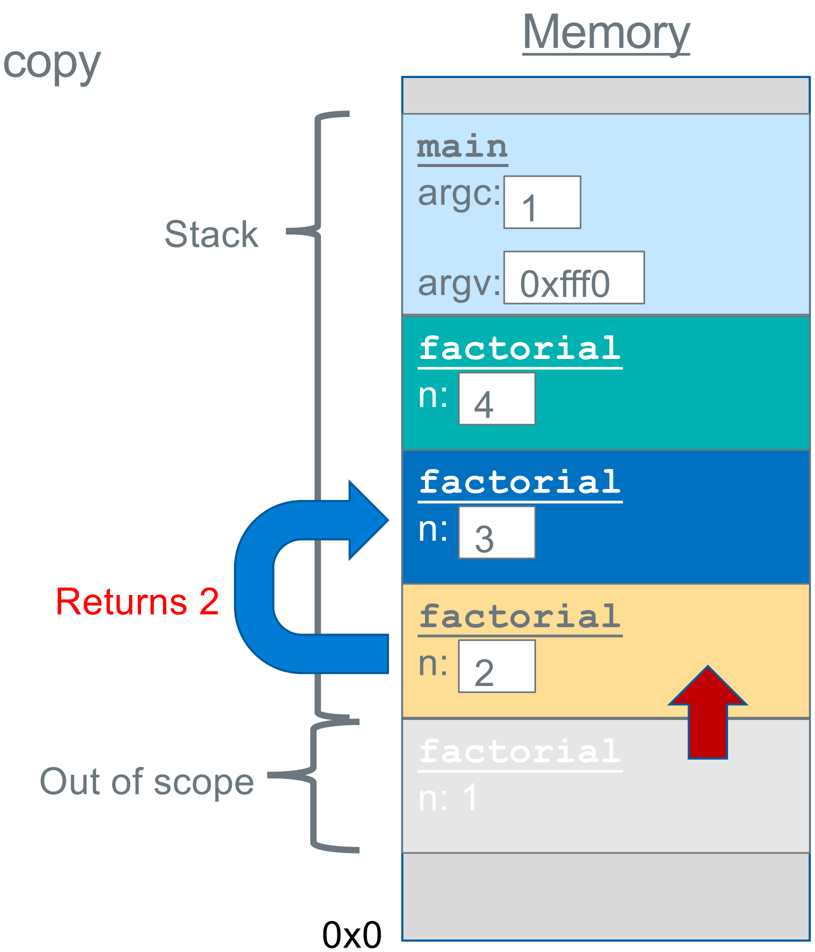


The Stack

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;
}
```

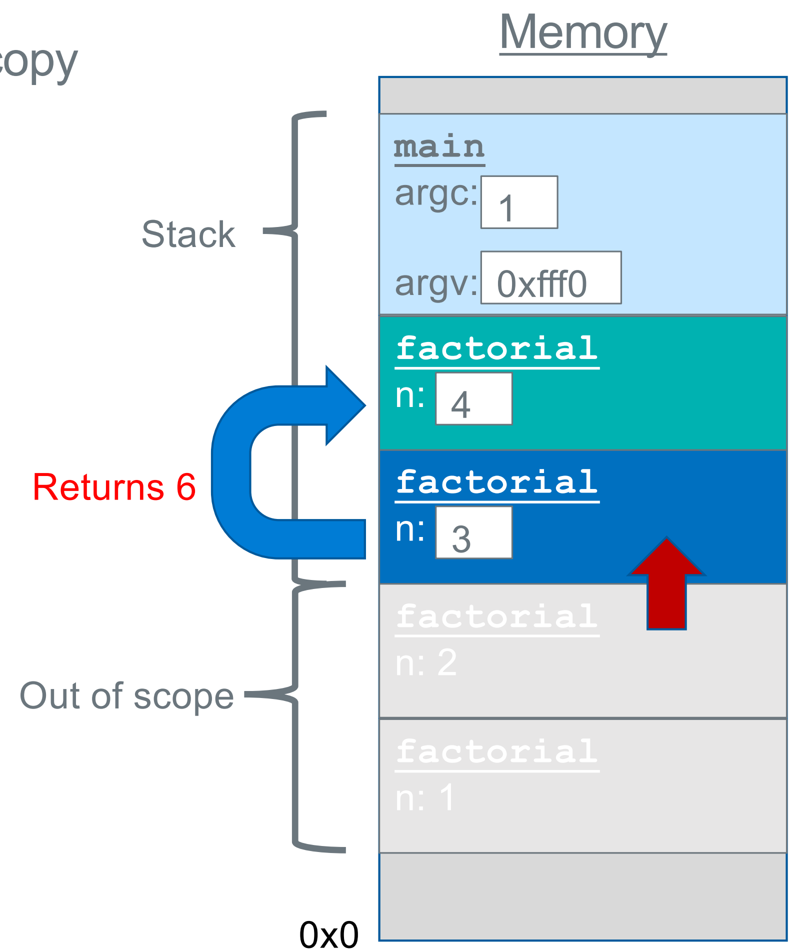


The Stack

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;
}
```

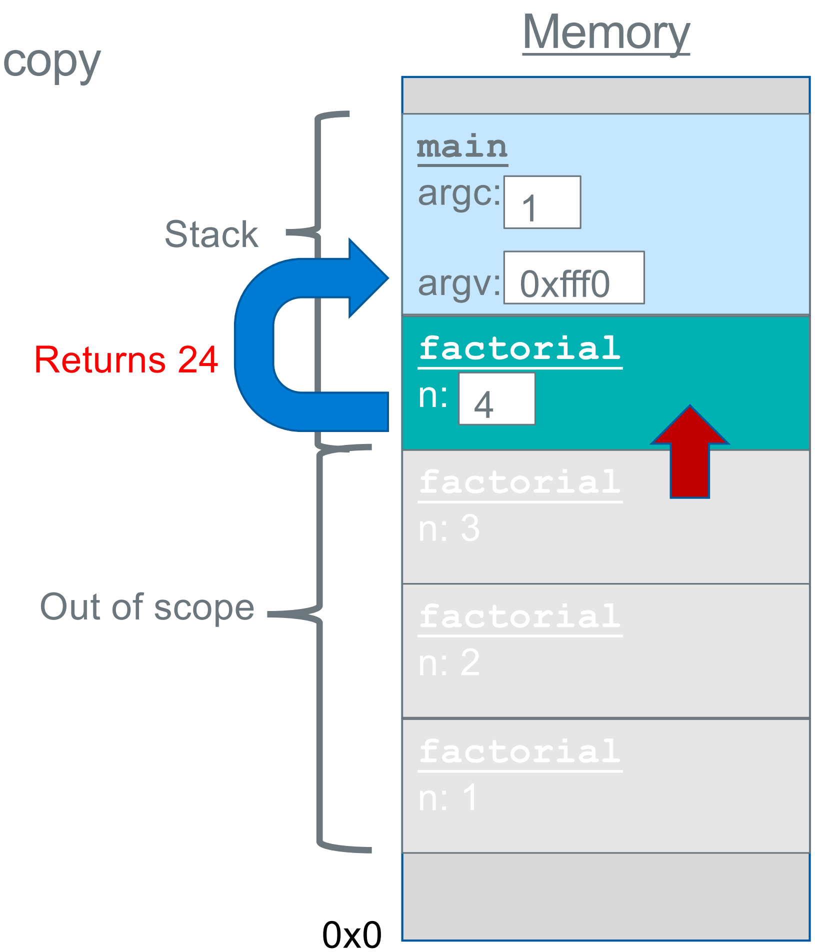


The Stack

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;
}
```

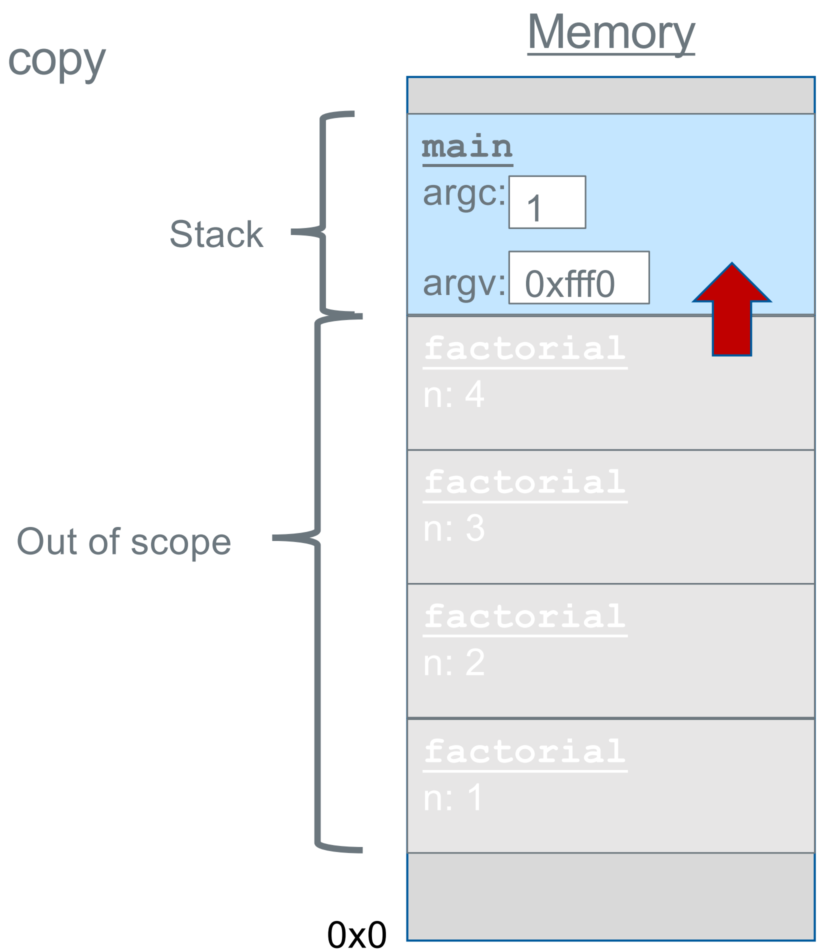


The Stack

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;
}
```

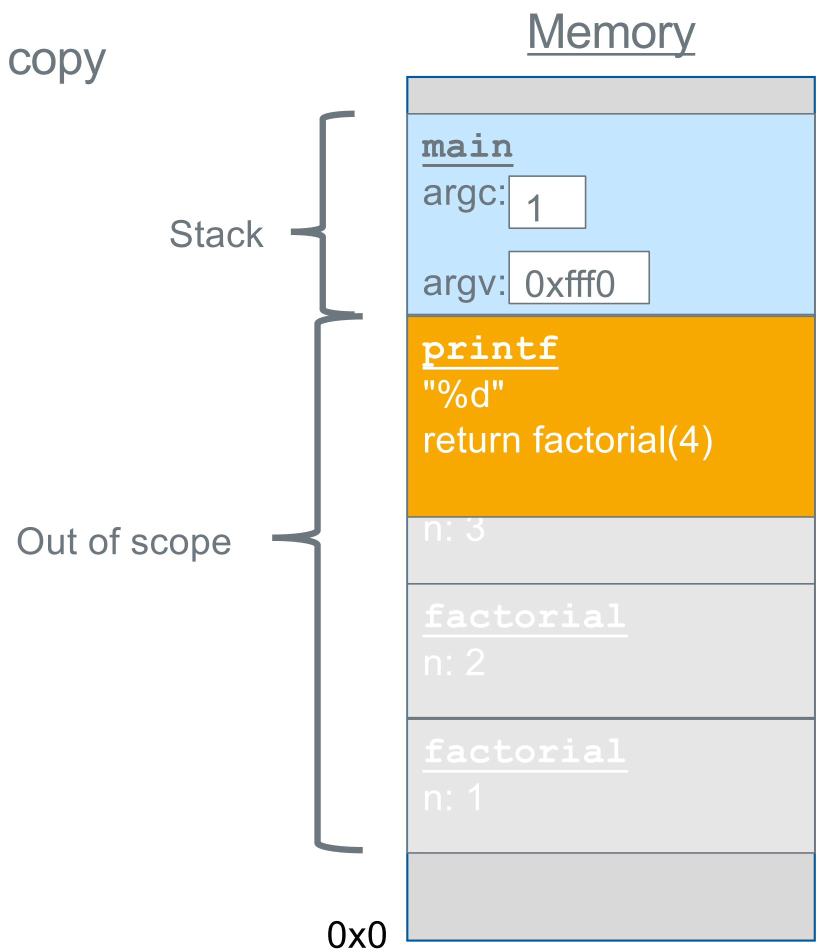


The Stack

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;
}
```



Ghost of Stack Frames Past…..

same stack frame
variable layout

```
% ./a.out
before ghost: 0 66328
after ghost: 30 300
wraith: 30 300
%
```

See how wraith has the old values left over from the prior call to ghost

```
void ghost(int n)
{
    int x;
    int y;

    printf("before ghost: %d %d\n", x, y);
    x = 10*n;
    y = 100*n;
    printf("after ghost: %d %d\n", x, y);
    return;
}

void wraith (void)
{
    int a;
    int b;

    printf("wraith: %d %d\n", a, b);
    return;
}

int main(void)
{
    ghost(3);
    wraith();
    return EXIT_SUCCESS;
}
```

Function Header and Footer Assembler Directives

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

```
Function Header
                 .text
                 .global myfunc              // make myfunc global for linking
                 .type   myfunc, %function   // define myfunc to be a function
                 .equ    FP_OFF,  4          // fp offset in main stack frame
myfunc:
                  // function prologue, stack frame setup
                  // your code
                  // function epilogue, stack frame teardown
Function Footer
                 .size myfunc, (. - myfunc)
```

`.global function_name`

- Exports the function name to other files. **Required for main function,** optional for others

`.type name, %function`

- The `.type` directive sets the **type of a symbol/label name**
- `%function` specifies that **name** is a function (name is the address of the first instruction)

`equ FP_OFF, 4`

- Used for basic stack frame setup; the number 4 will change – later slides

`.size name, bytes`

- The `.size` directive is used to set the size associated with a symbol
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- `bytes` **is best calculated as an expression: (period is the current address in a memory segment)**
  In CSE30 required use: `.size name, (. - name)`

Support For Function Calls and Function Call Return - 1

bl	imm24

Branch with Link (function call) instruction

```
bl label
```

- Function call to the instruction with the address `label` (no local labels for functions)
  - imm24 number of instructions from pc+8
- label **any function label** in the current file, or any function label that is defined as .global in any file that it is linked to
- BL **saves** the address of the instruction **immediately** following the **bl** 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:
 •                •
```

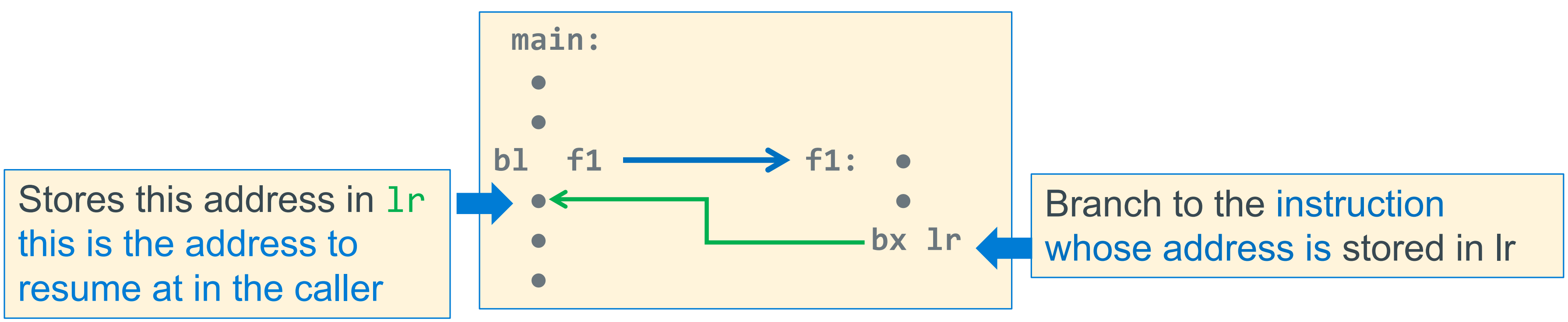
Support For Function Calls and Function Call Return - 2

bx	Rn

Branch & exchange (function return) instruction

```
bx lr // 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 **bl label**



x

bl and bx operation working together

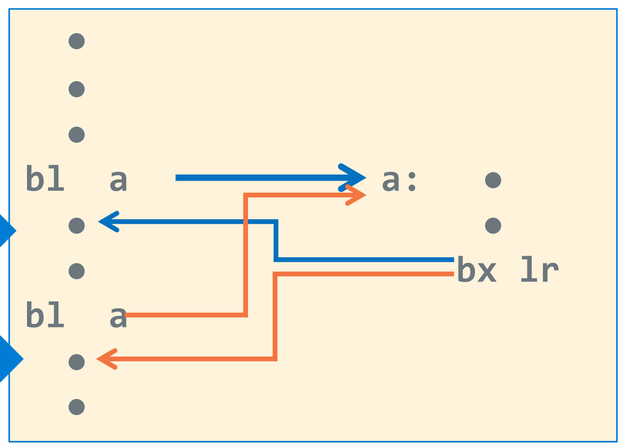
```
int main(void)
{
     a();
     // other code
     a();
     return EXIT_SUCCESS;
}
int a(void)
{
    // other code
    return 0;
}
```

```
        .text
        .type   main, %function
        .global main
         .equ    EXIT_SUCCESS, 0
main:
        // code
        bl      a               ra1 lr
ra1     // other code
        bl      a               ra2 lr

ra2     mov     r0, EXIT_SUCCESS
        // code
        bx      lr
        .size main, (. - main)


        .type   a, %function
a:
        // code
        mov     r0, 0
        // code
        bx      lr              ra2 lr
        .size a, (. - a)
```

address of next instruction is stored in lr



address of next instruction is stored in lr

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

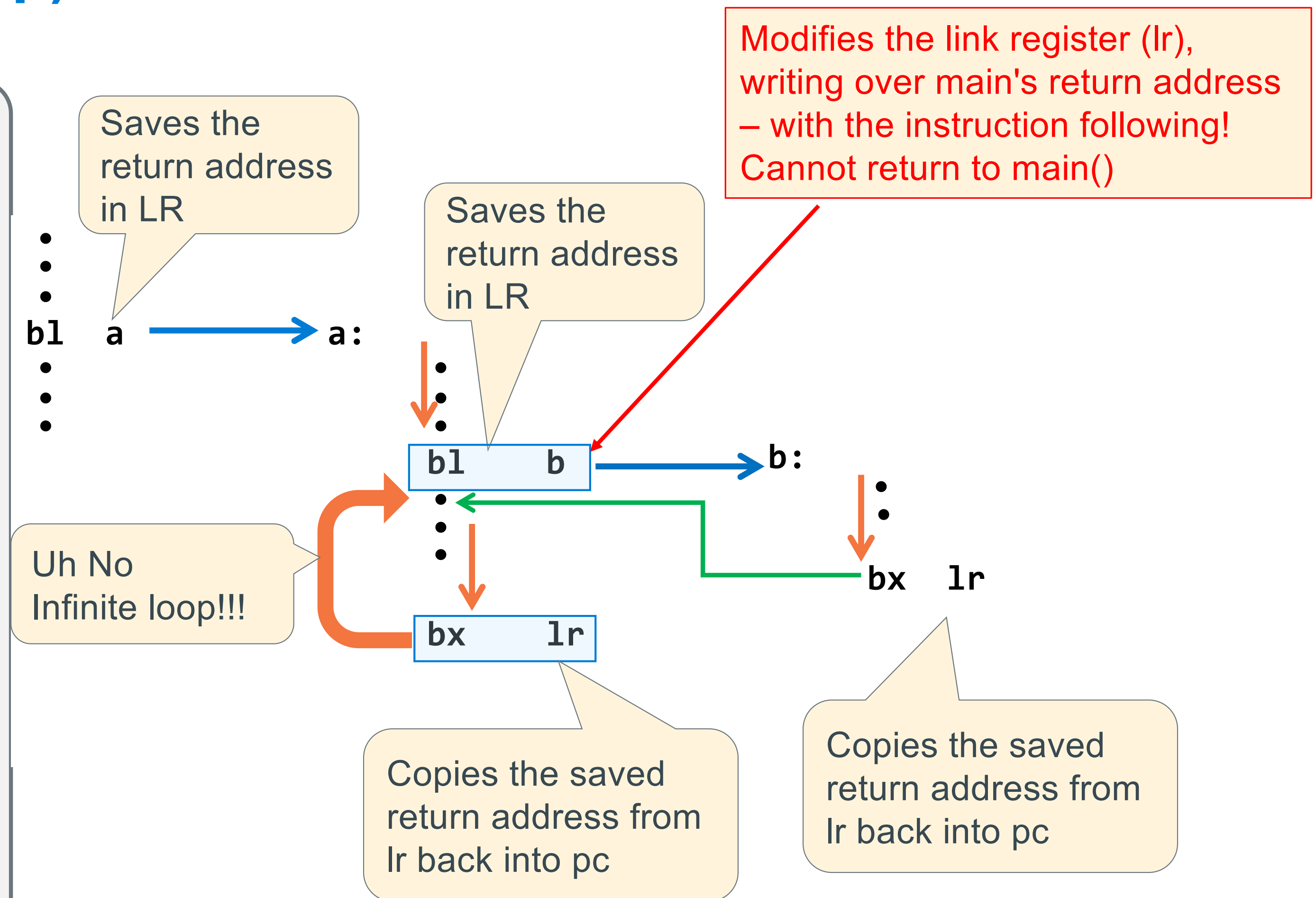
x

Preserving lr (and fp): The Foundation of a stack frame

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

int a(void)
{
    b();
    /* other code */
    return 0;
}

int b(void)
{
    /* other code */
    return 0;
}
```



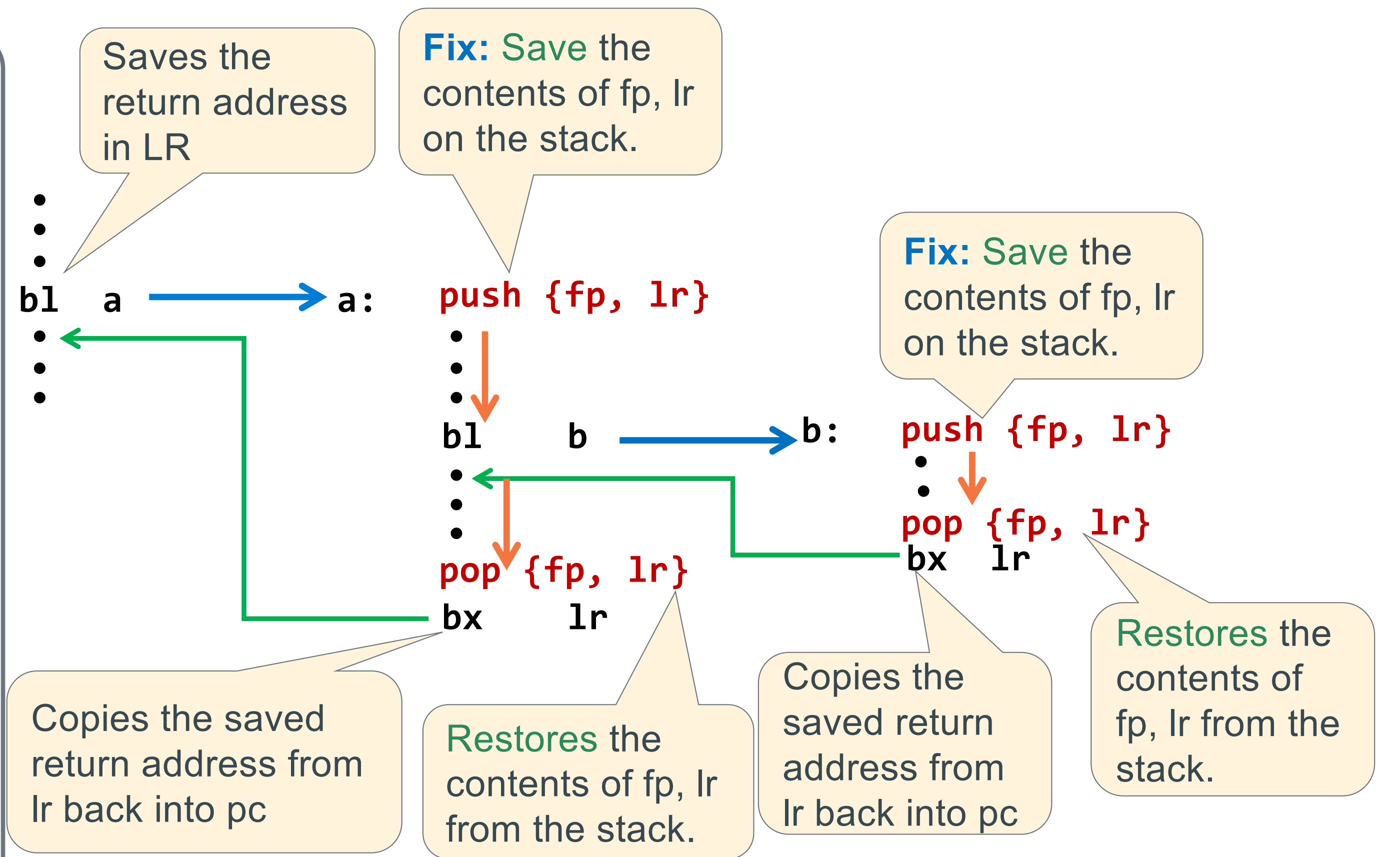
x

Preserving lr (and fp): The Foundation of a stack frame

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

int a(void)
{
    b();
    /* other code */
    return 0;
}

int b(void)
{
    /* other code */
    return 0;
}
```



The frame pointer is used to find variables on the stack – later

x

Review Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use
r0	1st parameter
r1	2nd parameter
r2	3rd parameter
r3	4th parameter

Register	Function Return Value Use
r0	8, 16 or 32-bit result, 32-bit address or least-significant half of a 64-bit result
r1	most-significant half of a 64-bit result

- Where `r0, r1, r2, r3` are arm registers, the function declaration is (first four arguments):

```
r0 = function(r0, r1, r2, r3)        // 32-bit return
r0, r1 = function(r0, r1, r2, r3)    // 64-bit return – long long
```

- Each **parameter and return value is limited to data that can fit in 4 bytes or less**
- You receive up to the first four parameters in these four registers
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes, a pointer to the parameter is passed (we will cover this later)
- **You MUST ALWAYS assume** that the called function will **alter the contents of all four registers: r0-r3**
- **Observation: When a function calls another function, the called function has the right to overwrite the first 4 parameters that were passed to it by the calling function**

Argument and Return Value Requirements

- When passing or returning values from a function you must do the following:

1. Make sure that the values in the registers r0-r3 are in their properly aligned position in the register **based on data type**
2. Upper bytes in byte and halfword values in registers r0-r3 when passing arguments and returning values are zero filled

Single Byte

	31			0
r0	0x00	0x00	0x00	0xe1

observe the zero fill

Single Halfword

	31			0
r0	0x00	0x00	0xe3	0xe1

observe the zero fill

Full Word

	31			0
r0	0x87	0x65	0xe3	0xe1

Preserved Registers: Protocols for Use

Register	Function Call Use	Function Body Use	Save before use Restore before return
r4-r10		contents preserved across function calls	Yes
r7	os system call number	contents preserved across function calls	Yes
r12, r15	Done not use		

- **Function Call Spec**:

  Preserved registers **will not be changed** by any function you call

- **Interpretation**: 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)

x

Preserved Registers: When to Use?

Register	***Function Call Use***	***Function Body Use***	***Save before use Restore before return***
r4-r10		contents preserved across function calls	Yes
r7	os system call number	contents preserved across function calls	Yes
r12, r15	Done not use		

- When to use a preserved register in a function you are writing:

1. Values that you want to protect from being changed by a function call
   a) Local variables stored in registers
   b) Parameters passed to you (in **`r0-r3`**) that you need to continue to use after calling another function
2. Need more than **`r0-r3`** whether you call another function or not

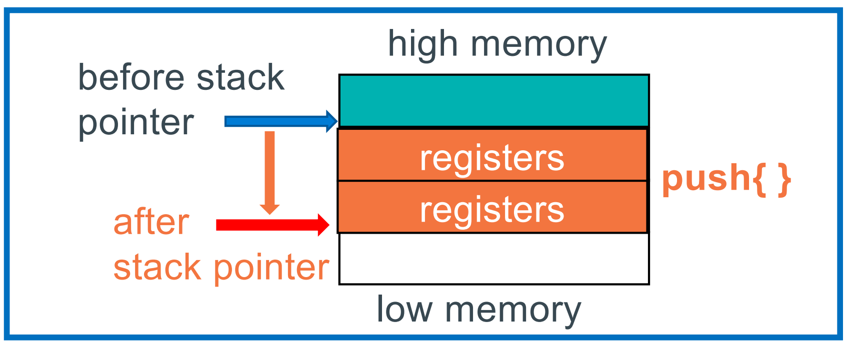
   Options are:
   a) preserved register *or*
   b) stack local variable (later slides)

x

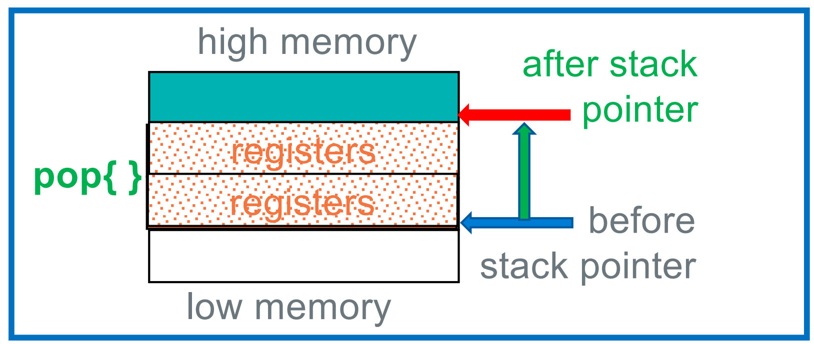
Preserving and Restoring Registers by copying to/from Stack Moves sp to allocate (Push) or deallocate (pop) stack space

Operation	*Pseudo Instruction (Use in CSE30)*	ARM instruction (reference only)	*Operations*
Push registers onto stack Function Entry	push *{reg list}*	`stmfd sp!, {reg list}`	`sp = sp – 4 × #registers` `Copy registers to mem[sp]`
Pop registers from stack Function Exit	pop *{reg list}*	`ldmfd sp!, {reg list}`	`Copy mem[sp] to registers,` `sp = sp + 4 × #registers`

push (multiple register `str` to memory operation)



push (multiple register `ldr` from memory operation)



x

Preserving and Restoring Registers on the Stack
Function entry and Function exit

Operation	*Pseudo Instruction*	*Operation*
Push registers Function Entry	push *{reg list}*	sp = sp – 4 × #registers Copy registers to mem[sp]
Pop registers Function Exit	pop *{reg list}*	Copy mem[sp] to registers, sp = sp + 4 × #registers

- Where **`{reg list}`** is a **list of registers** in numerically increasing order

  `example: push {r4-r10, fp, lr}`

- Registers cannot be: (1) duplicated in the list, nor be (2) listed out of numeric order
- Register ranges can be specified `{r4, r5, r8-r10, fp, lr}`

x

Preserving and Restoring Registers on the Stack
Function entry and Function exit

Operation	*Pseudo Instruction*	*Operation*
Push registers Function Entry	`push {reg list}`	sp = sp – 4 × #registers Copy registers to mem[sp]
Pop registers Function Exit	`pop {reg list}`	Copy mem[sp] to registers, sp = sp + 4 × #registers

- Where `{reg list}` is a **list of registers** in numerically increasing order

  example: `push {r4-r10, fp, lr}`

- Registers cannot be: (1) duplicated in the list, nor be (2) listed out of numeric order
- Register ranges can be specified `{r4, r5, r8-r10, fp, lr}`
- Do not push/pop r12, r13, or r15
  - the last two registers must always be fp , lr

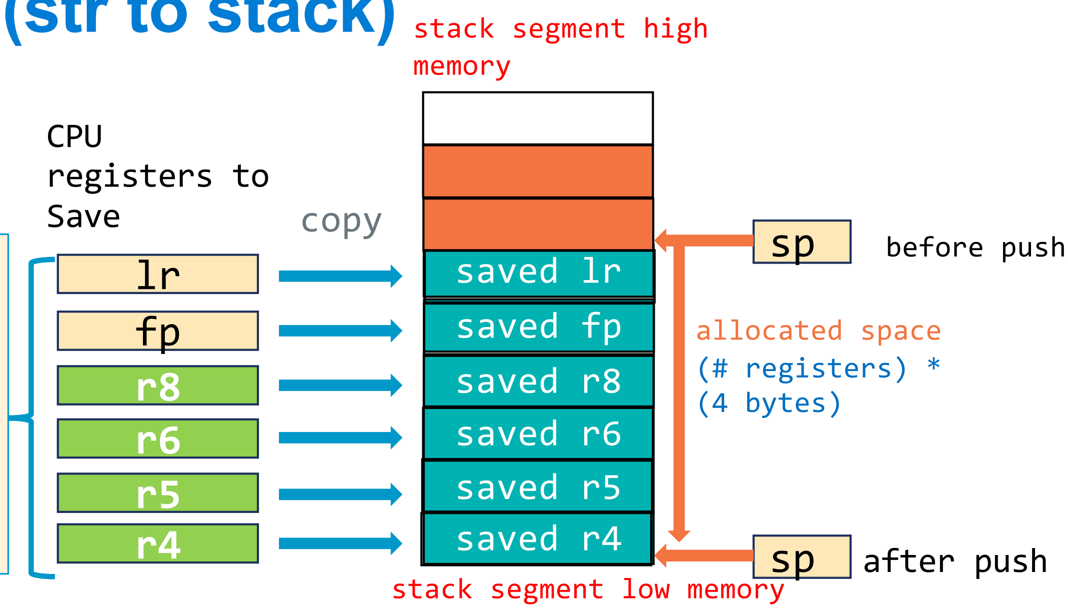
x

push: Multiple Register Save (str to stack)

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

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

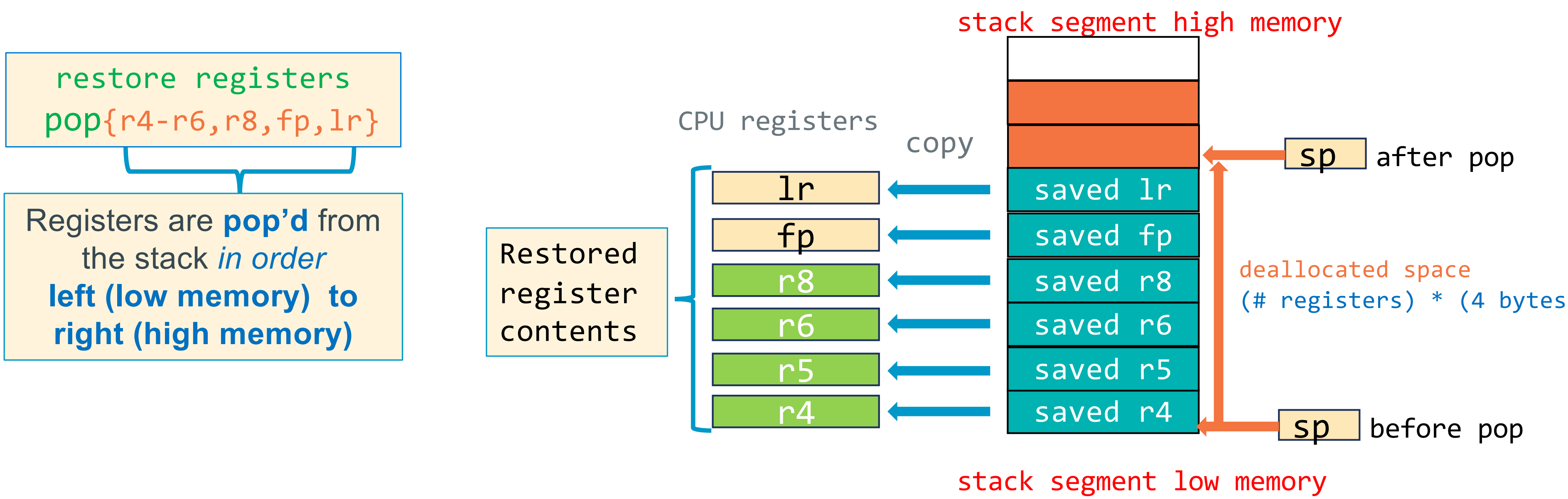
If you have no stack variables (later) then always push an **EVEN** number of registers!



- **`push`** copies the contents of the `{reg list}` to stack segment memory
- **`push`** **Also** <u>subtracts</u> (# of registers saved) * (4 bytes) from the **`sp`** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

x

pop: Multiple Register Restore (ldr from stack)



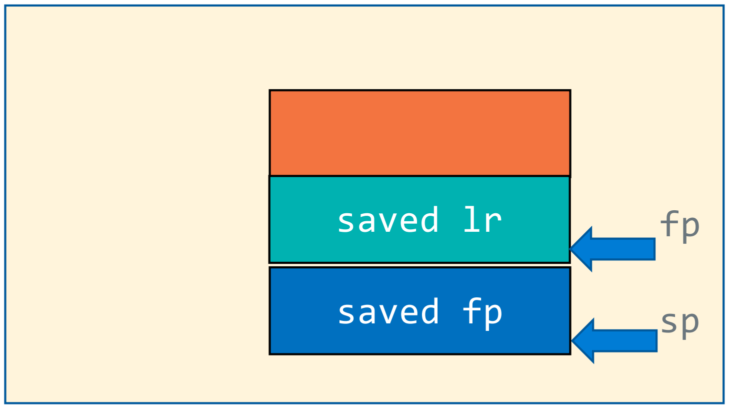
- `pop` copies the contents of stack segment memory to the `{reg list}`
- `pop` **adds:** (# of registers restored) * (4 bytes) to `sp` to ***deallocate*** space on the stack
  - sp = sp + (# registers restored * 4)
- **`Remember:`** `{reg list}` <u>must be the same</u> in both the `push` and the corresponding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

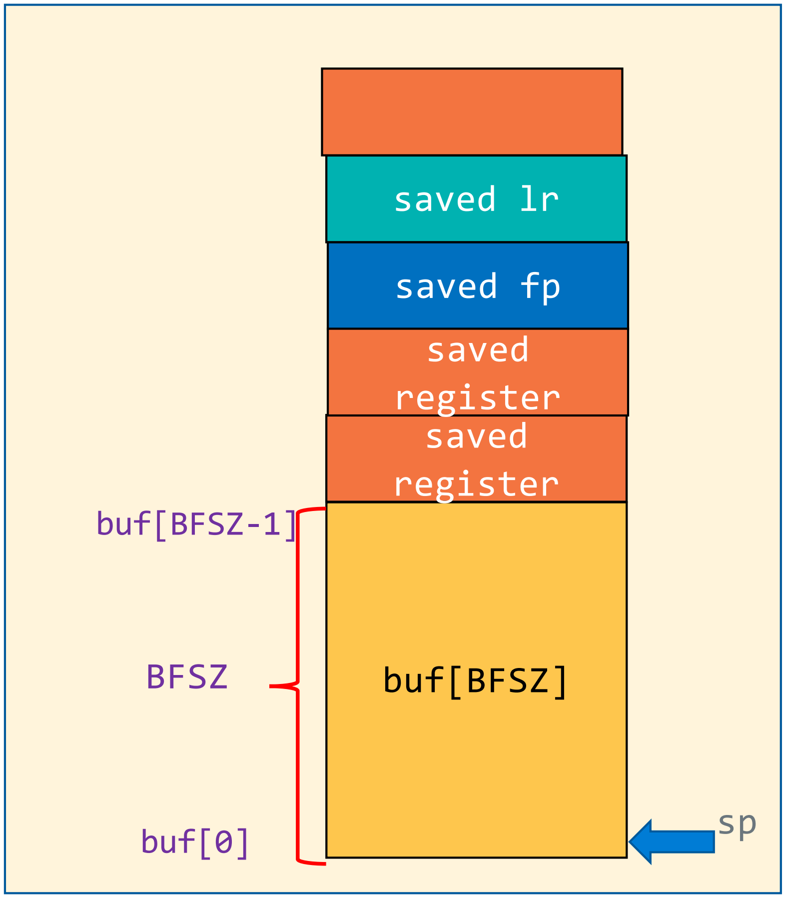
x

Local Variables are Part of Each Stack Frame

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

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





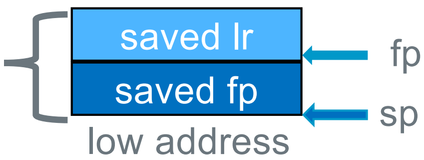
x

Stack Frame (Arm Arch32 Procedure Call Standards)

Stack Frame Requirements

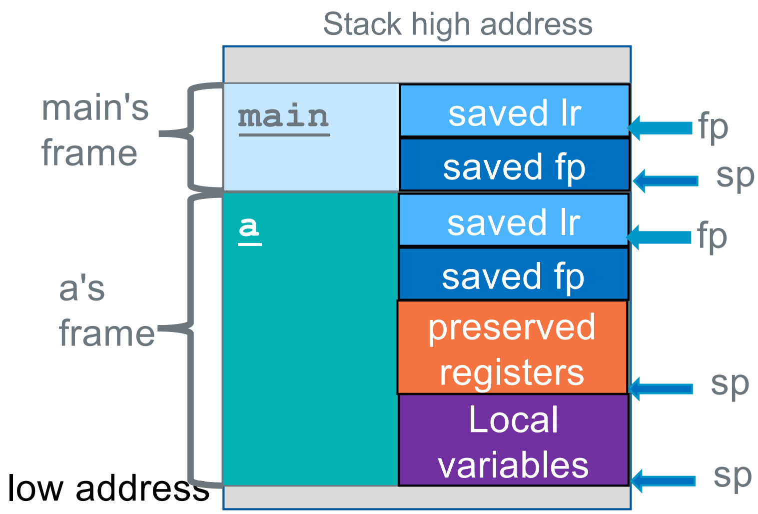
- **Minimal frame: at function entry** `push {fp, lr}`
- `sp` points at top element in the stack (lowest byte address)
- `fp` points at the `lr` copy stored in the current stack frame
- **Stack frames MUST ALWAYS BE aligned to 8-byte addresses**
  - So, this must always be true: sp % 8 == 0

lr & fp must always be at top of frame



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

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



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

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

- Function entry (Function **Prologue**):
  1. create frame (subtract
  2. save preserved registers
  3. allocate space for locals (subtracts from sp)
- Function return (Function **Epilogue**):
  1. deallocate space for locals (adds to sp)
  2. restores preserved registers
  3. removes the frame

Note slide has builds

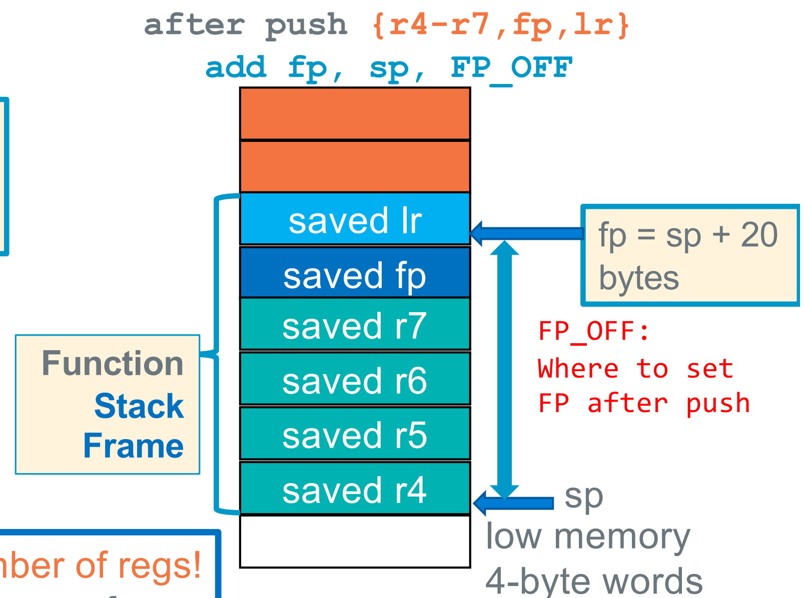
x

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

```
    // other code etc
     .equ    FP_OFF,  20
main:
        push    {r4-r7, fp, lr}
        add     fp, sp, FP_OFF
        …….
        sub     sp, fp, FP_OFF
        pop     {r4-r7, fp, lr}
        bx      lr
```

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

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



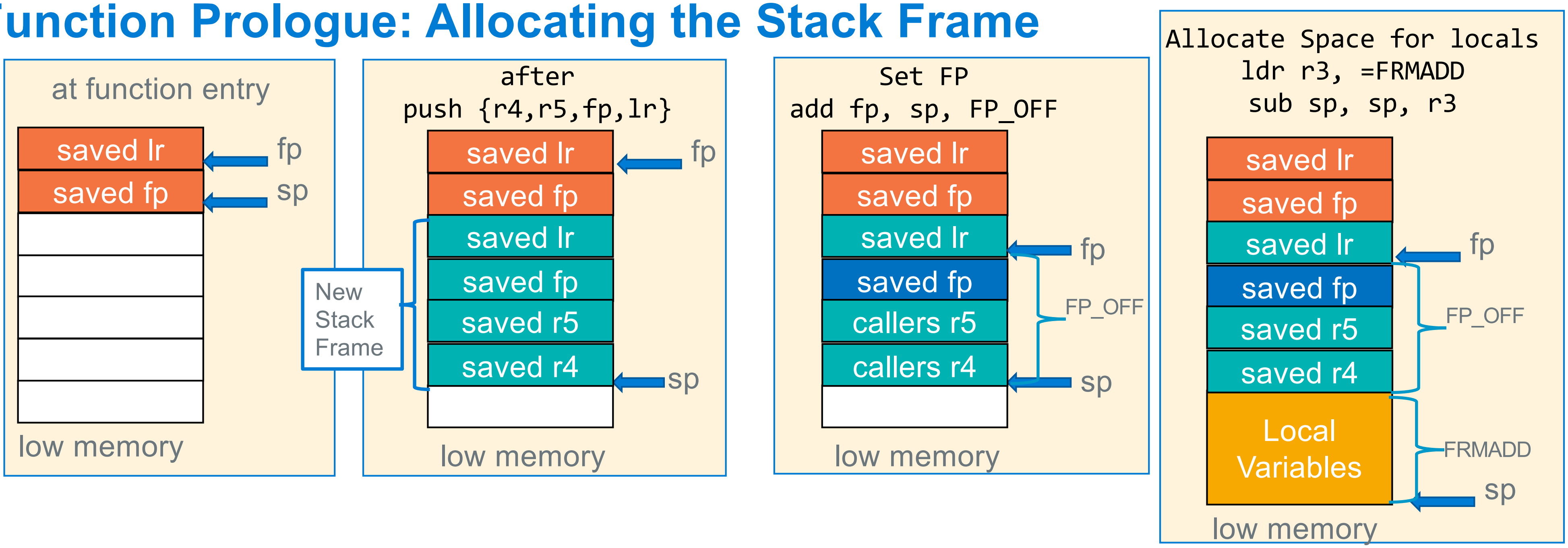
# regs saved	FP_OFF in Bytes
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

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

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

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

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

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

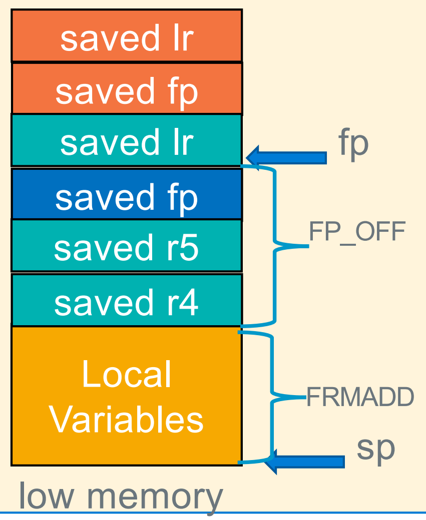
Allocate Space for Local Variables

Part of function prologue

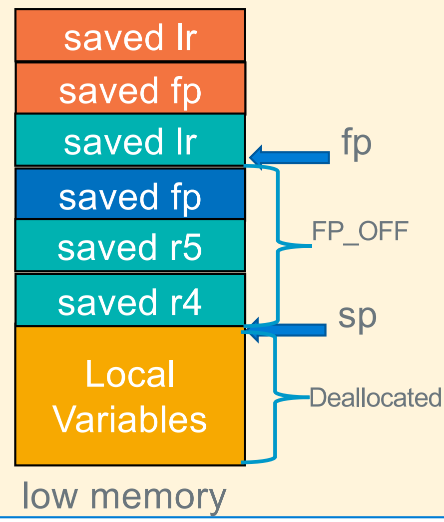
x

Function Epilogue: Deallocating the Stack Frame

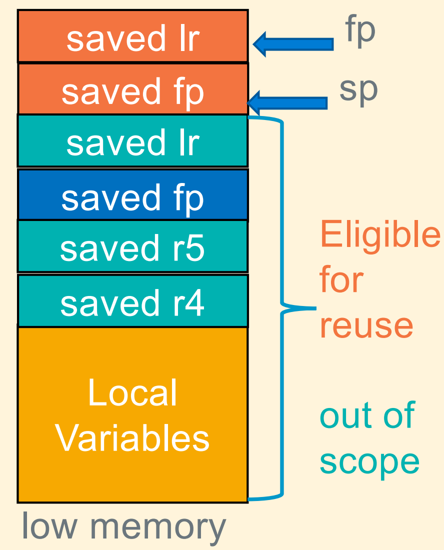
```
Stack frame while during
function body execution
```




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




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



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

Move SP back to where it was after the push so the pop works (this deallocates the local variables)

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

Part of function epilogue

x

Local Variables on the Stack

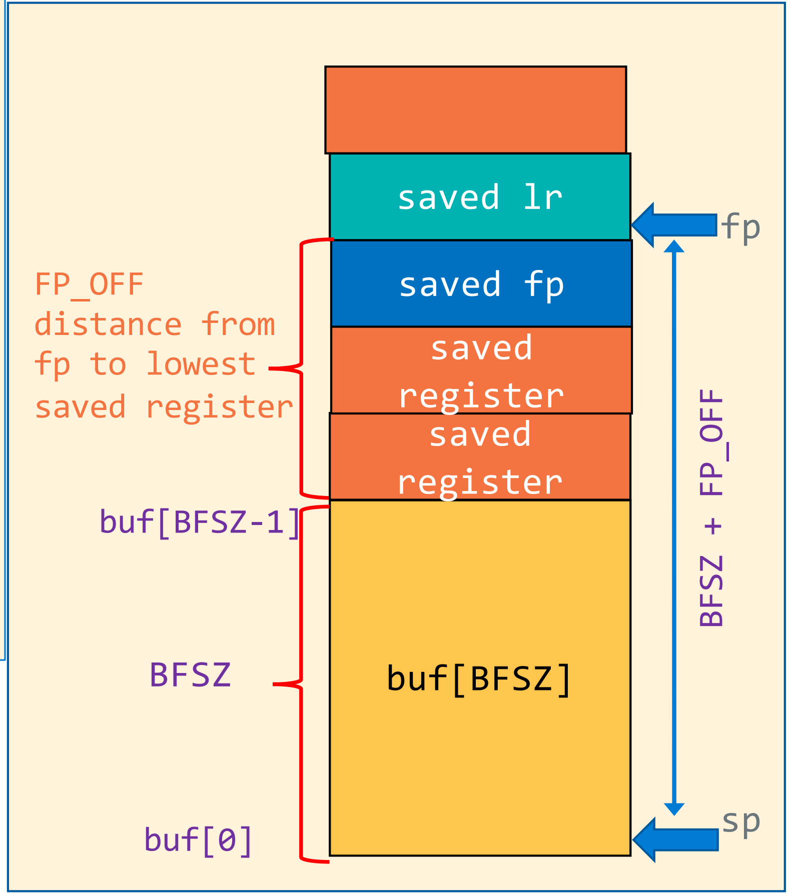
- Local variables are on the stack below the lowest numbered saved register
- frame pointer is used as a **pointer** to stack variables
- fp is the base register in ldr and str instructions
- Example load buf[0] into r4

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

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

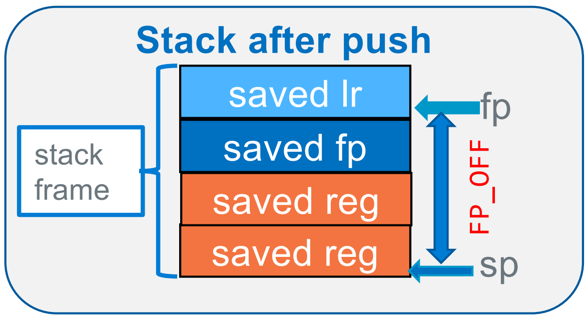
```
ldrb r4, [fp, -16]
```

1. Calculate how much additional space is needed by local variables
2. **After the register save push, Subtract from the sp** the size of the variable in bytes (+ padding - later slides)

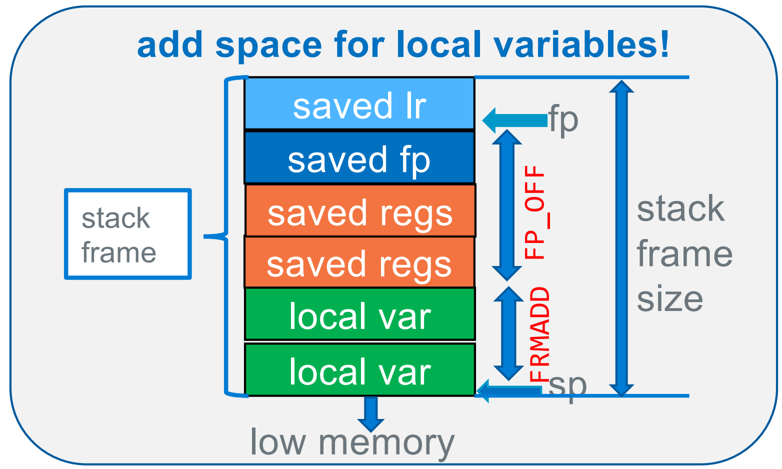


x

Function prologue with local variables




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

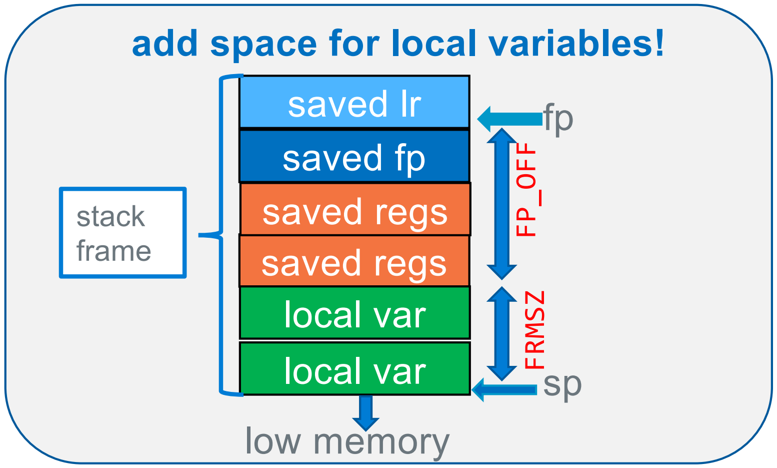


- move the sp to allocate space on the stack for local variables and outgoing parameters (later)

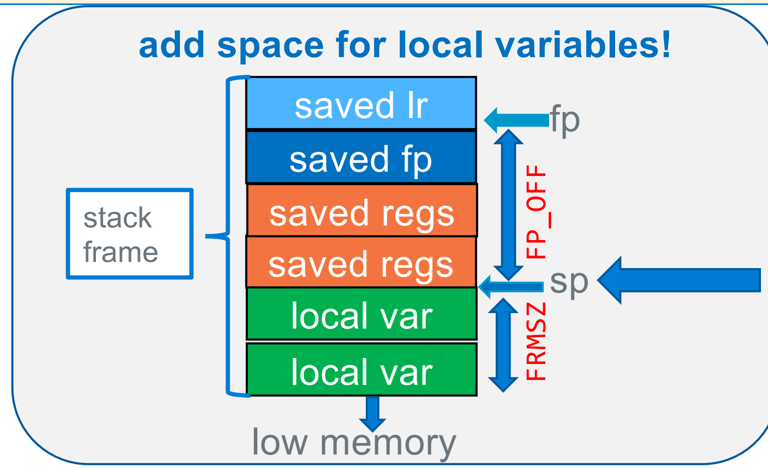
```
    .equ    FRMADD, 8
    push    {fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD // frames may be large
    sub     sp, sp, r3
  // your code
```


x

Function epilogue with local variables



- For pop to restore the registers correctly:
  - sp must point at the last saved preserved register put on the stack by the save register operation: the push




```
.equu    FRMADD, 8
push     {fp, lr}
add      fp, sp, FP_OFF
ldr      r3, =FRMADD
sub      sp, sp, r3
  // your code

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

bx       lr  // func return
```

- Return the **sp** (using the **fp**) to the same address it had after the push operation
  sub sp, fp, FP_OFF
- this works no matter how much space was allocated in the prologue

x