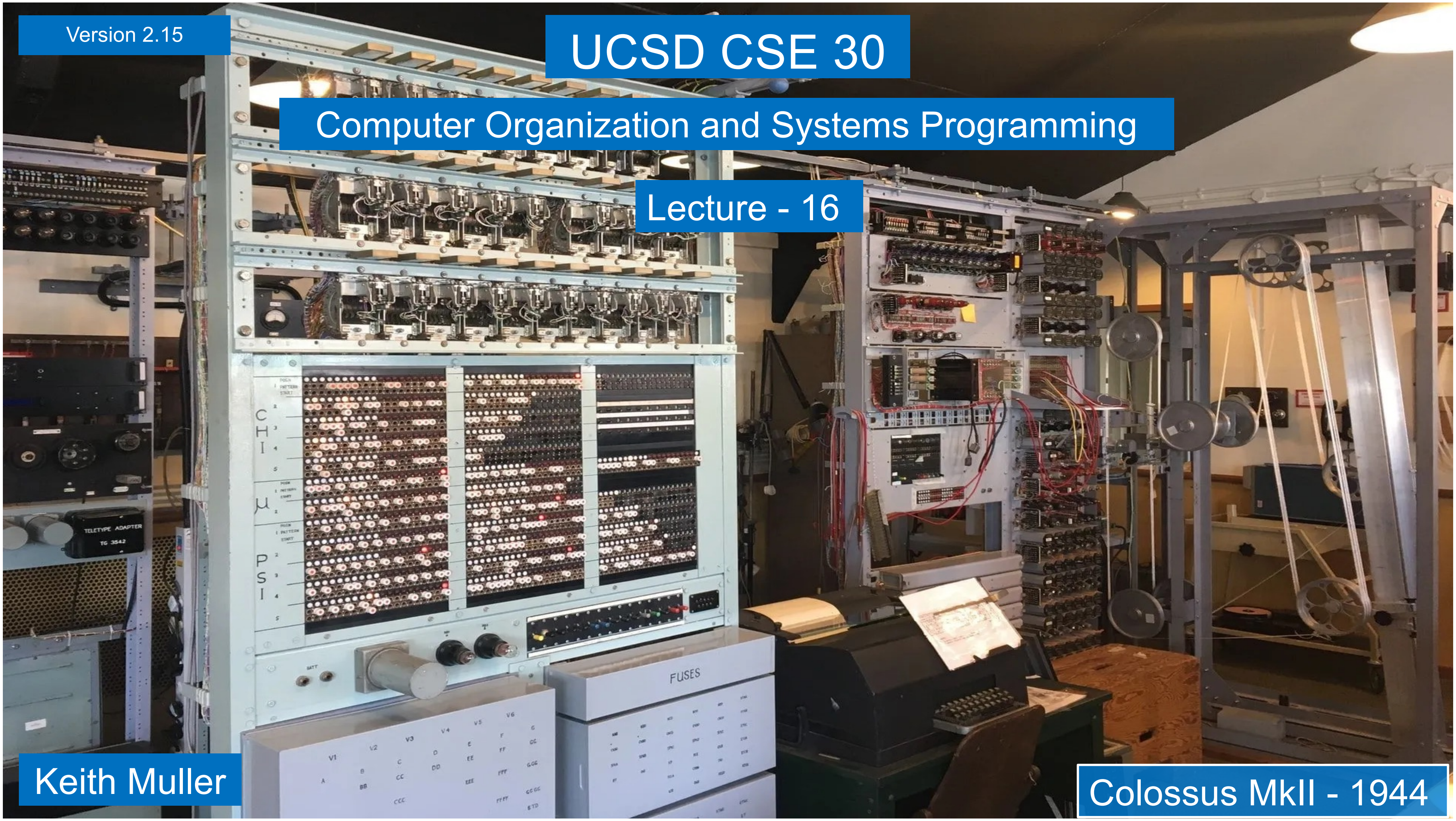
Version 2.15

UCSD CSE 30

Computer Organization and Systems Programming

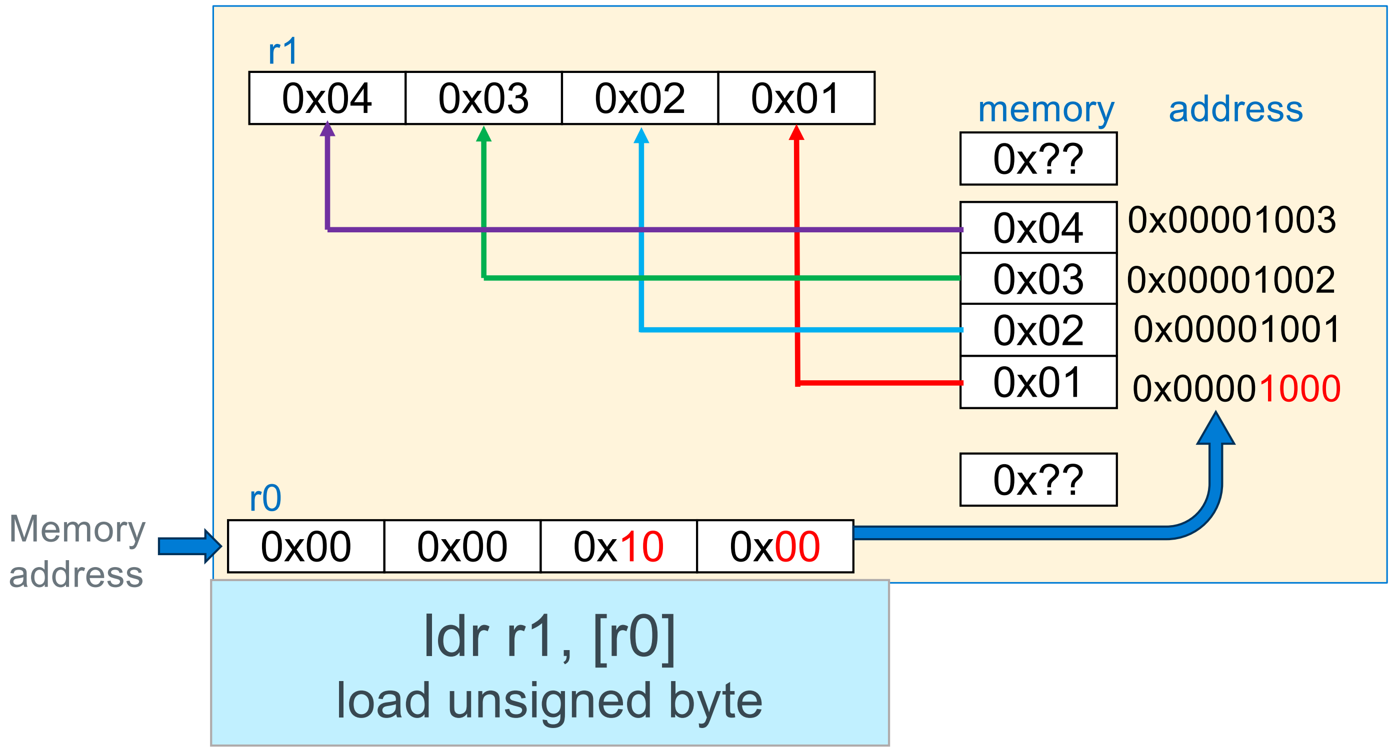
Lecture - 16



Keith Muller

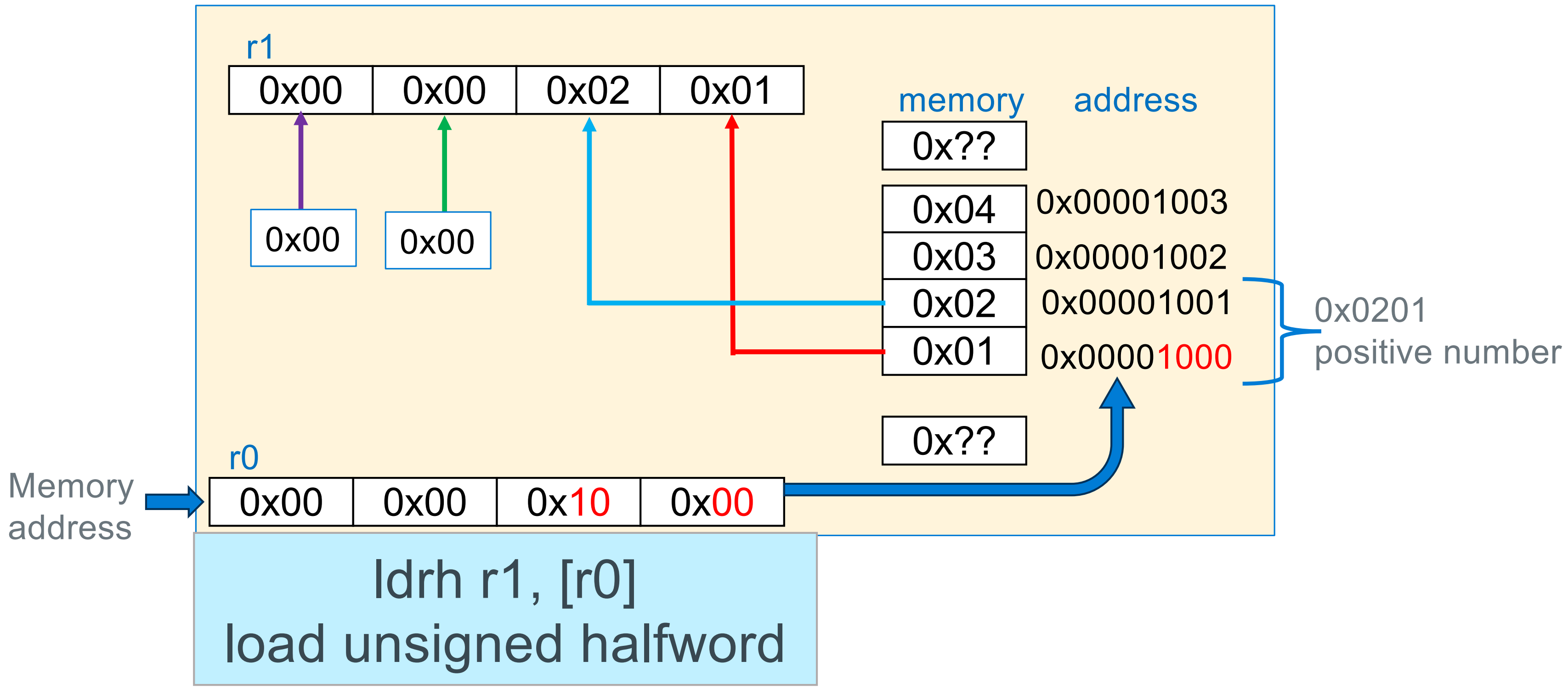
Colossus MkII - 1944

Loading 32-bit Registers From Memory, 32-bit



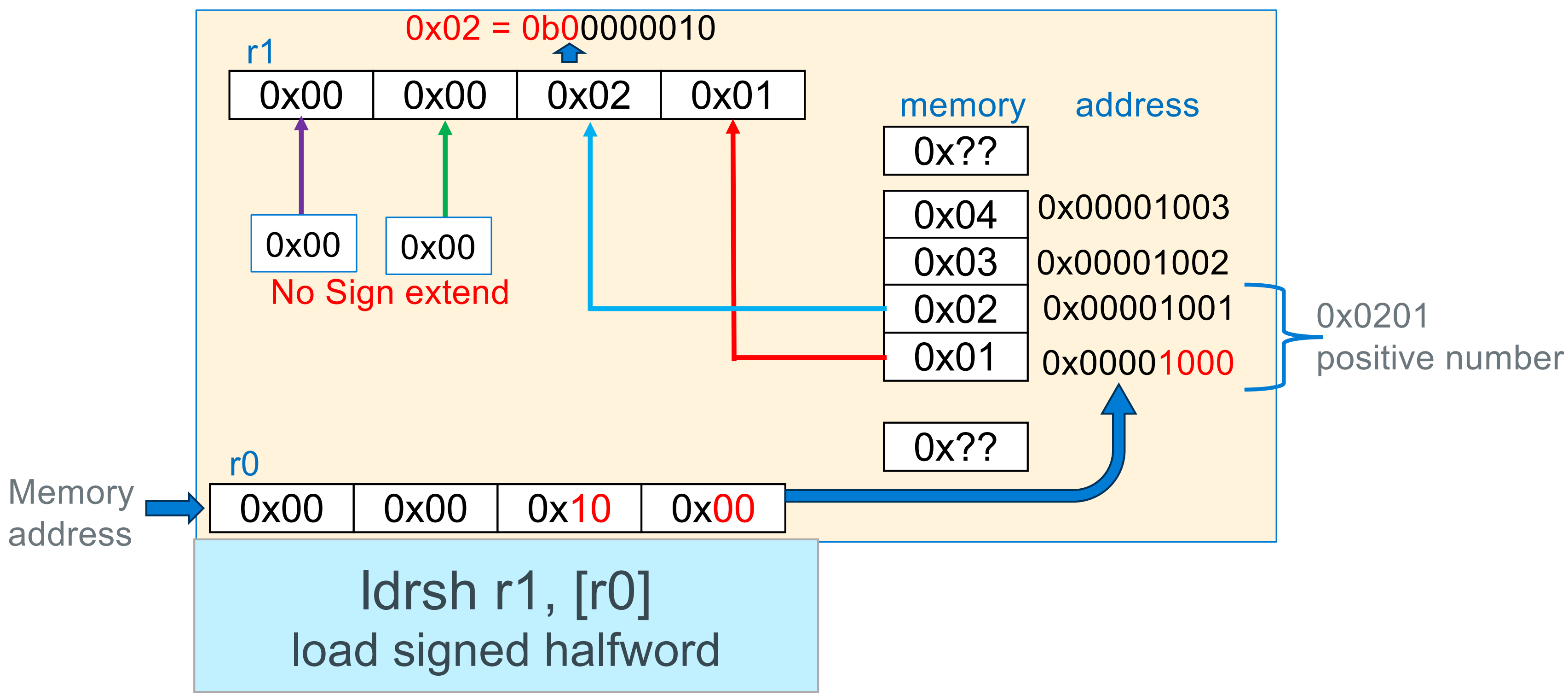
x

Loading 32-bit Registers From Memory, 16-bit



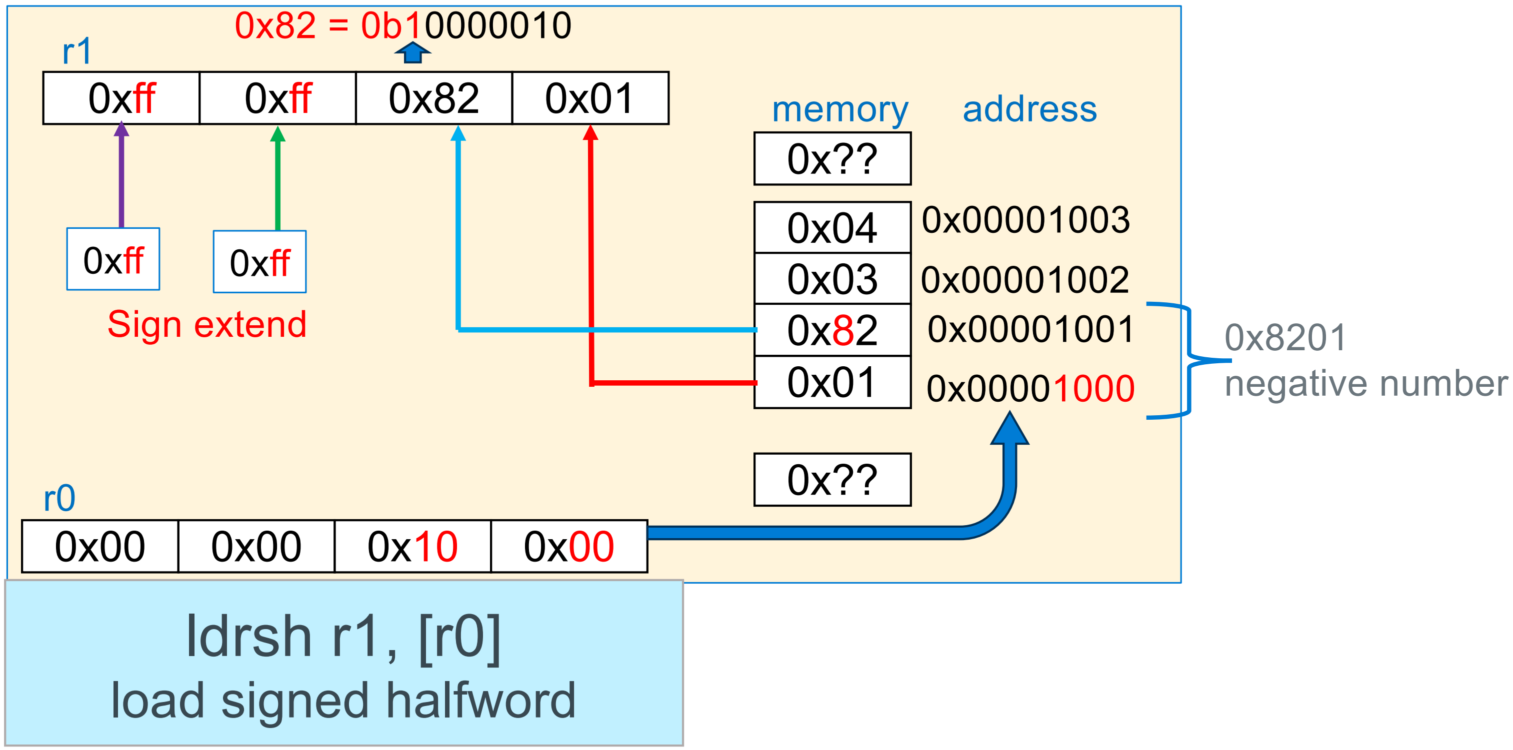
x

Loading 32-bit Registers From Memory, 16-bit



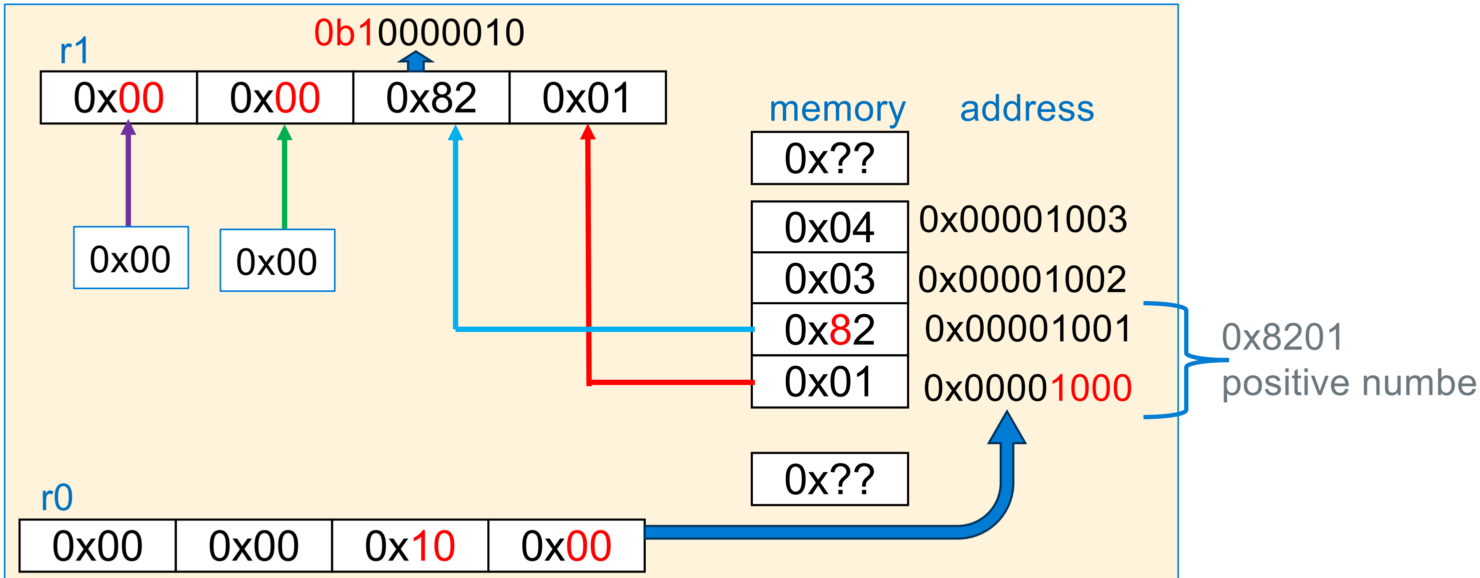
x

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



x

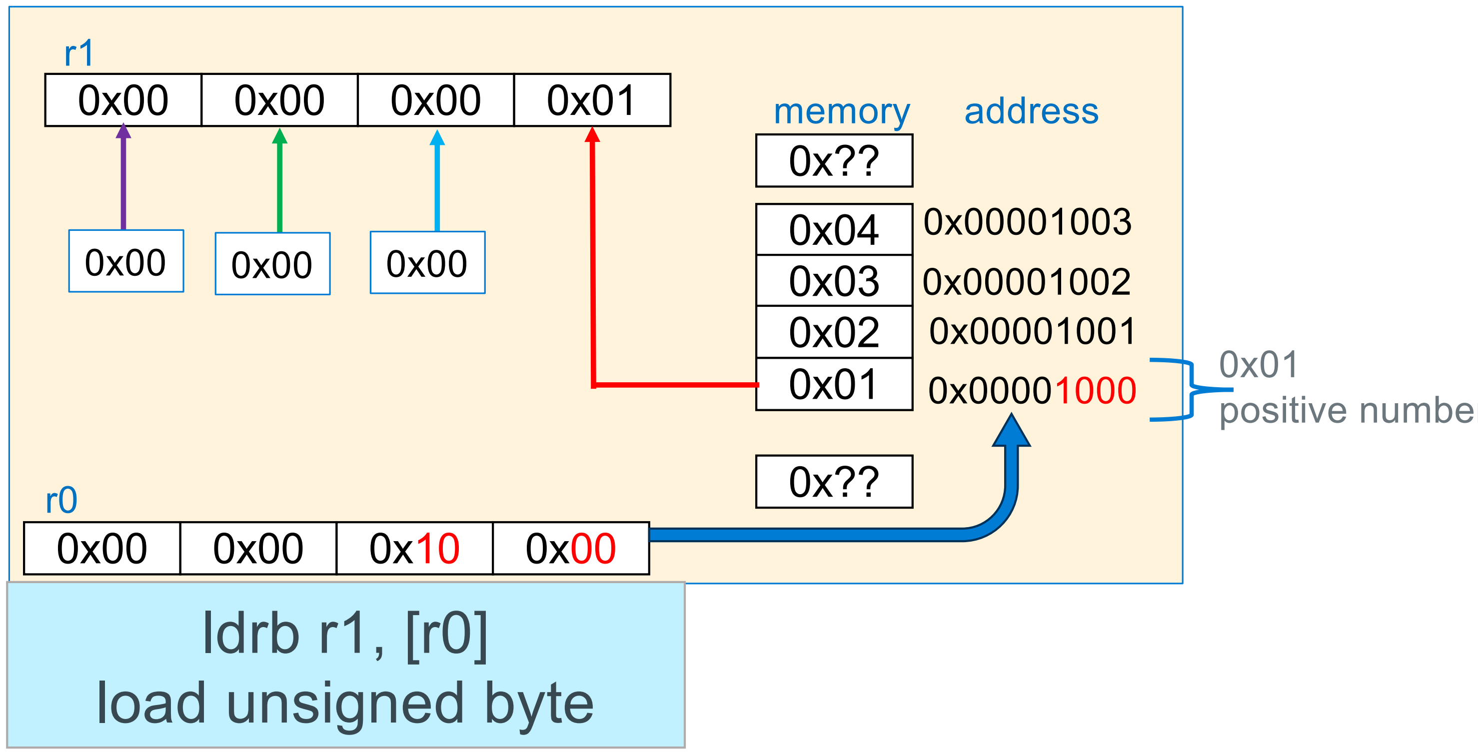
Loading 32-bit Registers From Memory, 16-bit Unsigned



ldrh r1, [r0]
load unsigned halfword

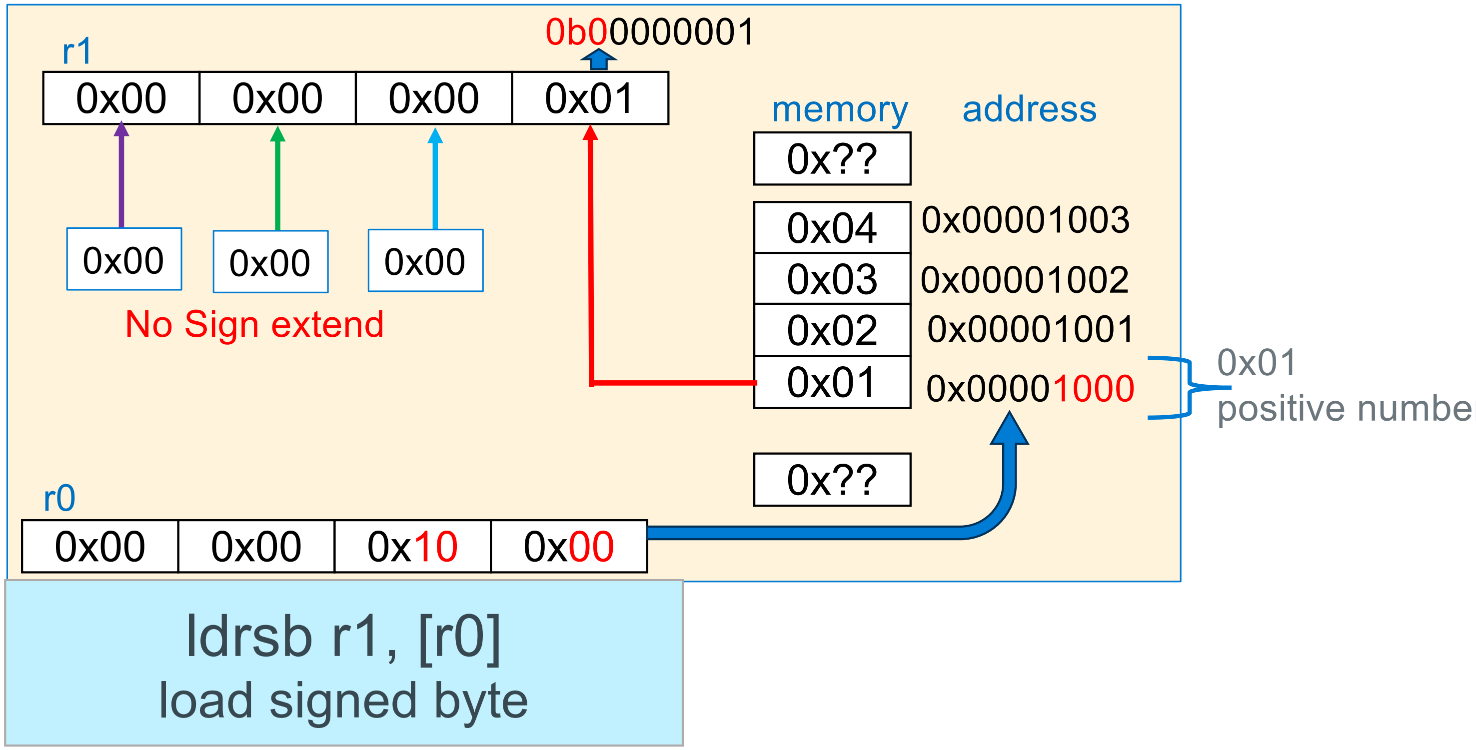
x

Loading 32-bit Registers From Memory, 8-bit



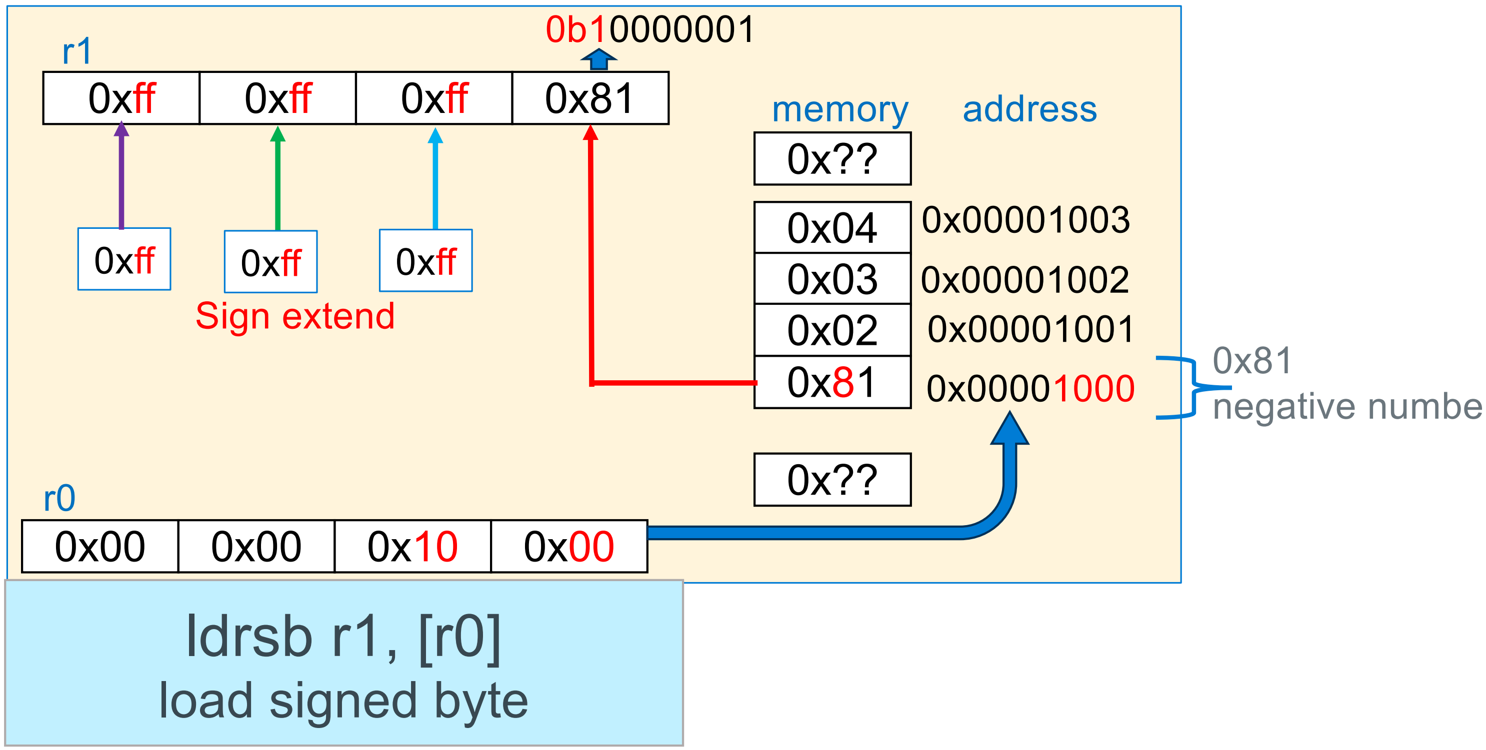
x

Loading 32-bit Registers From Memory, 8-bit



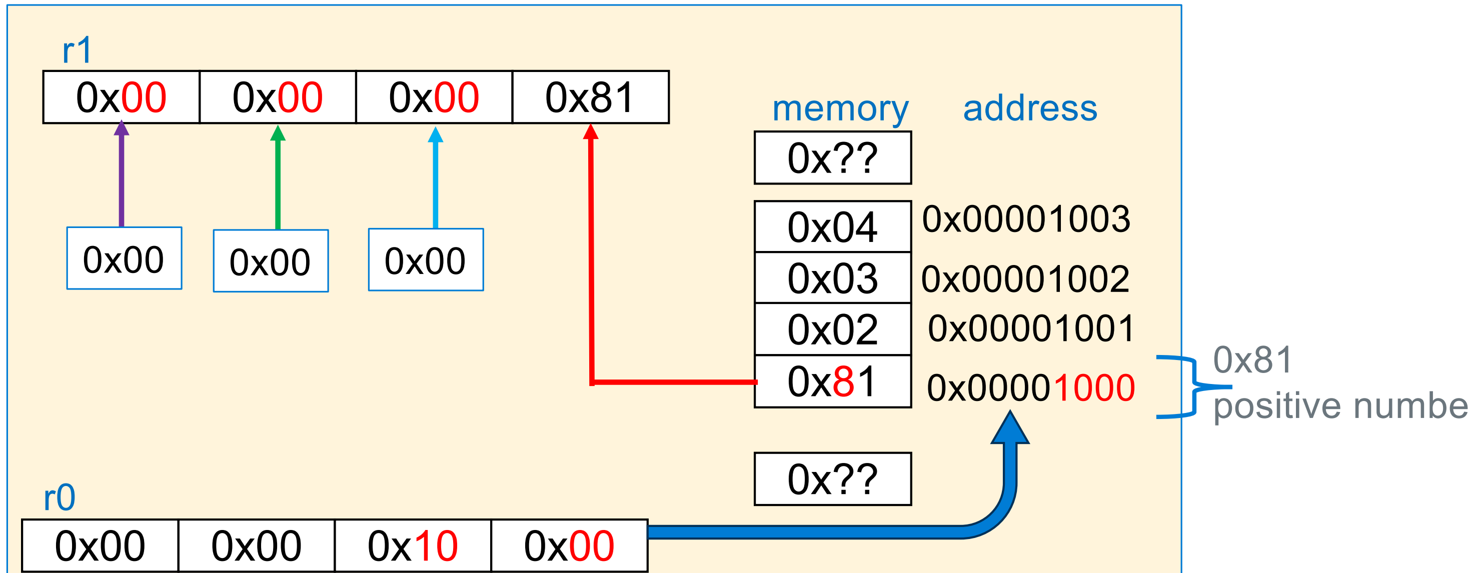
x

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



x

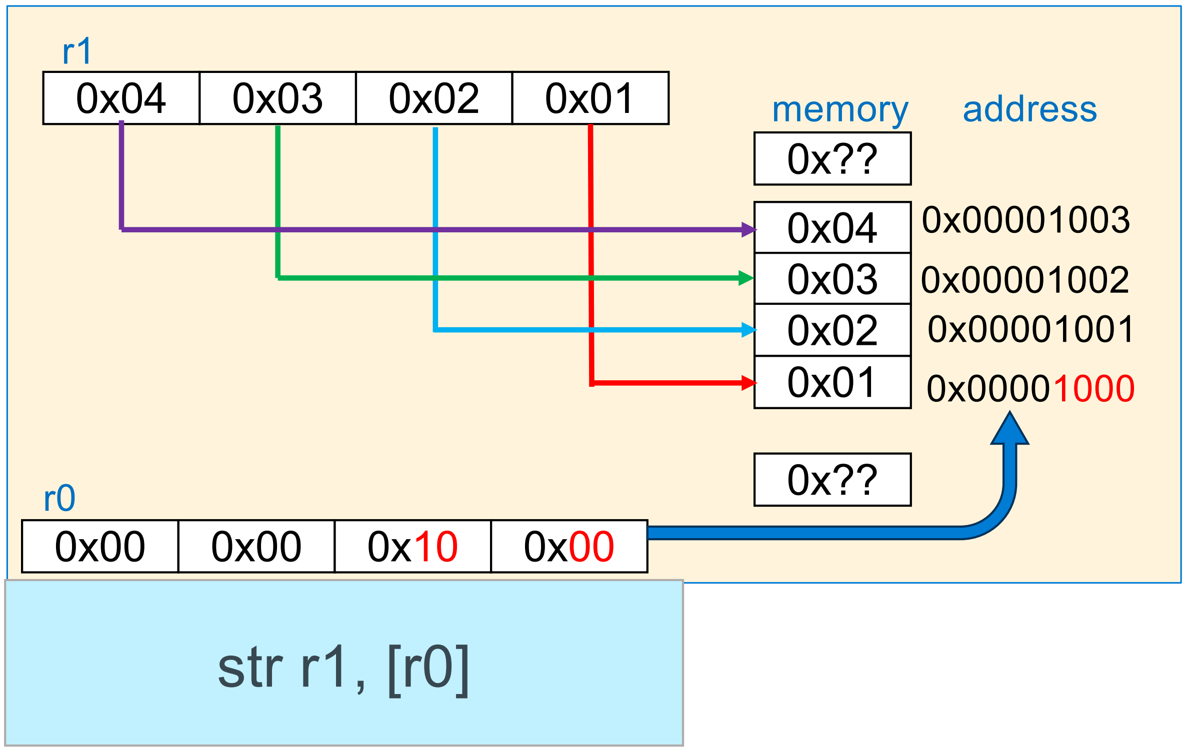
Loading 32-bit Registers From Memory, 8-bit Signed



ldrb r1, [r0]
load unsigned byte

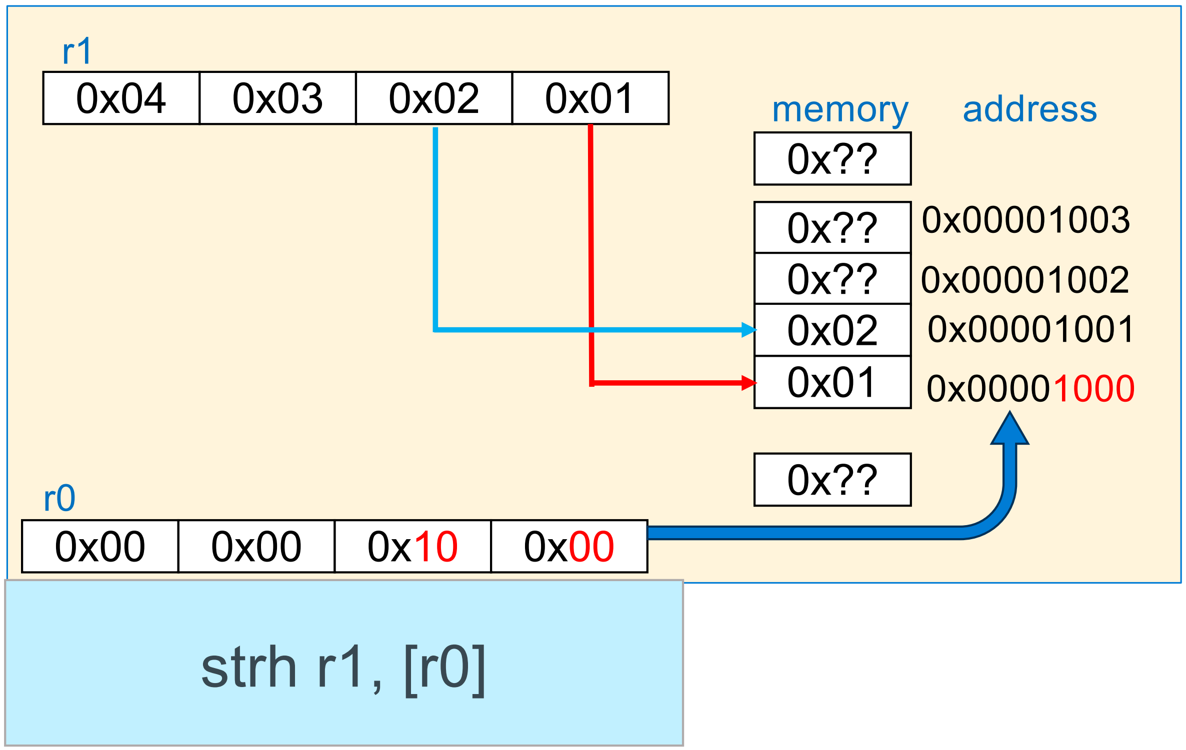
x

Storing 32-bit Registers To Memory, 32-bit



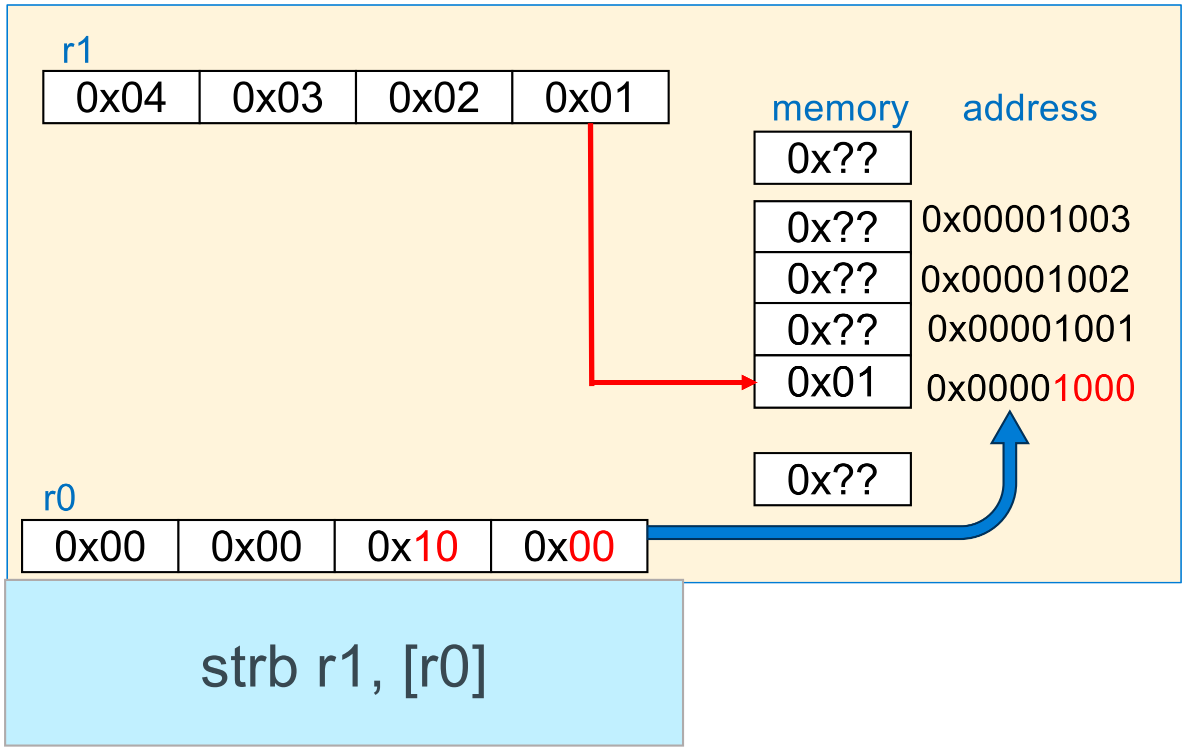
x

Storing 32-bit Registers To Memory, 16-bit



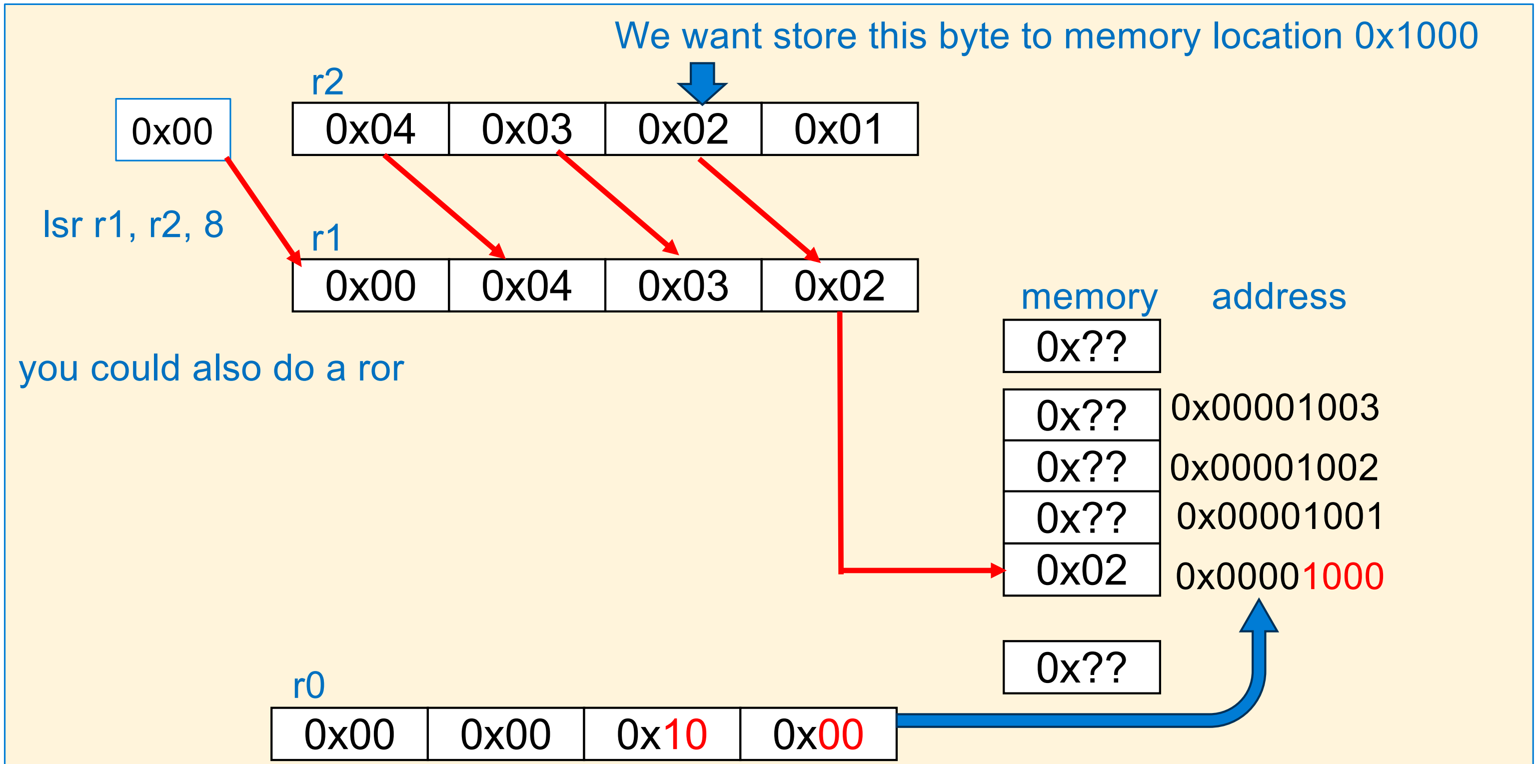
x

Storing 32-bit Registers To Memory, 8-bit



x

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



strb r1, [r0]

x

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;
}
```


X

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             pre loop guard
    ble     .Ldone

    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:
```


x

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           //convert cnt to int size
    mov     r3, 0               // initialize counter
.Ldo:
    ldr     r4, [r0, r3]        // load from src
    str     r4, [r1, r3]        // store to dest
    add     r3, r3, 4           // counter++
    cmp     r3, r2              // count < r3
    blt     .Ldo

.Ldone:
```

pre loop guard

loop guard

one increment covers both arrays

x

Base Register + Register Offset With chars

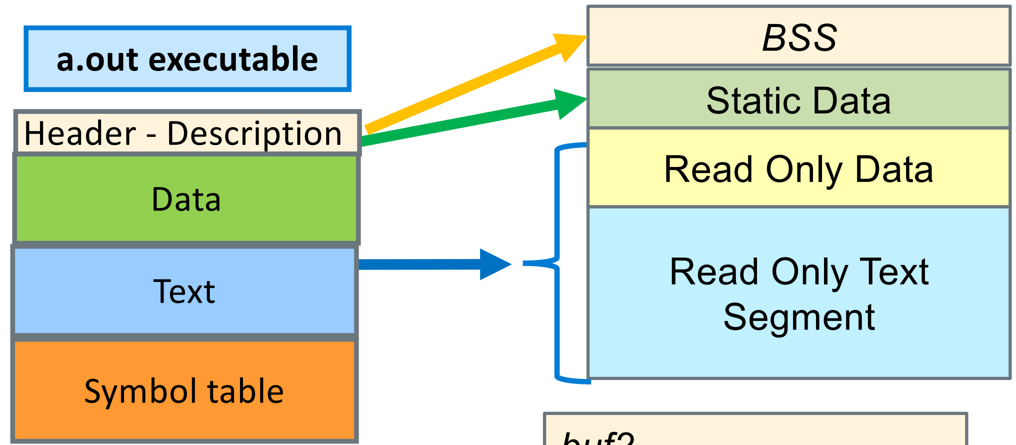
```c
#include <stdio.h>
#include <stdlib.h>
#define SZ 6
void cpy(char *, char *, int);
int main(void)
{
    char src[SZ] =
       {'a', 'b', 'c', 'd', 'e', '\0'};
    char dst[SZ];


    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:
```

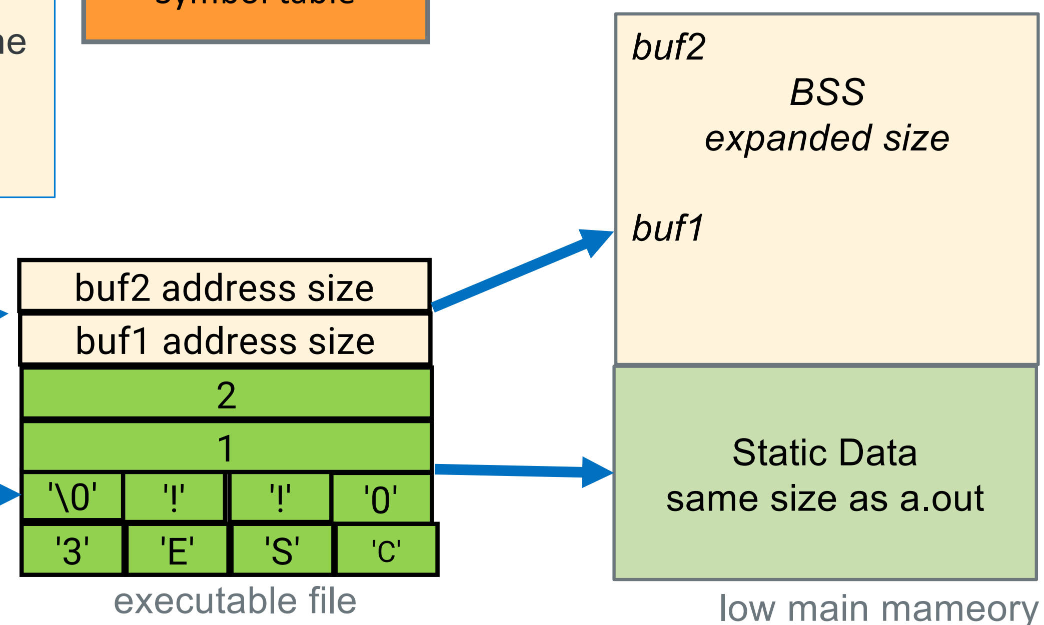

x

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];

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



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`

x

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:)
```


X

Defining **Static** Variables: Why the .align?

```
        .data
chx:    .byte 'A'
cnt:    .word 4
```

See cnt is not word aligned

label	address	contents
<chx>:	12028	00000041
<cnt>:	12029	00000004

Sets alignment for **next** variable

```
        .data
chx:    .byte 'A'
        .align 2
cnt:    .word 4
```

See cnt is word aligned

label	address	contents
<chx>:	12028	00000041
<cnt>:	1202c	00000004

x

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 .Llablel
```

```
.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
```


x

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

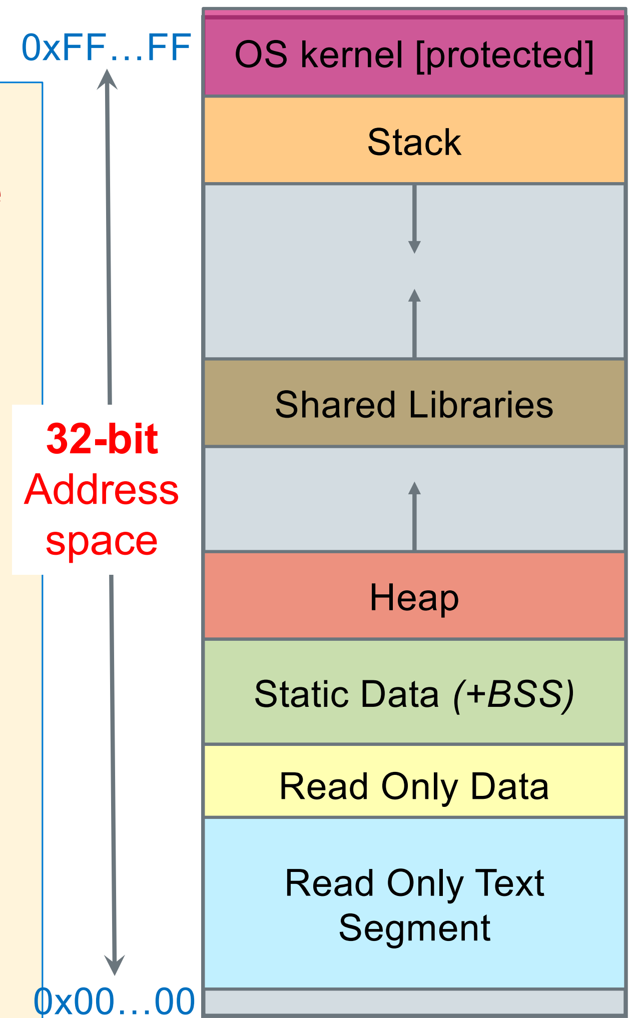
1. load the pointer to the memory
2. write (store) to *pointer

```
        .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

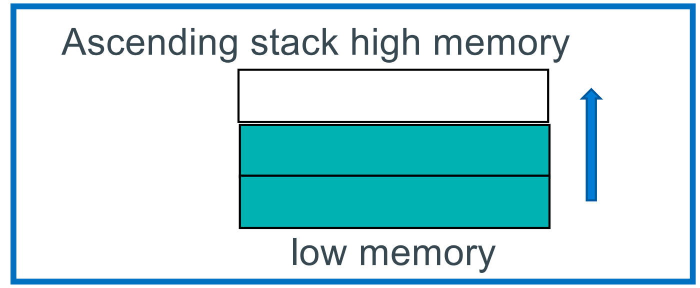
Stack Segment: Support of Functions

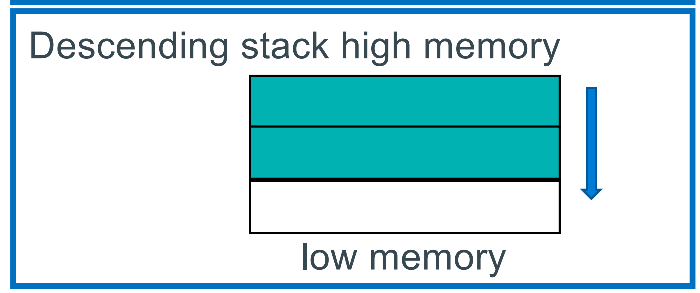
- The stack consists of a series of *"stack frames"* or *"activation frames"*, one is created each time a function is called at runtime
- Each frame represents a function that is currently being executed and has not yet completed (why activation frame)
- A function's stack "frame" goes away when the 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)

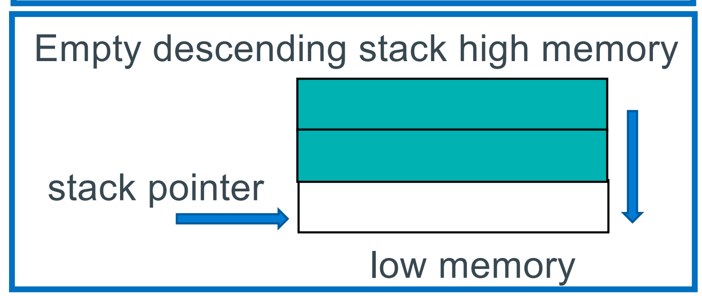


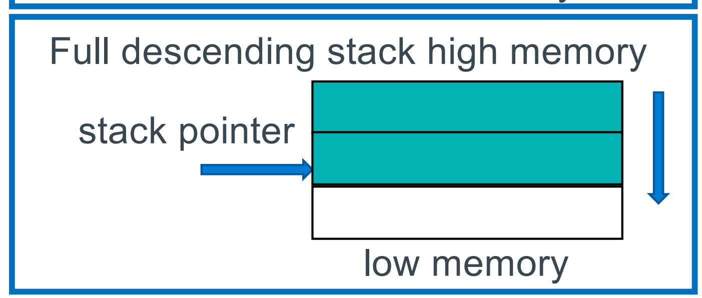
Stack types

- 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**









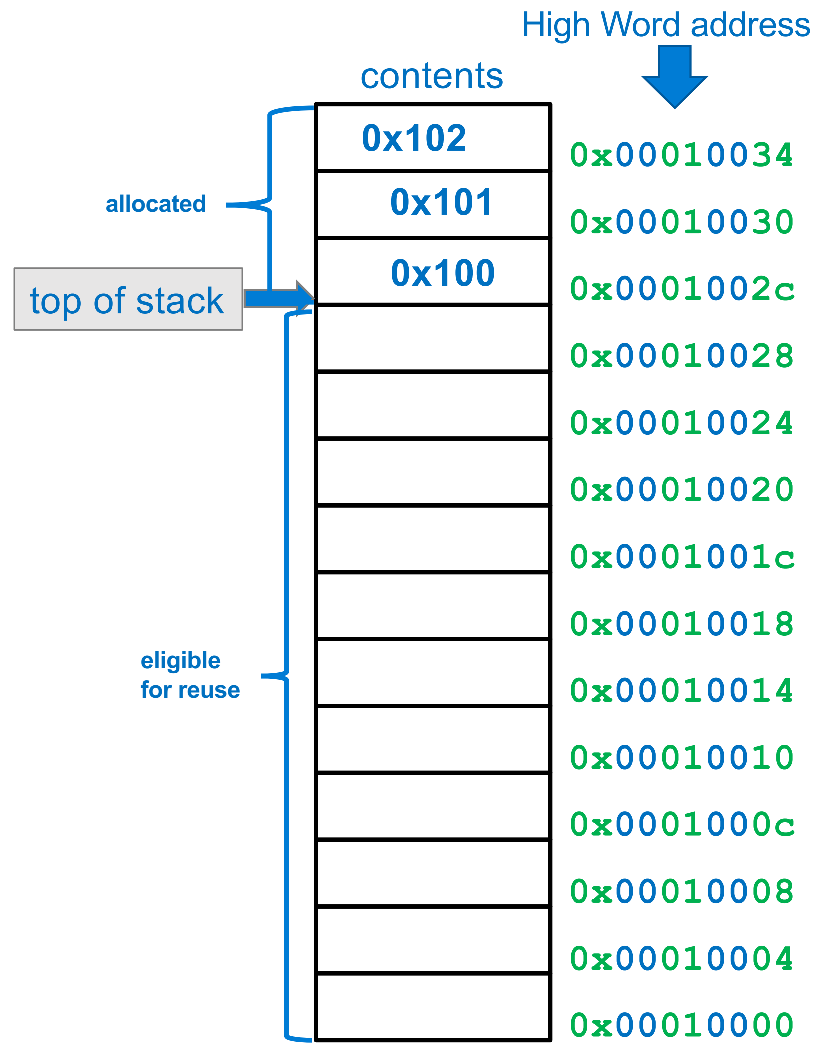
Arm: Stack Operation

- **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***)

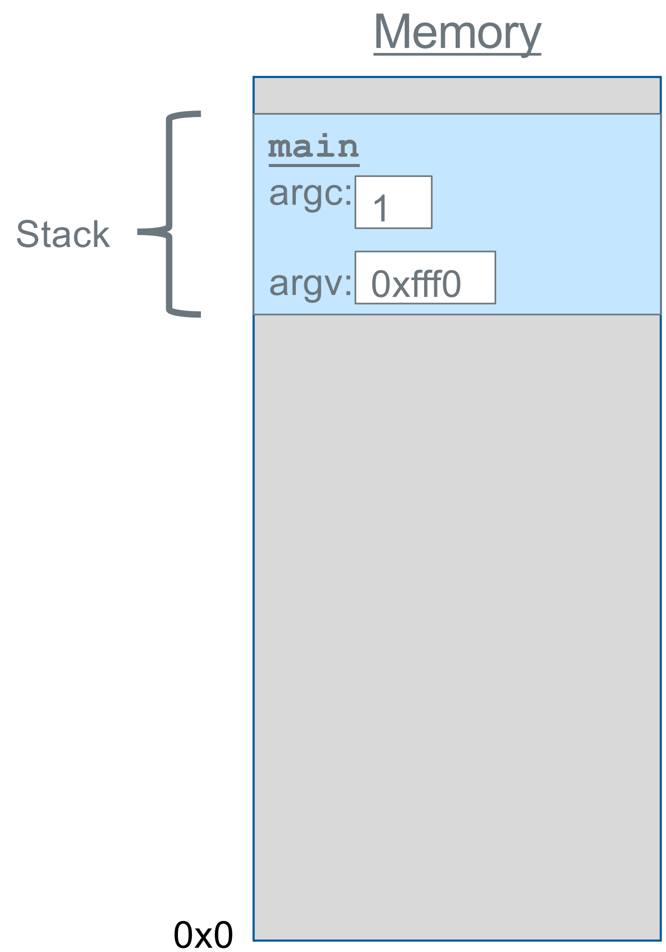
pop (sp + element size)



The Stack

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

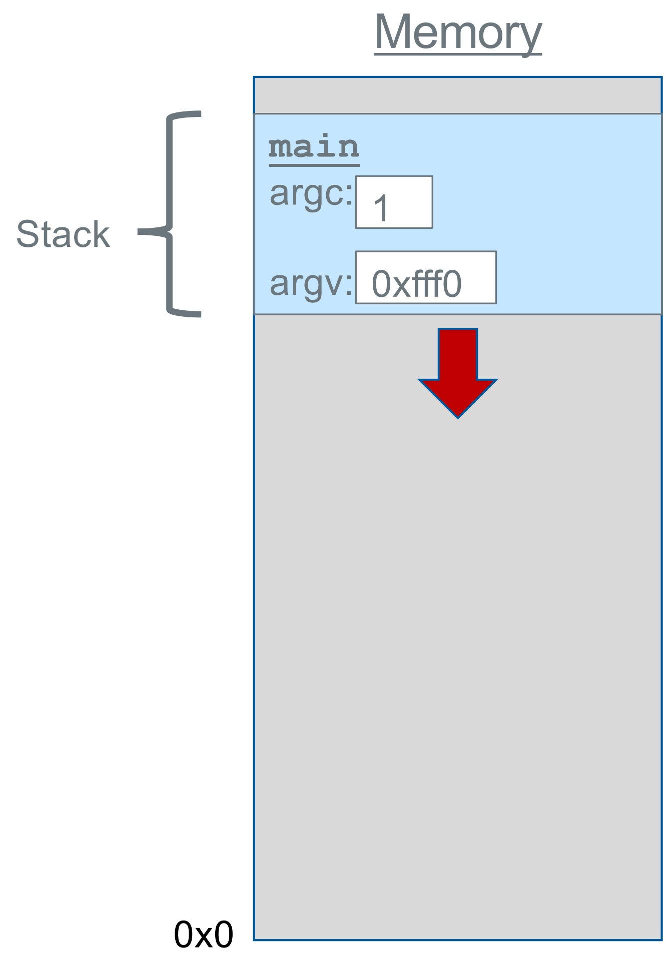
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

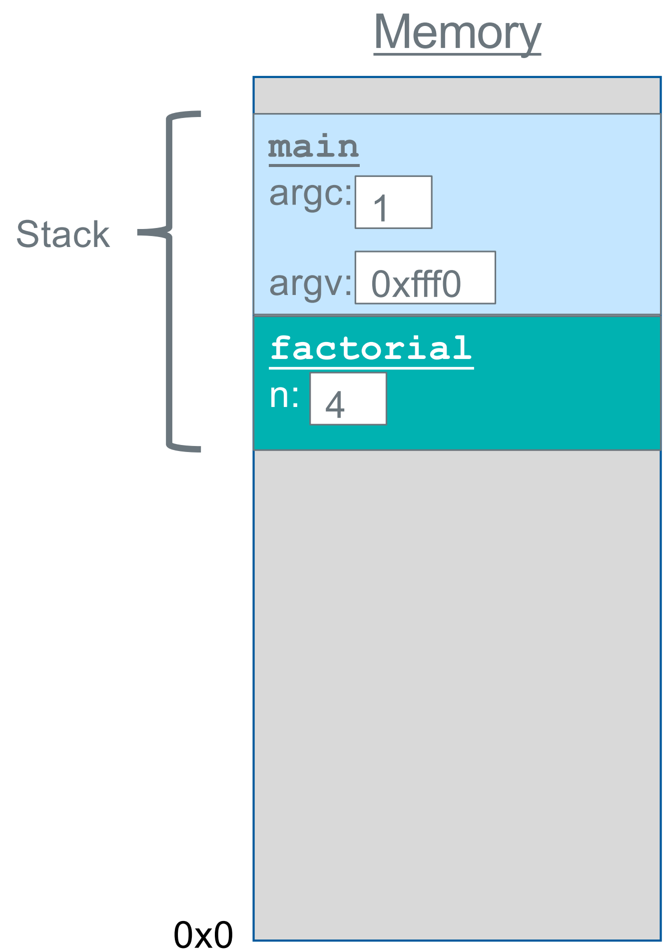
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

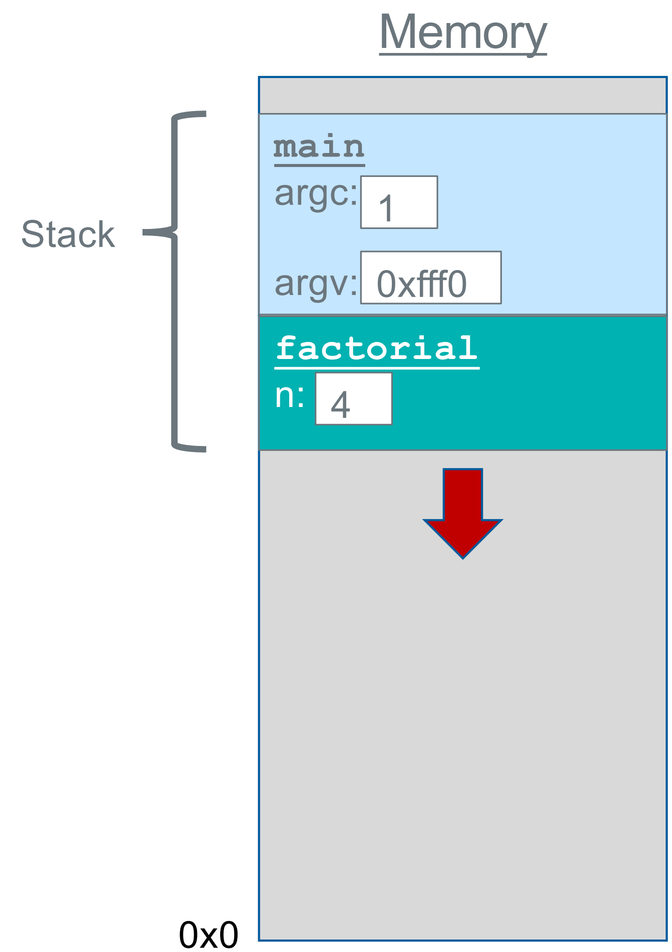
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

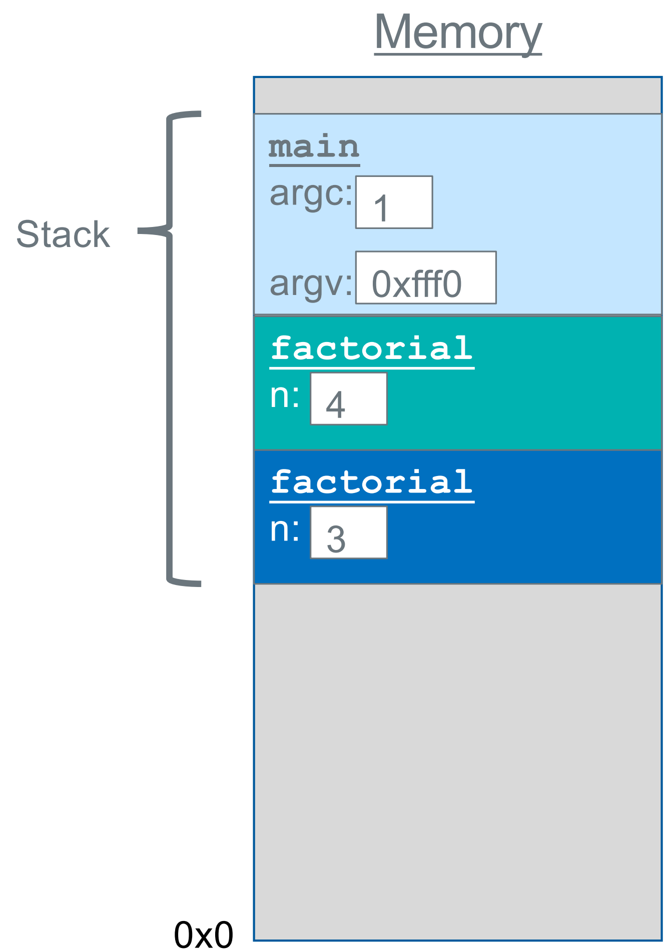
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

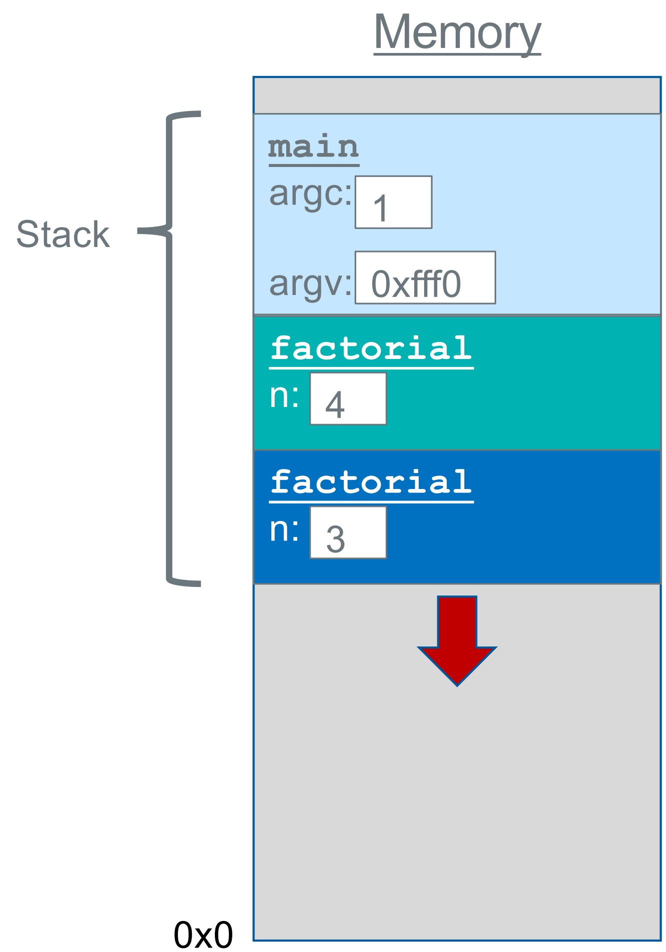
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

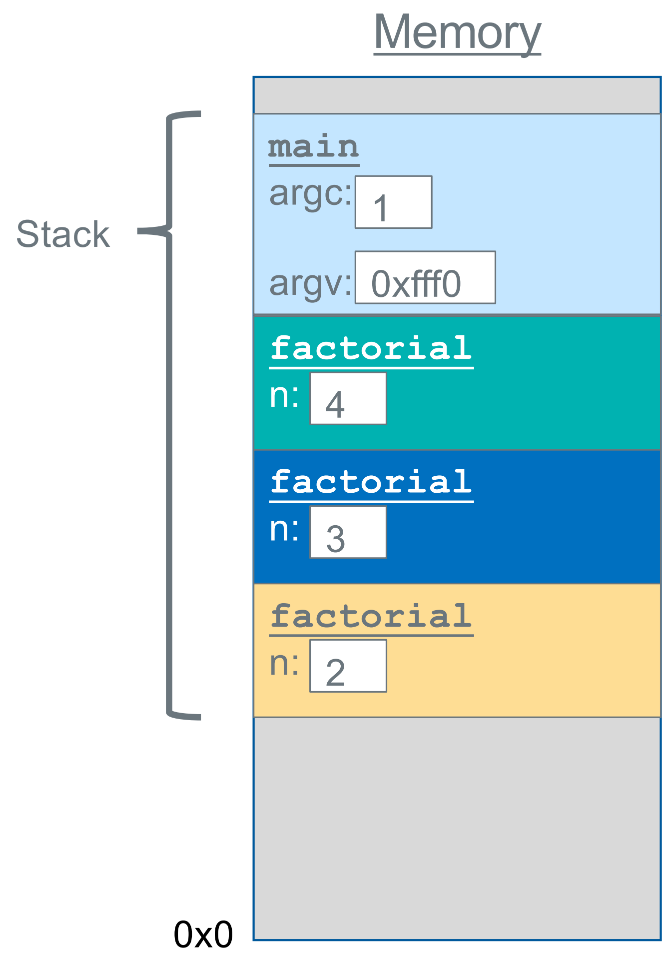
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

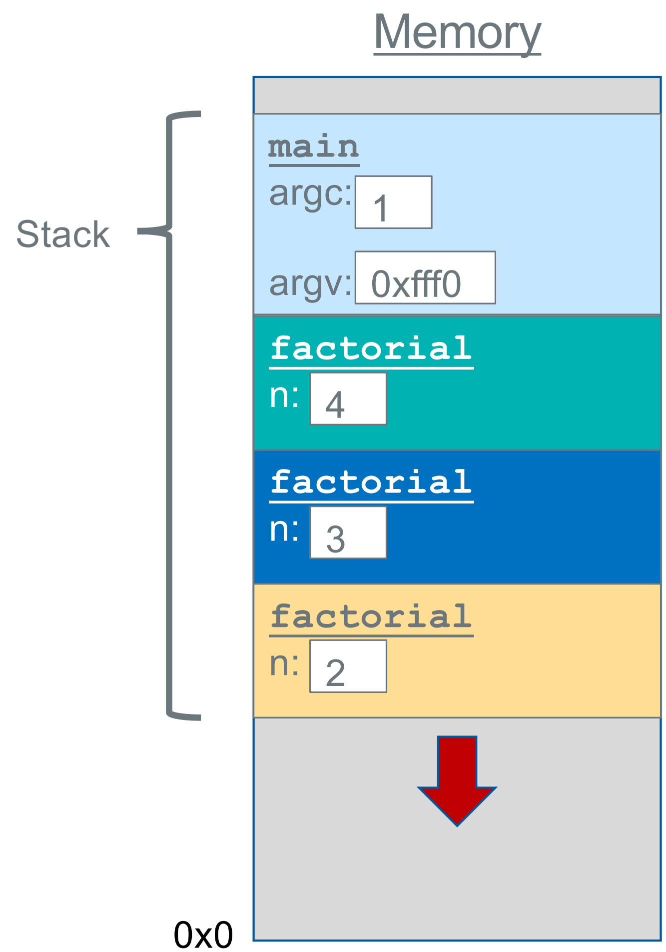
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

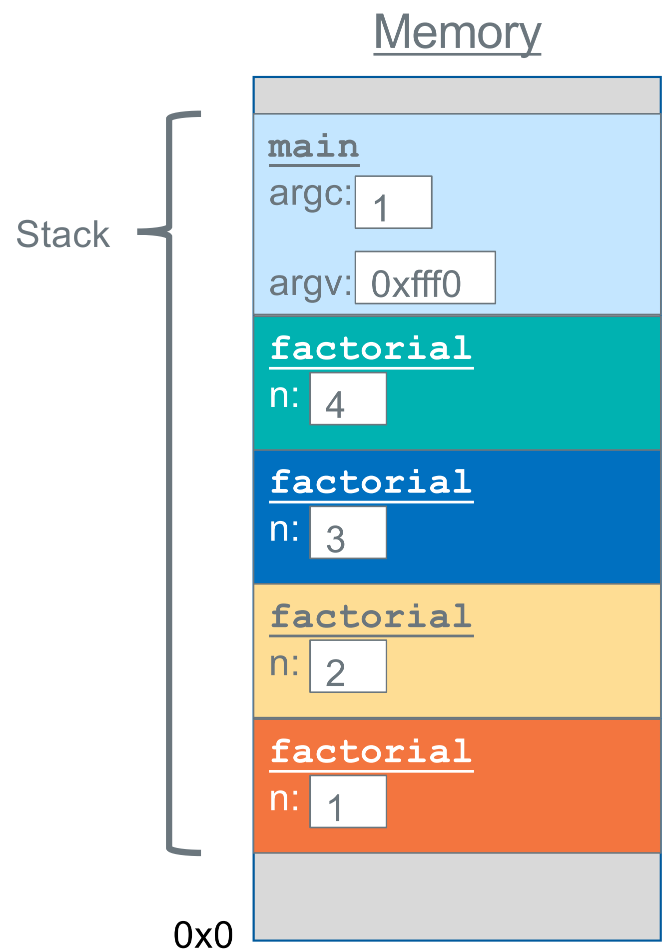
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

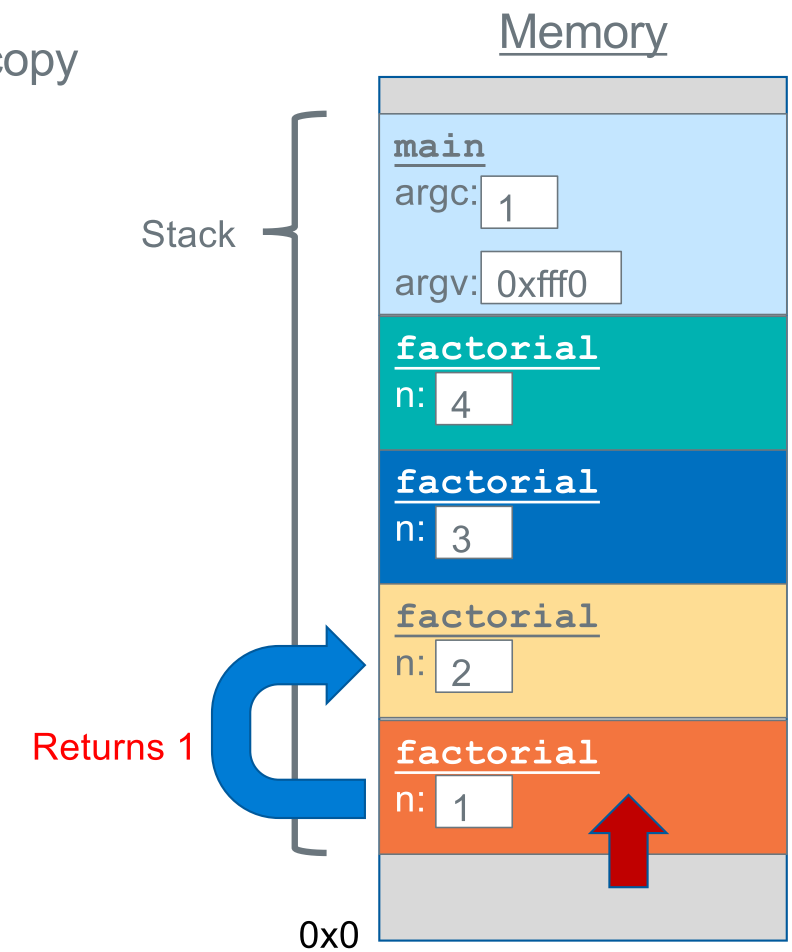
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

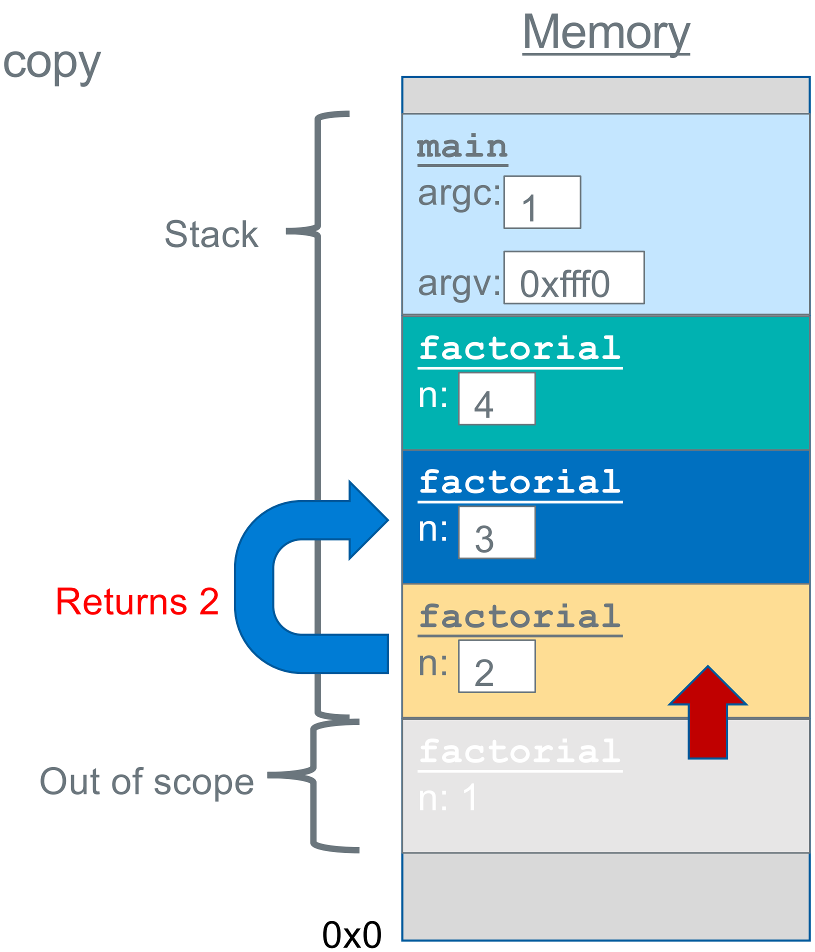
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

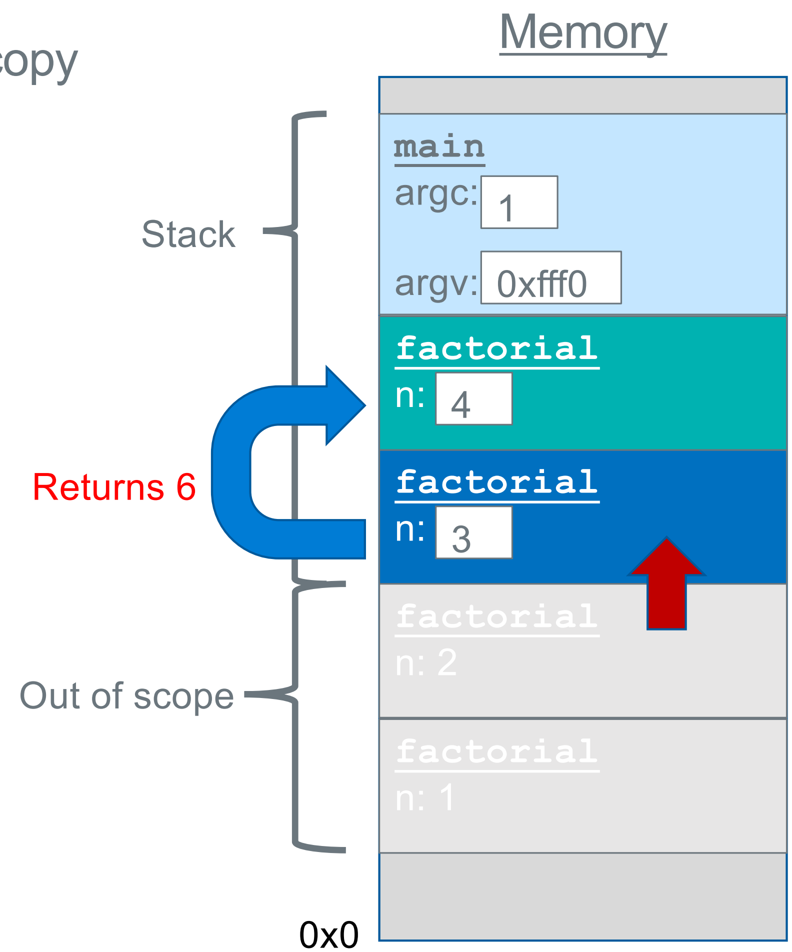
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

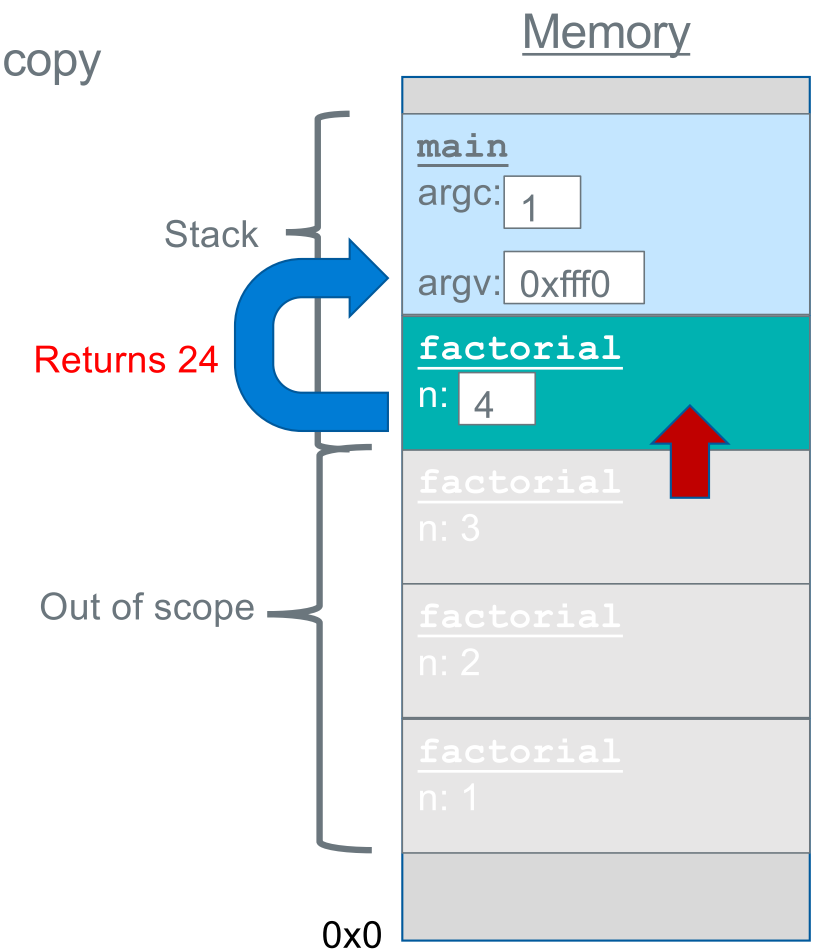
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

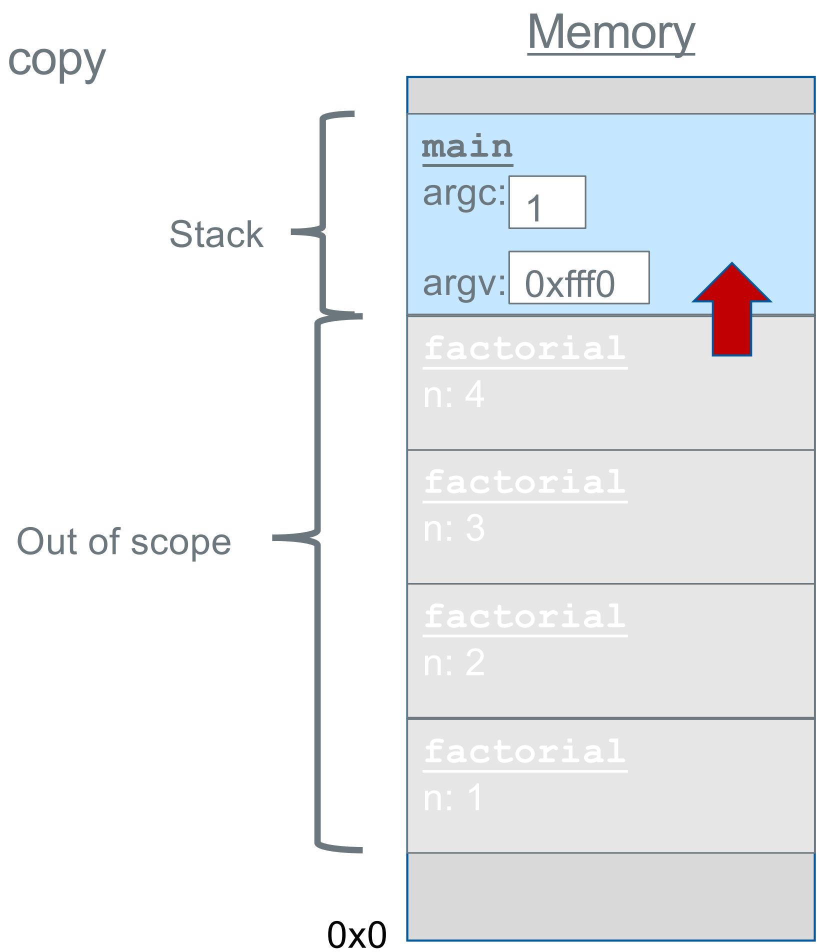
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

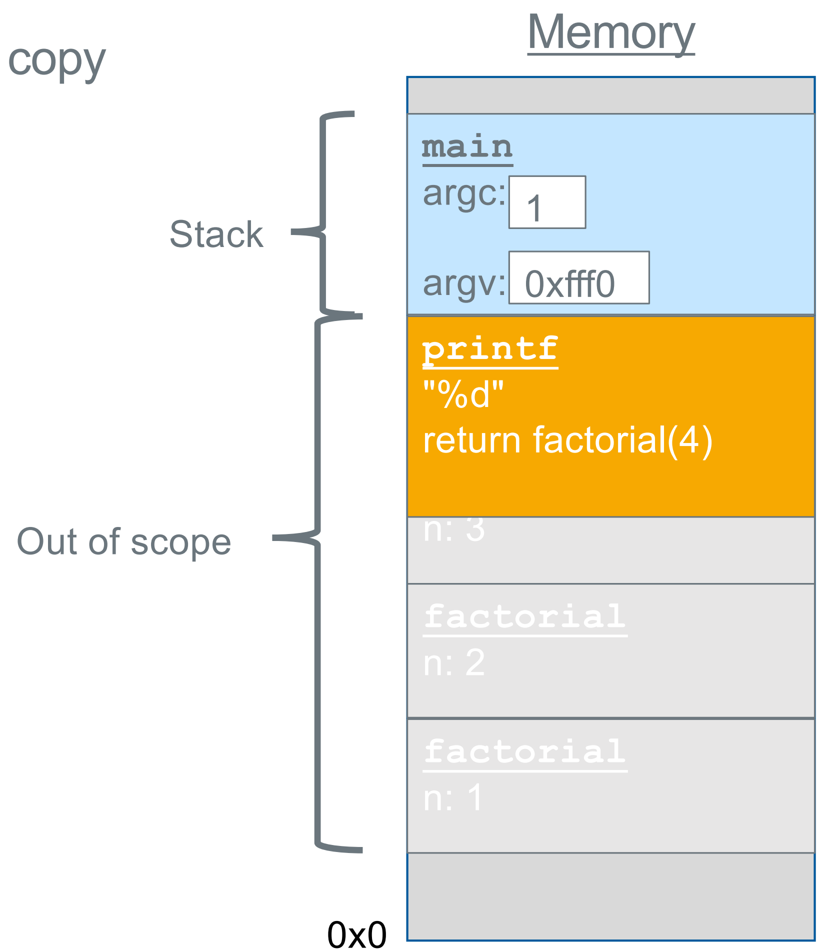
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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:
  •             •
```

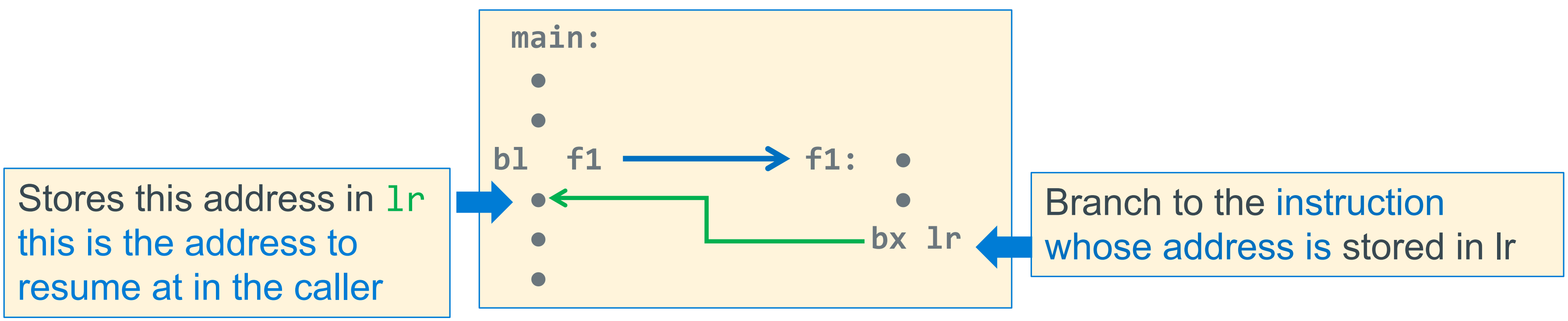

x

Function Call Return

Branch & exchange (function return) instruction

`bx lr` | bx | Rn | `// we will always use lr`

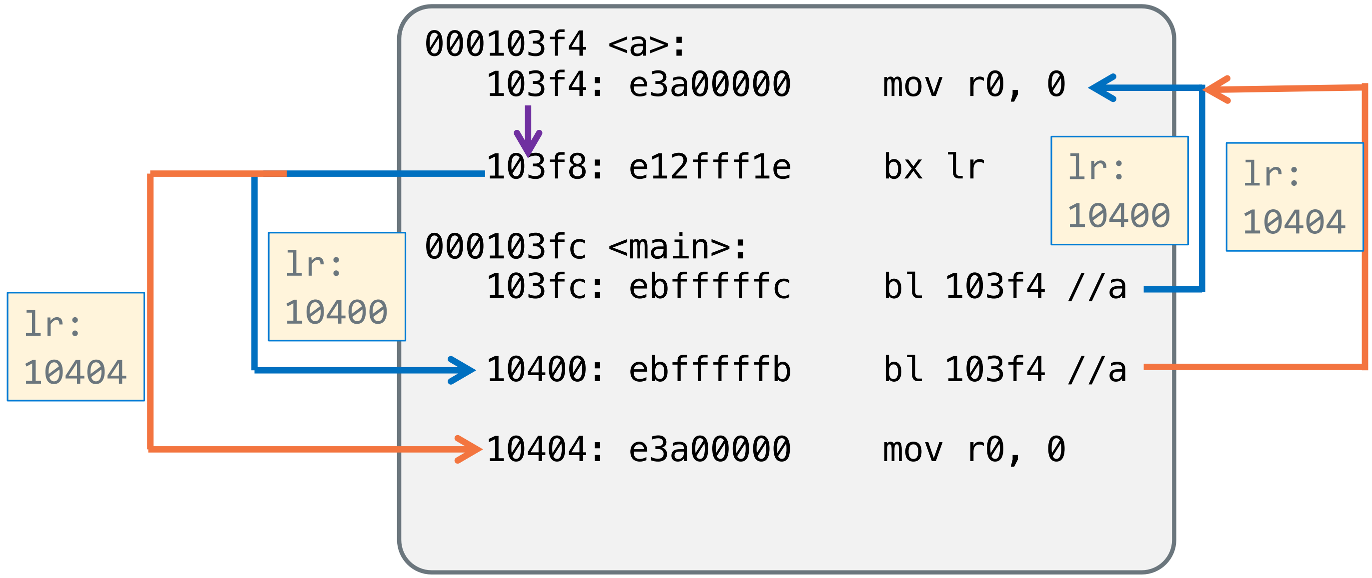
- Causes a branch to the instruction **whose address is stored** in register <**`lr`**>
  - It copies **`lr`** to the PC
- This is often used to implement a return from a function call (exactly like a C return) when the function is called using either **`bl label, or blx Rm`**



x

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

x

Understanding bl and bx - 2

Modifies the link register (lr),
writing over main's return address
Cannot return to main()

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

We need to preserve the lr!

lr: 10400

lr: 10400

Uh No
Infinite loop!!!

```
000103f4 <b>:
   103f4: e3a00000     mov r0, 0
   103f8: e12fff1e     bx lr

000103fc <a>:
   103fc: ebfffffc     bl 103f4 <b>
   10400: e3a00000     mov r0, 0
   10404: e12fff1e     bx lr

00010408 <main>:
   10408: ebfffffb     bl 103fc <a>
   1040c: ebfffffa     bl 103fc <a>
   10410: e3a00000     mov r0, 0
```

lr: 10400

lr: 1040c

x

Understanding bl and blx - 3

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

int (*func)() = a;

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

```
        .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!
```

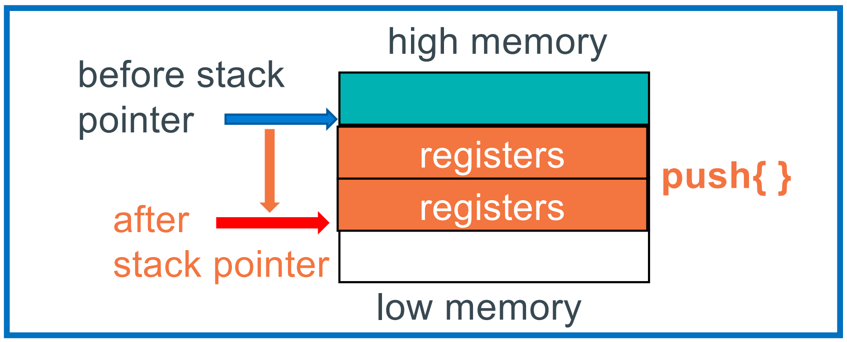
But this has the same infinite loop problem when main() returns!

x

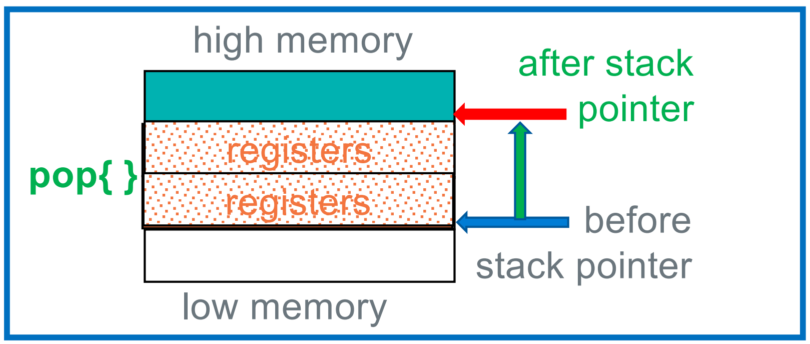
Preserving and Restoring Registers on the stack - 1

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

push (multiple register `str` to memory operation)



push (multiple register `ldr` from memory operation)



x

Preserving and Restoring Registers on the Stack - 2

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`

- **`{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

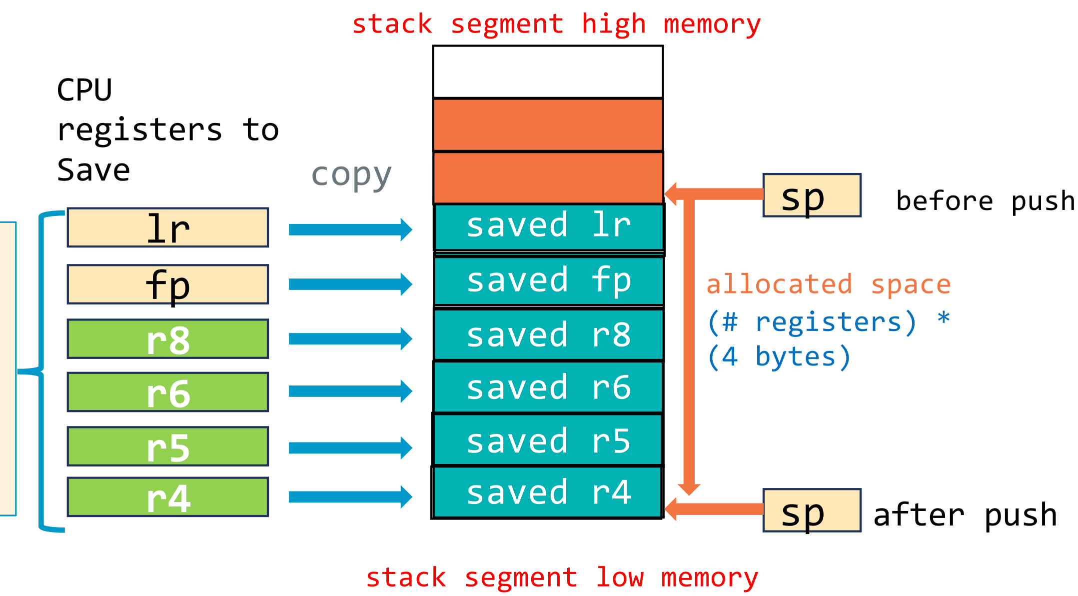
x

push: Multiple Register Save to the 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 slides) then always push an **EVEN** number of registers!



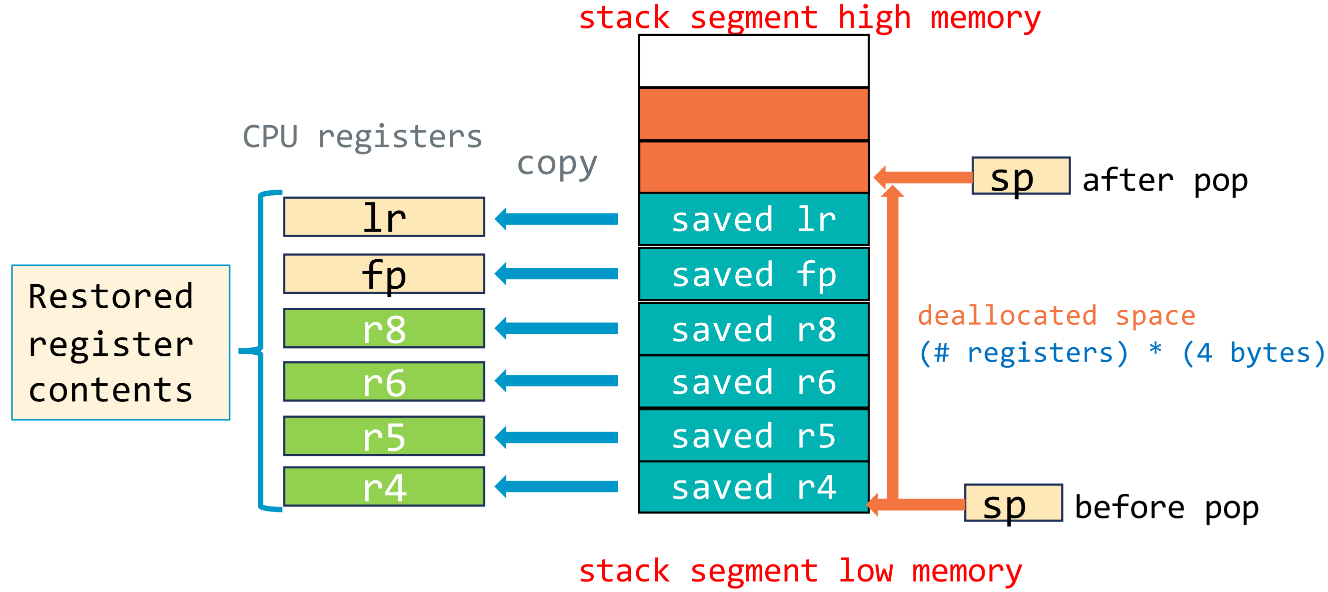
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** subtracts (# of registers saved) * (4 bytes) from the **sp** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

X

pop: Multiple Register Restore from the stack

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

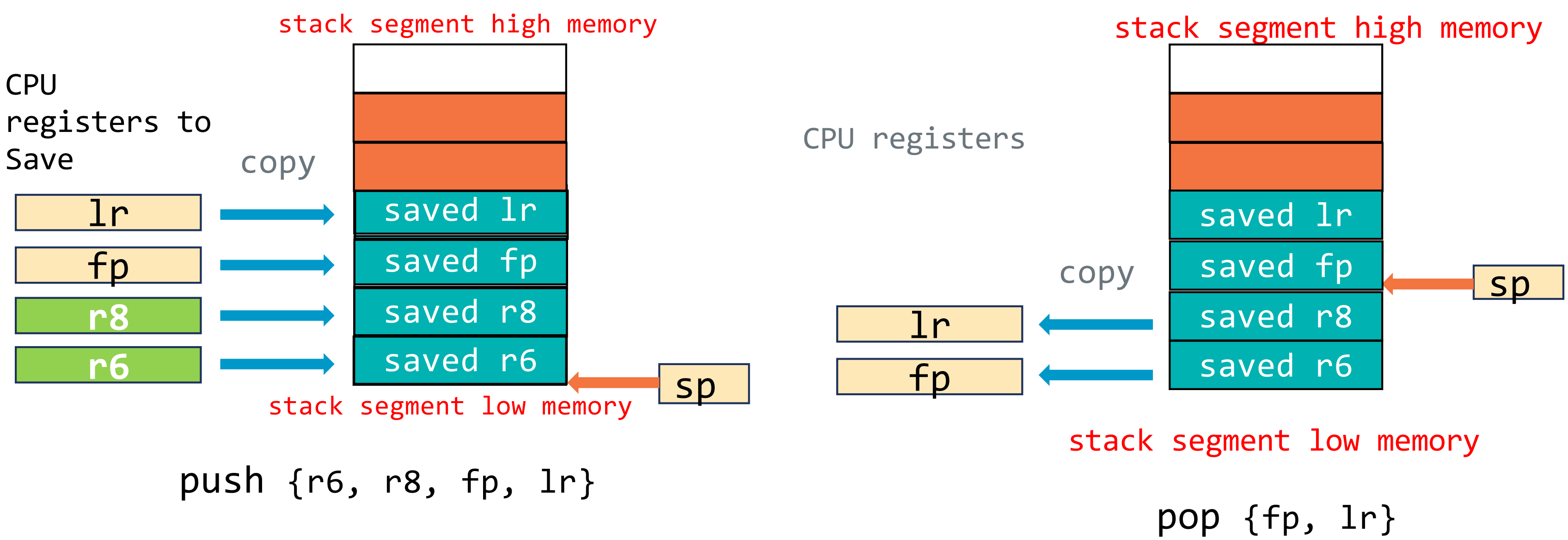
Registers are **pop'd** from the stack *in order* **left (low memory) to right (high memory)**



- **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}` must be the same in both the `push` and the corresponding `pop`

x

Consequences of inconsistent push and pop operands



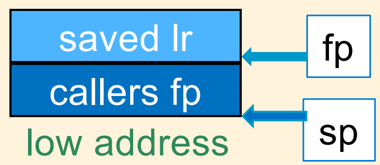
- lr gets an address on the stack, likely segmentation fault

x

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`)

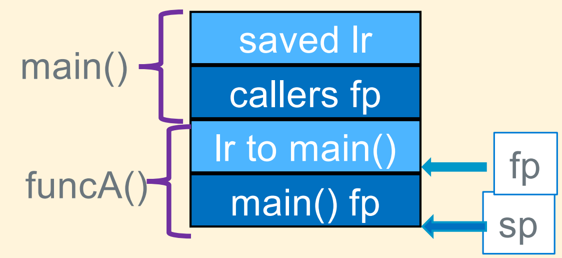
x

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

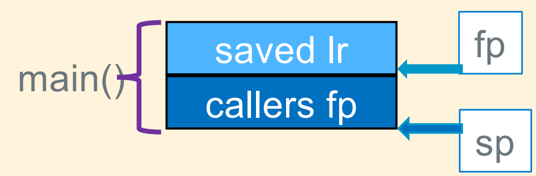
allocate stack space
SP = SP – "space"
grows "down"

main() calls funcA()



- Function return (Function **Epilogue**):
  1. deallocate space for locals
  2. restores preserved registers
  3. removes the frame

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



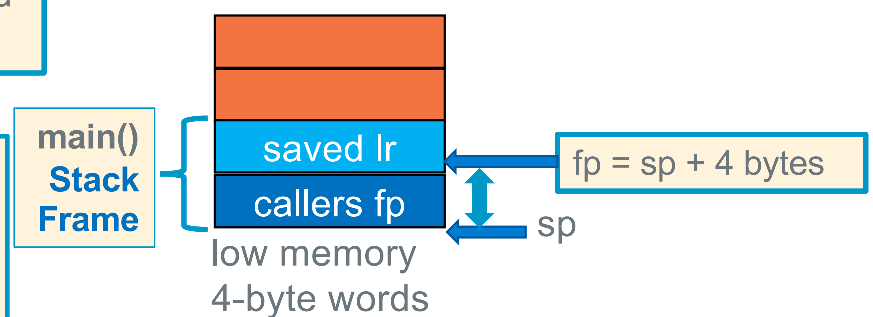
How to set the FP – Minimum Activation Frame

```
    // 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

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

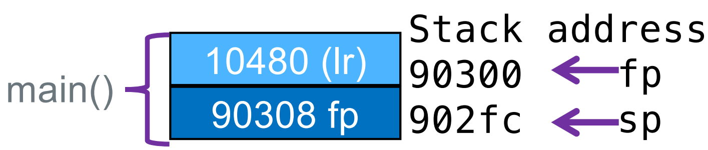



```
IMPORTANT: FP_OFF has two uses:
1.  Where to set fp after prologue push (remember sp position)
2.  Restore sp (deallocate locals) right before epilogue pop
```


x

Using Minimal Stack Frames

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

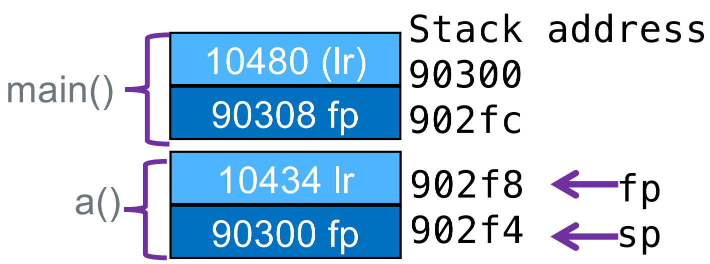



```
000103f4 <b>:
   103f4: e92d4800     push {fp, lr}
   103f8: e28db004     add fp, sp, 4
   103fc: e3a00000     mov r0, 0
   10400: e24bd004     sub sp, fp, 4
   10404: e8bd4800     pop {fp, lr}
   10408: e12fff1e     bx lr


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
```


X

Using Minimal Stack Frames

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



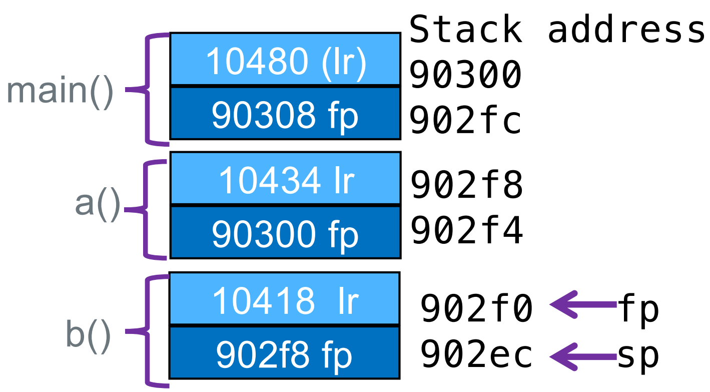

```
000103f4 <b>:
   103f4: e92d4800      push {fp, lr}
   103f8: e28db004      add fp, sp, 4
   103fc: e3a00000      mov r0, 0
   10400: e24bd004      sub sp, fp, 4
   10404: e8bd4800      pop {fp, lr}
   10408: e12fff1e      bx lr


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

x

Using Minimal Stack Frames

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

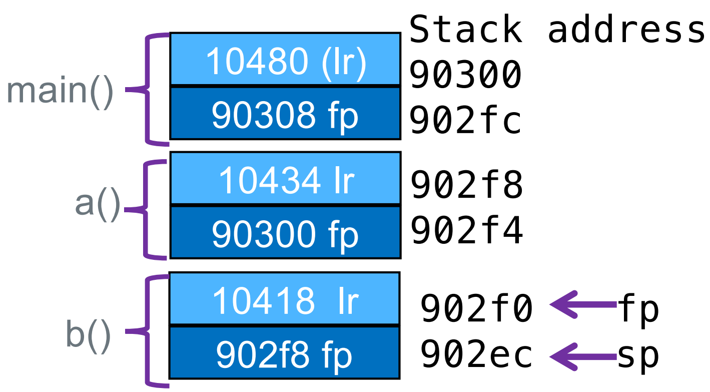



```
000103f4 <b>:
   103f4: e92d4800     push {fp, lr}
   103f8: e28db004     add fp, sp, 4
   103fc: e3a00000     mov r0, 0
   10400: e24bd004     sub sp, fp, 4
   10404: e8bd4800     pop {fp, lr}
   10408: e12fff1e     bx lr
                                   lr:
                                  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


x

Using Minimal Stack Frames

```c
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

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: 10418

lr: 10434

X

Using Minimal Stack Frames

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

		Stack address	
main()	10480 (lr)	90300	
	90308 fp	902fc	
a()	10434 lr	902f8	← fp
	90300 fp	902f4	← sp

lr: 10418

lr: 10418

lr: 10434

```
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();
```

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!

```
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: 10418

lr: 10418

lr: 10418

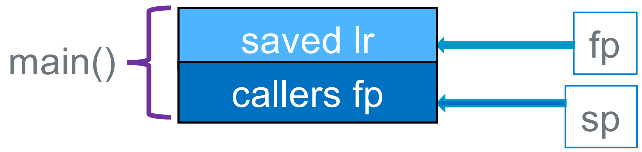
lr: 10434

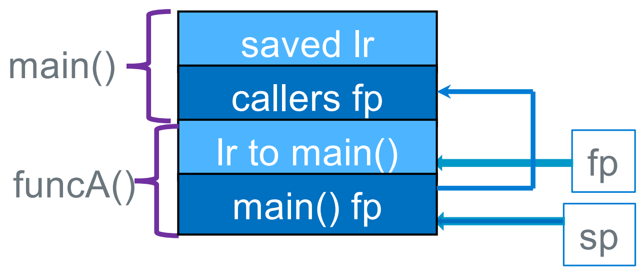
lr: 10434

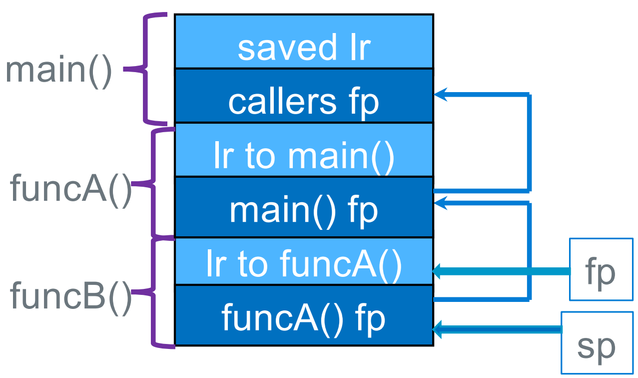
lr: 10434

x

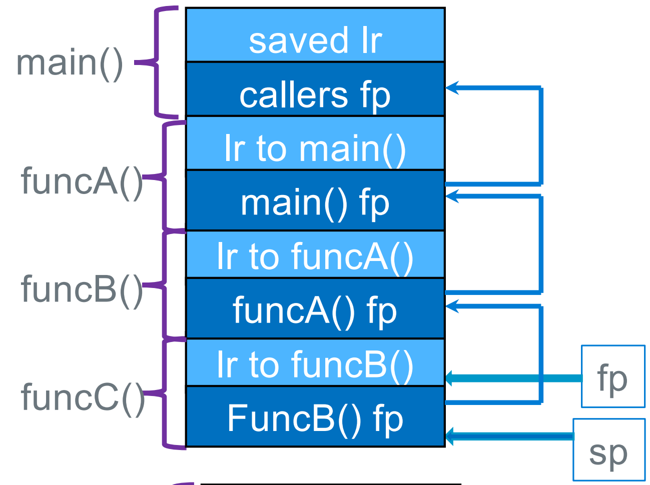
By following the saved fp, you can find each stack frame

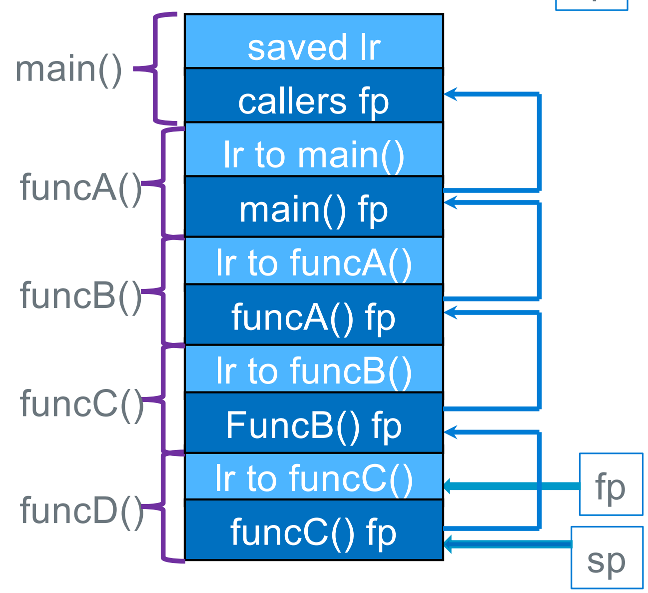






How gdb finds stack frames





x

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)

x

Register Arguments and Return Values

- 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
   a. zero filled for unsigned values
   b. sign extended for signed values

Single Byte (unsigned char)

	31			0
r0	0x00	0x00	0x00	0xe1

$\underbrace{\texttt{0x00 0x00 0x00}}_{\text{observe the zero fill}}$

Single Halfword (unsigned short)

	31			0
r0	0x00	0x00	0xe3	0xe1

$\underbrace{\texttt{0x00 0x00}}_{\text{observe the zero fill}}$

Full Word (int or pointer)

	31			0
r0	0x87	0x65	0xe3	0xe1

x

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

x