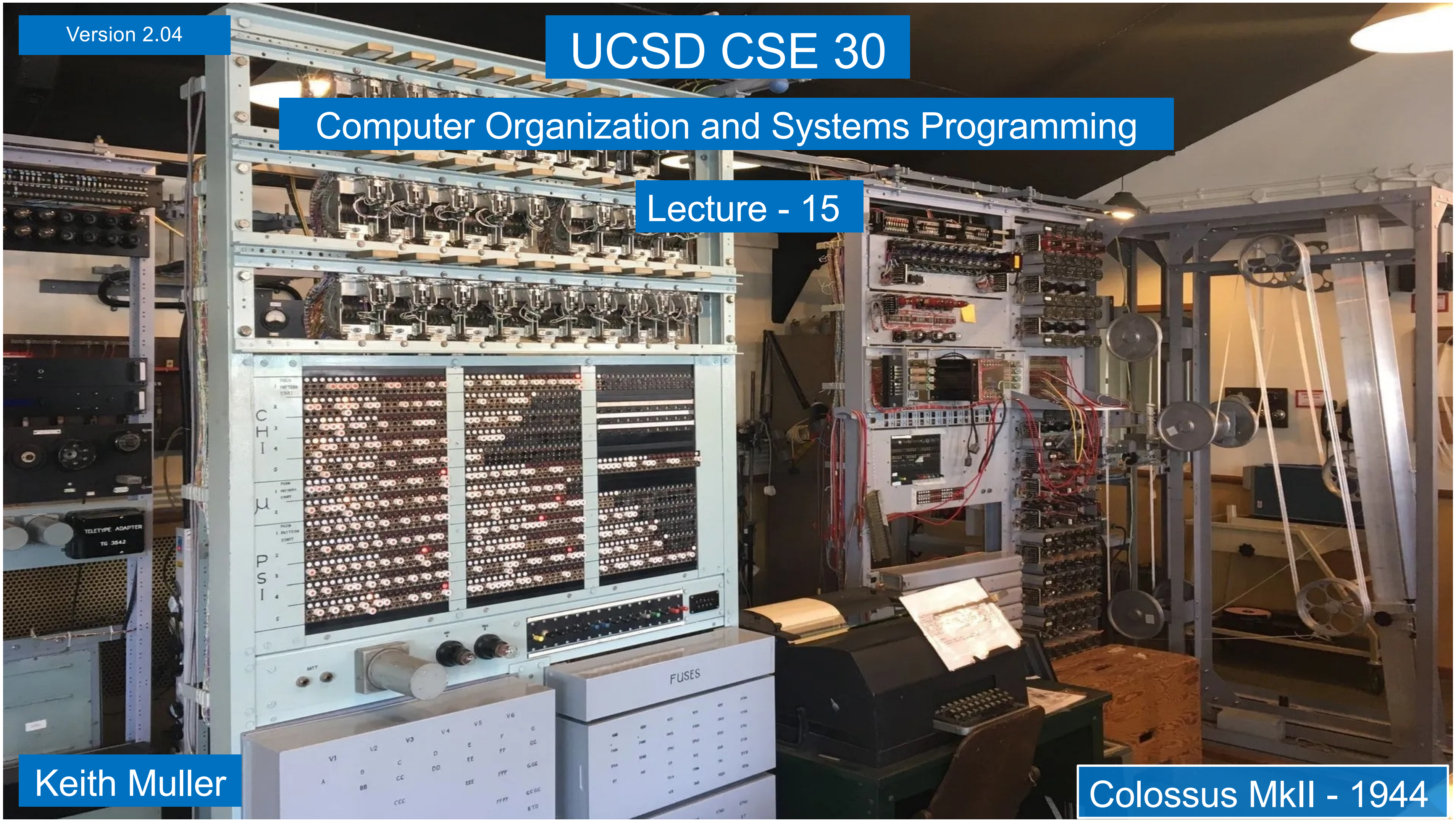

Version 2.04
UCSD CSE 30
Computer Organization and Systems Programming
Lecture - 15
Keith Muller
Colossus MkII - 1944

Masking Summary - 1

Select a field: Use `and` with a mask of one's surrounded by zeros to select the bits that have a 1 in the mask, all other bits will be set to zero

selects this field when used with and

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 1 1 1 1 0 0

selection mask 0x3c

Clear a field: Use `and` with a mask of zero's surrounded by ones to select the bits that have a 1 in the mask, all other bits will be set to zero

clears this field when used with and

1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 0 0 0 0 1 1

clear a field mask 0xffffffc3

Isolate a field: Use `lsr, lsl` to get a field surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

lsl this edge to bit 31 (left edge) then lsr to move back

lsr this edge to bit 0 (right edge) then lsl to move back

x

Masking Summary - 2

Isolate a field: Use `and` to get a field surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

`selection mask 0x00ff0000`

rotate a field: Use `ror` to move a field without changing other bits

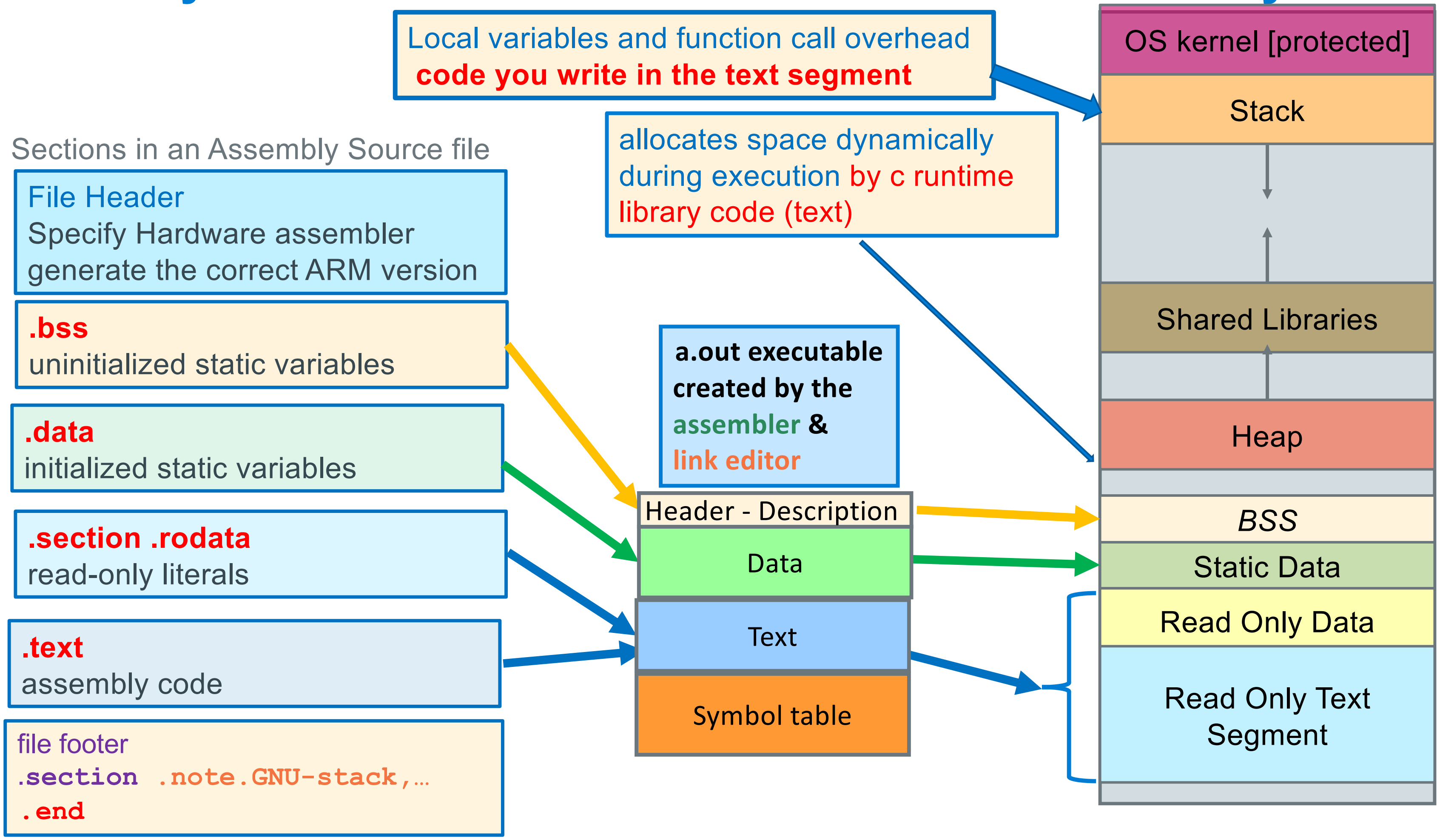
Keep these bits	source	Keep these bits

Keep these bits	Keep these bits	source

Insert a field: Use `orr` with fields surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Keep these bits	0 0 0 0 0 0 0 0	Keep these bits

Assembly Source File to Executable to Linux Memory



x

Assembly Source File Template

```
// File Header
        .arch armv6                 // armv6 architecture instructions
        .arm                        // arm 32-bit instruction set
        .fpu vfp                    // floating point co-processor
        .syntax unified             // modern syntax

// BSS Segment (only when you have initialized globals)
         .bss
// Data Segment (only when you have uninitialized globals)
         .data
// Read-Only Data (only when you have literals)
         .section .rodata
// Text Segment – your code
         .text

// Function Header
        .type   main, %function     // define main to be a function
        .global main                // export function name
main:
// function prologue                // stack frame setup
                  // your code for this function here
// function epilogue                //stack frame teardown

// function footer
         .size  main, (. – main)

// File Footer
          .section .note.GNU-stack,"",%progbits // stack/data non-exec
.end
```

- assembly programs end in .S
  - That is a **capital** .S
  - example: test.S
- Always use gcc to assemble
  - _start() and C runtime
- File has a complete program
  `gcc file.S`
- File has a partial program
  `gcc –c file.S`
- Link files together
  `gcc file.o cprog.o`

X

Assembler Directives: .equ and .equiv

```
.equ    BLKSZ,  10240      // buffer size in bytes
.equ    BUFCNT, 100*4      // buffer for 100 ints
.equ    BLKSZ,  STRSZ * 4  // redefine BLKSZ from here
```

```
.equ <symbol>, <expression>
```

- Defines and sets the value of a symbol to the evaluation of the expression
- Used for specifying constants, like a `#define` in C
- You can (re)set a symbol many times in the file, last one seen applies

```
.equ    BLKSZ,  10240      // buffer size in bytes
// other lines
.equ    BLKSZ,  1024       // buffer size in bytes
```


x

Function Template

```
        .text                          // start of the text segment

        .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:
        push    {fp, lr}               // push (save) fp and lr on stack
        add     fp, sp, FP_OFF         // set fp for this function

        // your code

        sub     sp, fp, FP_OFF
        pop     {fp, lr}               // pop (retore) fp and lr from stack
        bx      lr                     // return to caller

        .size myfunc, (. - myfunc)
```

myfunc label is the address of the first instruction in myfunc (the push below)

Function header

Function Prologue creates stack frame

Function Epilogue removes stack frame

Function footer

x

Preview: 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**
  - **In terms of C runtime support, these registers contain the copies given to the called function**
  - **C allows the copies to be changed in any way by the called function**

X

Preview: Writing an ARM32 function

```
#include <stdlib.h>
#include <stdio.h>
#include "sum4.h"
int main()
{
    int reslt;

    reslt = sum4(1,2,3,4);

    printf("%d\n", reslt);
    return EXIT_SUCCESS;
}
```

```
#ifndef SUM4_H
#define SUM4_H

#ifndef __ASSEMBLER__
int sum4(int, int, int, int);
#else
.extern sum4
#endif

#endif
```

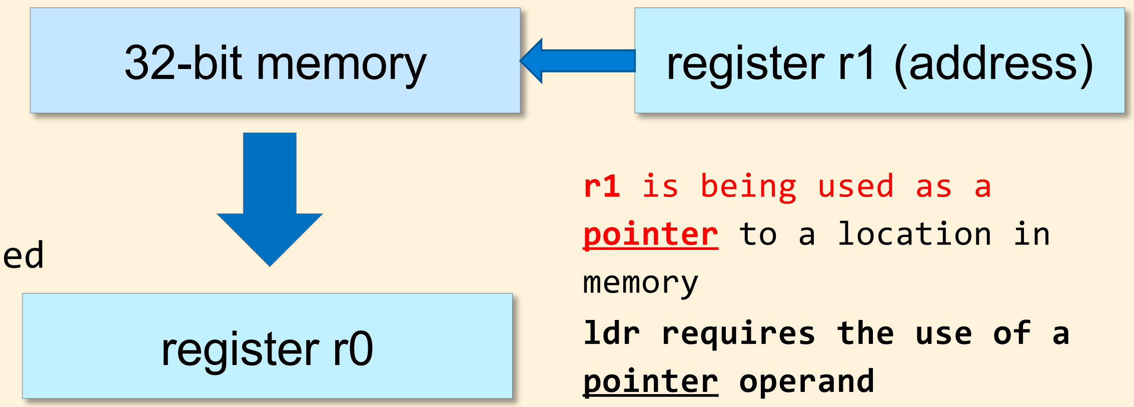
```
#include "sum4.h"
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .global sum4
    .type   sum4, %function
    .equ    FP_OFF, 28
    // r0 = sum4(r0, r1, r2, r3)
sum4:
    push    {r4-r9, fp, lr}
    add     fp, sp, FP_OFF

    add     r0, r0, r1
    add     r0, r0, r2
    add     r0, r0, r3

    sub     sp, fp, FP_OFF
    pop     {r4-r9, fp, lr}
    bx      lr
    .size sum4, (. - sum4)
    .section .note.GNU-stack,"",%progbits
.end
```

```
$ gcc -Wall -Wextra -c main.c
$ gcc -c sum4.S
$ gcc sum4.o main.o
$ ./a.out
10
```

Load/Store: Register Base Addressing

`ldr r0, [r1]`

Copies a 32-bit word from the memory location whose address is contained in r1 (r1 is a pointer) into register r0

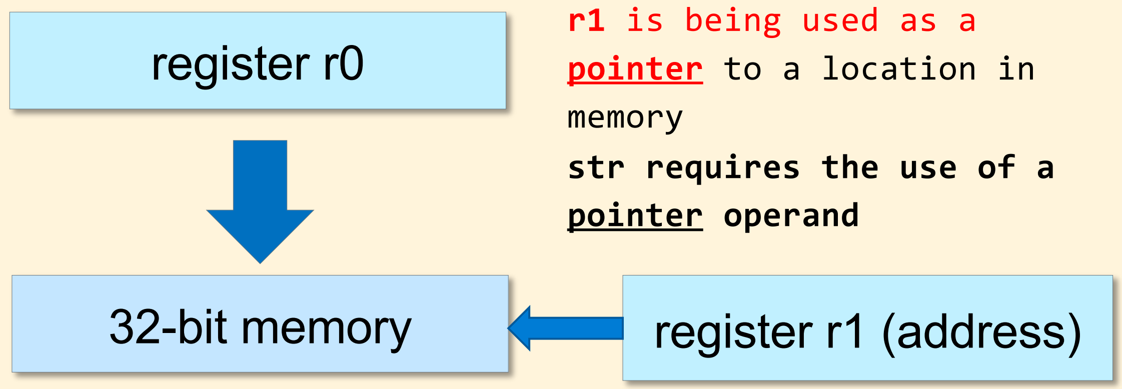


r1 is being used as a **pointer** to a location in memory

ldr requires the use of a pointer operand

`str   r0, [r1]`

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)

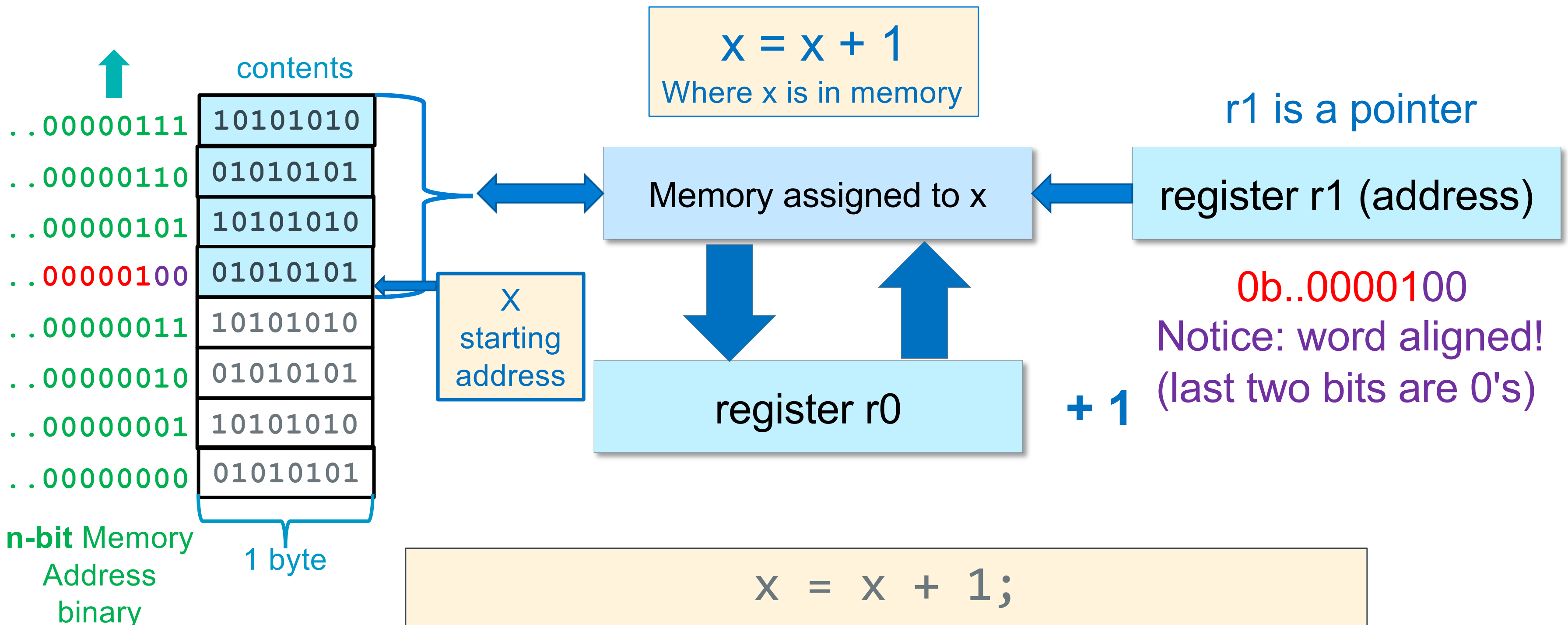


r1 is being used as a **pointer** to a location in memory

str requires the use of a pointer operand

x

Example Base Register Addressing Load – Modify – Store



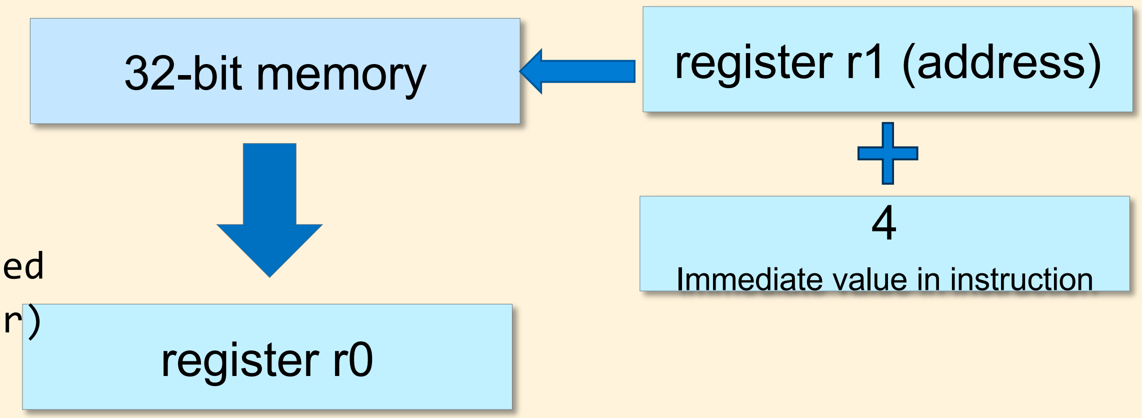

```
x = x + 1;
ldr r0, [r1]          // r0 = *r1 (read x)
add r0, r0, 1         // r0 = r0 + 1 (x++)
str r0, [r1]          // *r1 = r0 write x
```


x

Load/Store: Register Base Addressing + Immediate

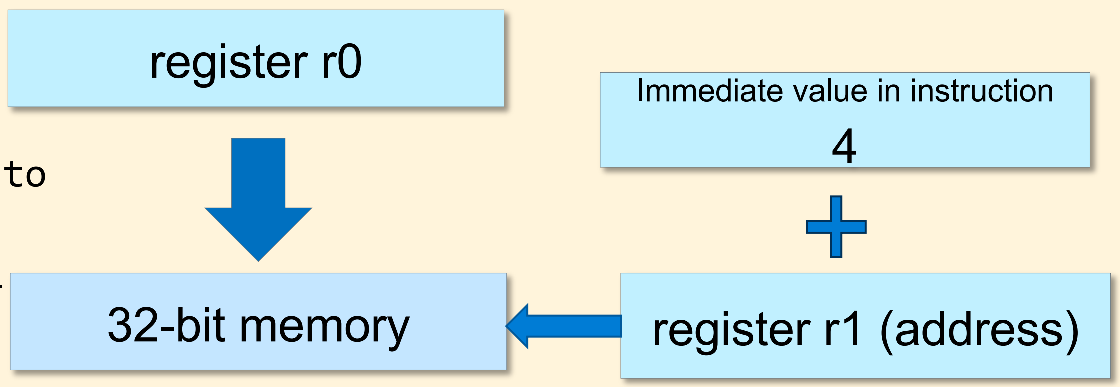
```
ldr r0, [r1, 4]
```

Copies a 32-bit word from the memory location whose address is contained in r1 +4 (r1 is a pointer) into register r0



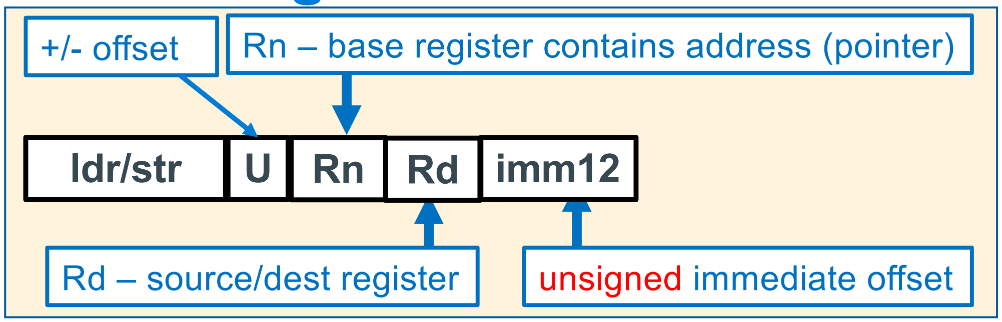

```
str   r0, [r1, 4]
```

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1+4 (r1 pointer)



x

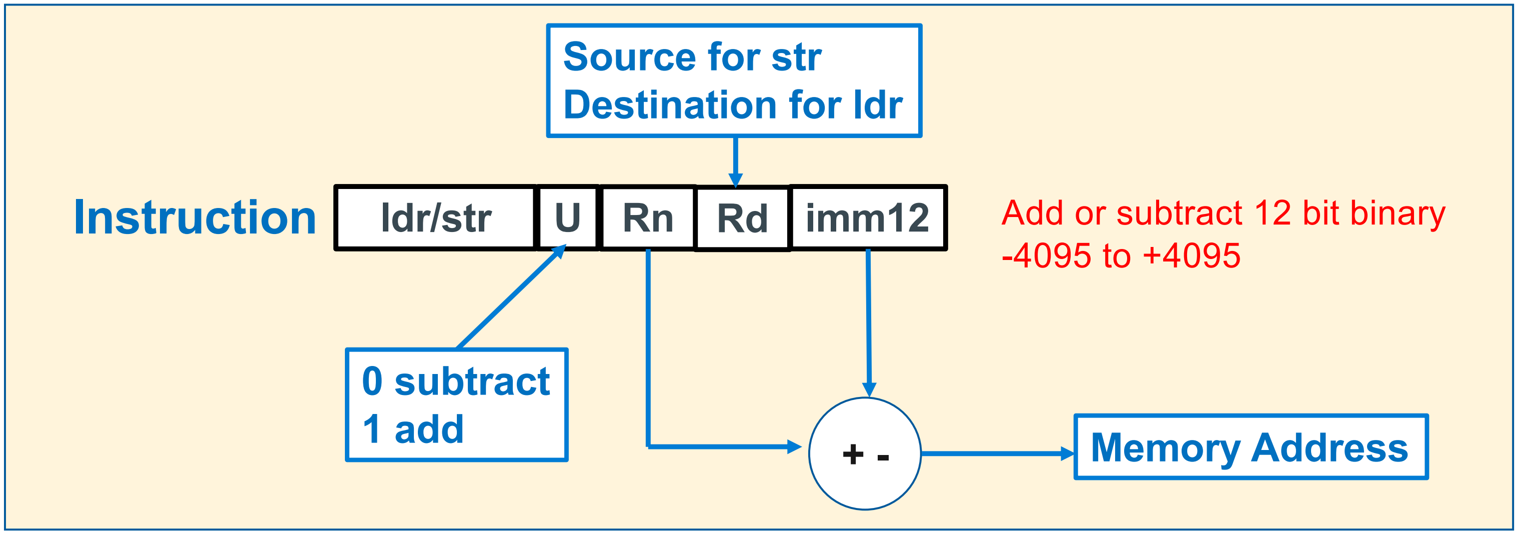
LDR/STR – Base Register + Immediate Offset Addressing



- **Register Base Addressing**:
  - Pointer Address: Rn; source/destination data: Rd
  - **Unsigned pointer address** in stored in the base register
- **Register Base + immediate offset Addressing:**
  - Pointer Address = register content + immediate offset `-4095 <= imm12 <= 4095 (bytes)`
  - Unsigned offset integer immediate value (bytes) is added or subtracted (U bit above says to add or subtract) from the pointer address in the base register
  - Often used to address struct members
    - Address of struct is address of the first member and subsequent members are a fixed offset from the first based on their size of the preceding members

x

ldr/str Register Base + Immediate Offset Addressing



Syntax	Address	Examples
ldr/str Rd, [Rn, +/- constant]	Rn + or – constant	ldr r0, [r5,100]
constant is in bytes	same	str r1, [r5, 0]
ldr/str Rd, [Rn]		str r1, [r5]

x

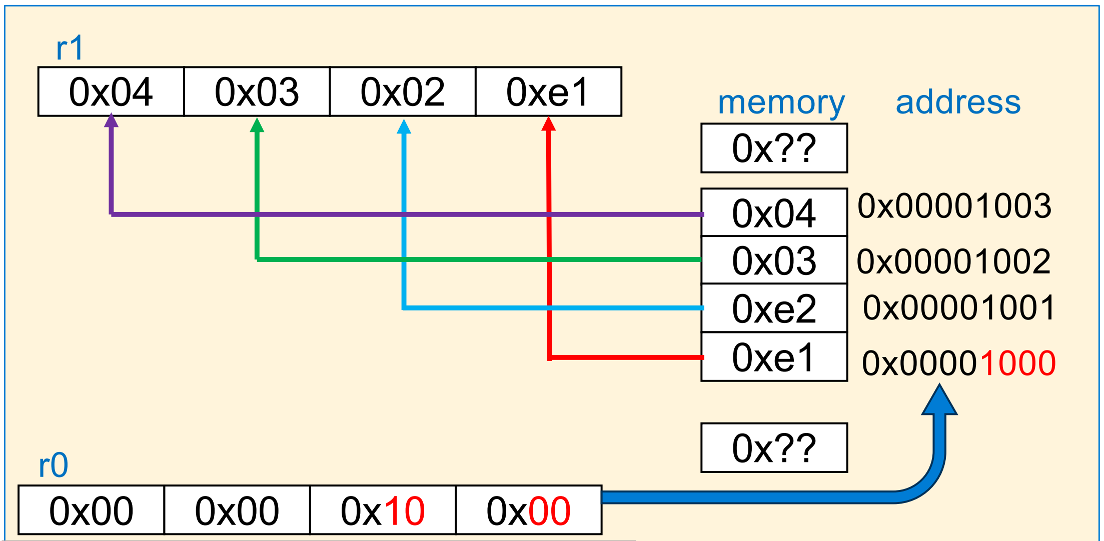
Loading and Storing: Variations List

- Load and store have variations that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will set the upper bits not filled from memory differently depending on which variation of the load instruction is used
- Store will only select the lower 8-bit, lower 16-bits or all 32-bits of the register to copy to memory, register contents are not altered

Instruction	Meaning	Sign Extension	Memory Address Requirement
`ldrsb`	load signed byte	sign extension	none (any byte)
`ldrb`	load unsigned byte	zero fill (extension)	none (any byte)
`ldrsh`	load signed halfword	sign extension	halfword (2-byte aligned)
`ldrh`	load unsigned halfword	zero fill (extension)	halfword (2-byte aligned)
`ldr`	load word	---	word (4-byte aligned)
`strb`	store low byte (bits 0-7)	---	none (any byte)
`strh`	store halfword (bits 0-15)	---	halfword (2-byte aligned)
`str`	store word (bits 0-31)	---	word (4-byte aligned)

x

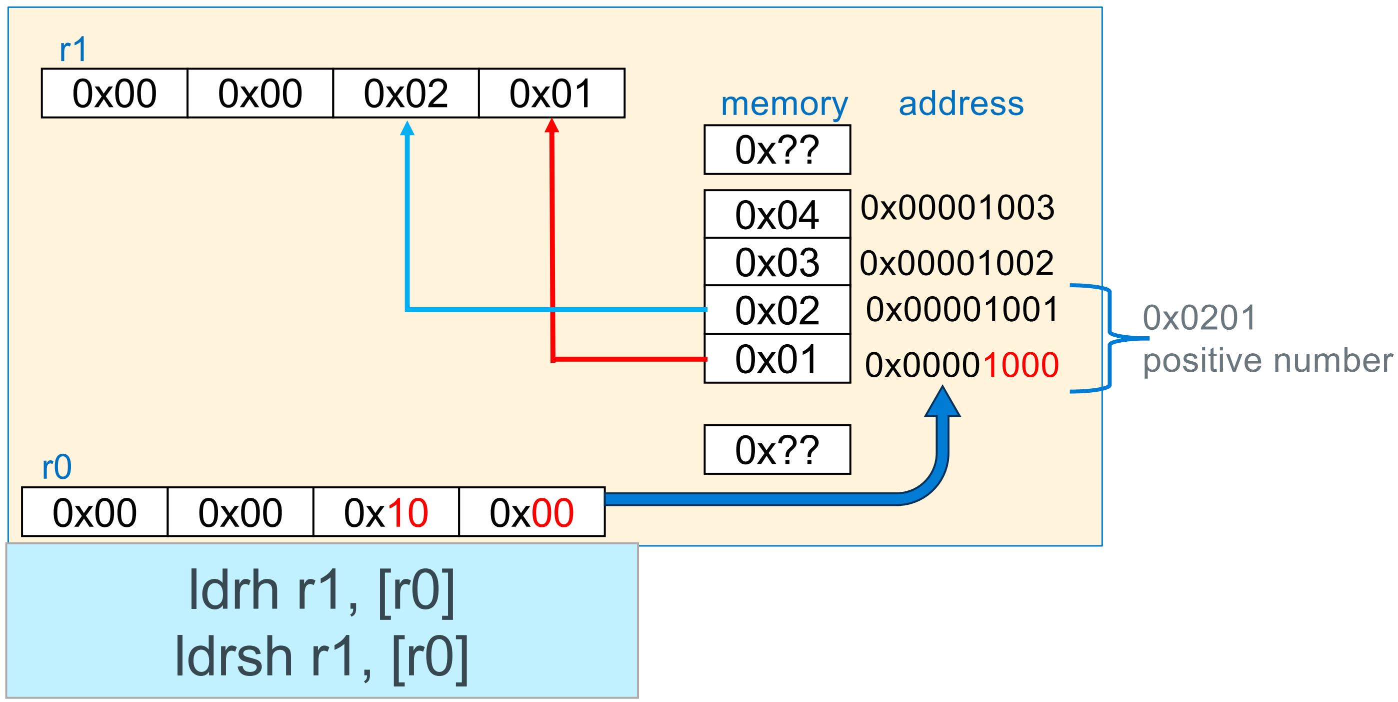
Loading 32-bit Registers From Memory, 32-bit



ldr r1, [r0]

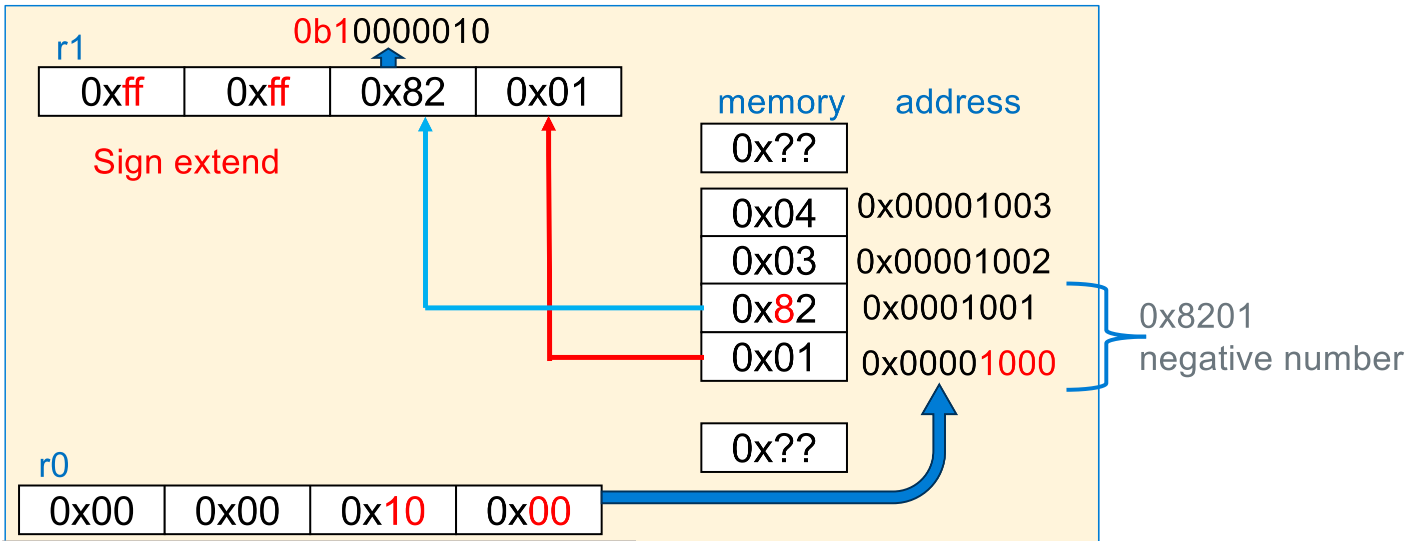
x

Loading 32-bit Registers From Memory, 16-bit



x

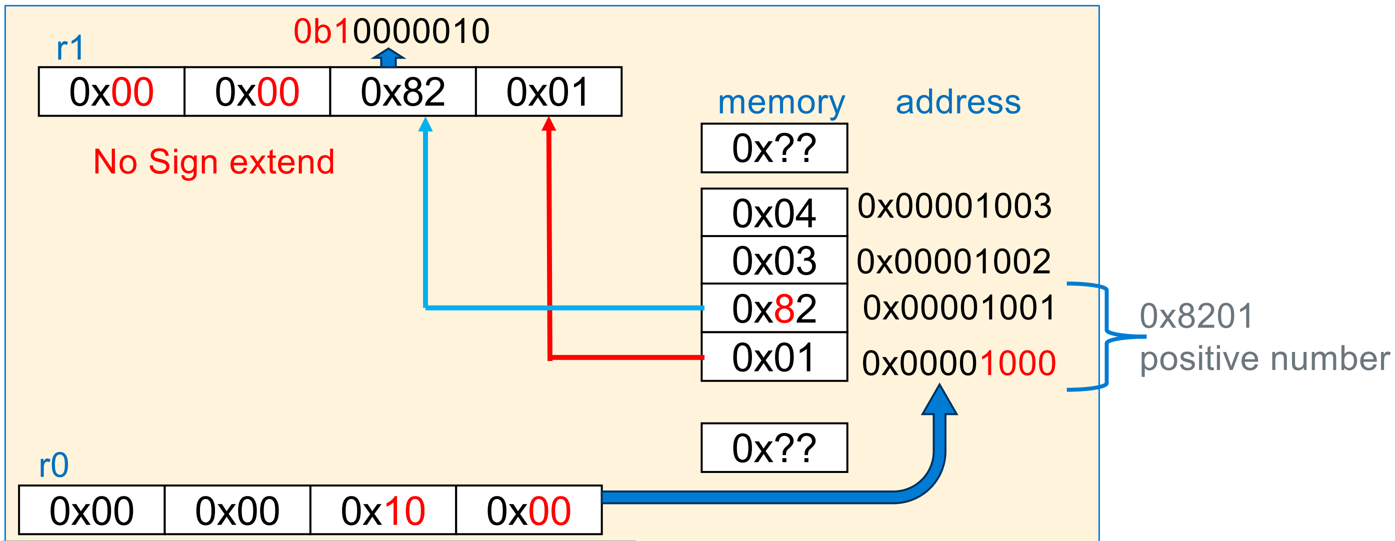
Loading 32-bit Registers From Memory, 16-bit Signed



ldrsh r1, [r0]

x

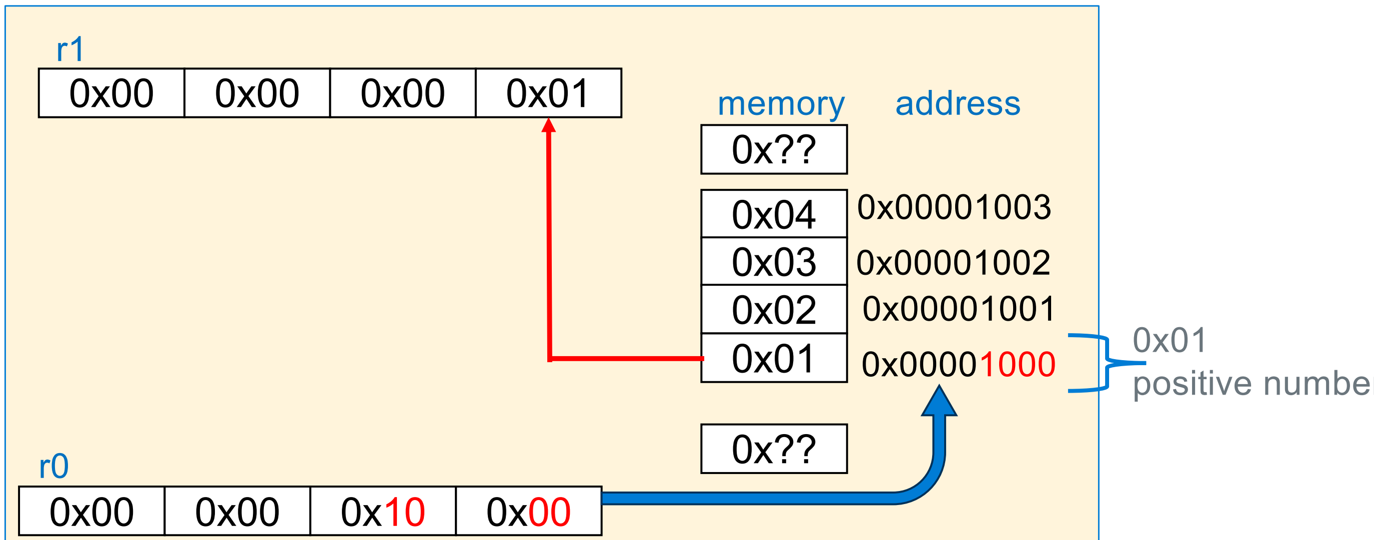
Loading 32-bit Registers From Memory, 16-bit Unsigned



ldrh r1, [r0]

x

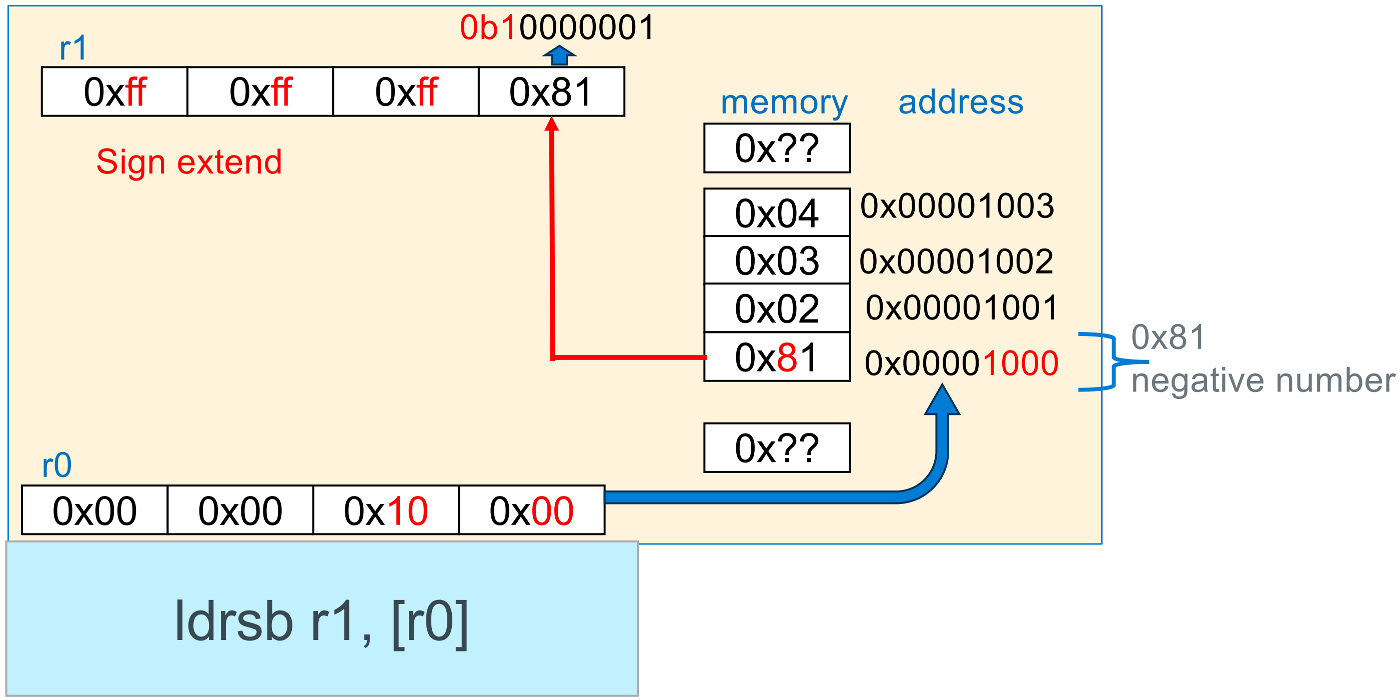
Loading 32-bit Registers From Memory, 8-bit



ldrb r1, [r0]
ldrsb r1, [r0]

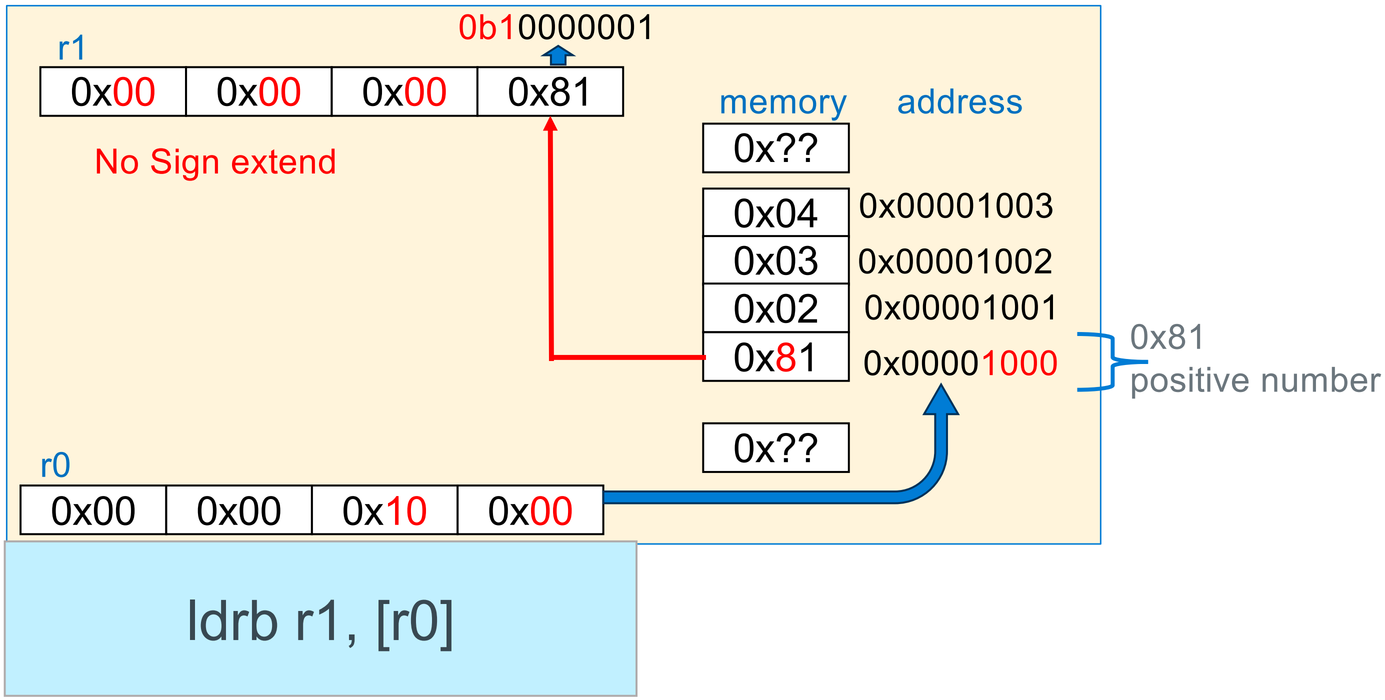
x

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



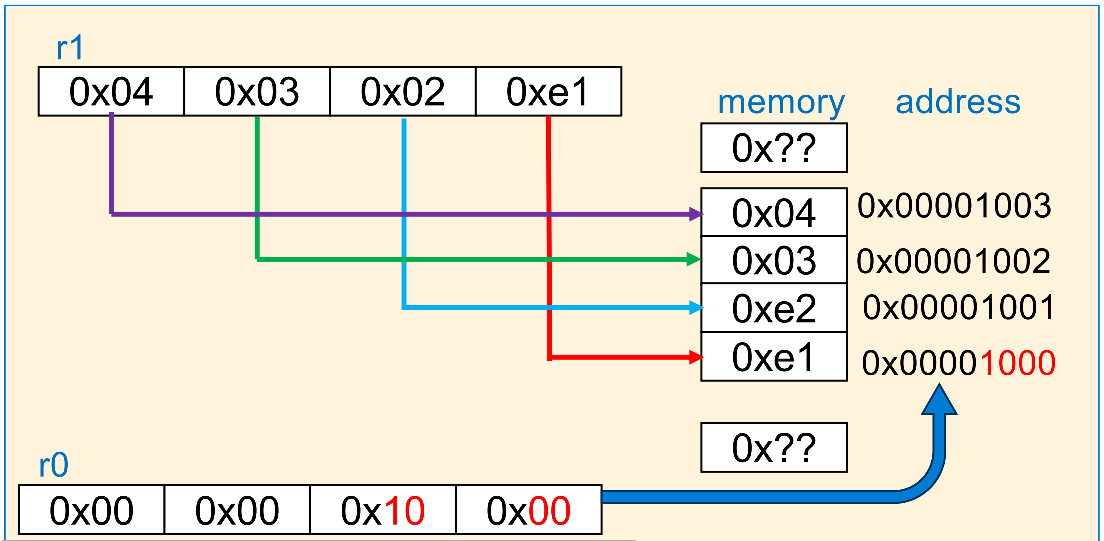
x

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



x

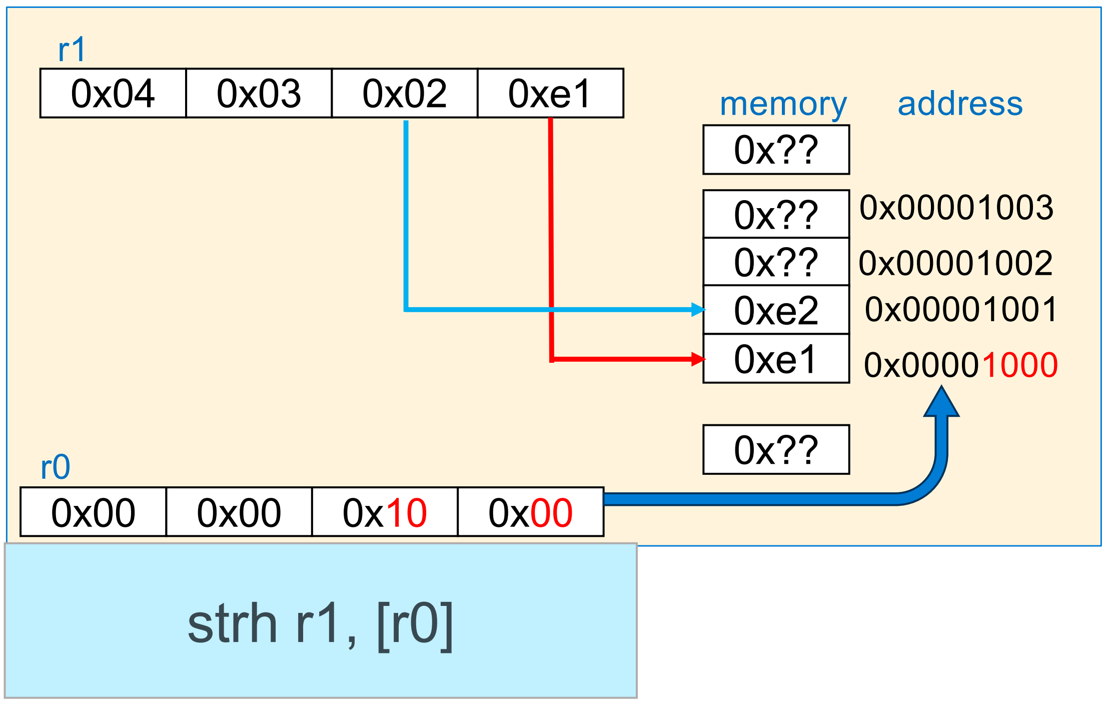
Storing 32-bit Registers To Memory, 32-bit



str r1, [r0]

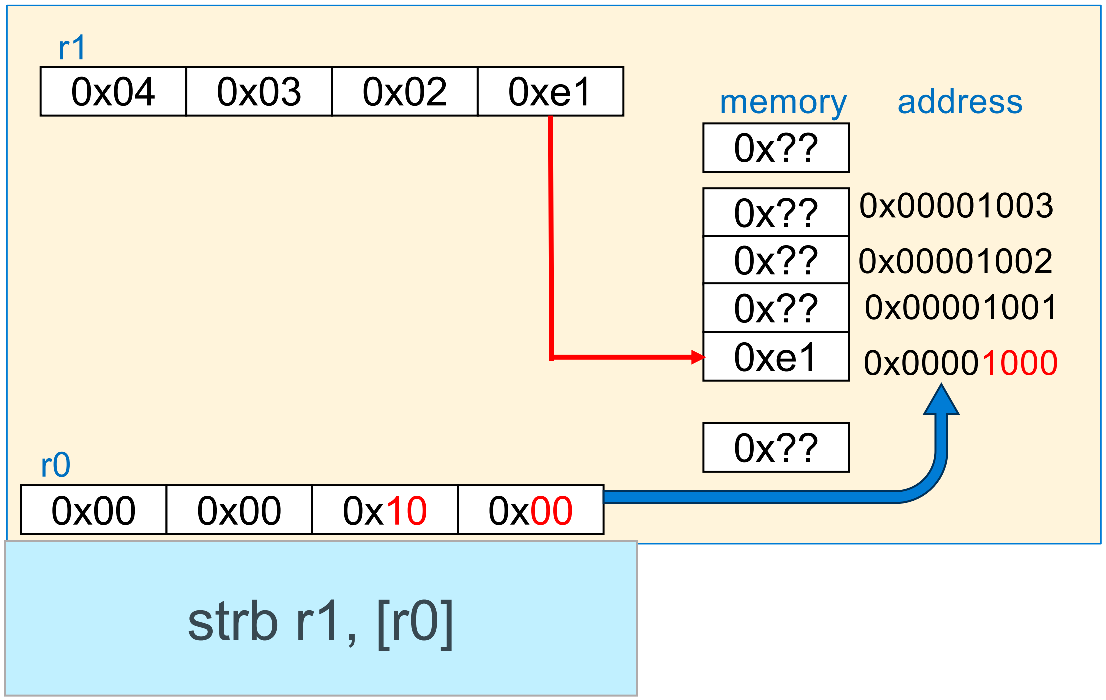
x

Storing 32-bit Registers To Memory, 16-bit



x

Storing 32-bit Registers To Memory, 8-bit



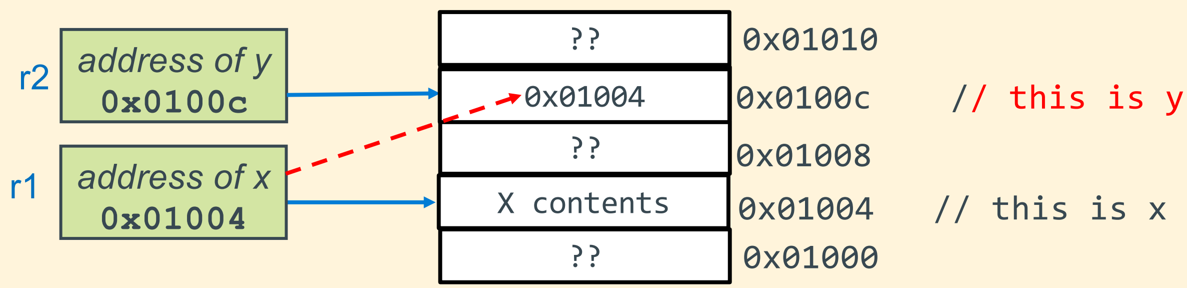
x

ldr/str practice - 1

r1 contains the Address of X (defined as int X) in memory; r1 points at X

r2 contains the Address of Y (defined as int *Y) in memory; r2 points at Y

write Y = &X;




```
str   r1, [r2]       // y ← &x
```

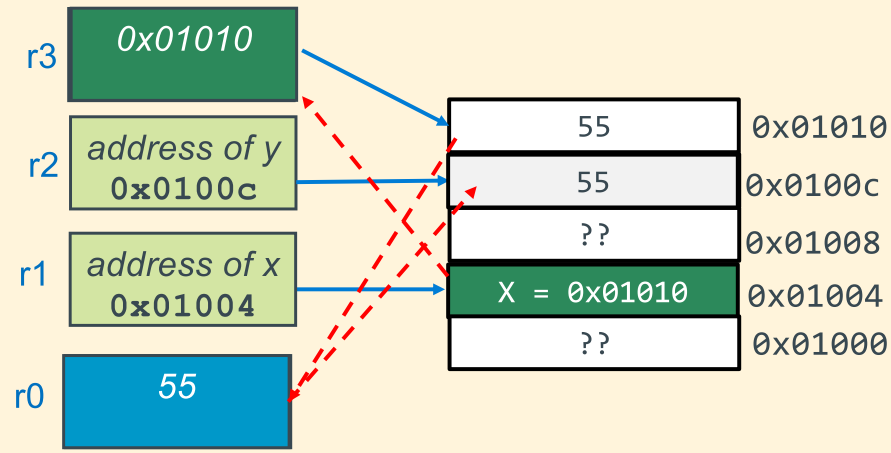

x

ldr/str practice - 2

r1 contains the Address of X (defined as int *X) in memory r1 points at X

r2 contains the Address of Y (defined as int Y) in memory; r2 points at Y

write Y = *X;




```
ldr    r3, [r1]  // r3 ← x (read 1)
ldr    r0, [r3]  // r0 ← *x (read 2)
str    r0, [r2]  // y ← *x
```


x

using ldr/str: array copy

```c
#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;
}
```

```c
void icpy(int *src, int *dst, int cnt)
{
    for (int i = 0; i < cnt; i++)
        *dst++ = *src++;

    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
    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

.Ldone:
```

pre loop guard

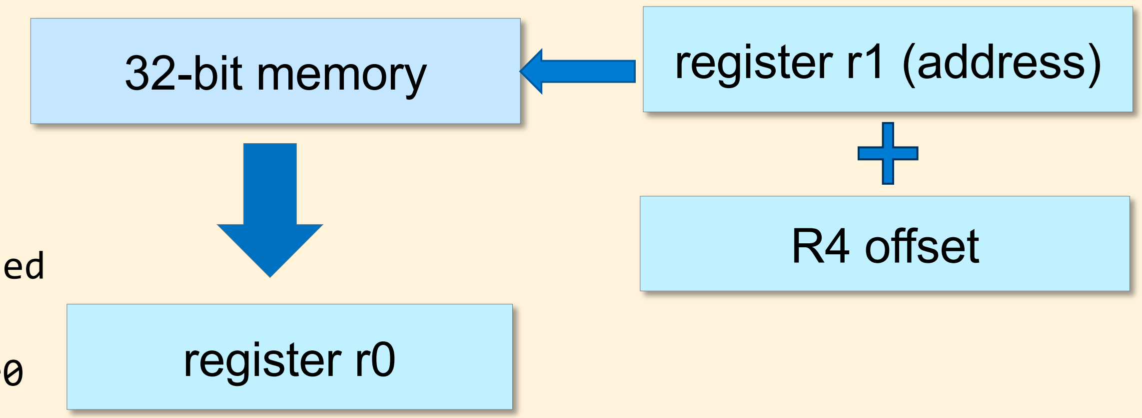
loop guard

x

Load/Store: Register Base Addressing + Register Offset

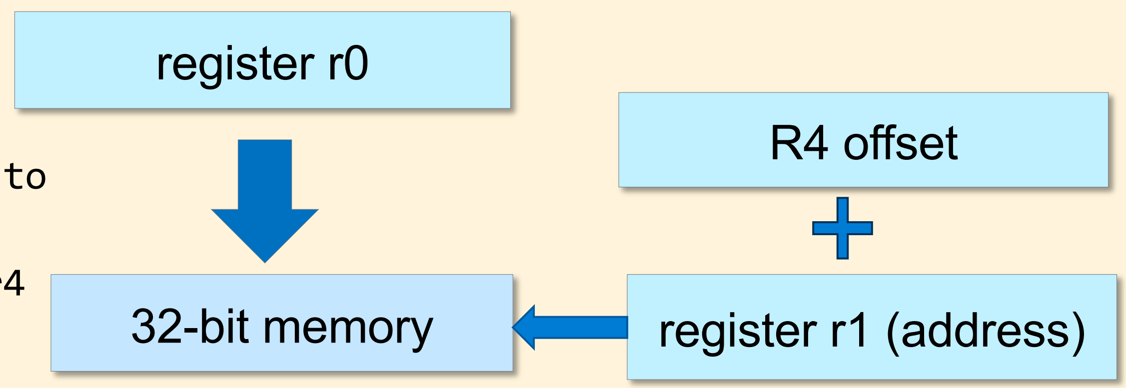
```
ldr r0, [r1, r4]
```

Copies a 32-bit word from the memory location whose address is contained in r1 + r4 (r1 is a pointer) into register r0



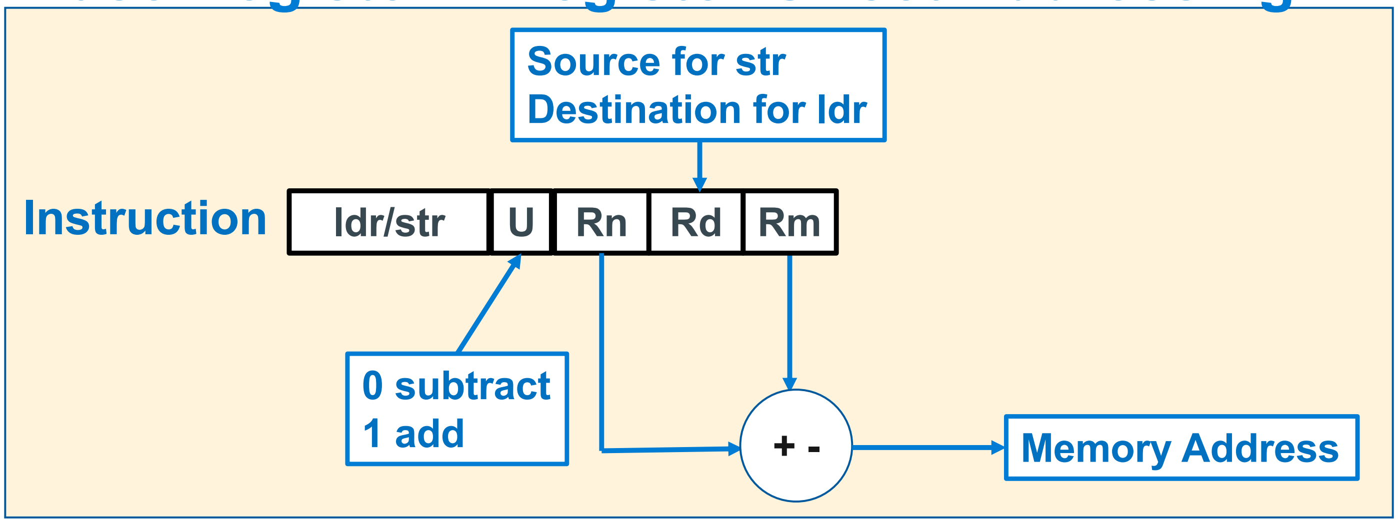

```
str   r0, [r1, r4]
```

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1+r4 (r1 pointer)



x

ldr/str Base Register + Register Offset Addressing



Pointer Address = Base Register + Register Offset

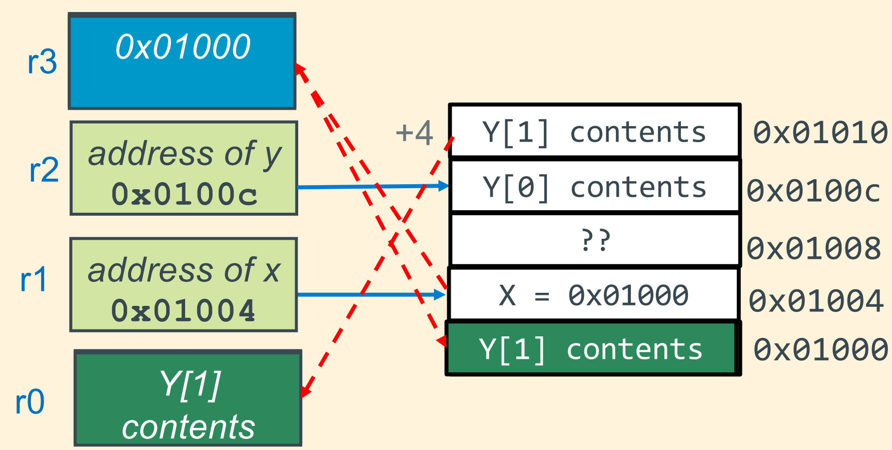
- **Unsigned** offset integer **in a register** **(bytes)** is either added/subtracted from the **pointer address** in the **base register**

Syntax	Address	Examples
`ldr/str Rd, [Rn +/- Rm ]`	`Rn + or – Rm`	`ldr r0, [r5, r4]` `str r1, [r5, r4]`

x

ldr/str practice - 3

```
r1 contains Address of X (defined as int *X) in memory; r1 points at X
r2 contains Address of Y (defined as int Y[2]) in memory; r2 points at &(Y[0])
write *X  = Y[1];
```

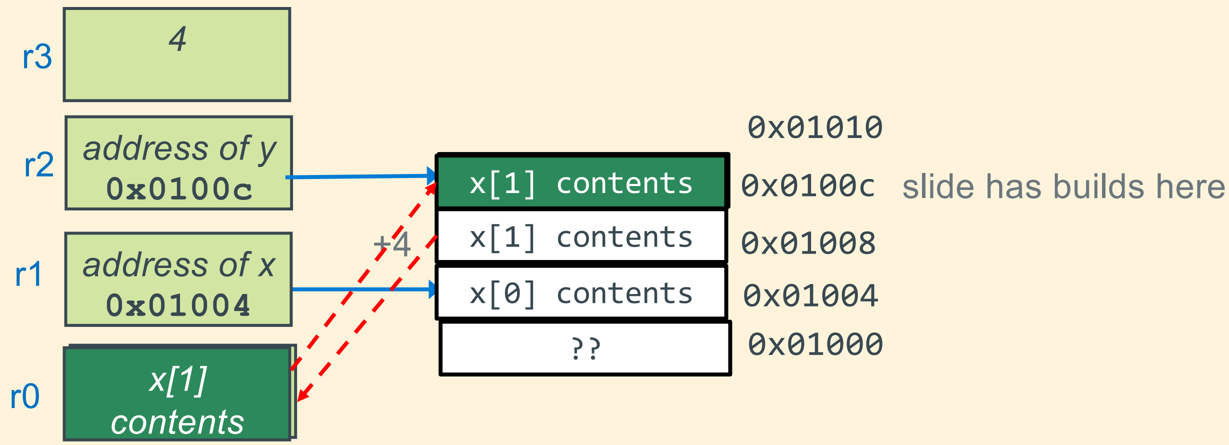



```
ldr    r0, [r2, 4]        // r0 ← y[1]
ldr    r3, [r1]           // r3 ← x
str    r0, [r3]           // *x ← y[1]
```


x

ldr/str practice - 4

```
r1 contains Address of X (defined as int X[2]) in memory; r1 points at &(x[0])

r2 contains Address of Y (defined as int Y) in memory; r2 points at Y

r3 contains a 4

write Y = X[1];
```




```
ldr   r0, [r1, r3]  // r0 ← x[1]

str   r0, [r2]    // y ← x[1]
```


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

```
#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

Reference: Addressing Mode Summary for use in CSE30

index Type	Example	Description
Pre-index immediate	`ldr r1, [r0]`	r1 ← memory[r0] r0 is unchanged
Pre-index immediate	`ldr r1, [r0, 4]`	r1 ← memory[r0 + 4] r0 is unchanged
Pre-index immediate	`str r1, [r0]`	memory[r0] ← r1 r0 is unchanged
Pre-index immediate	`str r1, [r0, 4]`	memory[r0 + 4] ← r1 r0 is unchanged
Pre-index register	`ldr r1, [r0, +-r2]`	r1 ← memory[r0 +- r2] r0 is unchanged
Pre-index register	`str r1, [r0, +-r2]`	memory[r0 +- r2] ← r1 r0 is unchanged

x

Variable Alignment In Memory and Performance

Accessing **address aligned** memory on many systems based on data type has the best performance (due to hardware implementation)

char

1 any address

short

2 bytes addresses that end in 0b0

integer

4 bytes addresses that end in 0b00

x

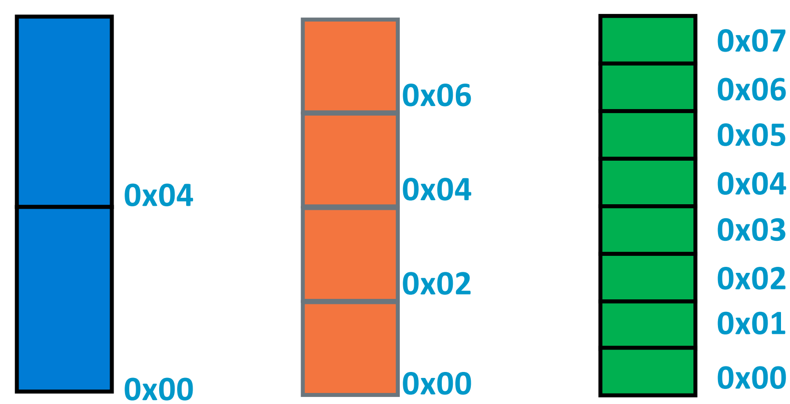
Defining Static Variables: Allocation and Initialization

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	1	`short length = 0x55aa;`	`length:  .short 0x55aa`

32-bit int (4 bytes)	.word
	.long
		2	`int dist = 5;`
			`int *distptr = &dist;`
			`unsigned int mask = 0xaa55aa55;`
			`int array[] = {12,~0x1,0xCD,-1};`
				`dist:    .word 5`
				`distptr: .word dist`
				`mask:    .word 0xff`
				`array:   .word 12,~0x1,0xCD,-3`

string with '\0'	.string		`char class[] = "cse30";`	`class:   .string "cse30"`

SIZE	Address ends in	Align
8-bit char -1 byte	0b..0 or 0b..1	
16-bit int -2 bytes	0b..0	.align 1
32-bit int -4 bytes and all arrays	0b..00	.align 2



x

Defining Static Variables: Allocation and Initialization

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

```
00012028 <chx>:
   12028:  00000041

00012029 <cnt>:
   12029: 00000004
```

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

```
00012028 <chx>:
   12028:  00000041

0001202c <cnt>:
   1202c: 00000004
```

```
int num;             //4 bytes
int *ptr = &num;     //4 bytes
char *lit = "456";   //4 bytes,"456" string literal
char msg[] = "123";  //4 bytes - array
```

```
.bss

num:        .word 0

.data
            .align 2
ptr:        .word num
            .align 2
lit:        .word .Lmsg
            .align 2
msg:        .string "123"

.section .rodata
.Lmsg:      .string "456"
```

Optional

initializes a pointer

Sets alignment for **next** variable

read-only string literal

x

Defining Static Array Variables

```
Label:   .size_directive expression, … expression
```

```
In C:      int int_buf[100];
           int array[] = {1, 2, 3, 4, 5};
           char buffer[100];
```

```
.bss
int_buf:   .space 400   // convert 100 to 400 bytes
            .align 2
char_buf:  .space 100
.data
array:     .word 1, 2, 3, 4, 5
           .align 2
one_buf:   .space 100, 1     // 100 bytes each byte filled with 1
```

`.space size, fill`

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

x

Loading Static variable address into a register

- Tell the assembler load the address (Lvalue) of a label into a register:

  `ldr/str  Rd, =Label // Rd = address`

- *Example to the right: y = x;*

two step to **load** a **memory** variable

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

two steps **store** to a **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

Loading large Constants into a register:

Error: invalid constant (3ff) after fixup

- In data processing instructions, the field **imm8 + rotate 4 bits** is too small to store store the immediate value, how do you get larger immediate values into a register?

mov	Rd	rot4	imm8

fails
```
mov     r0, 1023
```
xxx.s:24: Error: invalid constant (3ff) after fixup

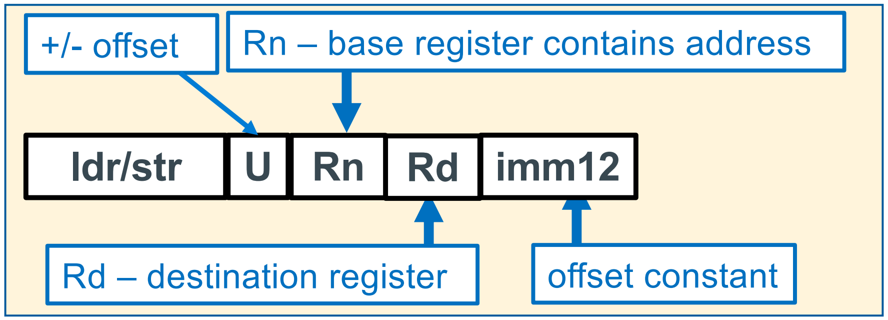
replacement
```
ldr     r0, =1023
```

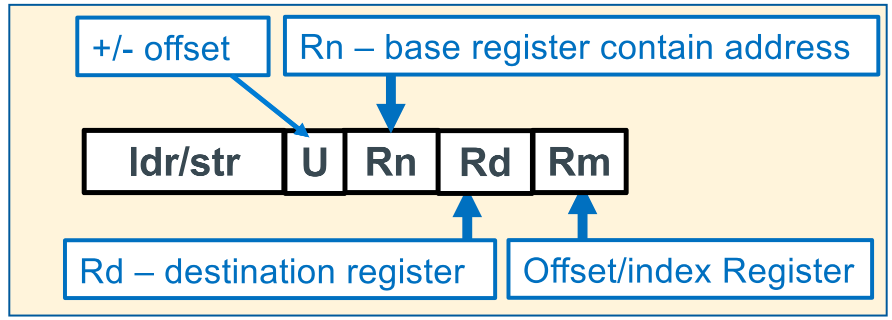
- Answer: use **`ldr`** instruction with the constant as an operand: **`=constant`**
- Assembler creates a **literal table entry** with the **constant**

```
ldr  Rd, =constant        // =constant
ldr  r1, =0x2468abcd      // loads the constant 0x246abcd into r1
```


x

LDR/STR – Register To/From Memory Copy






```
ldr/str  Rd,  [Rn, +- imm12] // base register pointer + offset  imm12 in bytes
                             -4095 <= imm12 <= 4095 (bytes)
ldr/str  Rd,  [Rn]           // base register pointer + 0 (imm12 is 0)
ldr/str  Rd,  [Rn, +- Rm]    // base register pointer +- offset register
```

```
ldr          r1, =var_x          // r1 = &var_x
str          r1, =mylabel+4      // *(mylabel+4) = r1
ldr          r1, =0x246abcd      // load an immediate into r1
ldr          r1, [r3]           // y = *r3 (4 bytes)
str          r1, [r0]           // *r0 = r1
ldr          r1, [r3, -4]       // y = *(r3 – 4) (4 bytes)
str          r1, [r0, r2]       // *(r0 + r2) = r1
```


x

Function Calls, Parameters and Locals: Requirements

```
int
main(int argc, char *argv[])
{
    int x, z = 4;

    x = a(z);
    z = b(z);
    return EXIT_SUCCESS;
}

int
a(int n)
{
    int i = 0;
    if (n == 1)
        i = b(n);
    return i;
}

int
b(int m)
{
    return m+1;
/* the return cannot be done with a
   branch */
}
```

- Since b() is called both by main and a() how does the **return m+1 statement in b() know where to return to? (Obviously, it cannot be a branch)**
- Where are the parameters (args) to a function stored so the function has a copy that it can alter?
- Where is the return value from a function call stored?
- How are Automatic variables *lifetime* and *scope* **implemented**?
    - When you enter a variables scope: memory is allocated for the variables
    - When you leave a variable scope: memory lifetime is ended (memory can be reused -- deallocated) – contents are **no longer valid**

x

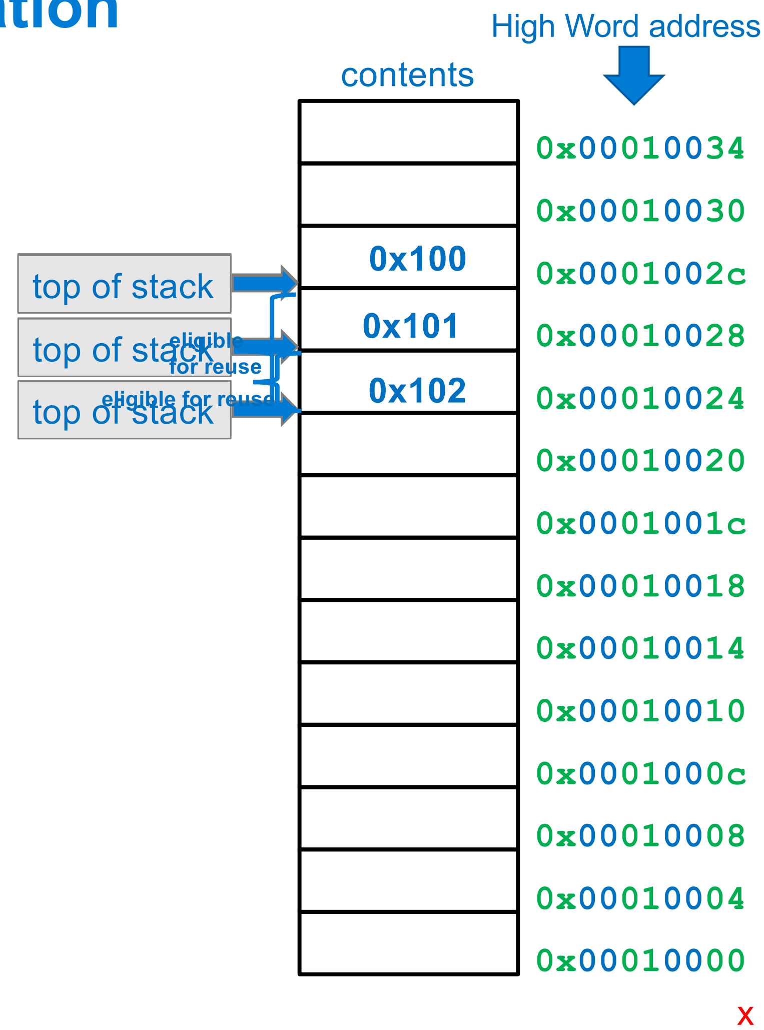
Data Structure Review: Stack Operation

- A Stack Implements a **last-in first-out** (LIFO) protocol
- **Stacks** are expandable and **grow downward** from high memory address towards low memory address
- **Stack pointer 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

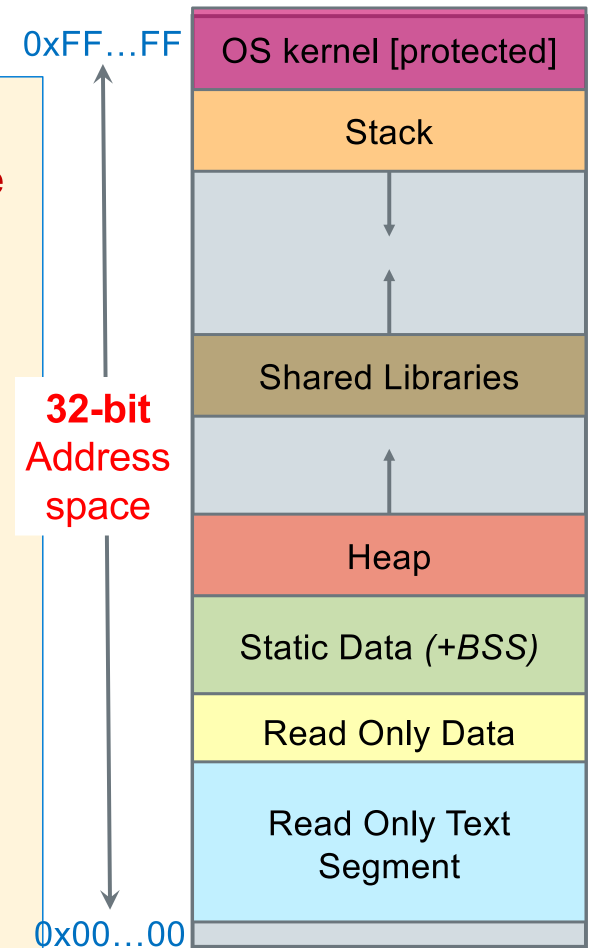
- Existing items are popped (*removed*) from the top of the stack by adding to the stack pointer the size of the element (leaving the ***old contents unchanged***)

pop (sp + element size)



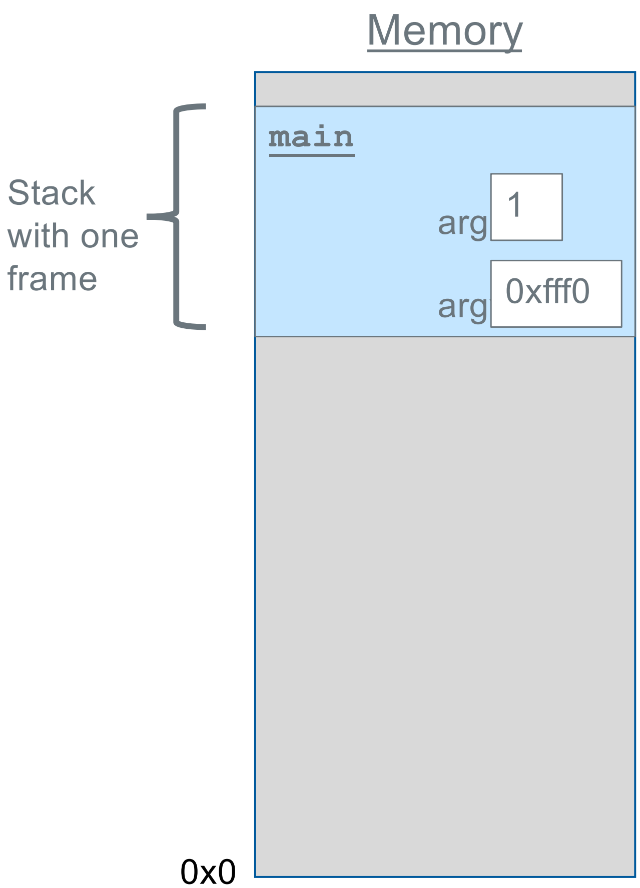
Stack Segment: Support of Functions

- 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)



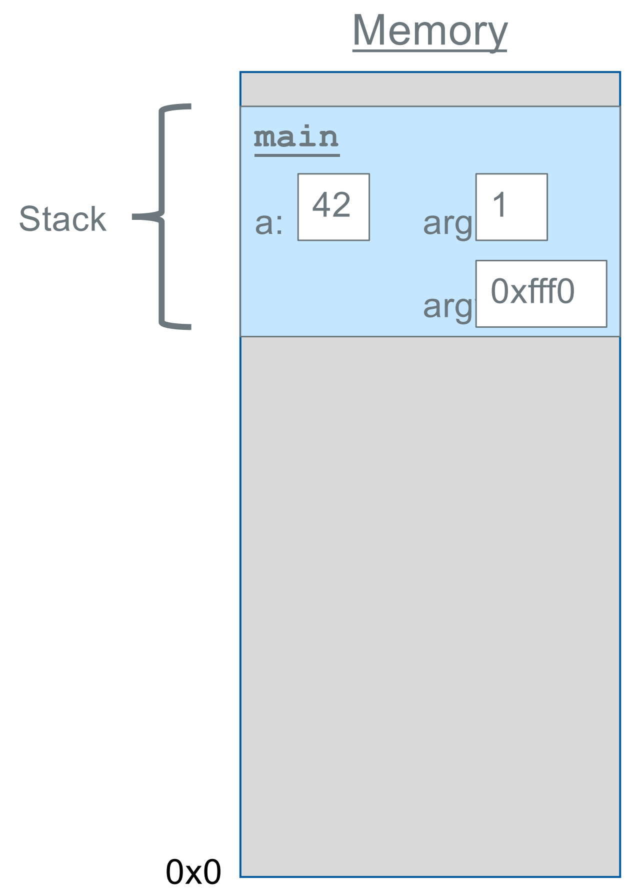
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



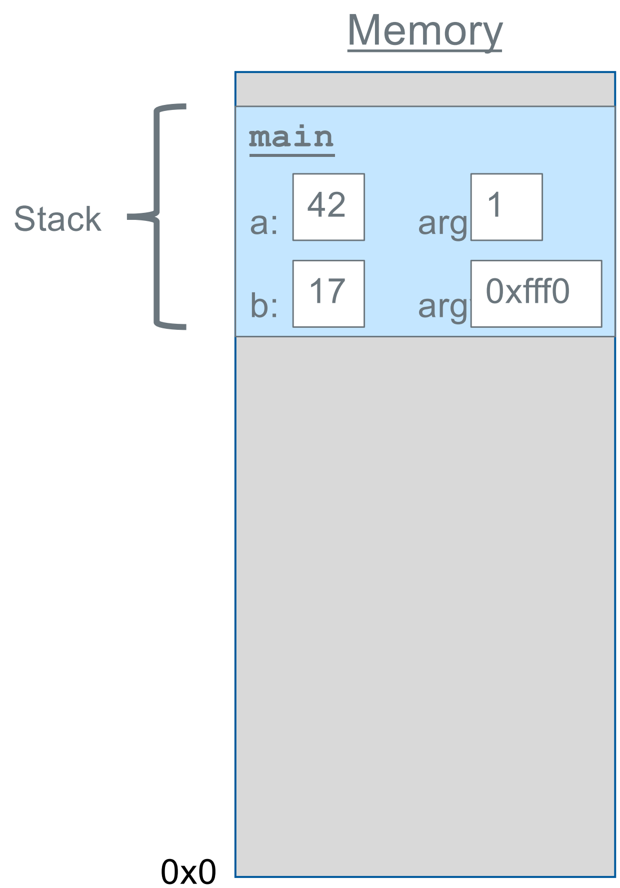
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



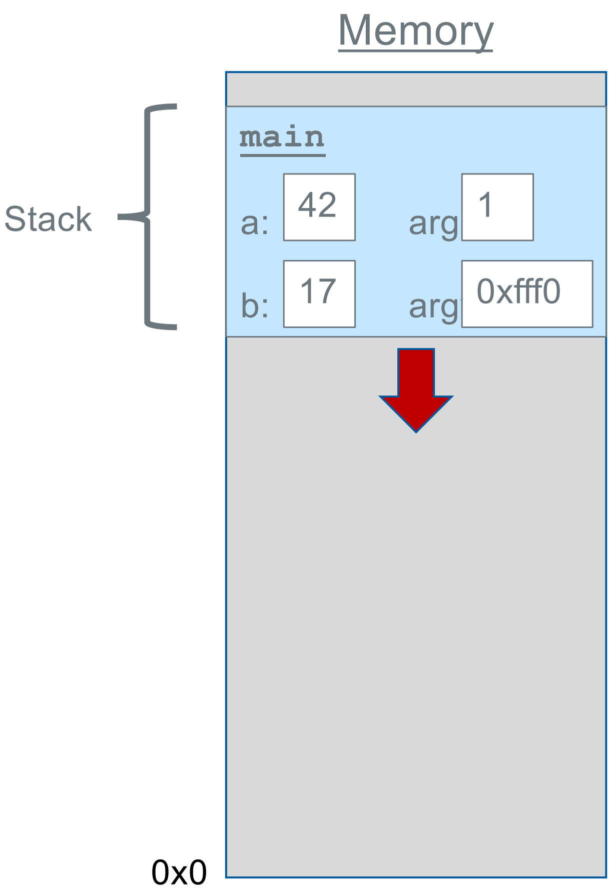
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



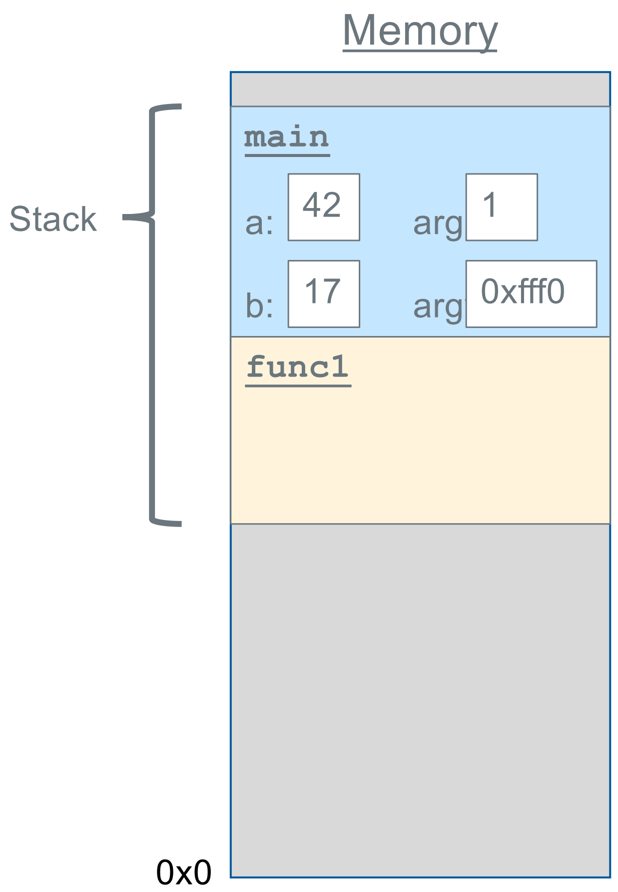
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



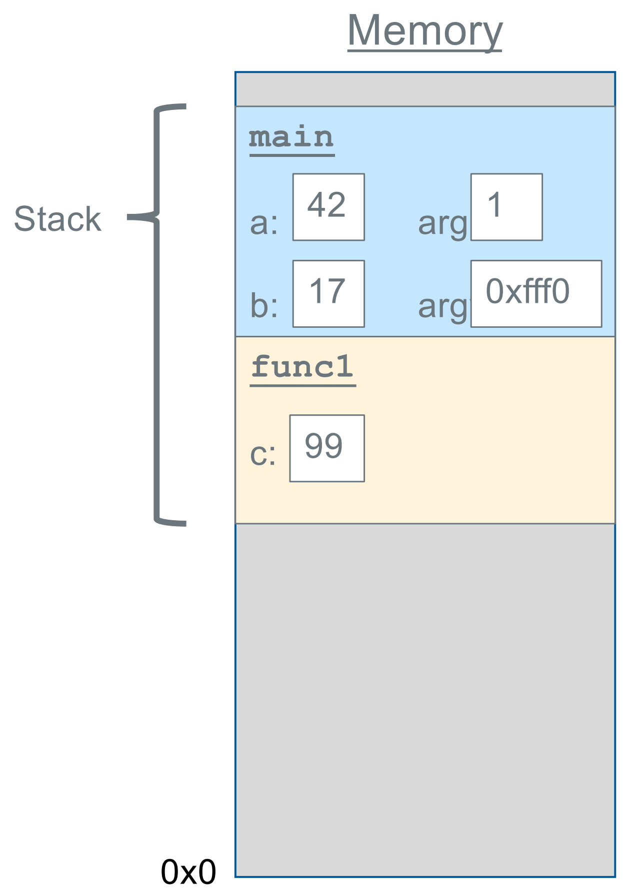
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



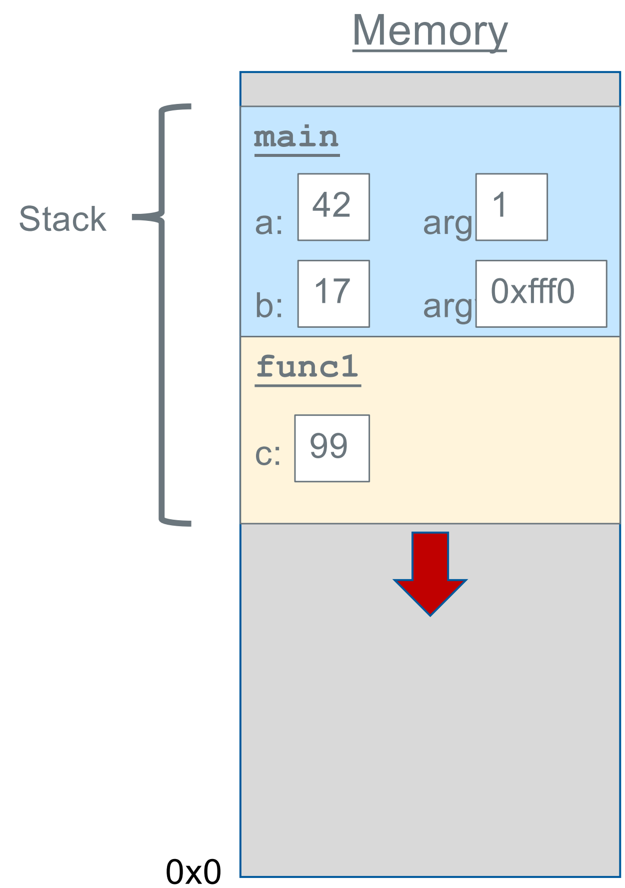
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



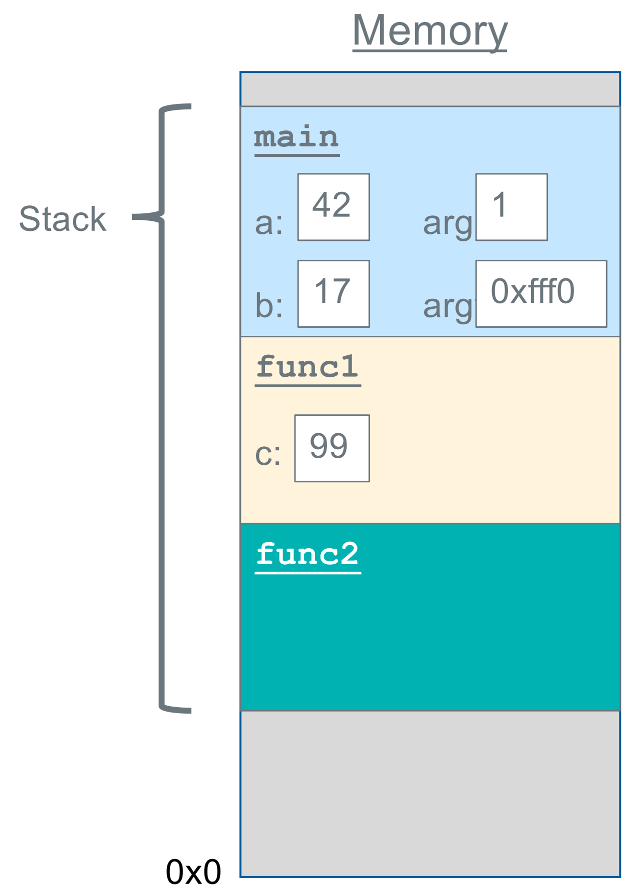
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



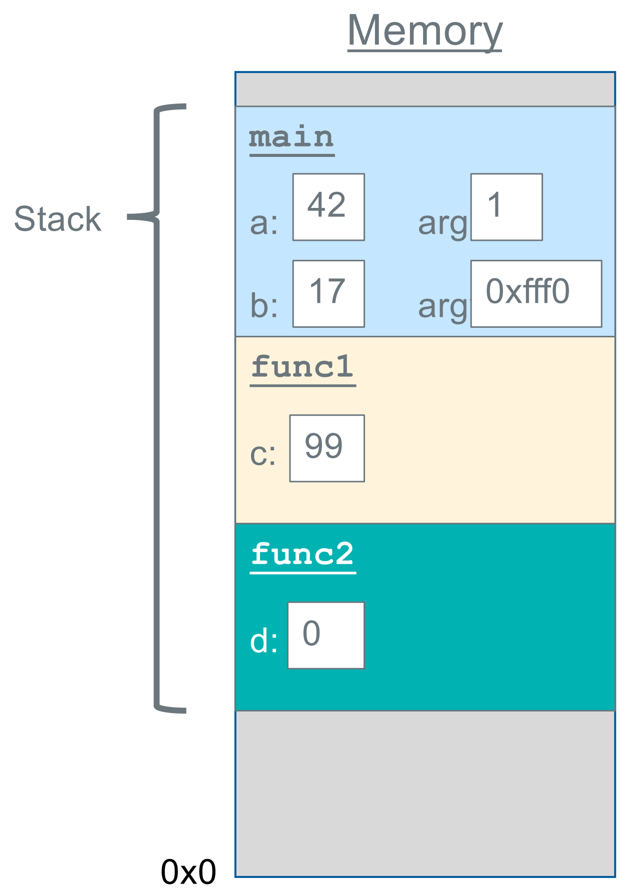
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



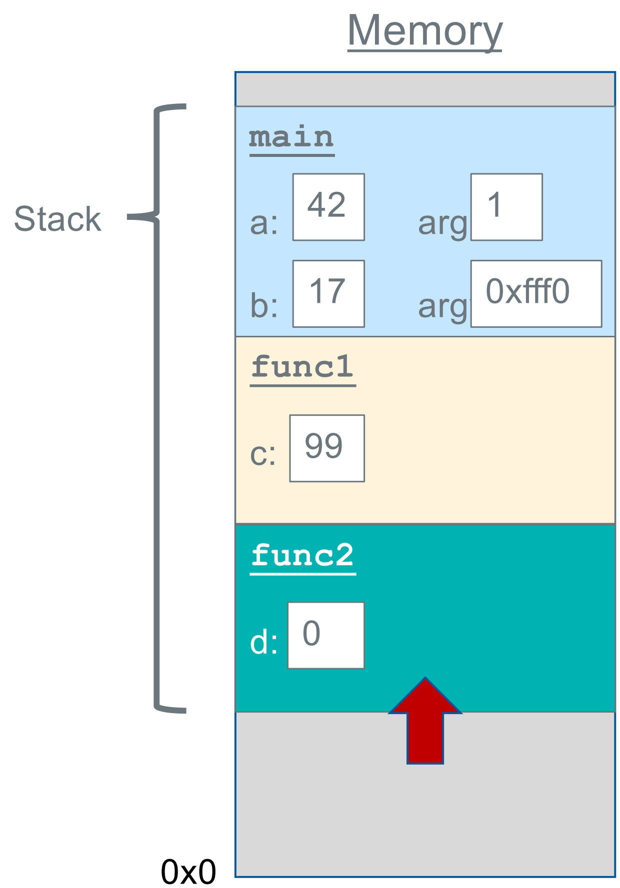
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



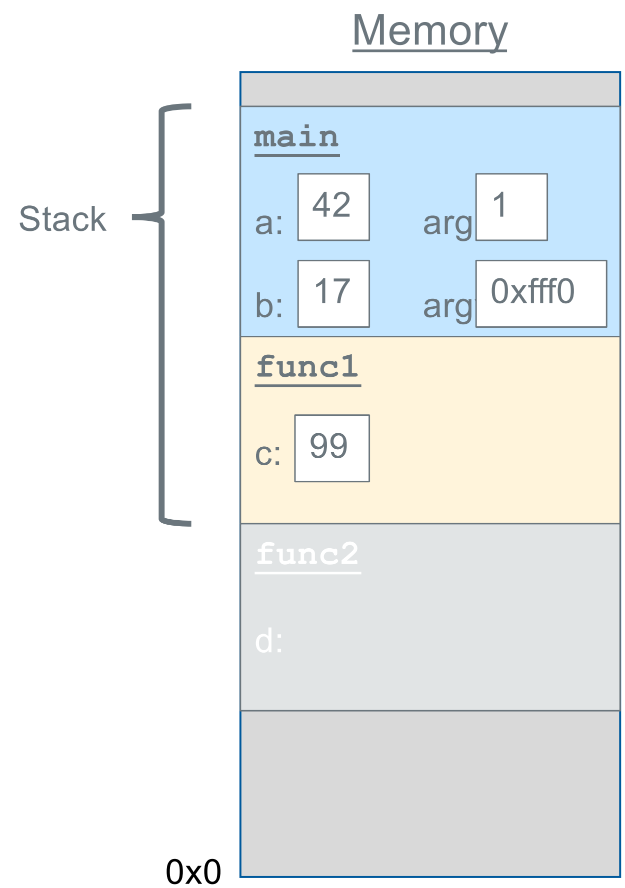
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



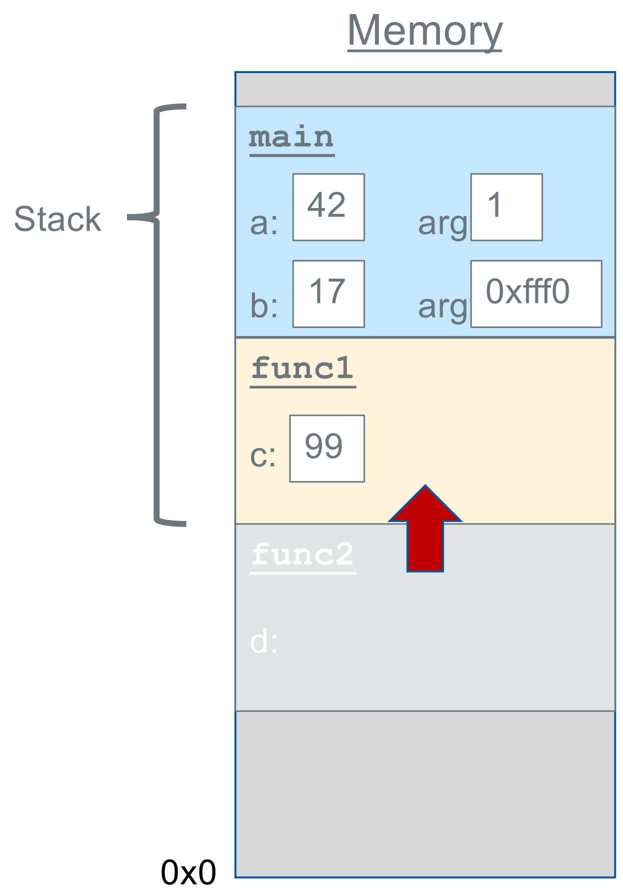
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



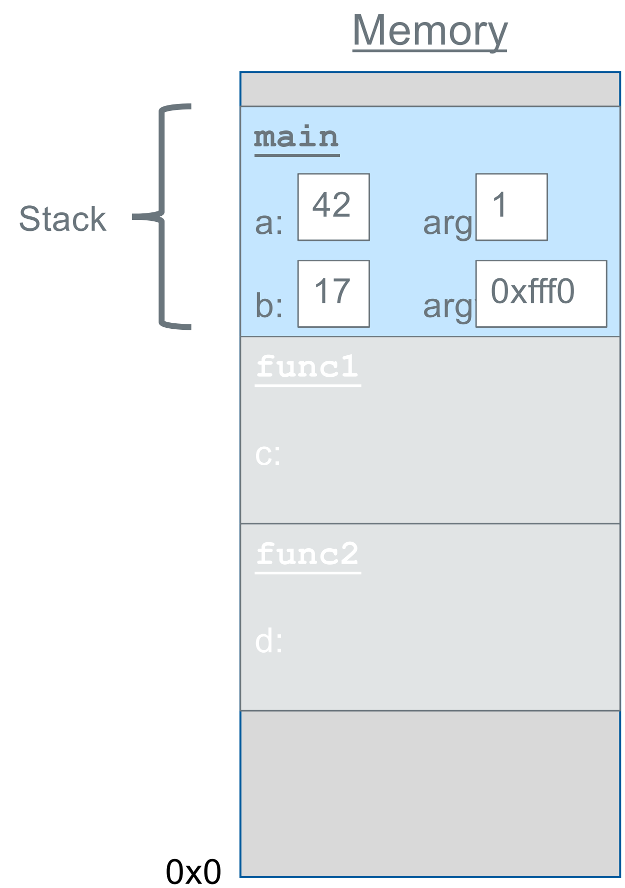
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



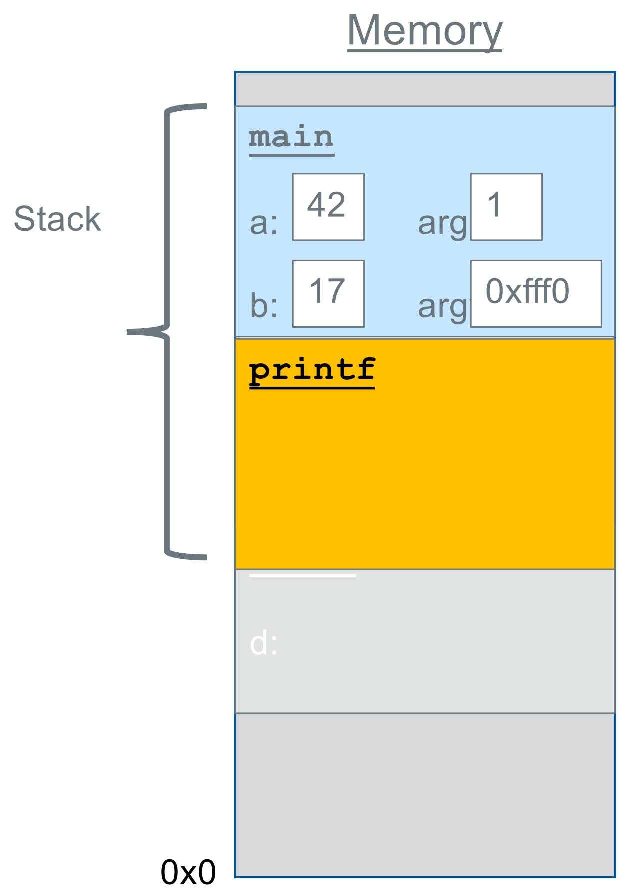
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



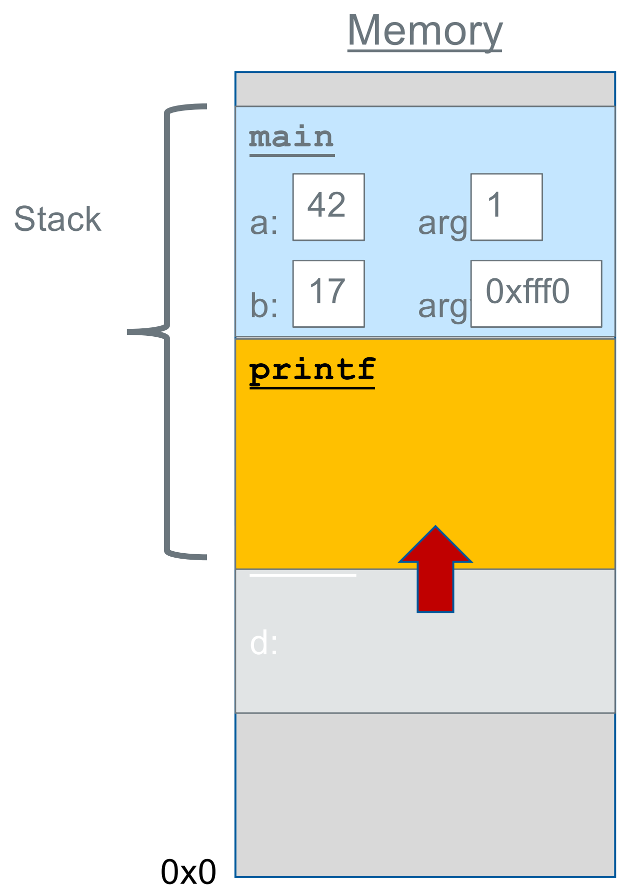
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



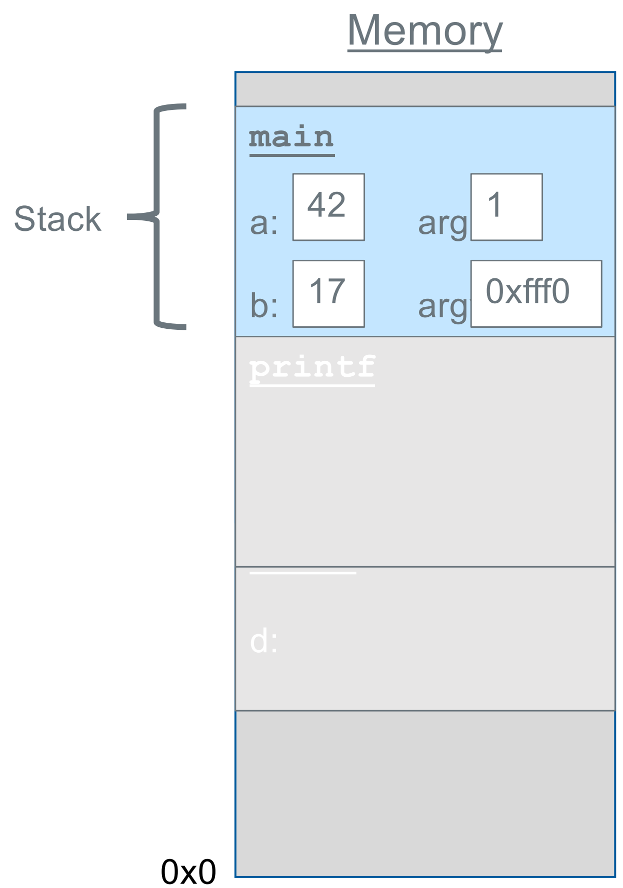
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



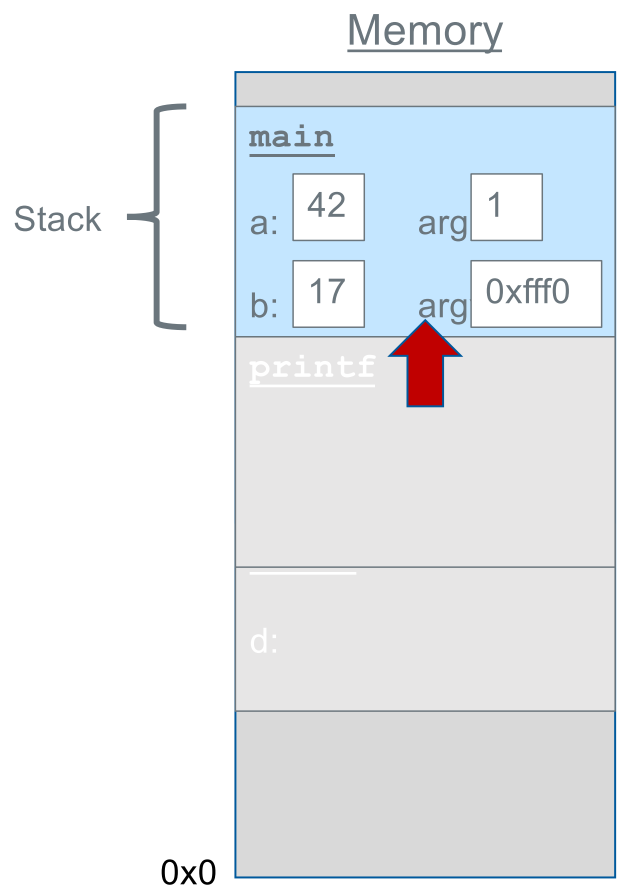
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



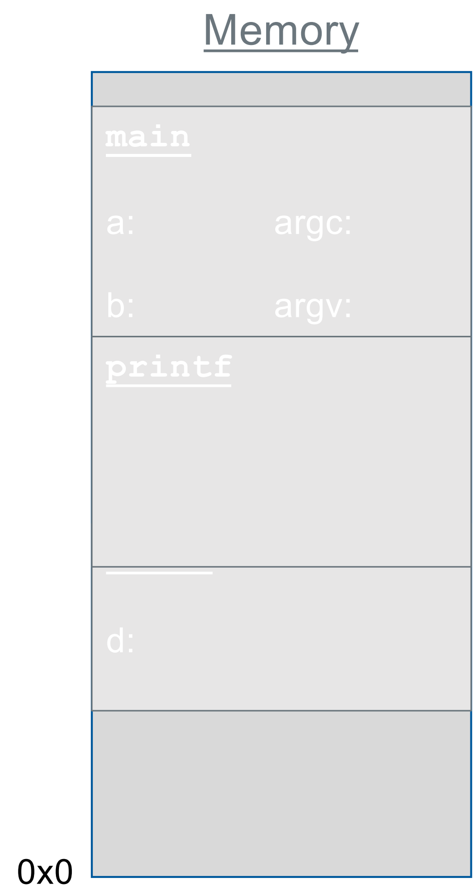
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



The Stack

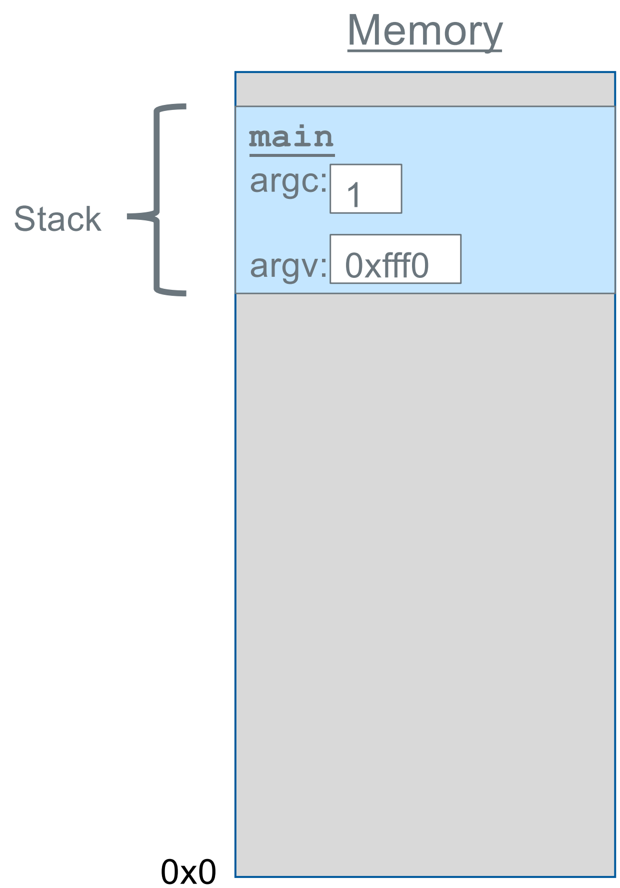
```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



The Stack - Recursion

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

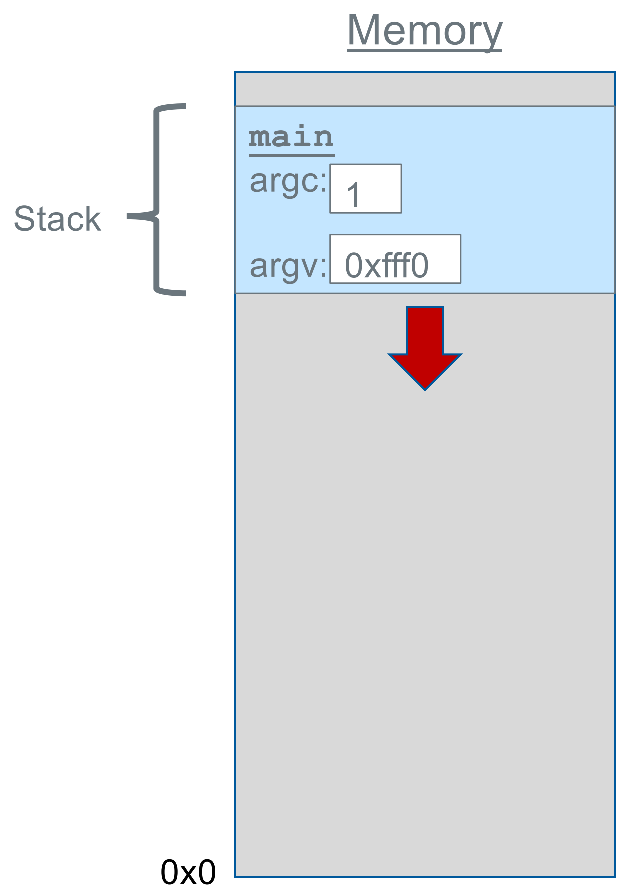
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

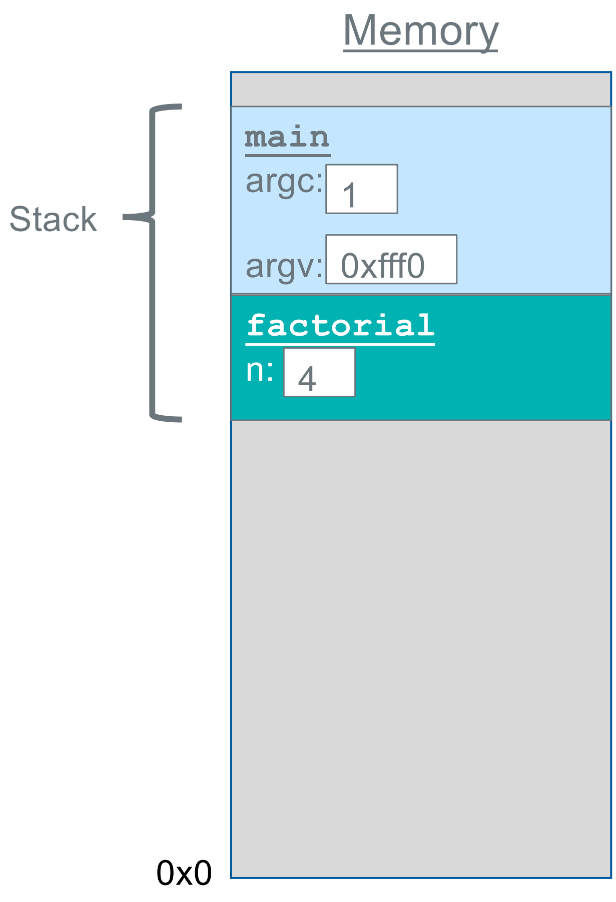
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

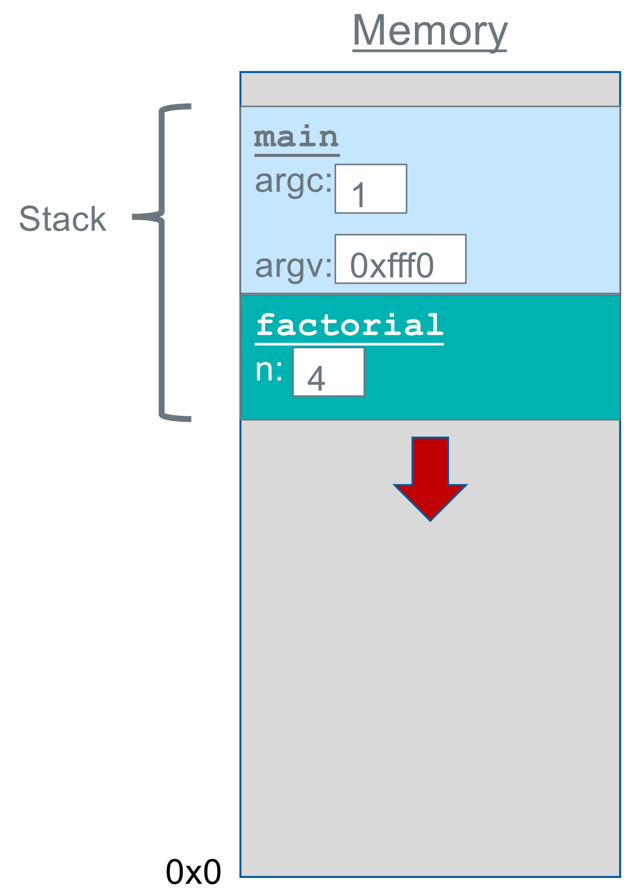
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

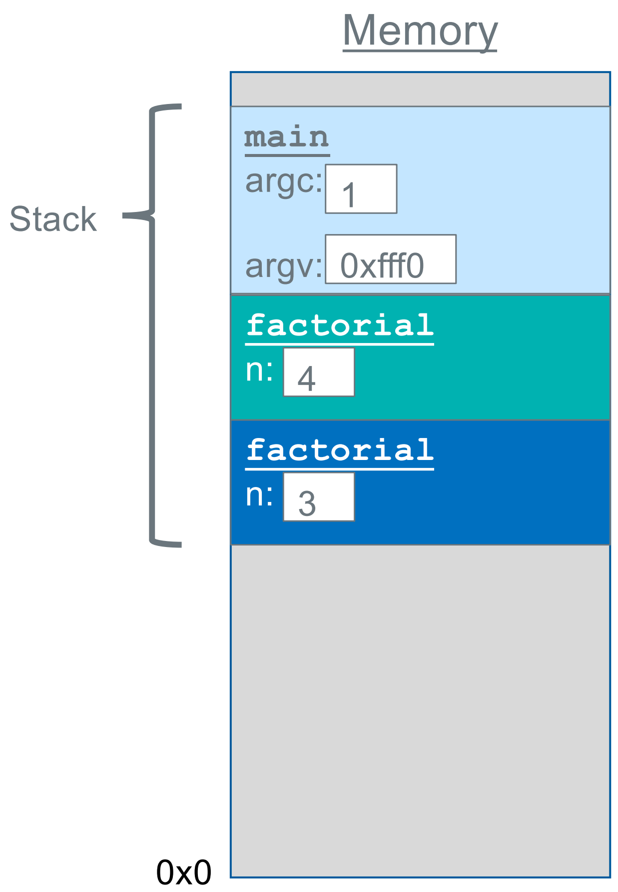
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

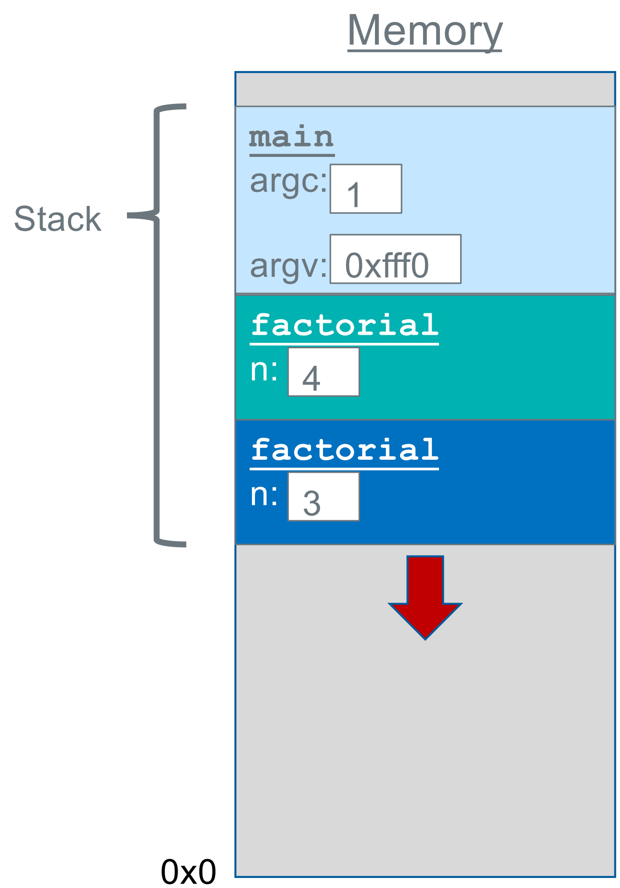
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

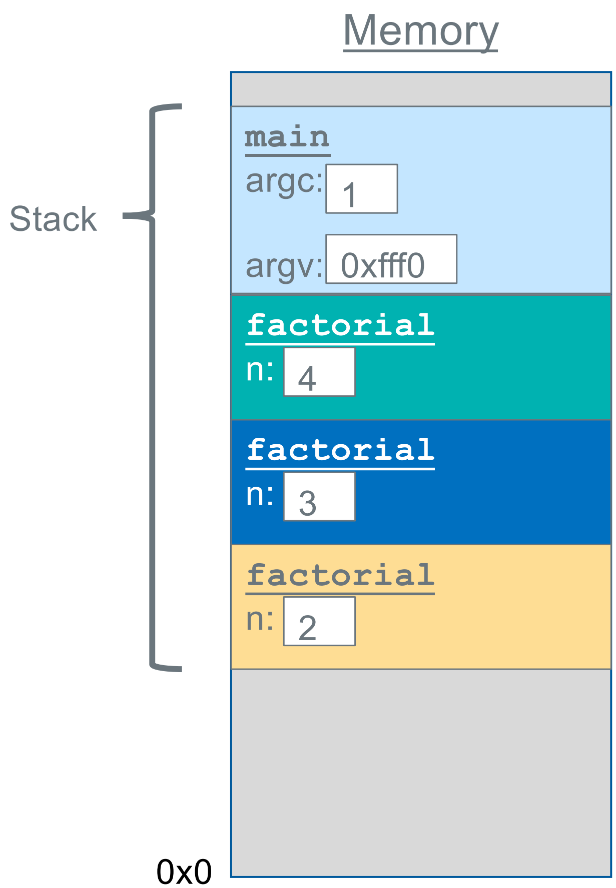
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

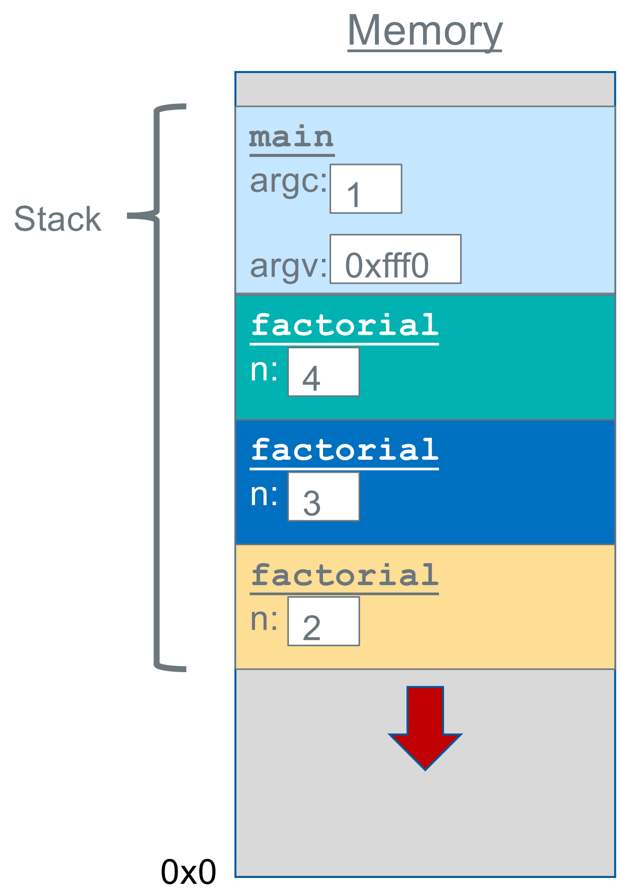
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

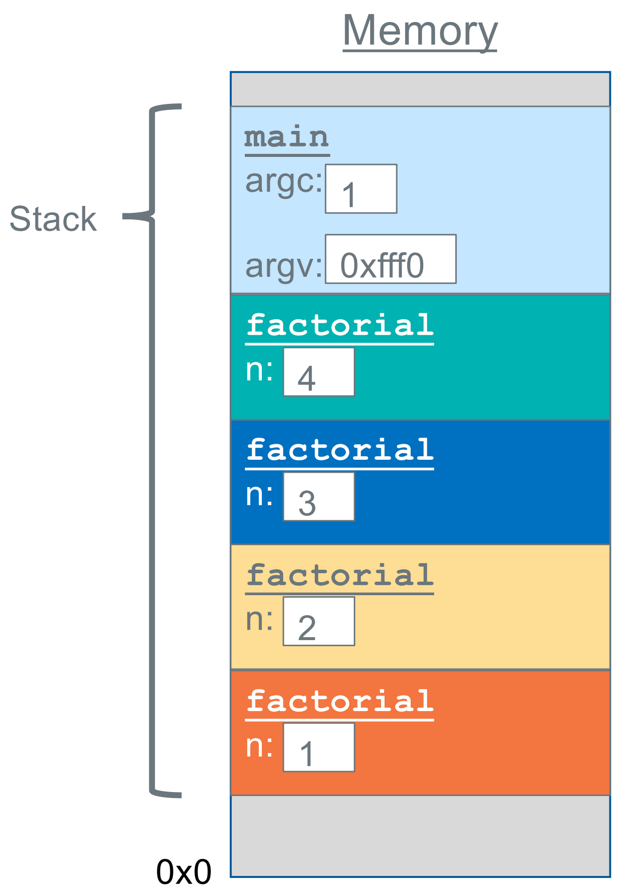
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

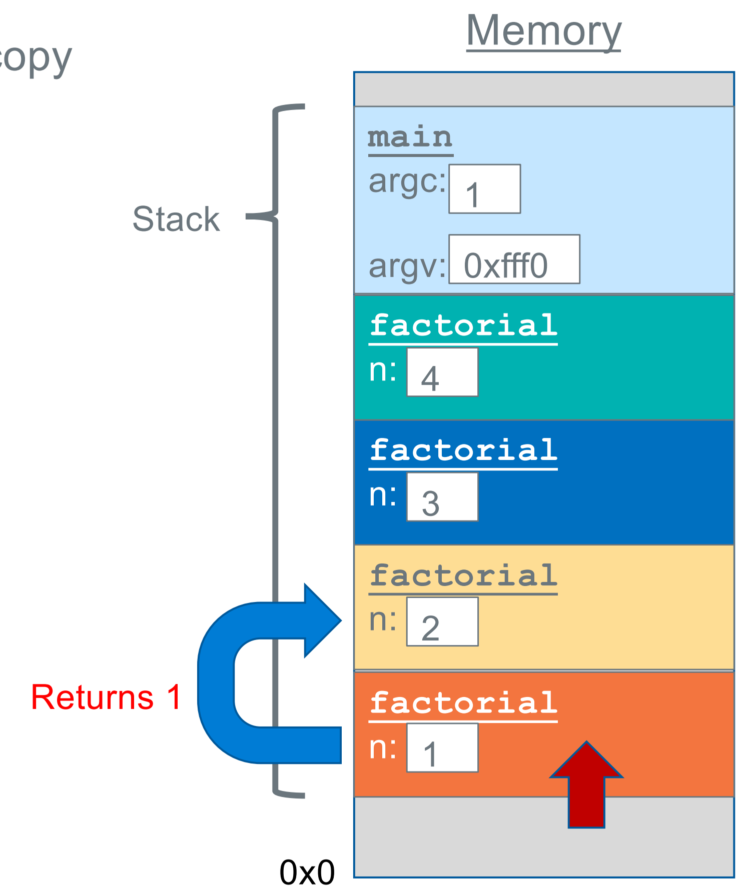
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

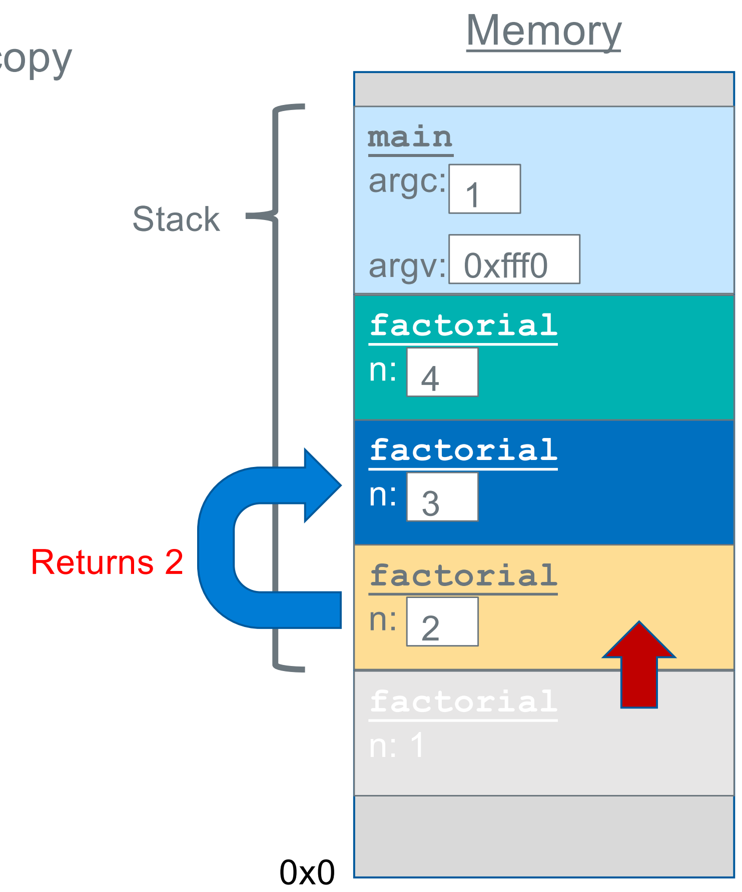
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

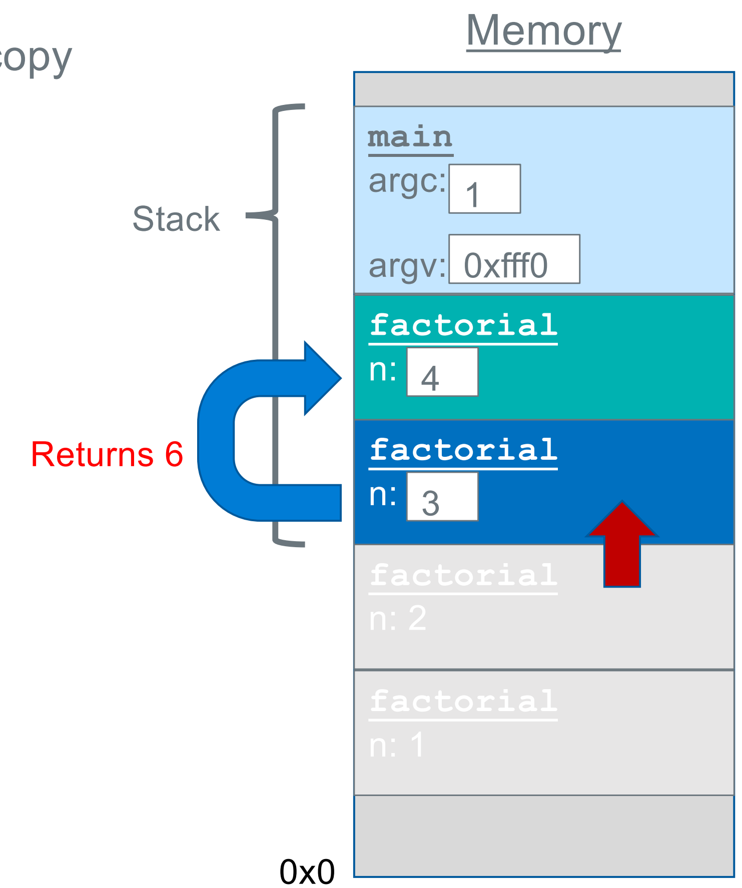
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

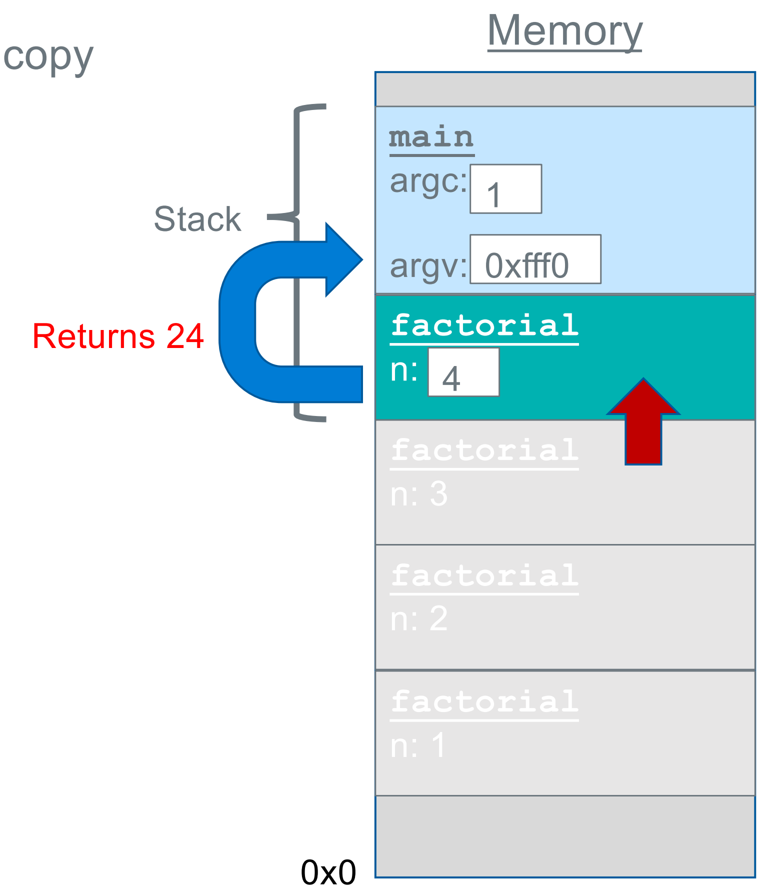
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

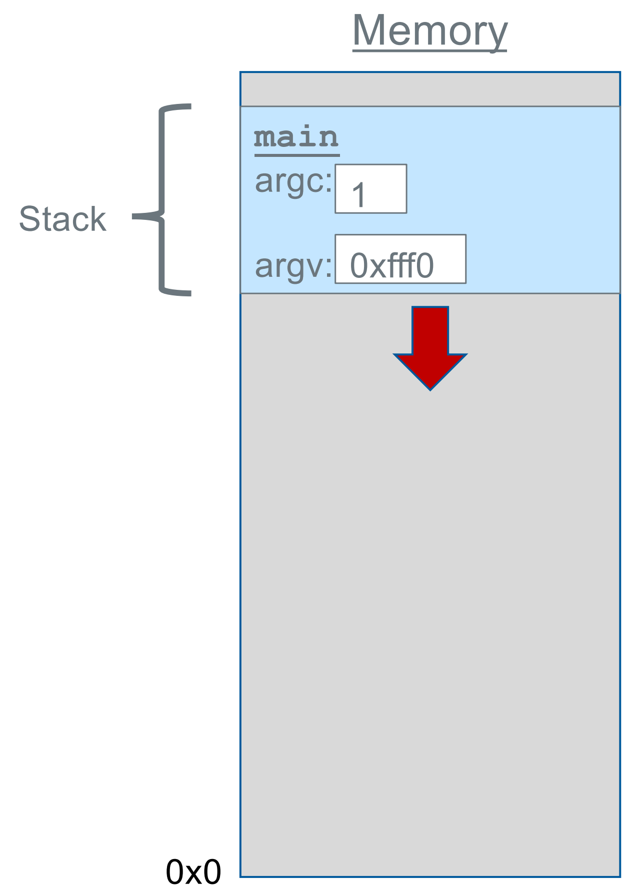
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

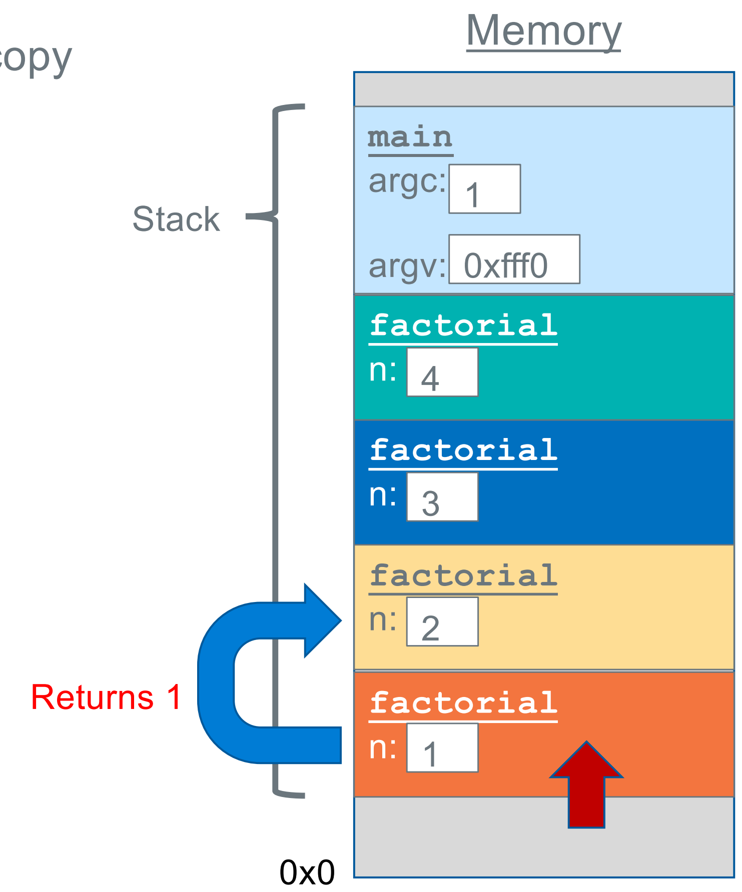
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

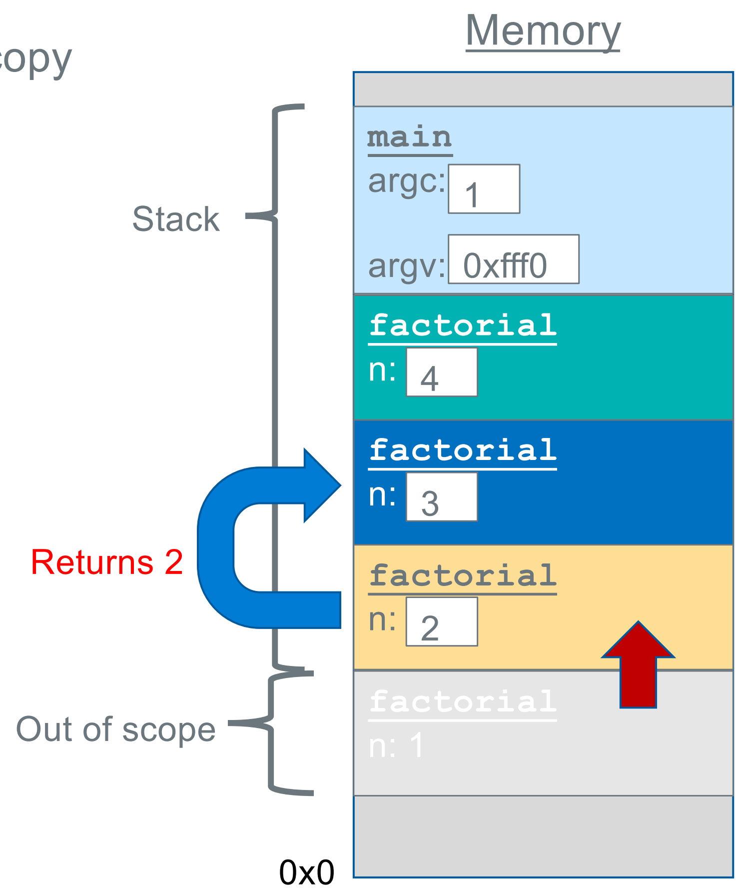
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

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

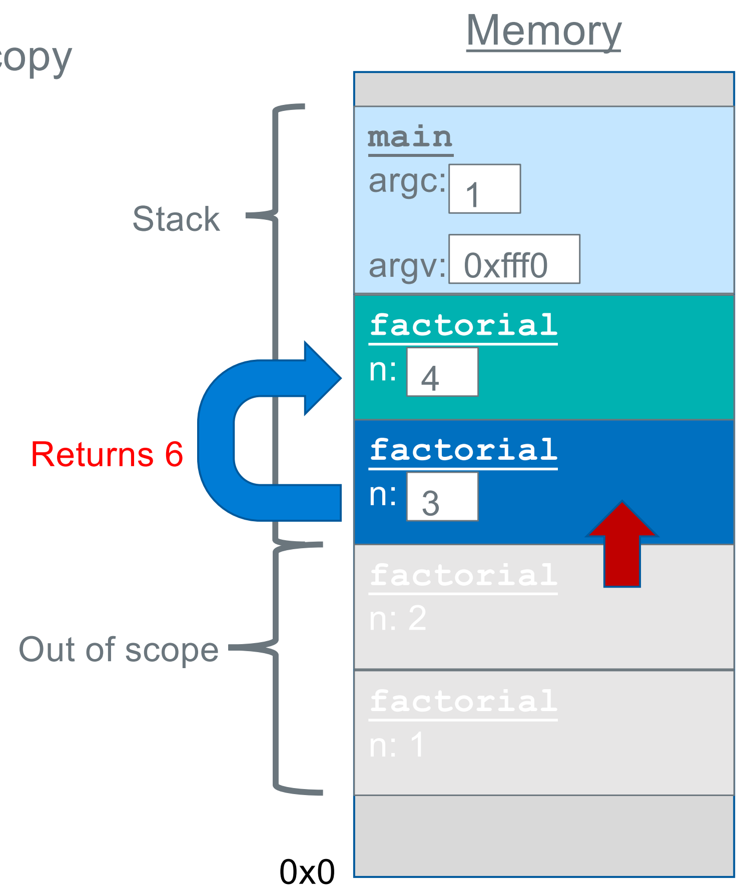
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

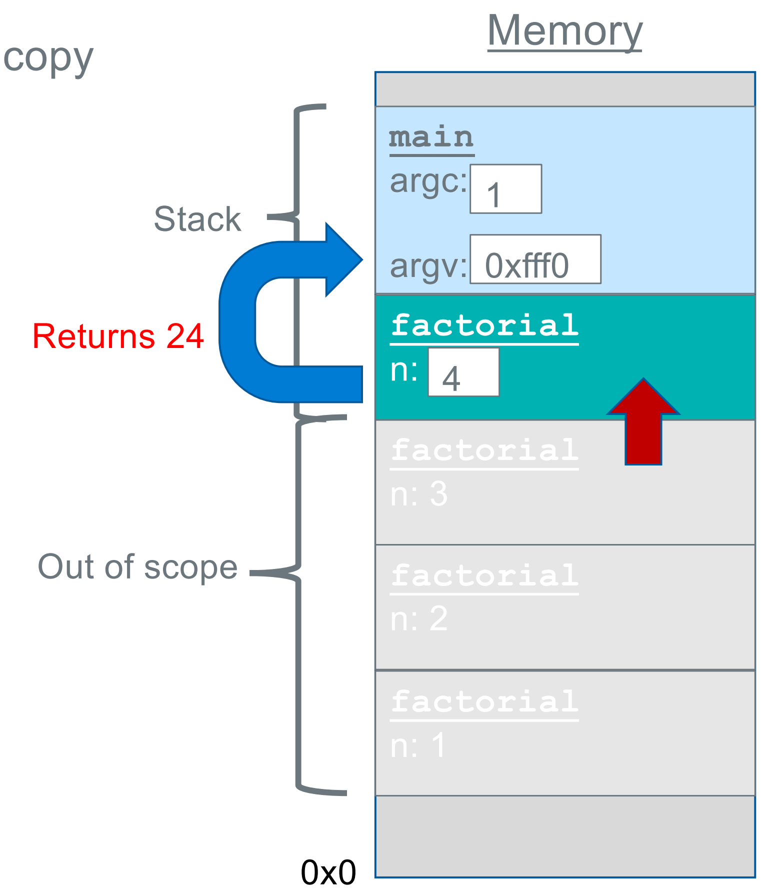
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

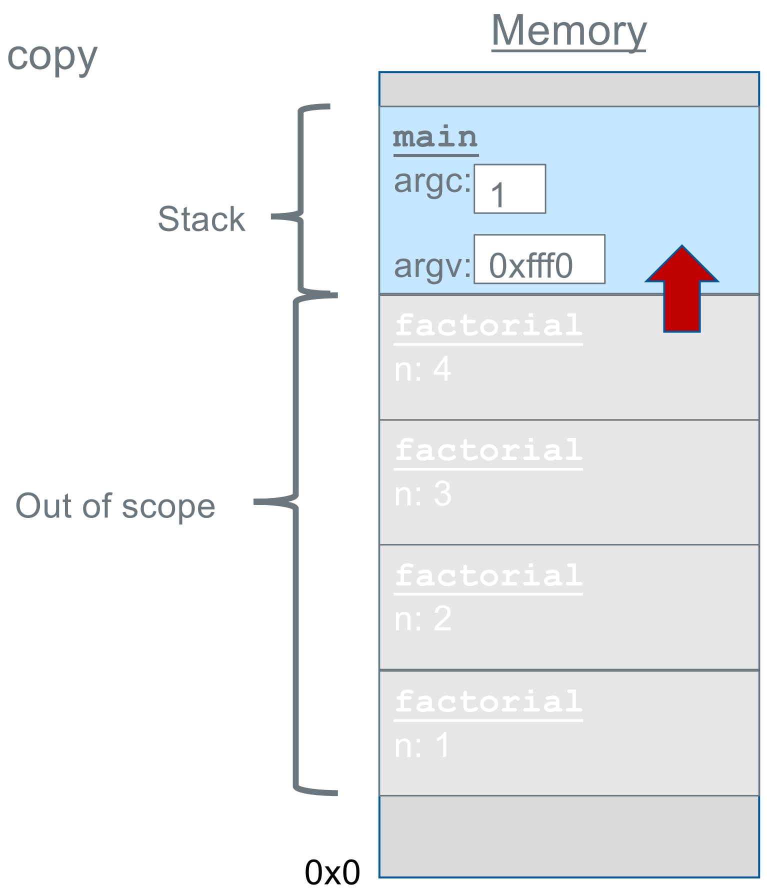
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

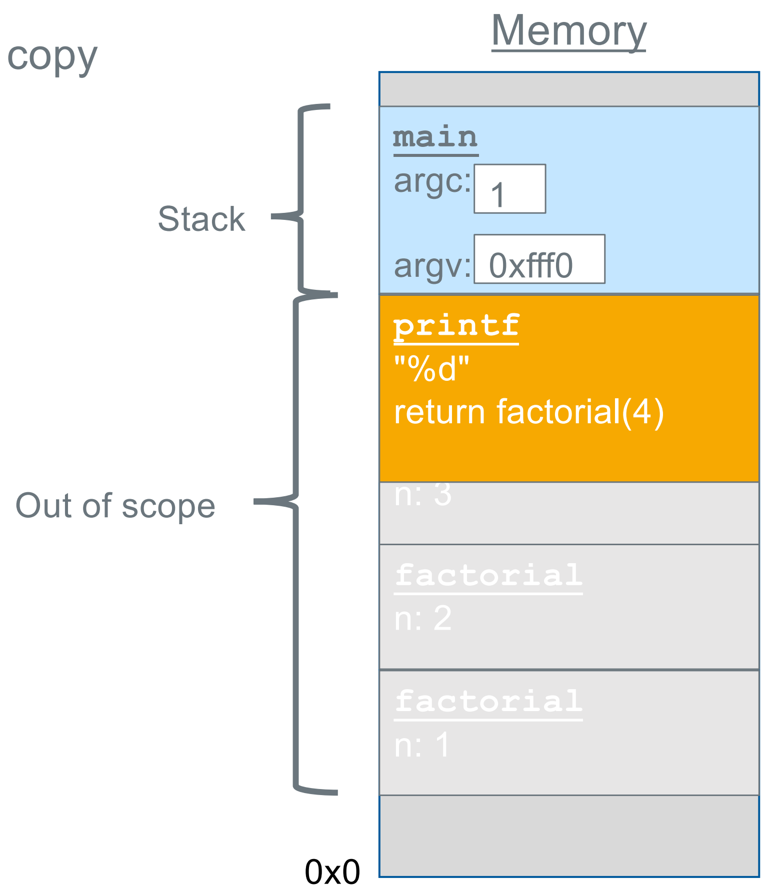
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

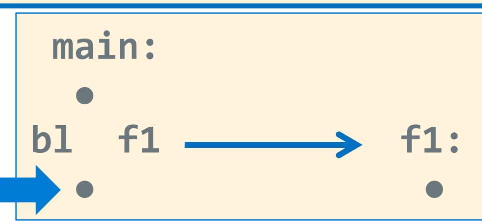
Branch with Link Indirect (function call) instruction

`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



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`

Stores this address in `lr`
this is the address to resume at in the caller

```
main:
 •
 •
bl  f1  ---->  f1:  •
 •  <----           •
 •               bx lr
 •
```

Branch to the instruction whose address is stored in lr

x

bl and bx operation working not together

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

lr: 10404

lr: 10400

```
000103f4 <a>:
   103f4: e3a00000     mov r0, 0
   103f8: e12fff1e     bx lr
000103fc <main>:
   103fc: ebfffffc     bl 103f4 //a
   10400: ebfffffb     bl 103f4 //a
   10404: e3a00000     mov r0, 0
```

lr: 10400

lr: 10404

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

x

bl and bx operation working together

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

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

```
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: 10400

lr: 10400

lr: 1040c

Uh No
Infinite loop!!!

We need to preserve the lr!

x

bl and blx operation not working together

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

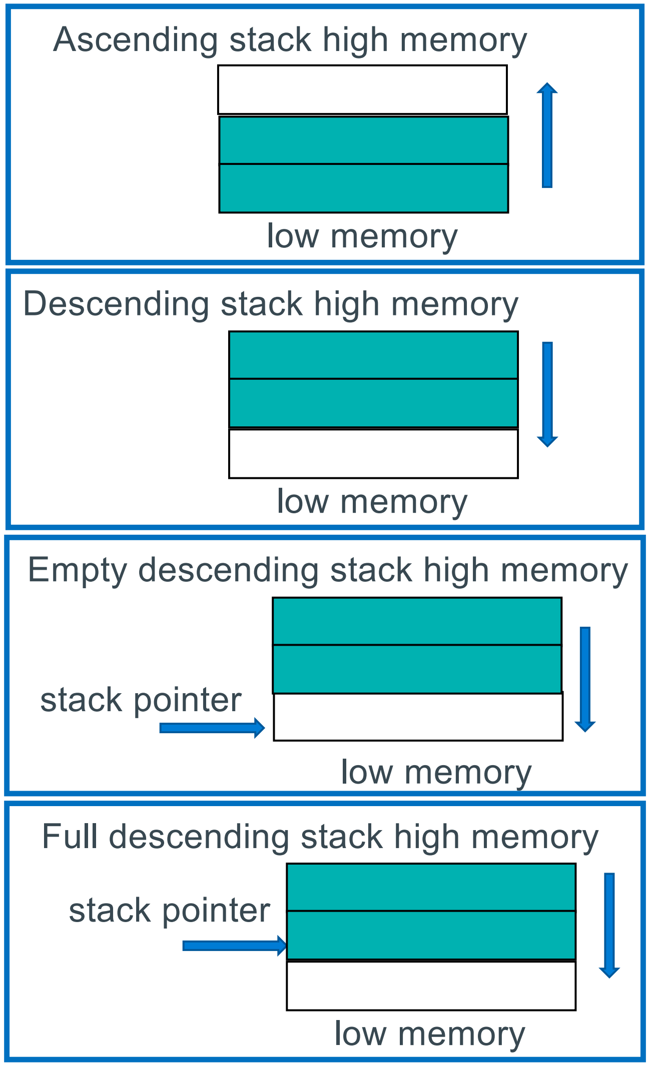
```
      .data
func:    .word a // func initialized with address of a()

     .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

Review: Stack types

- Each time a **function is called**, a **stack frame is activated**
  - space is allocated by moving the stack pointer
- 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**

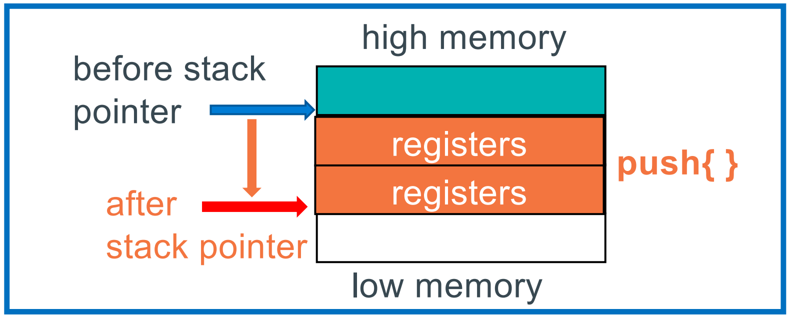


x

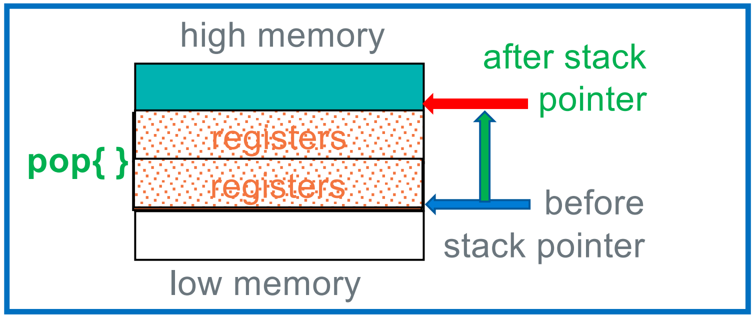
Preserving and Restoring Registers on the stack - 1

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
```

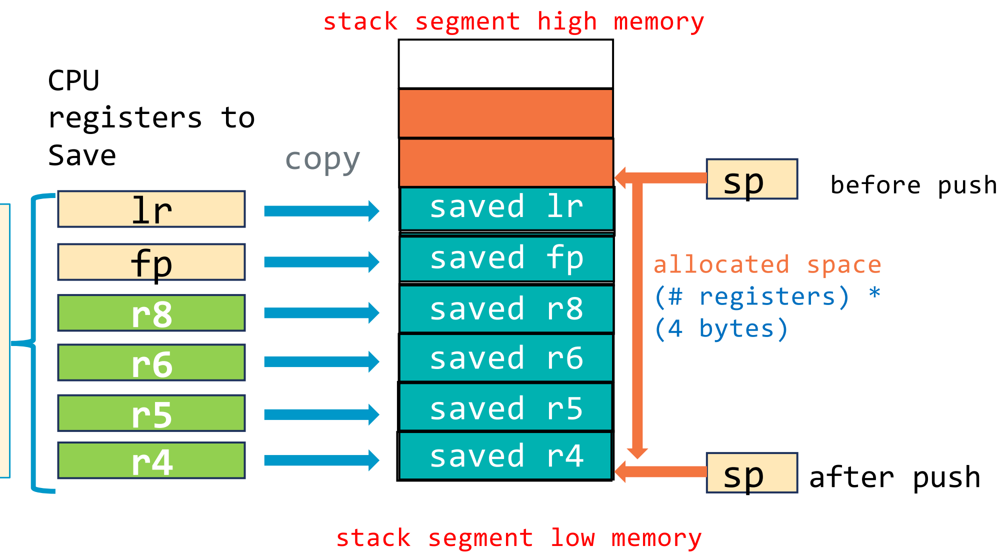
- Registers cannot be:
  1. duplicated in the list
  2. listed out of increasing numeric order (left to right)
- Register ranges can be specified `{r4, r5, r8-r10, fp, lr}`
- **Never!** `push/pop r12, r13, or r15`
  - the top two registers on the stack must always be `fp, lr` // ARM function spec – later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

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 slides) then always push an **EVEN** number of registers!



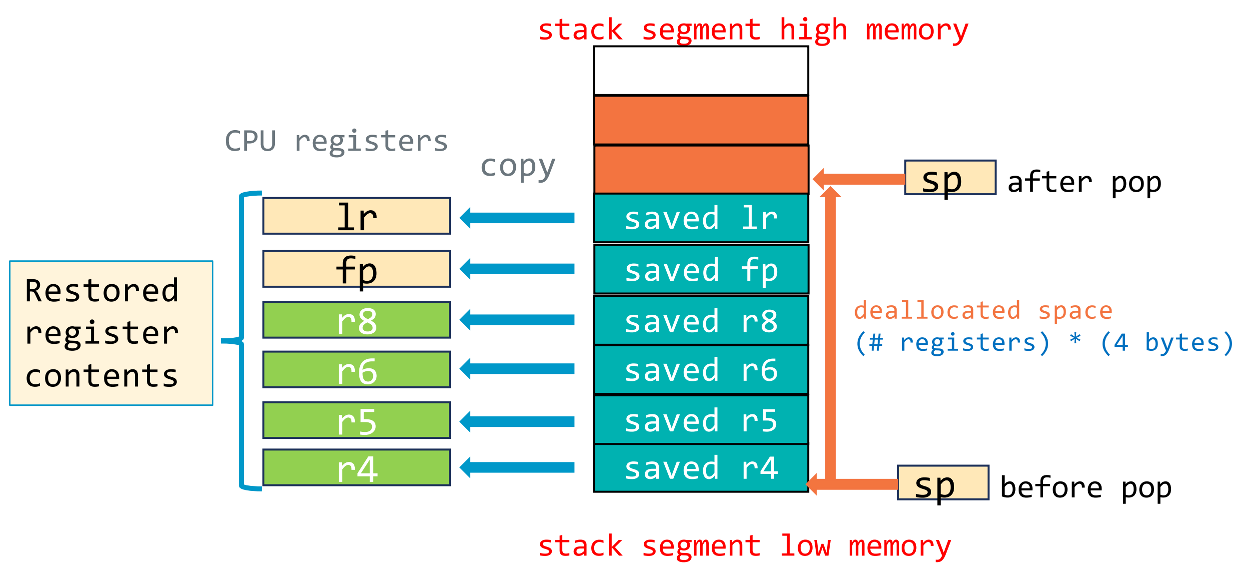
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** <u>subtracts</u> (# of registers saved) * (4 bytes) from the **sp** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

x

pop: Multiple Register Restore (ldr from stack)

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

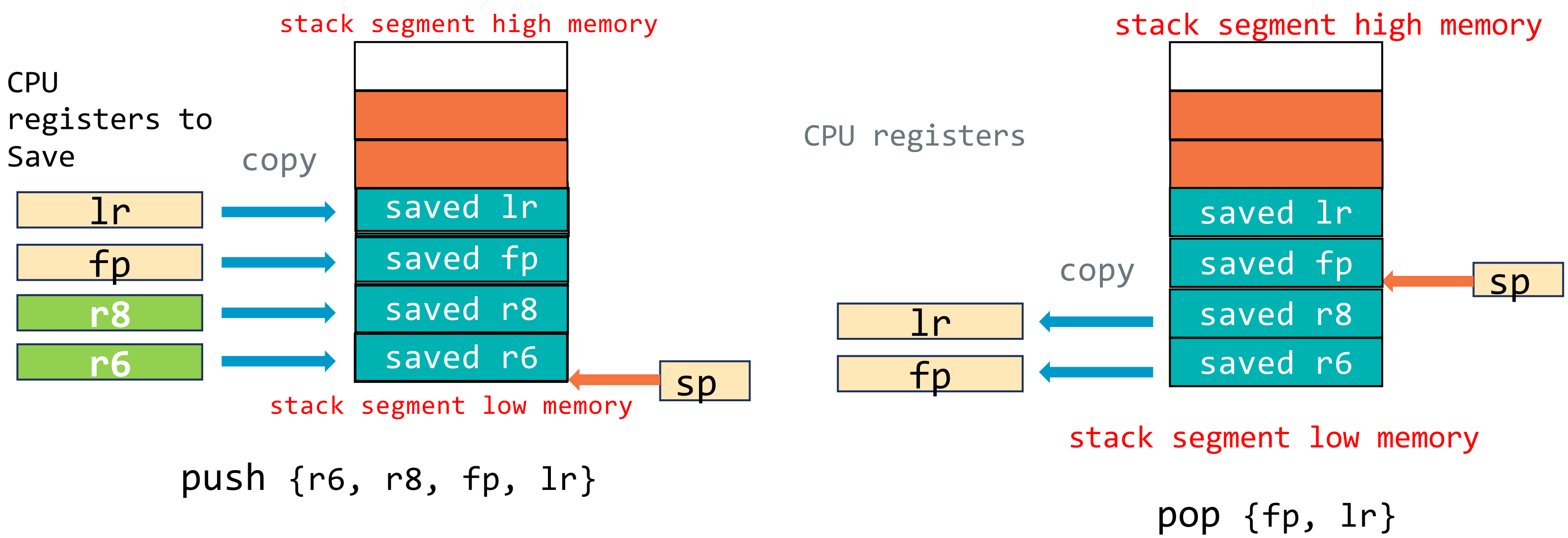
Registers are **pop'd** from the stack *in order* **left (low memory) to right (high memory)**



- **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

push and pop inconsistences

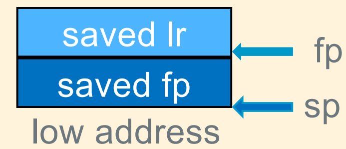


- lr gets an address on the stack, likely segmentation fault

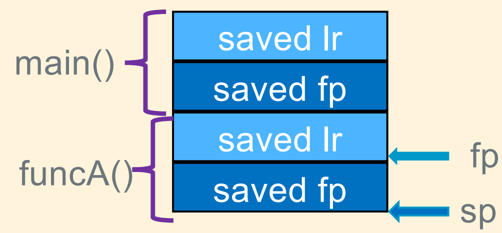
x

Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

- **Minimal frame: allocating at function entry:** `push {fp, lr}`
- `sp` always points at top element in the stack (lowest byte address)
- `fp` always points at the saved `lr` copy stored in the current stack frame
  - 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`)



main() calls funcA()

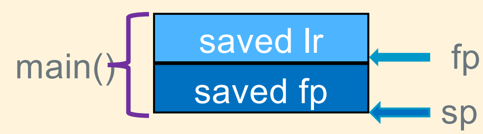


- Function entry (Function **Prologue**):
  1. create (activate) frame
  2. save preserved registers
  3. allocate space for locals

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

- 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"



x

Ghost of Stack Frames Past…..

same stack frame
variable layout

```
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;
}
```

```
% ./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
```

ARM Assembly Source File: Header and Footer

File Header
At the top of every ARM source file

```
        .arch   armv6          // armv6 architecture
        .arm                   // arm 32-bit instruction set
        .fpu    vfp            // floating point co-processor
        .syntax unified        // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

File Footer
At the bottom of every ARM source file

```
.section .note.GNU-stack,"",%progbits // set stack/data non-exec
.end
       // everything past the .end is ignored!
       // Debugging notes etc
```

`.syntax unified`

- use the standard ARM assembly language syntax called ***Unified Assembler Language*** (***UAL)***

`.section .note.GNU-stack,"",%progbits`

- tells the linker to **make the stack and all data segments not-executable** (no instructions in those sections) – security measure

`.end`

- at the end of the source file, everything written after the .end is ignored

x

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
         .size myfunc, (. - myfunc)
```

Function Footer

`.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)**

  `.size name, (. - name)`

X

Example: Assembler Directive and Instructions

assembler directive .equ does not allocate any memory (NULL = 0)

Regular label main is associated with memory location 0x3000

Local label .Lloop is associated with memory location 0x3004

space.S

```
10                            .equu   NULL, 0
11                   main:
12 3000 0310A0E1               mov     r1, r3
13                   .Lloop:
14 3004 043083E2               add     r3, r3, 4
15 3008 001093E5               ldr     r1, [r3]
16 300c 000051E3               cmp     r1, NULL
17 3010 FBFFFF1A               bne     .Lloop
```

Memory Contents
Warning contents shown in *"reverse"* byte order: Lsb – Msb

```
output generated with
gcc -c -Wa,-ahlns space.S
partial output is shown
```

Instruction Memory Addresses (lowest 2-bits are always are 00)
Notice alignment and how addresses increase by 4 (32-bit instructions)

x

Preview: 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**
  - **In terms of C runtime support, these registers contain the copies given to the called function**
  - **C allows the copies to be changed in any way by the called function**

X

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 **local to the file** from the point where they are defined

x

Variable Alignment In Memory and Performance

Accessing **address aligned** memory on many systems based on data type has the best performance (due to hardware implementation)

char

1 any address

short

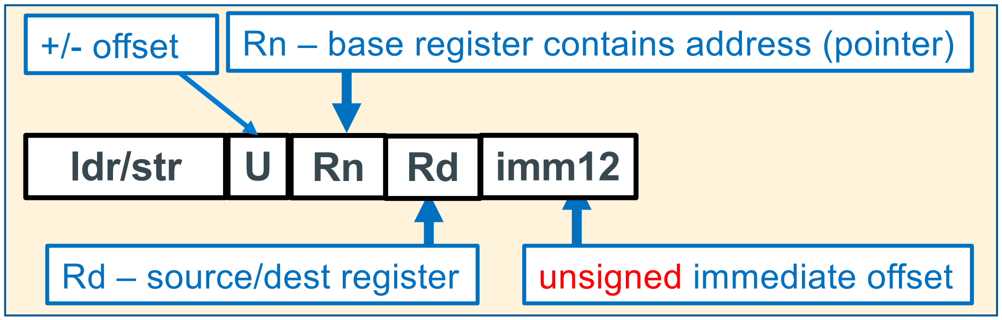
2 bytes addresses that end in 0b0

integer

4 bytes addresses that end in 0b00

x

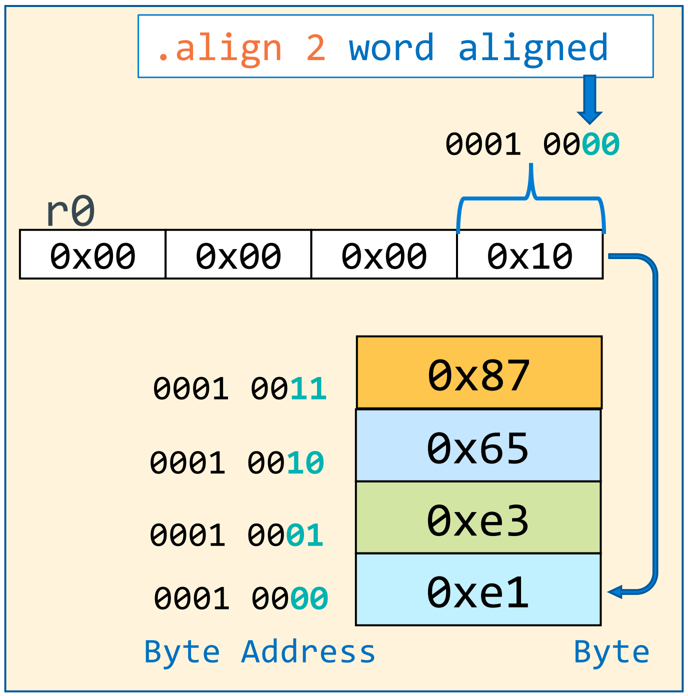
LDR/STR – Base Register + Immediate Offset Addressing

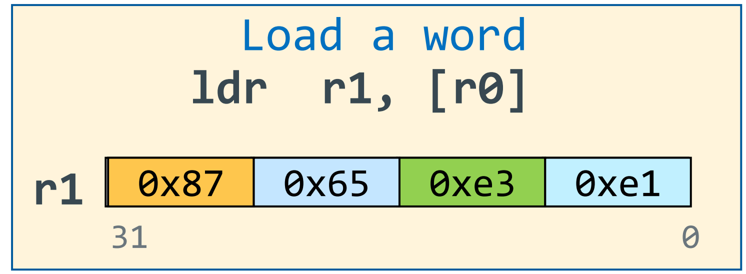


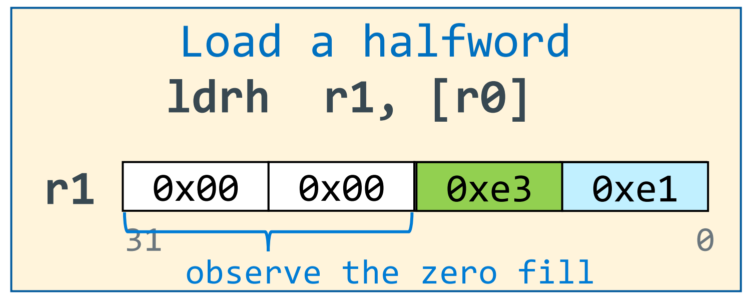
- **Register Base Addressing**:
  - Pointer Address: Rn; source/destination data: Rd
  - **Unsigned pointer address** in stored in the base register
- **Register Base + immediate offset Addressing:**
  - Pointer Address = register content + immediate offset `-4095 <= imm12 <= 4095 (bytes)`
  - Unsigned offset integer immediate value (bytes) is added or subtracted (U bit above says to add or subtract) from the pointer address in the base register

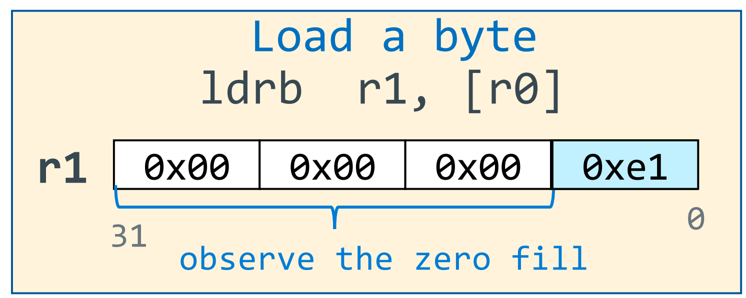
x

Load a Byte, Half-word, Word



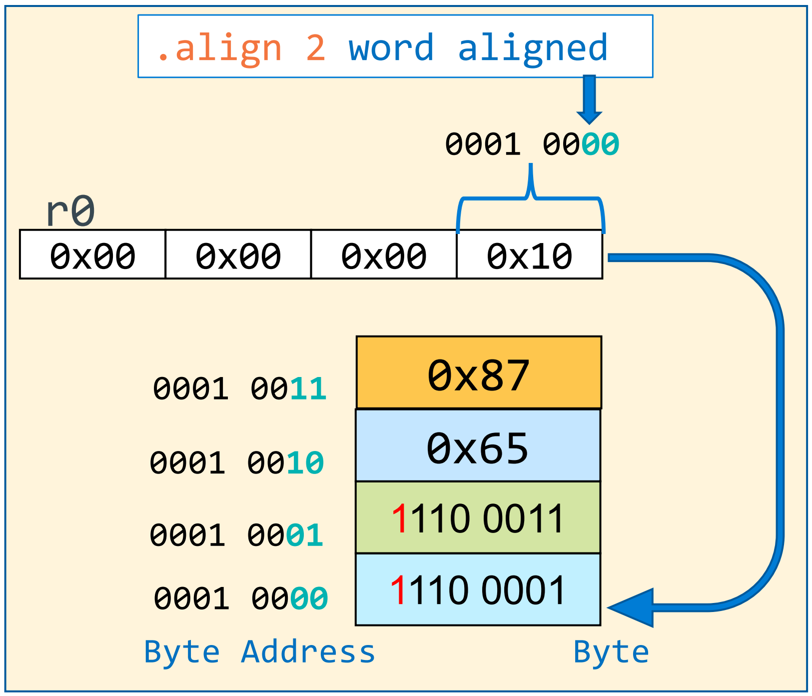


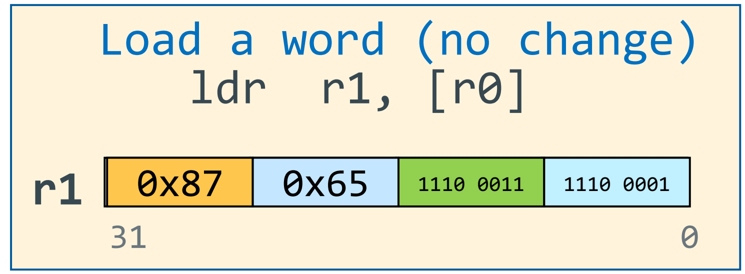


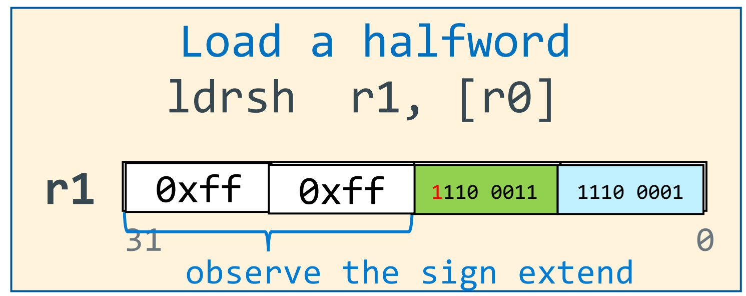


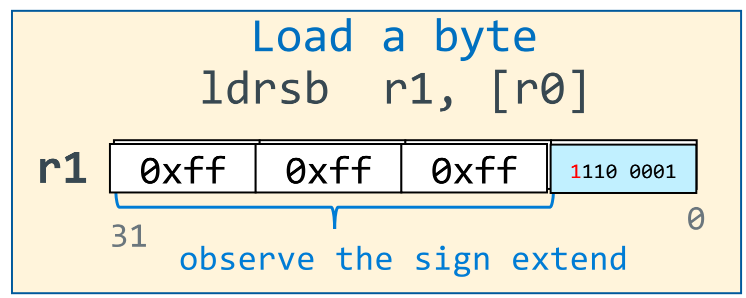
x

Signed Load a Byte, Half-word, Word



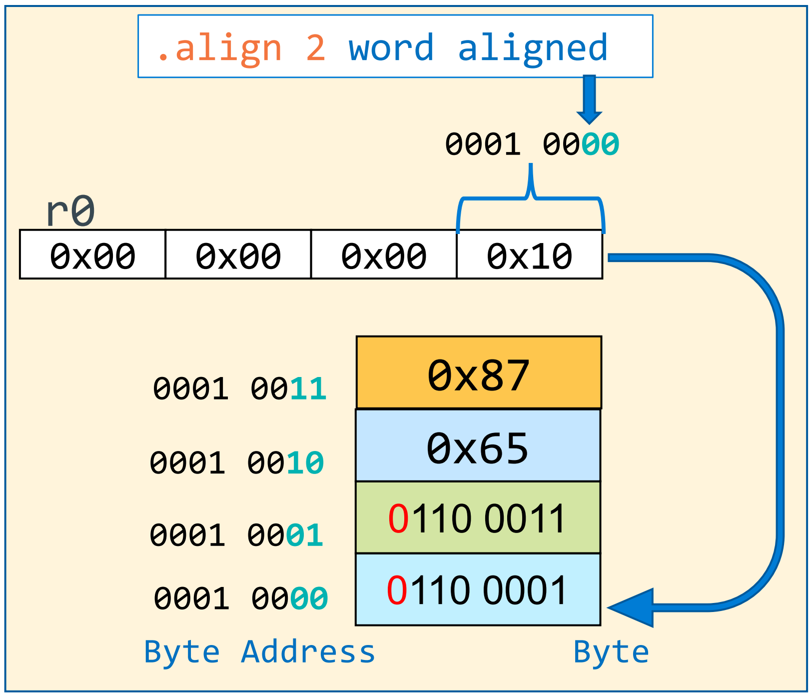


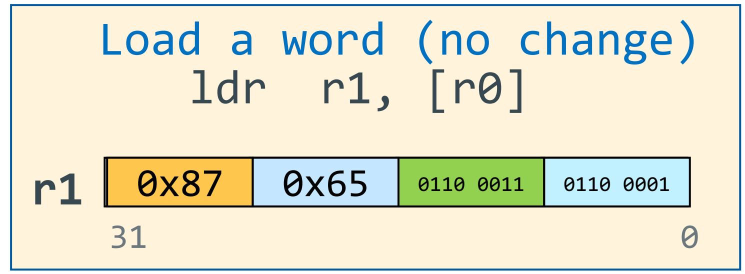


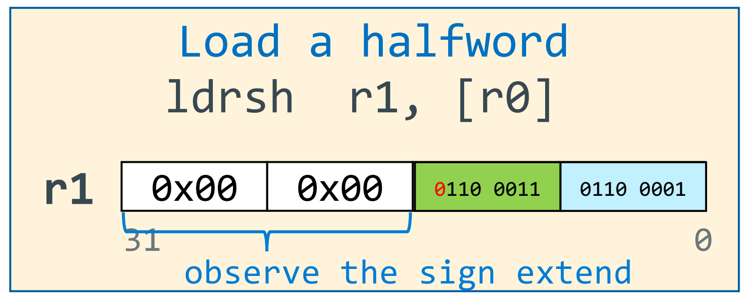


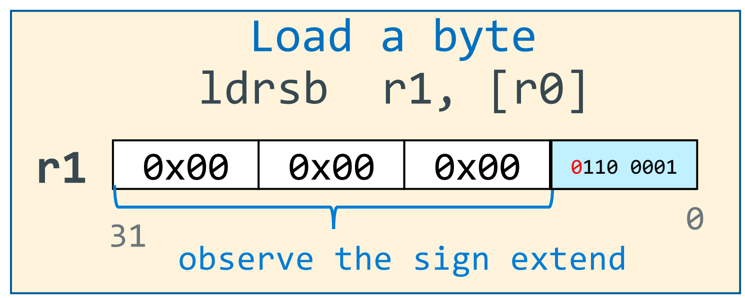
x

Signed Load a Byte, Half-word, Word









x

Store a Byte, Half-word, Word

initial value in r0

0x20	0x00	0x00	0x00

Store a byte

```
strb  r1, [r0]
```

	31			0
r1	0x87	0x65	0xe3	0xe1

Byte Address	Byte
0x20000003	0x33
0x20000002	0x22
0x20000001	0x11
0x20000000	0xe1

observe other bytes NOT altered

Store a halfword

```
strh r1, [r0]
```

	31			0
r1	0x87	0x65	0xe3	0xe1

Byte Address	Byte
0x20000003	0x33
0x20000002	0x22
0x20000001	0xe3
0x20000000	0xe1

Store a word

```
str  r1, [r0]
```

	31			0
r1	0x87	0x65	0xe3	0xe1

Byte Address	Byte
0x20000003	0x87
0x20000002	0x65
0x20000001	0xe3
0x20000000	0xe1

x

Base Register Addressing + Offset register

```
#include <stdio.h>
#include <stdlib.h>
int count(char *, int);
int main(void)
{
    char msg[] ="Hello CSE30! We Are CountinG UpPER cASe letters!";

    printf("%d\n", count(msg, sizeof(msg)/sizeof(*msg)));
    return EXIT_SUCCESS;
}
```

```
int count(char *ptr, int len)
{
    int cnt = 0;
    int i;

    for (i = 0; i < len; i++) {
        if ((ptr[i] >= 'A') && (ptr[i] <= 'Z'))
            cnt++;
   }
    return cnt;
}
```


x

Base Register + Offset register

```
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .text
    .global count
    .type   count, %function
    .equ    FP_OFF, 12
    // r0 contains char *ptr
    // r1 contains int len
    // r2 contains int cnt
    // r3 contains int i
    // r4 contains char

count:
    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 count, (. - count)
    .end
```

byte array
Also use ldrb here
offsets are 0,1,2,…

```
count:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF

    mov     r2, 0
    cmp     r1, 0
    ble     .Ldone
    mov     r3, 0
.Lfor:
    cmp     r3, r1
    bge     .Ldone

    ldrb    r4, [r0, r3]
    cmp     r4, 'A'
    blt     .Lendif
    cmp     r4, 'Z'
    bgt     .Lendif
    add     r2, r2, 1

.Lendif:
    add     r3, r3, 1
    b       .Lfor

.Ldone:
    mov     r0, r2
```

loop guard

x