Version 1.03

UCSD CSE 30

Computer Organization and Systems Programming

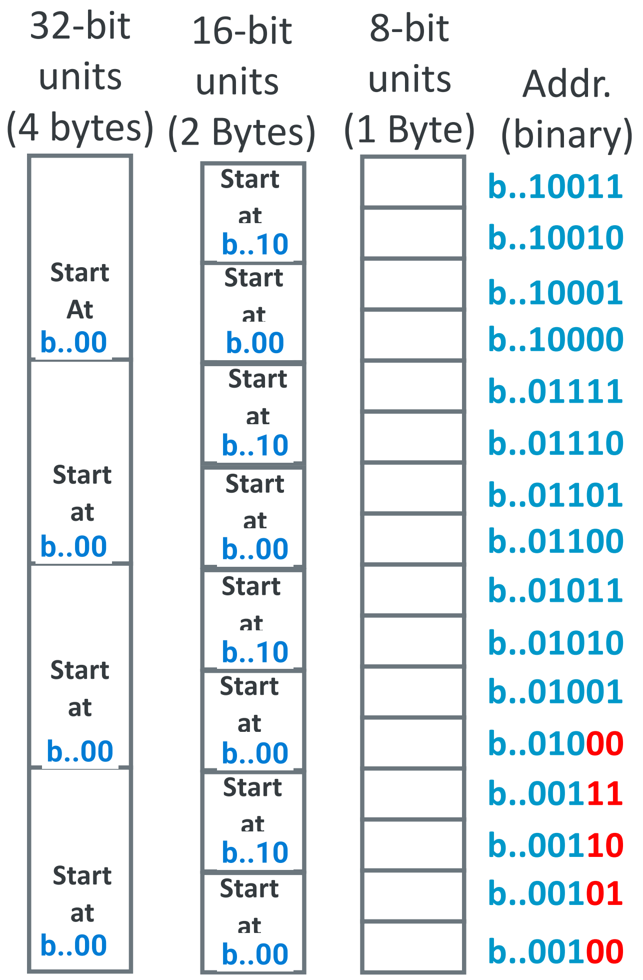
Aarch32 Assembly – Part 3

Lecture 17 – November 15, 2022

Keith Muller

Memory Data Alignment

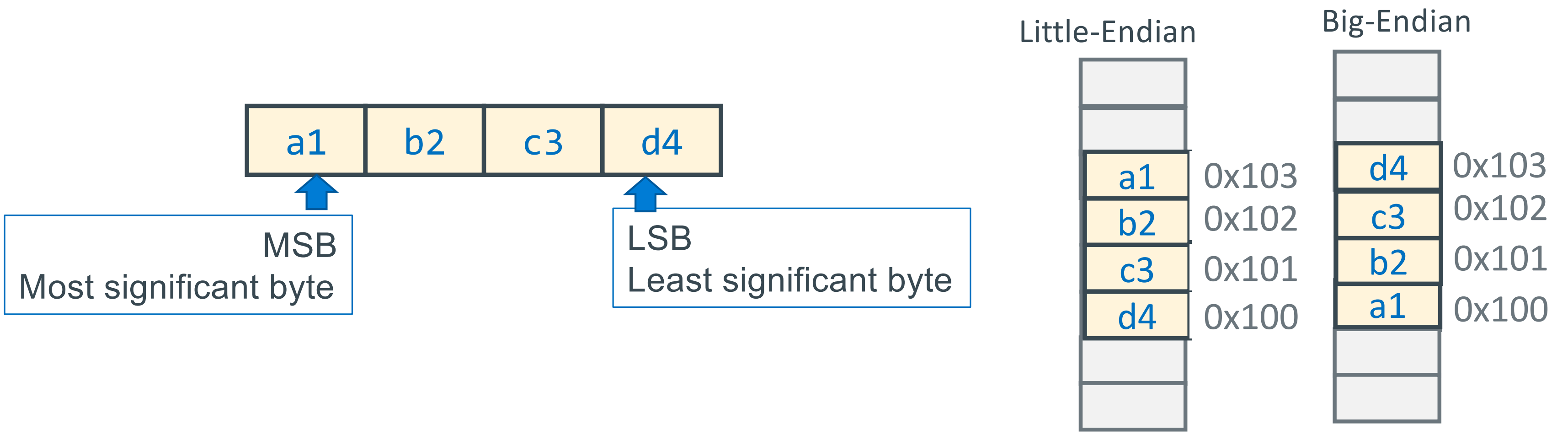
- **Word** is the number of bytes necessary to store an address (32-bits on Pi-cluster) – hardware defined
- The address of *any sized* unit of memory is always the address of the **first byte**
- Hardware often requires Variables to be *"aligned"* to specific starting addresses based on type
- `char (1 byte)`
  - can start at any address
- `short (2 bytes)` start only at addresses ending in
  - b..00 or b..10 (.align 1) // last **bit** must be 0
- `int (4 bytes)` can start only at addresses ending in
  - 0b..00 (.align 2) // last **two bits** must be 0



x

Byte Ordering of Numbers In Memory: Endianness

- Two different ways to place multi-byte integers in a byte addressable memory
- Big-endian: Most Significant Byte (“big end”) starts at the *lowest (starting)* address
- Little-endian: Least Significant Byte (“little end”) starts at the *lowest (starting)* address
- Example: 32-bit integer with 4-byte data



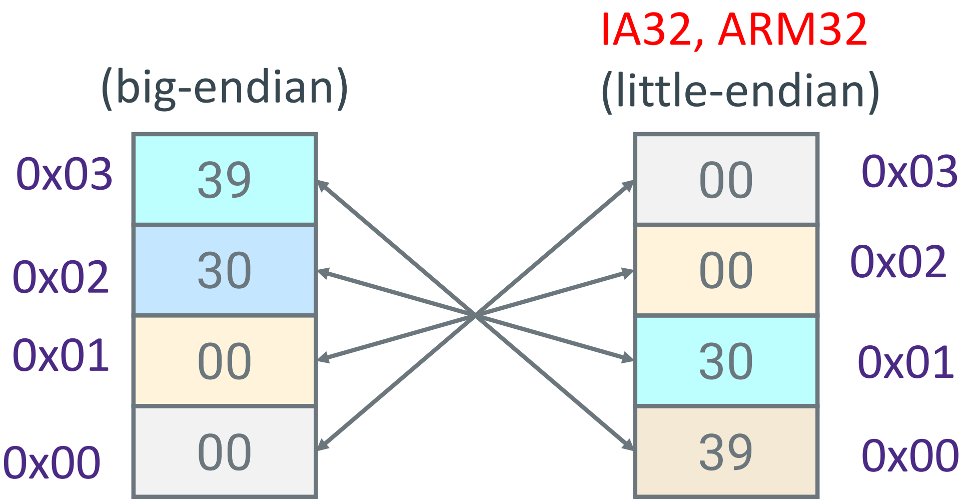
MSB
Most significant byte

LSB
Least significant byte

Byte Ordering Example

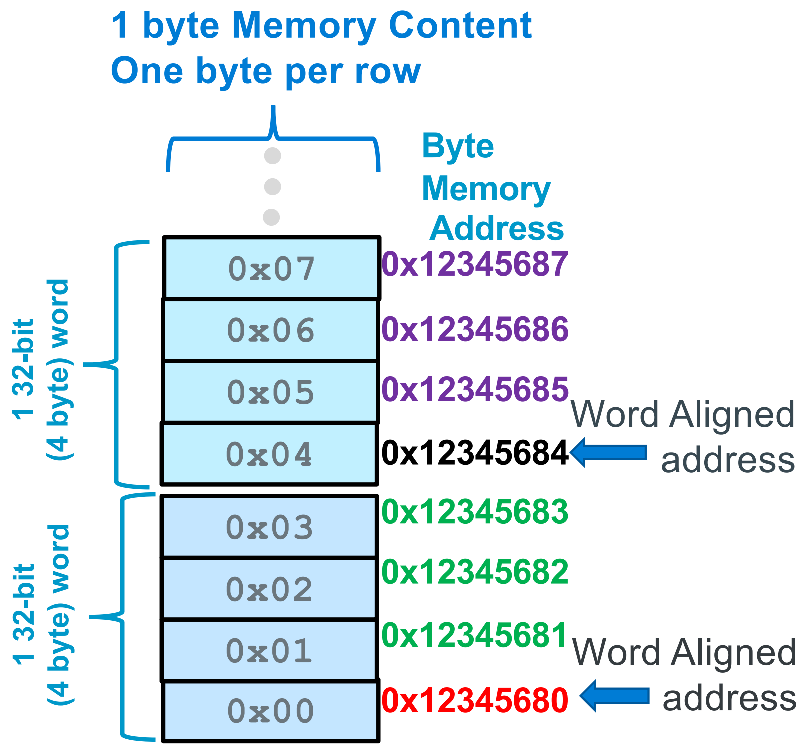
```
Decimal:   12345
Binary:     0011   0000   0011   1001
Hex:           3      0      3      9
```

```
int x = 12345;
// or x = 0x00003039;  // show all 32 bits
```



x

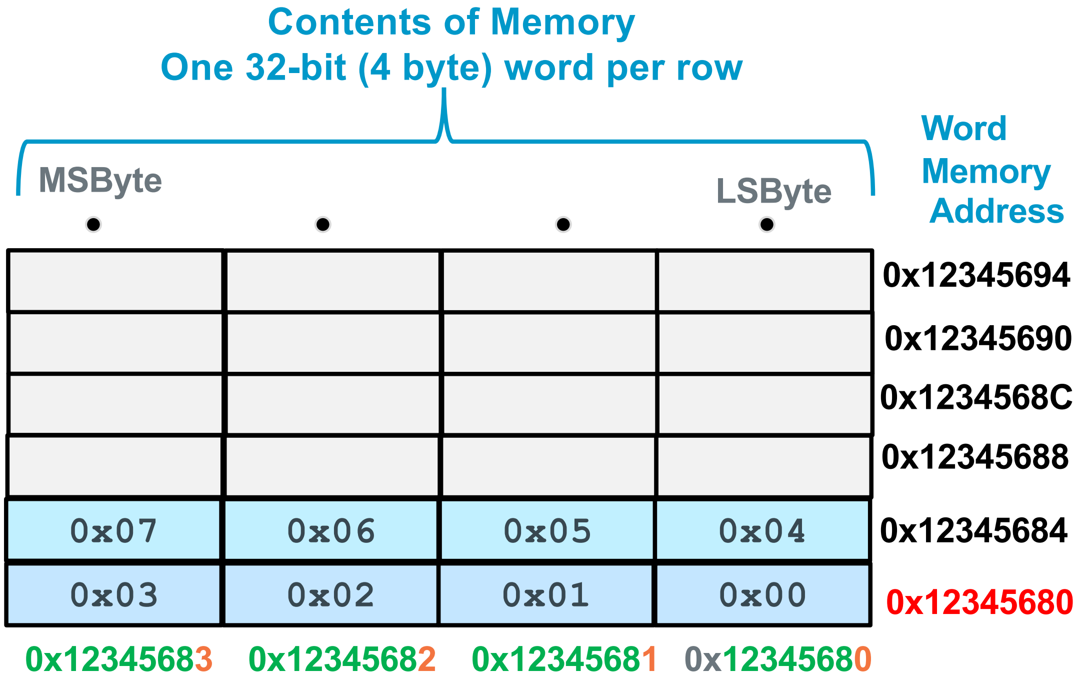
Byte Addressable Memory Shown as 32-bit words



1 byte Memory Content
One byte per row

Byte
Memory
Address

0x07	0x12345687
0x06	0x12345686
0x05	0x12345685
0x04	0x12345684
0x03	0x12345683
0x02	0x12345682
0x01	0x12345681
0x00	0x12345680



Contents of Memory
One 32-bit (4 byte) word per row

MSByte			LSByte	Word Memory Address
				0x12345694
				0x12345690
				0x1234568C
				0x12345688
0x07	0x06	0x05	0x04	0x12345684
0x03	0x02	0x01	0x00	0x12345680
0x12345683	0x12345682	0x12345681	0x12345680	

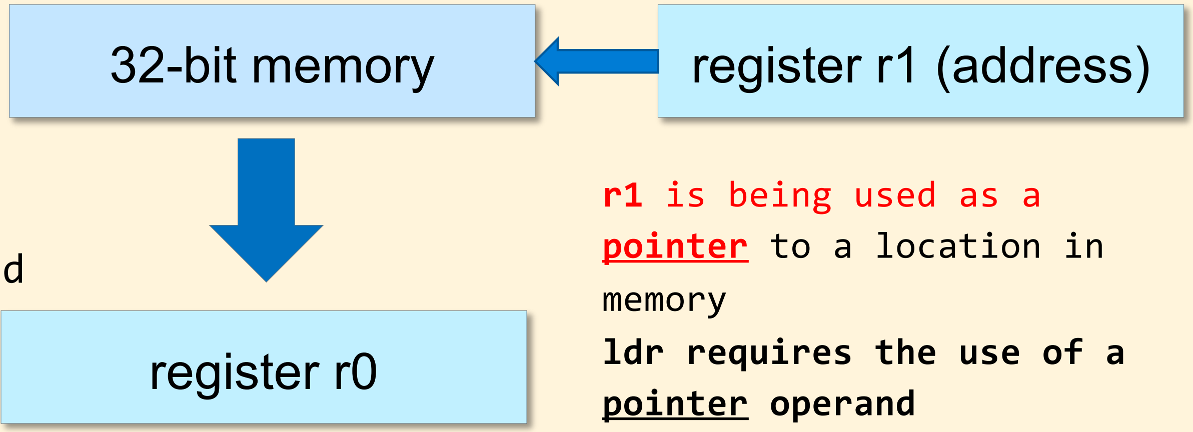
Byte address

Observation
32-bit aligned addresses
rightmost 2 bits of the address are always 0

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

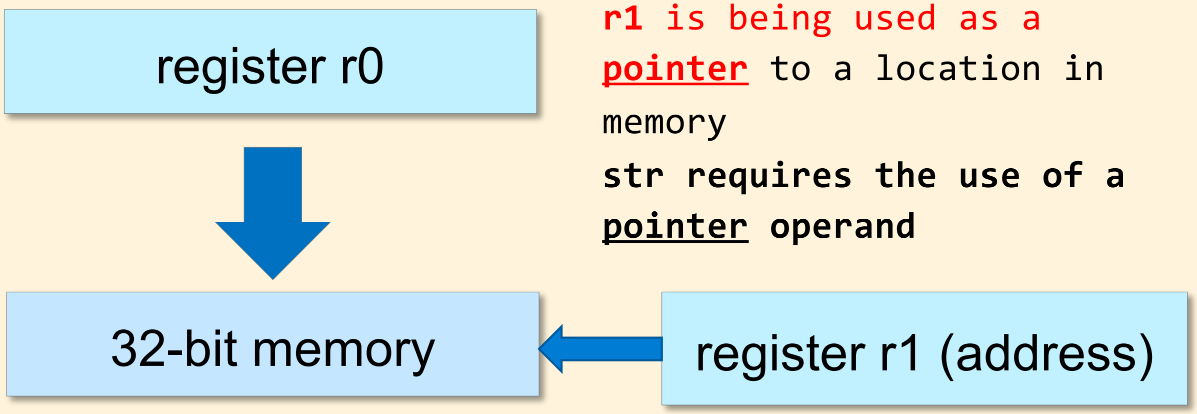


r1 is being used as a **pointer** to a location in memory

ldr requires the use of a pointer operand

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

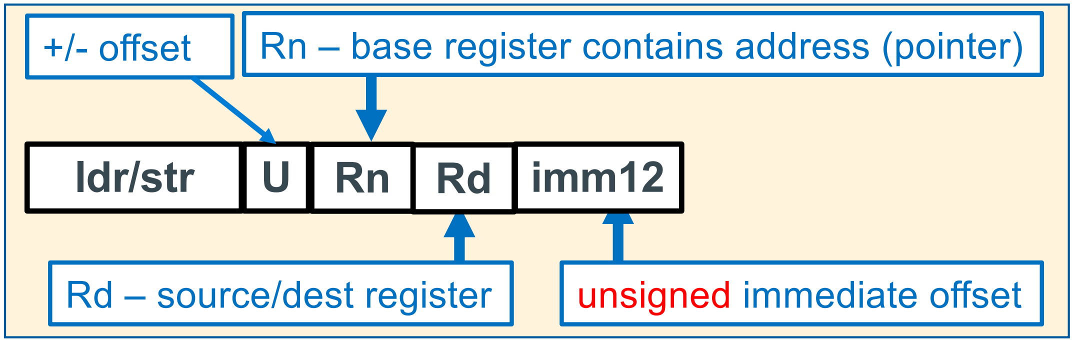
Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)



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

str requires the use of a pointer operand

LDR/STR – Base Register + Immediate Offset Addressing



+/- offset

Rn – base register contains address (pointer)

ldr/str | U | Rn | Rd | imm12

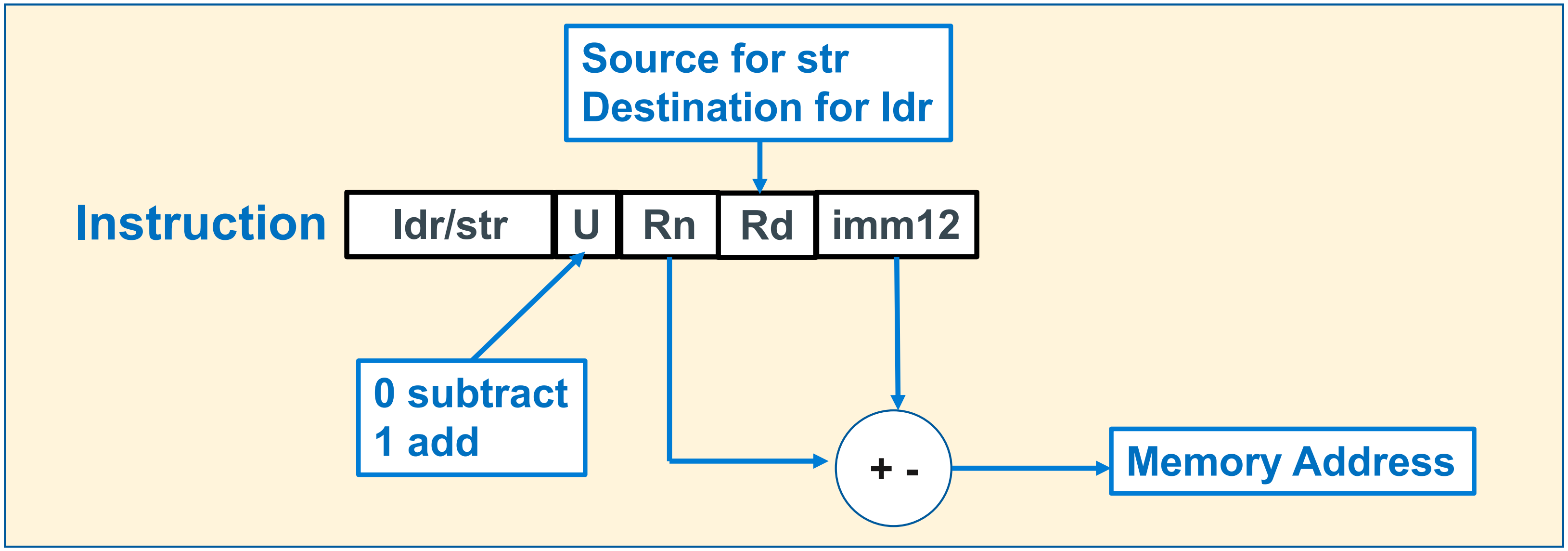
Rd – source/dest register

unsigned immediate offset

- **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
  - 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

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

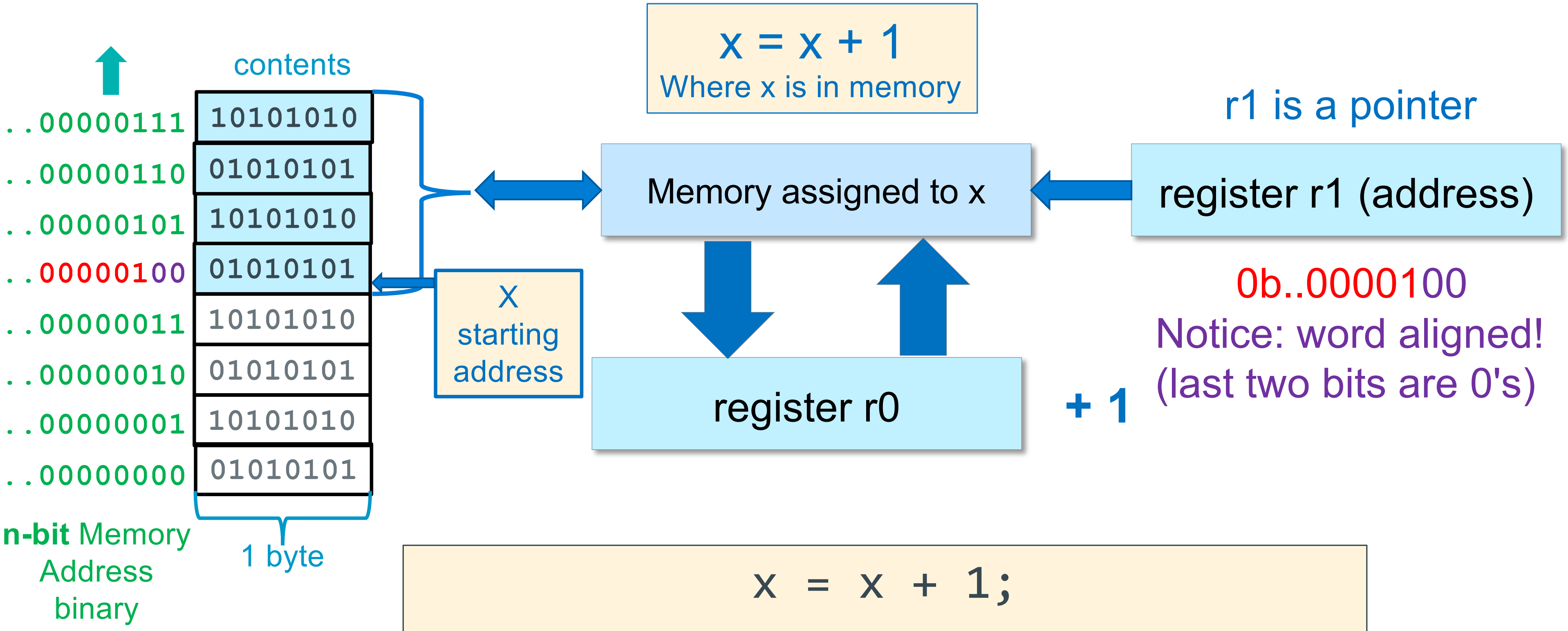
ldr/str Register Base and Register + Immediate Offset Addressing



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

x

Example Base Register Addressing Load – Modify – Store



x = x + 1
Where x is in memory

n-bit Memory Address binary	contents
..00000111	10101010
..00000110	01010101
..00000101	10101010
..00000100	01010101
..00000011	10101010
..00000010	01010101
..00000001	10101010
..00000000	01010101

r1 is a pointer

0b..0000100
Notice: word aligned!
(last two bits are 0's)

```
              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

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)

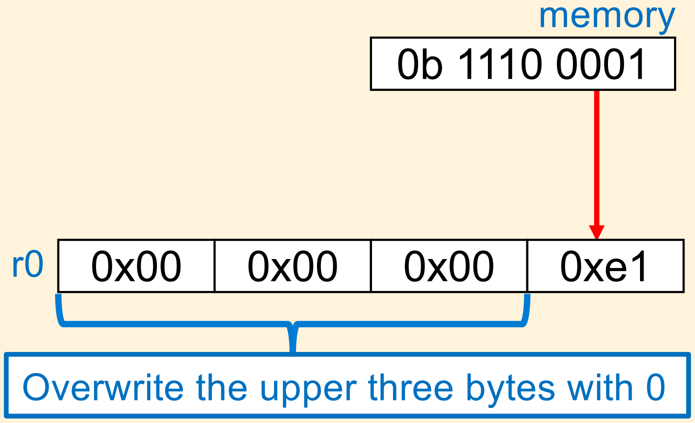
x

Loading 32-bit Registers From Memory Variables < 32-Bits Wide

Unsigned

Zero-Extend: Add leading 0's

example `ldrb`



memory

0b 1110 0001

r0

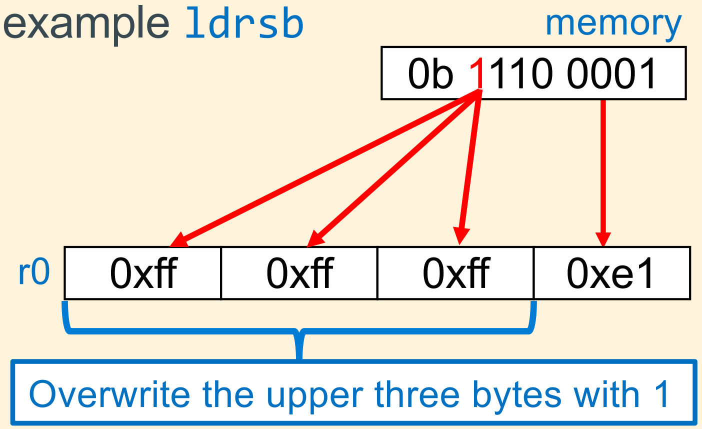
0x00	0x00	0x00	0xe1

Overwrite the upper three bytes with 0

Signed (2's complement)

Sign-Extend: Replicate sign bit

example `ldrsb`



memory

0b 1110 0001

r0

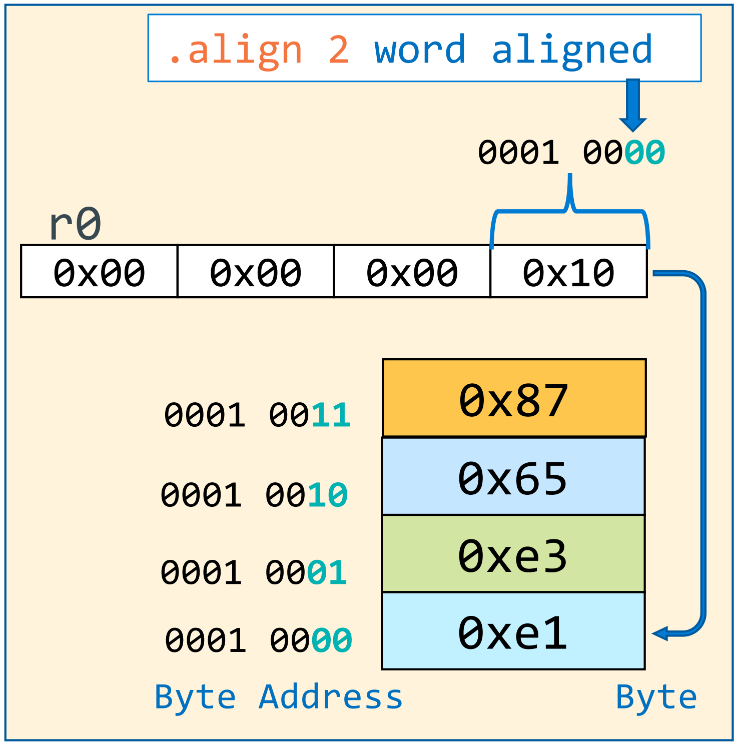
0xff	0xff	0xff	0xe1

Overwrite the upper three bytes with 1

Instructions that zero-extend:
ldrb, ldrh

Instructions that sign-extend:
ldrsb, ldrsh

Load a Byte, Half-word, Word



.align 2 word aligned

0001 0000

r0

0x00	0x00	0x00	0x10

Byte Address	Byte
0001 0011	0x87
0001 0010	0x65
0001 0001	0xe3
0001 0000	0xe1

Load a word

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

r1

0x87	0x65	0xe3	0xe1
31			0

Load a halfword

```
ldrh  r1, [r0]
```

r1

0x00	0x00	0xe3	0xe1
31			0

observe the zero fill

Load a byte

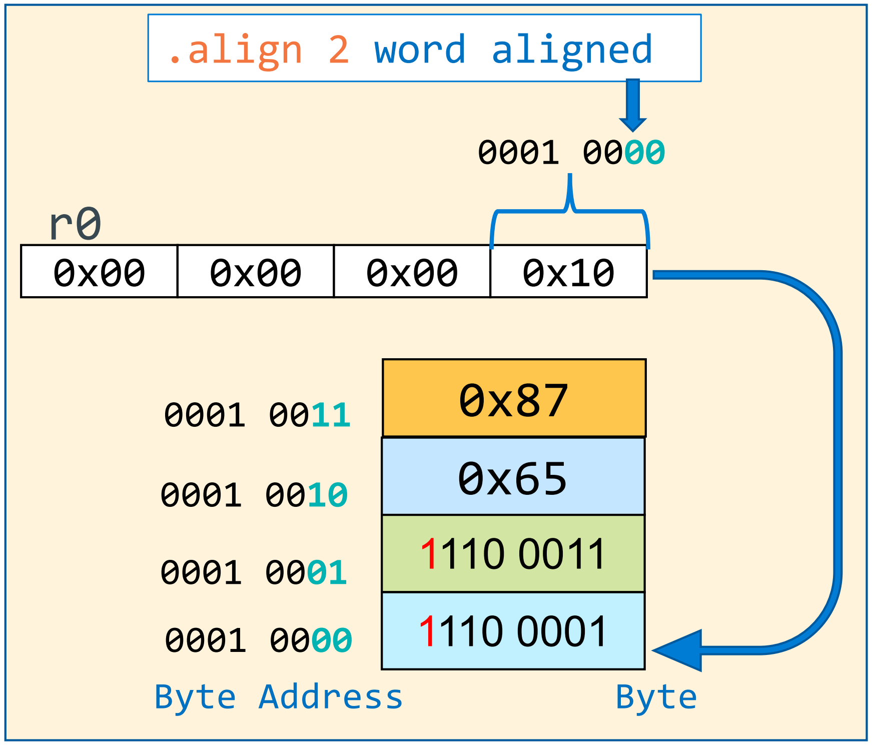
```
ldrb  r1, [r0]
```

r1

0x00	0x00	0x00	0xe1
31			0

observe the zero fill

Signed Load a Byte, Half-word, Word



.align 2 word aligned

0001 0000

r0

0x00 0x00 0x00 0x10

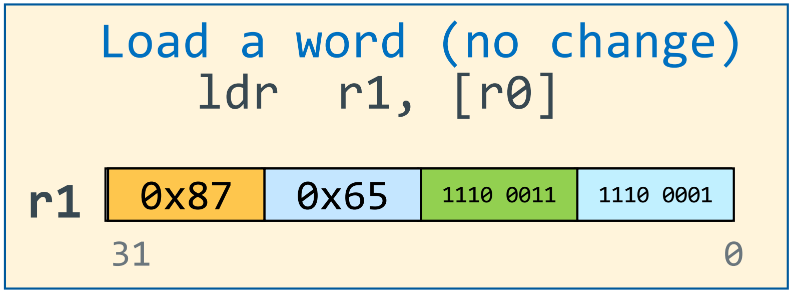
0001 0011 0x87

0001 0010 0x65

0001 0001 1110 0011

0001 0000 1110 0001

Byte Address Byte

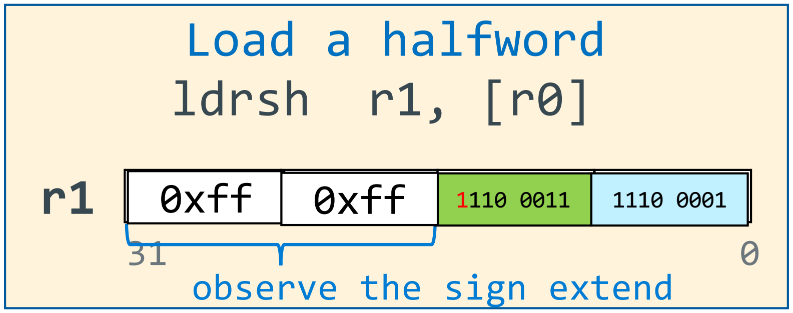


Load a word (no change)

`ldr r1, [r0]`

r1 0x87 0x65 1110 0011 1110 0001

31 0



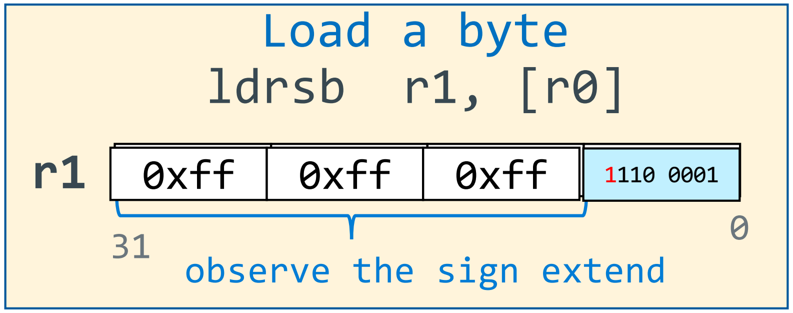
Load a halfword

`ldrsh r1, [r0]`

r1 0xff 0xff 1110 0011 1110 0001

31 0

observe the sign extend



Load a byte

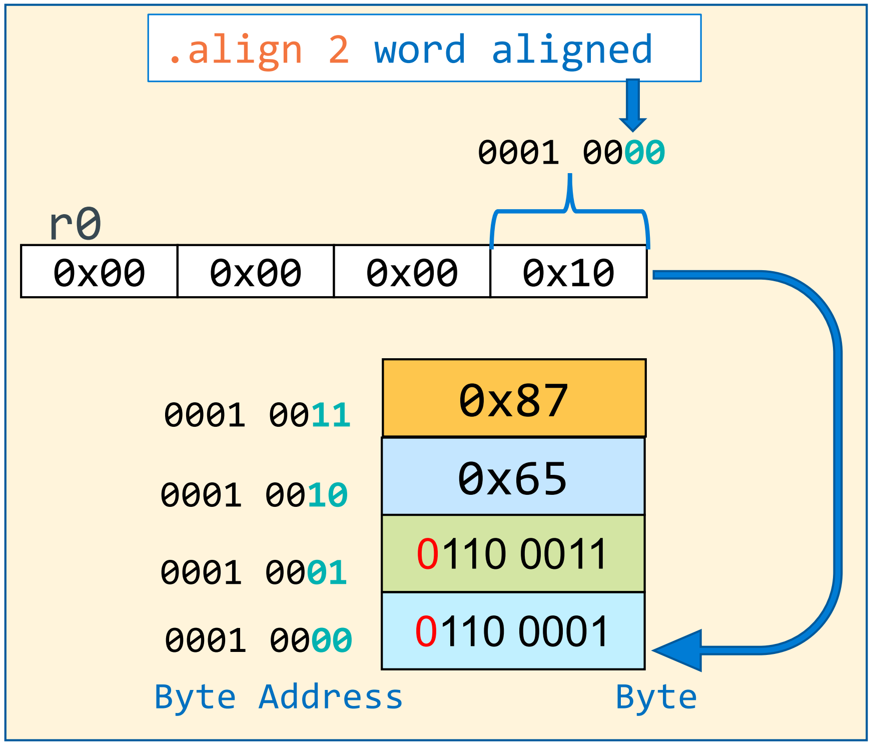
`ldrsb r1, [r0]`

r1 0xff 0xff 0xff 1110 0001

31 0

observe the sign extend

Signed Load a Byte, Half-word, Word



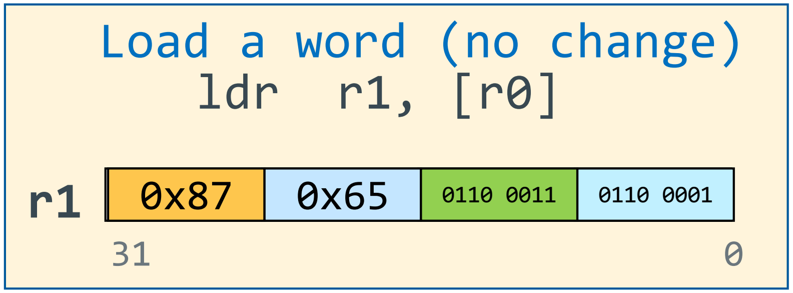
.align 2 word aligned

0001 0000

r0

0x00	0x00	0x00	0x10

Byte Address	Byte
0001 0011	0x87
0001 0010	0x65
0001 0001	0110 0011
0001 0000	0110 0001



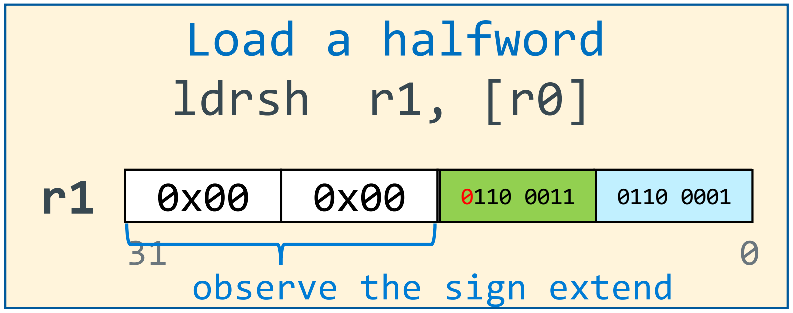
Load a word (no change)

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

r1

0x87	0x65	0110 0011	0110 0001

31 0



Load a halfword

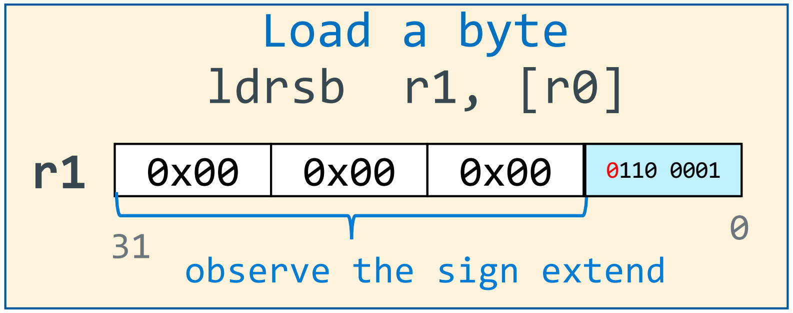
```
ldrsh  r1, [r0]
```

r1

0x00	0x00	0110 0011	0110 0001

31 0

observe the sign extend



Load a byte

```
ldrsb  r1, [r0]
```

r1

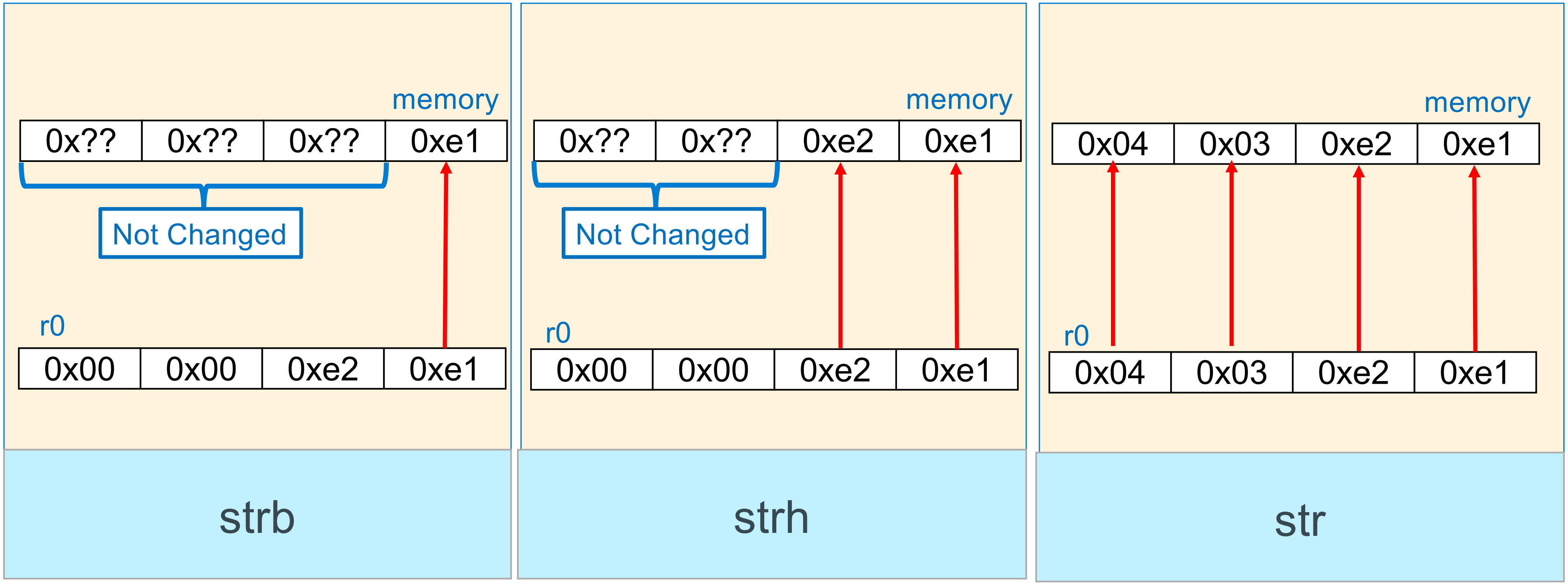
0x00	0x00	0x00	0110 0001

31 0

observe the sign extend

x

Storing 32-bit Registers To Memory 8-bit, 16-bit, 32-bit



memory

0x??	0x??	0x??	0xe1

Not Changed

r0

0x00	0x00	0xe2	0xe1

strb

memory

0x??	0x??	0xe2	0xe1

Not Changed

r0

0x00	0x00	0xe2	0xe1

strh

memory

0x04	0x03	0xe2	0xe1

r0

0x04	0x03	0xe2	0xe1

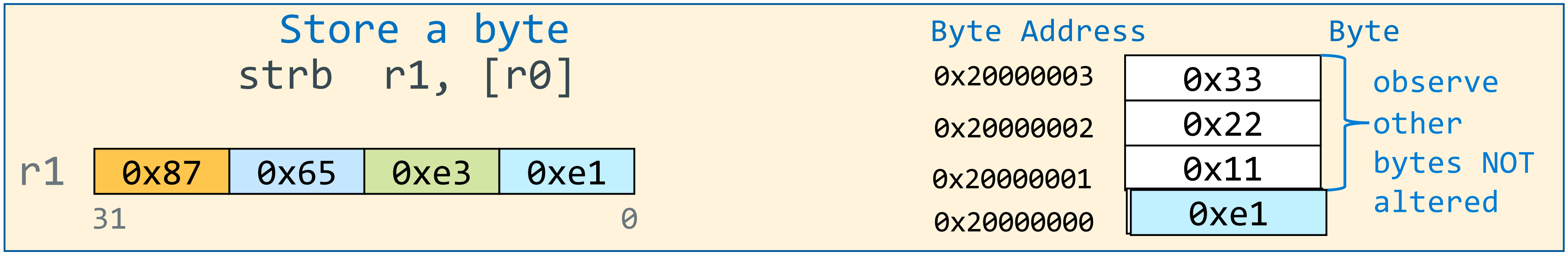
str

x

Store a Byte, Half-word, Word

initial value in r0

0x20	0x00	0x00	0x00



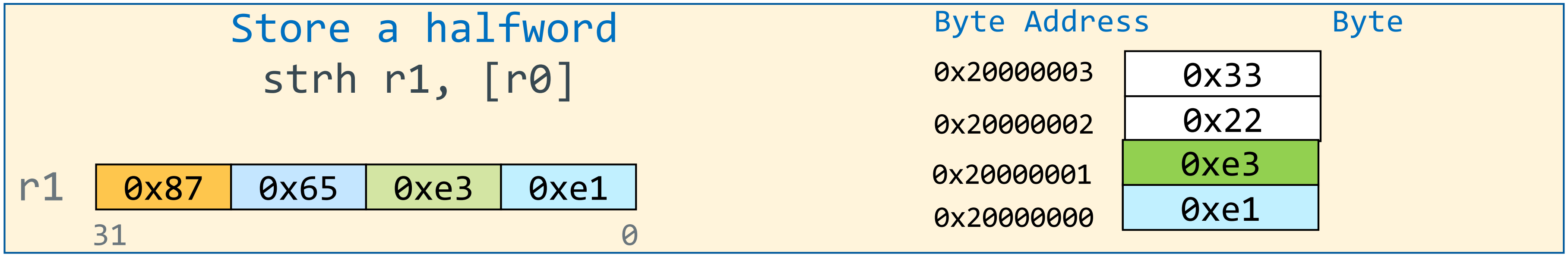
Store a byte

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

r1	31			0
	0x87	0x65	0xe3	0xe1

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

observe other bytes NOT altered

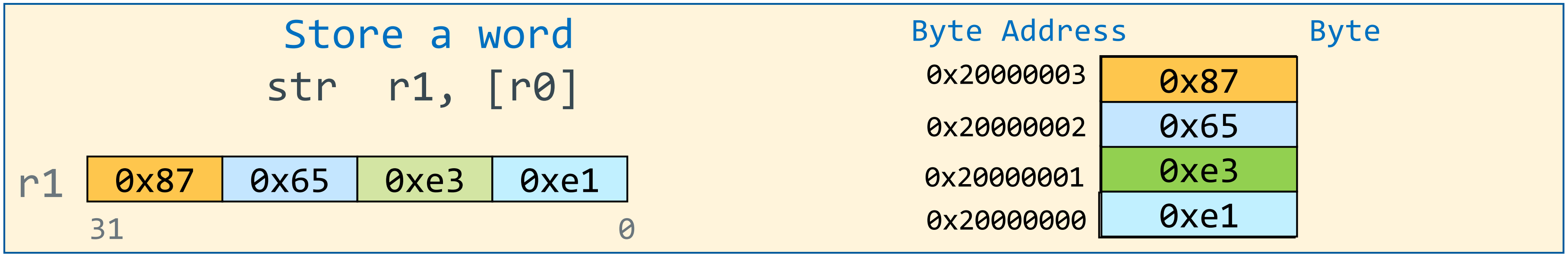


Store a halfword

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

r1	31			0
	0x87	0x65	0xe3	0xe1

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



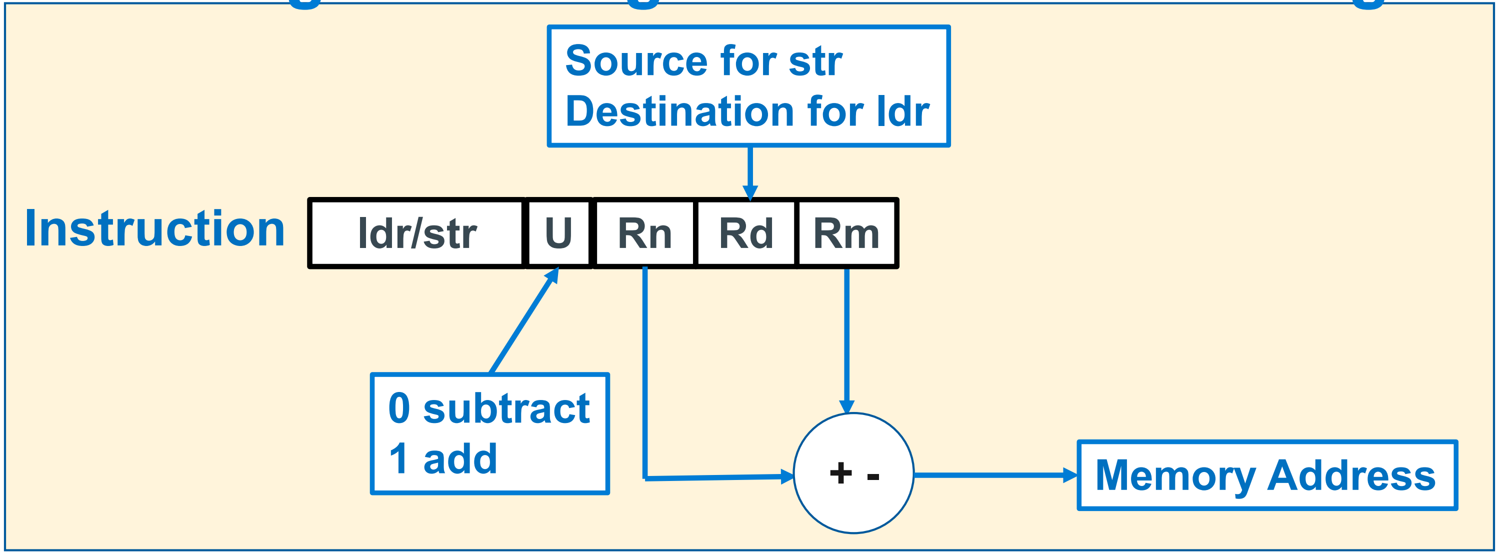
Store a word

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

r1	31			0
	0x87	0x65	0xe3	0xe1

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

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]`

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

Array addressing with ldr/str

Array element	Base addressing	Immediate offset	register offset

`char ch[0]`	`ldrb r2, [r0]`	`ldrb r2, [r0, 0]`	`mov r4, 0`
			`ldrb r2, [r0, r4]`

`char ch[1]`	`add r0, r0, 1`
	`ldrb r2, [r0]`
		`ldrb r2, [r0, 1]`	`mov r4, 1`
			`ldrb r2, [r0, r4]`

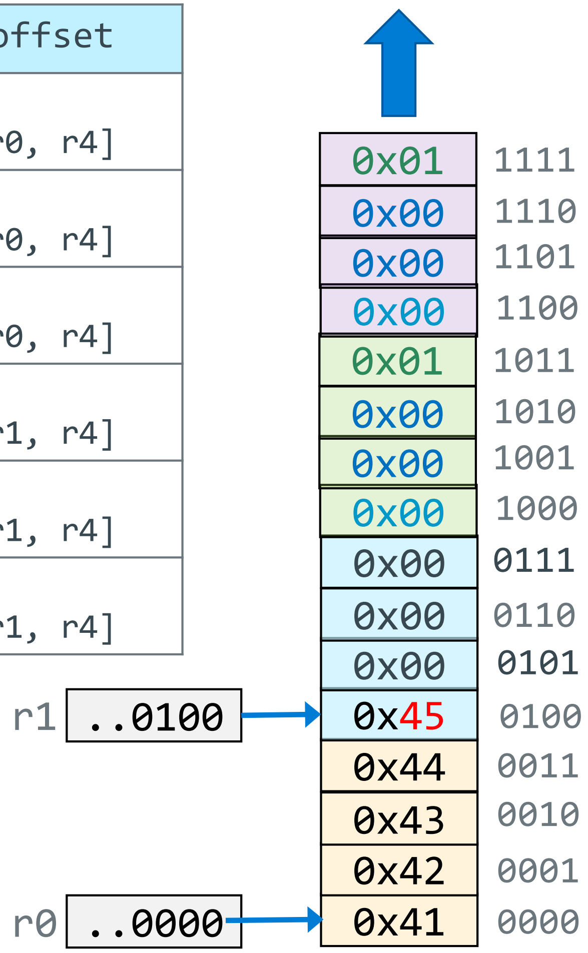
`char ch[2]`	`add r0, r0, 2`
	`ldrb r2, [r0]`
		`ldrb r2, [r0, 2]`	`mov r4, 2`
			`ldrb r2, [r0, r4]`

`int x[0]`	`ldr r2, [r1]`	`ldr r2, [r1, 0]`	`mov r4, 0`
			`ldr r2, [r1, r4]`

`int x[1]`	`add r1, r1, 4`
	`ldr r2, [r1]`
		`ldr r2, [r1, 4]`	`mov r4, 4`
			`ldr r2, [r1, r4]`

`int x[2]`	`add r1, r1, 8`
	`ldr r2, [r1]`
		`ldr r2, [r1, 8]`	`mov r4, 8`
			`ldr r2, [r1, r4]`

table rows are
independent instructions not a sequence



0x01	1111
0x00	1110
0x00	1101
0x00	1100
0x01	1011
0x00	1010
0x00	1001
0x00	1000
0x00	0111
0x00	0110
0x00	0101
0x45	0100
0x44	0011
0x43	0010
0x42	0001
0x41	0000

r1 ..0100

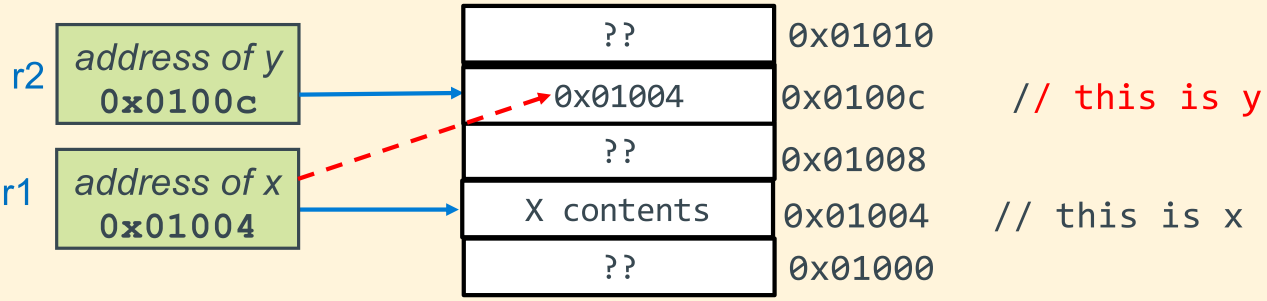
r0 ..0000

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;



r2 — address of y 0x0100c

r1 — address of x 0x01004

??	0x01010	
0x01004	0x0100c	// this is y
??	0x01008	
X contents	0x01004	// this is x
??	0x01000	

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

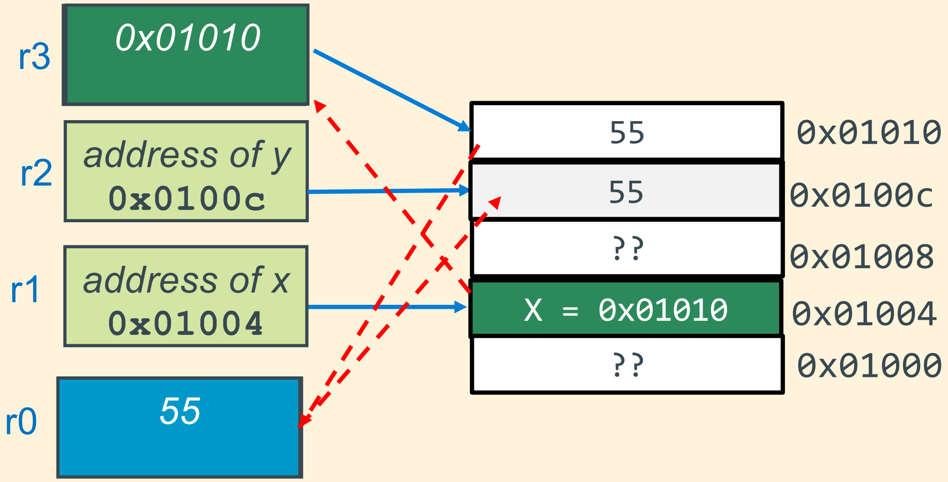
x

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
```

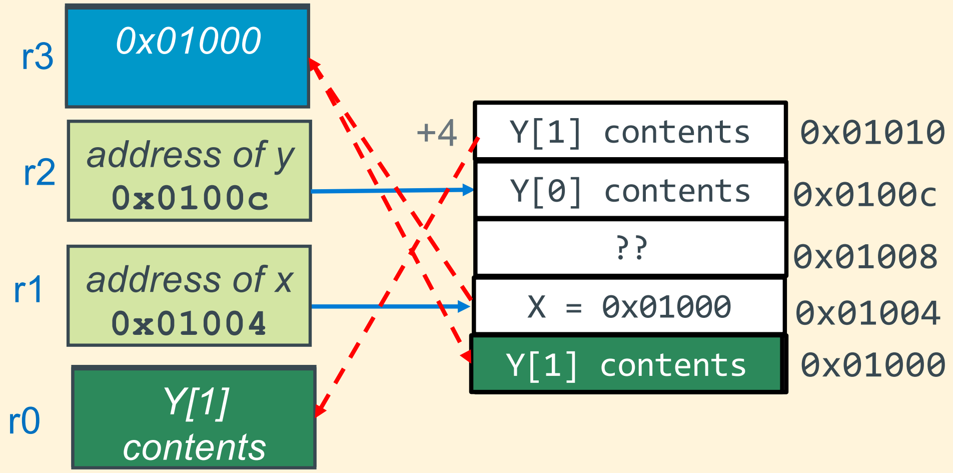
x

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

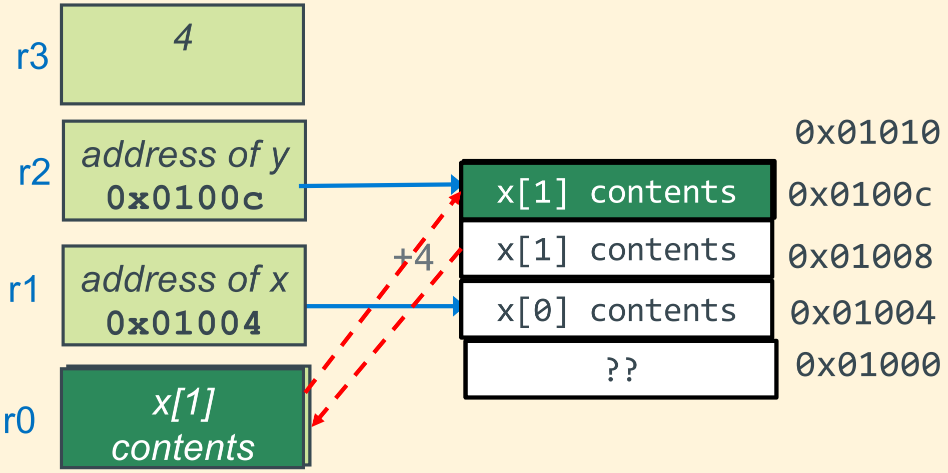
ldr/str practice - 4

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

PA8 Assembly Functions

```
    #include "cipher.h"
    .text     // start of text segment

    // int encrypt(char *iobuf, char *bookbuf, int cnt)
    // encrypts  iobuf with bookbuf; updating iobuf

    .global encrypt
    .type   encrypt, %function
    .equ    FP_OFF, 28

encrypt:
    push    {r4-r9, fp, lr}
    add     fp, sp, FP_OFF
    // do not alter anything above this line
    // r0 contains char *iobuf
    // r1 contains char *bookbuf
    // r2 contains cnt
    // r3 is ok to use
    // r4-r9 preserved registers are ok to use

    cmp     r2, 0               // if buffer empty we are done
    ble     .Ldone

    // your code here

    // do not alter anything below this line
.Ldone:
    mov     r0, r2              // return cnt processed
    sub     sp, fp, FP_OFF
    pop     {r4-r9, fp, lr}
    bx      lr
    .size encrypt, (. - encrypt)
    .section .note.GNU-stack,"",%progbits
.end
```

Function Header
Assembly directives

Function Prologue
always at top of function

Description of the register state at this point in the code

Function Epilogue
always at bottom of function

Function Footer
Assembly directive

Version 1.03

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 3

Lecture 18 – November 17, 2022

Keith Muller

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**

Base Register Addressing

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

void cpy(int *, int *, int);

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

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

    return EXIT_SUCCESS;
}
```

x

Base Register

```
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .text
    .global cpy
    .type   cpy, %function
    .equ    FP_OFF, 12
    // r0 contains int *src
    // r1 contains int *dst
    // r2 contains int cnt
    // r3 loop term pointer
    // r4 contains int

cpy:
    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 cpy, (. - cpy)
    .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

Base Register Addressing + Immediate offset

```
#include <stdio.h>
#include <stdlib.h>
#define SZ 6 // must be even (should add checks!)

void swb(int *, int);  /swap even and odd elements!

int main(void)
{
    int  src[SZ] = {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

    swb(src,SZ);
    for (int i = 0; i < SZ; i++)
        printf("%x\n", *(src + i));
    return EXIT_SUCCESS;
}
```

```
./a.out
b
a
d
c
f
e
```

x

Base Register + Immediate Offset

```
    .arch armv6
    .arm
    .fpu vfp
    .syntax unified
    .text
    .global swb
    .type   swb, %function
    .equ    FP_OFF, 12
    // r0 contains int *src
    // r1 contains int len must be even!
    // r2 loop termination pointer
    // r3 contains odd offset int
    // r4 contains even even int
cpy:
    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 swb, (. - swb)
    .end
```

```
    cmp     r1, 0
    ble     .Ldone
```

pre loop guard

```
    lsl     r1, r1, 2
    add     r2, r0, r1
.Ldo:
    ldr     r3, [r0, 0]   // odd index element
    ldr     r4, [r0, 4]   // even index element
    str     r3, [r0, 4]   // odd to even
    str     r4, [r0, 0]   // even to odd
    add     r0, r0, 8    // step to next PAIR!
    cmp     r0, r2
    blt     .Ldo
```

loop guard

```
.Ldone:
```

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
```

```
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

Base Register + Register Offset Two Buffers

```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;
}
```

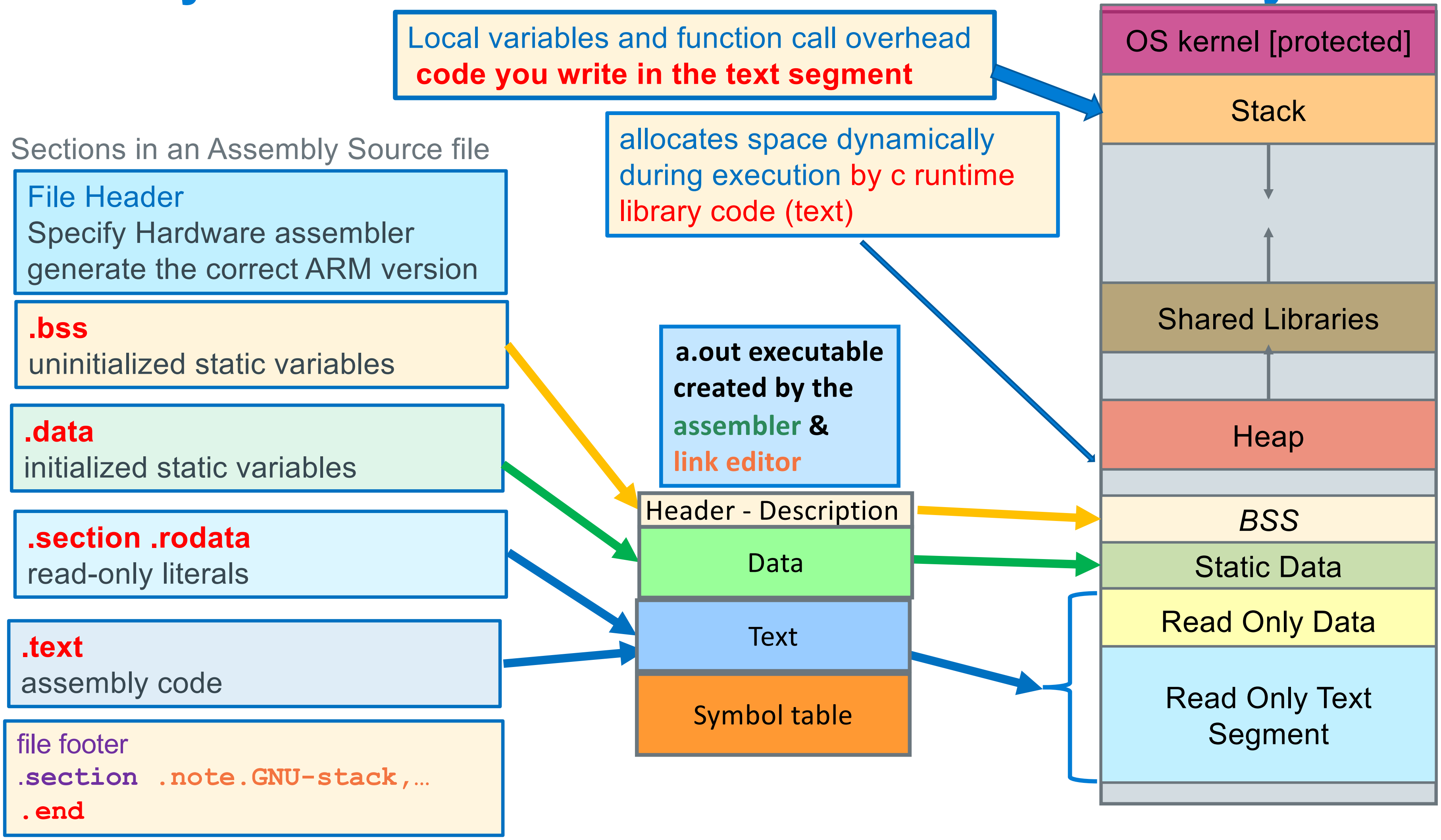
```
//code
cpy:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    // r0 contains char *src
    // r1 contains char *dst
    // r2 contains int len
    // r3 contains int i
    // r4 contains char
    mov     r3, 0
.Lfor:
    cmp     r3, r2
    bge     .Ldone
    ldrb    r4, [r0, r3]
    strb    r4, [r1, r3]
    add     r3, r3, 1
    b       .Lfor
.Ldone:
    sub     sp, fp, FP_OFF
    pop     {r4, r5, fp, lr}
    bx      lr
// code
```

one increment covers both arrays

- Make sure to index by bytes and increment the index register by sizeof(int) = 4

x

Assembly Source File to Executable to Linux Memory



Sections in an Assembly Source file

File Header
Specify Hardware assembler
generate the correct ARM version

.bss
uninitialized static variables

.data
initialized static variables

.section .rodata
read-only literals

.text
assembly code

file footer
`.section .note.GNU-stack,…`
`.end`

a.out executable created by the assembler & link editor

Local variables and function call overhead
code you write in the text segment

allocates space dynamically during execution by c runtime library code (text)

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`

Creating Segments, Definitions In Assembly Source

- The following assembler directives indicate the ***start*** of a **memory segment specification**
  - **Remains in effect** until the next segment directive is seen

```
.bss
       // start uninitialized static segment variables definitions
       // does not consume any space in the executable file
.data
       // start initialized static segment variables definitions
.section .rodata
       // start read-only data segment variables definitions
.text
       // start read-only text segment (code)
```

- Define a literal, static variable or global variable in a segment

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

  - Label: this is the **variables *name***
  - Size_Directive tells the assembler *how much space to **allocate*** for that **variable**
- Each **optional** `expression` specifies the contents of one memory location of `.size_directive`
  - `expression` can be in decimal, hex (0x…), octal (0…), binary (0b…), ASCII (' '), string " "

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)	.hword .short	1	short length = 0x55aa;	length: .hword 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"

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

```
.bss
             .align 2
num:         .word 0
.data
             .align 2
ptr:         .word num
lit:         .word .Lmsg
msg:         .string "123"
.section .rodata
.Lmsg:       .string "456"
```

initializes a pointer

x

Static Variable Alignment: Using .align

Accessing **address aligned** memory based on data type has the best performance

integer
4 bytes
short
2 bytes
char
1

SIZE	Directive	Address ends in	Align Directive
`8-bit char -1 byte`	`.byte`	`0b..0 or 0b..1`	

`16-bit int -2 bytes`	`.hword`
	`.short`
		`0b..0`	`.align 1`

`32-bit int -4 bytes`	`.word`
	`.long`
		`0b..00`	`.align 2`

4 bytes	2 Bytes	1 Byte	Addr. (hex)
Addr = 0x0C	Addr = 0x0E		0x0F
			0x0E
	Addr = 0x0C		0x0D
			0x0C
Addr = 0x08	Addr = 0x0A		0x0B
			0x0A
	Addr = 0x08		0x09
			0x08
Addr = 0x04	Addr = 0x06		0x07
			0x06
	Addr = 0x04		0x05
			0x04
Addr = 0x00	Addr = 0x02		0x03
			0x02
	Addr = 0x00		0x01
			0x00

`.align n` **before** variable definition to specify memory alignment requirements

- Tells the assembler **the next line that allocates memory** must **start** at the next higher memory address **where** the lower **n** address bits are zero
- At the **first use of any Segment directive**, alignment **starts at an 8-byte aligned address** (for doubles)
- Easy approach: Allocate from largest size variables to smallest size variables

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
char_buf:   .space 100
.data
array:      .word 1, 2, 3, 4, 5
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**

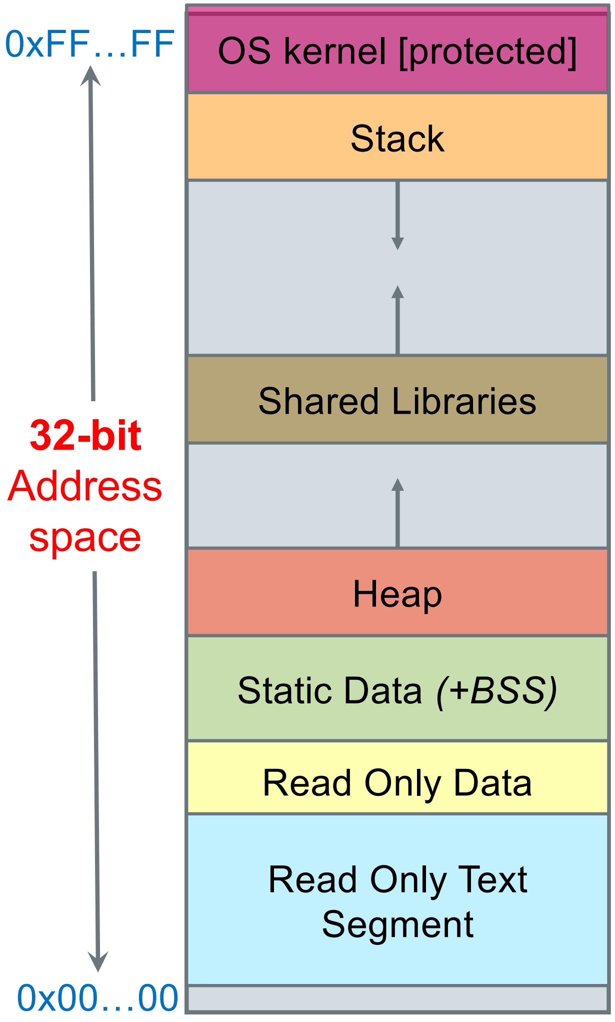
How to get a memory pointer into a register?

- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
- For each variable in one of the data segments you reference in a special form of the `ldr` instruction (next slide), the assembler makes an entry (it does this while assembling, so it is not seen in your source code) for that variable whose contents is the 32-bit Label address

```
        .bss
y:      .space 40
```

```
        .data
x:      .word 200
```

```
        .text
        // your code
        // last line of your code
        // below is added by the assembler
       .word  y       // contents: 32-bit address of y
       .word  x       // contents: 32-bit address of x
```



x

Loading and using pointers in registers

- Tell the assembler to create and USE a literal table to obtain 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

How to use the literal table to get a big constant into a register

- In data processing instructions, the field **imm8 + rotate 4 bits** is too small to store many numbers outside of the range of -256 to 255, 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

Preview: Simple Function Calls: An Example with printf()

- Where `r0, r1, r2, r3` are registers

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

  `printf("arg1", arg2, arg3, arg4)`

- We need to create a literal string for arg1 which tells `printf()` how to interpret the remaining arguments (up to three arguments total at this point in the class; more later)
  - Create the string and tell the assembler to place it into the read only data section

```c
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int a = 2;
    int b = 3;
    int c;

    c = a + b;
    printf("c=%d\n", c);

    return EXIT_SUCCESS;
}
```

We are going to put these variables in temporary registers

two passed args in this use of printf

`r0, r1`

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

```
// part of the text segment below

        mov     r2, 2           // int a = 2;
        mov     r3, 3           // int b = 3;
        add     r1, r2, r3      // int c = a + b;
                                // r1 is second arg
        ldr     r0, =.Lfst      // =literal address
        bl      printf
```

x

Preview: Simple Function Calls: An Example with fprintf()

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int a = 2;
    int b = 3;
    int c;

    c = a + b;
    fprintf(stdout, "c=%d\n", c);
                r0,   r1,    r2

    return EXIT_SUCCESS;
}

// stdout is a global var FILE * !!!!!
```

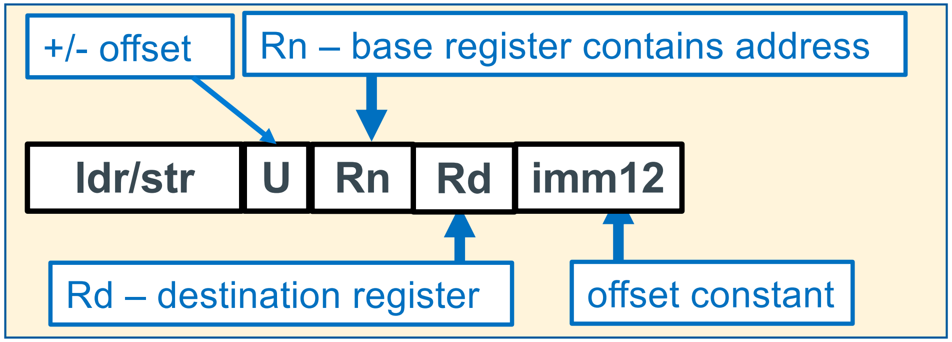
```
        .extern printf       //declare printf
        .section .rodata     // note the dots "."
.Lfst:  .string  "c=%d\n"
```

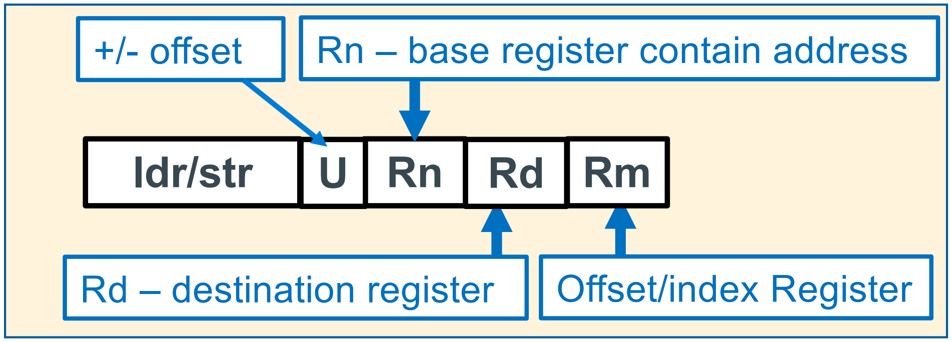
```
// part of the text segment below

        mov     r2, 2         // int a = 2;
        mov     r3, 3         // int b = 3;
        add     r2, r2, r3    // int c = a + b;
                              // r1 is second arg
        ldr     r1, =.Lfst    // =literal address
        ldr     r0, =stdout   // get stdout address
        ldr     r0, [r0]      // get stdout contents
        bl      printf
```

PA9 info:

- stdin, stdout, stderr are all global variable and are part of libc
  - these names are their lside (label names)
- to use them you must get their contents to pass to fprintf(), fread(), fwrite()

Reference: LDR/STR – Register To/From Memory Copy


+/- offset
Rn – base register contains address
ldr/str
U
Rn
Rd
imm12
Rd – destination register
offset constant



```
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

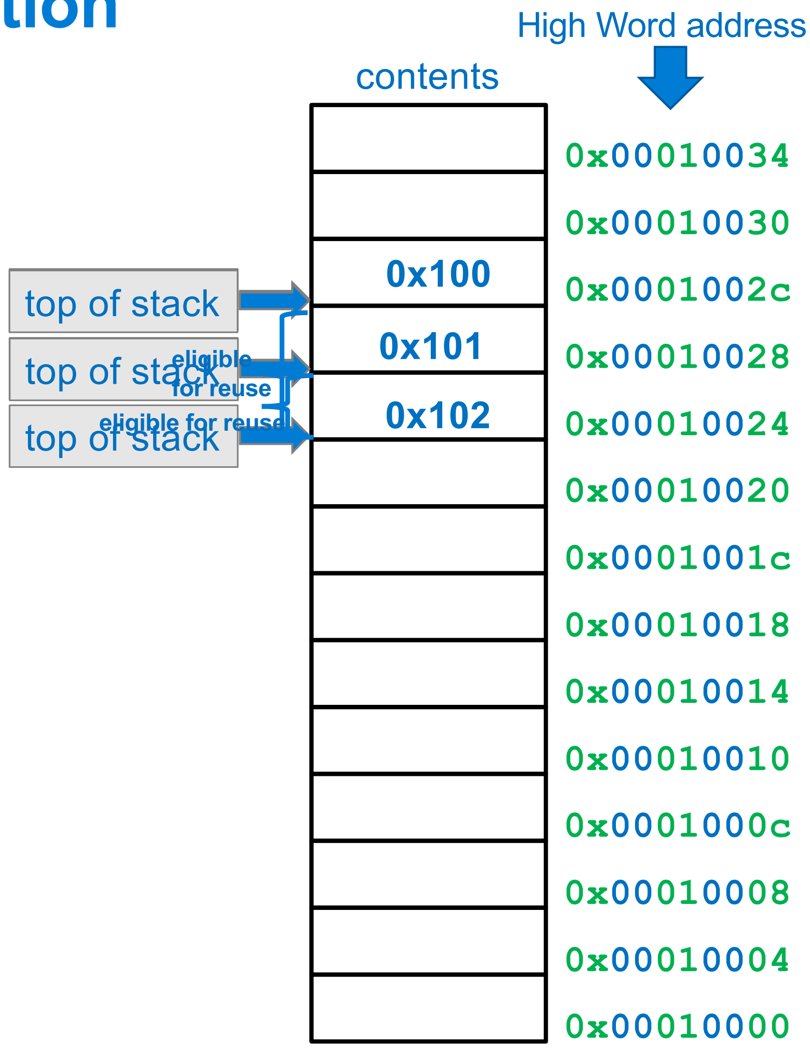
Data Structure Review: Stack Operation

- 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)

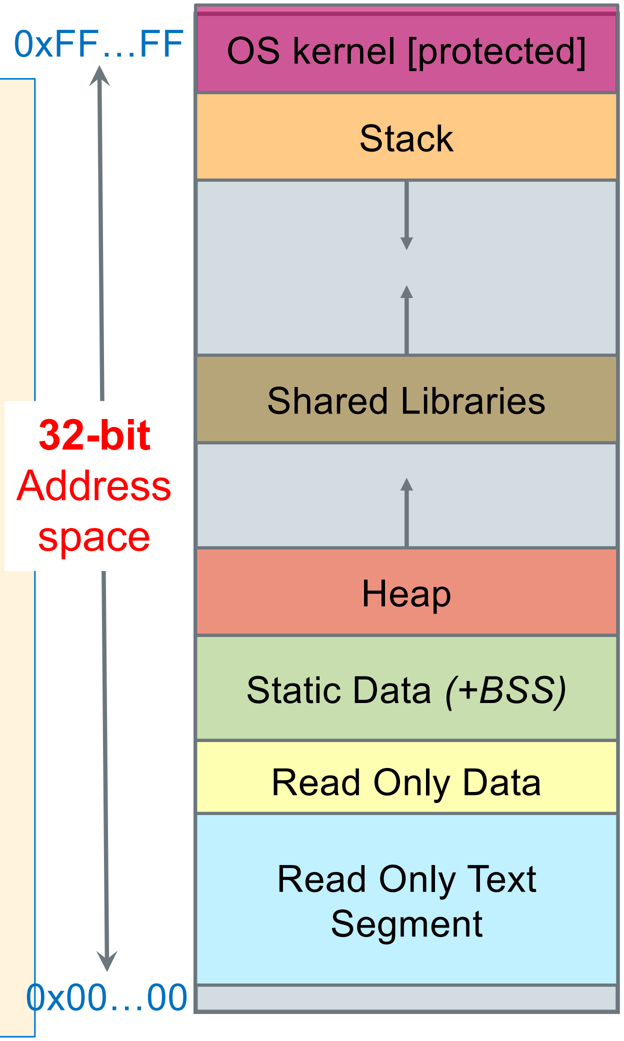


contents	High Word address
	0x00010034
	0x00010030
0x100	0x0001002c
0x101	0x00010028
0x102	0x00010024
	0x00010020
	0x0001001c
	0x00010018
	0x00010014
	0x00010010
	0x0001000c
	0x00010008
	0x00010004
	0x00010000

x

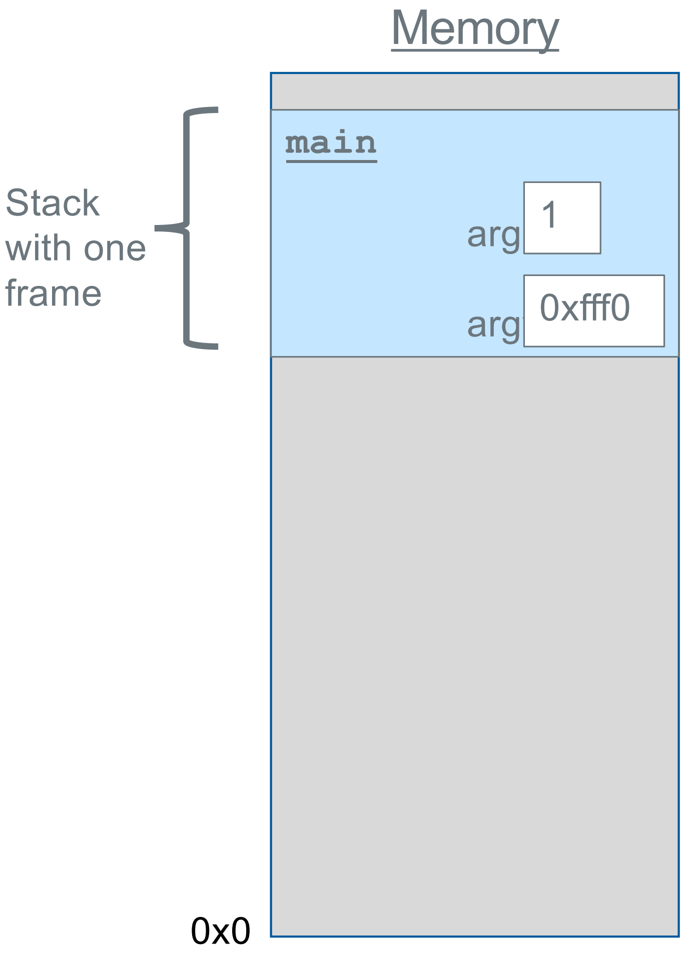
Stack Segment: Support of Functions

- 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)



The Stack

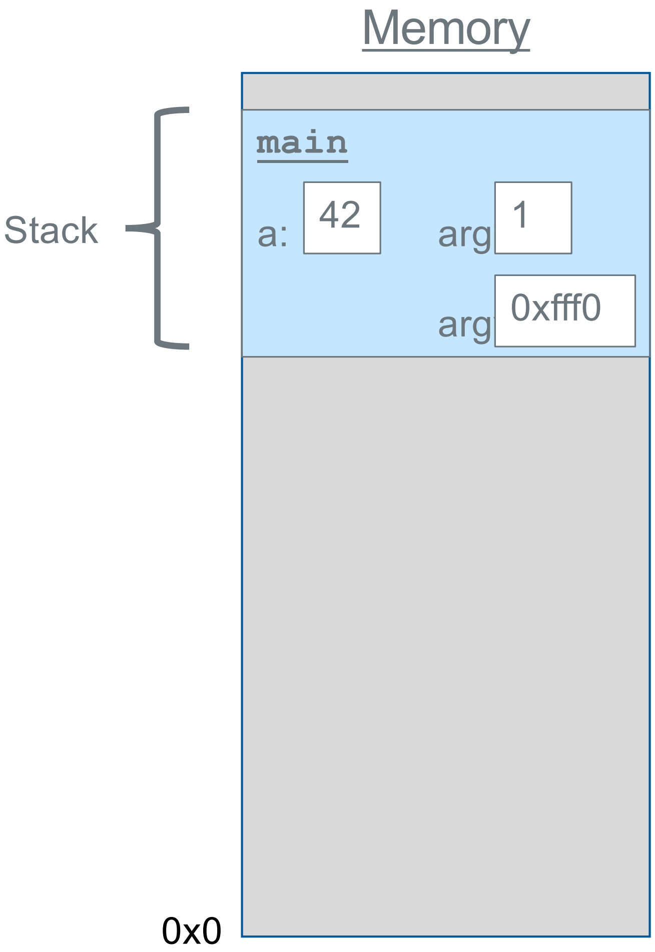
```
void func2() {
    int d = 0;
}

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

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



Stack with one frame

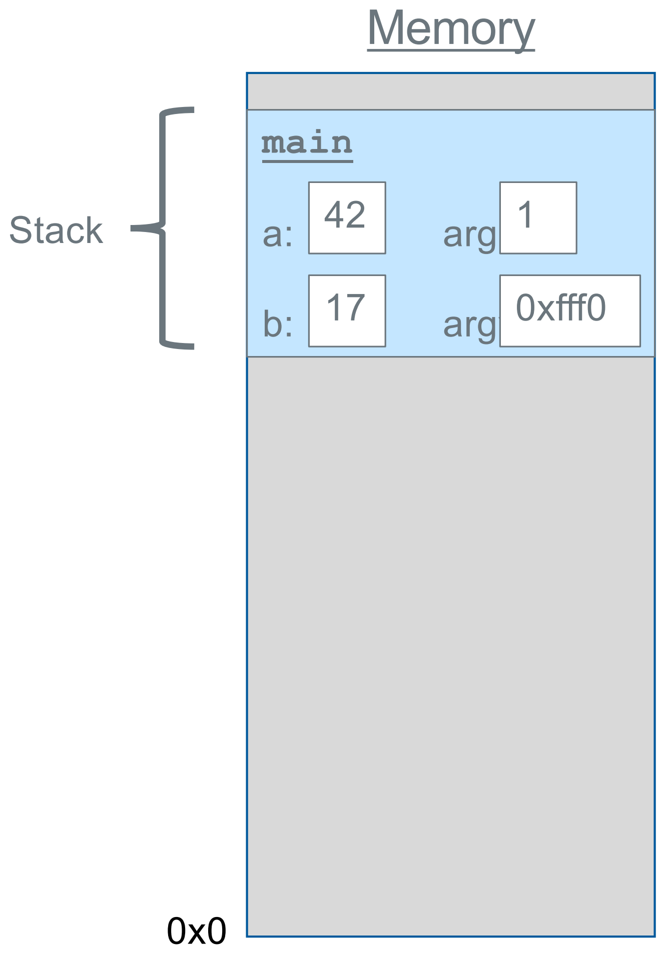
The Stack

```
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;
}
```



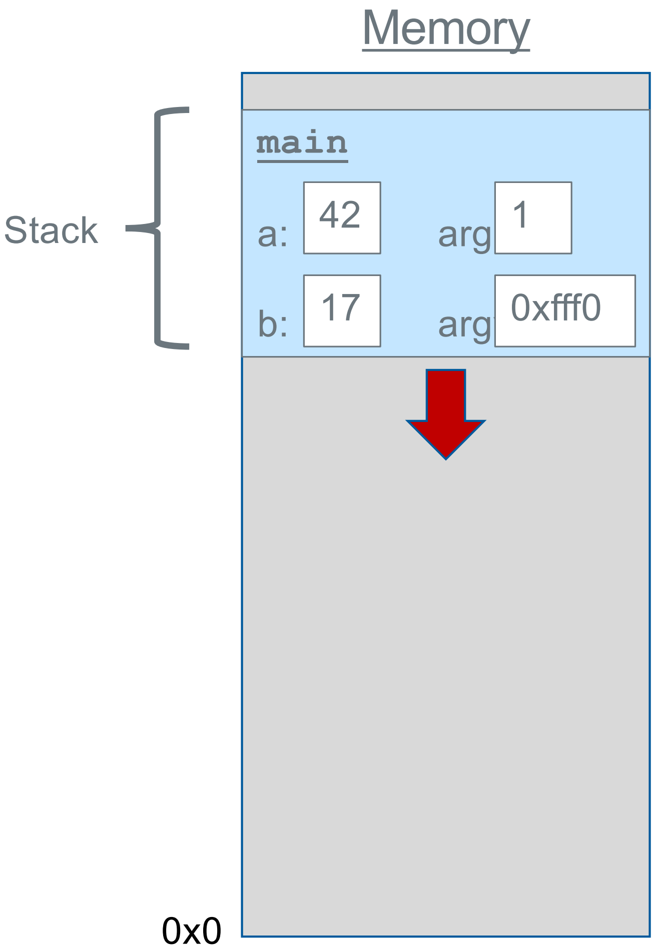
The Stack

```
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;
}
```



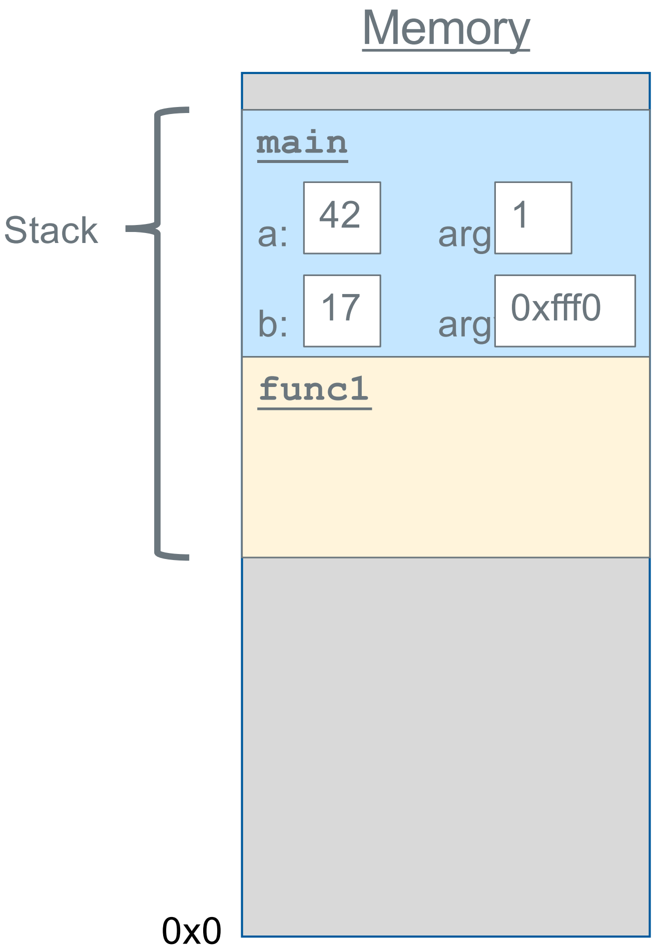
The Stack

```
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;
}
```



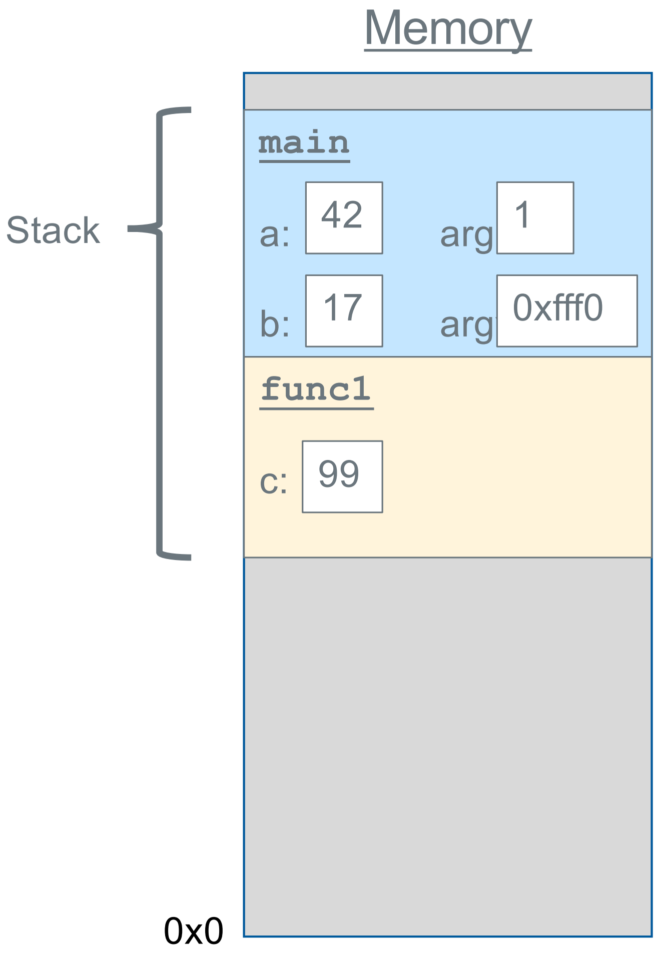
The Stack

```
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;
}
```



Memory

Stack

main

a: 42

arg 1

b: 17

arg 0xfff0

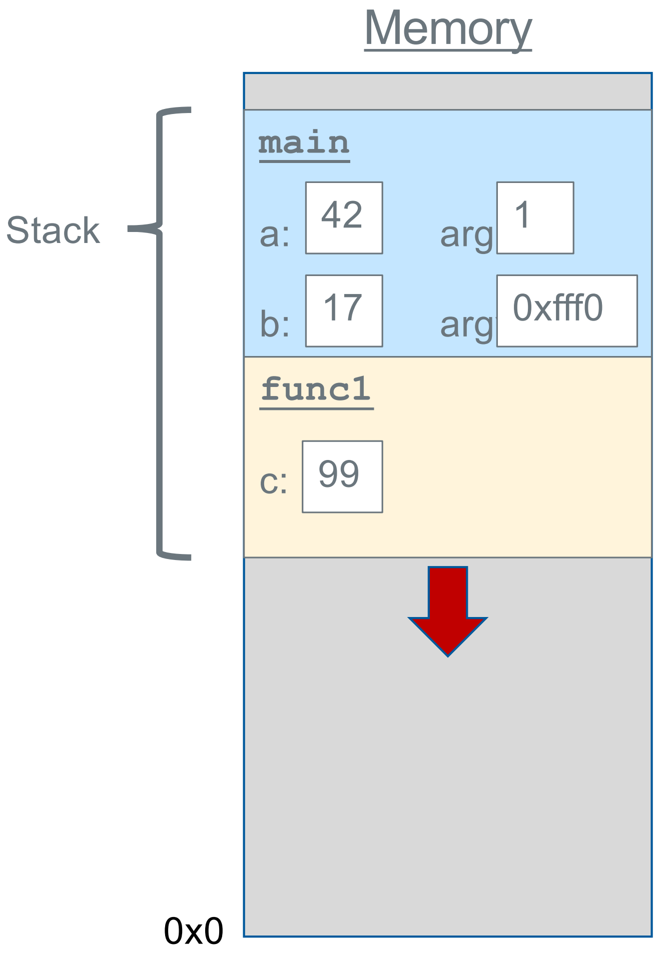
func1

c: 99

0x0

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;
}
```



Memory

Stack

main

a: 42

arg 1

b: 17

arg 0xfff0

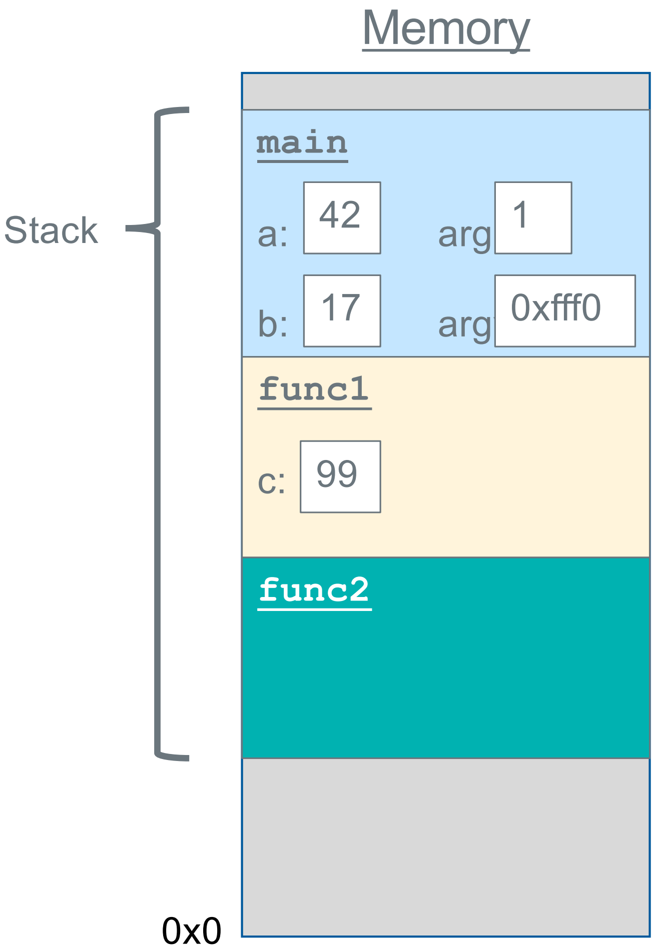
func1

c: 99

0x0

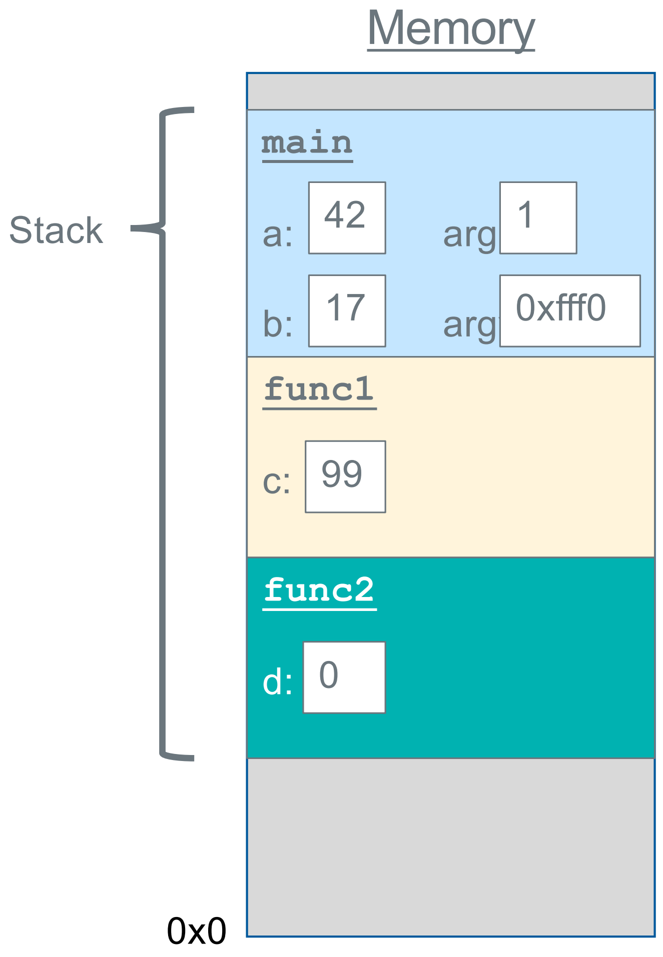
The Stack

```
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;
}
```



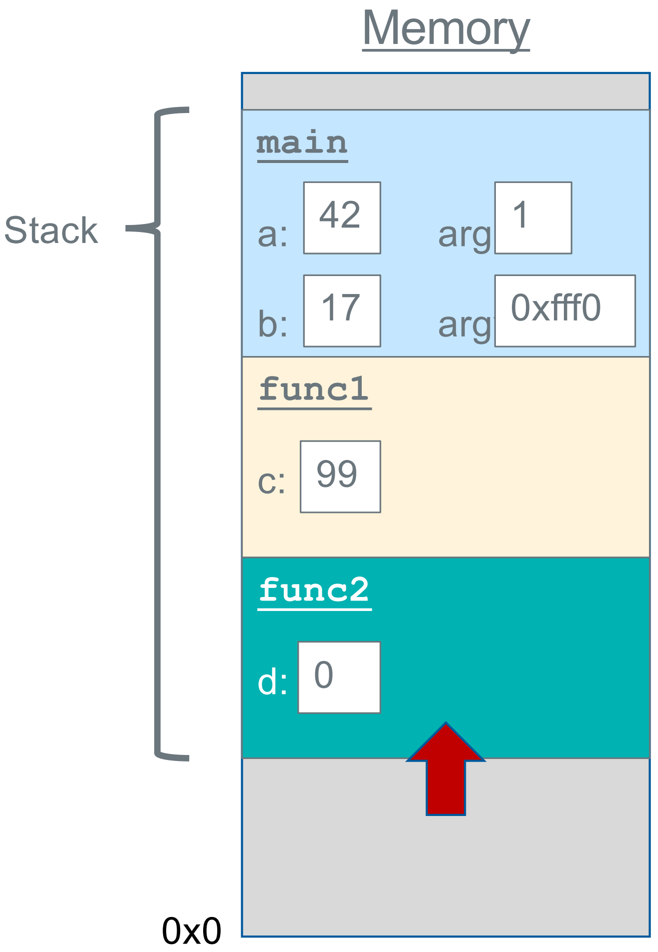
The Stack

```
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;
}
```



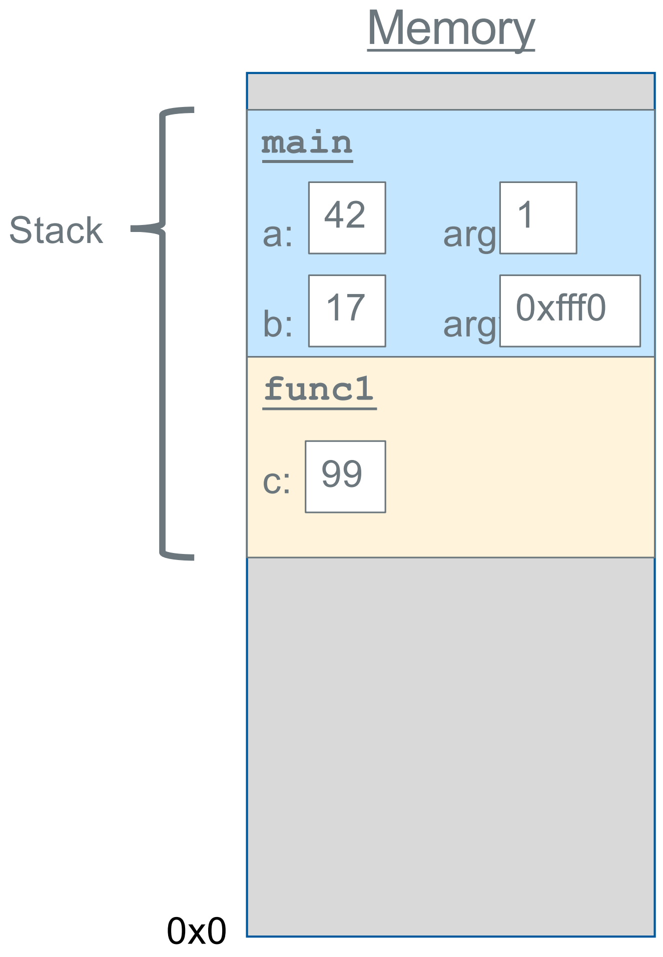
The Stack

```
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;
}
```



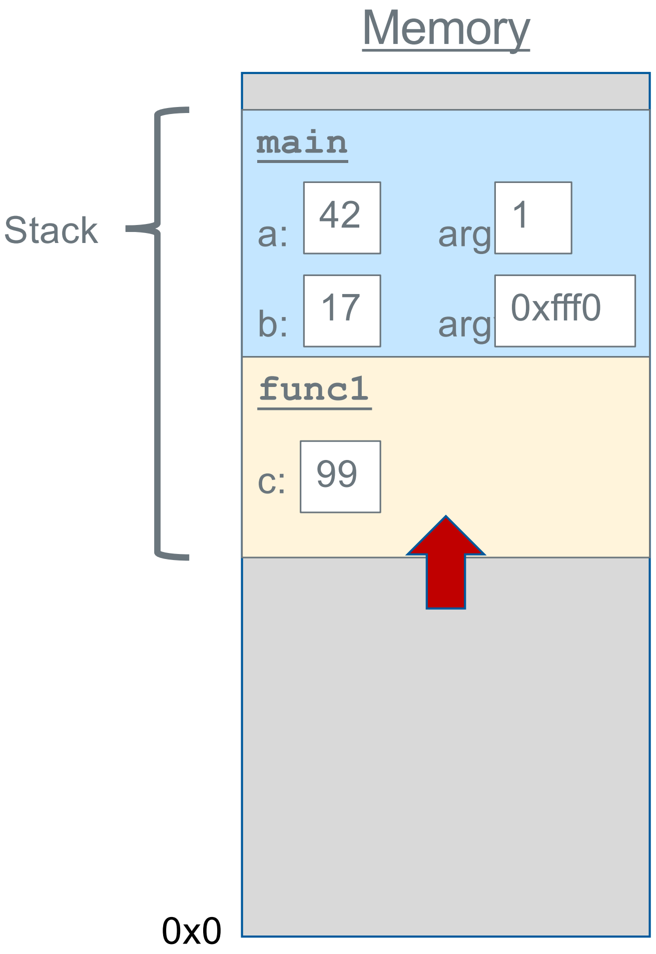
The Stack

```
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;
}
```



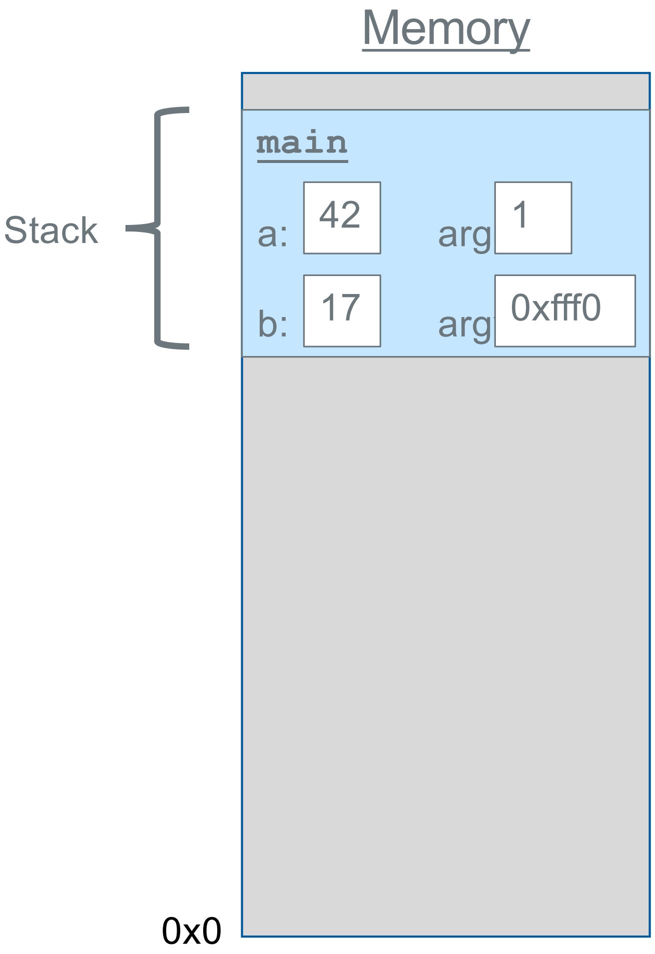
The Stack

```
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;
}
```



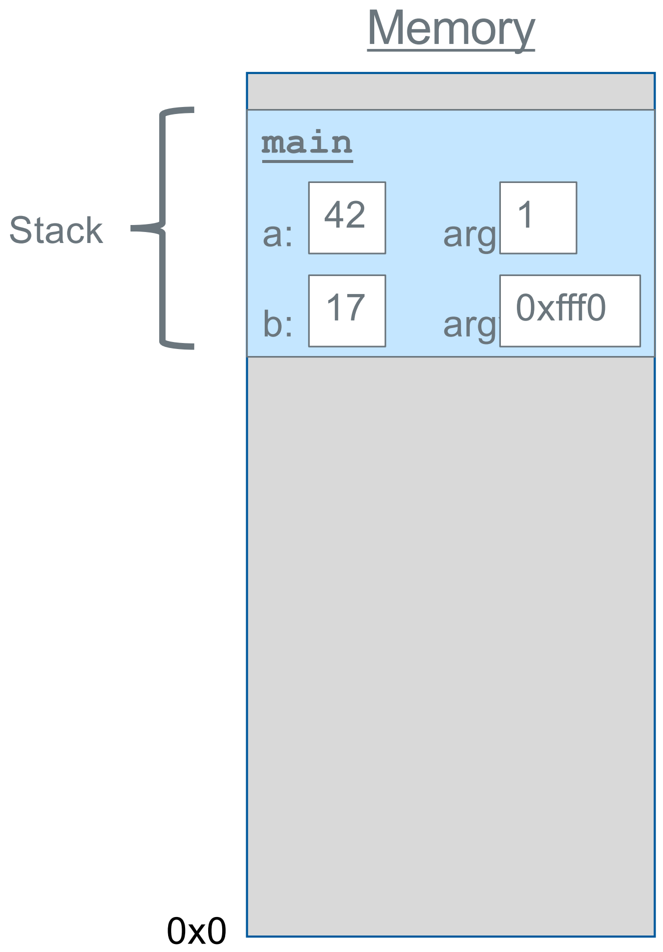
The Stack

```
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;
}
```



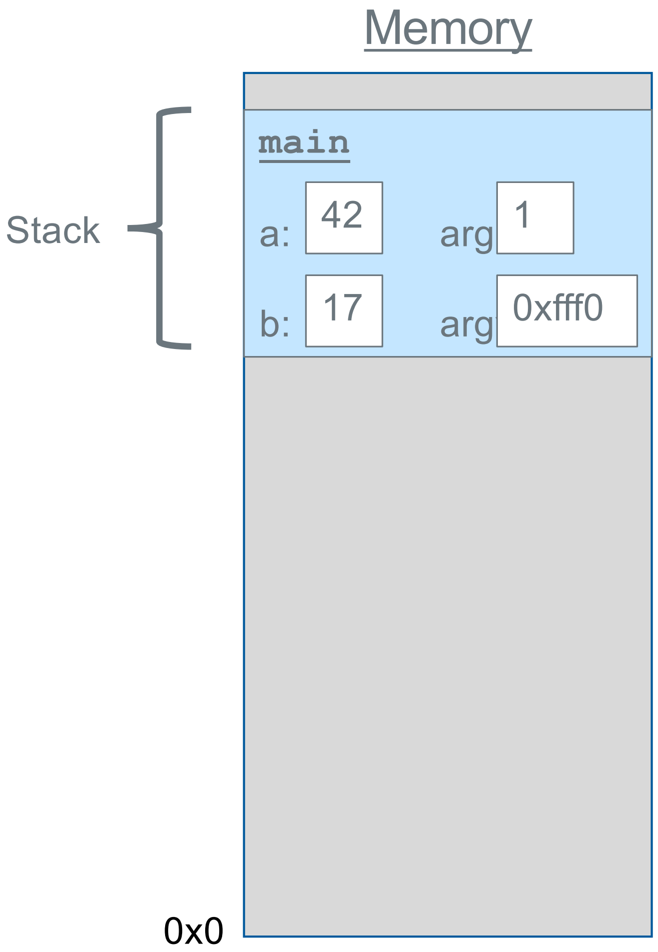
The Stack

```
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;
}
```



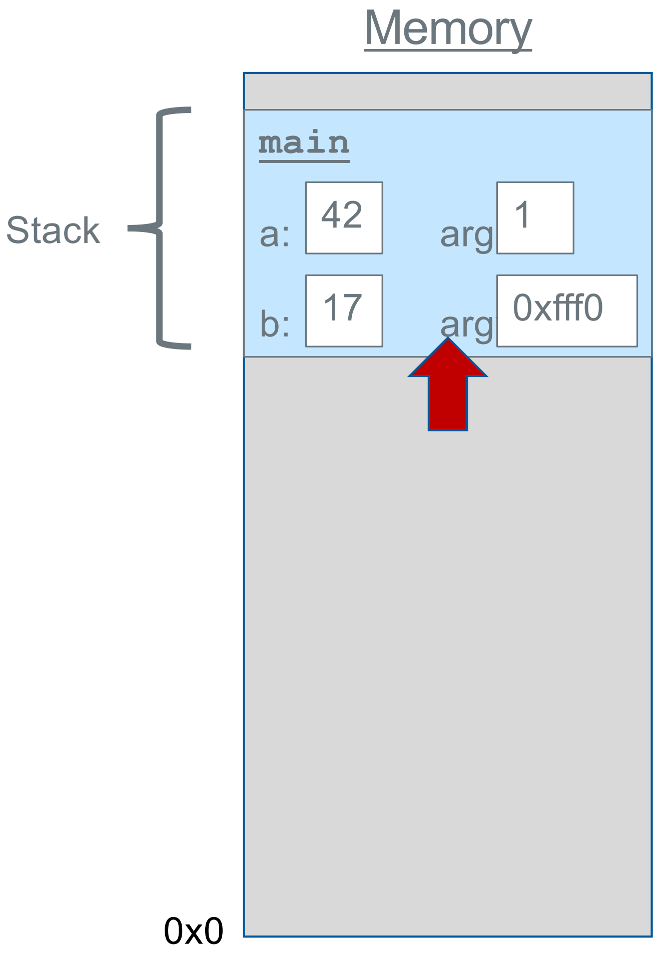
The Stack

```
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;
}
```



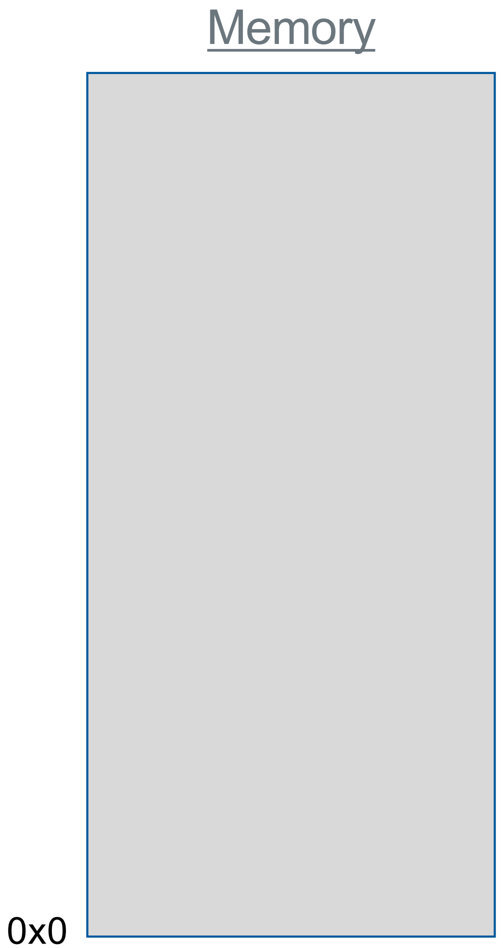
The Stack

```
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

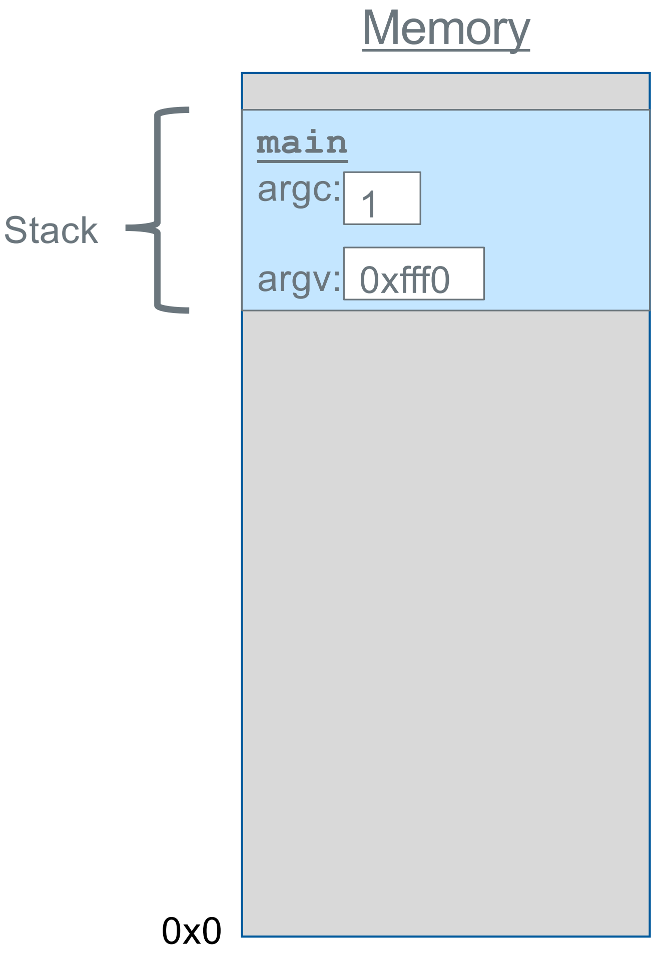
```
void func2() {
    int d = 0;
}

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

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

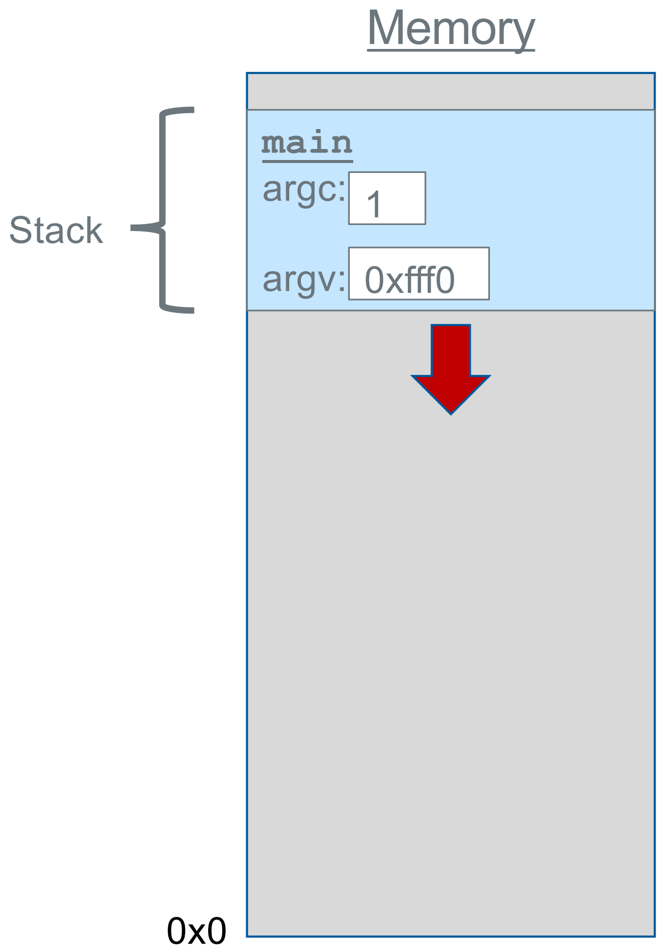
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

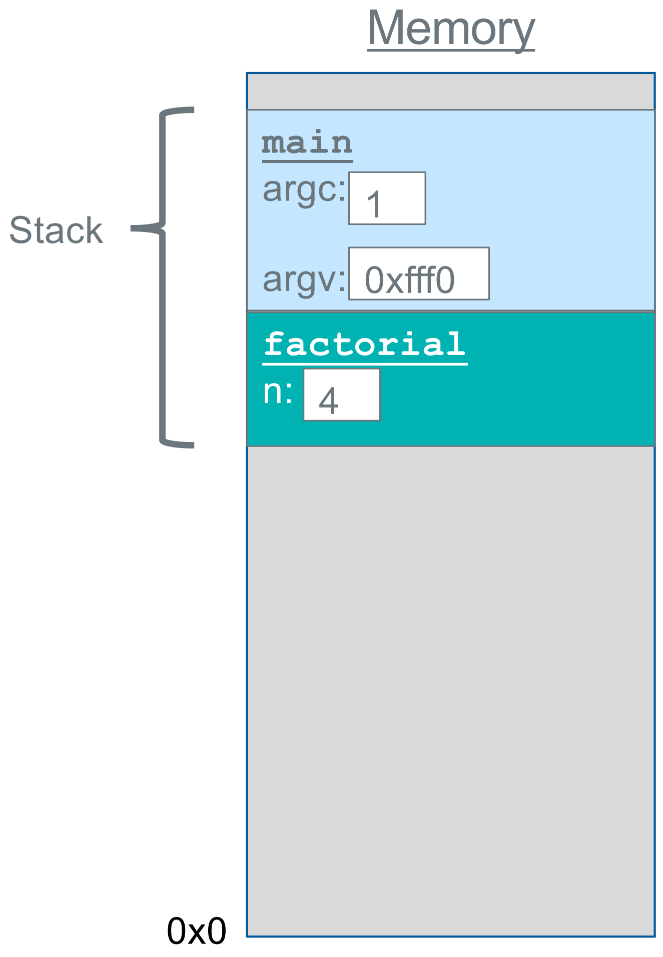
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

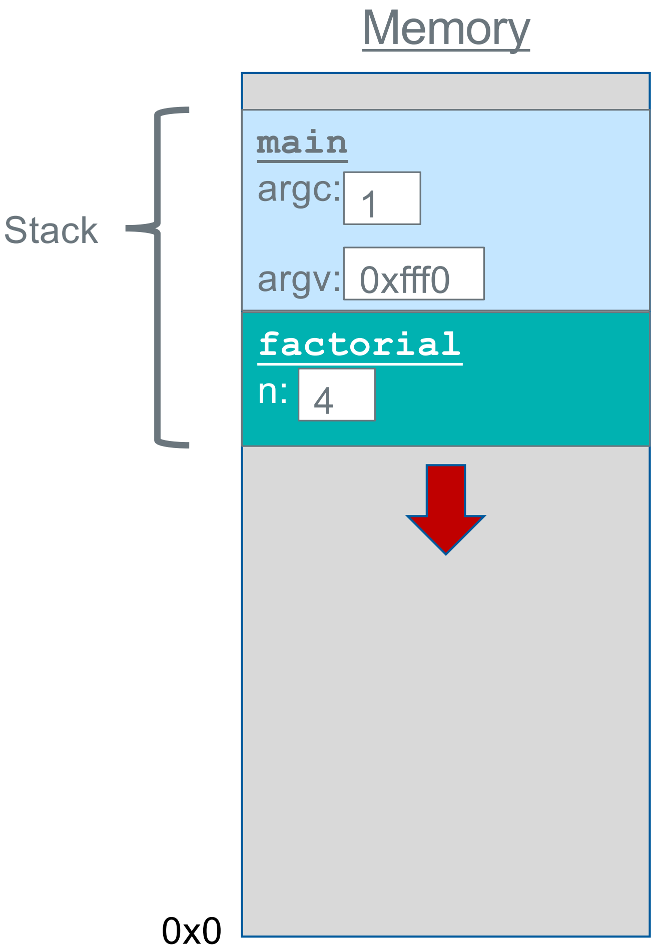
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

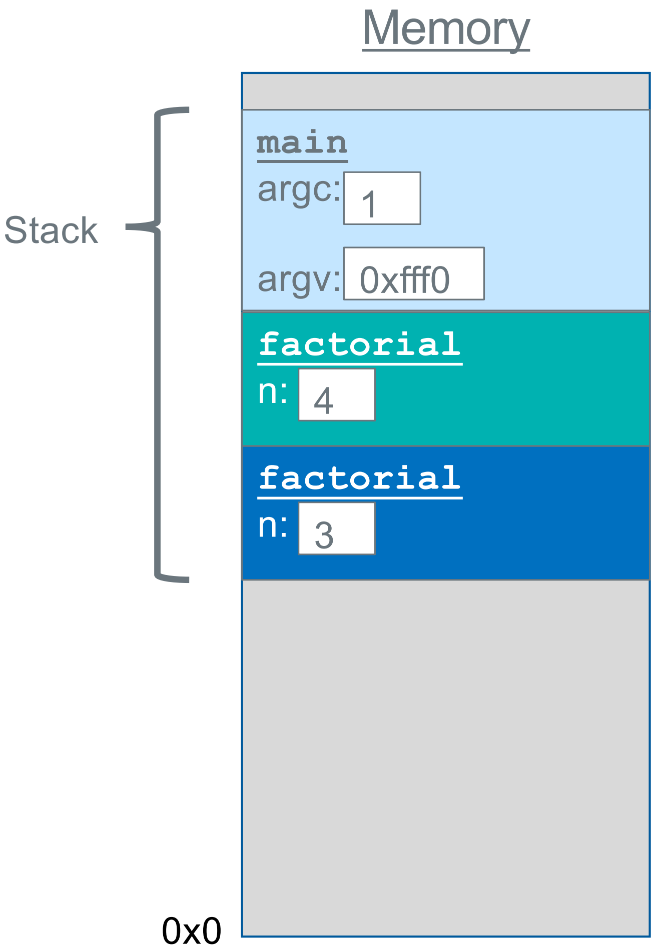
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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;
}
```



Memory

Stack

main

argc: 1

argv: 0xfff0

factorial

n: 4

factorial

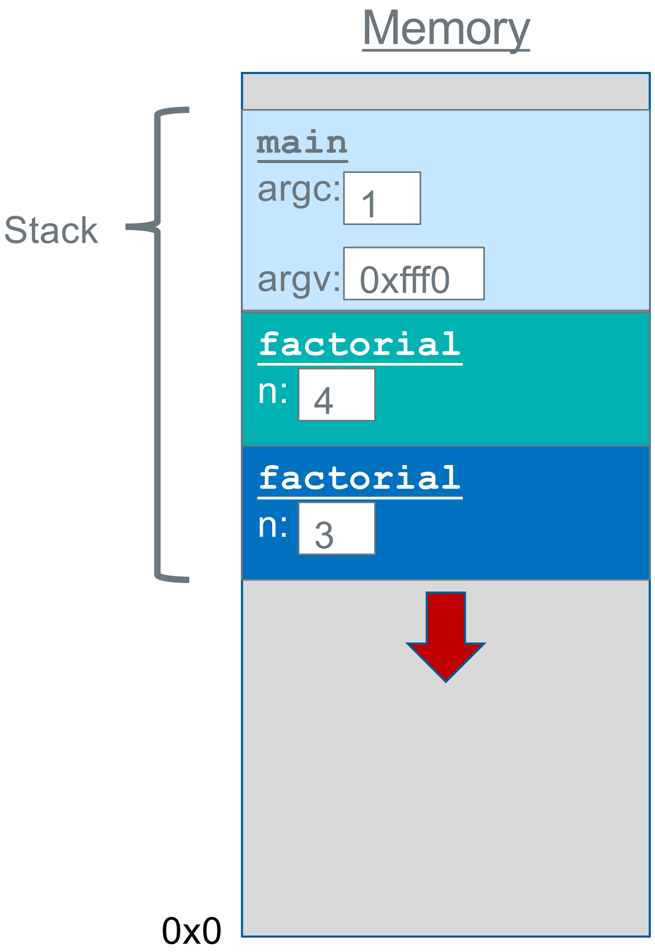
n: 3

0x0

The Stack

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

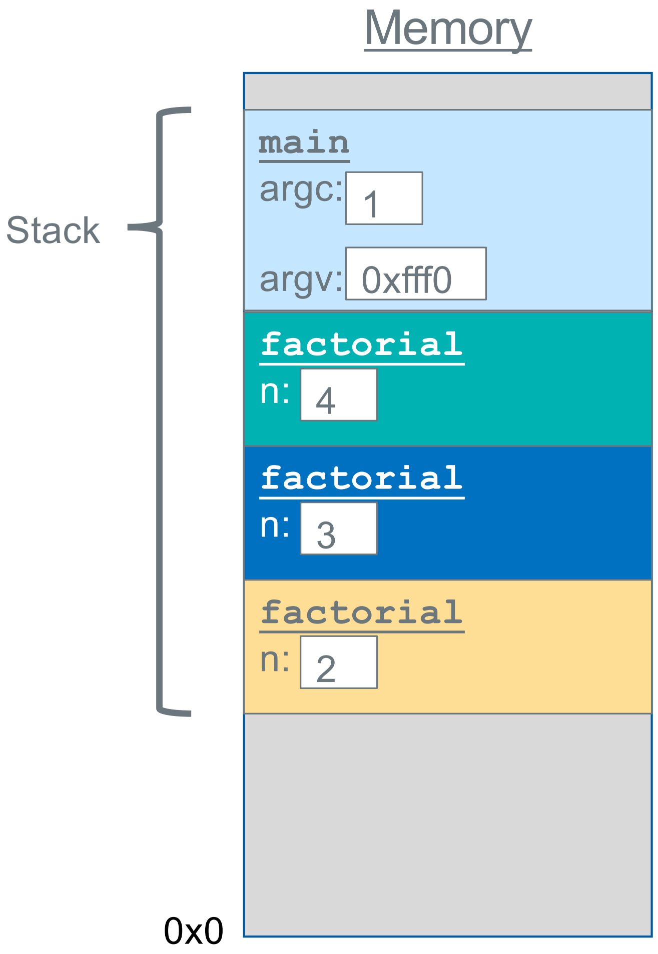
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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;
}
```



Memory

Stack

main

argc: 1

argv: 0xfff0

factorial

n: 4

factorial

n: 3

factorial

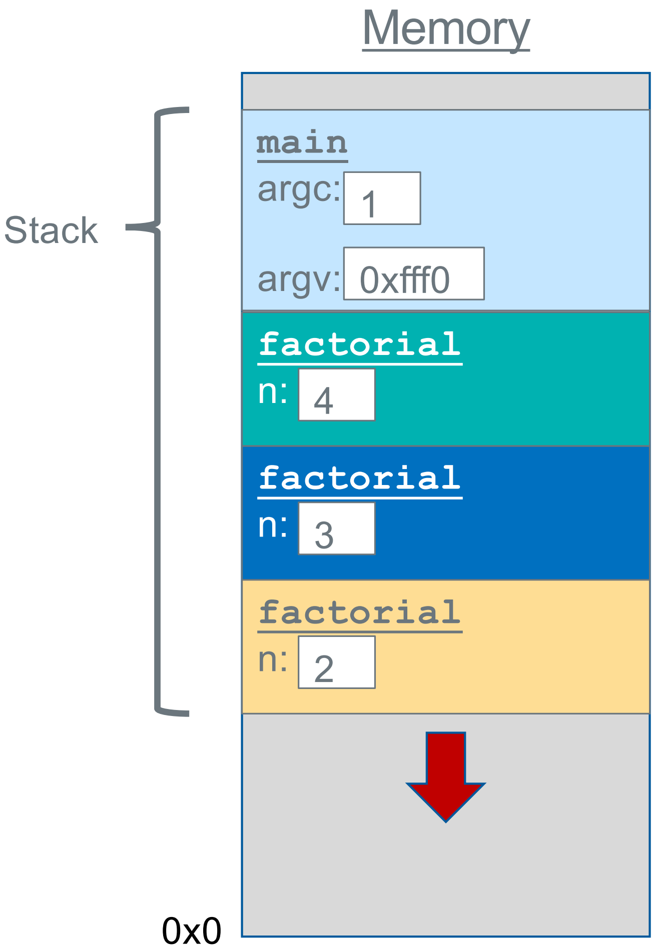
n: 2

0x0

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;
}
```



Memory

Stack

main

argc: 1

argv: 0xfff0

factorial

n: 4

factorial

n: 3

factorial

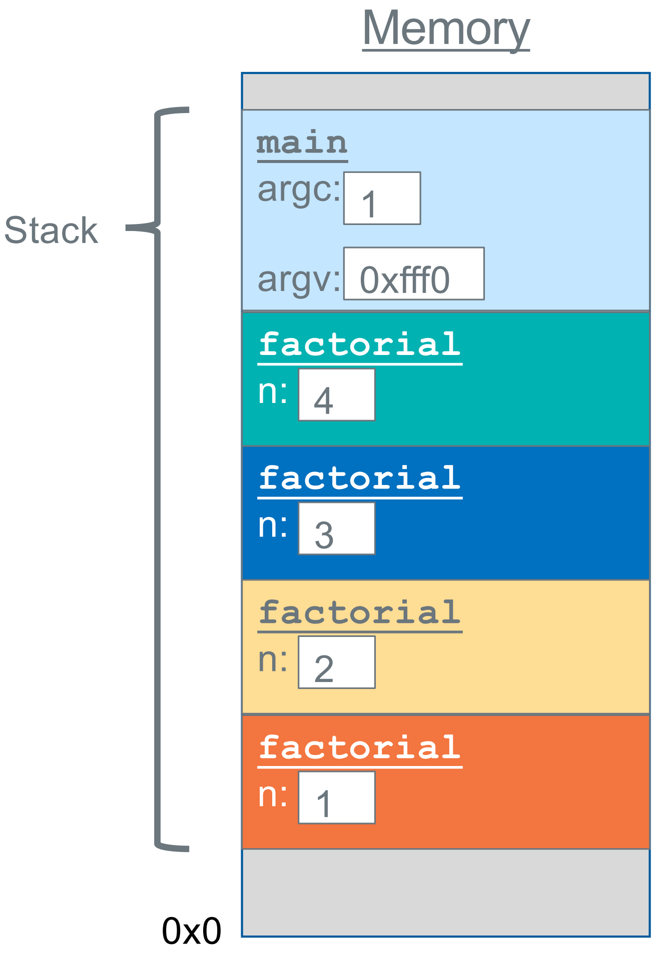
n: 2

0x0

The Stack

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

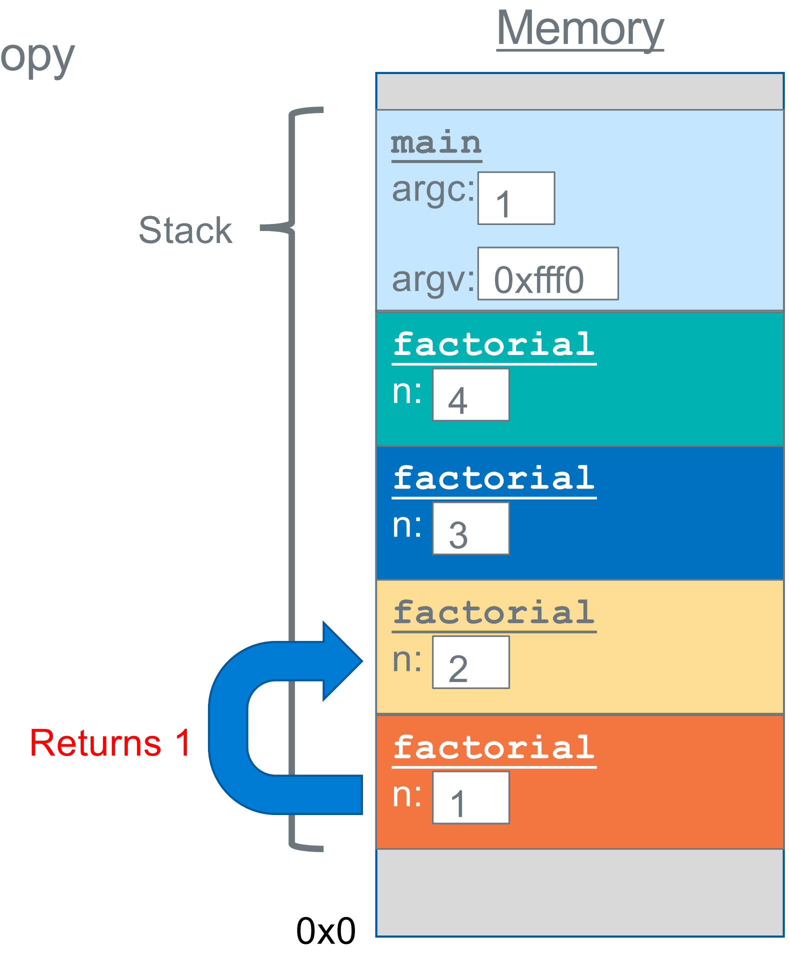
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

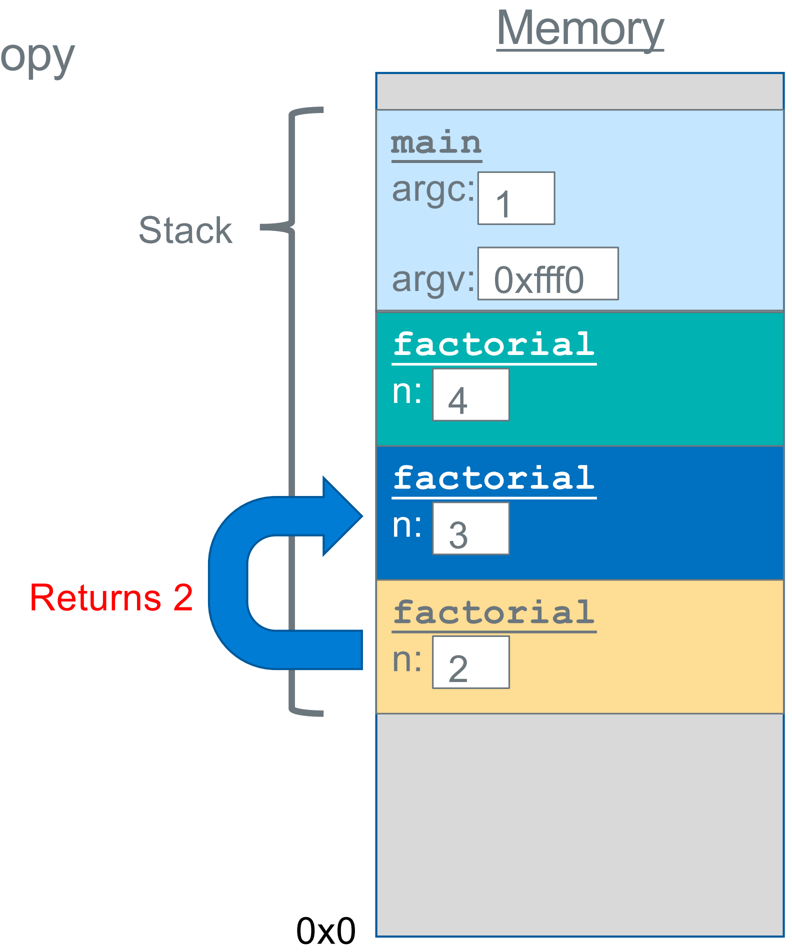
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

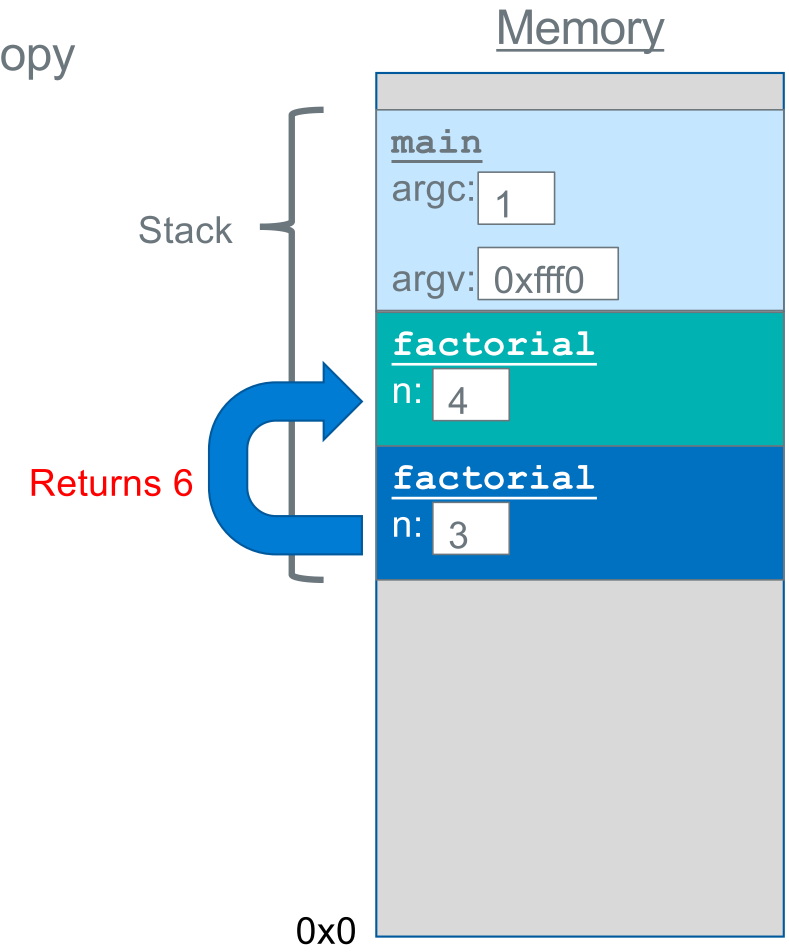
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

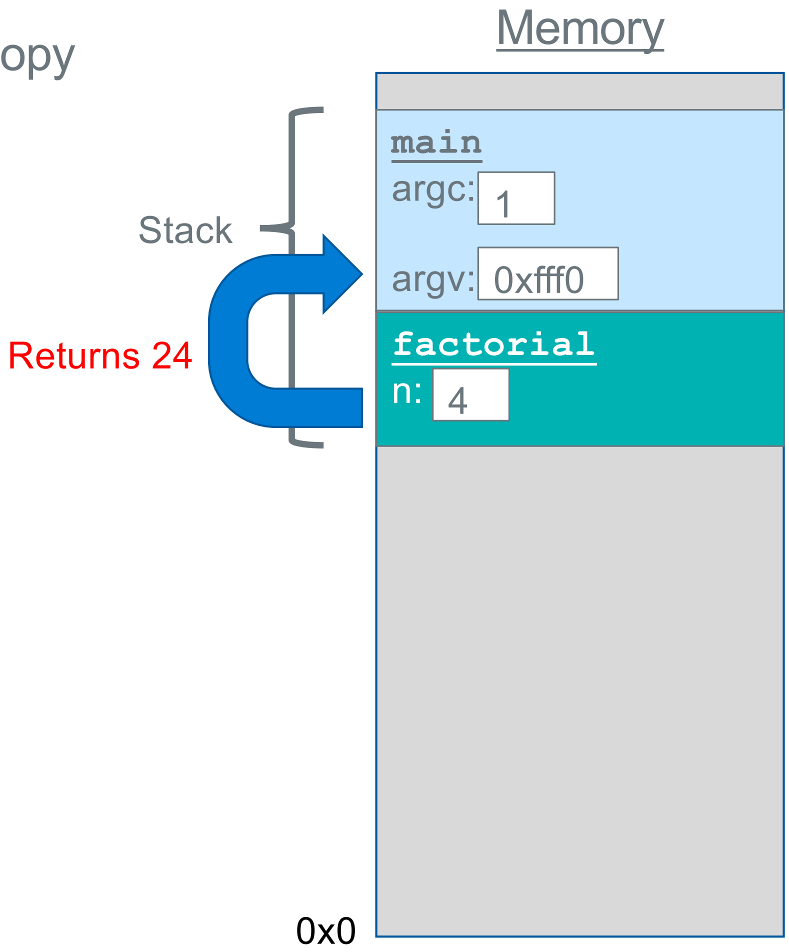
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

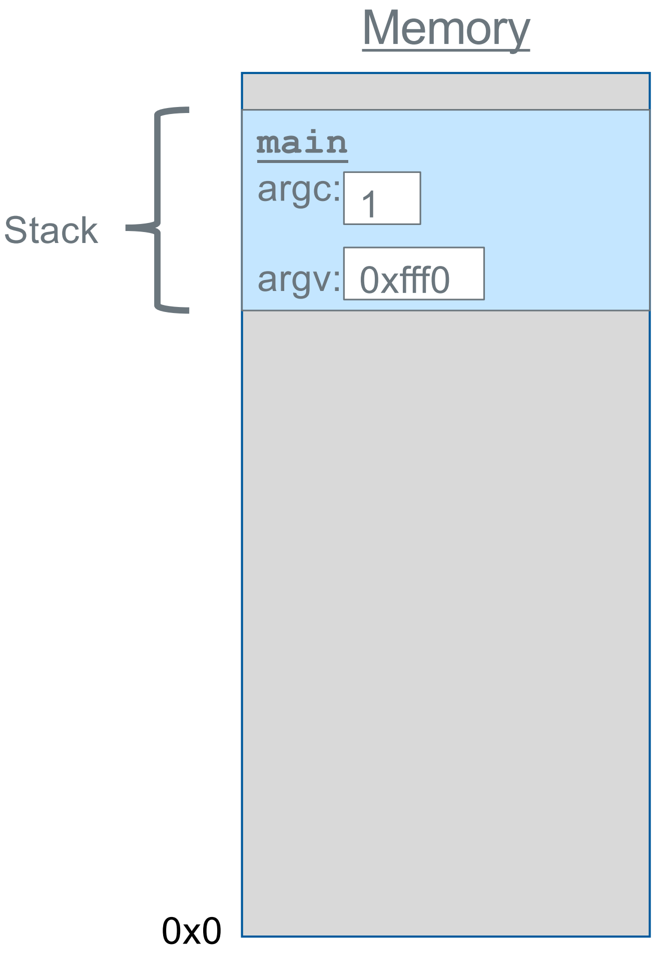
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

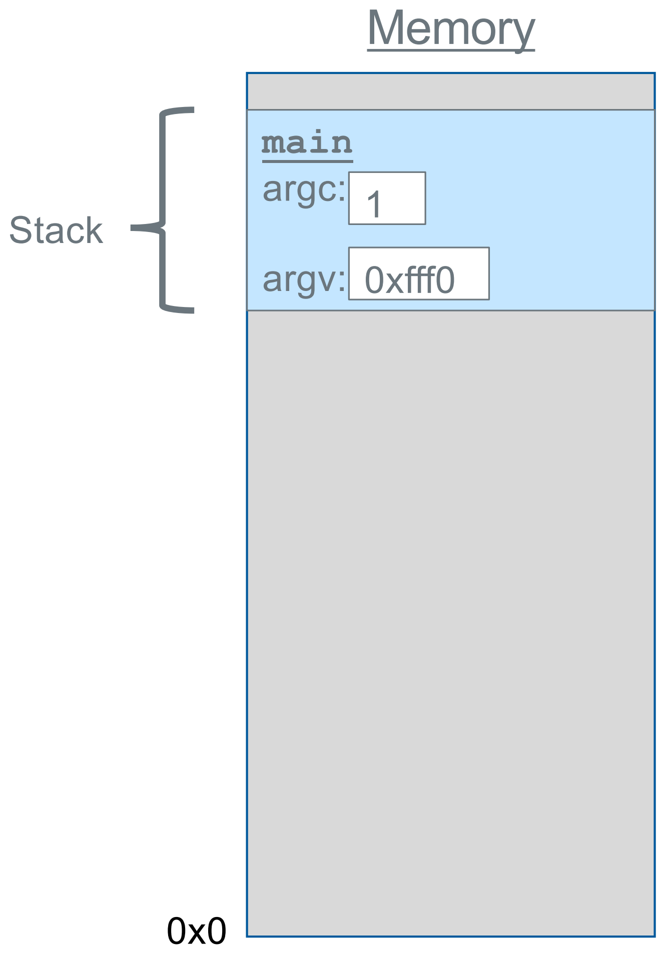
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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 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)**

  In CSE30 required use: `.size name, (. - name)`

x

Support For Function Calls and Function Call Return - 1

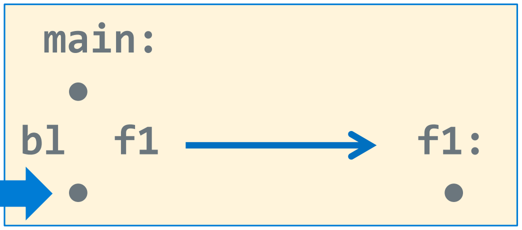
bl	imm24

Branch with Link **(function call)** instruction

```
bl label
```

- Function call to the instruction with the address `label` (no local labels for functions)
  - imm24 number of instructions from pc+8
- label **any function label** in the current file, or any function label that is defined as .global in any file that it is linked to
- BL **saves** the address of the instruction **immediately** following the **bl** instruction **in register lr** (link register is also known as r14)
- **The contents of the link register is the return address to the calling function**

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



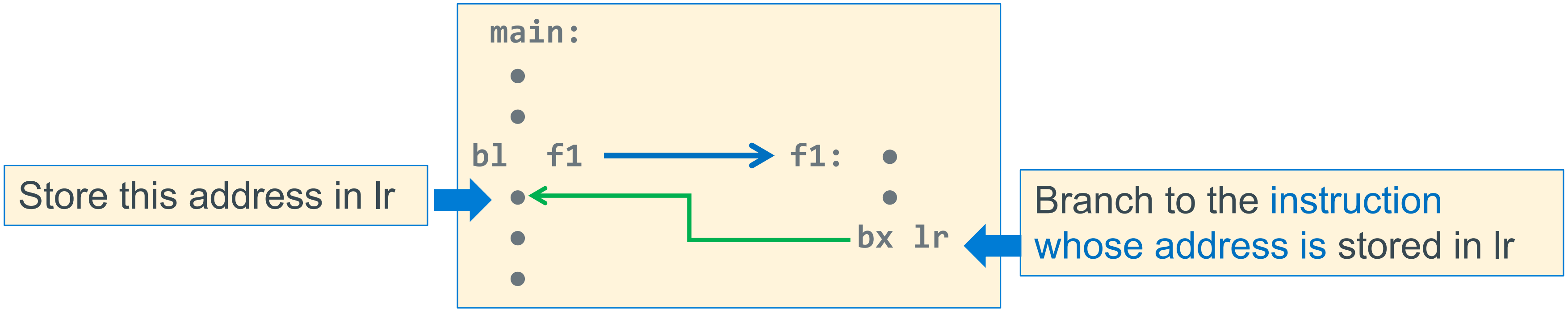
Support For Function Calls and Function Call Return - 2

bx	Rn

Branch & exchange (function return) instruction

```
bx lr
```

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



Store this address in lr

Branch to the instruction whose address is stored in lr

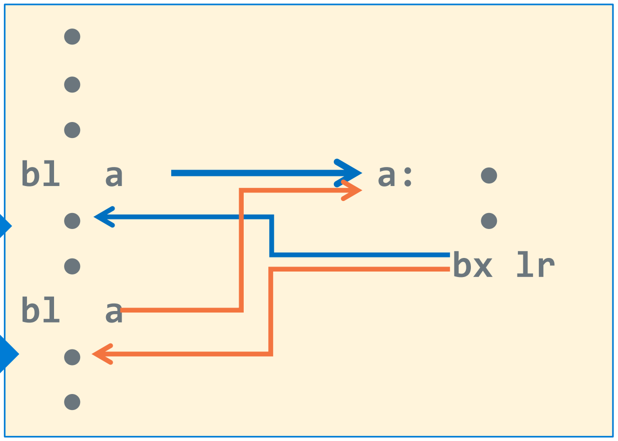
x

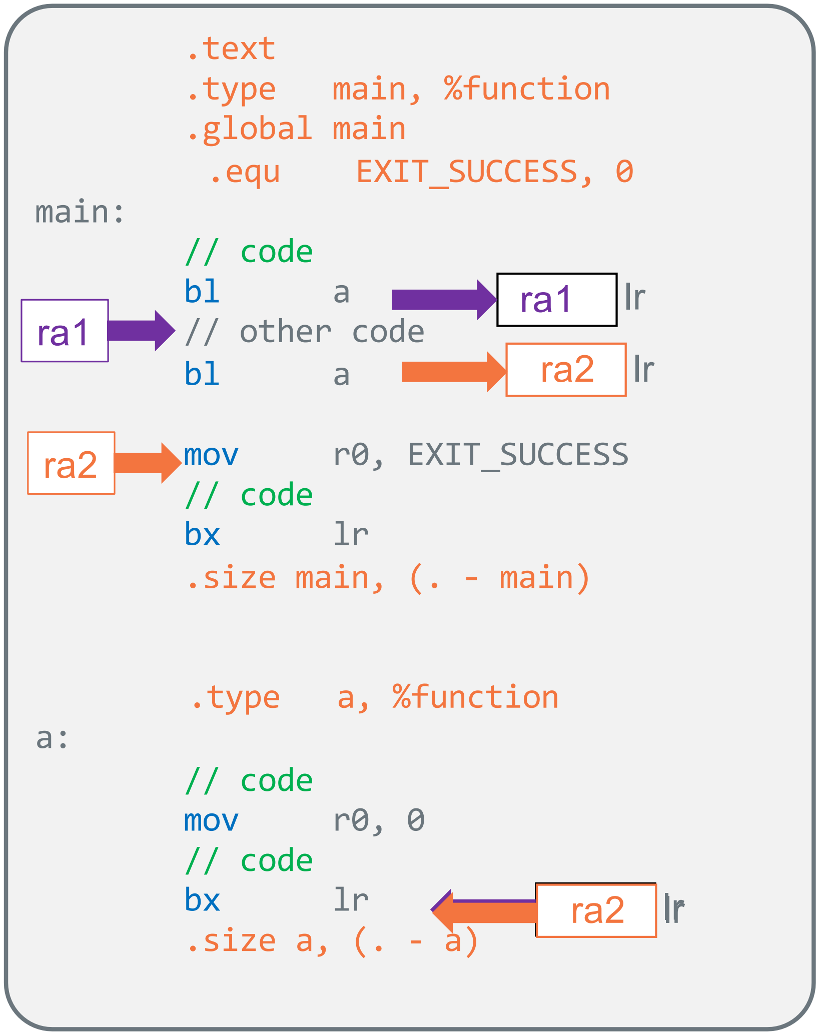
bl and bx operation working together

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

address of
next instruction
is stored in lr

address of
next instruction
is stored in lr





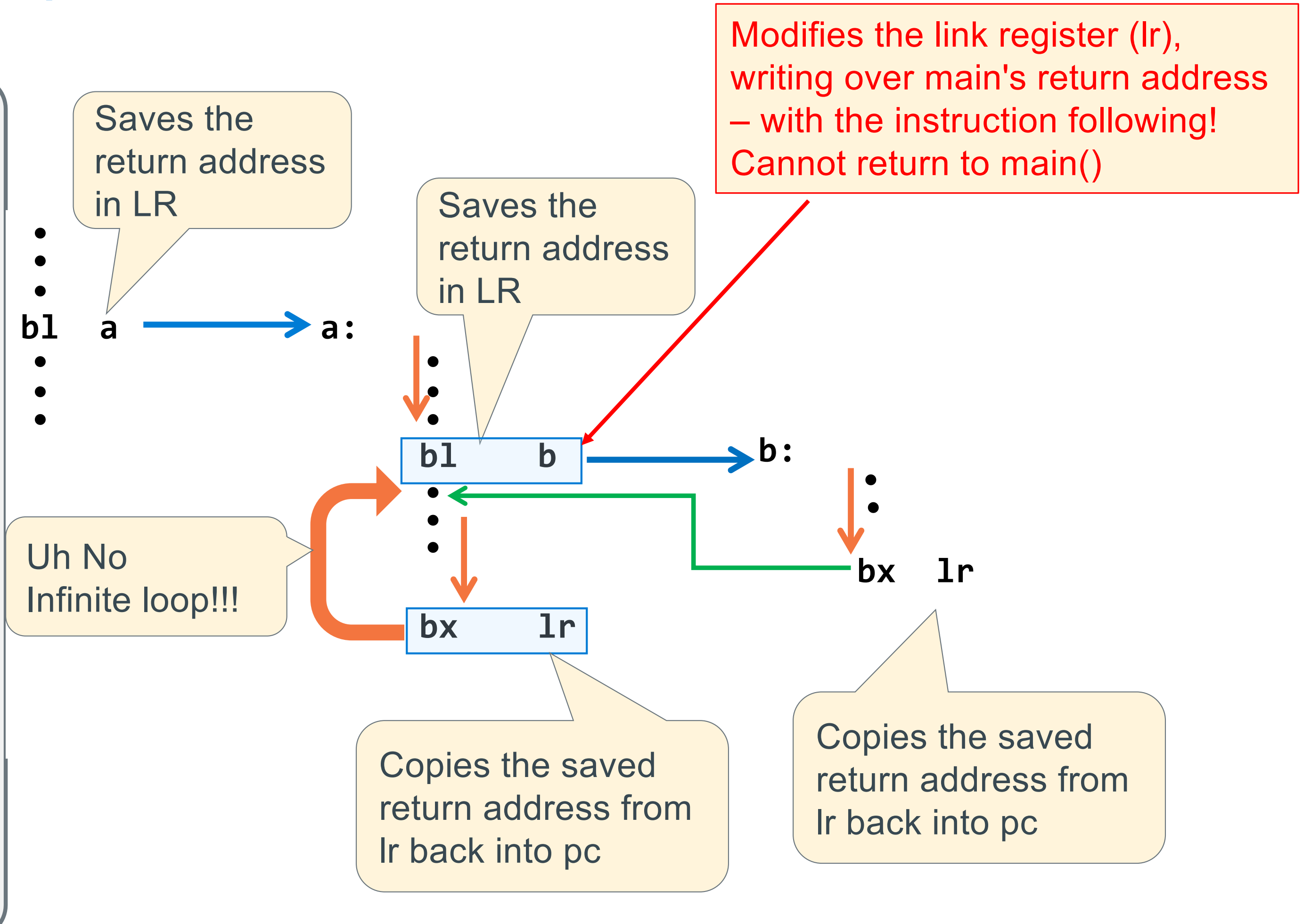
```
            .text
            .type   main, %function
            .global main
             .equ    EXIT_SUCCESS, 0
main:
            // code
            bl      a
            // other code
            bl      a

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


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

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

x

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

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

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

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



Saves the return address in LR

Saves the return address in LR

Modifies the link register (lr), writing over main's return address – with the instruction following! Cannot return to main()

Uh No Infinite loop!!!

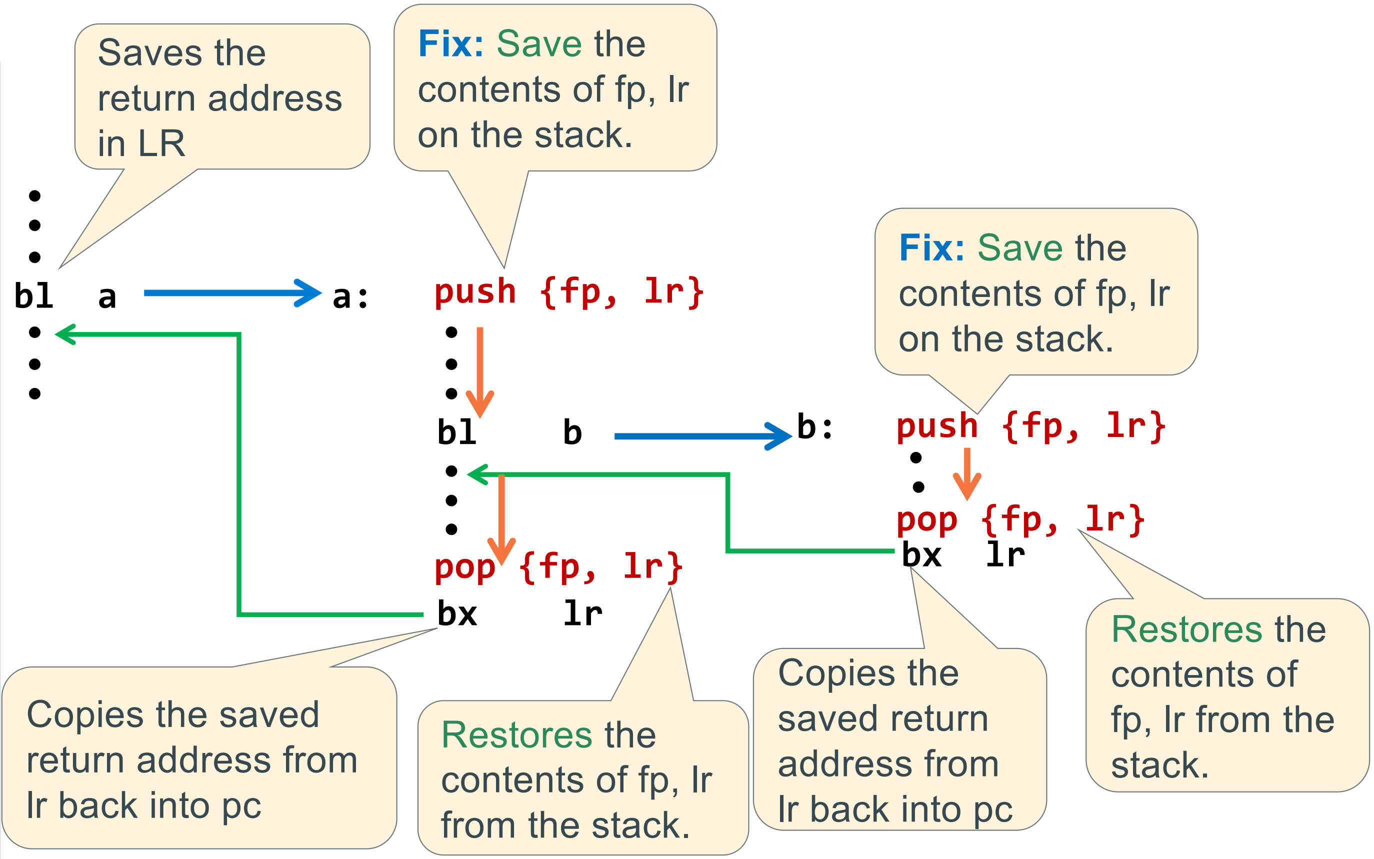
Copies the saved return address from lr back into pc

Copies the saved return address from lr back into pc

x

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

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

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

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



Saves the return address in LR

Fix: Save the contents of fp, lr on the stack.

Fix: Save the contents of fp, lr on the stack.

```
bl   a
```

```
a:   push {fp, lr}
     bl    b
     pop {fp, lr}
     bx    lr
```

```
b:   push {fp, lr}
     pop {fp, lr}
     bx   lr
```

Copies the saved return address from lr back into pc

Restores the contents of fp, lr from the stack.

Copies the saved return address from lr back into pc

Restores the contents of fp, lr from the stack.

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

x

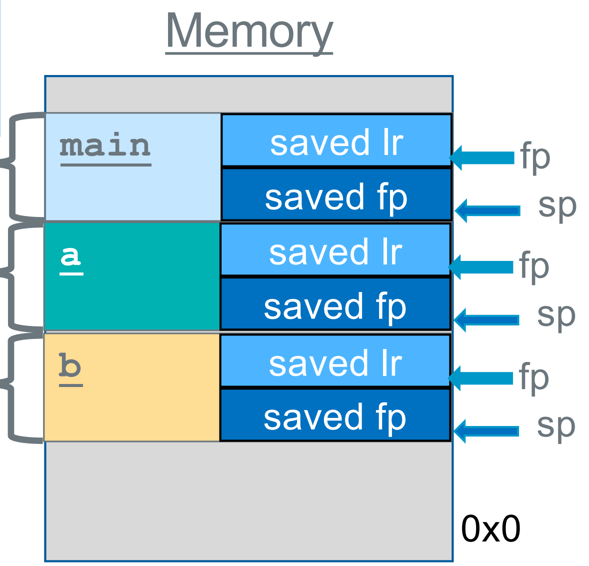
Minimal Stack Frame (Arm Arch32 Procedure Call Standards)

Requirements

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

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

- Function entry (Function **Prologue**):
  1. creates the frame (subtracts from sp)
  2. saves values
- Function return (Function **Epilogue**):
  1. restores values
  2. removes the frame (adds to sp)



We will see how the fp is used in a few slides

x

Review Return Value and Passing Parameters to Functions

(Four parameters or less)

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

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

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

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

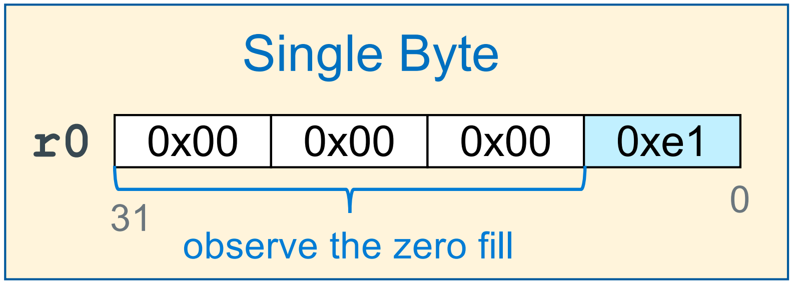
- Each **parameter and return value is limited to data that can fit in 4 bytes or less**
- You receive up to the first four parameters in these four registers
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes,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

Argument and Return Value Requirements

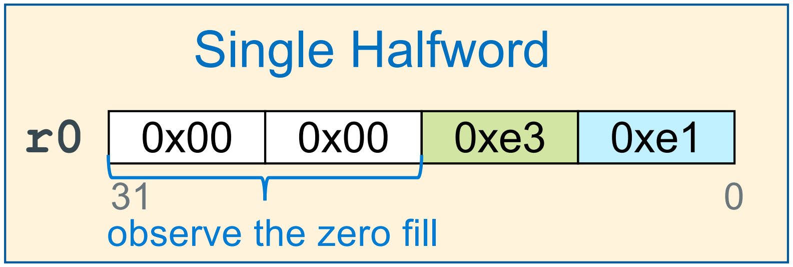
- When passing or returning values from a function you must do the following:

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

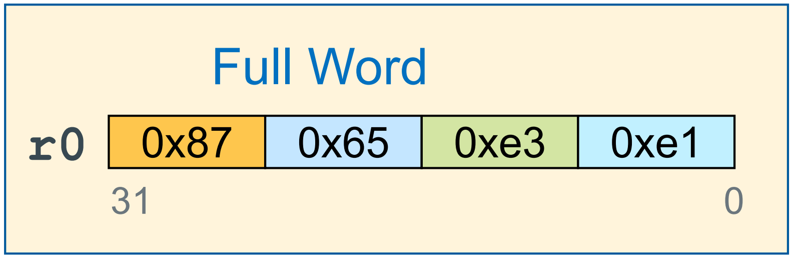
Single Byte



Single Halfword



Full Word



Preserved Registers: Protocols for Use

Register	*Function Call Use*	*Function Body Use*	*Save before use* *Restore before return*
r4-r10		contents preserved across function calls	Yes
r7	os system call number	contents preserved across function calls	Yes

- **Function Call Spec**:

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

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

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

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

x

Preserved Registers: When to Use?

Register	*Function Call Use*	*Function Body Use*	*Save before use* *Restore before return*
r4-r10		contents preserved across function calls	Yes
r7	os system call number	contents preserved across function calls	Yes

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

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

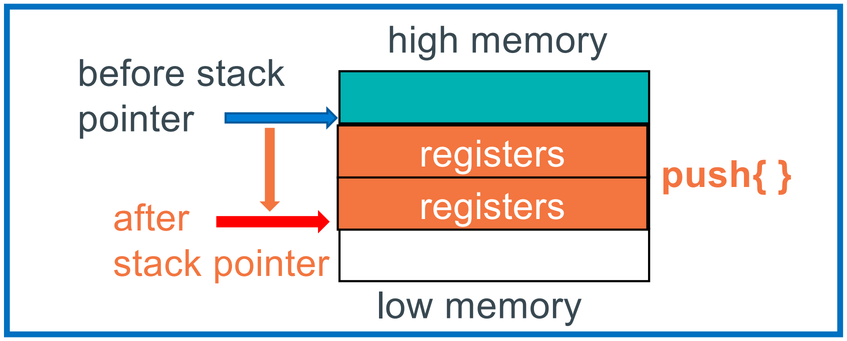
   Options are:
   a) preserved register *or*
   b) stack local variable (later slides)

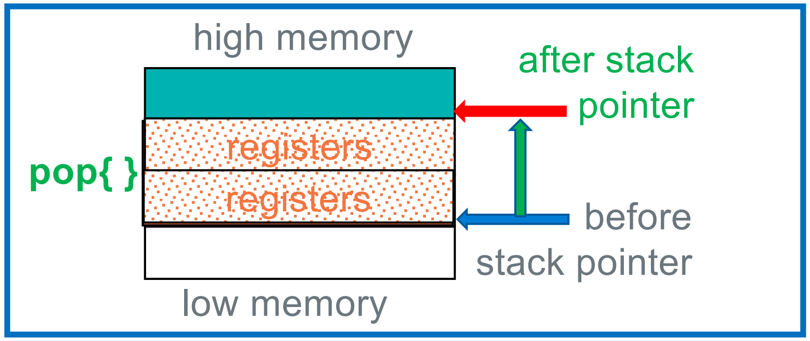
Preserving and Restoring Registers on the Stack
Used at Function entry and exit

Operation	***Pseudo Instruction (Use in CSE30)***	ARM instruction (reference only)	***Operation***

Push registers onto stack Function entry	push *{reg list}*	`stmfd sp!, {reg list}`	`sp ← sp - 4 × #registers`
			`Copy registers to mem[sp]`

Pop registers from stack Function Exit	pop *{reg list}*	`ldmfd sp!, {reg list}`	`Copy mem[sp] to registers,`
			`sp ← sp + 4 × #registers`





x

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

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

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

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

- Registers cannot be: (1) duplicated in the list, nor be (2) listed out of numeric order
- Register ranges can be specified `{r4, r5, r8-r11, fp, lr}`

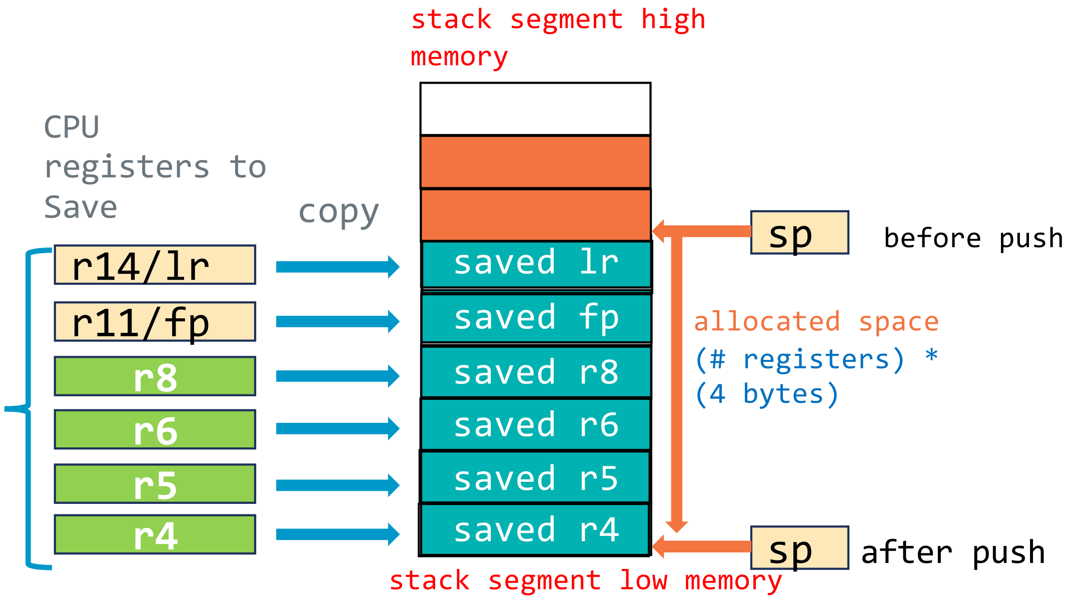
x

push: Multiple Register Save

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

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

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



stack segment high memory

CPU registers to Save

copy

r14/lr → saved lr

r11/fp → saved fp

r8 → saved r8

r6 → saved r6

r5 → saved r5

r4 → saved r4

sp before push

allocated space (# registers) * (4 bytes)

sp after push

stack segment low memory

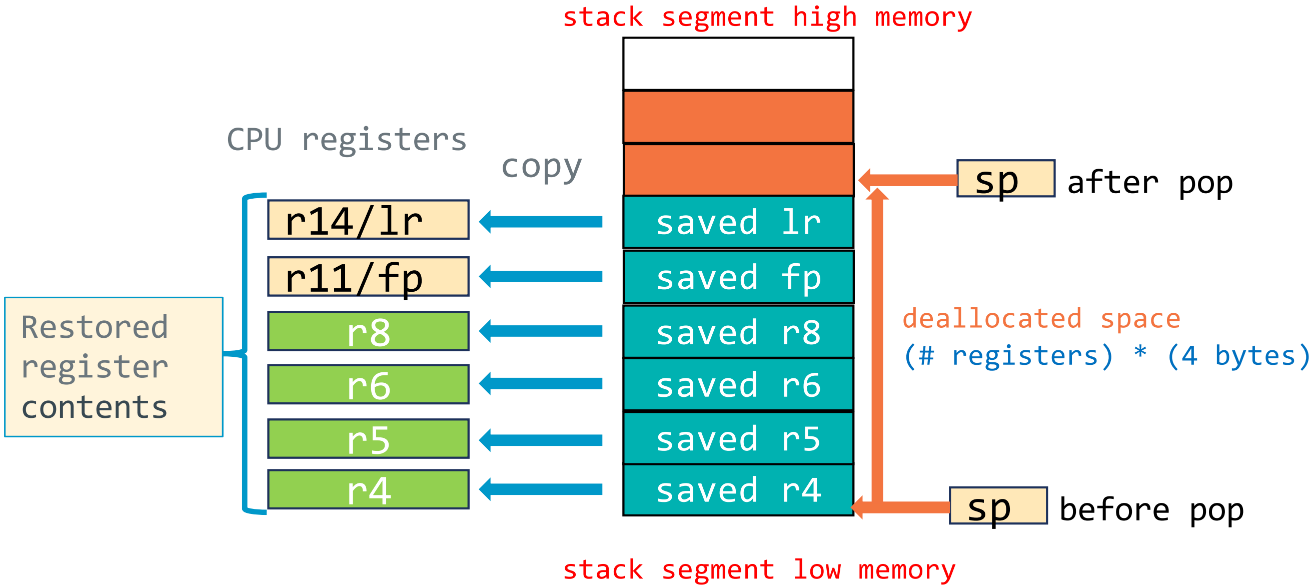
- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** **Also** subtracts (# of registers saved) * (4 bytes) from the **sp** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

x

pop: Multiple Register Restore

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

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

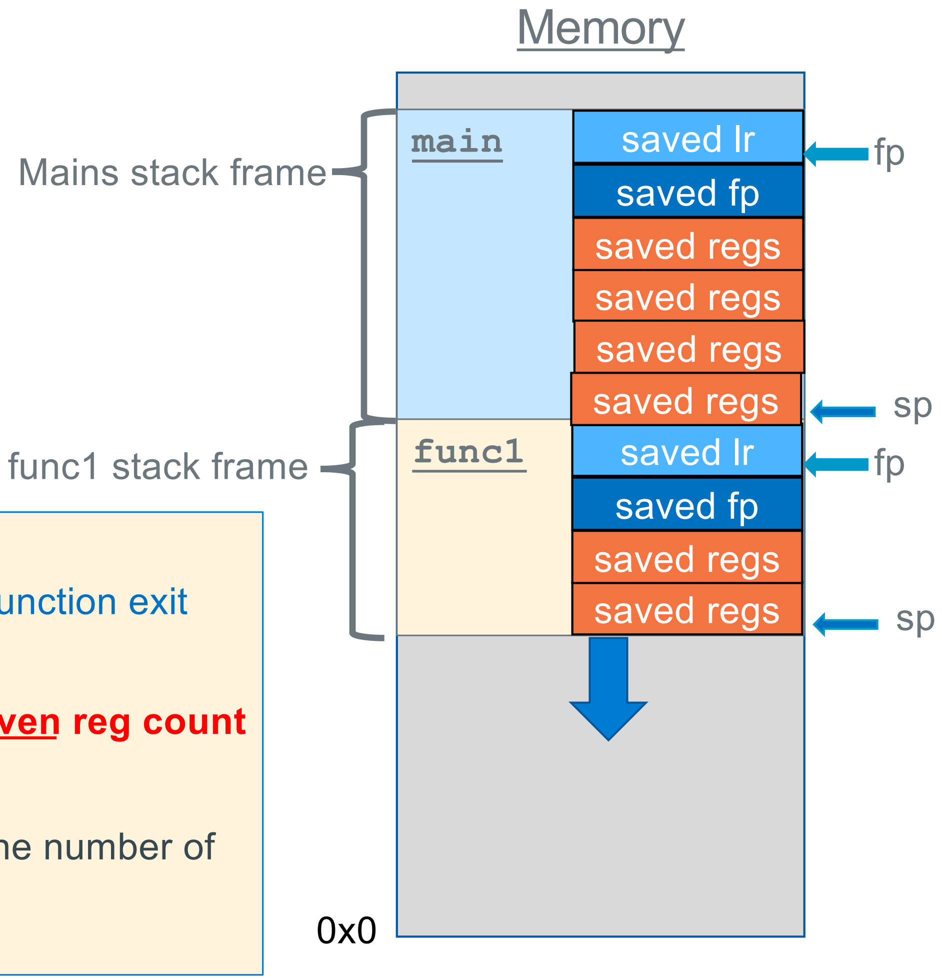


- `pop` copies the contents of stack segment memory to the `{reg list}`
- `pop` **adds:** (# of registers restored) * (4 bytes) to `sp` to ***deallocate*** space on the stack
  - sp = sp + (# registers restored * 4)
- **`Remember:`** `{reg list}` must be the same in both the `push` and the corresponding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

x

Basic Stack Frames (Arm Arch32 Procedure Call Standards)

```
void func1() {
    int c = 99;
}
int main(int argc, char *argv[])
{
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

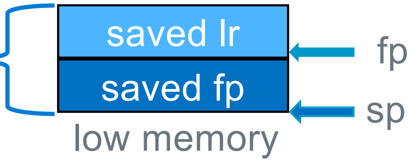


- **On each function call start (entry)**
  - Preserved registers: push at function entry and pop at function exit
- **Rules**
  - Keep `sp` 8-byte aligned strategy: `{reg list}` has an **even reg count**
  - **Remember** `fp` must always points at the saved `lr`
- Issue: number of registers saved on the stack varies with the number of registers in the `{reg list}`
  - So how do we always set `fp` properly?

Function Prologue and Epilogue: Minimum Stack Frame

- **Function prologue** creates stack frame
  1. push/save registers (**`lr`** & **`fp`** minimum) on stack
  2. set **`fp`** **`(add fp, …)`** to point at the saved lr as required for use by this function (later)
- **Function epilogue** removes stack frame
  1. set **`sp`** to where it was at the push (we may have **moved sp** to allocate space, later slides)
  2. pop/restore registers (**`lr`** & **`fp`** minimum) from stack
- In this example fp is 4 bytes from sp, (FP_OFF) but this will vary…

smallest frame has:
4 bytes between sp & fp



fp must point at saved lr

```
        .global one
        .type   one, %function
        .equ    FP_OFF,  4

one:
        push    {fp, lr}
        add     fp, sp, FP_OFF

       // your code

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

        bx      lr  // func return

       .size one, (. - one)
```

Function Header
Assembly directives

Position the fp

Function Prologue
always at top of function

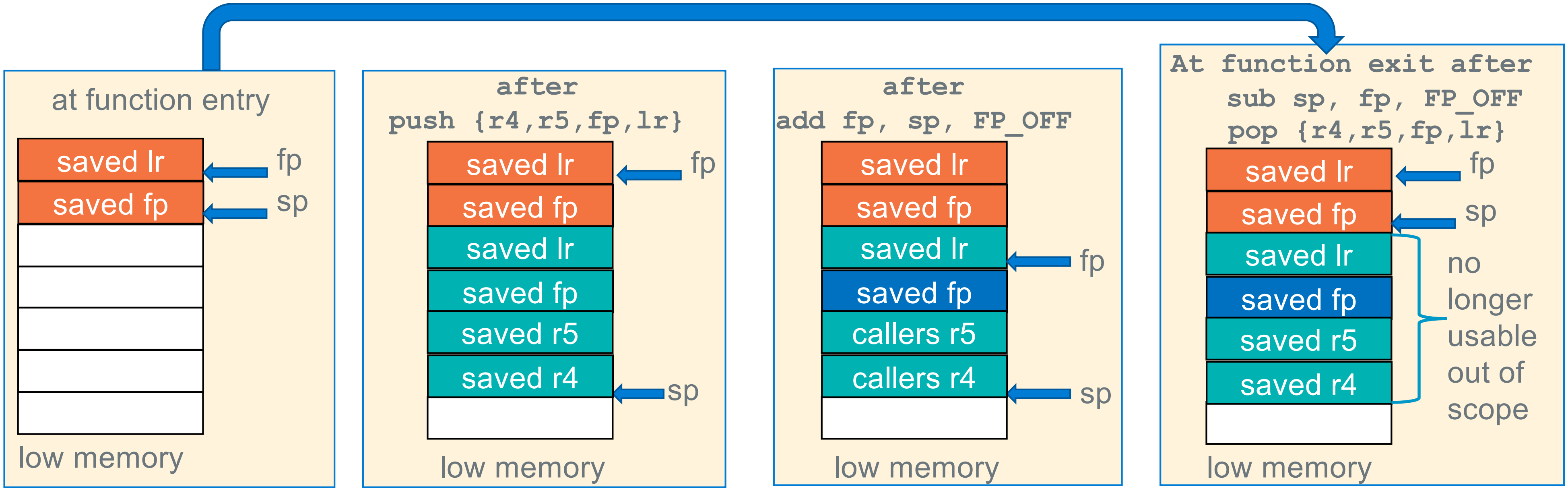
Restore the sp

Function Epilogue
always at bottom of function

Function Footer
Assembly directive

x

Saving/Restoring Preserved Registers At Function entry/exit



at function entry

saved lr
saved fp
fp
sp
low memory

```
after
push {r4,r5,fp,lr}
```

saved lr
saved fp
saved lr
saved fp
saved r5
saved r4
fp
sp
low memory

```
after
add fp, sp, FP_OFF
```

saved lr
saved fp
saved lr
saved fp
callers r5
callers r4
fp
sp
low memory

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

saved lr
saved fp
saved lr
saved fp
saved r5
saved r4
fp
sp
no longer usable out of scope
low memory

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

Function saves lr, fp using a push and only those preserved registers it wants to use on the stack

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

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

Part of function prologue

Part of function epilogue

x

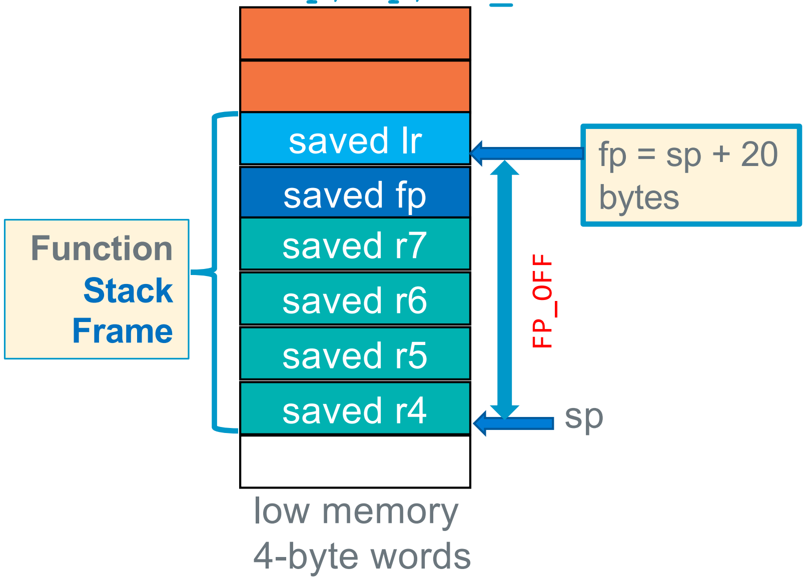
Setting FP_OFF: Distance from FP to SP

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

always at top of function saves regs and sets fp

always at bottom of function restores regs including the sp

after push {r4-r7,fp,lr}
add fp, sp, FP_OFF



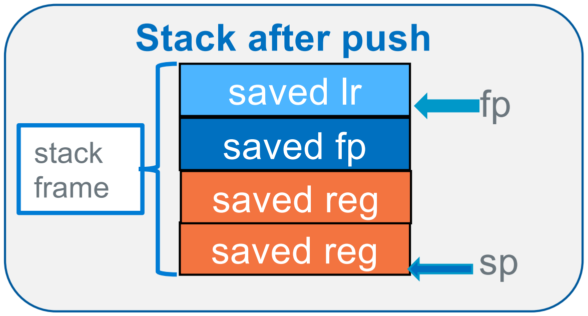
# regs saved	FP_OFF in Bytes
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

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

Where # regs = #preserved + lr + fp
```

▨ Means Caution, odd number of regs!
If odd number pushed, add pad to frame (later)
this must always be true: sp % 8 == 0

x

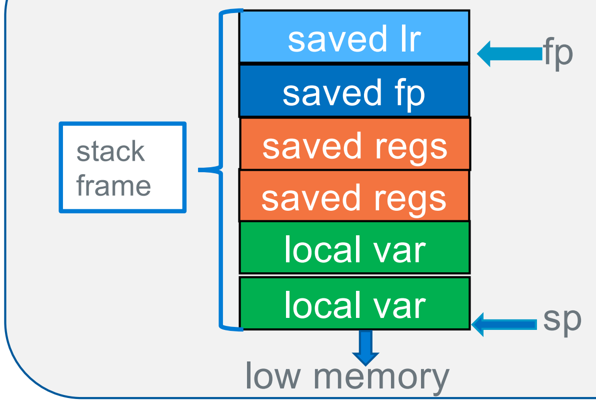
Why is there a `sub, fp, FP_OFF ?`



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

- As you will see, we will move the sp to allocate space on the stack for local variables and parameters, so for the pop to restore the registers correctly:
- sp must point at the last saved preserved register put on the stack bay the save register operation: the push

So we can add space for local variables!



```
 .equ     FRMSZ, 8
 push     {fp, lr}
 add      fp, sp, FP_OFF
 sub      sp, sp, FRMSZ
// your code

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

 bx       lr  // func return
```

- force the **sp** (using the **fp**) to contain the same address it had after the push operation **sub sp, fp, FP_OFF**

x

Stack Creation Overview

1. Calculate how much additional space is needed by local variables
2. **After the push, Subtract from the sp** the required byte count (+ padding - later slides)
3. If the variable has an initial value specified: add code to set the initial value
   a) mov and str are useful for initializing simple variables
   b) loops of mov and str for arrays

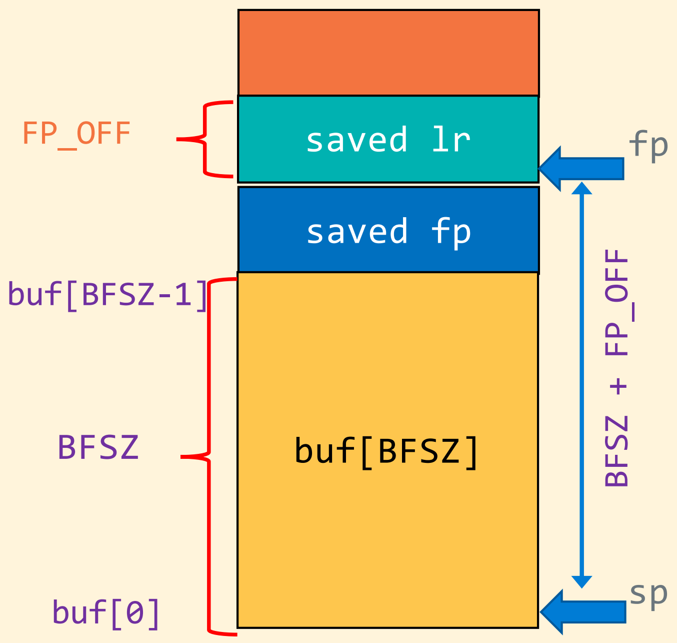
```
#define BFSZ 256
int main(void)
{
  char buf[BFSZ]; // BFSZ bytes
...
```

```
        .equ    FP_OFF, 4
        .equ    BFSZ, 256

main:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        sub     sp, sp, BFSZ
```

Function Prologue Extended

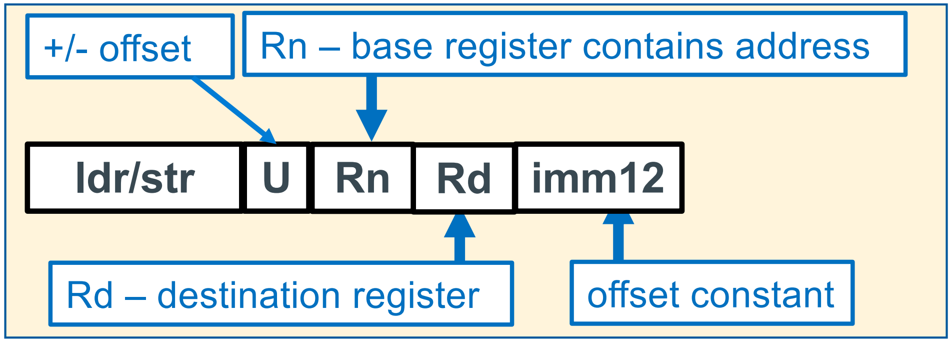
allocate space for buf[256]

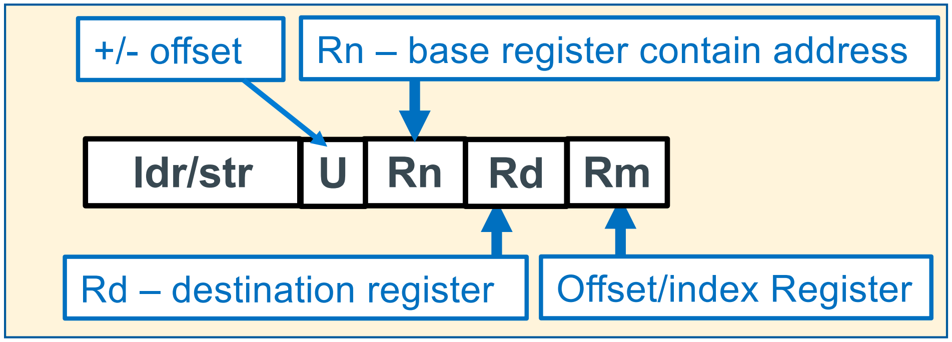
```
stack after allocating local space After
        sub sp, sp, BFSZ
```



Extra Slides

Reference: LDR/STR – Register To/From Memory Copy


+/- offset
Rn – base register contains address
ldr/str
U
Rn
Rd
imm12
Rd – destination register
offset constant



```
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

Data Segment Variable Alignment

```
.data
ch:      .byte 'A','B','C','D','E'
str:     .string "HIT"
ary:     .hword 0, 1
a:       .byte 'A'
b:       .byte 'B'
xx:      .word 2
```

```
% gcc -c -Wa,-ahlns al1.S
  1                .data
  2 0000 41424344 ch:     .byte 'A','B','C','D','E'
  2      45
  3 0005 48495400 str:    .string "HIT"
  4 0009 00000100 ary:    .hword 0, 1
  5 000d 41       a:      .byte 'A'
  6 000e 42       b:      .byte 'B'
  8 000f 02000000 xx:     .word 2
```

address contents

- Output on the right side is generated by:
- %gcc -c -Wa,-ahlns al1.S

```
.data
xx:      .word 2
ch:      .byte 'A','B','C','D','E'
         .align 2
str:     .string "HI"
         .align 1
ary:     .hword 0, 1
a:       .byte 'A'
b:       .byte 'B'
```

```
gcc -c -Wa,-ahlns al1.S
  1                .data
  2 0000 02000000 xx:     .word 2
  3 0004 41424344 ch:     .byte 'A','B','C','D','E'
  3      45
  4 0009 000000           .align 2
  5 000c 484900   str:    .string "HI"
  6 000f 00               .align 1
  7 0010 00000100 ary:    .hword 0, 1
  8 0014 41       a:      .byte 'A'
  9 0015 42       b:      .byte 'B'
```

x

Literal Table (Array) each entry is a pointer to a different Label

- **Assembler automatically inserts into the text** segment an array (table) of pointers
- Each entry contains a 32-bit address of one of the labels
- Uses r15 (PC) as base register to load the entry into a reg

displacement (bytes) - 8

The assembler creates this table before generating the .o file

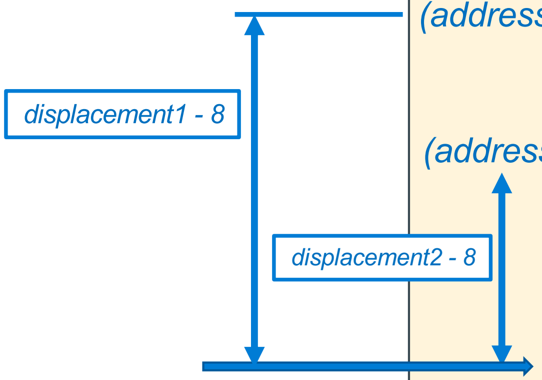
```
        .bss
y:      .space 4
         .data
x:      .word 200
         .section .rodata
.Lmsg:  .string "Hello World"
        .text
main:

(address)ldr r0, [PC, displacement]  // replaces:  ldr r0, =y

      <last line of your assembly, typically a function return>

     .word  y       // entry #1 32-bit address for y
     .word  x       // entry #2 32-bit address for x
     .word  .Lmesg  // entry #3 32-bit address for .Lmesg
```

X

Literal Table (Array) each entry is a pointer to a different Label

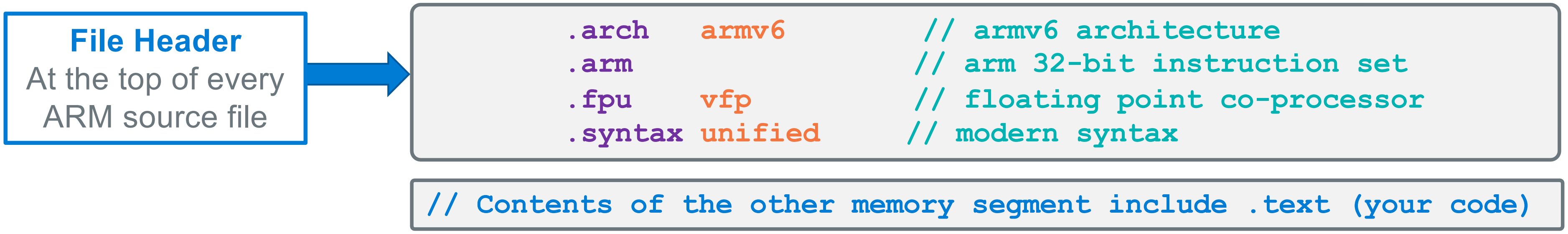
The displacement is different for each use.
As the PC is different at each instruction

```
        .bss
y:      .space 4
         .data
x:      .word 200
         .section .rodata
.Lmsg:  .string "Hello World"
         .text
main:
(address)ldr r0, [PC, displacement1]  // replaces:  ldr r0, =y


(address)ldr r0, [PC, displacement2]  // replaces:  ldr r0, =y

         <last line of your assembly, typically a function return>

        .word  y       // entry #1 32-bit address for y
        .word  x       // entry #2 32-bit address for x
        .word  .Lmesg  // entry #3 32-bit address for .Lmesg
```



x

ARM Assembly Source File: Header



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)
```

`.arch <architecture>`

- Specifies the target architecture to generate machine code
- Typically specify oldest ARM arch you want the code to run on – most arm CPUs are backwards compatible

`.arm`

- Use the 32-bit ARM instructions, There is an alternative 16-bit instruction set called thumb that we will not be using

`.fpu <version>`

- Specify which floating point co-processor instructions to use (OPTIONAL we will not be using floating point)

x

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)**

  In CSE30 required use: `.size name, (. - name)`

Reference For PA8/9: C Stream Functions Opening Files

```
FILE *fopen(char filename[], const char mode[]);
```

- Opens a stream to the specified file in specified file access mode
  - returns NULL on failure – always check the return value; make sure the open succeeded!
- Mode is a string that describes the actions that can be performed on the stream:

"r" Open for reading.

The stream is positioned at the beginning of the file. Fail if the file does not exist.

"w" Open for writing.

The stream is positioned at the beginning of the file. Create the file if it does not exist.

"a" Open for writing.

The stream is positioned at the end of the file. Create the file if it does not exist.

Subsequent writes to the file will always be at current end of file.

- An optional "+" following "r", "w", or "a" opens the file for both reading and writing

x

Reference: C Stream Functions Closing Files and Usage

```
int fclose(FILE *stream);
```

- Closes the specified stream, forcing output to complete (eventually)
  - returns EOF on failure (often ignored as no easy recovery other than a message)

- Usage template for **`fopen()`** and **`fclose()`**
  1. Open a file with **`fopen()`** **always** checking the return value
  2. do i/o – keep calling stdio io routines
  3. close the file with **`fclose()`** when done with that I/O stream

C Stream Functions Array/block read/write

- These do not process contents they simply **transfer** a fixed number of bytes to and from a buffer passed to them
- `size_t fwrite(void *ptr, size_t size, size_t count, FILE *stream);`
  - Writes an array of *count* ***elements*** of ***size*** bytes from **stream**
  - *Updates the write file pointer forward by the number of bytes written*
  - returns number of elements written
  - error is short element count or 0

- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream);`
  - Reads an array of ***count elements*** of ***size*** bytes from *stream*
  - *Updates the read file pointer forward by the number of bytes read*
  - returns number of elements read, EOF is a return of 0
  - error is short element count or 0
- **I almost always set size to 1 to return bytes read/written**

x

C fread/fwrite Example - 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BFSZ      8192 /* size of read */
int main(void)
{
  char fbuf[BFSZ];
  FILE *fin, *fout;
  size_t readlen;
  size_t bytes_copied = 0;
  retval = EXIT_SUCCESS;

  if (argc != 3){
    fprintf(stderr, "%s requires two args\n", argv[0]);
    return EXIT_FAILURE;
  }
  /* Open the input file for read */
  if ((fin = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"fopen for read failed\n");
    return EXIT_FAILURE;
  }
  /*  Open the output file for write */
  if ((fout = fopen(argv[2], "w") == NULL) {
    fprintf(stderr, "fopen for write failed\n");
    fclose(fin);
    return EXIT_FAILURE;
  }
```

To handle bytes moved

```
% ls -ls ZZZ
ls: ZZZ: No such file or directory
% ./a.out cp.c ZZZ
bytes copied: 1122
% ls -ls cp.c ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:51 ZZZ
8 -rw-r--r--  1 kmuller  staff  1122 Jul  2 08:49 cp.c
```

x

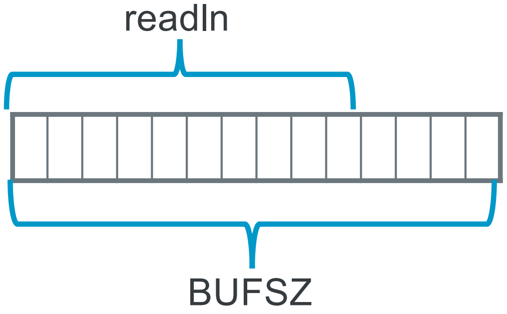
C fread/fwrite Example - 2

```
  /* Read from the file, write to fout */

  while ((readlen = fread(fbuf, 1, BUFSIZ, fin)) > 0) {

    if (fwrite(fbuf, 1, readlen, fout) != readlen) {
        fprintf(stderr, "write failed\n");
        retval =  EXIT_FAILURE;
        break;
    }
    bytes_copied += readlen; //running sum bytes copied
  }

  if (retval == EXIT_FAILURE)
    printf("Failure Copy did not complete only ");
  printf("Bytes copied: %zu\n", bytes_copied);

  fclose(fin);
  fclose(fout);

  return retval;
}
```

By using an element size of 1 with a char buffer, this is byte I/O

Capture the bytes read so you know how many bytes to write

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



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

x

putchar/getchar
Setting up and Usage

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n”, count);
    return EXIT_SUCCESS;
}
```

r0 r1

```
        .extern getchar
        .extern putchar
        .section .rodata
.Lfstr: .string  "Echo count: %d\n"
        .text
        .equ    EOF,          -1
        .type   main, %function
        .global main
        .equ    FP_OFF,    12
        .equ    EXIT_SUCCESS, 0
main:   push    {r4, r5, fp, lr}
        add     fp, sp, FP_OFF
        mov     r4, 0  //r4 = count

/* while loop code will go here */
.Ldone:
        mov     r1, r4 // count
        ldr     r0, =.Lfstr
        bl      printf
        mov     r0, EXIT_SUCCESS
        sub     sp, fp, FP_OFF
        pop     {r4, r5, fp, lr}
        bx      lr
        .size main, (. - main)
```

Putchar/getchar:
The while loop

```c
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    int c;
    int count = 0;

    while ((c = getchar()) != EOF) {
        putchar(c);
        count++;
    }
    printf("Echo count: %d\n”, count);
    return EXIT_SUCCESS;
}
```

```
        mov     r4, 0  //count
        bl      getchar
        cmp     r0, EOF
        beq     .Ldone
.Lloop:
        bl      putchar
        bl      getchar
        add     r4, r4, 1
        cmp     r0, EOF
        bne     .Lloop
.Ldone:
        mov     r1, r4
        ldr     r0, =pfstr
        bl      printf
```

initialize count

pre loop test with a call to getchar()
if it returns EOF in r0 we are done

echo the character read with getchar and then read another and increment count

did getchar() return EOF if not loop

saw EOF, print count

File header and footers are not shown

x

printing error messages in assembly

```
.Lmsg0: .string "Read failed\n"
        ldr     r0, =.Lmsg0                  // read failed print error
        bl      errmsg
```

```
        // int errmsg(char *errormsg)
        // writes error messages to stderr
        .type   errmsg, %function               // define to be a function
        .equ    FP_OFF,         4               // fp offset in stack frame
errmsg:
        push    {fp, lr}                        // stack frame register save
        add     fp, sp, FP_OFF                  // set the frame pointer

        mov     r1, r0
        ldr     r0, =stderr
        ldr     r0, [r0]
        bl      fprintf
        mov     r0, EXIT_FAILURE                // Set return value
        sub     sp, fp, FP_OFF                  // restore stack frame top
        pop     {fp, lr}                        // remove frame and restore
        bx      lr                              // return to caller
        // function footer
        .size   errmsg, (. - errmsg)            // set size for function
```