Version 1.04

UCSD CSE 30

Aarch32 Assembly –Load & Store, Bit Ops, Functions

Week 8

Lecture 22

Keith Muller

Data Segment Alignments
.bss, .data and ,section .rodata

```
        .data               // al1.S
ch:     .byte 'A','B','C','D','E'
ary:    .hword 0x0000, 0x0001
x:      .word 0x00000037
str:    .string "HI"
```

```
% gcc -c -Wa,-ahlns al1.S
   1                   .data
   2 0000 41424344 ch: .byte 'A','B','C','D','E'
   2      45
   3 0005 00000100 ary: .hword 0x0000, 0x0001
   4 0009 37000000 x: .word 0x0037
   5 000d 484900   str: .string "HI"
```

not half word aligned

not word aligned

address

contents (lsb to msb order!)

1000	0x00
0111	0x01
0110	0x00
0101	0x00
0100	0x45
0011	0x44
0010	0x43
0001	0x42
0000	0x41

low address bytes

```
        .data               // al1.S
ch:     .byte 'A','B','C','D','E'
        .align 1
ary:    .hword 0x0000, 0x0001
        .align 2
x:      .word 0x00000037
str:    .string "HI"
```

```
gcc -c -Wa,-ahlns al1.S
   1                   .data
   2 0000 41424344 ch: .byte 'A','B','C','D','E'
   2      45
   3 0005 00           .align 1  // pad
   4 0006 00000100 ary: .hword 0x0000, 0x0001
   5 000a 0000         .align 2  // pad not shown
   6 000c 37000000 x: .word 0x00000037
   7 0010 484900   str: .string "HI"
```

half word aligned

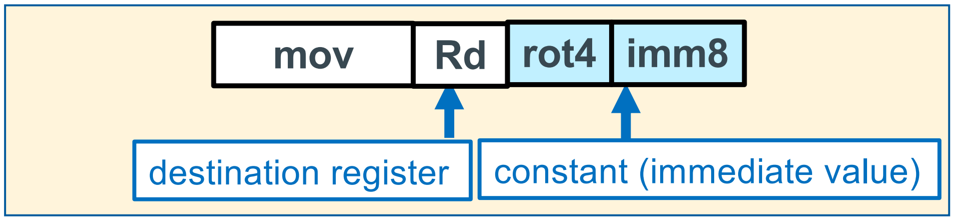
word aligned

1001	0x00
1000	0x01
0111	0x00
0110	0x00
0101	0x00
0100	0x45
0011	0x44
0010	0x43
0001	0x42
0000	0x41

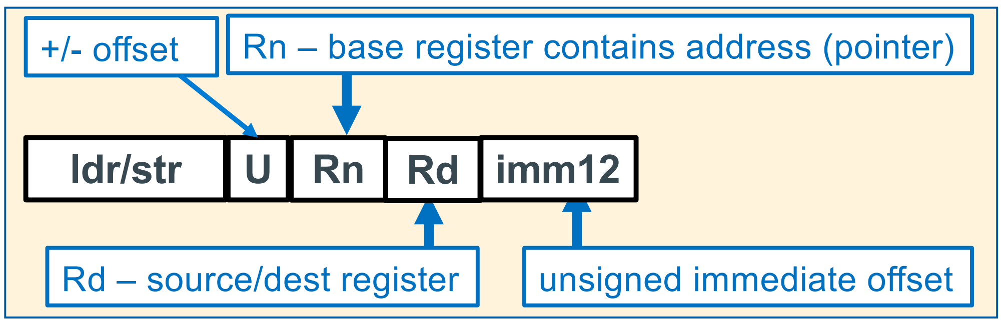
x

Review From Earlier week: How to Access Memory?

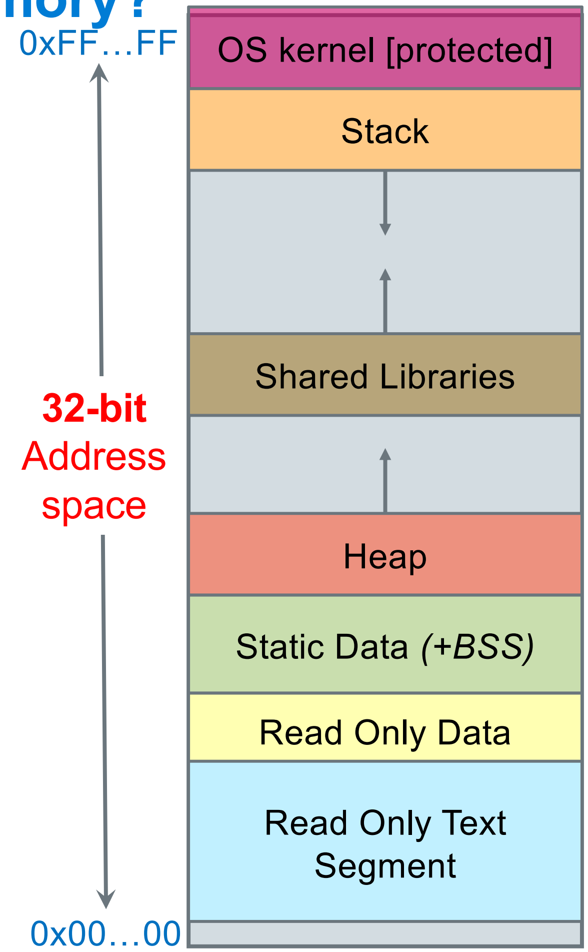
- Address space is 32 bits wide – POINTERS in registers



rot4/imm8 is too small



Even if you changed the instruction to reuse the base register bits (4 bits) + imm12 to get 16-bits, it is still too small!



x

How to Access variables in a Data Segment

- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
  - Each entry contains a **32-bit Label address**
  - The table is located within than 12 bits of address offset from the instruction being executed (the PC has the address of the current instruction + 8 in it), so r15 is the base register
- Tell the assembler to use a literal table to get the address of a label into a register:

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

to **load** a **memory** variable

1. load the pointer
2. read (load) from the pointer

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

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

assembly source file ex.S

```
        .bss
y:      .space 4

        .data
x:      .word 200

         .section .rodata
.Lmsg:  .string "Hello World"

         .text
        // function header
main:

        // load the contents into r2
        ldr r2, =x     // int *r2 = &x
        ldr r2, [r2]   // r2 = *r2;
        // &x was only needed once above

        // store the contents of r0
        ldr r1, =y     // r1 = &y
        str r0, [r1]   // y = r0
        // keeping &y in r1 above
…
```


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
```

displacement1 - 8

displacement2 - 8

x

Using ldr for immediate values to big for mov, add, sub, and, etc

- 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

Loading and Storing: Variations List

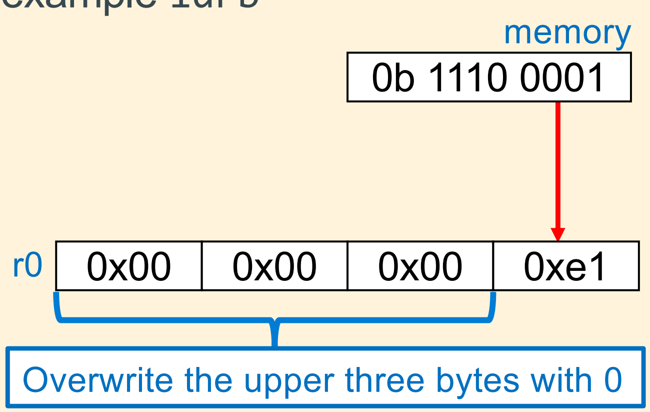
- Load and store have variations that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will set the upper bits not filled from memory differently depending on which variation of the load instruction is used
- Store will only select the lower 8-bit, lower 16-bits or all 32-bits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

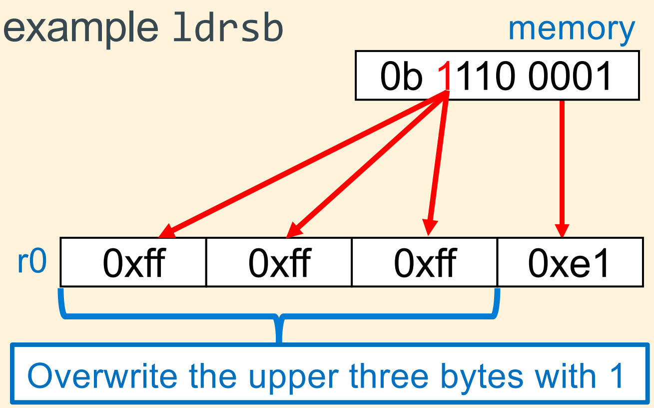
Instruction	Meaning	Sign Extension	Memory Address Requirement
`ldrsb`	load signed byte	sign extension	none (any byte)
`ldrb`	load unsigned byte	zero fill (extension)	none (any byte)
`ldrsh`	load signed halfword	sign extension	halfword (2-byte aligned)
`ldrh`	load unsigned halfword	zero fill (extension)	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	Signed (2's complement)
Zero-Extend: Add leading 0's	Sign-Extend: Replicate sign bit
example `ldrb`	example `ldrsb`



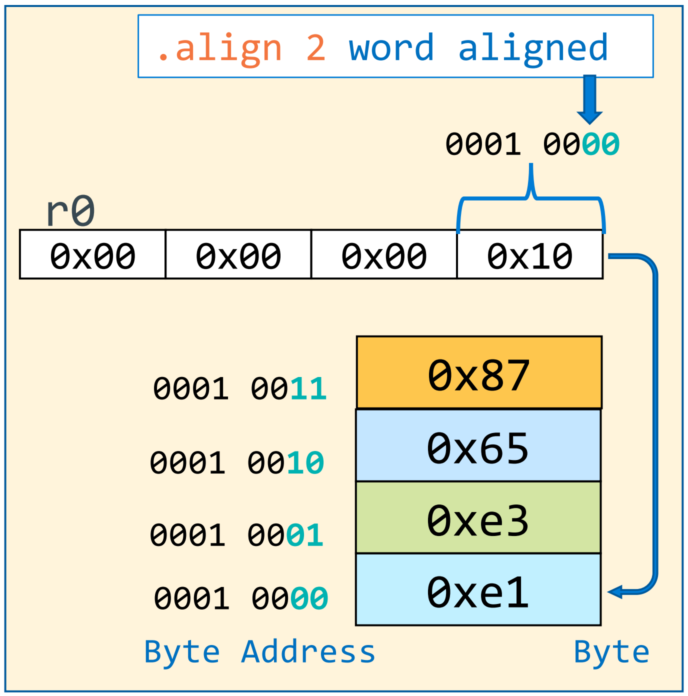


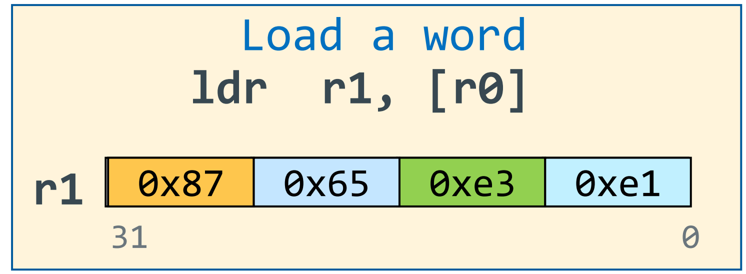
Instructions that zero-extend: ldrb, ldrh

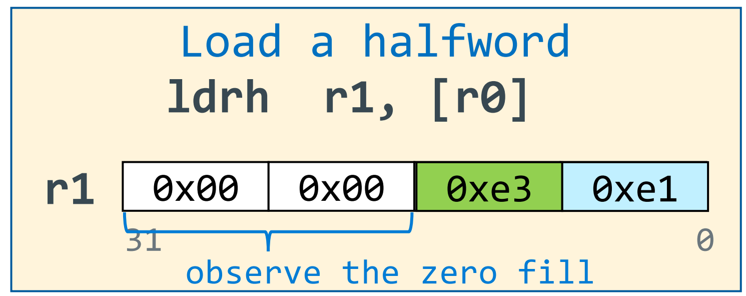
Instructions that sign-extend: ldrsb, ldrsh

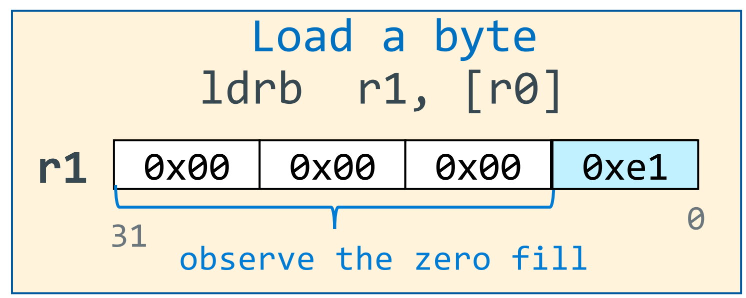
x

Load a Byte, Half-word, Word



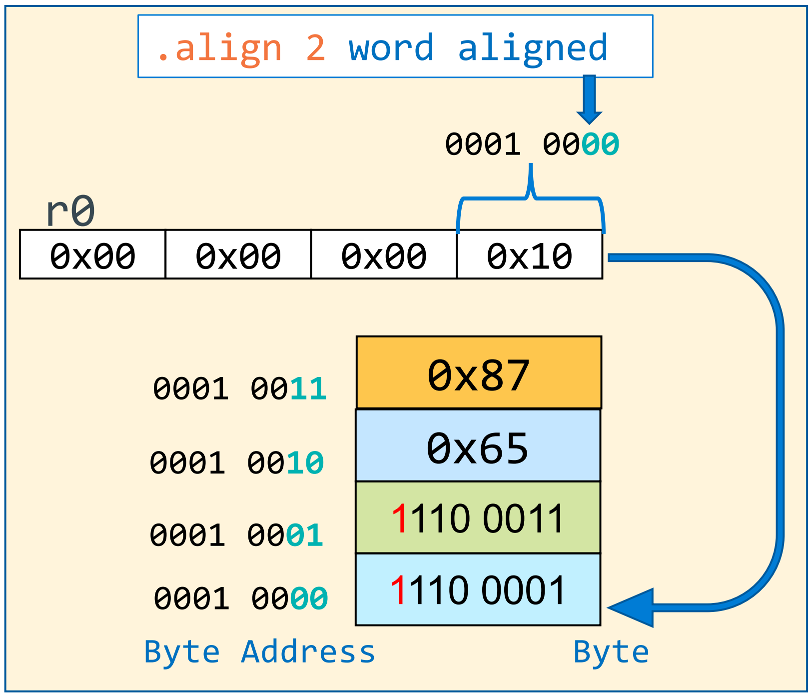


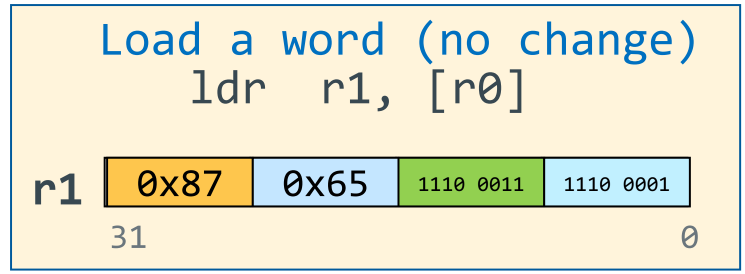


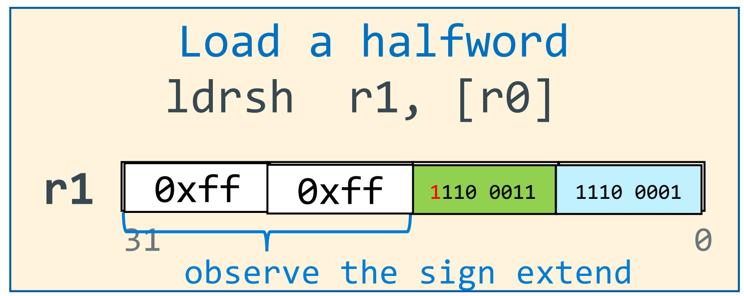


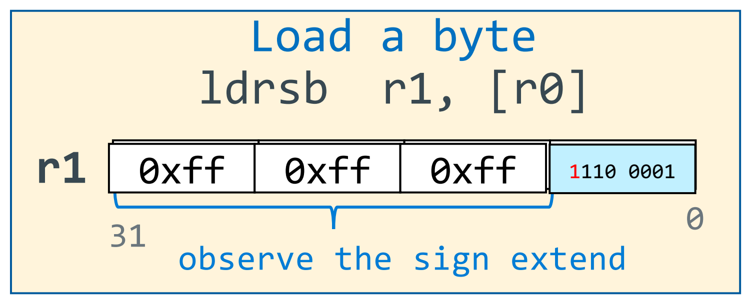
x

Signed Load a Byte, Half-word, Word

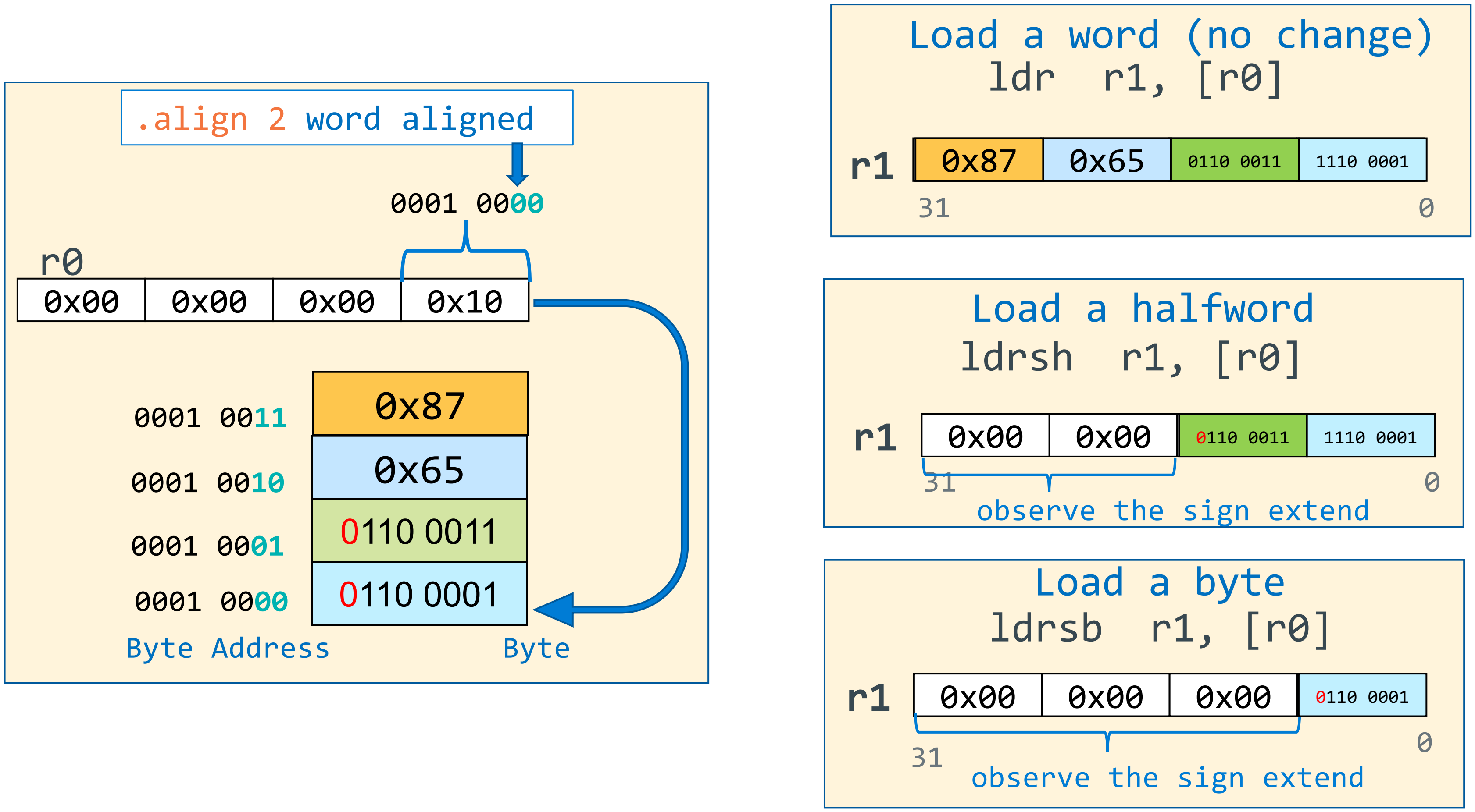






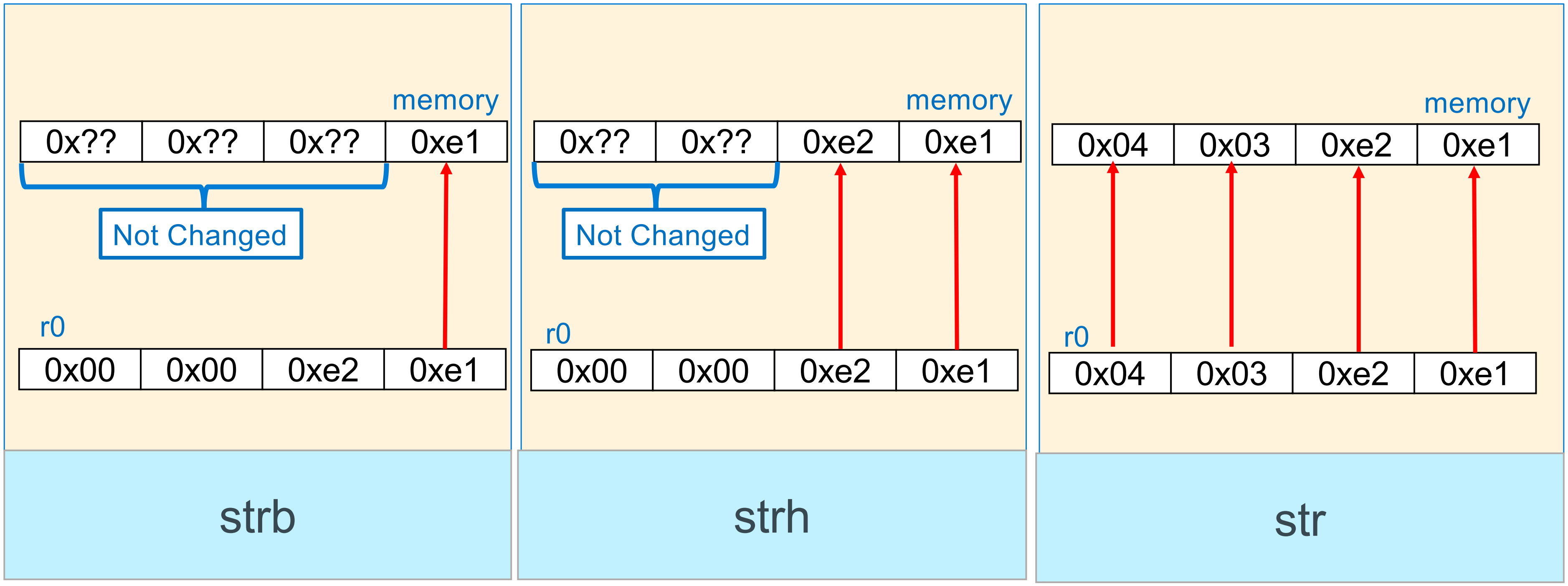


Signed Load a Byte, Half-word, Word



x

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



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

$\left.\begin{matrix}0x33\\0x22\\0x11\end{matrix}\right\}$ 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

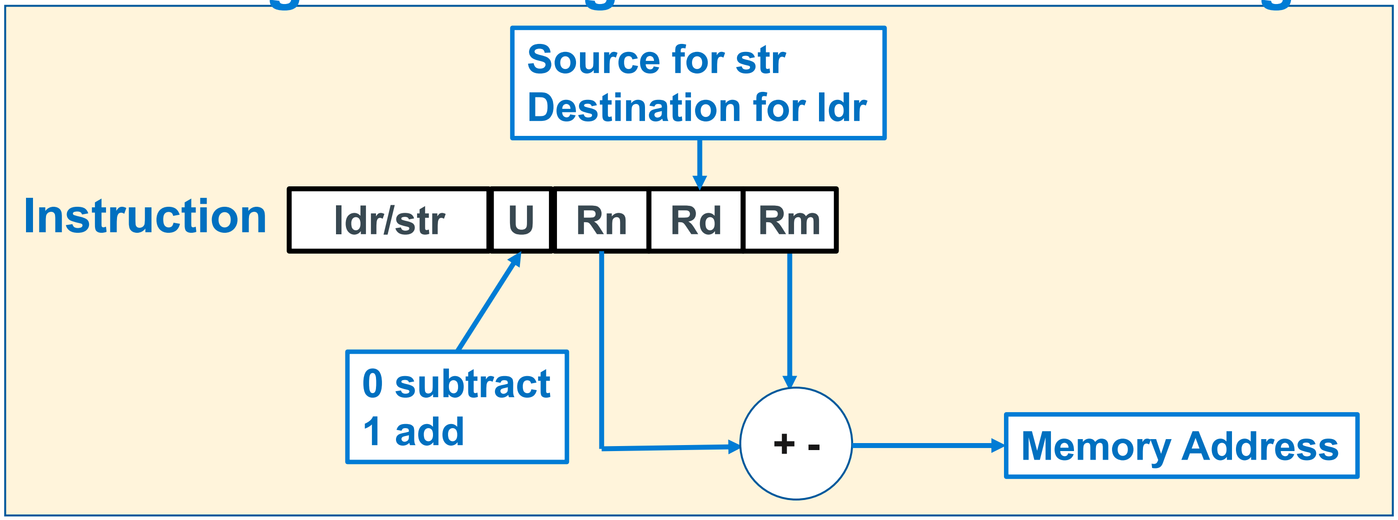
```
str  r1, [r0]
```

r1	31			0
	0x87	0x65	0xe3	0xe1

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

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]`

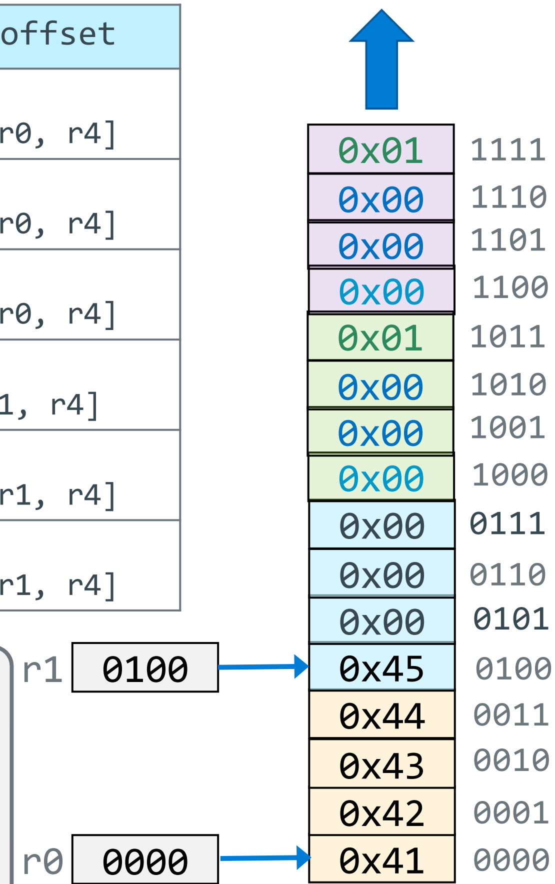
x

Array addressing with ldr/str

Array element	Base addressing	Immediate offset	register offset
ch[0]	ldrb r2, [r0]	ldrb r2, [r0, 0]	mov r4, 0 ldrb r2, [r0, r4]
ch[1]	add r0, r0, 1 ldrb r2, [r0]	ldrb r2, [r0, 1]	mov r4, 1 ldrb r2, [r0, r4]
ch[2]	add r0, r0, 2 ldrb r2, [r0]	ldrb r2, [r0, 2]	mov r4, 2 ldrb r2, [r0, r4]
x[0]	ldr r2, [r1]	ldr r2, [r1, 0]	mov r4, 0 ldr r2, [r1, r4]
x[1]	add r1, r1, 4 ldrb r2, [r1]	ldrb r2, [r1, 4]	mov r4, 4 ldrb r2, [r1, r4]
x[2]	add r1, r1, 8 ldrb r2, [r0]	ldrb r2, [r1, 8]	mov r4, 8 ldrb r2, [r1, r4]

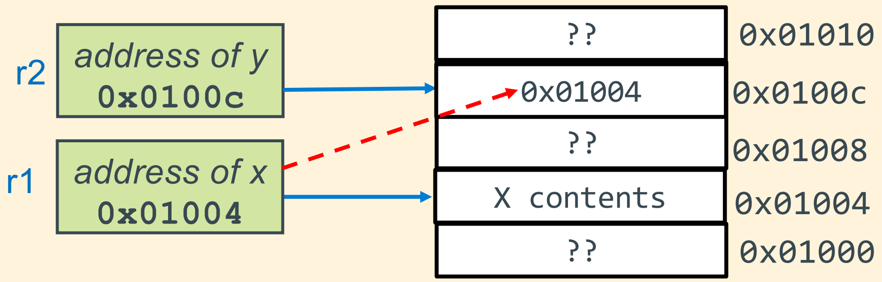
table rows are
independent instructions

```
        .data
ch:     .byte 0x41, 0x42, 0x43, 0x44
x:      .word 0x00000045
        .word 0x01000000
        .word 0x01020304
        .text
         ldr    r0, =ch
         ldr    r1, =x
```



ldr/str practice - 1

- r1 contains the Address of X (int X) in memory (register r1 points at X)
- r2 contains the Address of Y (int *Y) in memory (register r2 points at Y)
- write Y = &X;

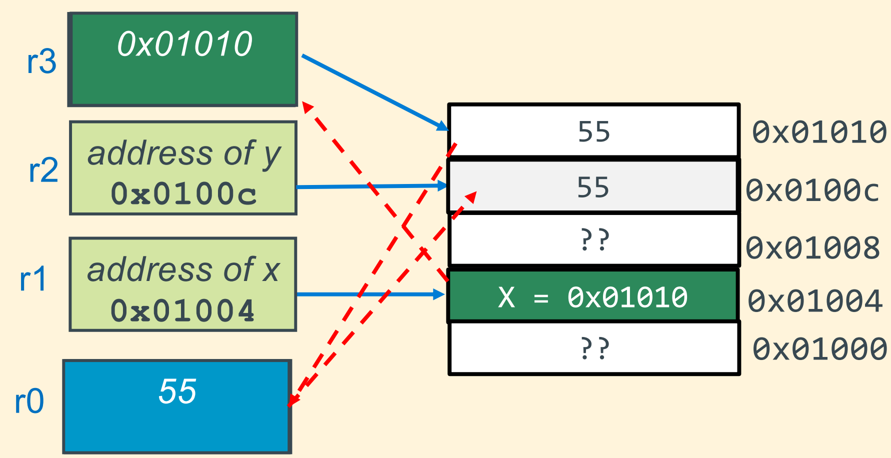



```
str   r1, [r2]        // y ← &x
```


x

ldr/str practice - 2

- r1 contains the Address of X (int *X) in memory (r1 points at X)
- r2 contains the Address of Y (int Y) in memory (r2 points at Y)
- write Y = *X;

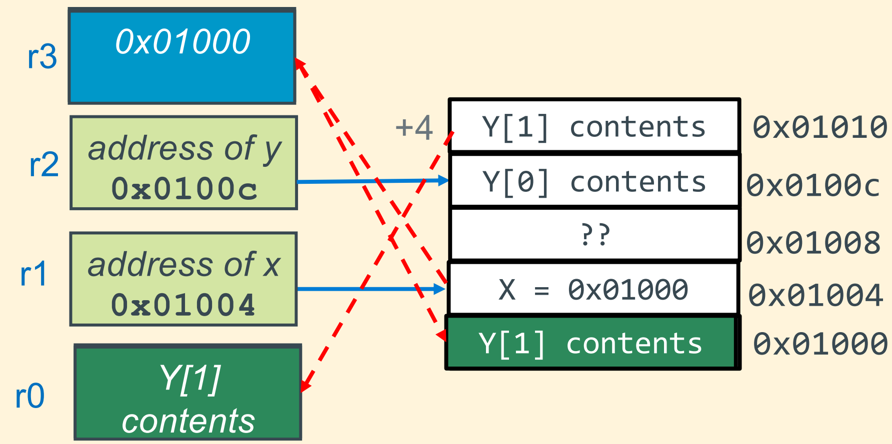



```
ldr   r3, [r1]  // r3 ← x (read 1)
ldr   r0, [r3]  // r0 ← *x (read 2)
str   r0, [r2]  // y ← *x
```


x

ldr/str practice - 3

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

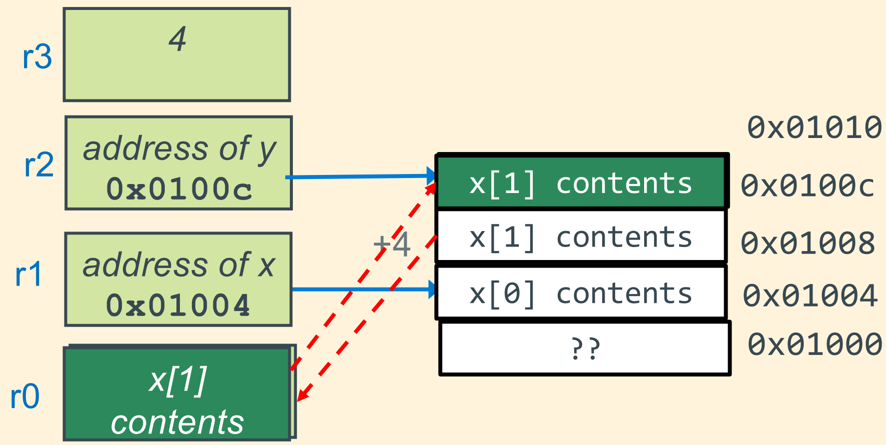



```
ldr    r0, [r2, 4]        // r0 ← y[1]
ldr    r3, [r1]           // r3 ← x
str    r0, [r3]           // *x ← y[1]
```


x

ldr/str practice - 4

- r1 contains the Address of X (int X[2]) in memory (r1 points at &x[0])
- r2 contains the Address of Y (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

Label (Address) Math

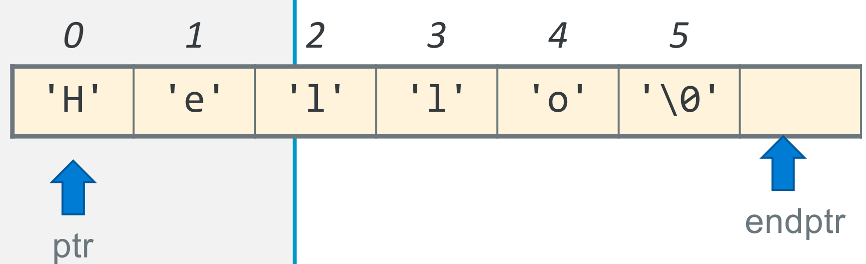
- You can have the assembler calculate some useful values for you
- One common use is calculating the distance in bytes between two labels
- The dot (.) refers to the address on the current line (the next byte after a previous space allocation)

```
       .section .rodata
.Lst:  .string "The value of x is %d\n"
       .equ STSZ, (. - .Lst)    // number of bytes in .Lst includes \0
       .equ STLEN, STSZ - 1     // string length of .Lst
```


x

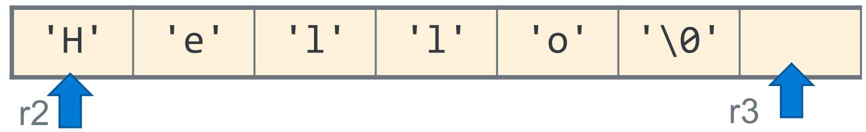
Example: Base Register Addressing with Arrays

```
#include <stdio.h>
#include <stdlib.h>

char msg[] ="Hello CSE30! We Are CountinG UpPER cASe letters!";

int
main(void)
{
    int cnt = 0;
    char *endpt = msg + sizeof(msg)/sizeof(*msg);
    char *ptr = msg;

    while(ptr < endpt) {
        if ((*ptr >= 'A') && (*ptr <= 'Z'))
            cnt++;
        ptr++;
    }
    printf("%d\n", cnt);
    return EXIT_SUCCESS;
}
```



x

Example: Base Register Addressing with Arrays



- Iterates a pointer (r2) through the array
- r3 contains the address +1 past the end of the string
- MSGSZ is the size of the array (including the '\0) if you wanted to excluded the '\0', then subtract 1 from MSGSZ

```
    .data       // segment
msg:.string     "Hello CSE30! We Are CountinG UpPER cASe letters!"
    .equ        MSGSZ, (. - msg)  // number of bytes in msg
    .section .rodata
.Lpf:.string    "%d\n"            // literal for printf
...
    ldr     r2, =msg            // ptr point to &msg
    add     r3, r2, MSGSZ       // endpt points after end
.Lwhile:
    cmp     r2, r3              // at end of buffer yet?
    bge     .Lexit

    ldrb    r0, [r2]            // get next char (base addressing)
    cmp     r0, 'A'             // is it less than an 'A" ?
    blt     .Lendif             // if so, not CAP (short circuit)
    cmp     r0, 'Z'             // is it greater than a 'Z"?
    bgt     .Lendif            // if so, not CAP
    add     r1, r1, 1          // is a CAP increment
.Lendif:
    add     r2, r2, 1          // move to next char
    b       .Lwhile            //go to loop guard at top of while
.Lexit:
```

loop guard

x

Example: Base Register + Offset Register

```
    ldr     r2, =msg        // ptr point to &msg
    add     r3, r2, MSGSZ   // endpt points after end
.Lwhile:
    cmp     r2, r3          // at end of buffer yet?
    bge     .Lexit

    ldrb    r0, [r2]        // get next char
    cmp     r0, 'A'         // is it less than an 'A" ?
    blt     .Lendif         // if so, not CAP
    cmp     r0, 'Z'         // is it greater than a 'Z"?
    bgt     .Lendif         // if so, not CAP
    add     r1, r1, 1       // is a CAP increment

.Lendif:
    add     r2, r2, 1       // move to next char
    b       .Lwhile         //go to loop guard while top
.Lexit:
```

Using Base register pointer with an end pointer

```
    ldr     r2, =msg
    mov     r3, 0        // index reg
.Lwhile:
    cmp     r3, MSGZ     // are we done?
    bge     .Lexit

    ldrb    r0, [r2, r3]
    cmp     r0, 'A'
    blt     .Lendif
    cmp     r0, 'Z'
    bgt     .Lendif
    add     r1, r1, 1

.Lendif:
    add     r3, r3, 1    // index++
    b       .Lwhile
.Lexit:
```

Using Base register pointer + Offset register

x

Example: Base Register + Register Offset Two Buffers

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

int src[SZ] = {1, 3, 5, 7, 9, 11};

int dest[SZ];
int
main(void)
{
    for (int i = 0; i < SZ; i++)
        dest[i] = src[i];


    return EXIT_SUCCESS;
}
```

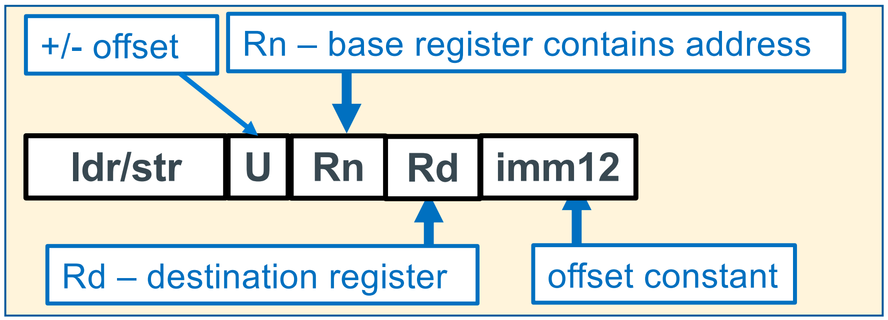
```
    .data       // segment
src:.word       1, 3, 5, 7, 9, 11
    .equ        SRCSZ, (. - src)  // bytes
msg
dest:.space     SRCSZ
    .equ        INT_STEP, 4
…

    ldr     r0, =src            // ptr to src
    ldr     r1, =dest           // ptr to
dest
    mov     r2, 0

.Lfor:
    cmp     r2, SRCSZ           // in bytes!
    bge     .Lexit

    ldr    r3, [r0, r2]
    str    r3, [r1, r2]
    add    r2, r2, INT_STEP
    b      .Lfor
.Lexit:
```

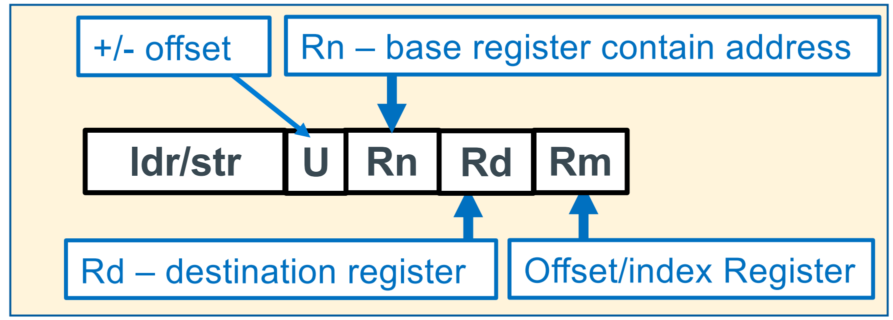
one increment covers both arrays

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

X

Reference: 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

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

Version 1.04

UCSD CSE 30

Aarch32 Assembly –Load & Store, Bit Ops, Functions

Week 8

Lecture 23

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**

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

```
#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

`r0, r1`

two passed args in this use of printf

```
        .extern printf    //declare printf
        .section .rodata
.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

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

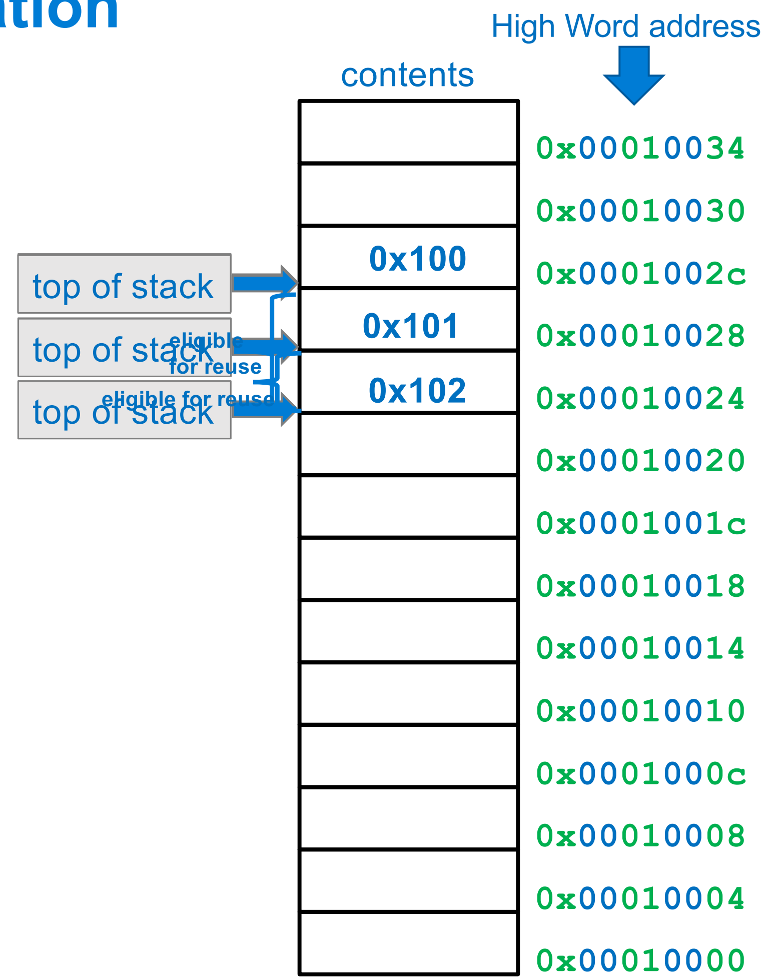
Data Structure Review: Stack Operation

- 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***)

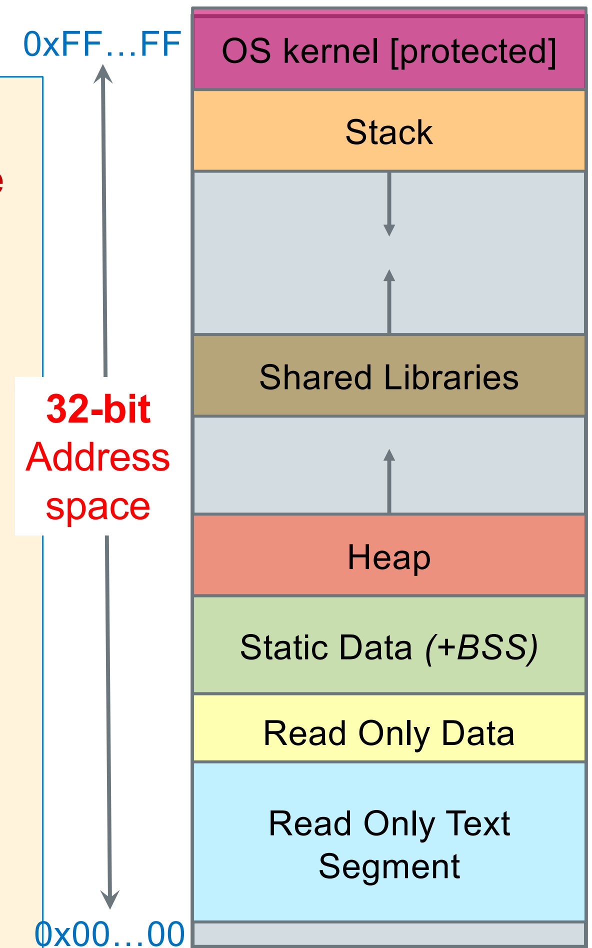
pop (sp + element size)



x

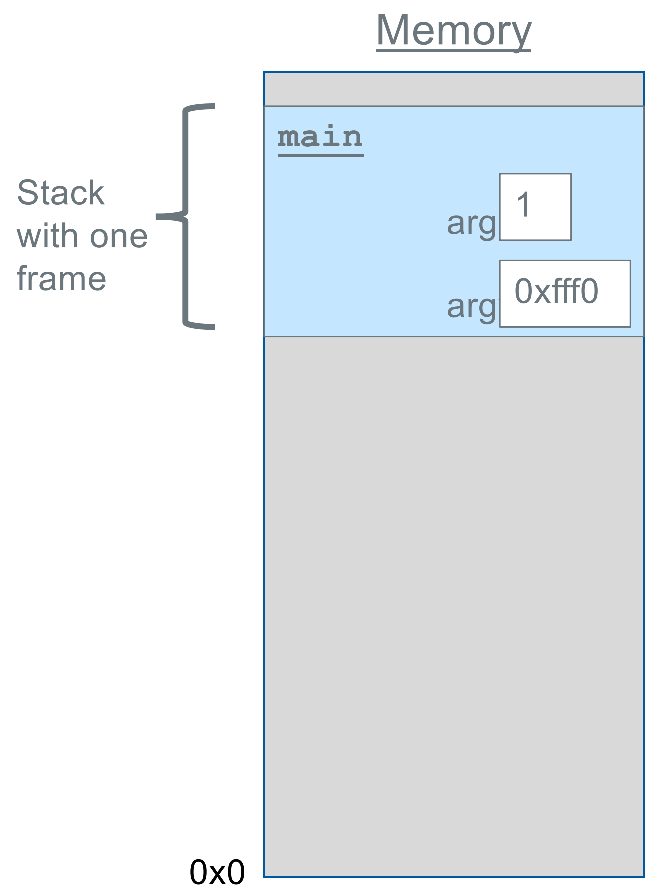
Stack Segment: Support of Functions

- 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)



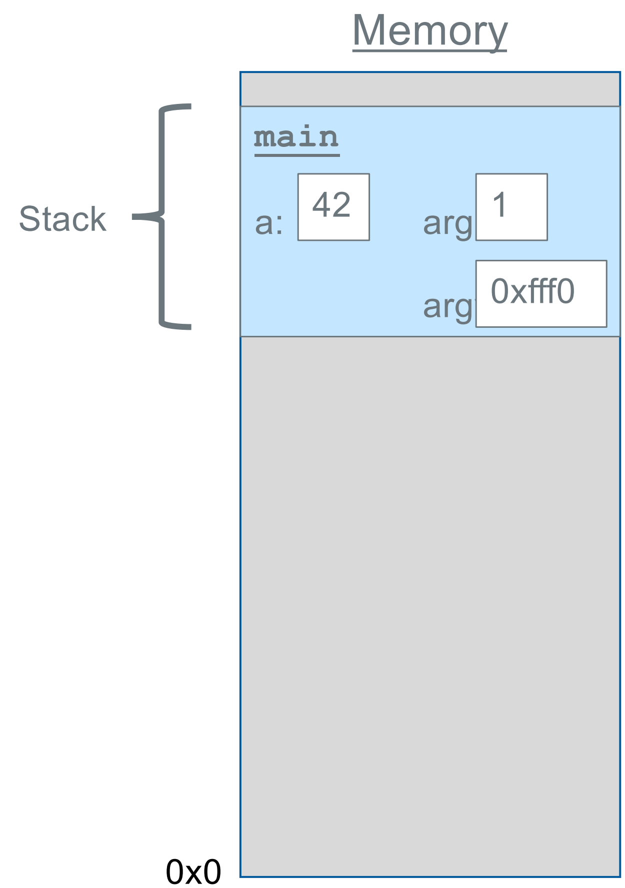
The Stack

```
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;
}
```



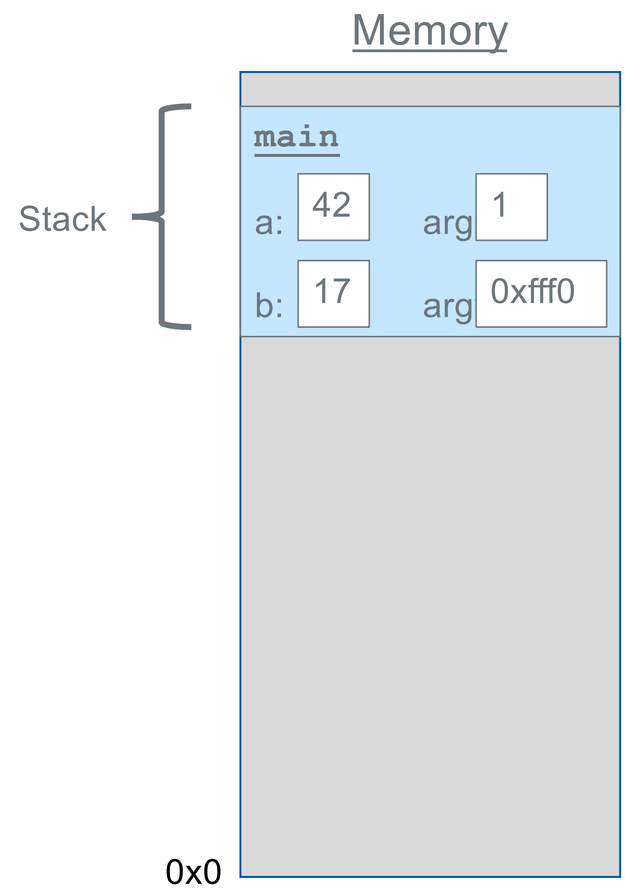
The Stack

```
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;
}
```



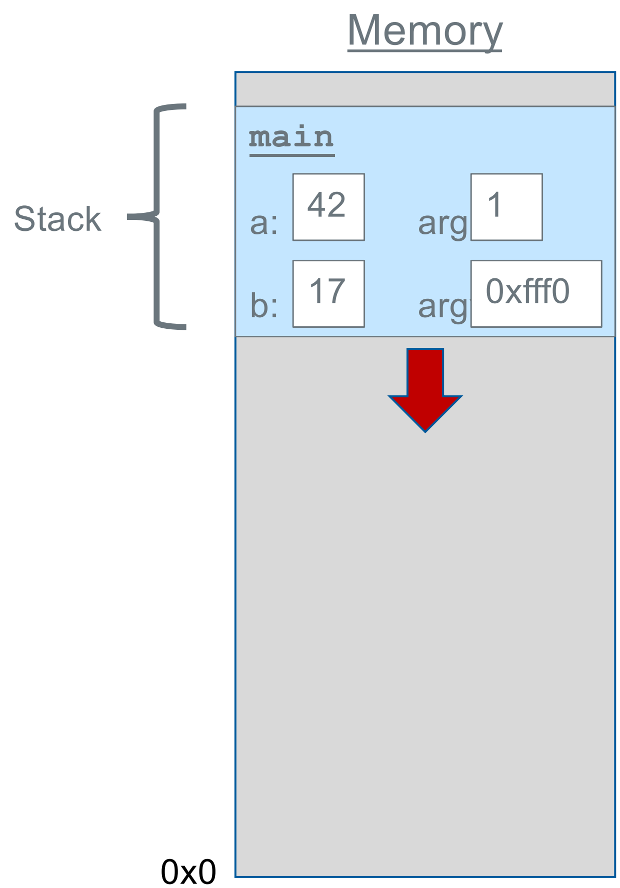
The Stack

```
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;
}
```



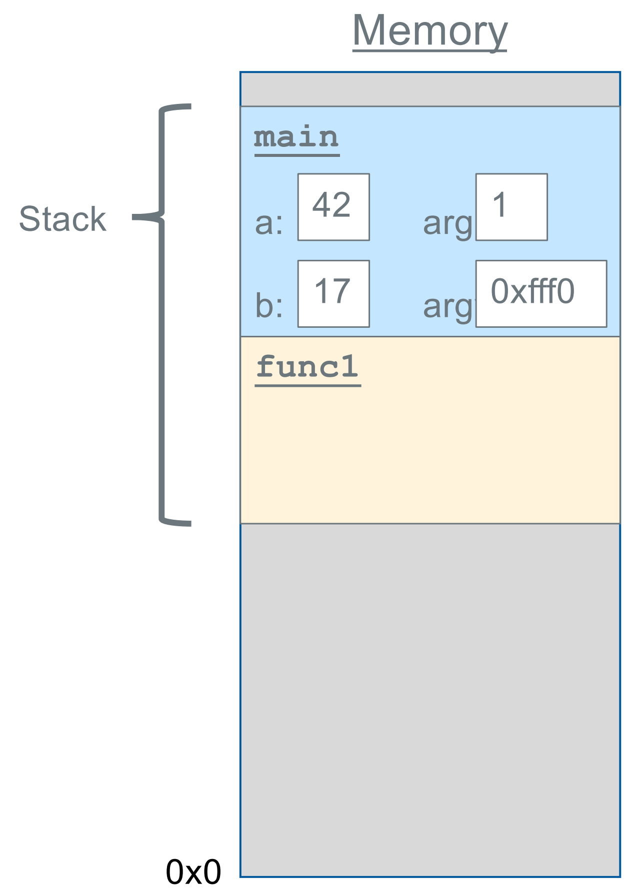
The Stack

```
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;
}
```



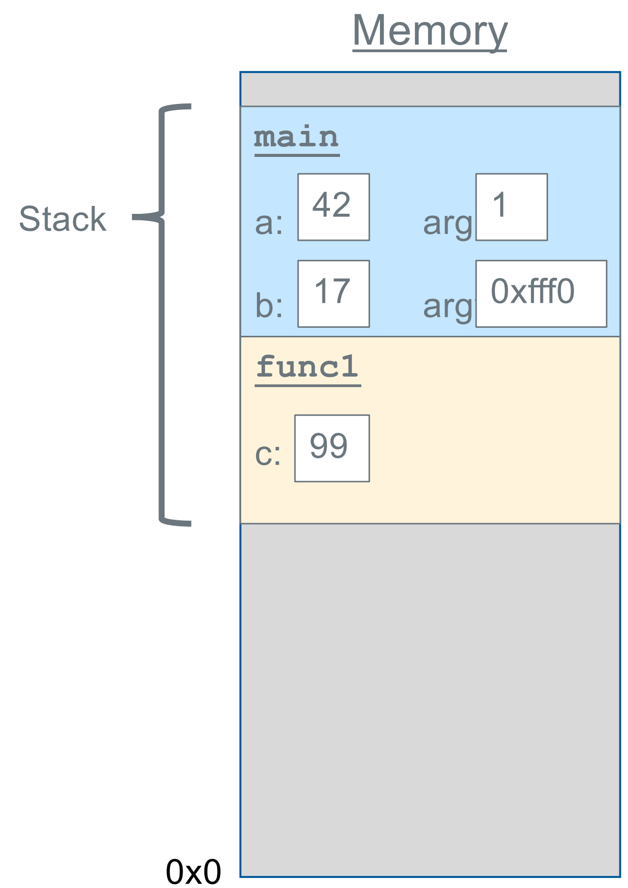
The Stack

```
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;
}
```



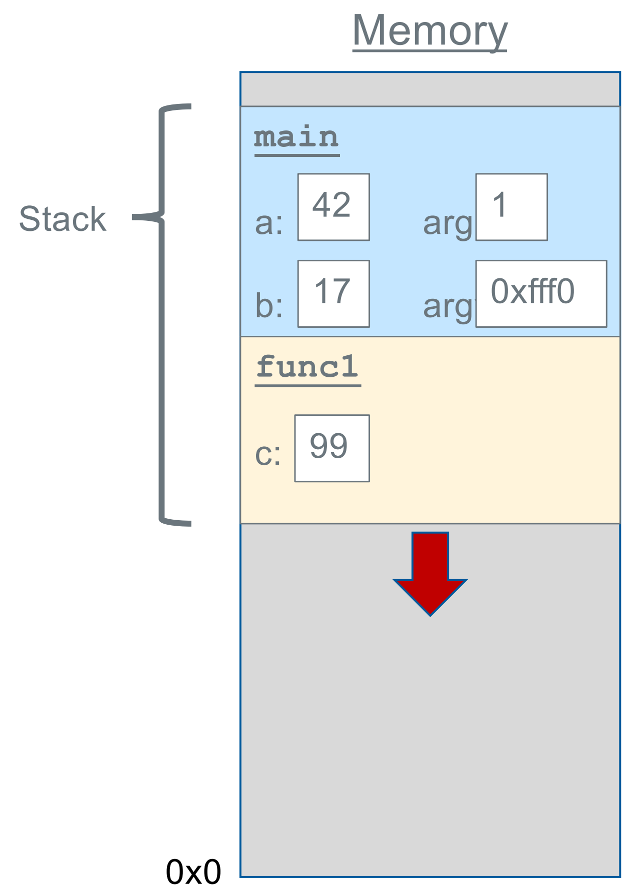
The Stack

```
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;
}
```



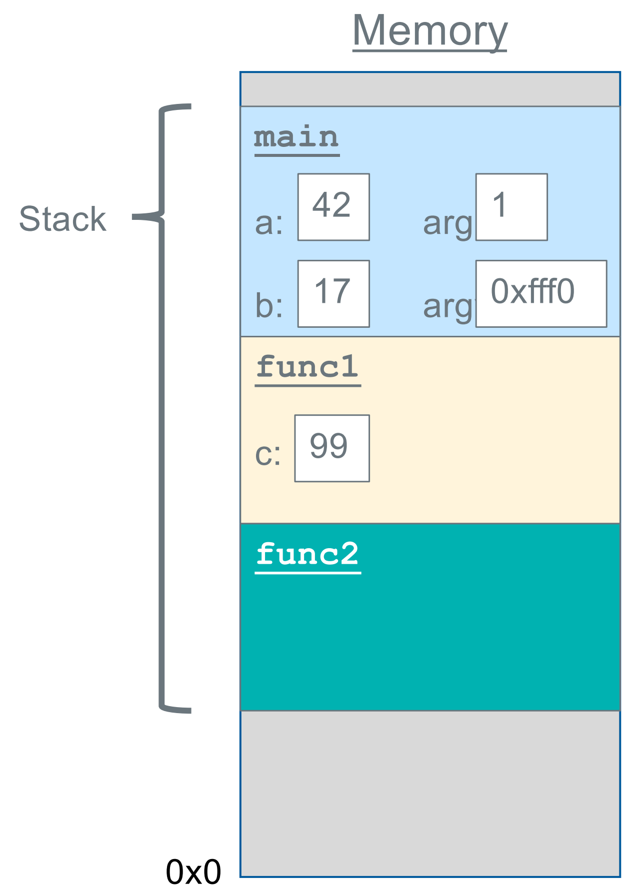
The Stack

```
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;
}
```



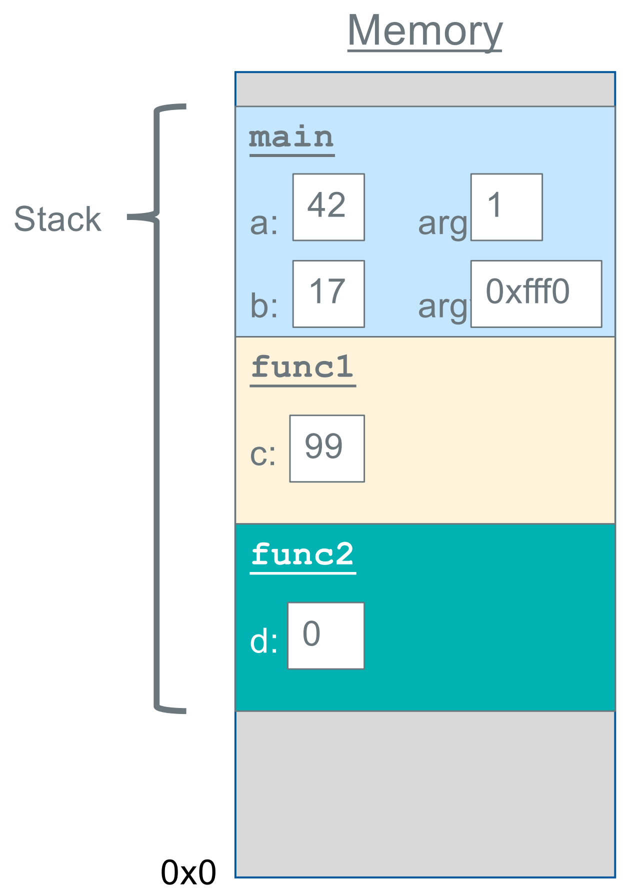
The Stack

```
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;
}
```



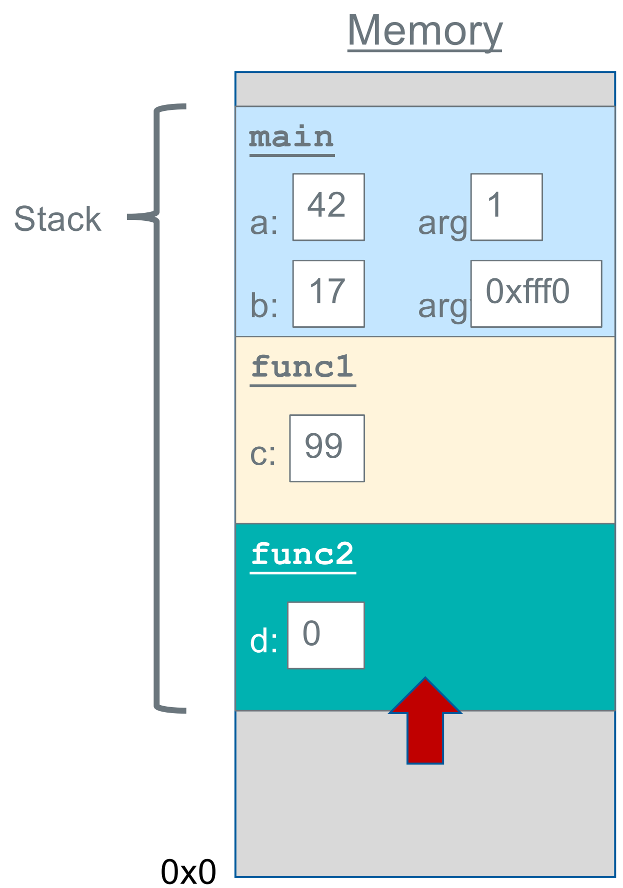
The Stack

```
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;
}
```



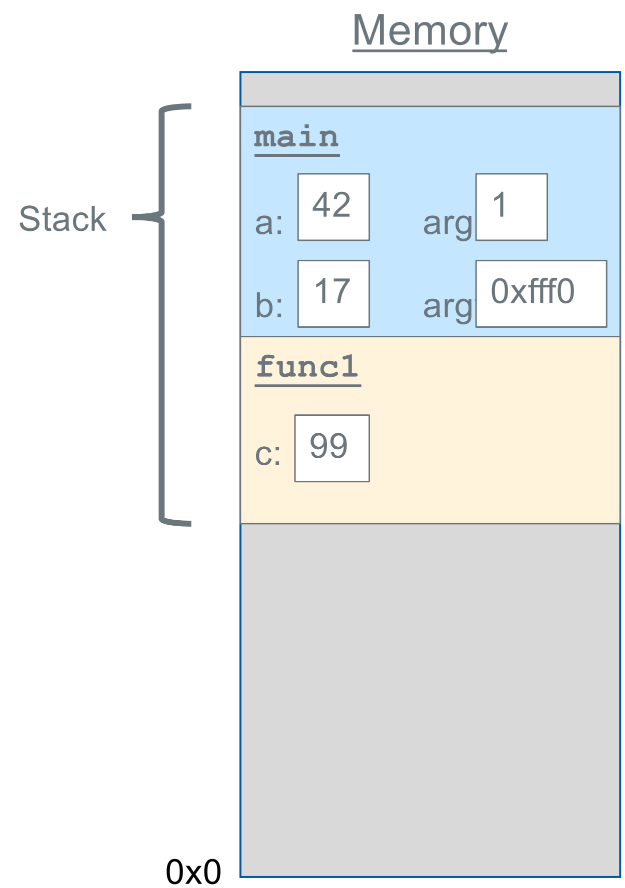
The Stack

```
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;
}
```



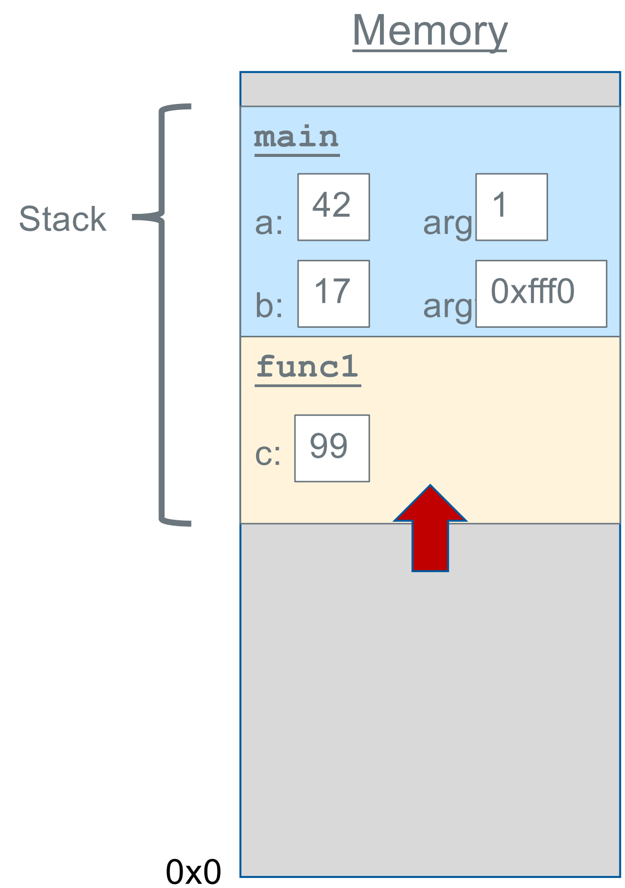
The Stack

```
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;
}
```



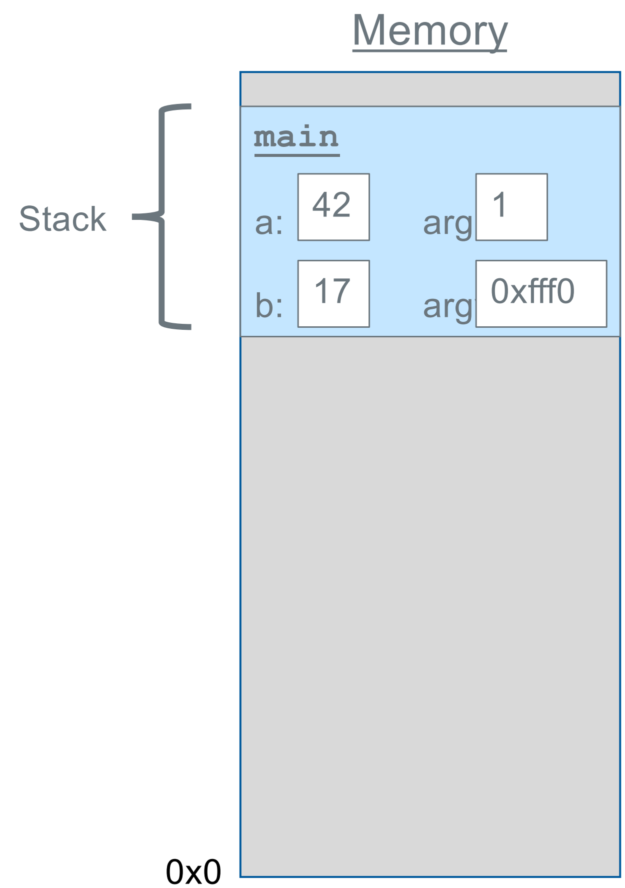
The Stack

```
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;
}
```



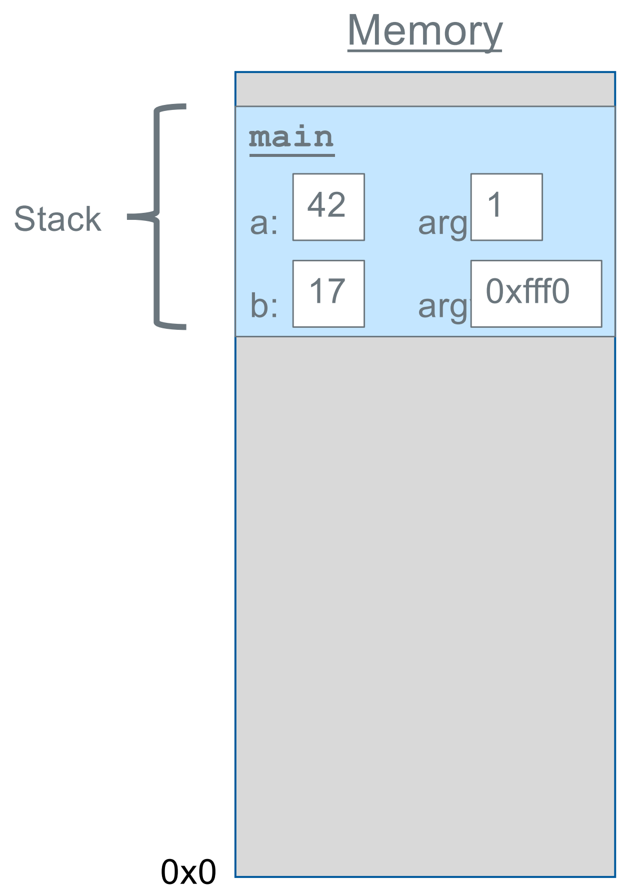
The Stack

```
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;
}
```



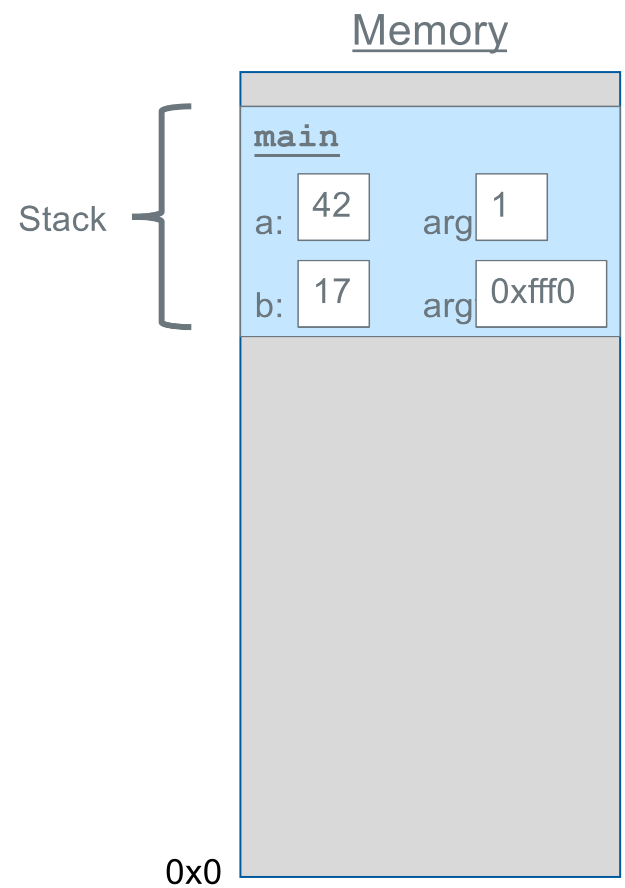
The Stack

```
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;
}
```



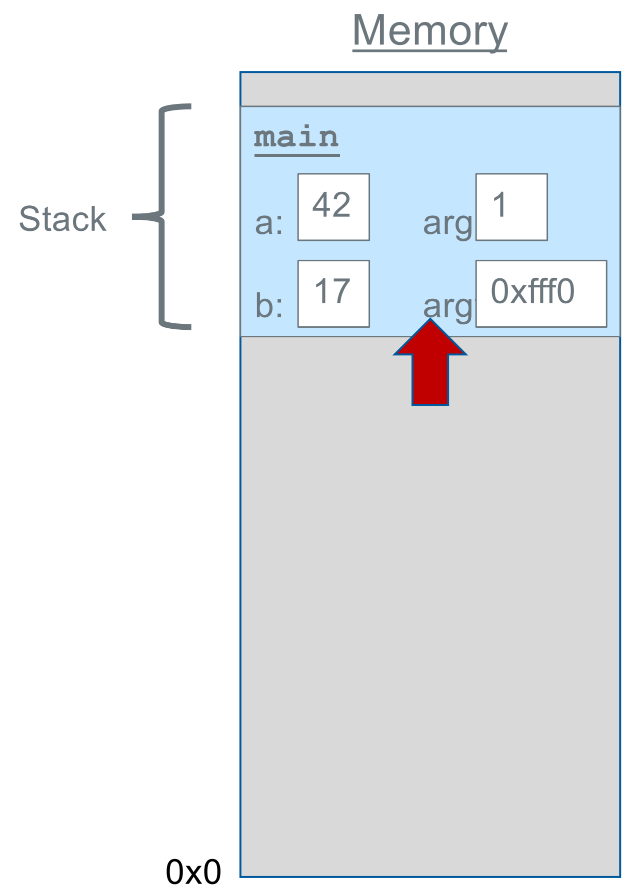
The Stack

```
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;
}
```



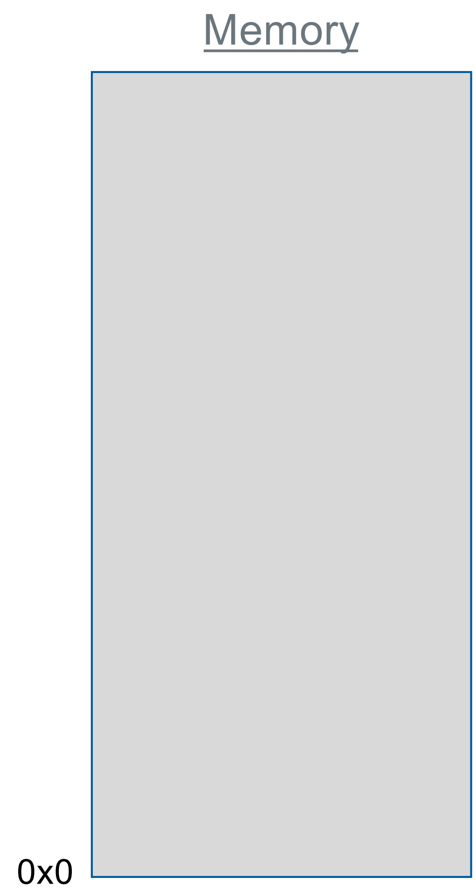
The Stack

```
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

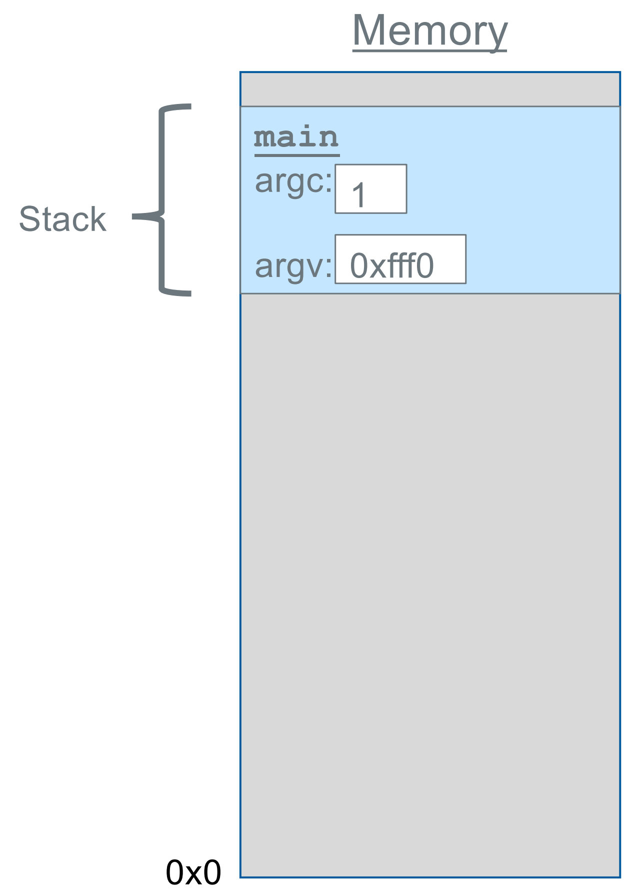
```c
void func2() {
    int d = 0;
}

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.

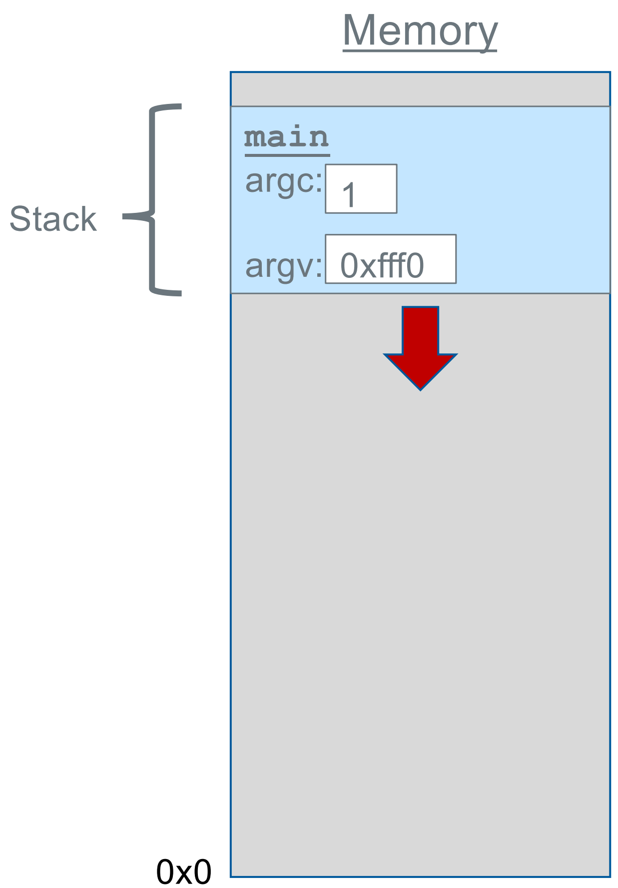
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

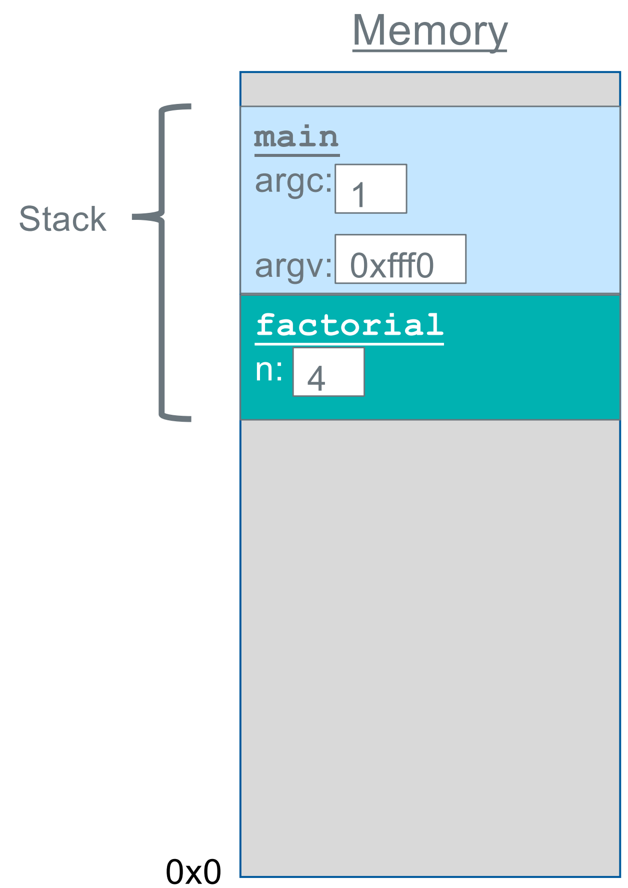
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

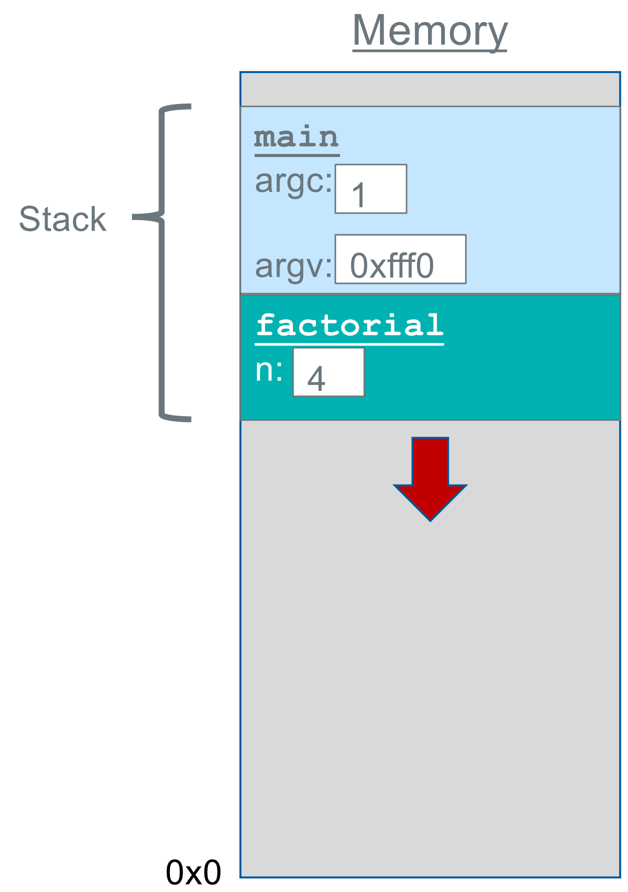
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

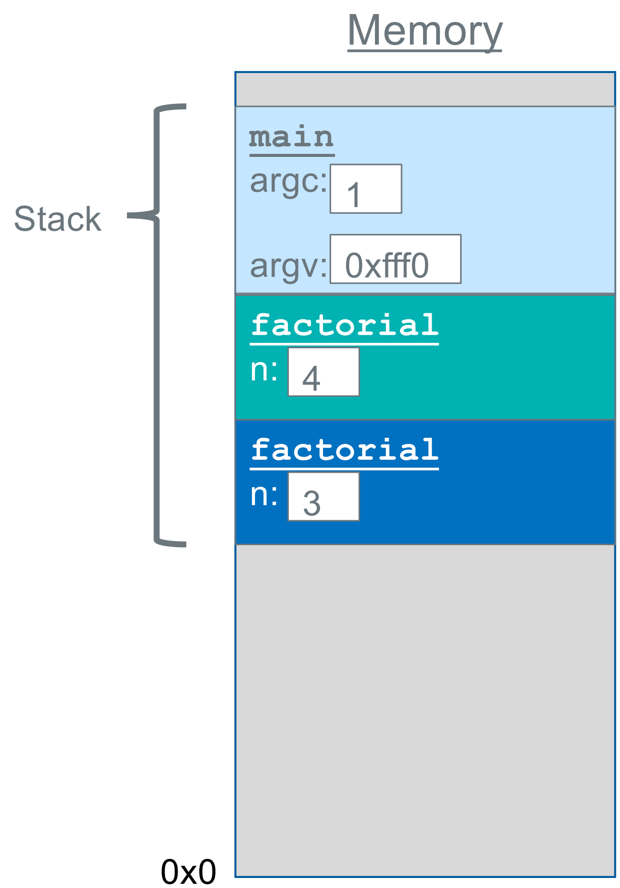
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

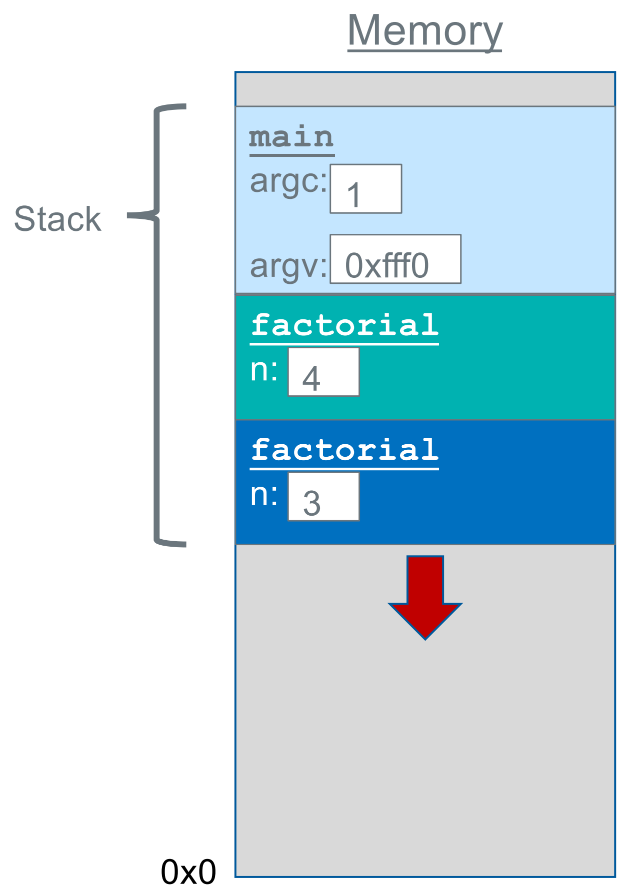
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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



The Stack

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

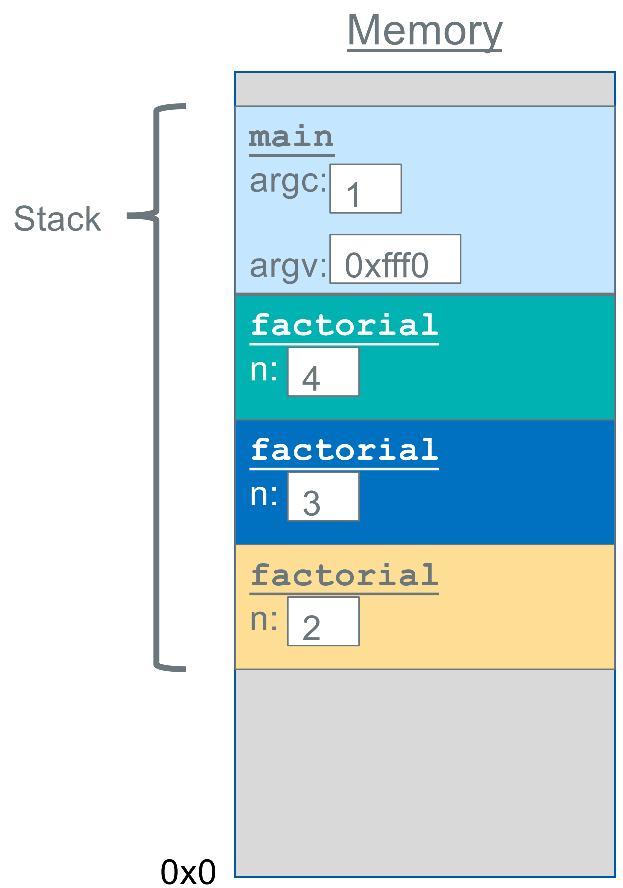
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

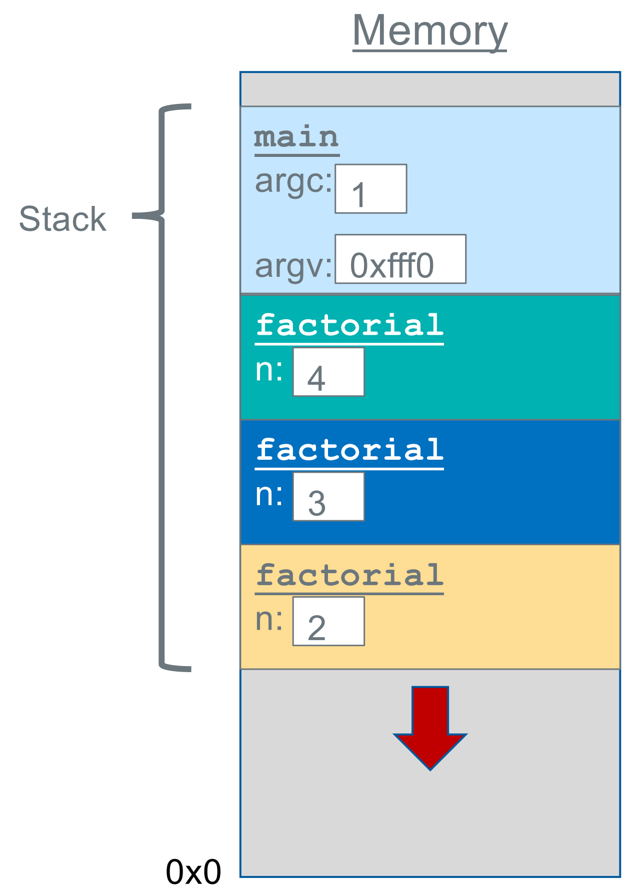
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

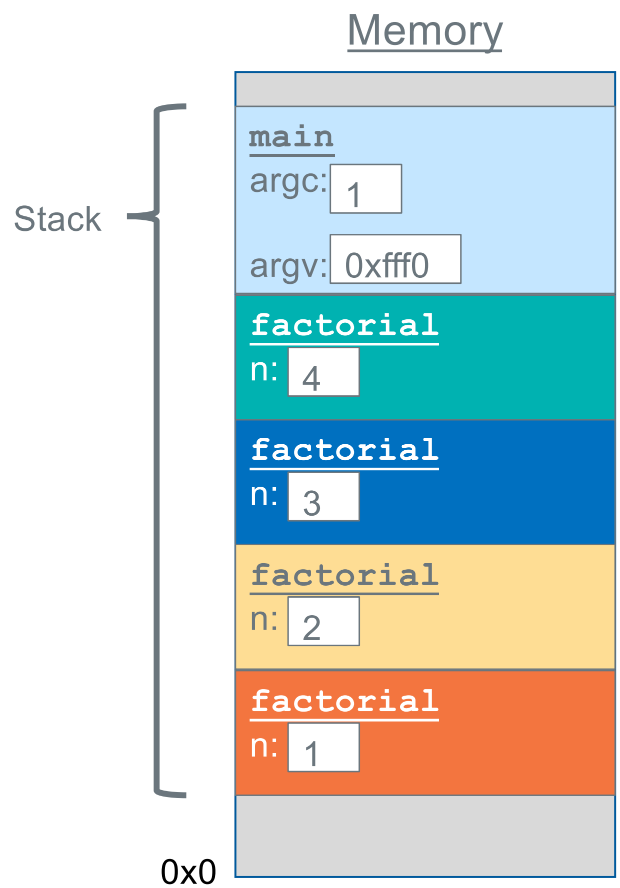
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

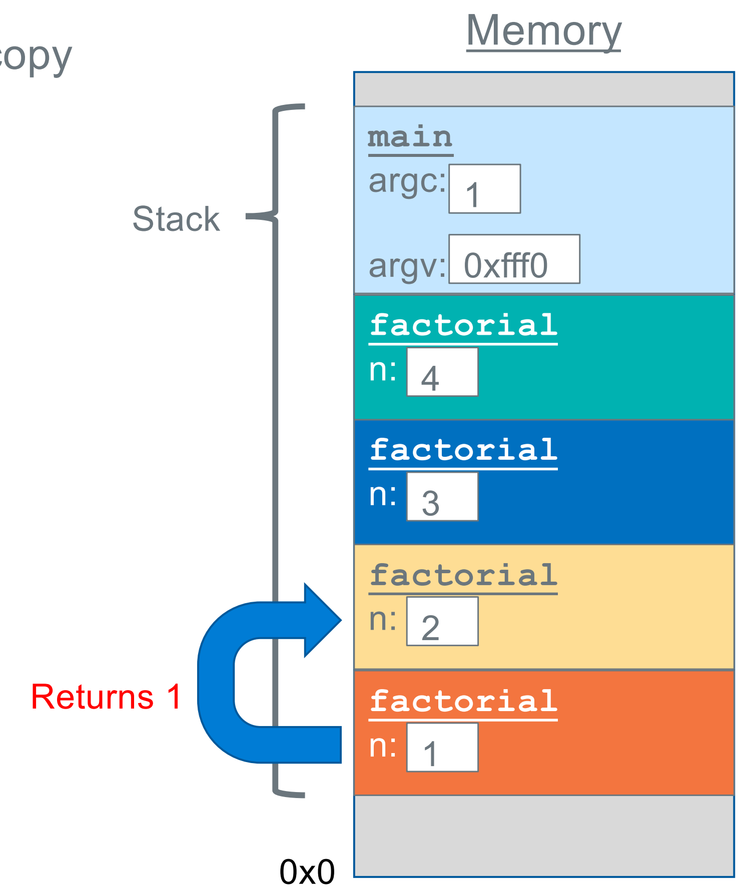
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

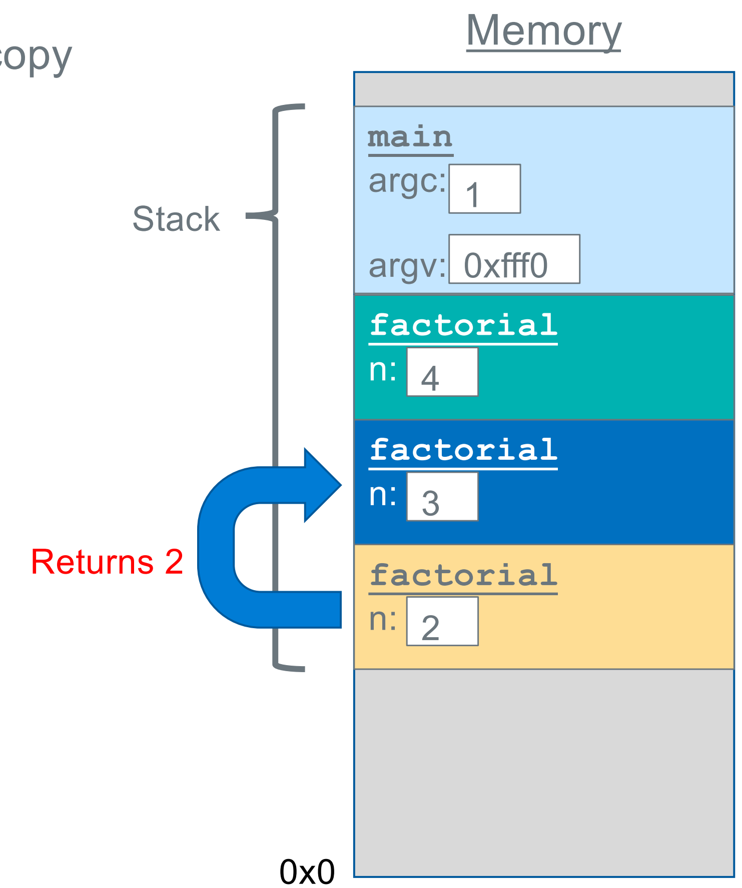
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

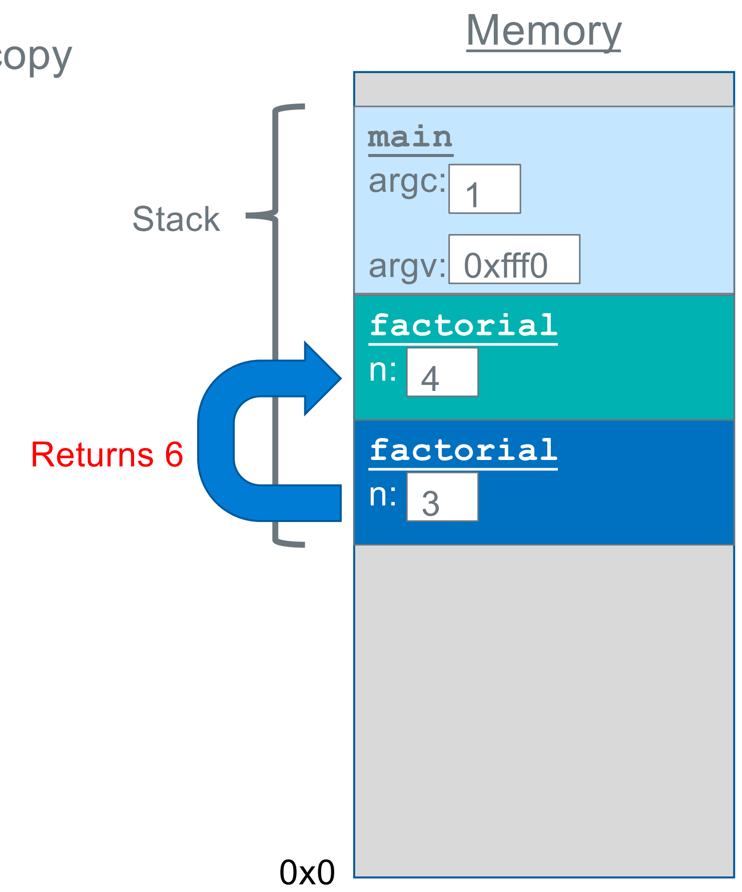
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

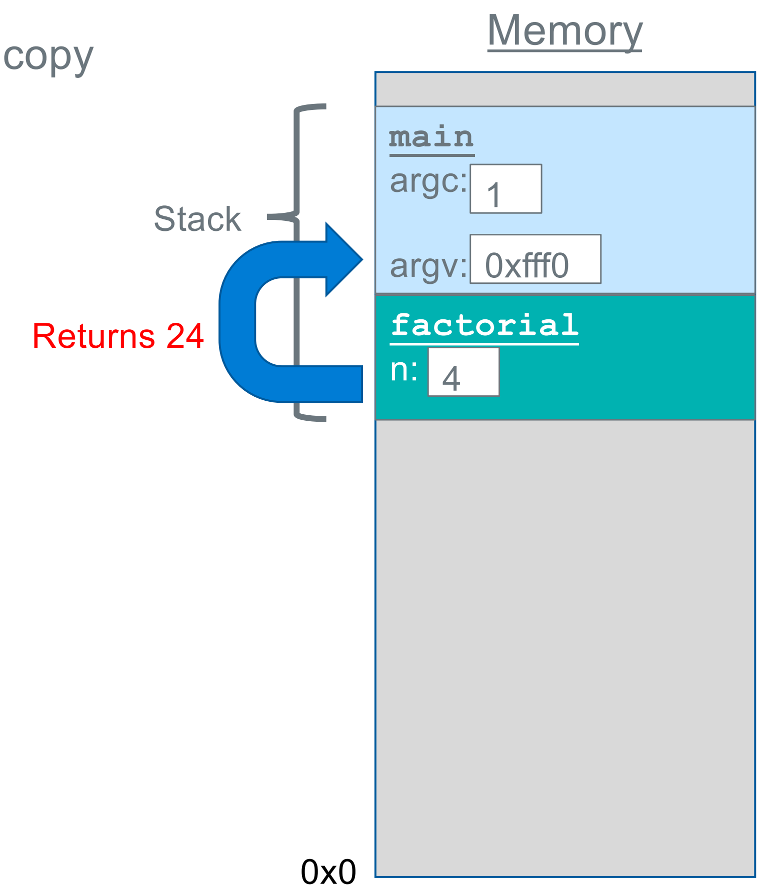
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

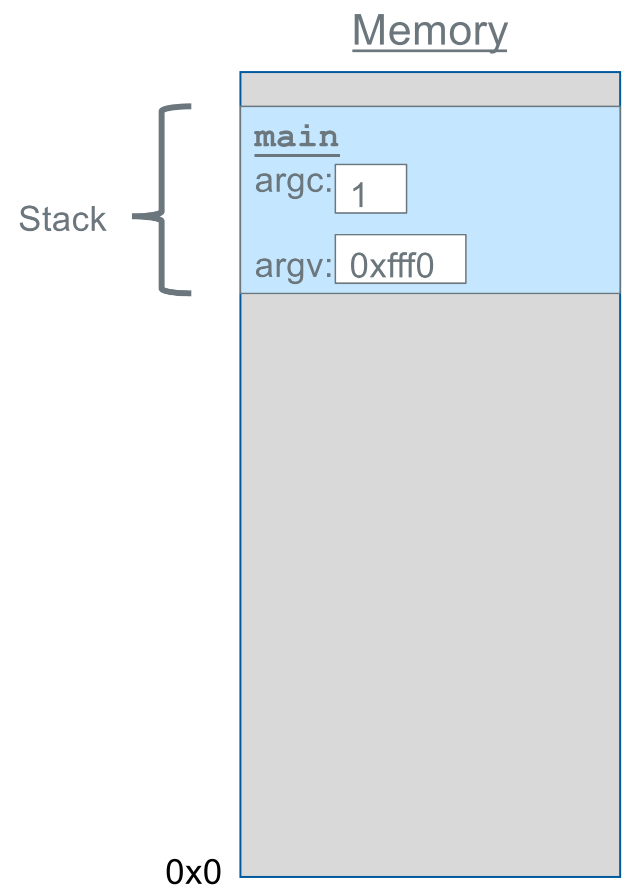
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

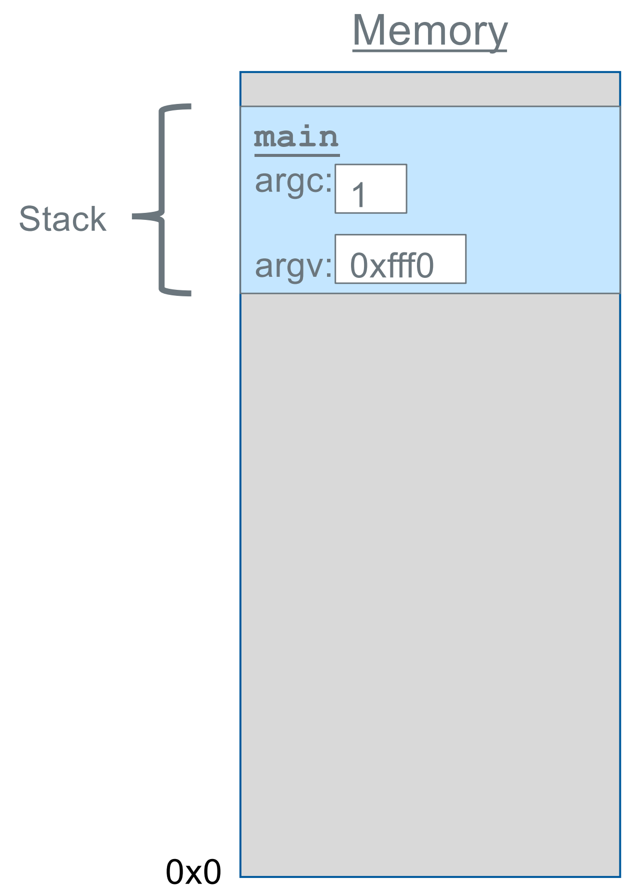
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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 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)

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

`.global function_name`

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

`.type name, %function`

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

`equ FP_OFF, 4`

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

`.size name, bytes`

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

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)
- Therefore, the **contents of the link register is the return address**
  - used to return to the calling function at the point right after the call

x

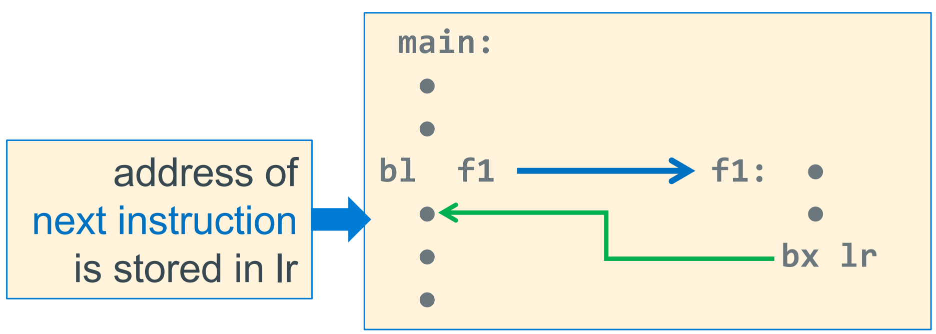
Support For Function Calls and Function Call Return - 2

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`



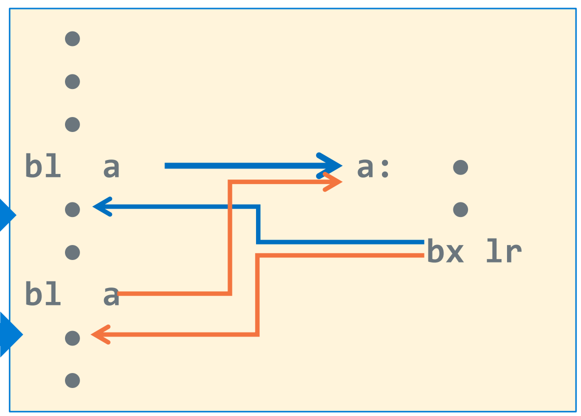
x

bl and bx operation working together

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

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

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


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

address of next instruction is stored in lr



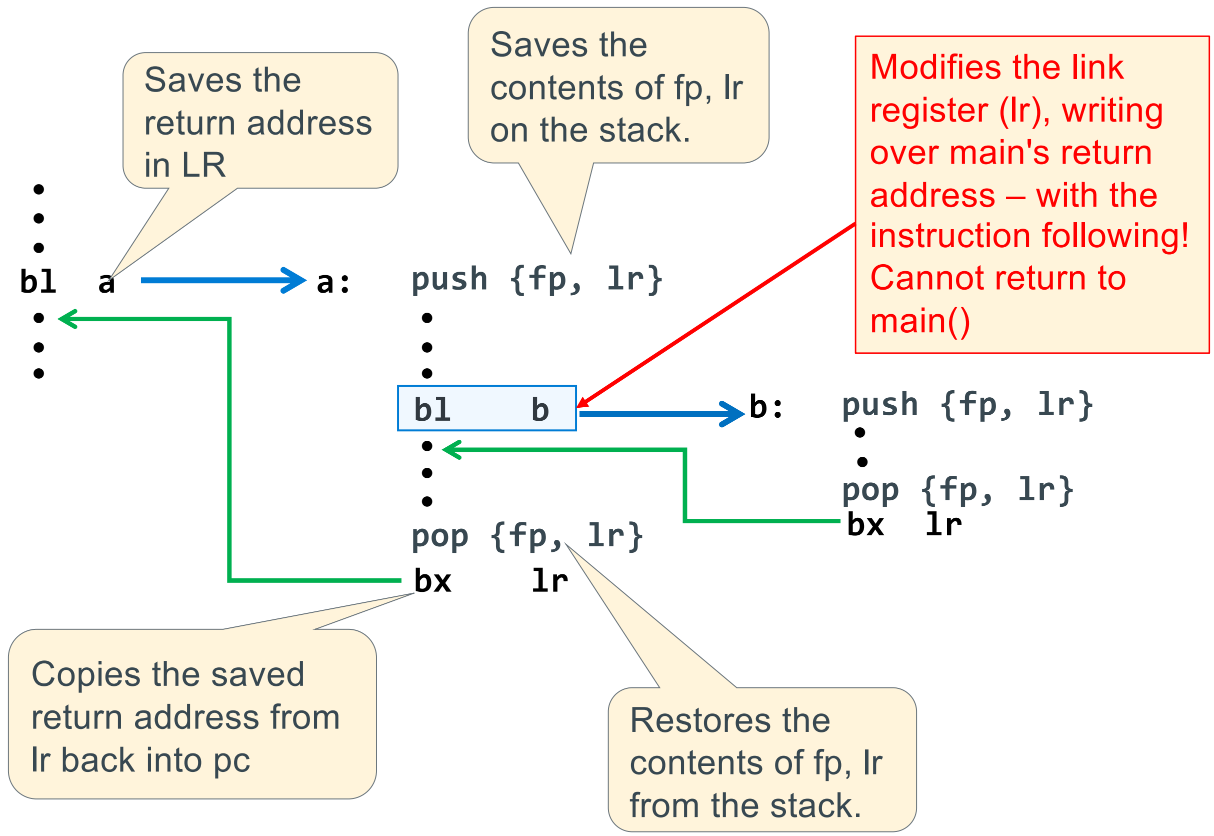
address of next instruction is stored in lr

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

x

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

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

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

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

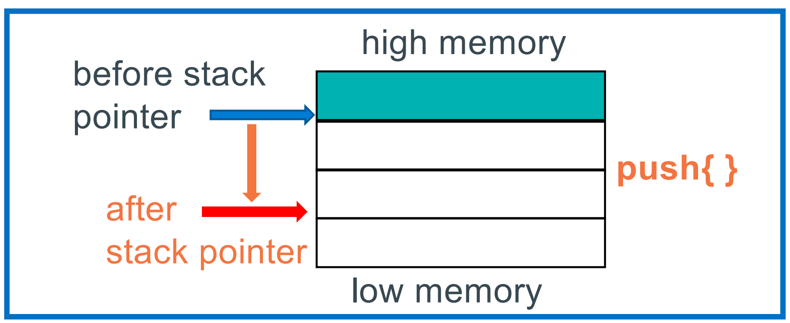


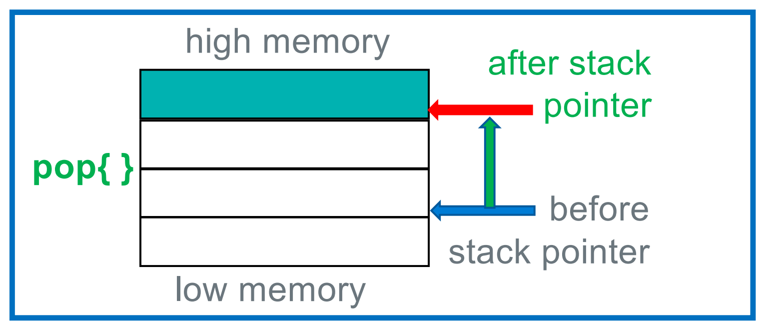
The frame pointer is used to find variables on the stack – later

x

Preserving and Restoring Registers on the Stack

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





x

Preserving and Restoring Registers on the Stack

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

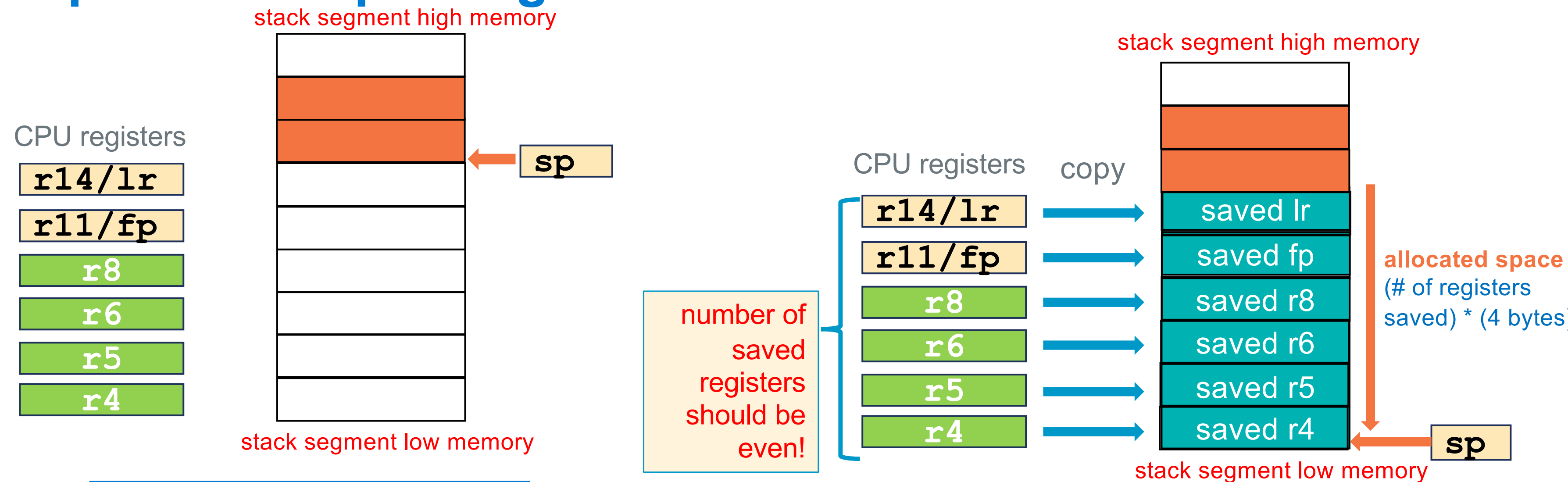
- Where `{reg list}` is a **list of registers** in numerically increasing order

  `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}`
  - The **count of registers specified** in the `{reg list}` **for now is an even number, 2 or greater**
  - The smallest `{reg list}` you should specify is two registers `{fp, lr}`

x

push: Multiple Register Save



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

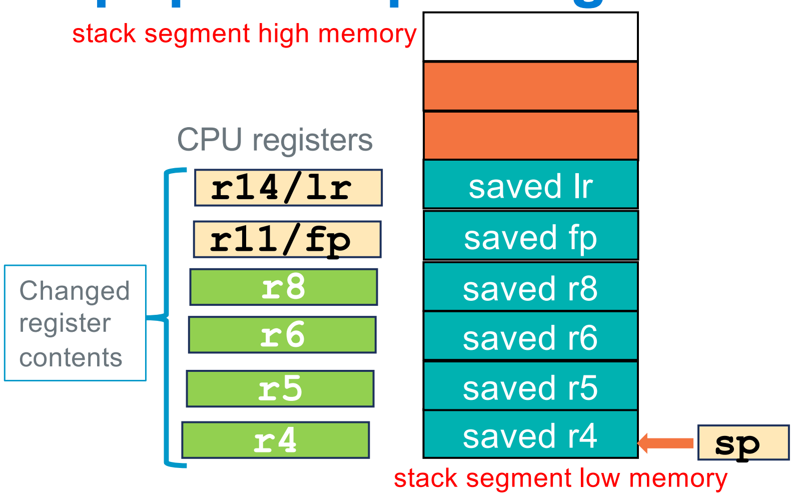
`after saved registers`
`push{r4-r6,r8,fp,lr}`

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

- `push` copies the contents of the `{reg list}` to stack segment memory
- `push` subtracts (# of registers saved) * (4 bytes) from the `sp` to ***allocate*** space on the stack

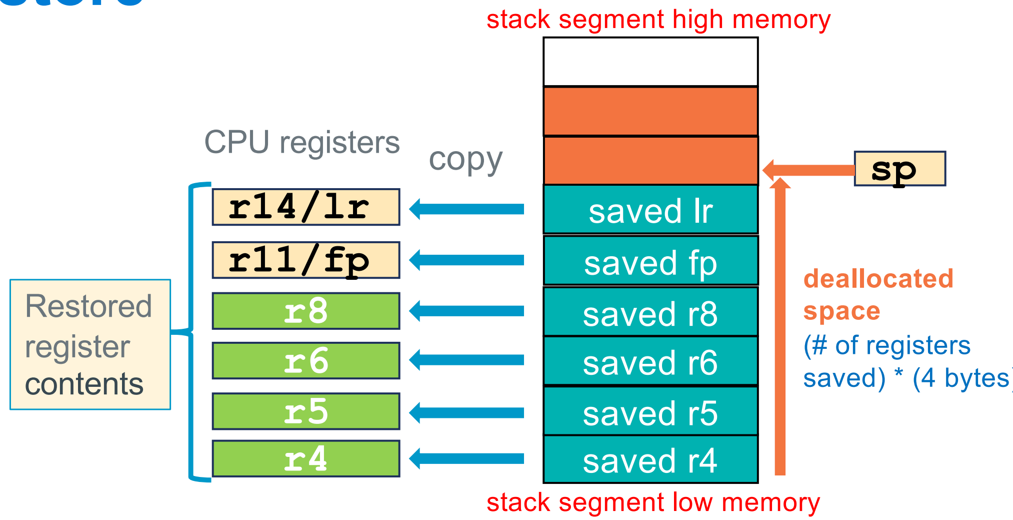
x

pop: Multiple Register Restore



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

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



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

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

x

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

Argument and Return Value Requirements

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

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

Single Byte

	31			0
`r0`	0x00	0x00	0x00	0xe1

observe the zero fill

Single Halfword

	31			0
`r0`	0x00	0x00	0xe3	0xe1

observe the zero fill

Full Word

	31			0
`r0`	0x87	0x65	0xe3	0xe1

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

```
#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

`r0, r1`

two passed args in this use of printf

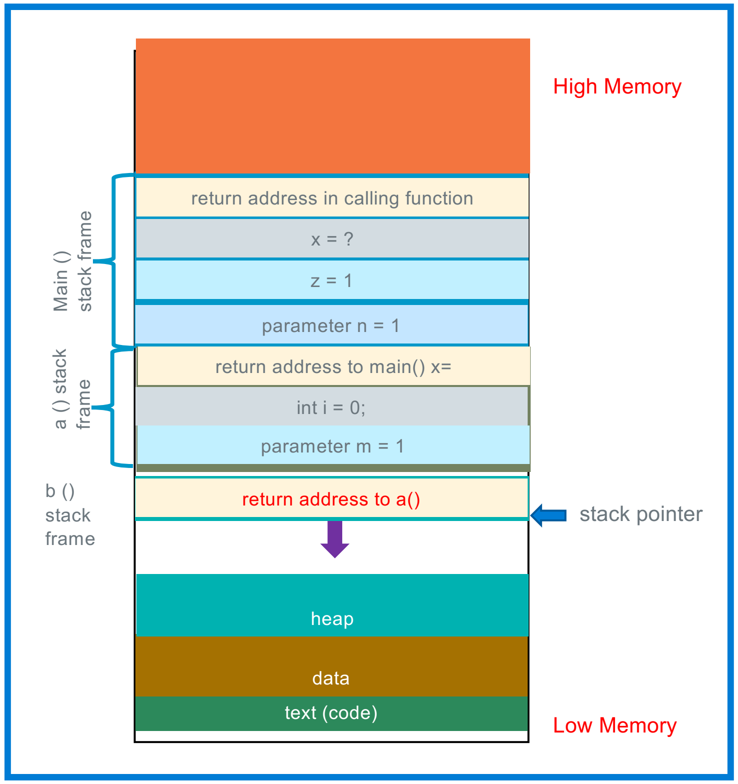
```
        .extern printf    //declare printf
        .section .rodata
.Lfst:  .string  "c=%d\n"
```

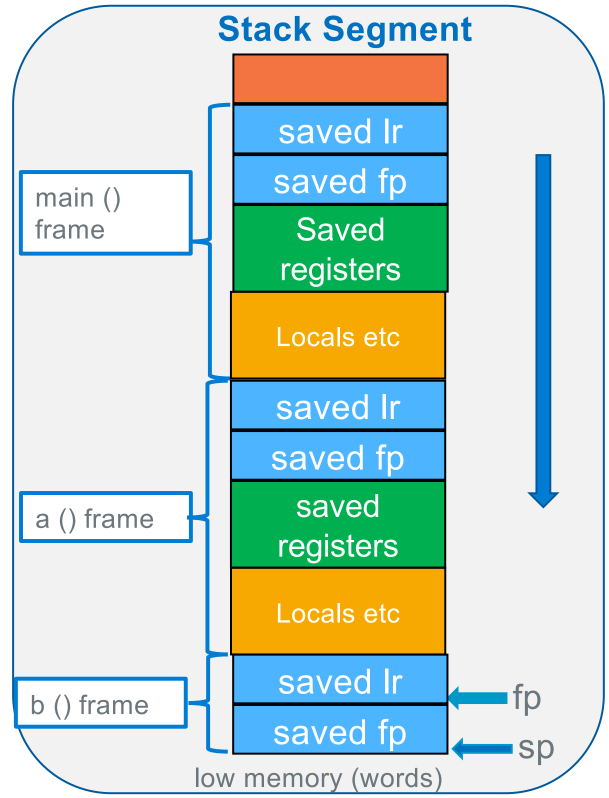
```
// part of the text segment below

        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

Basic Stack Frames (Arm Arch32 Procedure Call Standards)

```
int
main(int argc, char *argv[])
{
    int x, z = 1;
    while (--argc >0)
                 /* code */;
    x = a(z);
     z = b(z);
    /* code */
    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;
}
```

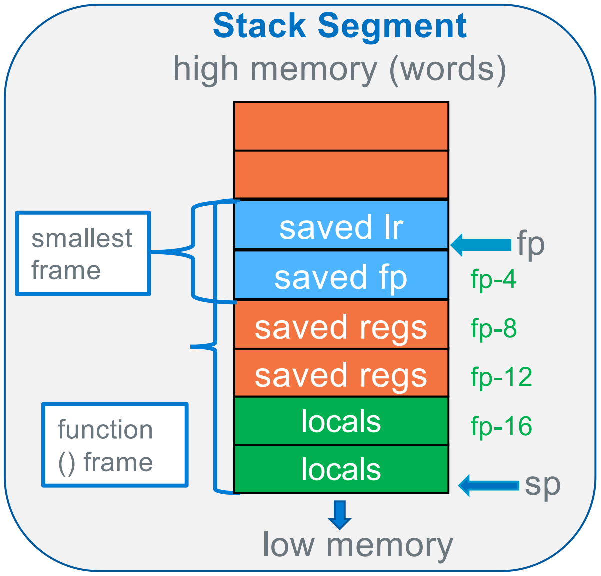




X

Stack Frames (Arm Arch32 Procedure Call Standards)

- **Stack frames are 8-byte aligned and expands from high to low memory**
- The `sp` contains the starting byte address (points at lowest address) of the top element in the stack
- `fp` must point at the base element (always `lr`) in the current stack frame and once set is not changed during function execution
- `fp` -4 is the saved copy of the **callers** `fp`
- You move items between the data on the stack and the CPU registers using `ldr/str` instructions with **register base** (`fp` or sometimes the `sp`) **with offset addressing** (either register offset or immediate offset)



We will describe the sections of the stack frame in following slides

More to come

Week 9 Slide Preview

Bitwise (Bit to Bit) Operators in C

`output = ~a;`

a	~a
0	1
1	0

`output = a & b;`

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

& with 1 to let a **bit through**
& with 0 to **set a bit to 0**

`output = a | b;`

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

| with 1 to **set a bit to 1**
| with 0 to let a **bit through**

`output = a ^ b; //EOR`

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

^ with 1 **will flip the bit**
^ with 0 to let a **bit through**

Bitwise NOT

```
~ 1100
  ----
  0011
```

Bitwise AND

```
  0110
& 1100
  ----
  0100
```

Bitwise OR

```
  0110
| 1100
  ----
  1110
```

Bitwise EOR

```
  0110
^ 1100
  ----
  1010
```


x

Bitwise versus C Boolean Operators

Meaning	*Operator*	*Operator*	*Meaning*
Boolean AND	`a && b`	`a & b`	Bitwise AND
Boolean OR	`a || b`	`a	b`	Bitwise OR
Boolean NOT	`!b`	`~b`	Biwise NOT

Boolean operators **act on the entire value not the individual bits**

& versus &&

```
0x10 &   0x01 = 0x00 (bitwise)
0x10 &&  0x01 = 0x01 (Boolean)
```

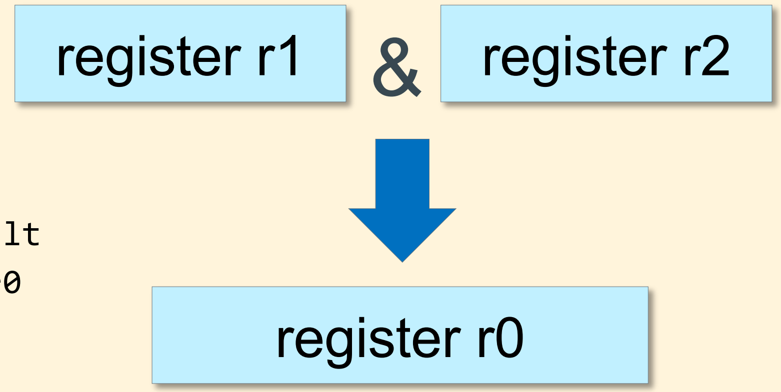
! versus ~

```
~0x01 = 0xfffffffe (bitwise)
!0x01 = 0x0 (Boolean)
```

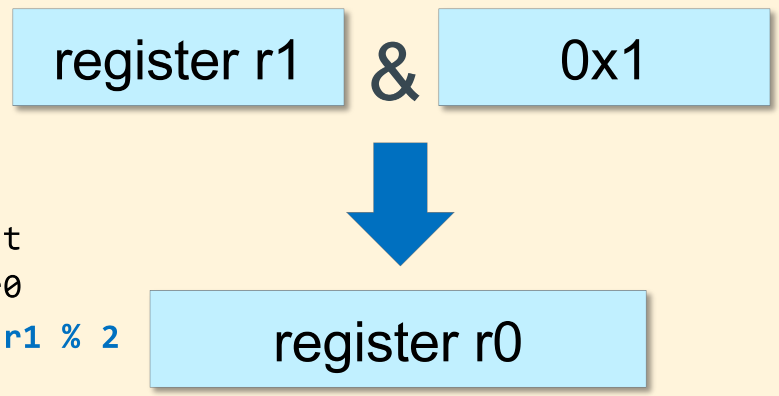

x

First Look: AND Registers

```
and   r0, r1, r2

// Copies all 32 bits
// of the bitwise result
// from r1 & r2 into r0
```




```
and   r0, r1, 1

// Copies all 32 bits
// of the bitwise result
// from r1 & 0x1 into r0
// Aside: This is r0 = r1 % 2
```



Bitwise Instructions

<op>	Rd	Rn	rot4	imm8
	destination	operand 1	Operand 2 constant	

<op>	Rd	Rn	Rm
	destination	operand 1	Operand 2

```
<op>   Rd,   Rn,  constant     // Rd = Rn <op> constant
<op>   Rd,   constant          // Rd = Rd <op> constant
<op>   Rd,   Rn,   Rm          // Rd = Rn <op> Rm
<op>   Rd,   Rm                // Rd = Rd <op> Rm
```

Bytes: 0 <= imm8 <= 255 + values from "rotating" rot 4 bits

Bitwise <op> description	*<op> Syntax*	*Operation*
Bitwise **AND**	`and` R_d, R_n, Op2	$R_d \leftarrow R_n$ & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	`bic` R_d, R_n, Op2	$R_d \leftarrow R_n$ & ~Op2
Bitwise **OR**	`orr` R_d, R_n, Op2	$R_d \leftarrow R_n$ | Op2
Exclusive **OR**	`eor` R_d, R_n, Op2	$R_d \leftarrow R_n$ ^ Op2
Bitwise **NOT**	`mvn` R_d, Op2	$R_d \leftarrow$ ~Op2

x

Bit Masks: Masking - 1

- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value
- `or:` 0 passes bit unchanged, 1 sets bit to 1
- `eor:` 0 passes bit unchanged, 1 inverts the bit
- `bic:` 0 passes bit unchanged, 1 clears it
- `and:` 0 clears the bit, 1 passes bit unchanged

mask force lower 16 bits to 1 "**mask on**" operation

```
orr  r1, r2, r3

DATA: r2 0xab ab ab 77

MASK: r3 0x00 00 ff ff lower half to 1

RSLT: r1 0xab ab ff ff
```

mask to invert the lower 8-bits "**bit toggle**" operation

```
eor  r1, r2, r3

DATA: r2 0xab ab ab 77

MASK: r3 0x00 00 00 ff flip LSB bits

RSLT: r1 0xab ab ab 88


MASK: r3 0x00 00 00 ff apply a 2nd time

RSLT: r1 0xab ab ab 77 original value!
```

x

Bit Masks: Masking - 2

mask to **extract top 8 bits** of r2 into r1

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff 00 00 00
RSLT: r1 0xab 00 00 00
```

mask to query the status of a bit "**bit status**" operation

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 01 is bit 0 set?
RSLT: r1 0x00 00 00 01 (0 if not set)
```

mask to force lower 8 bits to 0 "**mask off**" operation

```
and   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff ff ff 00 clear LSB
RSLT: r1 0xab ab ab 00
```

clear bit 5 to a 0 without changing the other bits

```
bic   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 20 clear bit 5 (0010)
RSLT: r1 0xab ab ab 57
```


x

Bit Masks: Masking - 3

mask to get **1's complement** operation

(like mvn)

```
eor   r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0xff ff ff ff
RSLT: r1 0x54 54 54 88
```

remainder (mod): num % d where n ≥ 0 and d = 2^k

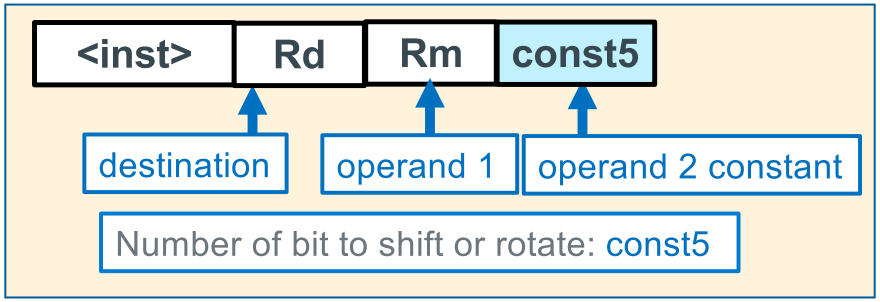
```
mask = 2^k  - 1 so for mod 2, mask = 2 -1 = 1
and  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 01 (mod 2 even or odd)
RSLT: r1 0xab 00 00 01 (odd)
```

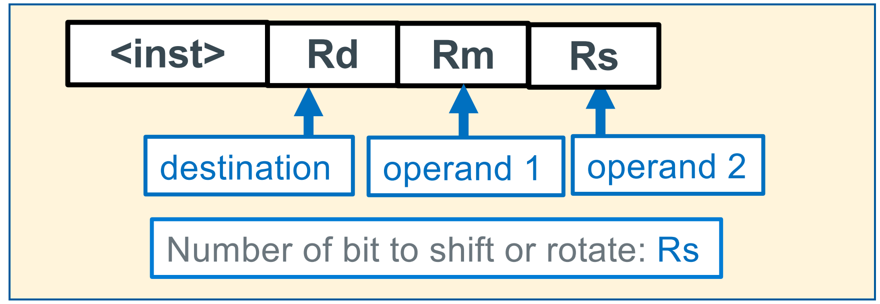
remainder (mod): num % d where n ≥ 0 and d = 2^k

```
mask = 2^k -1 so for mod 16, mask = 16 -1 = 15
and  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 0f (mod 16)
RSLT: r1 0xab 00 00 07 (if 0: divisible by)
```


x

Shift and Rotate Instructions





Instruction	Syntax	Operation	Notes	Diagram
Logical Shift Left	LSL R_d, R_m, *const5* LSL R_d, R_m, R_s	$R_d \leftarrow R_m$ << *const5* $R_d \leftarrow R_m$ << R_s	Zero fills shift: 0 - 31	
Logical Shift Right	LSR R_d, R_m, *const5* LSR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Zero fills shift: 1 - 32	
Arithmetic Shift Right	ASR R_d, R_m, *const5* ASR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Sign extends shift: 1 - 32	
Rotate Right	ROR R_d, R_m, *const5* ROR R_d, R_m, R_s	$R_d \leftarrow R_m$ ror *const5* $R_d \leftarrow R_m$ ror R_s	right rotate rot: 0 - 31	

x

Shift & Rotate Operations

```
asr r2, r0, 8

r0 0xab ab ab 77
r2 0xff ab ab ab (see the sign extend)
```

Test for sign
-1 if r0 negative

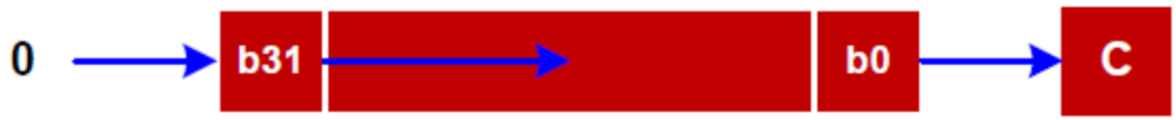
```
asr r2, r0, 31

r0 0xab ab ab 77
r2 0xff ff ff ff
```

Test for sign
0 if r0 positive

```
asr r2, r0, 31

r0 0x7b ab ab 77
r2 0x00 00 00 00
```


x

Shift & Rotate Operations




```
lsr r2, r0, 8

r0 0xab ab ab 77
r2 0x00 ab ab ab
```

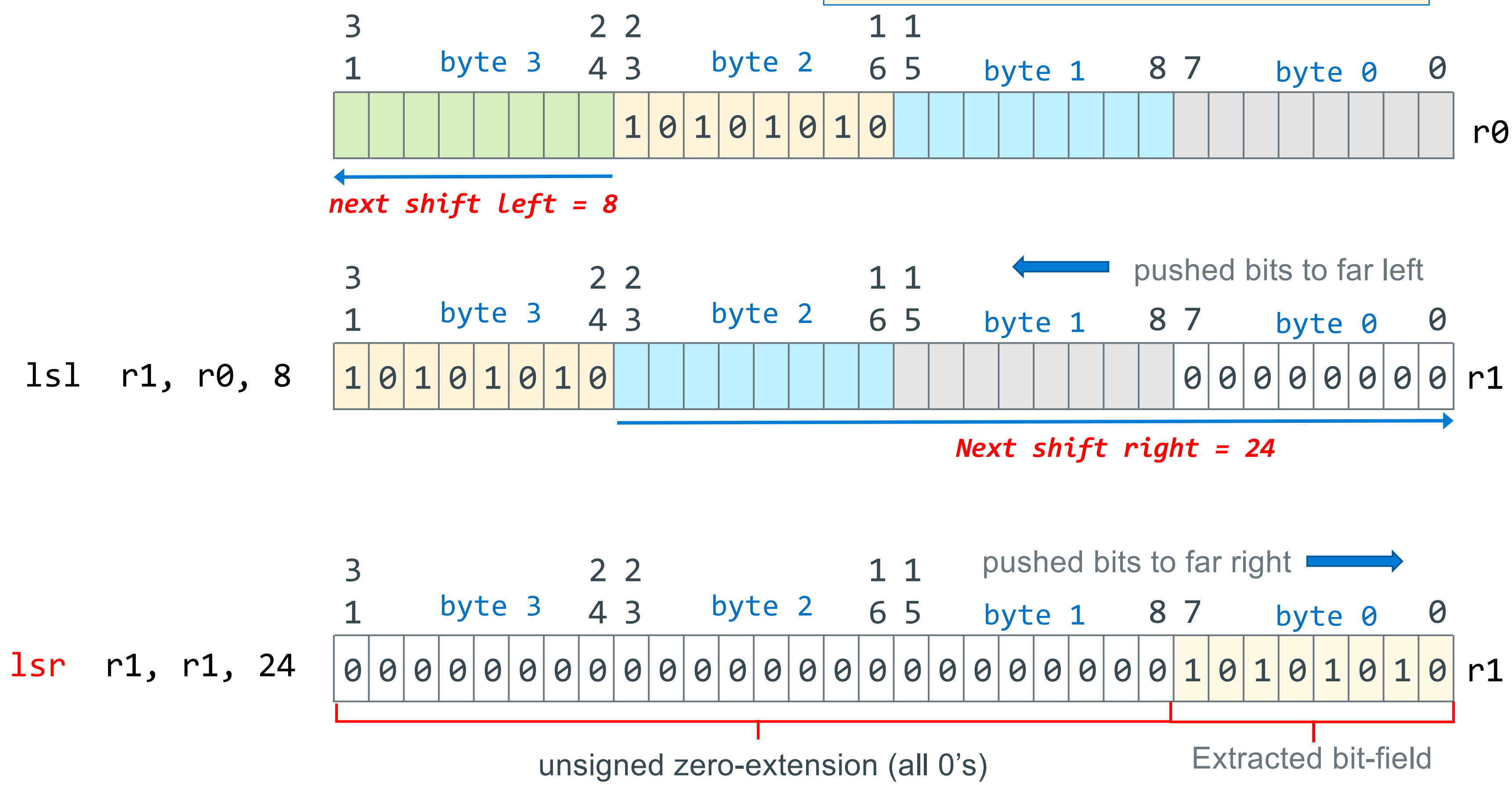

```
lsl r2, r0, 8

r0 0xab ab ab 77
r2 0xab ab 77 00
```


```
ror r2, r0, 8

r0 0xab ab ab 77
r2 0x77 ab ab ab
```


x

Extracting **Unsigned** Bitfields

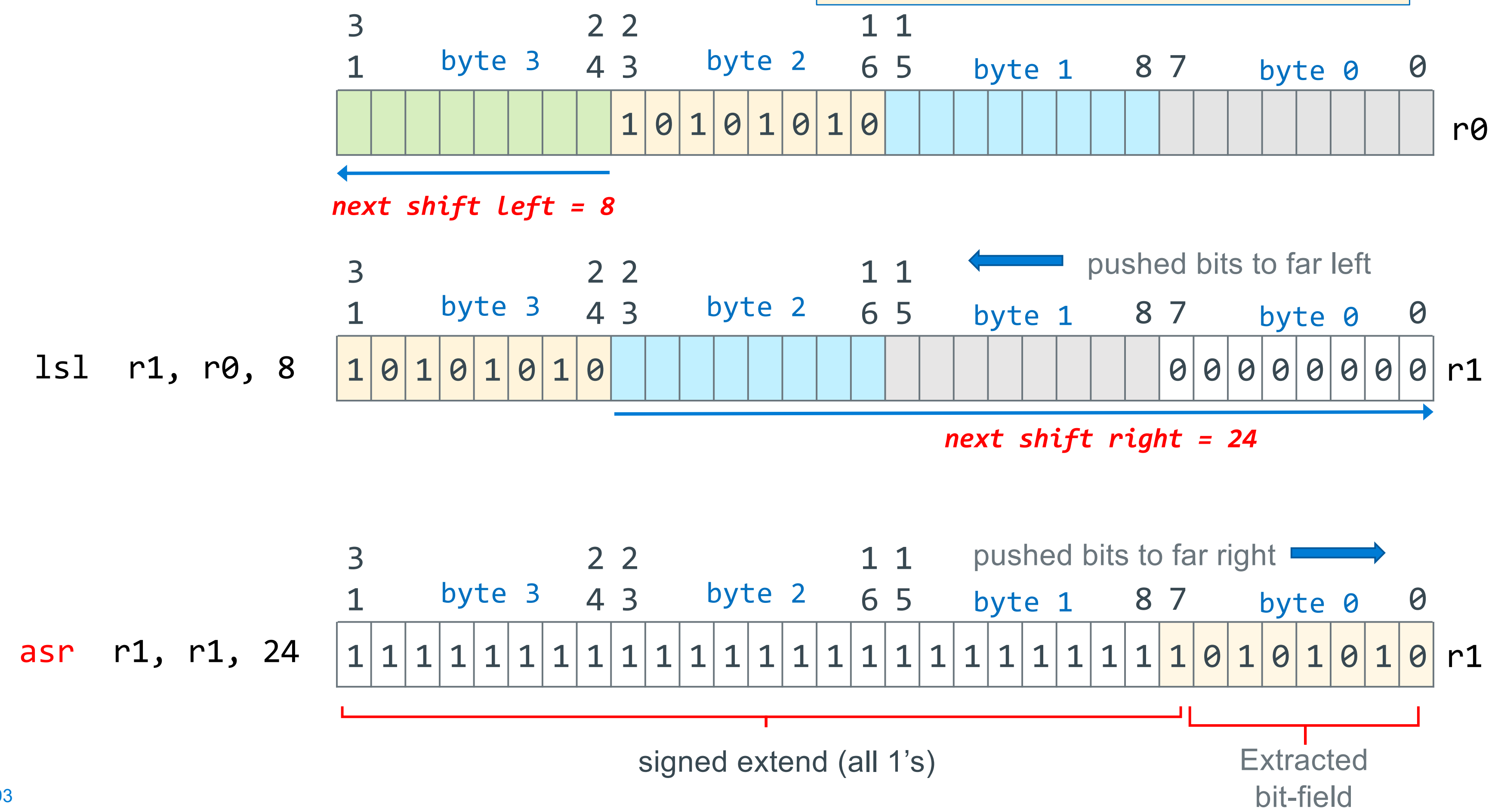
- Move byte 2 in r0 to byte 0 in r1



x

Extracting Signed Bitfields

- Move byte 2 in r0 to byte 0 in r1



x

Inserting Bitfields – Inserting Source Field into Destination Field

Task: Insert source into destination

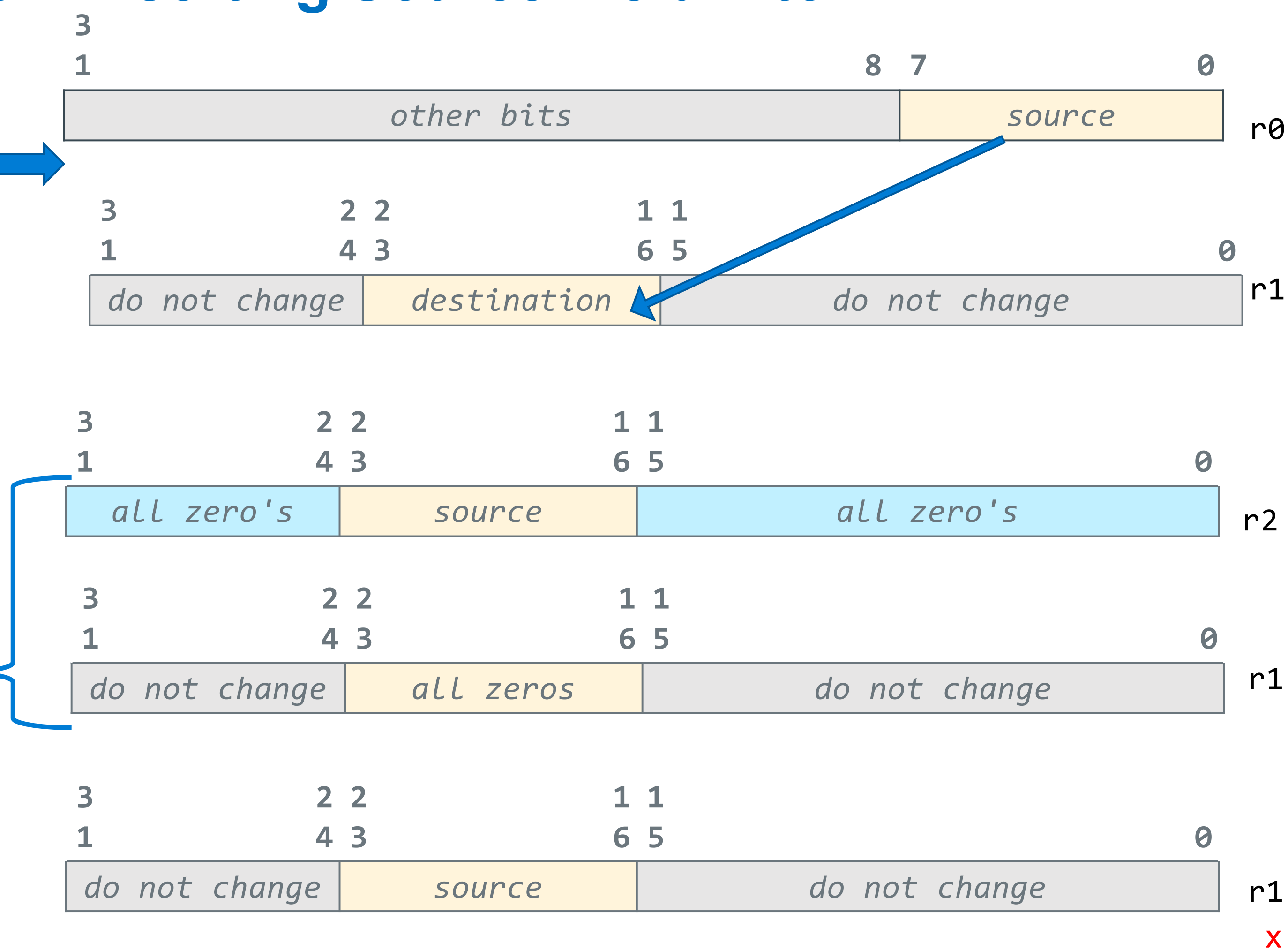
a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Approach

(1) isolate source field
(2) clear destination field
(3) Bitwise or together

```
orr    r1, r1, r2
```

results in



Creating a Mask - 1

```
option #1 (1 mask)
ldr    r3, =0x00ffff0000

for a 0 mask
ldr    r3, =0xff0000ffff
```

31							24	23							16	15							8	7							0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

```
option #2 (1 mask)
small mask
mov    r3, 255



lsl    r3, r3, 16
  or do
ror    r3, r3, 16
```

31							24	23							16	15							8	7							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	1	1	1	1

next shift left = 16 bits

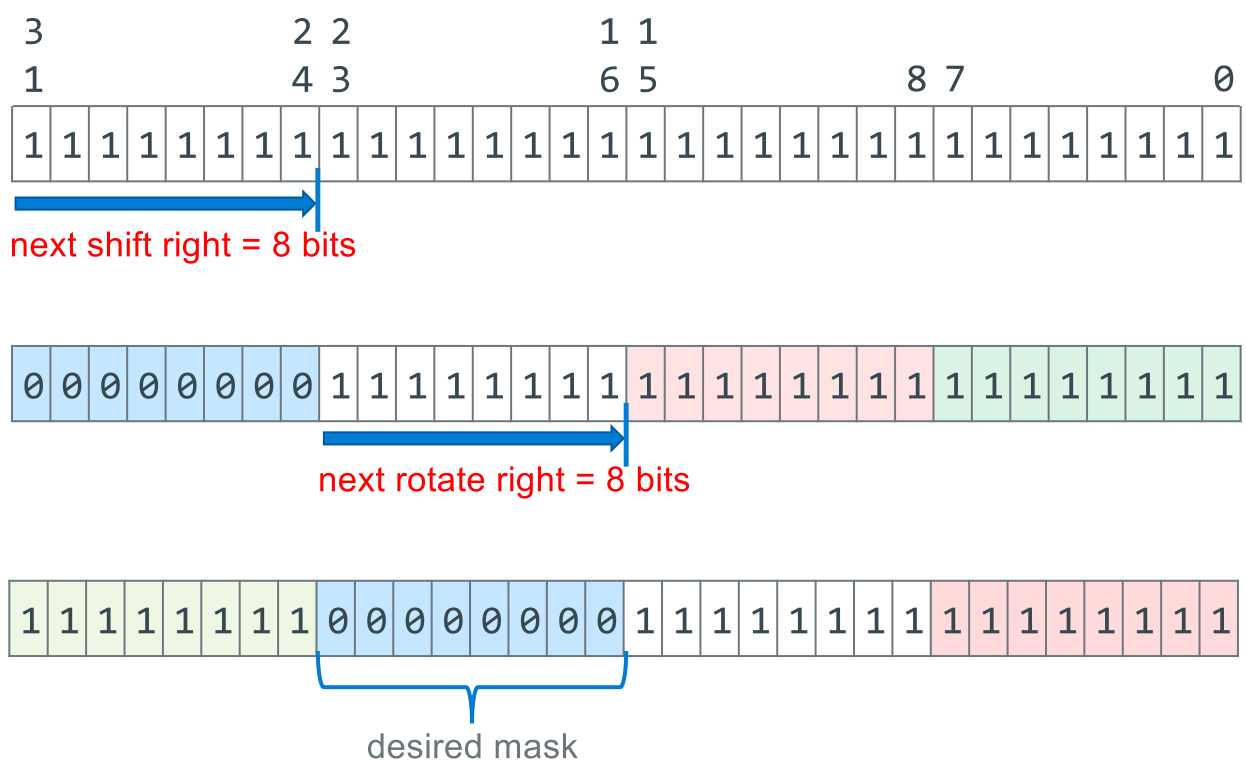
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

x

Creating a Mask- 0 mask

```
option #3
any size field

mov    r3, -1

asr    r3, r3, 8

ror    r3, r3, 8
```

number of bits you need in the mask, 8 for example



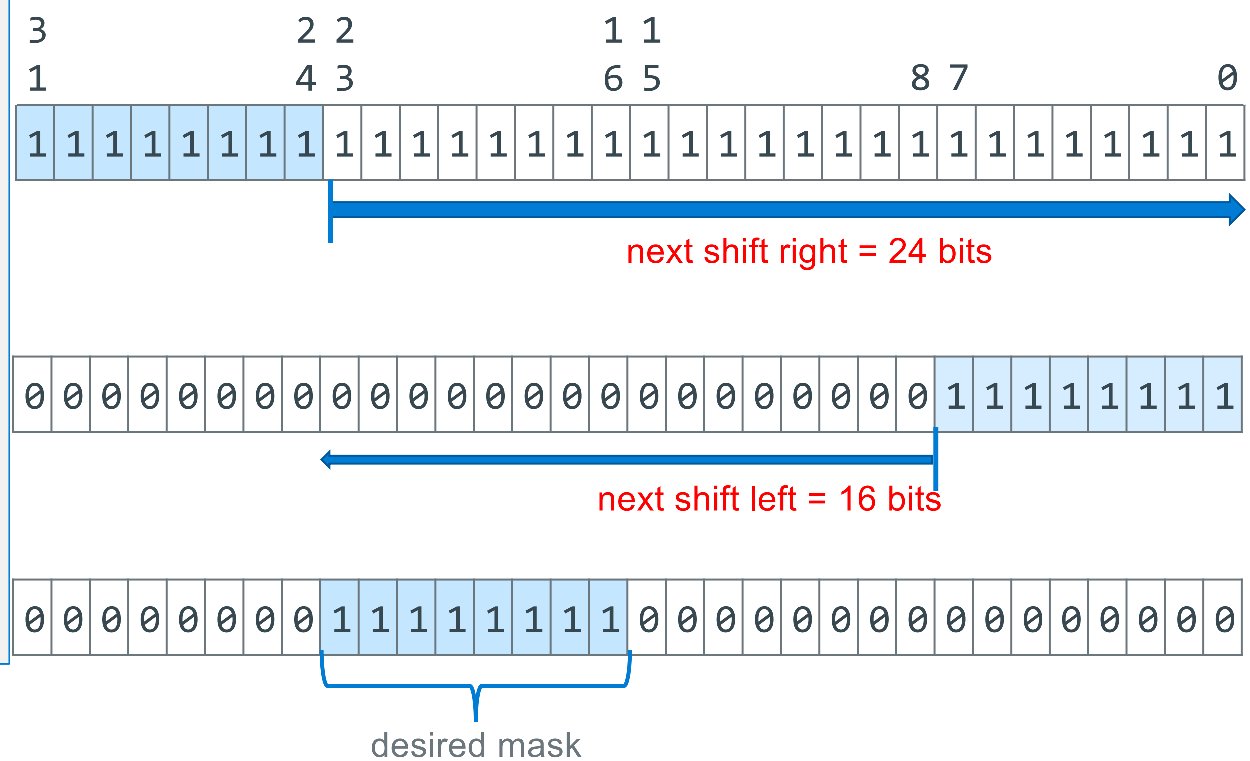
x

Creating a Mask- 1 mask

```
option #3
any size field

mov    r3, -1
```

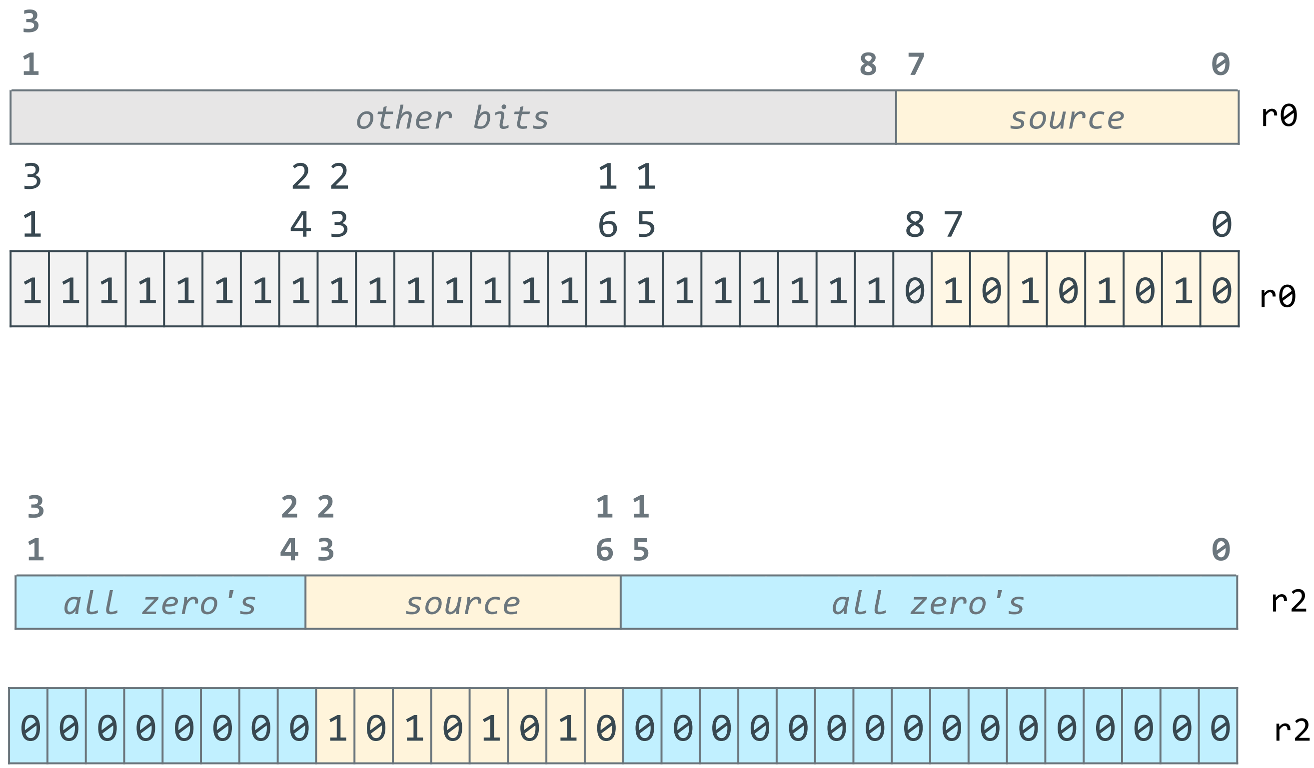
32 - number of bits you need in the mask, 8 for example is mask size

```
lsr    r3, r3, 24

lsl    r3, r3, 16
```



x

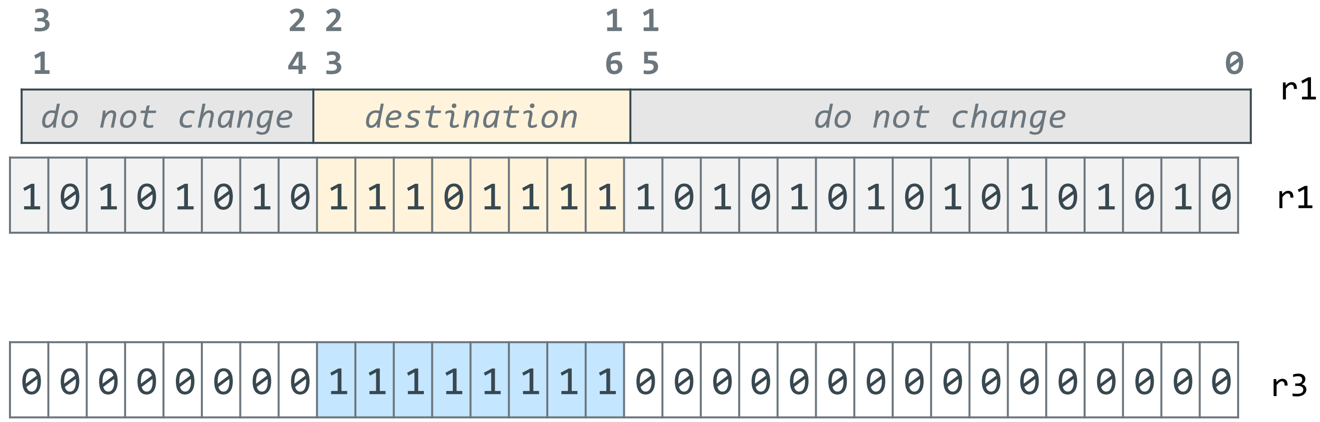
Inserting Bitfields – Isolating the Source Field




```
isolate source field

lsl     r2, r0, 24
lsr     r2, r2, 8
```


x

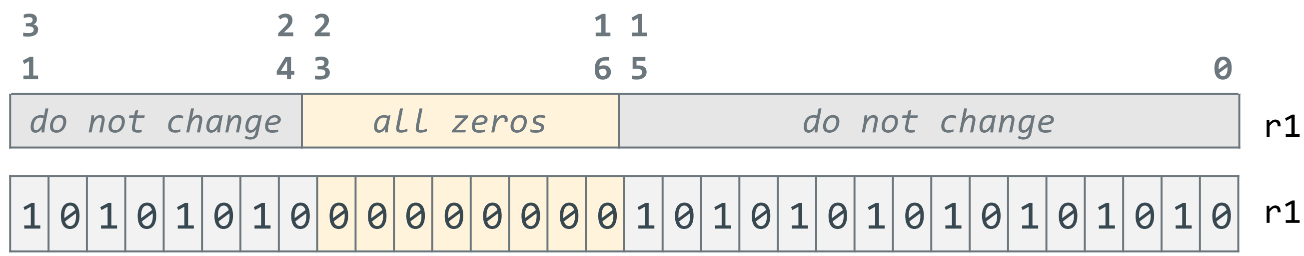
Inserting Bitfields – Clearing the Destination Field



create a 1 mask

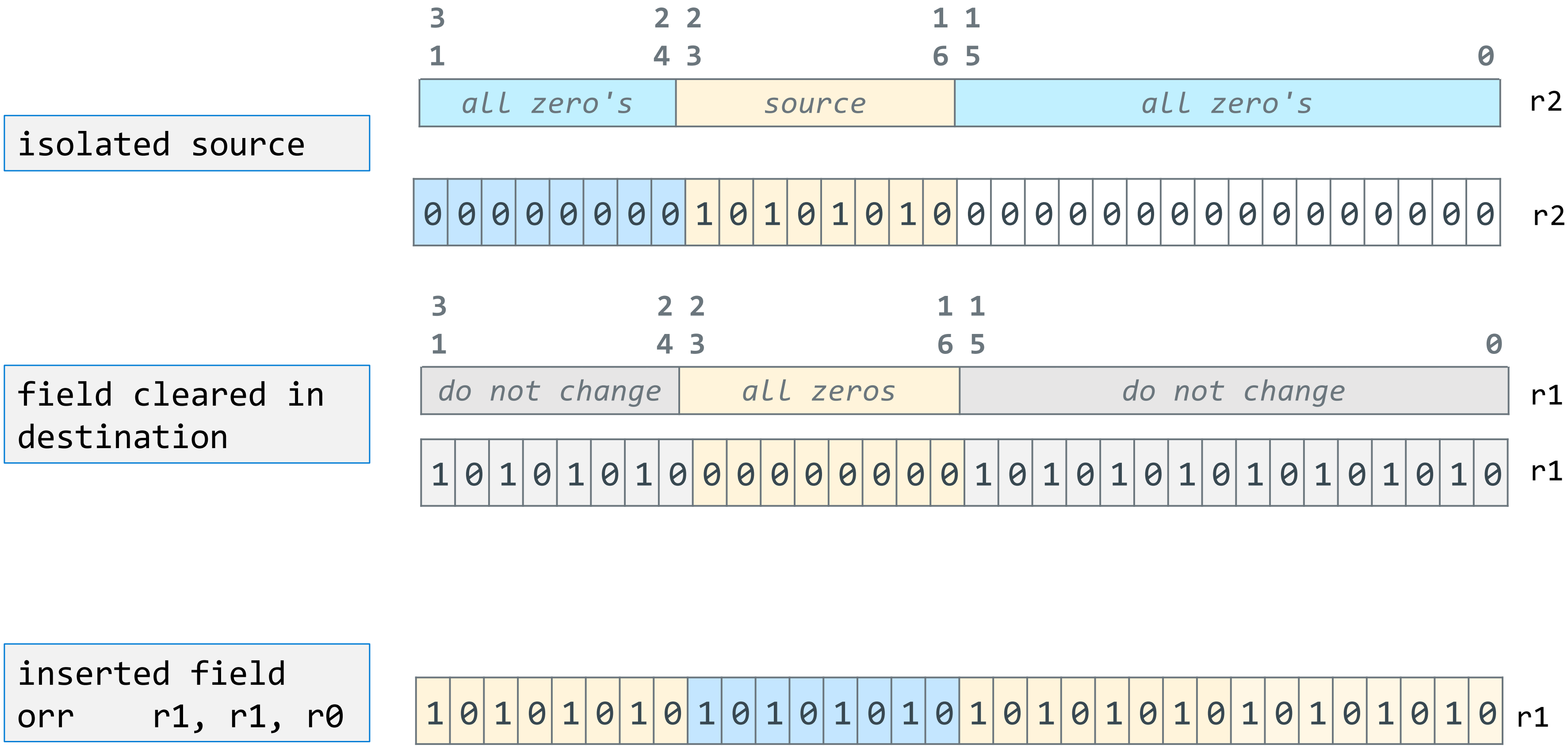
clear the destination field

```
bic    r1, r1, r3
```



x

Inserting Bitfields – Combining Isolated Source and Cleared Destination



x

Masking Summary

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

31 … 24	23 … 15 (isolates this field)	14 … 0	
0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	isolation mask

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

31 … 24	23 … 15 (clears this field)	14 … 0	
1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	clear a field mask

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

31 … 24	23 … 15	14 … 0
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 to get this edge into msb

lsr to get this edge into lsb

x