



Version 2.01

UCSD CSE 30

Computer Organization and Systems Programming

Lecture – 20

Final Exam Review

Keith Muller

Frontier Exascale



Logistics

Monday 6/10/2024 11:30am - 2:30pm (Arrive at 11:20 AM if possible)

Two Rooms

PCYNH 109

PCYNH 122

Seating Chart (look it up before arriving)

See Piazza post or canvas

Logistics

By appointment email: kmuller@ucsd.edu

Q&A zoom session as follows:

Friday: Jun 7, 2024 04:00 PM Pacific Time (US and Canada)

<https://ucsd.zoom.us/j/97367536447>

Meeting ID: 973 6753 6447

Saturday Jun 8, 2024 03:00 PM Pacific Time (US and Canada)

<https://ucsd.zoom.us/j/96404434647>

Meeting ID: 964 0443 4647

Sunday Jun 9, 2024 03:00 PM Pacific Time (US and Canada)

<https://ucsd.zoom.us/j/92978836290>

Meeting ID: 929 7883 6290

CSE 30 Final

380 points total
Spring 2024

DO NOT LOOK AT YOUR EXAM UNTIL TOLD TO DO SO

Name _____

PID _____

"While taking this examination, I have not witnessed any wrongdoing, nor have I personally violated any conditions of this course's integrity policy." [sign before turning in]

If you can honestly attest to the statement above, **write "I excel with Integrity" below and sign.** If you do not write and sign, the instructor will contact you by e-mail to request you clarify/explain.

Written Statement

Signature

LOOK!! There are references/notes for your use at the end of the exam

- 0. It is suggested you use a pencil and have a good eraser for taking this exam.**
- 1. Place your PID at the top of each page.**
- 2. FILL IN THE BUBBLES COMPLETELY.** Bubbles that have just dots or distinct lines will not qualify for any points!
- 3. For fill-in the box questions, your answers must be within the box provided.** Answers placed outside the box will not qualify for any points!
- 4. No calculators or other electronic devices allowed.**
- 5. Three (3) pages of 8.5" x 11" notes (both sides) allowed**
- 6. Be as neat as possible.** It's your responsibility to make your answers legible. If we cannot read it, **we will not give you credit for your answer.**
- 7. Read each question carefully, start working on it, and then reread it.** A large number of mistakes occur from not carefully reading the question.
- 8. Be sure to place all your answers on the exam in the spaces provided. We will not look at scratch paper for answers.**
- 9. All questions refer to the pi-cluster (32-bit ARM v6 and Linux) environment and the version of C presented in lecture (unless stated otherwise).**

<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
----------------------------------	----------------------------------	-----------------------	-----------------------	----------------------------------

Asking me to Count the most dots...
WILL NOT Get Credit

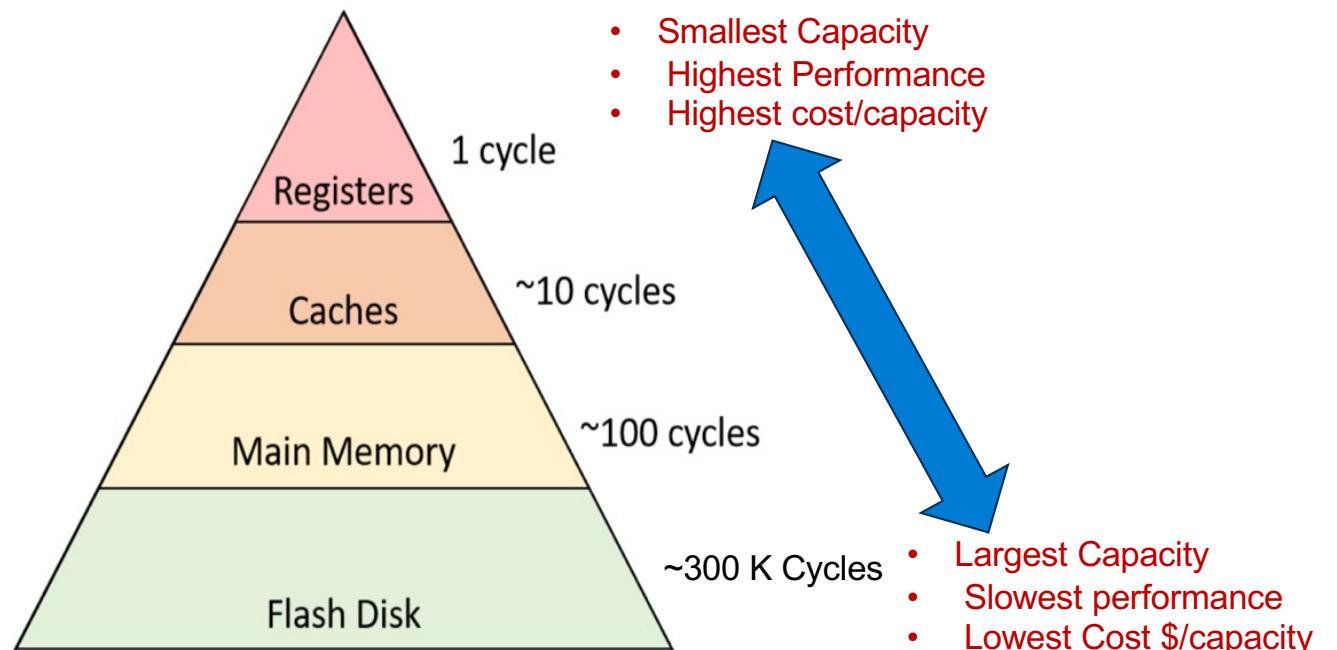
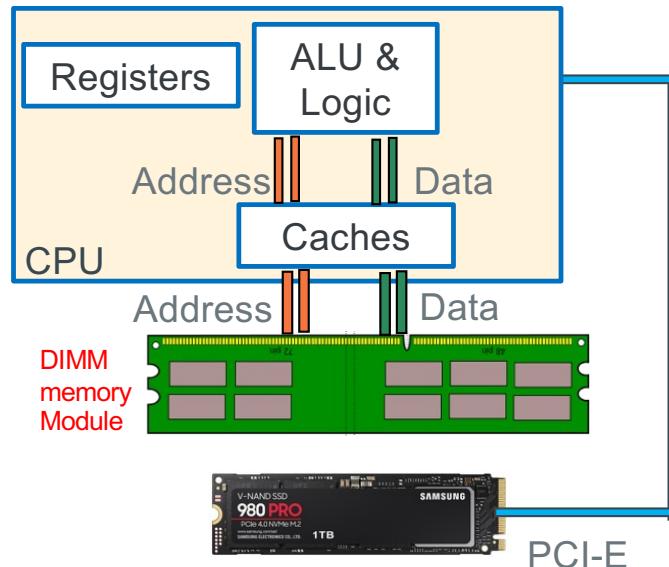
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
-----------------------	-----------------------	-----------------------	-----------------------	----------------------------------

Nice Job, Will get Credit

Memory Triangle: Hardware Cost/Performance/Capacity Tiers

Assume each instruction takes 1 clock cycle

Clock cycle =~ time to access; larger is **slower**



Design Goal: best performance at the lowest (or specific) cost

Other goals: performance/energy (operating cost), expandability, high margin (price/cost)

Variables in C

- **Global variables**
 - Defined at file scope (outside of a block)
 - have static storage duration
 - global variables defined without an initial value default to 0 (set prior to program execution start)
 - global variables defined with an initial value are set at program start
- **Local (block scope, or automatic) variables** (including function parameter variables)
 - Defined at block scope (inside of a block)
 - have automatic storage duration, with one exception (see below)
 - block scope variables defined without an initial value have an unspecified initial value
 - block scope variables defined with an initial are set each time by code when the block is entered
 - All block scope variables become unspecified at block exit
- Variable definitions preceded by the keyword static always have static storage duration including variables defined with block scope (when used global variables it restricts scope – later slides)

```
int global;           // global with static storage duration, initial value = 0
int foo(void)
{
    static int s;    // "Local" with static storage duration, initial value = 0
    int x;          // "Local" with automatic storage duration
}
```

Example: Block scope (local) static storage duration variables

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int foo(void)
{
    static int s; //static storage duration, set to 0 at program start
    return s += 1;
}

int main(void)
{
    for (int i = 0; i < MAX; i++)
        printf("%d ", foo());
    printf("\n");
    return EXIT_SUCCESS;
}
```

```
% ./a.out
1 2 3 4 5
%
```

Conditional Statements (if, while, do...while, for)

- C conditional test expressions: 0 (NULL) is FALSE, any non-0 value is TRUE
- C comparison operators (==, !=, >, etc.) evaluate to either 0 (false) or 1 (true)
- Legal in Java and in C:

```
i = 0;  
if (i == 5)  
    statement1;  
else  
    statement2;
```

Which statement is executed after the if statement test?

- Illegal in Java, but legal in C (often a typo!):

```
i = 0;  
if (i = 5)  
    statement1;  
else  
    statement2;
```

Assignment operators evaluate to the value that is assigned, so....
Which statement is executed after the if statement test?

Program Flow – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal** evaluation

```
if ((x == 5) || (y > 3)) // if x == 5 then y > 3 is not evaluated
```



- Each expression argument is evaluated **in sequence** from left to right including any **side effects** (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated (*for performance*)

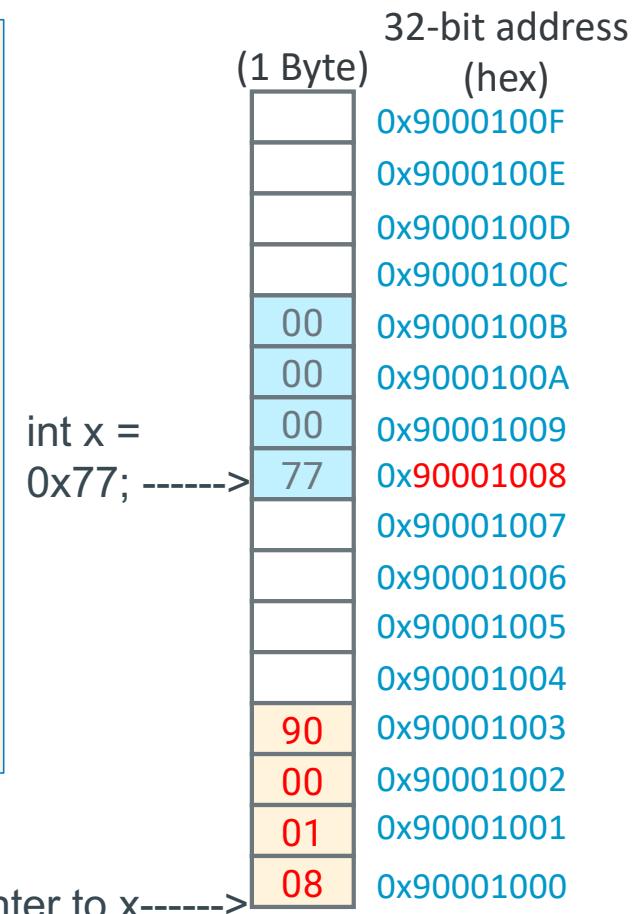
Program Flow – Short Circuit or Minimal Evaluation

```
if ((a != 0) && func(b))      // if a is 0, func(b) is not called  
    do_something();
```

```
// if (((x > 0) && (c == 'Q')) evaluates to non zero (true)  
// then (b == 3) is not tested  
  
while (((x > 0) && (c == 'Q')) || (b == 3)) {  // c short circuit  
    x = x / 2;  
    if (x == 0) {  
        return 0;  
    }  
}
```

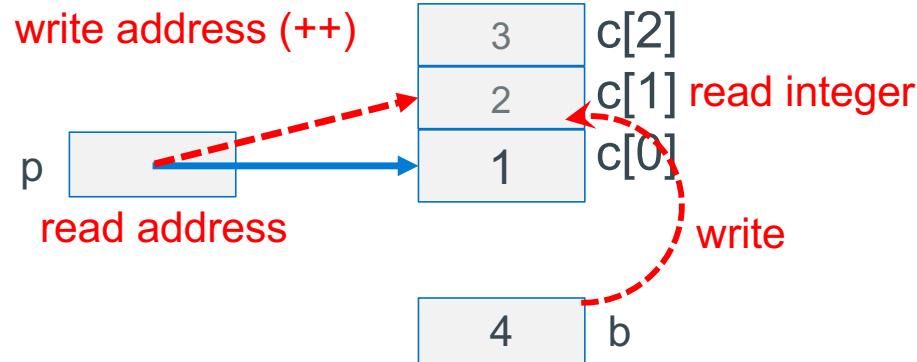
Address and Pointers

- An **address** refers to a location in memory, the **lowest or first byte** in a **contiguous sequence of bytes**
- A **pointer** is a **variable** whose **contents** (or value) can be properly used as an **address**
 - The **value in a pointer** *should* be a **valid address allocated to the process** by the **operating system**
- The **variable x** is at **memory address 0x90001008**
- The **variable pt** is at **memory location 0x90001000**
- The **contents** of **pt** is the **address of x** **0x90001008**

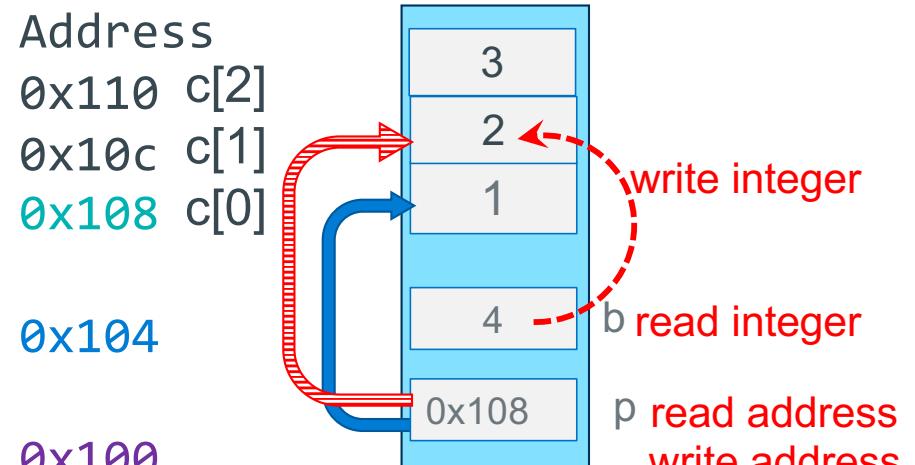


Each use of a * operator results in one additional read : both sides

```
int c[] = {1, 2, 3};  
int b = 4;  
int *p;  
  
p = c;  
*(++p) = b;
```

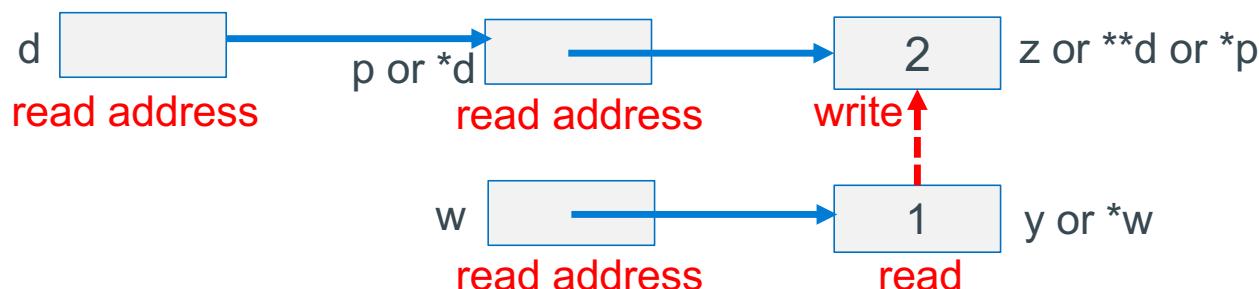
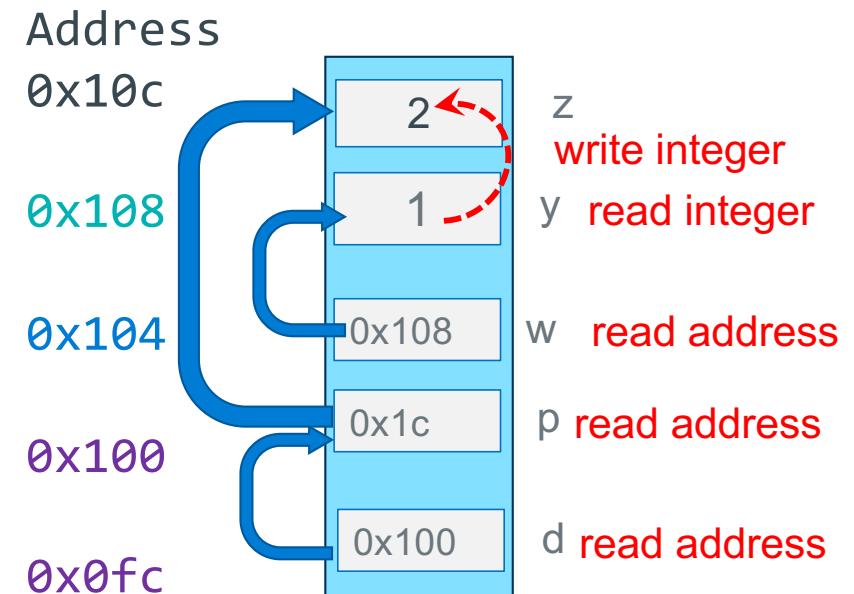


2 reads and 2 writes



Double Indirection: Lside

```
int z = 2;  
int y = 1;  
int *w;  
int *p;  
int **d;  
  
p = &z;  
w = &y;  
d = &p;  
**d = *w;
```

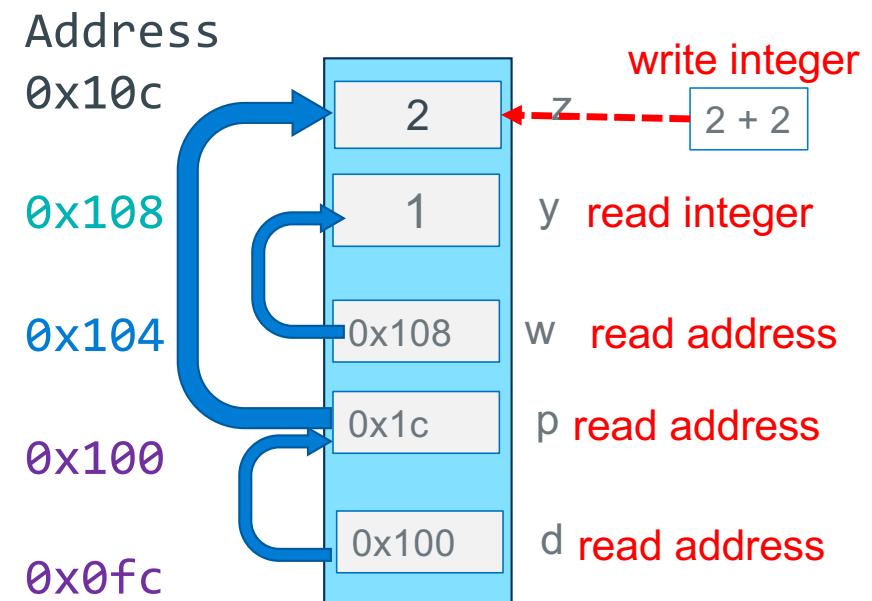
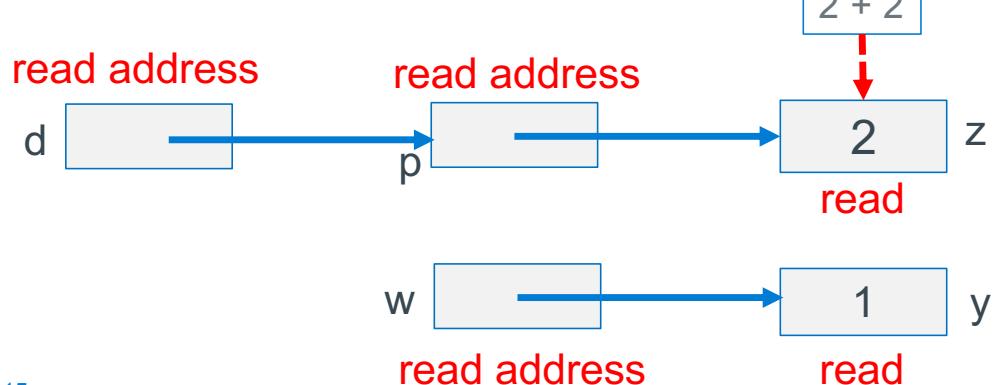


Double Indirection: Rside

```

int z = 2;
int y = 1;
int *w;
int *p;
int **d;

p = &z;
w = &y;
d = &p;
**d = **d + *p;
    
```

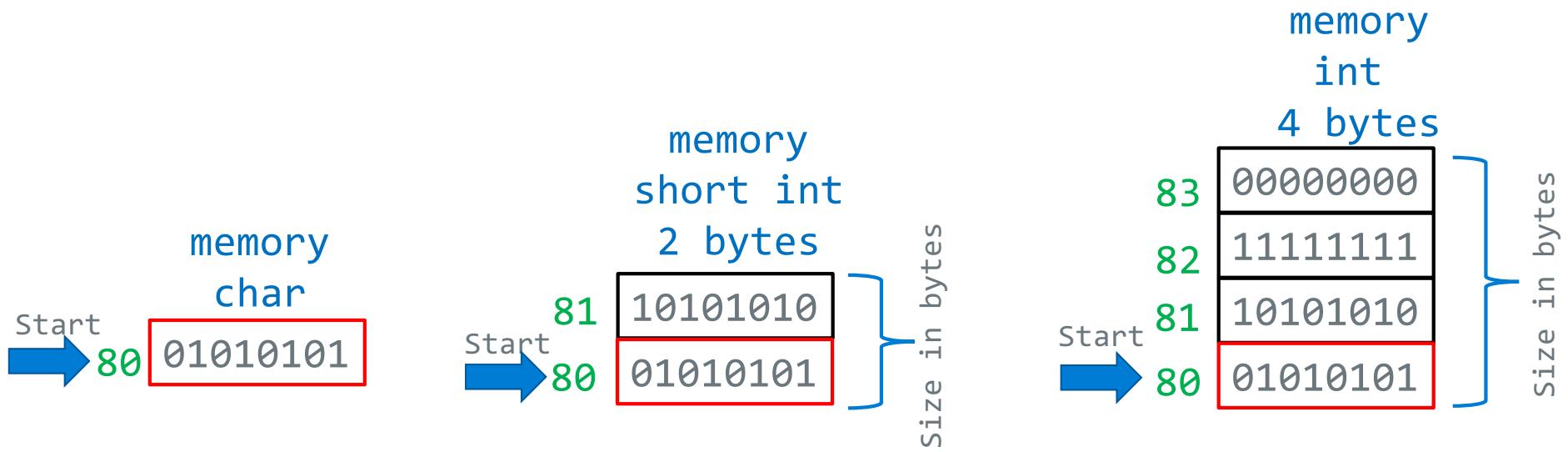


Important Observation

**d on Lside is two reads
 **d on Rside is three reads

Variables in Memory: Size and Address

- The number of **contiguous bytes** a variable uses is based on the **type** of the variable
 - Different variable types require different numbers of contiguous bytes
- **Variable names** map to a starting address in memory
- Example Below: Variables all starting at address 0x80, each box is a byte



sizeof(): Variable Size (number of bytes) Operator

```
#include <stddef.h>
/* size_t type may vary by system but is always unsigned */
```

sizeof() operator returns a value of type **size_t**:

the number of bytes used to store a variable or variable type

```
size_t size = sizeof(variable_type);
```

or

```
size_t size = sizeof(variable_name); // preferred!
```

- The argument to sizeof() is often an expression:

```
size = sizeof(int * 10);
```

- reads as:

- number of bytes required to store 10 integers (an array of [10])

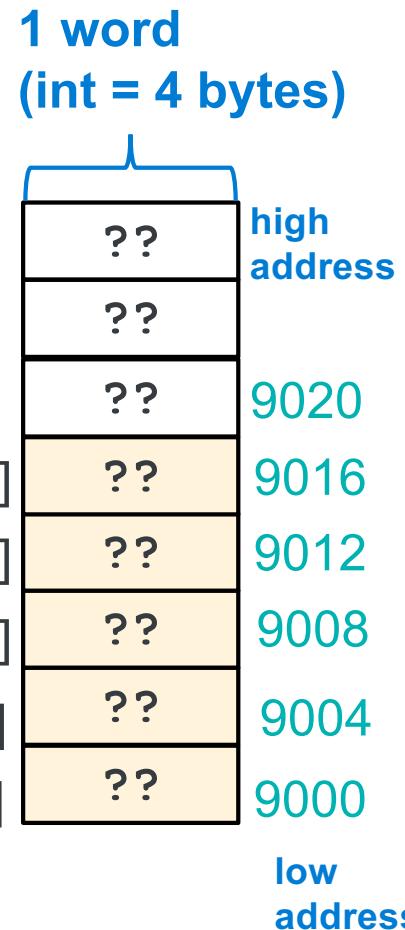
Accessing Arrays Using Indexing

- `name [index]` selects the `index` element of the array
 - `index` should be `unsigned`
 - Elements range from: 0 to `count – 1` (`int x[count];`)
- `name [index]` can be used as an `assignment target` or as a `value in an expression`

```
int a[2] = {1, 2};  
a[0] = a[1];
```
- Array name (by itself with no []) on the `Rside` evaluates to the address of the first element of the array

```
int b[5];  
int *p = b;
```

p 9000 → b[0]



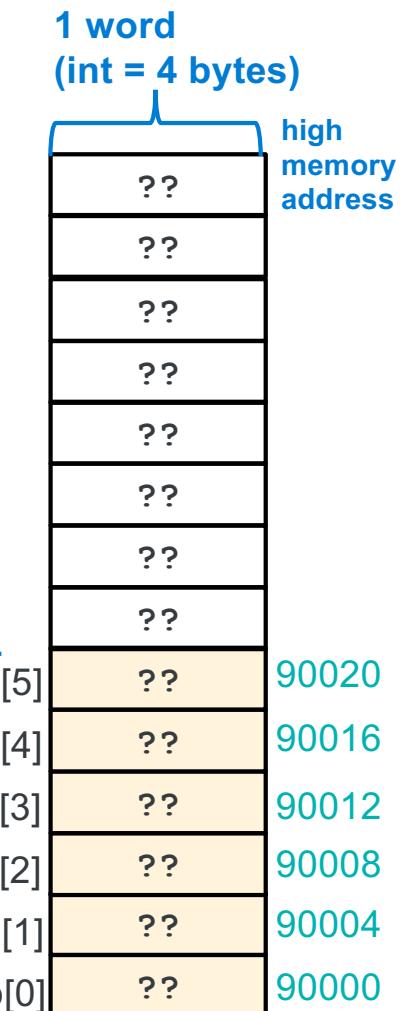
How many elements are in an array?

- The number of elements of space allocated to an array (called element count) and indirectly the total size in bytes of an array is not stored anywhere!!!!!!
- An **array name** is just the **address of the first element in a block of contiguous memory**
 - So, an array does not know its own size!

```
#define SZ 6
int block[SZ];           // you specify the array has SZ elements
int indx;                 // use when SZ is defined

for (indx = 0; indx < SZ; indx++)
    block[indx] = 0;
```

```
int b[6];
```



Pointers and Arrays - 1

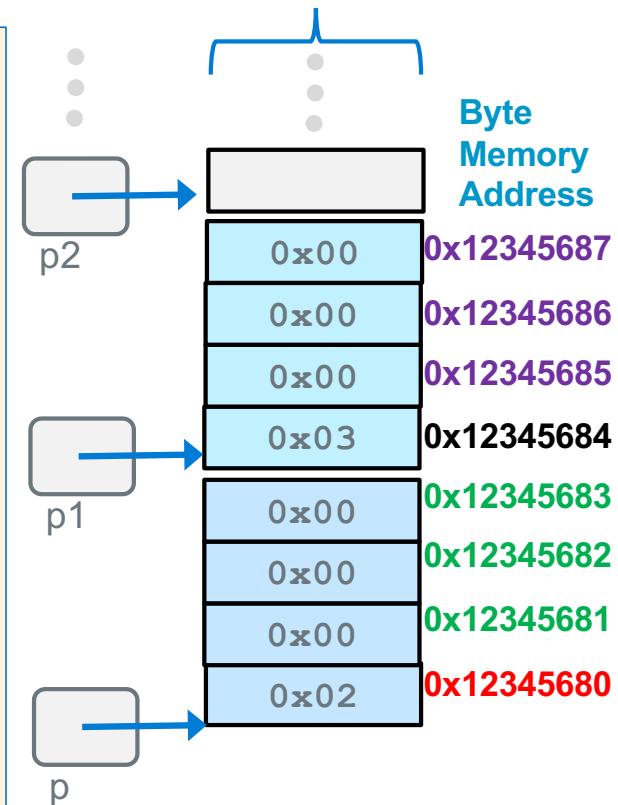
- A few slides back we stated: **Array name (by itself)** on the Rside evaluates to the **address of the first element of the array**

```
int buf[] = {2, 3, 5, 6, 11};
```

- Array indexing syntax (`[]`) an operator that performs *pointer arithmetic*
- **buf** and **&buf[0]** on the **Rside** are equivalent, **both evaluate** to the address of the first array element

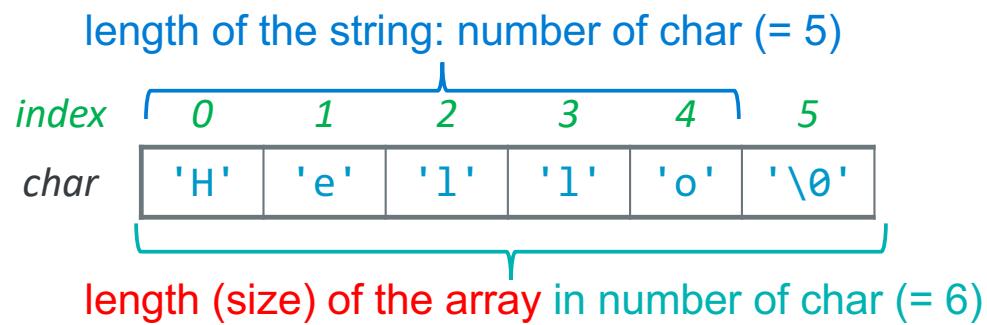
```
int *p = buf;           // or int *p = &buf[0];
int *p1 = &buf[1];
int *p2 = &buf[2];
int *p3 = &buf[3];
```

1 byte Memory Content
One byte per row



C Strings - 1

- C does not have a dedicated type for strings
- Strings are an **array of characters terminated by a sentinel termination character**
- '\0' is the **Null termination character**; has the **value of zero (do not confuse with '0')**
- An **array of chars** contains a **string only when** it is terminated by a '\0'
- **Length of a string** is the **number of characters in it, not including the '\0'**
- Strings in C are not objects
 - No embedded information about them, you just have a name and a memory location
 - You cannot use + or += to concatenate strings in C
 - For example, you must **calculate string length** using code at runtime looking for the sentinel



Output Parameters (Mimics Call by Reference)

- Passing a pointer parameter with the **intent** that the called function will use the address **it** to store values for use by the **calling function**, then **pointer parameter** is called an **output parameter**
- To pass the address of a variable **x** use the **address operator (&x)** or the contents of a pointer variable that points at **x**, or the name of an array (the arrays address)
- To receive an **address** in the called function, define the corresponding parameter type to be a pointer (add *****)
 - It is common to describe this method as: "pass a **pointer to x**"
- C is still using "**pass by value**"
 - we pass the **value of the address/pointer** in a **parameter copy**
 - **The called routine** uses the address to change a variable in the caller's scope

```
void inc(int *p);  
int  
main(void)  
{  
    int x = 5;  
    → inc(&x);  
  
}  
  
void  
inc(int *p) ←  
{  
  
}
```

Array Parameters: Call-By-Value or Call-By-Reference?

- **Type []** array parameter is automatically “**promoted**” to a pointer of type **Type ***, and a copy of the **pointer** is **passed by value**

the name is the address, so this is passing a pointer to the start of the array

```
void passa(int []);  
int main(void)  
{  
    int numbers[] = {9, 8, 1, 9, 5};  
  
    passa(numbers);  
    printf("numbers size:%lu\n", sizeof(numbers)); // 20  
    return EXIT_SUCCESS;  
}
```

```
void passa(int a[])  
{  
    printf("a size:%lu\n", sizeof(a)); // 4  
    return;  
}
```

IMPORTANT:
See the size difference 20 in main() in passa() is 4 bytes (size of a pointer) on pi-cluster and 8 on ieng6

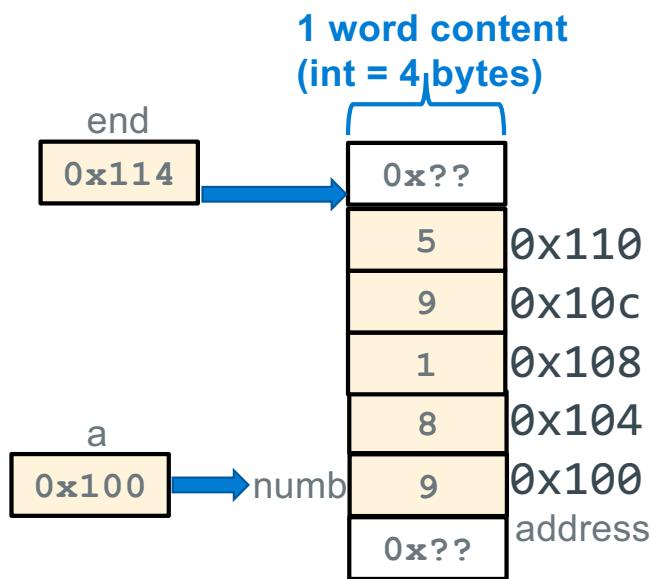
- Call-by-value pointer (callee can change the pointer parameter to point to something else!)
- Acts like call-by-reference (called function can change the contents caller's array)

Arrays As Parameters, Approach 1: Pass the size

Two ways to pass array size

1. pass the count as an additional argument
2. add a **sentinel element** as the last element

remember you can only use `sizeof()` to calculate element count where the array is defined



```
int sumAll(int *a, int size);
int main(void)
{
    int numb[] = {9, 8, 1, 9, 5};
    int cnt = (int)(sizeof(numb)/sizeof(numb[0]));

    printf("sum is: %d\n", sumAll(numb, cnt));
    return EXIT_SUCCESS;
}
```

```
int sumAll(int *a, int size)
{
    int sum = 0;
    int *end;
    end = a + size;

    while (a < end)
        sum += *a++;
    return sum;
}
```

same as:
`sum = sum + *a;`
`a++;`

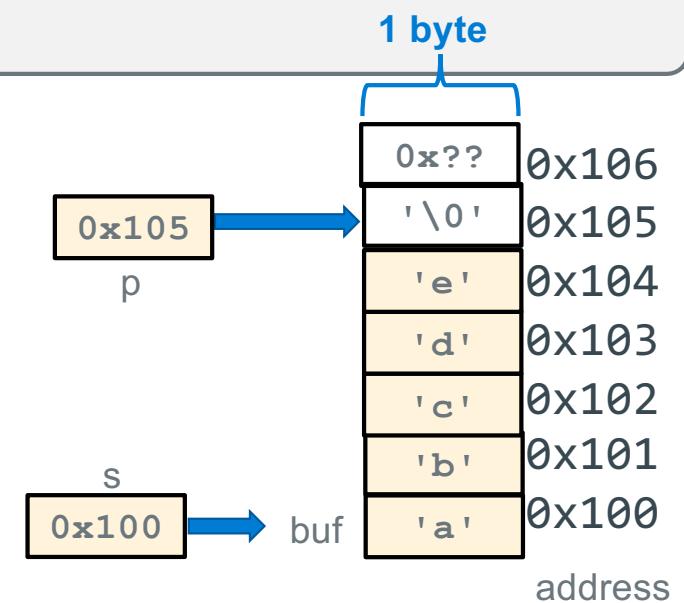
Arrays As Parameters, Approach 2: Use a sentinel element

- A **sentinel** is an element that contains a value that is not part of the normal data range
 - Forms of 0 are often used (like with strings). Examples: '\0', NULL

```
int strlen(char *a); // returns number of chars in string, not counting \0
int main(void)
{
    char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // or buf[] = "abcde";

    printf("Number of chars is: %d\n", strlen(buf));
    return EXIT_SUCCESS;
}
```

```
/* Assumes parameter is a terminated string */
int strlen(char *s)
{
    char *p = s;
    if (p == NULL)
        return 0;
    while (*p != '\0')
        p++;
    return (p - s);
}
```



Returning a Pointer To a Local Variable (Dangling Pointer)

- There are many situations where a function will return a pointer, but a function must never return a pointer to a memory location that is no longer valid such as:
 1. Address of a passed parameter copy as the caller may or will deallocate it after the call
 2. Address of a local variable (automatic) that is invalid on function return
- These errors are called a **dangling pointer**

n is a parameter with
the scope of bad_idea
it is no longer valid after
the function returns

```
int *bad_idea(int n)
{
    return &n; // NEVER do this
}
```

a is an automatic (local)
with a scope and
lifetime within
bad_idea2
a is no longer a valid
location after the
function returns

```
int *bad_idea2(int n)
{
    int a = n * n;
    return &a; // NEVER do this
}
```

```
/*
 * this is ok to do
 * it is NOT a dangling
 * pointer
 */
int *ok(int n)
{
    static int a = n * n;
    return &a; // ok
}
```

String Literals, Mutable and Immutable arrays - 1

- mess1 is a **mutable array** (type is `char []`) with enough space to hold the string + '`\0`'

```
char mess1[] = "Hello World";
*(mess1 + 5) = '\0'; // shortens string to "Hello"
```

mess1[] Hello World\0

- mess2 is a **pointer** to an **immutable array** with space to hold the string + '`\0`'

```
char *mess2 = "Hello World";      // "Hello World" read only string literal
                                    // mess2 is a pointer NOT an array!
*(mess2 + 1) = '\0';            // Not OK (bus error)
```



- mess3 is a **pointer** to a **mutable array**

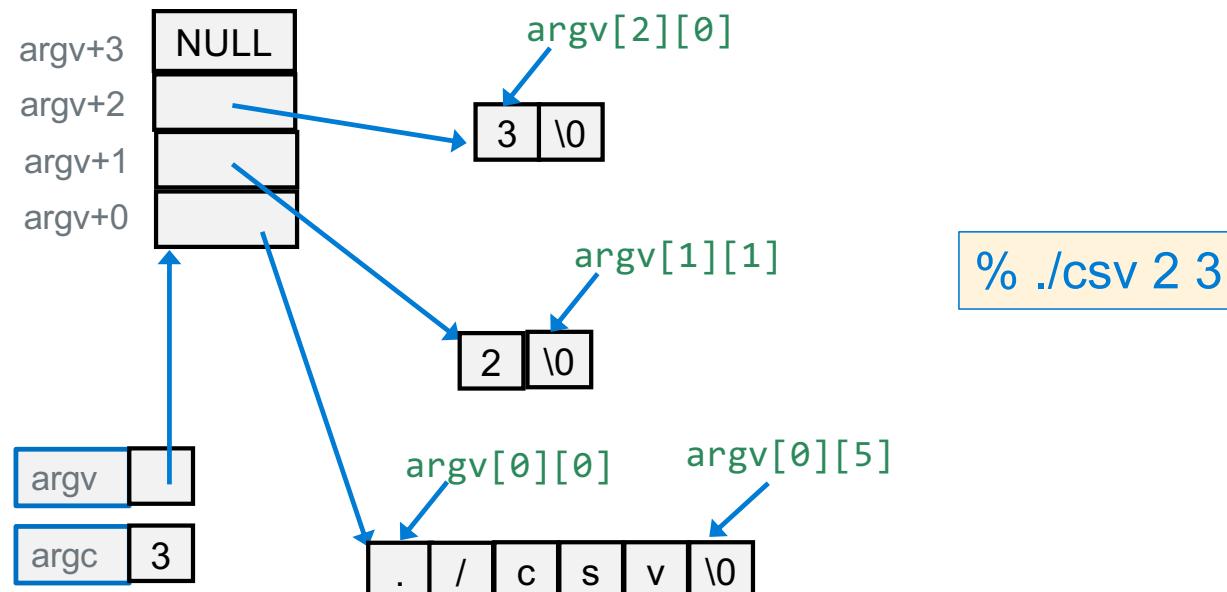
```
char *mess3 = (char []) {"Hello World"}; // mutable string
*(mess3 + 1) = '\0';                  // ok
```

using the cast (`char []`) makes it mutable

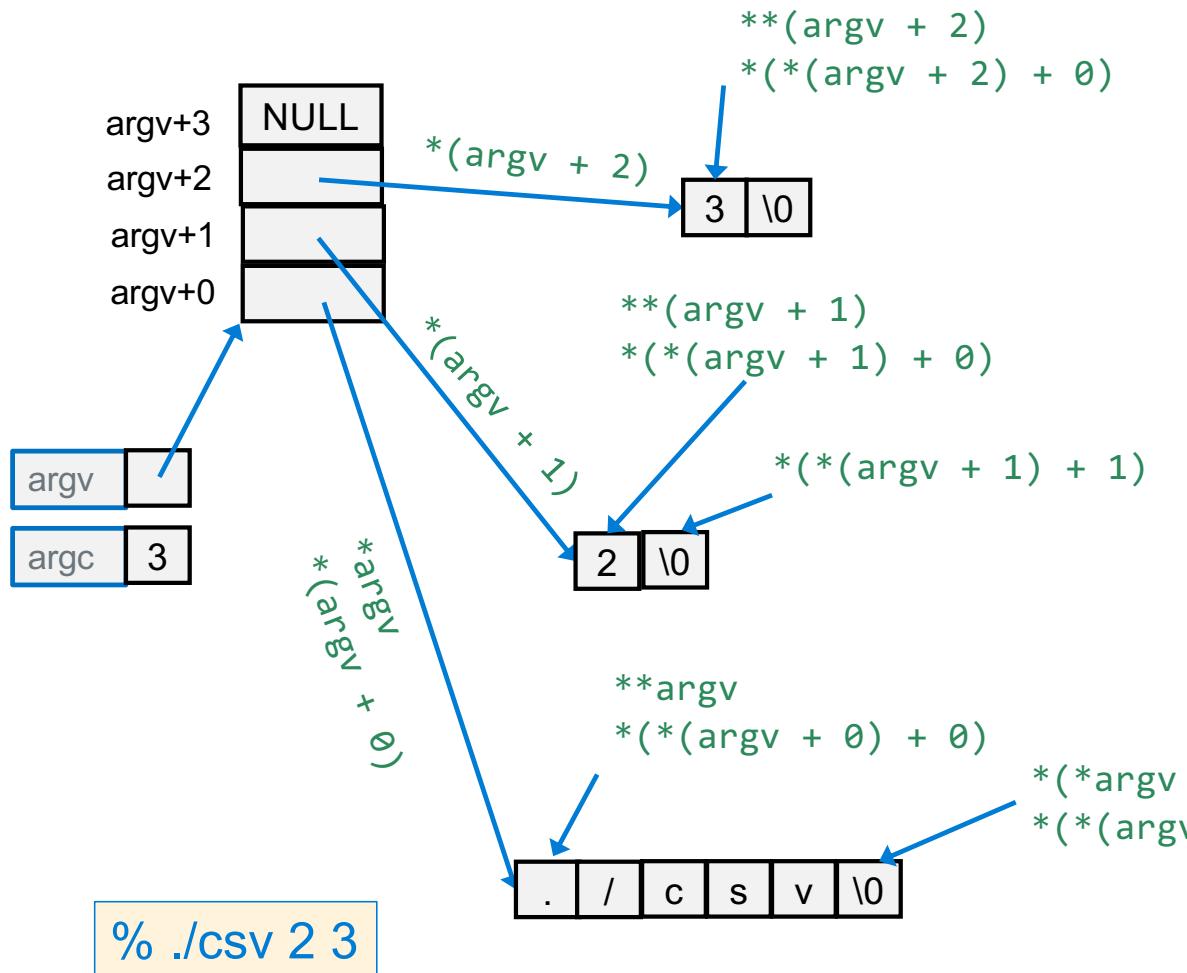


main() Command line arguments: argc, argv

- Arguments are passed to main() as a pointer to an array of pointers to char arrays (strings) (**argv)
Conceptually: % *argv[0] *argv[1] *argv[2] ...
- argc is the number of VALID elements (they point at something)
- *argv (argv[0]) is usually the name of the executable file (% ./vim file.c)
- argv[argc] or *(argv + argc) always contains a NULL (0) sentinel
- argv elements point at mutable (fixed size) strings!



Accessing argv char at a time



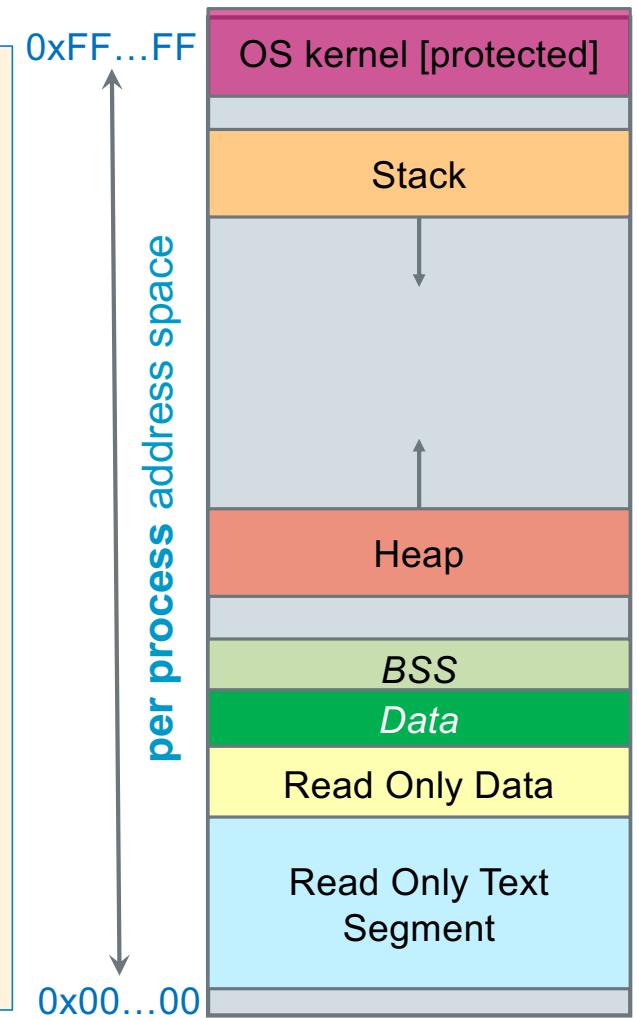
- argv is a pointer variable, whose contents can be changed
- it is not an array name, which is just an address that cannot be changed

```
int main(int argc, char **argv)
{
    char *pt;
    (void)argc; // shut up the compiler

    while ((pt = *argv++) != NULL) {
        while (*pt != '\0')
            putchar(*pt++);
        putchar('\n');
    }
    return EXIT_SUCCESS;
}
```

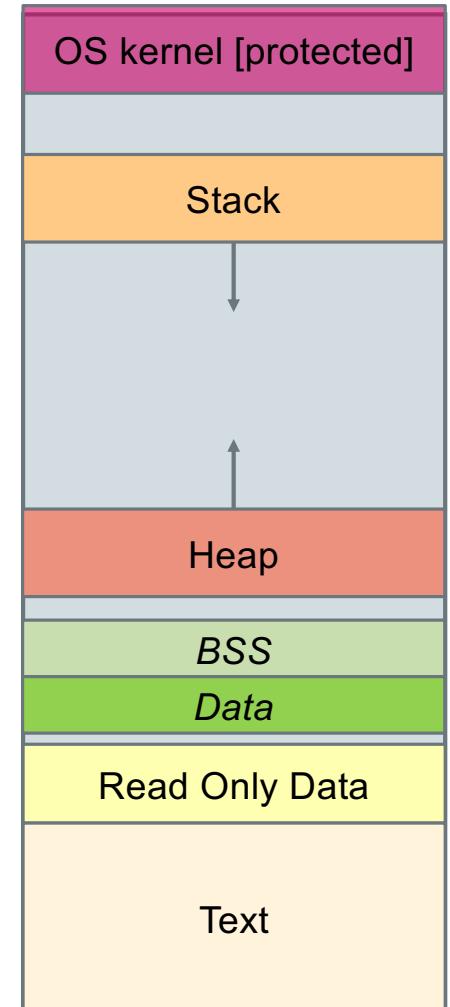
Process Memory Under Linux

- When your **program is running** it has been **loaded into memory** and is **called a process**
- **Stack segment:** Stores **Local variables**
 - Allocated and freed at function call entry & exit
- **Data segment + BSS:** Stores **Global and static variables**
 - Allocated/freed when the process **starts/exits**
 - **BSS** - Static variables with an implicit initial value
 - **Static Data** - Initialized with an explicit initial value
- **Heap segment:** Stores **dynamically-allocated** variables
 - Allocated with a function call
 - Managed by the stdio library malloc() routines
- **Read Only Data:** Stores **immutable** Literals
- **Text:** Stores your code in machine language + libraries



The Heap Memory Segment

- **Heap:** "pool" of memory that is available to a program
 - Managed by C runtime library and linked to your code; **not managed by the OS**
- Heap memory is **dynamically "borrowed"** or **"allocated"** by calling a **library** function
- When heap memory is no longer needed, it is **"returned"** or **deallocated** for **reuse**
- Heap memory has a lifetime from allocation until it is deallocated
 - Lifetime is independent of the scope it is allocated in (it is like a static variable)
- If **too much memory has already been allocated**, the library will attempt to borrow additional memory from the OS and will fail, returning a **NULL**



Heap Dynamic Memory Allocation Library Functions

#include <stdlib.h>	args	Clears memory at runtime
void *malloc(...)	size_t size	no
void *calloc(...)	size_t nmemb, size_t memsize	yes
void free(...)	void *ptr	no

- **void *** means these library functions return a pointer to **generic (untyped) memory**
 - Be careful with **void *** pointers and **pointer math** as void * points **at untyped memory**
 - When assigned to a typed pointer, it "*converts*" it from a **void *** to the **type of the pointer variable**
- **size_t** is an **unsigned integer data type**, the result of a **sizeof()** operator

```
int *ptr = malloc(sizeof(*ptr) * 100); // allocate an array of 100 ints
```
- **please read: % man 3 malloc**

Use of Malloc

```
void *malloc(size_t size)
```

- Returns a pointer to a **contiguous block** of **size** bytes of **uninitialized memory** from the heap
 - The block is **aligned to an 8-byte (arm32) or 16-byte (64-bit arm/intel) boundary**
 - returns **NULL** if allocation failed (also sets `errno`) **always CHECK for NULL RETURN!**
- Blocks returned on **different calls to malloc()** are **not necessarily adjacent**
- **void *** is implicitly cast into **any pointer type on assignment to a pointer variable**

```
char *bufptr;  
  
/* ALWAYS CHECK THE RETURN VALUE FROM MALLOC!!!! */  
  
if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {  
    fprintf(stderr, "Unable to malloc memory");  
    return NULL;  
}  
// allocates a character array with 10 elements
```

bufptr



Calloc()

```
void *calloc(size_t elementCnt, size_t elementSize)
```

- calloc() variant of malloc() but zeros out every byte of memory during program execution before returning a pointer to it (so this has a runtime cost!)
 - First parameter is the number of elements you would like to allocate space for
 - Second parameter is the size of each element

```
// allocate 10-element array of pointers to char, zero filled
char **arr;
arr = calloc(10, sizeof(*arr));
if (arr == NULL) // handle the error
```



- Originally designed to allocate arrays but works for any memory allocation
 - `calloc()` multiplies the two parameters together for the total size
 - `calloc()` is more expensive at runtime (uses both cpu and memory bandwidth) than `malloc()` because it must zero out memory it allocates at runtime
- Use `calloc()` only when you need the buffer to be zero filled prior to FIRST use

Using and Freeing Heap Memory

- `void free(void *p)`
 - Deallocates the **whole block pointed to by p** to the pool of available memory
 - **Freed memory is used in future allocations** (**expect the contents to change after freed**)
 - Pointer **p must be the same address as originally returned** by one of the heap allocation routines `malloc()`, `calloc()`, `realloc()`
 - **Pointer argument to free() is not changed by the call to free()**
- Defensive programming: **set the pointer to NULL after passing it to free()**

```
char *bufptr;

if ((bufptr = malloc(cnt * sizeof(*bufptr))) == NULL) {
    fprintf(stderr, "Unable to malloc memory");
    return NULL;
}
// other code
free(bufptr);                      // returns memory to the heap
bufptr = NULL;                      // set bufptr to NULL
```

Mis-Use of Free() - 1

- Call **free()**
 - With the same address that you obtained with malloc() (or other allocators)
 - It is NOT an error to pass **free()** a pointer to NULL

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes))) != NULL) {
    /* some code */
    free(bytes + 5); // Program aborts free(): invalid pointer
```

- **Freeing unallocated memory:** Only call free() to free memory address that you obtain from one of the allocators (malloc(), calloc(), etc.)

```
char *ptr = "cse30";
...
/* some code */
free(ptr);           // Program aborts free(): invalid pointer
```

Mis-Use of Free() - 2

- Continuing to write to memory after you `free()` it is likely to corrupt the heap or return changed values
 - Later calls to heap routines (`malloc()`, `calloc()`, `strdup()`) may fail or seg fault

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes))) != NULL) {
    /* some code */
    free(bytes);
    strcpy(bytes, "cse30");    // INVALID! used after free
....
```

- Double Free:** Freeing allocated memory more than once will cause your program to abort (terminate)

```
char *bytes;
if ((bytes = malloc(1024 * sizeof(*bytes))) != NULL) {
    /* some code */
    free(bytes);
    free(bytes); // Program abort double free detected...
....
```

More Dangling Pointers: Continuing to use "freed" memory

- Review: Dangling pointer points to a memory location that is no longer "valid"
- Really hard to debug as the use of the return pointers may not generate a seg fault

```
char *dangling_freed_heap(void)
{
    char *buff = malloc(BLKSZ * sizeof(*buff));
    ...
    free(buff);      // memory pointed at buf may be reused
    return buff;     // but it is returned to the caller anyway - bad
}
```

- `dangling_freed_heap()` may cause the allocators (`malloc()` and friends) to seg fault when called later to allocate memory
 - Why? Because it corrupts data structures the heap code uses to manage the memory pool (it often stores meta-data in the freed memory)

strup(): Allocate Space and Copy a String

```
char *strup(char *s);
```

- **strup** is a function that has a **side effect** of returning a **null-terminated**, heap-allocated string copy of the provided text
- Alternative: **malloc** and copy the string with **strncpy()**;
- The caller is responsible for freeing this memory when no longer needed

```
char *str = strup("Hello");
*str = 'h'; // str points at a mutable string

free(str); // caller correctly frees up space allocated by strup()
str = NULL;
```



Heap Memory "Leaks"

- A **memory leak** is when you **allocate memory on the heap, but never free it**

```
void  
leaky_memory (void)  
{  
    char *buf = malloc(BLKSZ * sizeof(*bytes));  
    ...  
    /* code that never calls free() to deallocates the memory */  
    return; // you lose the address in buf when leaving scope  
}
```

- Best practice: free up memory **you allocated** when you no longer need it
 - If you keep allocating memory, you may run out of memory in the heap!
- Memory leaks may cause **long running programs to fault** when they **exhaust OS memory limits**
- **Valgrind** is a tool for finding memory leaks (not pre-installed in all linux distributions though!)

Introduction to Structs – An Aggregate Data Type

- **Structs** are a **collection (or aggregation) of values** grouped **under a single name**
 - Each **variable** in a **struct** is called a **member** (sometimes **field** is used)
 - Each **member** is identified with a **name**
 - Each **member** can be (and quite often are) **different types**, include other structs
 - Like a Java class, but no associated methods or constructors with a struct
- Structure definition **does not** define a variable instance, nor does it allocate memory:
 - It creates a **new variable type** uniquely identified by its **tagname**:
"struct tagname" includes the **keyword struct** and the **tagname** for this type

Easy to forget
semicolon!

```
struct tagname {  
    type1 member1;  
    ...  
    typeN memberN;  
};
```

```
struct vehicle {  
    char *state;  
    char *plate;  
    char *make;  
    int year;  
};
```

Creating a Node & Inserting it at the Front of the List

```
// create node; insert at front when passed head
struct node *
creatNode(int year, char *name, struct node *link)
{
    struct node *ptr = malloc(sizeof(*ptr));
    if (ptr != NULL) {
        if ((ptr->name = strdup(name)) == NULL) {
            free(ptr);
            return NULL;
        }
        ptr->year = year;
        ptr->next = link;
    }
    return ptr;
}
```

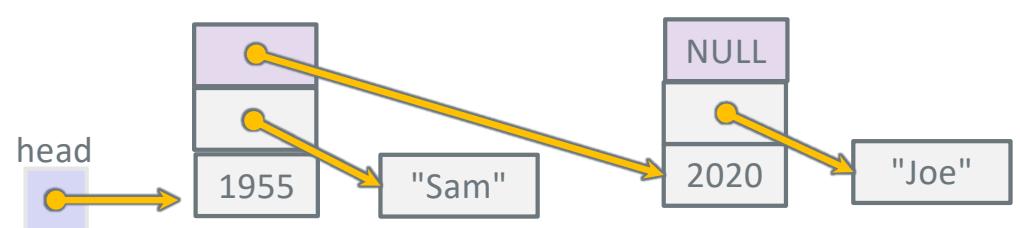
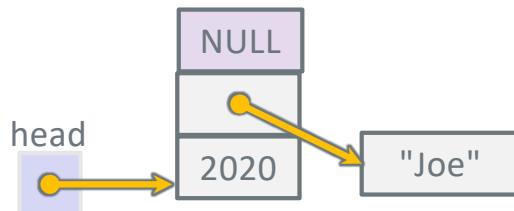
```
struct node {
    int year;
    char *name;
    struct node *next;
};
```

```
// calling function body
struct node *head = NULL; // insert at front
struct node *ptr;

if ((ptr = creatNode(2020, "Joe", head)) != NULL) {
    head = ptr; // error handling not shown
}
if ((ptr = creatNode(1955, "Sam", head)) != NULL) {
    head = ptr; // error handling not shown
}
```

Never use head here!
you can lose your linked list
if creatNode() fails!

head
NULL



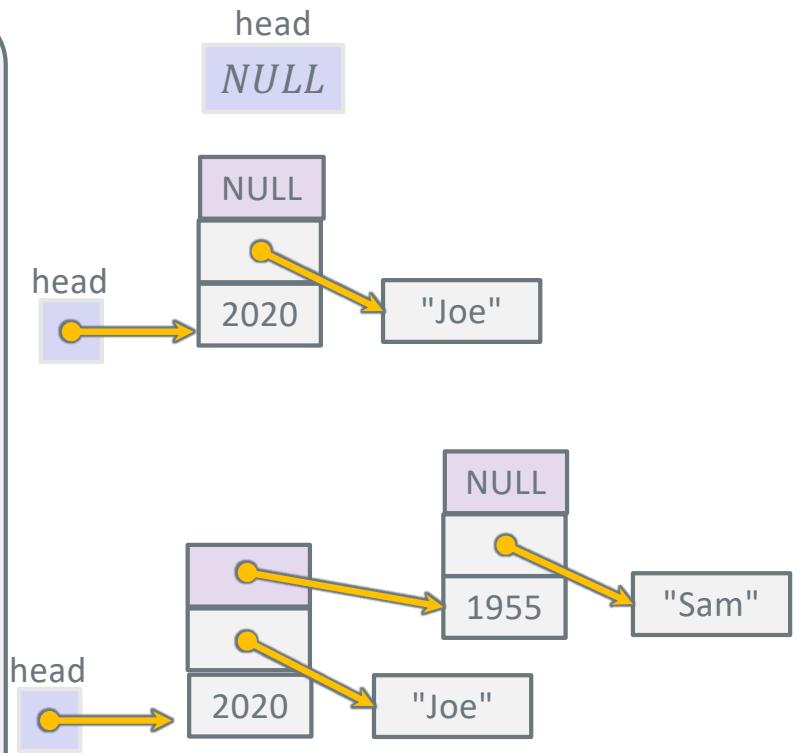
Creating a Node & Inserting it at the End of the List

```
// create a node and insert at the end of the list
struct node *
insertEnd(int year, char *name, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = NULL; // base case
    struct node *new;

    if ((new = creatNode(year, name, NULL)) == NULL)
        return NULL;

    while (ptr != NULL) {
        prev = ptr;
        ptr = ptr->next;
    }
    if (prev == NULL)
        return new;
    prev->next = new;
    return head;
}
```

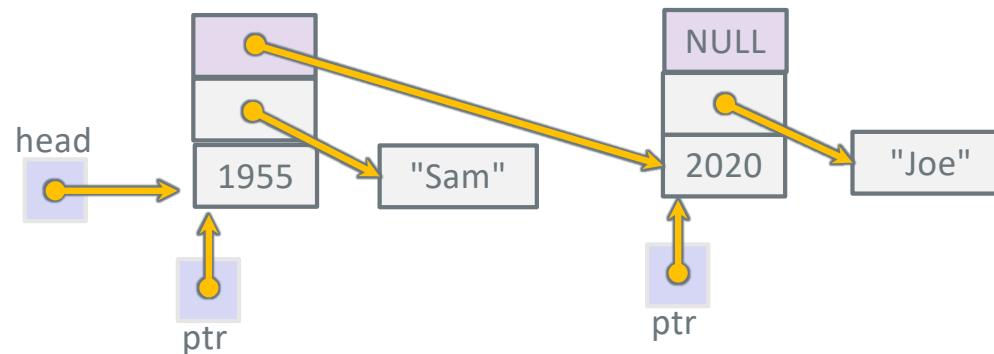
```
struct node *head = NULL; // insert at end
struct node *ptr;
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)
    head = ptr;
```



"Dumping" the Linked List

"walk the list from head to tail"

```
struct node {  
    int year;  
    char *name;  
    struct node *next;  
};
```



∅
ptr

```
struct node *head;  
struct node *ptr;  
...  
printf("\nDumping All Data\n");  
ptr = head;  
while (ptr != NULL) {  
    printf("year: %d name: %s\n", ptr->year, ptr->name);  
    ptr = ptr->next;  
}
```

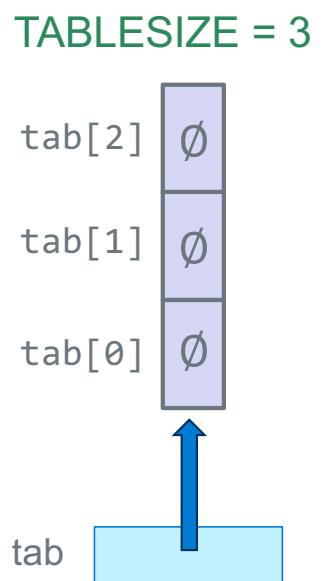
Dumping All Data
year: 1955 name: Sam
year: 2020 name: Joe

Allocating an empty Hash Table

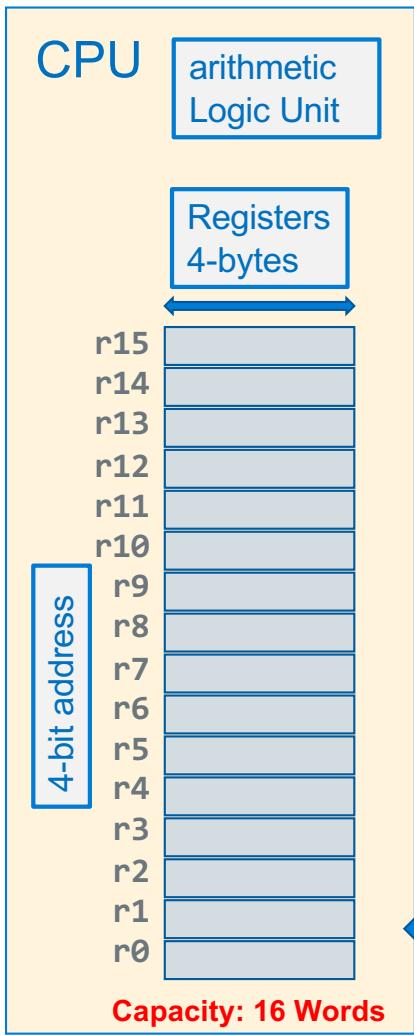
Good use for calloc() to initialize all entries with NULL

```
#define TBSZ 3
int main(void)
{
    struct node *ptr;
    struct node **tab; // pointer to hashtable
    uint32_t index;

    if ((tab = calloc(TBSZ, sizeof(*tab))) == NULL) {
        fprintf(stderr,"Cannot allocate hash table\n");
        return EXIT_FAILURE;
    }
// continued on next slide
```



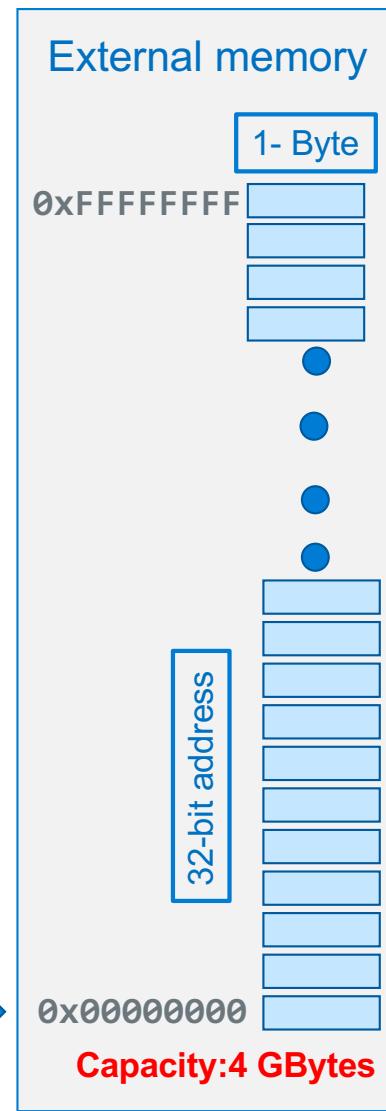
32-Bit Arm - Registers



- Registers are **memory** located within the **CPU**
- Registers are the **fastest** read and write storage
- Register is **word size in length** stores 32-bit values
 - Memory is accessed using **pointers** in registers
- In assembly language, register "addresses" are specified using **predefined names** starting with an **r** to differentiate them from main **memory addresses** which are **labels** (address)
- 16 registers: from **r0** to **r15** (encoded: **0x0 – 0xf**)

CPU Memory Bus consists of two parts:

Address bus + Data bus



Using Arm-32 Registers

- There are two basic groups of registers, **general purpose** and **special use**
- **General purpose registers** can be used to contain up to 32-bits of data, but you must follow the **rules** for their use
 - Rules specify how registers are to be used so software can communicate and share the use of registers (later slides)
- **Special purpose registers:** have a dedicated hardware use (r15 the pc) or **special use** when used with certain instructions (r13 & r14)
- r15/pc is the program counter that contains the address of an instruction being executed (not exactly ... later)

Special Use Registers
program counter

r15/pc

Special Use Registers
function call implementation
& long branching

r14/lr
r13/sp
r12/ip
r11/fp

Preserved registers
Called functions **can't change**

r10
r9
r8
r7
r6
r5
r4

Scratch Registers
First 4 Function Parameters
Function return value
Called functions **can change**

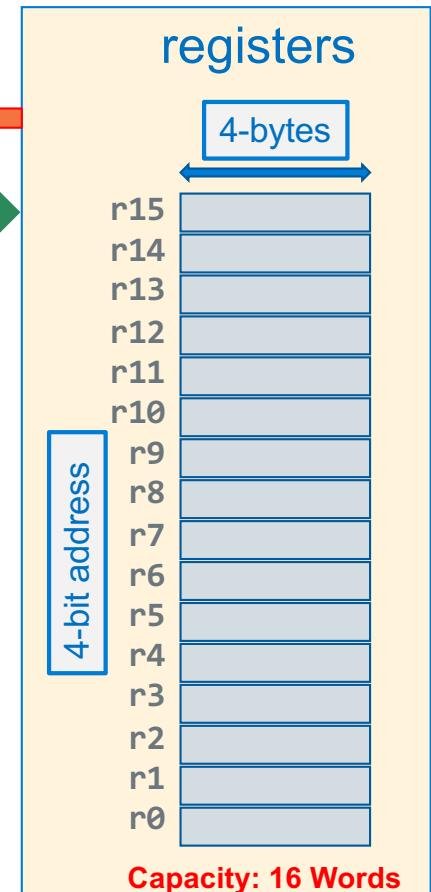
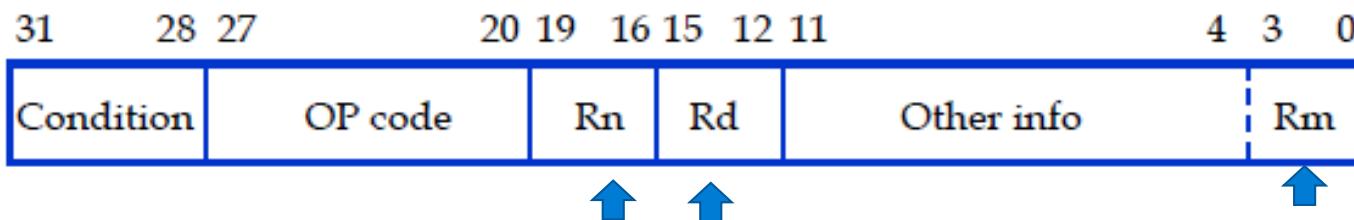
r3
r2
r1
r0

32-Bit Arm - Registers

All computations (add, subtract, etc.) are performed in the ALU

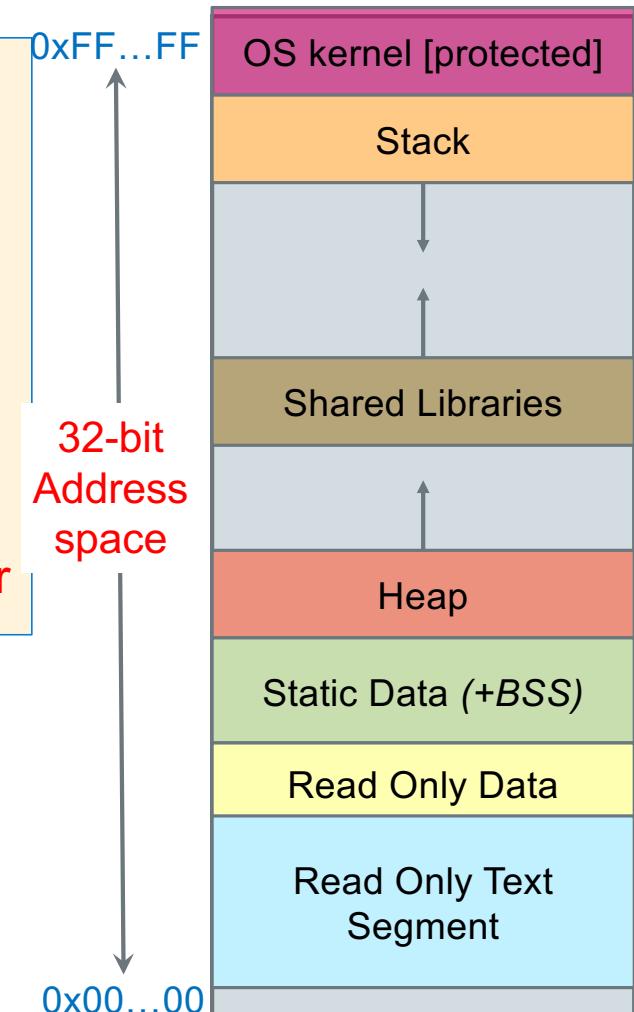
Arithmetic & Logic Unit (ALU)

- Almost all arithmetic, logic operations and data movement operations involve at least one register
- As a result, Register addresses are **directly encoded** into 4-bit fields in machine instructions (see below)

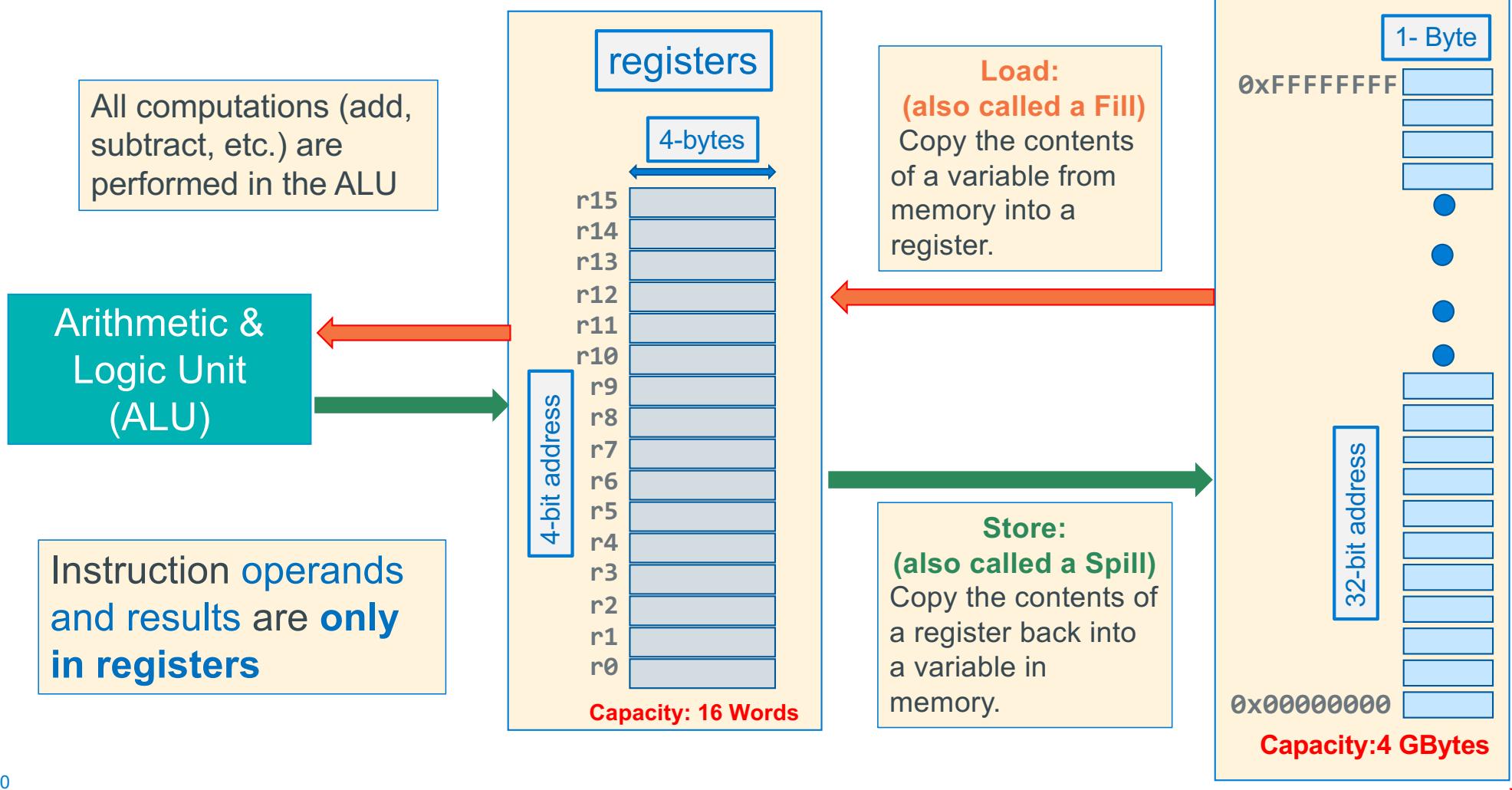


How to Access Memory?

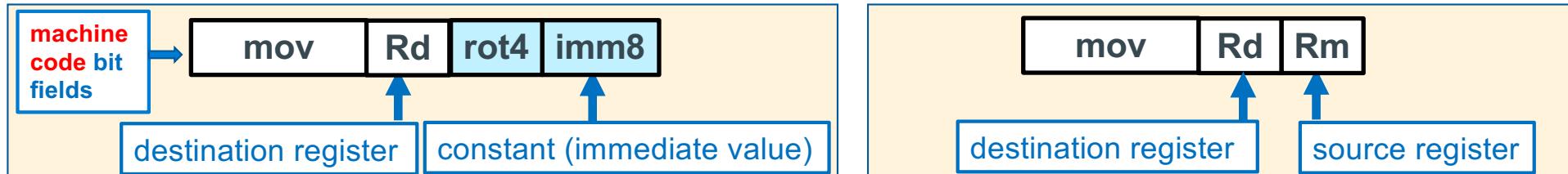
- Consider $a = b + c$ where operands are in memory
 - Operation code: add
 - Operand 1: b
 - Destination: a
 - Operand 2: c
- Aarch32 Instructions are always word size: 32 bits wide
 - Some bits must be used to specify the operation code
 - Some bits must be used to specify the destination
 - Some bits must be used to specify the operands
- Address space is 32 bits wide so put a **POINTER** in a register



32-Bit Arm is a Load/Store Architecture



mov – Copies Register Content between registers



Immediate "addressing"

register direct "addressing"

R type:	4	3	4	1	4	4	8	4
	1110	000	Opcode	S	0000	Rd	00000000	Rm
I type:	1110	001	Opcode	S	0000	Rd	Rotate	Imm8
	↑				← { 1101 - MOV 1111 - MVN }			

imm8 is 8 bits in size!

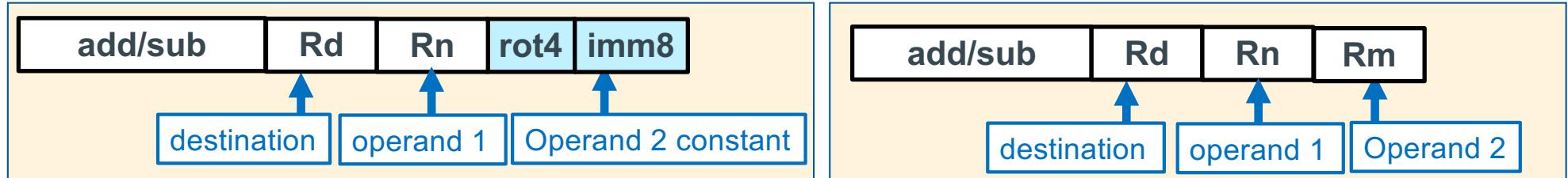
IMPACT: limited range of values

```
mov Rd, constant // Rd = constant
mov Rd, Rm // Rd = Rm
```

(-256) <= imm8 <= 255 + values from "rotating" bits - later

```
mov r1, r5 // r1 = r5
mov r1, 1 // r1 = 1
mov r1, -4 // r1 = -4
```

add/sub – Add or Subtract two integers

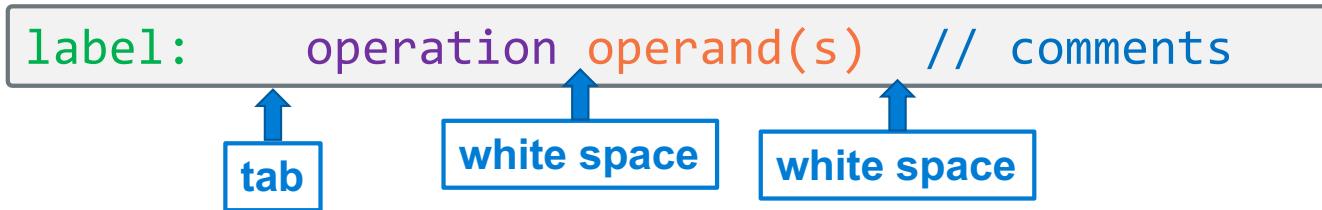


```
add Rd, Rn, constant      // Rd = Rn + constant  
sub Rd, Rn, constant      // Rd = Rn - constant  
add Rd, Rn, Rm            // Rd = Rn + Rm  
sub Rd, Rn, Rm            // Rd = Rn - Rm
```

```
add r1, r2, r3            // r1 = r2 + r3  
sub r1, r1, 1              // r1 = r1 - 1; or r1--  
add r1, r2, 234            // r1 = r2 + 234
```

Line Layout in an Arm Assembly Source

column 1 column 2 column 3 column 4



- Assembly language source text files are **line oriented** (each ending in a '\n')
- **Each line represents** a **starting address in memory** and does **one of**:
 1. Specifies the **contents of memory** for a **variable** (segments containing data)
 2. Specifies the **contents of memory** for an **instruction** (text segment)
 3. **Assembler directives** tell the assembler to do something (for example, change label scope, define a macro, etc.) that **does not allocate memory**
- **Each line is organized into up to four columns**
 - Not every column is used on each line
 - Not every line will result in **memory being allocated**

Labels in Arm Assembly - 2

```
.Lmesg: .string "Hello CSE30! We Are Counting UpPpER cASE letters!"  
label .Lmesg is the starting address for the ascii string  
  
main:  
    push {fp, lr}  
  
.Lwhile:  
    add r2, r2, 1      // increment char pointer  
label .Lwhile is the starting address of the add instruction
```

- Remember, a **Label** associates a **name** with **memory location**
- **Regular Label:**
 - Used with a **Function name** (label) or all **static variables** in any of the data segments
- **Local Label:** Name starts with **.L** (local label prefix) only usable in the same file
 1. Targets for
 - branches: if switch, goto, break, continue,
 - loops: for, while, do-while
 2. **Anonymous variables** (the address of **literal** not the address of **foo** in the following)
char *foo = "literal";

Unconditional Branching – Forces Execution to Continue at a Specified Label (goto)



imm24 is Relative direction
from the branch instruction (in +/- instructions)

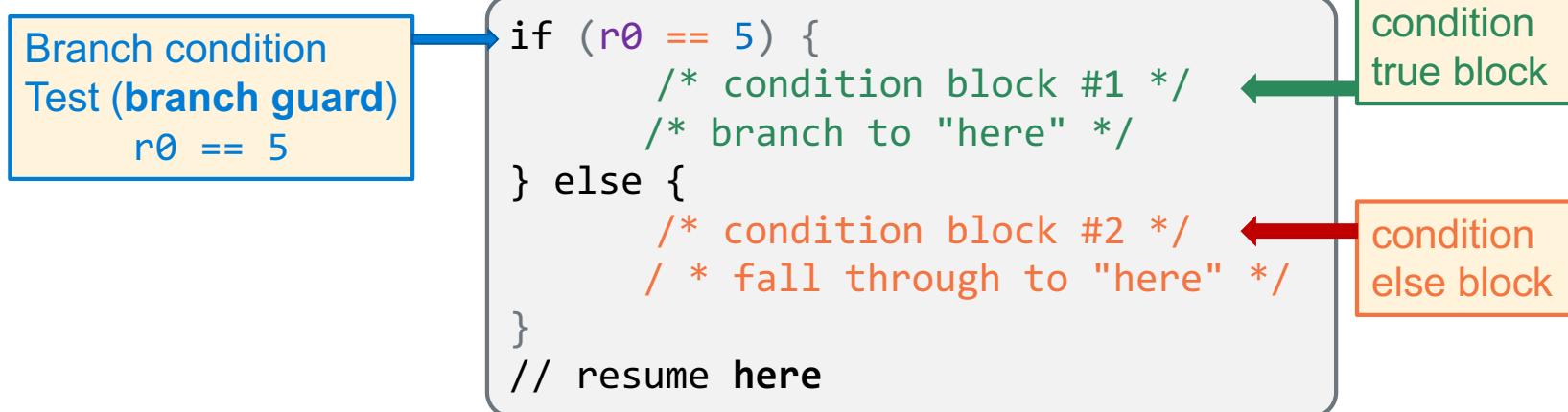
Unconditional Branch instruction (*branch to only local labels in CSE30*)

b .Llabel

- Causes an unconditional branch (aka goto) to the instruction with the address **.Llabel**
- **.Llabel** is called a **branch target label** (the "*target*" of a branch instruction)
- **Be careful! do not to branch to a function label!**
- **.Llabel**: translates into an number offset being **imm24 shifted left two bits** (+/- 32 MB)

```
b      .Ldone                                // goto .Ldone
      :
.Ldone:
      add    r0, EXIT_SUCCESS                  // set return value
```

Anatomy of a Conditional Branch: If statement



- **Branch guard:** determines when to execute the "condition true block" or the "condition false block"
- In C, when the branch guard (condition test) evaluates non-zero you **fall through** to the **condition true** block, otherwise you branch to the **condition false (else)** block
- Step 1: evaluate the branch guard(s) (involves one or more compares/tests)
- Step 2: If branch guard evaluates to be **false**
 - **branch around** the **true block** and execute the **else block**
 - otherwise **"fall through"** and execute the **true block**
- **Block order** in C is where the **True Block** appears above the **False block**

cmp/cmm – Making Conditional Tests

cmp/cmm Rn rot4 imm8

operand 1 Operand 2 constant

Bytes: $0 \leq \text{imm8} \leq 255 + \text{values from "rotating" rot 4 bits}$

cmp/cmm Rn Rm

operand 1 Operand 2

```
cmp Rn, constant      // Rn - constant; then sets condition flags  
cmm Rn, constant      // Rn + constant; then sets condition flags  
cmp Rn, Rm            // Rn - Rm; then sets condition flags  
cmm Rn, Rm            // Rn + Rm; then sets condition flags
```

The values stored in the registers Rn and Rm are not changed

The assembler will automatically substitute `cnn` for negative immediate values

```
cmp r1, 0              // r1 - 0 and sets flags on the result  
cmp r1, r2              // r1 - r2 and sets flags on the result
```

Conditional Tests: Implementing ARM Branch guards

imm24 is Relative direction
from the branch instruction
(in +/- instructions)

cond **bsuffix** **imm24**

Branch instruction

bsuffix .Llabel

Use a local label with branch
instructions

Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
BNE	Not equal	Z = 0
BGE	Signed \geq ("Greater than or Equal")	N = V
BLT	Signed $<$ ("Less Than")	N \neq V
BGT	Signed $>$ ("Greater Than")	Z = 0 && N = V
BLE	Signed \leq ("Less than or Equal")	Z = 1 N \neq V
BMI	Minus/negative	N = 1
BPL	Plus - positive or zero (non-negative)	N = 0
B	Branch Always (unconditional)	

- Bits in the condition field specify the **conditions** when the branch happens
- If the **condition evaluates** to be **true**, **next instruction executed** is at **.Llabel**:
- If the **condition evaluates** to be **false**, **next instruction executed** is **immediately after the branch**
- Unconditional branch** is when the condition is "**always**"

Program Flow: Simple If statement, No Else

<i>C source Code</i>	<i>Incorrect Assembly</i>	<i>Correct Assembly</i>
int r0; if (r0 > 10) { // True Block }	cmp r0, 10 bgt .Lendif // True Block .Lendif:	cmp r0, 10 ble .Lendif // True Block .Lendif:

- Realize that in ARM assembly you can only either "fall through" to the next instruction or branch to a specific instruction
- Approach: **adjust** the conditional test then **branch around** the **true block**
- Use a **conditional test** that specifies the **inverse** of the condition used in C
 - This preserves **C block order**

Branch Guard "*Adjustment*" Table Preserving C Block Order In Assembly

Compare in C	"Inverse" Compare in C	Assembly using Inverse Compare
<code>==</code>	<code>!=</code>	<code>bne</code>
<code>!=</code>	<code>==</code>	<code>beq</code>
<code>></code>	<code><=</code>	<code>ble</code>
<code>>=</code>	<code><</code>	<code>blt</code>
<code><</code>	<code>>=</code>	<code>bge</code>
<code><=</code>	<code>></code>	<code>bgt</code>

```
if (r0 compare 5)
    /* condition true block */
    /* then fall through */
}
```

```
cmp r0, 5
inverse compare .Lelse
// condition true block
// then fall through
.Lendif:
```

Arm Conditional Branching Simple IF no else

C If statement

```
int r0;  
if (r0 == 5) {  
    /* condition true block */  
    /* then fall through */  
}  
/* branch around to this code */
```

If r0 == 5 true
then **fall through** to
the true block

ARM If statement

```
cmp r0, 5  
bne .Lendif  
/* condition true block */  
/* then fall through */  
.Lendif:  
/* branch around to this code */
```

If r0 == 5 false
then branch **around**
the true block

- If statements in ARM
- **Step 1:** make a conditional test using a **cmp** instruction
- **Step 2:** if test evaluates to FALSE, **branch around** the **condition true** block with a one of the conditional branch instruction

Program Flow: If with an Else

Approach:

1. adjust the conditional test to branch to the **False Block**
2. Fall through to the **True Block**
3. Bottom of the **True Block unconditionally branches** around the **False block**

<i>C source Code</i>	<i>Assembly</i>
<pre>int r0; if (r0 > 10) { // true block // branch always around the false block } else { // false block }</pre>	<pre>cmp r0, 10 ble .Lelse /fall through // true block b .Lendif .Lelse: // false block // fall through .Lendif:</pre>

Review – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal evaluation**

```
if ((x == 5) || (y > 3)) // if x == 5 then y > 3 is not evaluated
```

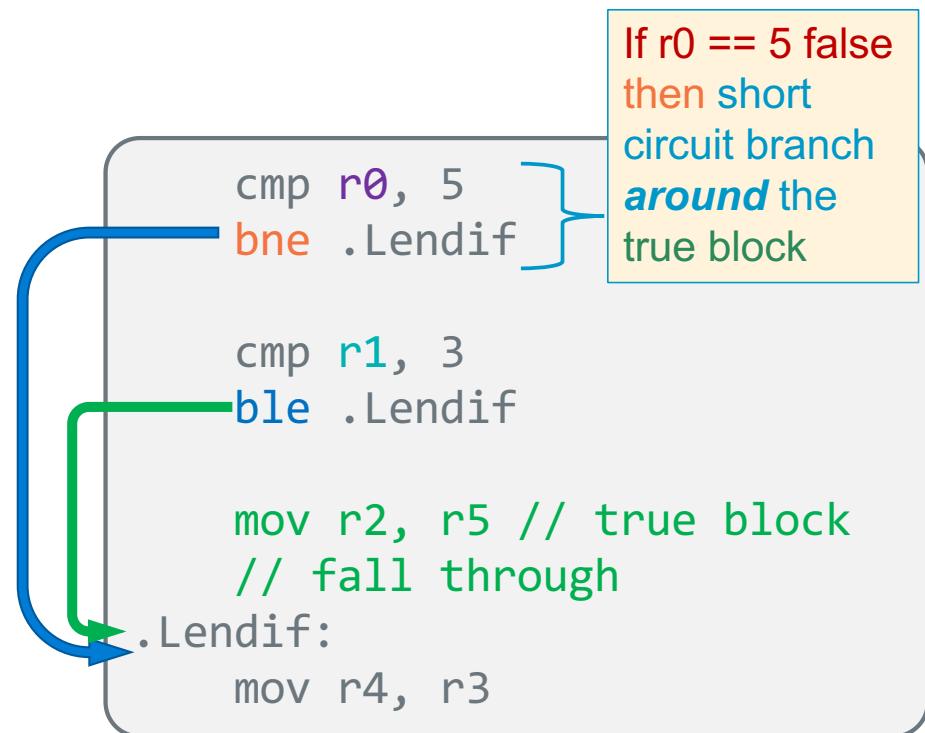


- Each expression argument is evaluated in sequence from left to right including any side effects (modified using parenthesis), before (optionally) evaluating the next expression argument
- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated (for performance)

```
if ((a != 0) && func(b)) // if a is 0, func(b) is not called  
    // do something
```

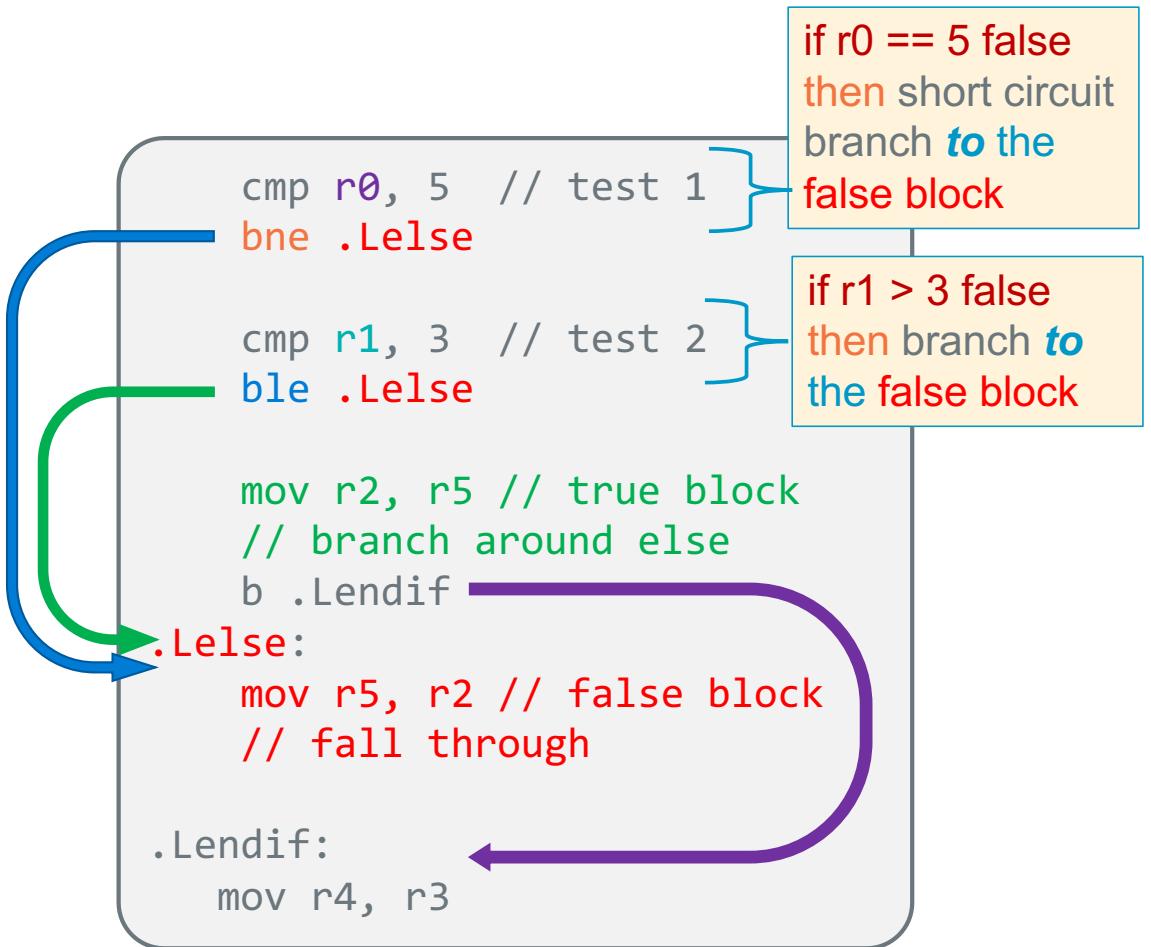
Program Flow – If statements & compound tests - 1

```
if ((r0 == 5) && (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```



Program Flow – If statements & compound tests - 2

```
test1           test2
if ((r0 == 5) && (r1 > 3))
{
    r2 = r5; // true block
    // branch around else
} else {
    r5 = r2; // False block */
    /* fall through */
}
r4 = r3;
```



Program Flow – If statements & compound tests - 3

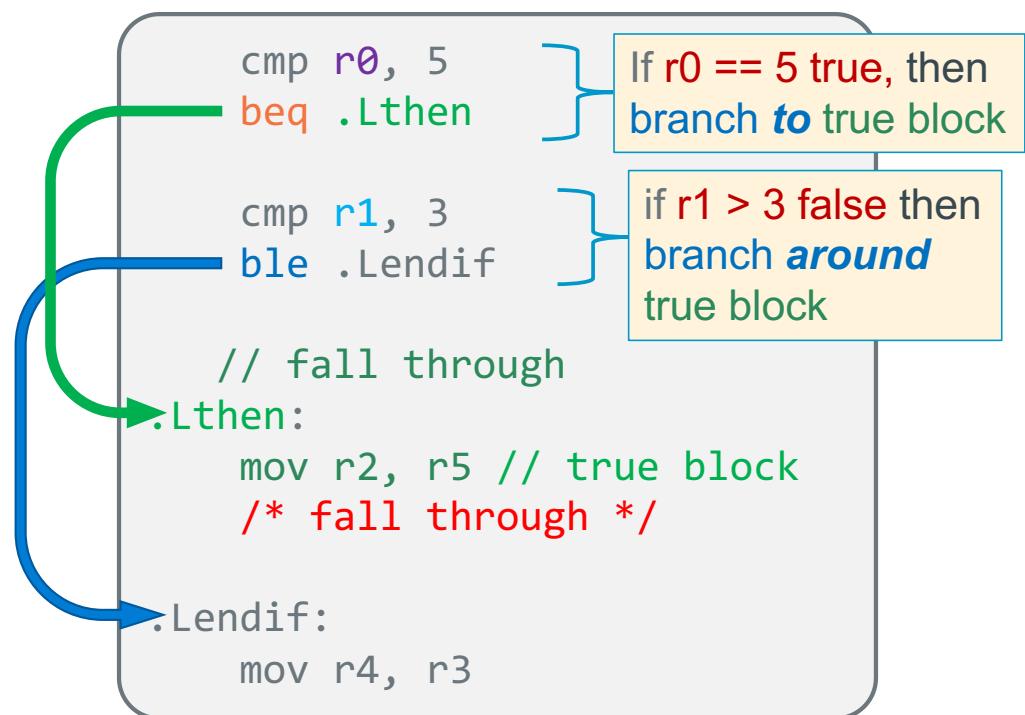
```
if (test1 && test2 & ... & testn) {  
    // true block  
    // branch around else  
} else {  
    // False block  
    /* fall through */  
}
```

All compound && tests use an **inverse test** and if they evaluate to **true** branch **to the false block**

```
cmp xx,xx // test 1  
inverse test branch .Lelse  
  
cmp xx, xx // test 2  
inverse test branch .Lelse  
  
// more tests  
  
cmp xx, xx // test n  
inverse test branch .Lelse  
  
mov r2, r5 // true block  
// branch around else  
b .Lendif  
.Lelse:  
    mov r5, r2 // false block  
    // fall through  
  
.Lendif:  
    mov r4, r3
```

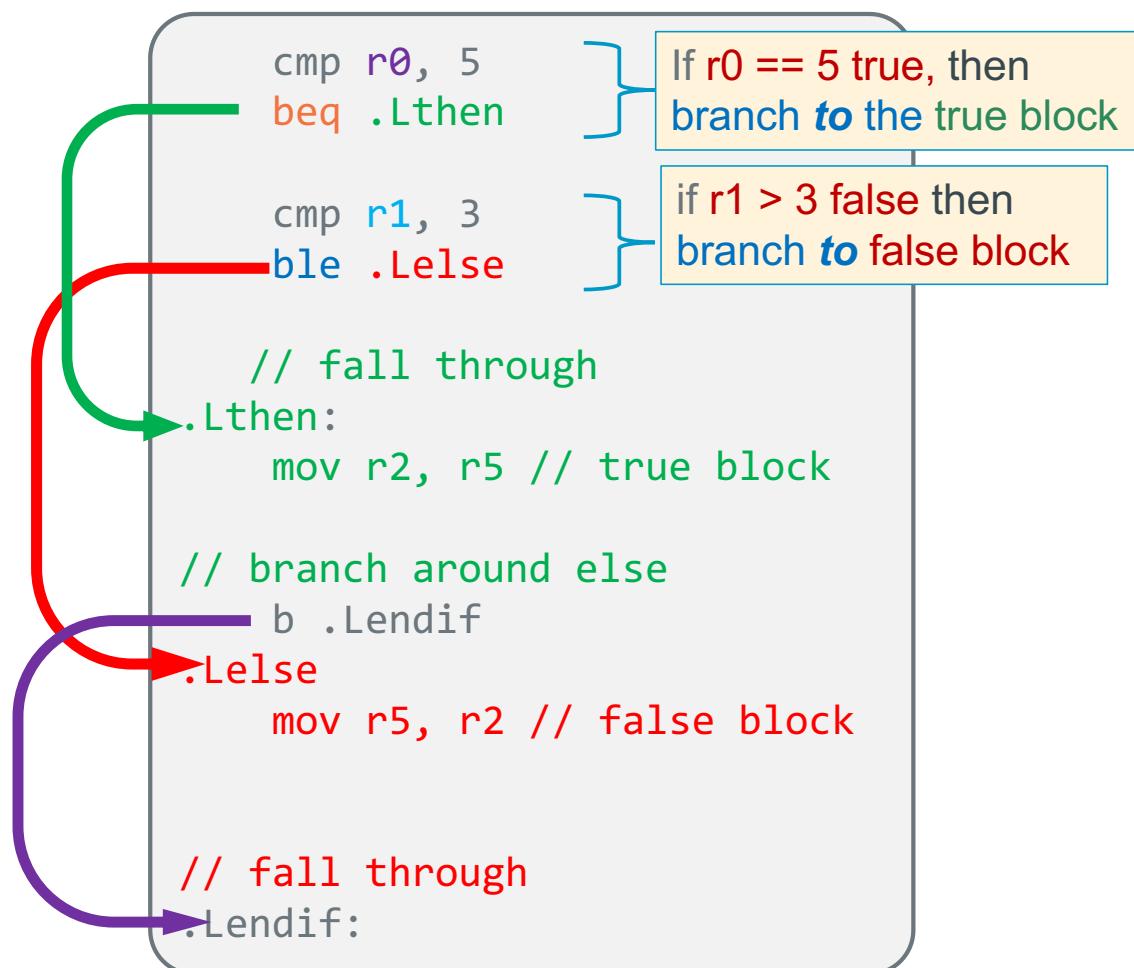
Program Flow – If statements || compound tests - 1

```
if ((r0 == 5) || (r1 > 3)) {  
    r2 = r5; // true block  
    /* fall through */  
}  
r4 = r3;
```



Program Flow – If statements || compound tests - 2

```
if ((r0 == 5) || (r1 > 3)) {  
    r2 = r5; // true block  
    /* branch around else */  
} else {  
    r5 = r2; // false block  
    /* fall through */  
}
```



Program Flow – If statements || compound tests - 3

```
if (test1 || test2 || .. || testn) {  
    // true block  
    /* branch around else */  
} else {  
    // false block  
    /* fall through */  
}
```

All compound || tests (except the last one) use the **same test** as C and if they evaluate to **true branches to the true block**

The last (leftmost) compound || test uses an **inverse test** and if it evaluates to **true branches to false block**

```
cmp XX, XX      // test 1  
same test branch .Lthen
```

```
cmp XX, XX      // test 2  
same test branch .Lthen
```

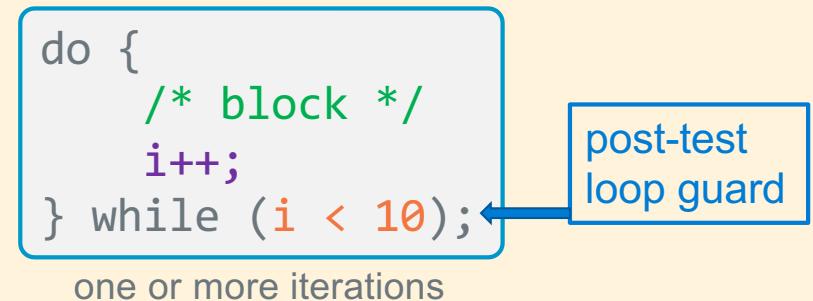
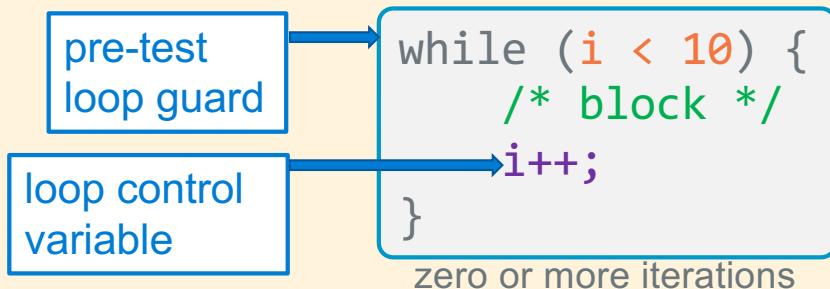
// other tests

```
cmp XX, XX      // test n  
inverse test branch .Lelse
```

```
// fall through  
.Lthen:  
// true block  
// branch around else  
b .Lendif  
.Lelse  
// false block  
// fall through  
.Lendif:
```

Program Flow – Pre-test and Post-test Loop Guards

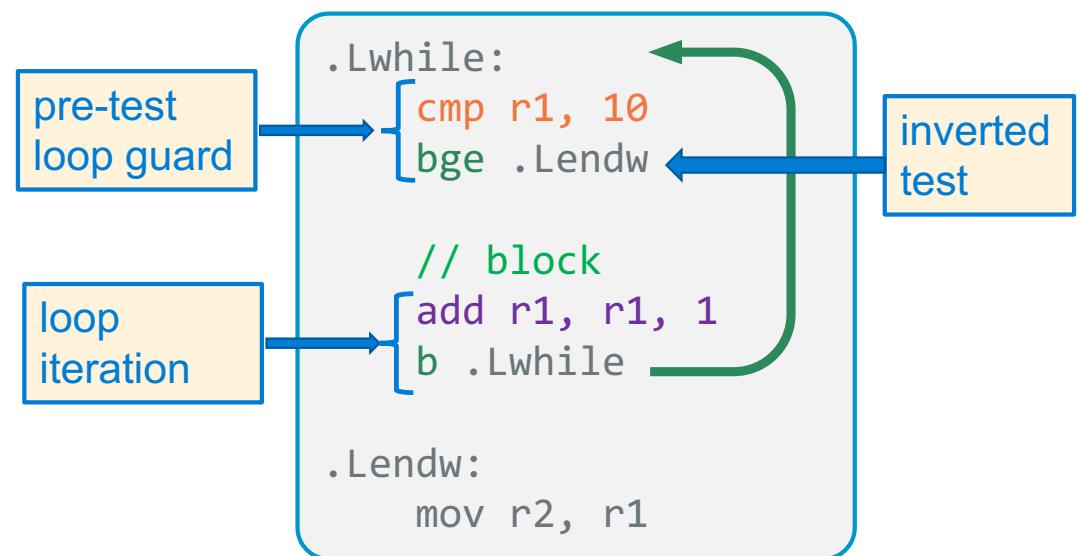
- loop guard: code that must evaluate to true before the next iteration of the loop
- If the **loop guard test(s)** evaluate to true, the *body of the loop* is executed again
- **pre-test loop guard** is at the **top of the loop**
 - If the **test evaluates to true**, execution **falls through** to the loop body
 - if the **test evaluates to false**, execution **branches** around the loop body



- **post-test loop guard** is at the **bottom of the loop**
 - If the **test evaluates to true**, execution **branches** to the top of the loop
 - If the **test evaluates to false**, execution **falls through** the instruction following the loop

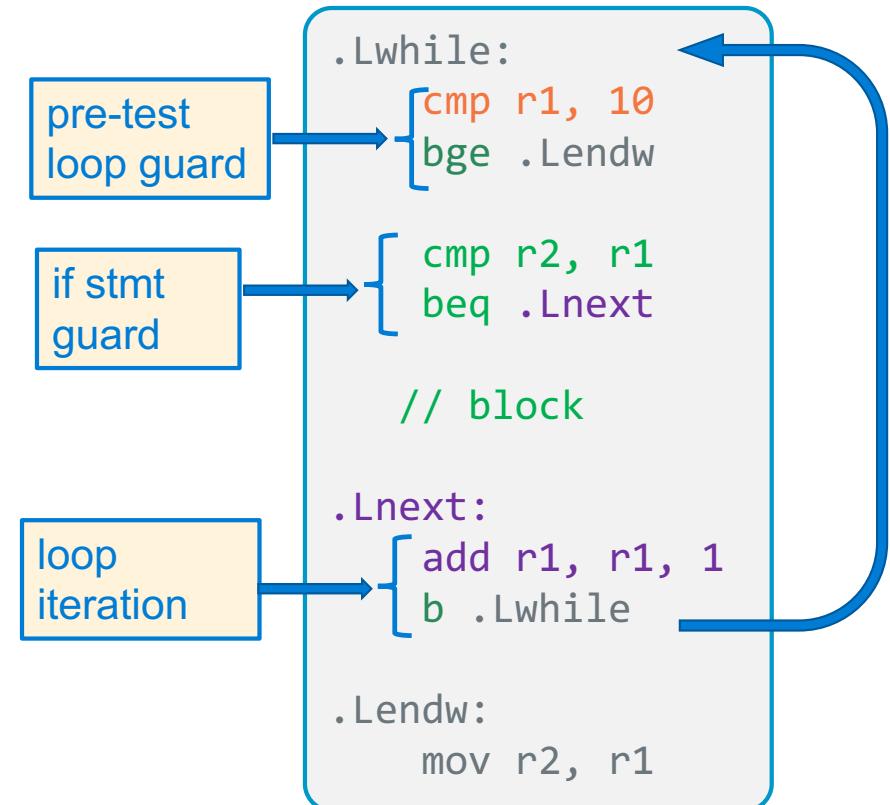
Pre-Test Guards - While Loop

```
while (r1 < 10) {  
    /* block */  
    r1++;  
}  
r2 = r1;
```



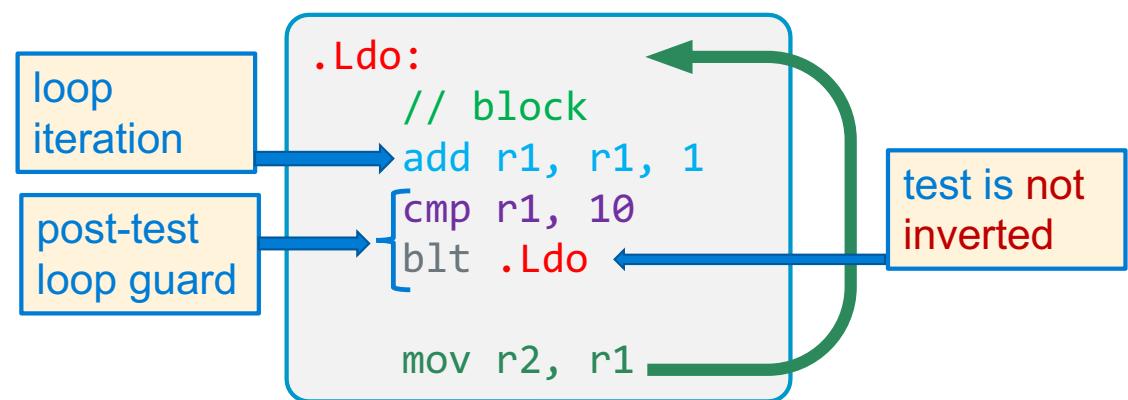
Pre-Test Guards - While Loop

```
while (r1 < 10) {  
    if (r2 != r1) {  
        /* block */  
    }  
    r1++;  
}  
r2 = r1;
```



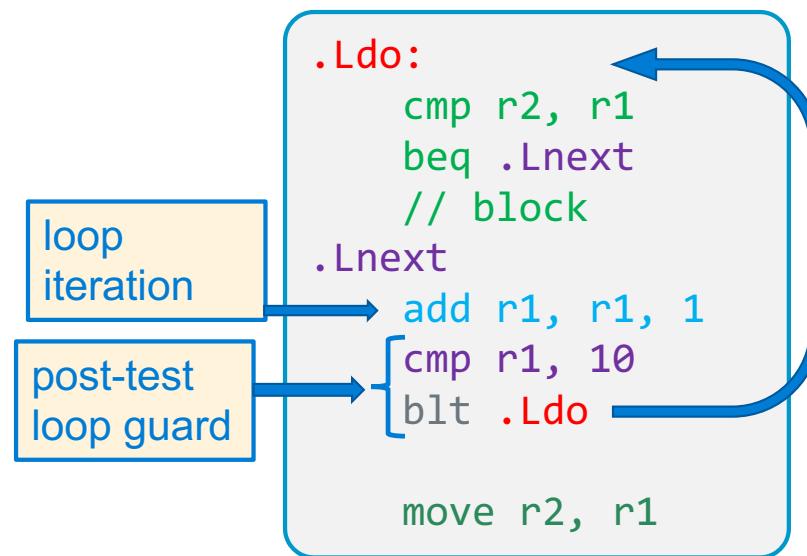
Post-Test Guards – Do While Loop

```
do {  
    /* block */  
    r1++;  
} while (r1 < 10);  
  
r2 = r1;
```

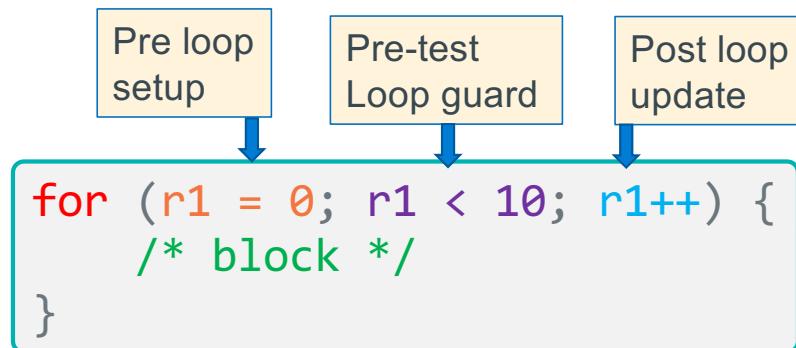


Post-Test Guards – Do While Loop

```
do {  
    if (r2 != r1) {  
        /* block */  
    }  
    r1++;  
} while (r1 < 10);  
  
r2 = r1;
```

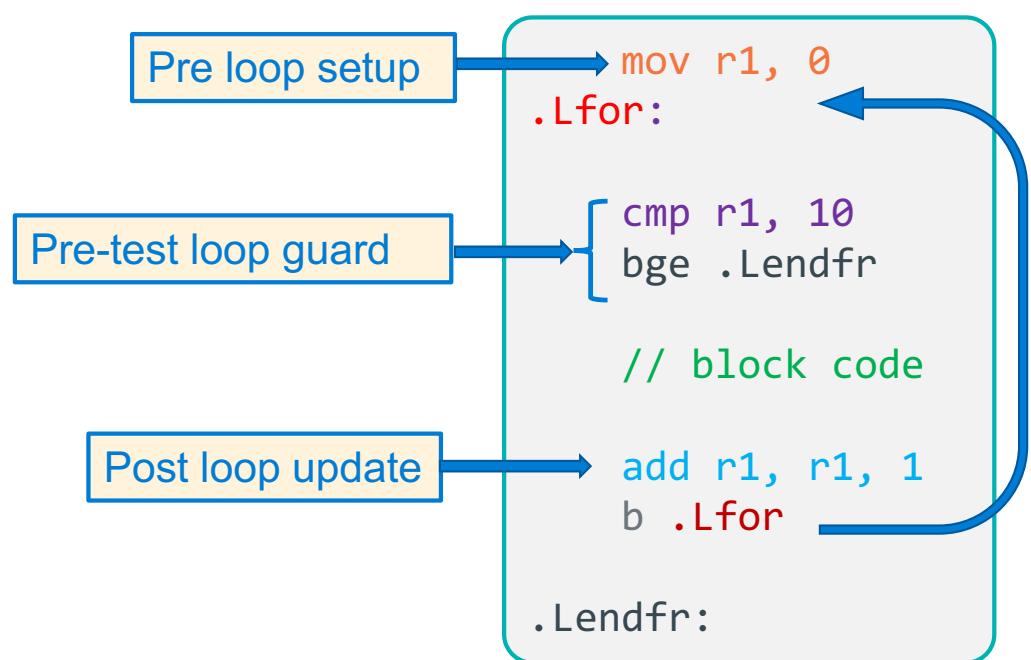


Program Flow – Counting (For) Loop Version 1

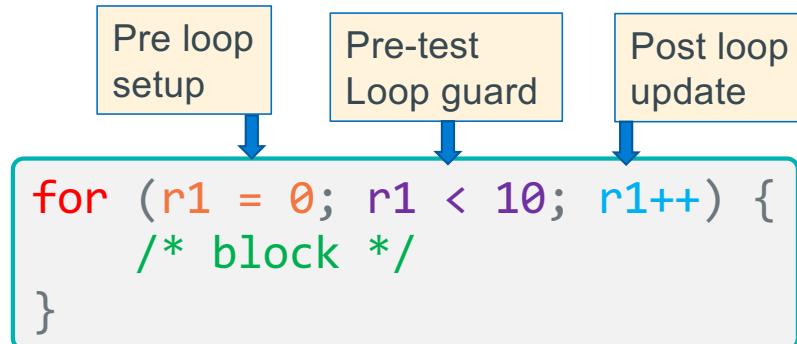


A **counting loop** has **three parts**:

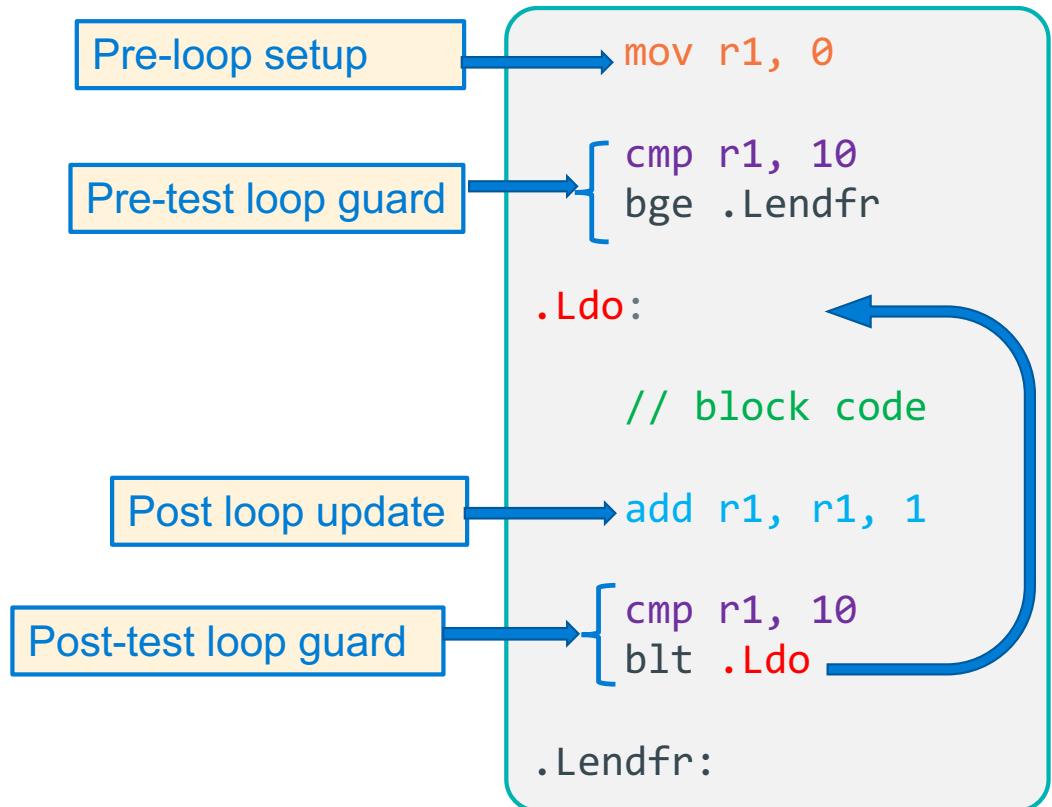
1. Pre-loop setup
2. Pre-test loop guard conditions
3. Post-loop update



Program Flow – Counting (For) Loop – Version 2



- Alternative:
- move Pre-test loop guard before the loop
- Add post-test loop guard
 - converts to *do while*
 - removes an unconditional branch



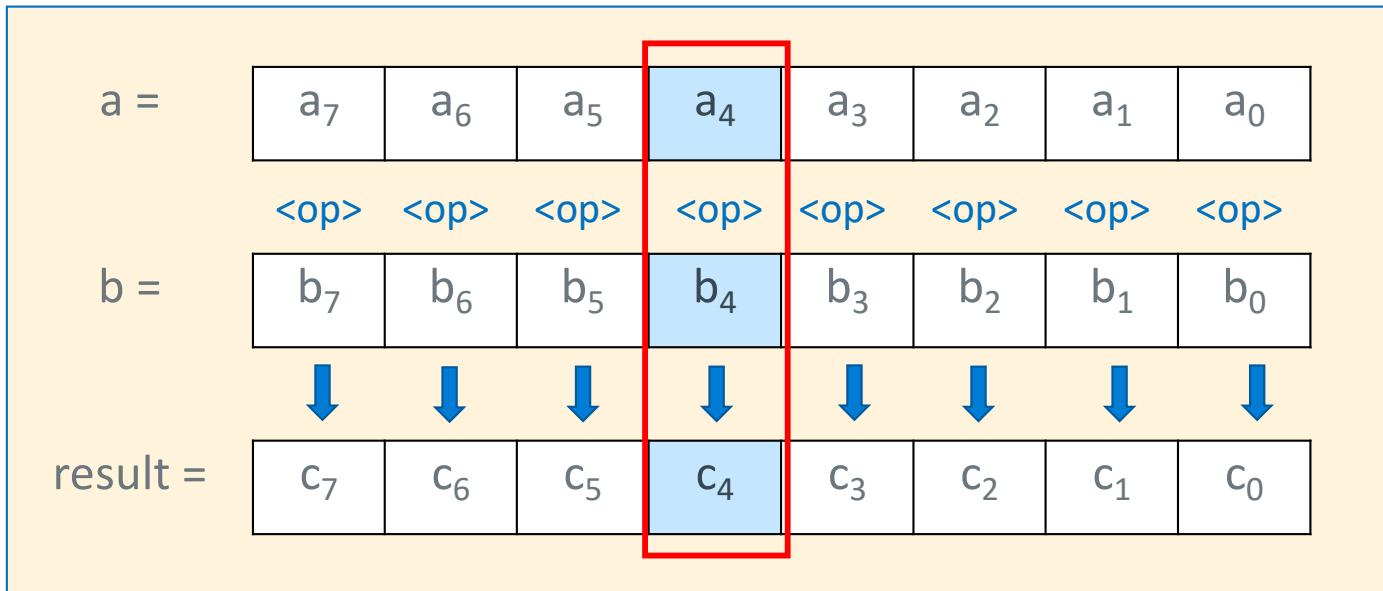
Nested loops

```
for (r3 = 0; r3 < 10; r3++) {  
    r0 = 0;  
  
    do {  
        r0 = r0 + r1++;  
    } while (r1 < 10);  
  
    // fall through  
    r2 = r2 + r1;  
}  
r5 = r0;
```

- Nest loop blocks as you would in C or Java

```
.Lfor:  
    mov r3, 0  
    cmp r3, 10      // loop guard  
    bge .Lendfor  
  
    mov r0, 0  
  
.Ldo:  
    add r0, r0, r1  
    add r1, r1, 1  
  
    cmp r1, 10      // loop guard  
    blt .Ldo  
  
    // fall through  
    add r2, r2, r1  
  
    add r3, r3, 1 // loop iteration  
    b .Lfor  
.Lendfor:  
    mov r5, r0
```

What is a Bitwise Operation?



- Bitwise operators are applied independently to each of the corresponding bit positions in each variable
- Each bit position of the result depends only on bits in the **same bit position** within the operands

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

<code>~</code>	<code>1100</code>
-----	-----
	<code>0011</code>

Bitwise
AND

<code>0110</code>
<code>& 1100</code>

<code>0100</code>

Bitwise
OR

<code>0110</code>
<code> 1100</code>

<code>1110</code>

Bitwise
EOR

<code>0110</code>
<code>^ 1100</code>

<code>1010</code>

Bitwise Not (vs Boolean Not)

```
in C  
int output = ~a;
```

a	$\sim a$
0	1
1	0

Bitwise NOT
 \sim 1100

0011

	Bitwise Not							
number	0101	1010	0101	1010	1111	0000	1001	0110
\sim number	1010	0101	1010	0101	0000	1111	0110	1001

Meaning	Operator	Operator	Meaning
Boolean NOT	$!b$	$\sim b$	Bitwise NOT

Boolean operators act on the entire value not the individual bits

Type	Operation	result
bitwise	$\sim 0x01$	1111 1111 1111 1111 1111 1111 1111 1111 1110
Boolean	$!0x01$	0000 0000 0000 0000 0000 0000 0000 0000 0000

MVN (not)

`mvn r0, r1`

// Copies all 32 bits
// of the value held
// in register r1 into
// the register r0
// then does a bitwise NOT

register r1



register r0

`mvn r0, 12`

// Expands an imm8 value 0x0c
// stored in the instruction
// into a register then does
// a bitwise NOT

register r0

0x0c



0xffff fff3

Bitwise NOT

~ 1100

0011

- A **bitwise NOT** operation

0x 0c

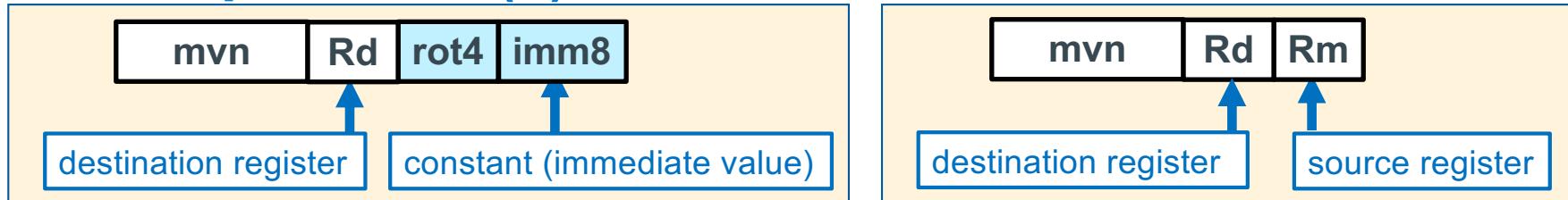
↓ imm8 expansion

0x0000000c

↓ bitwise not

0xffffffff3

mvn – Copies NOT (\sim)



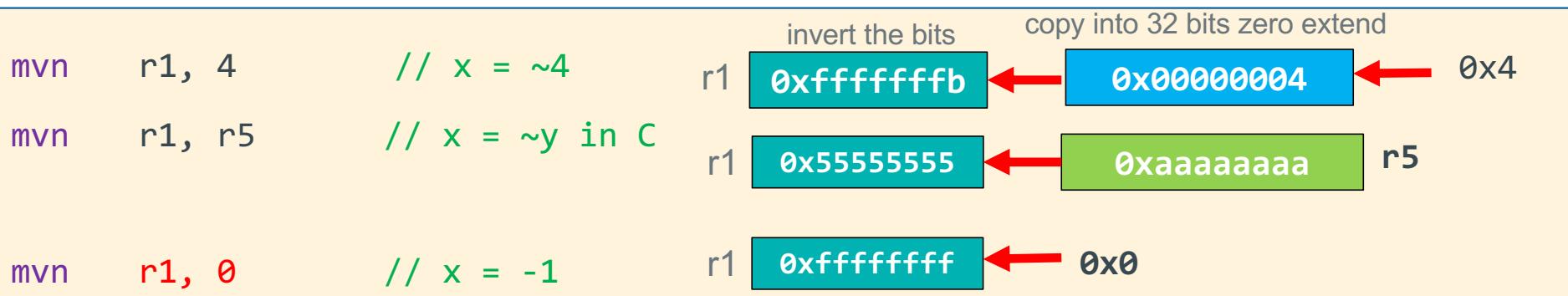
```
mvn Rd, constant // Rd = constant
mvn Rd, Rm // Rd = Rm
```

\sim 1100
 - - - -
 0011

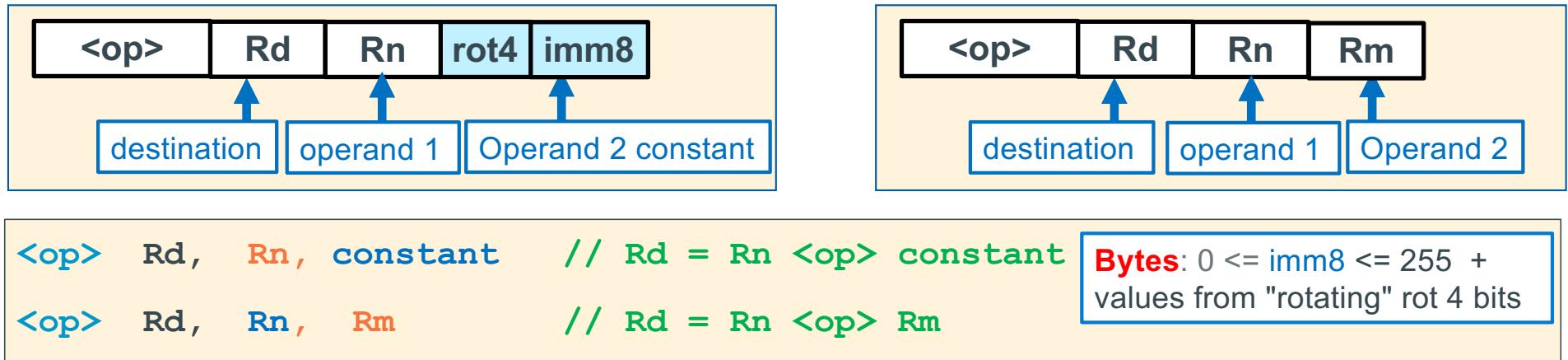
Bitwise NOT

bitwise NOT operation. Immediate (constant) version copies to 32-bit register, then does a bitwise NOT

imm8	extended imm8	inverted imm8	signed base 10
0x00	0x00 00 00 00	0xff ff ff ff	-1
0xff	0x00 00 00 ff	0xff ff ff 00	-256



Bitwise Instructions



Bitwise <op> description	C Syntax	Arm <op> Syntax Op2: either register or constant value	Operation
Bitwise AND	a & b	and Rd, Rn, Op2	$R_d = R_n \& Op2$
Bitwise OR	a b	orr Rd, Rn, Op2	$R_d = R_n Op2$
Exclusive OR	a ^ b	eor Rd, Rn, Op2	$R_d = R_n ^ Op2$
Bitwise NOT	a = ~b	mvn Rd, Rn	$R_d = \sim R_n$

Bitwise versus C Boolean Operators

Boolean Operators		Bitwise 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	Bitwise NOT

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

bitwise & **versus** **Boolean &&**

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

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

bitwise ~ **versus** **Boolean !**

`~0x01 = 0xffffffff (bitwise)`

`!0x01 = 0x0 (Boolean)`

`!0xff = 0x0 (Boolean)`

The act (operation) of Masking

a =

a ₇	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

<op> <op> <op> <op> <op> <op> <op> <op>

b =

b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

result =

c ₇	c ₆	c ₅	c ₄	c ₃	c ₂	c ₁	c ₀
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value with a specific op:
 - **orr**: 0 passes bit unchanged, 1 sets bit to 1 (a = b | c; // in C)
 - **eor**: 0 passes bit unchanged, 1 inverts the bit (a = b ^ c; // in C)
 - **and**: 0 clears the bit, 1 passes bit unchanged (a = b & c; // in C)

Mask on

force bits to 1 "mask on" operation

- 1 to **set a bit to 1**
- 0 to let a bit through **unchanged**

orrr r1, r2, r3

r1 = r2 | r3; // in C

Example: force lower 16 bits to 1

DATA: r2 0xab ab ab 77

orrr

MASK: r3 0x00 00 ff ff

unchanged

forces to a 1

RSLT: r1 0xab ab ff ff

Example: force lower 8 bits to 1

DATA: r2 0xab ab ab 77

orrr r1 r2, 0xff

r1 = r2 | 0xff; // in C

RSLT: r1 0xab ab ab ff

Mask off

force bits to 0 "mask off" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to **let a bit through unchanged**

and r1, r2, r3

r1 = r2 & r3; // in C

Example: force lower 8 bits to 0

DATA: r2 0xab ab ab 77

and

MASK: r3 0xff ff ff 00

unchanged

forces to a 0

RSLT: r1 0xab ab ab 00

Example: force lower 8 bits to 0

DATA: r2 0xab ab ab 77

and r1 r2, 0xfffffff00

r1 = r2 & 0xfffffff00; // in C

RSLT: r1 0xab ab ab 00

Extracting (Isolate) a Field of Bits with a mask

extract top 8 bits of r2 into r1

- 0 to **set a bit to 0** ("clears the bit")
- 1 to **let a bit through unchanged**

and r1, r2, r3

DATA: r2 0xab ab ab 77

and

MASK: r3 0xff 00 00 00

unchanged

forces to a 0

RSLT: r1 0xab 00 00 00

extract top 8 bits of r2 into r1

DATA: r2 0xab ab ab 77

and r1, r2, 0xff000000

RSLT: r1 0xab 00 00 00

r1 = r2 & 0xff000000; // in C

Finding if a bit is set

query the status of a bit "bit status" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

and r1, r2, 0x02

cmp r1, 0

beq .Lendif

// code for is set

.Lendif:

```
unsigned int r1, r2;  
// code  
r1 = r2 & 0x02  
if (r1 != 0) {  
    // code for is set  
}
```

Example is bit 1 set

DATA: r2 0xab ab ab 77

and

MASK: 0x00 00 00 02 is bit 1 set?

forces to a 0 unchanged

RSLT: r1 0x00 00 00 02 != 0 if set

```
unsigned int r2;  
// code  
if ((r2 & 0x02) != 0) {  
    // code for is set  
}
```

Even/Odd

Even or odd, check LSB (same as mod %2)

```
check LSB (bit 0) if set then odd, else even  
    and r1, r2, 0x01  
    cmp r1, 0x01  
    bne .Lendif  
    // code for handling odd numbers  
.Lendif:
```

```
unsigned int r2;  
// code  
if ((r2 & 0x01) != 0) {  
    // code for handling odd numbers  
}
```

DATA: r2 0xab ab ab 77
and
MASK: r3 0x00 00 00 01 (mod 2 even or odd)
forces to a 0 unchanged
RSLT: r1 0x00 00 00 01 (odd)

MOD %<power of 2>

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

mask = $2^k - 1$ so for mod 16, mask = 16 - 1 = 15

and r1, r2, r3

Example: %2

DATA: r2 0xab ab ab 77

and

MASK: r3 0x00 00 00 01

forces to a 0

(mod 2 even or odd)

unchanged

RSLT: r1 0x00 00 00 01

(odd)

Example: Mod 16

DATA: r2 0xab ab ab 77

and

MASK: r3 0x00 00 00 0f

forces to a 0

(mod 16)

unchanged

RSLT: r1 0x00 00 00 07

Flipping bits: bit toggle

Used in PA7/PA8

invert (*flip*) bits "bit toggle" operation

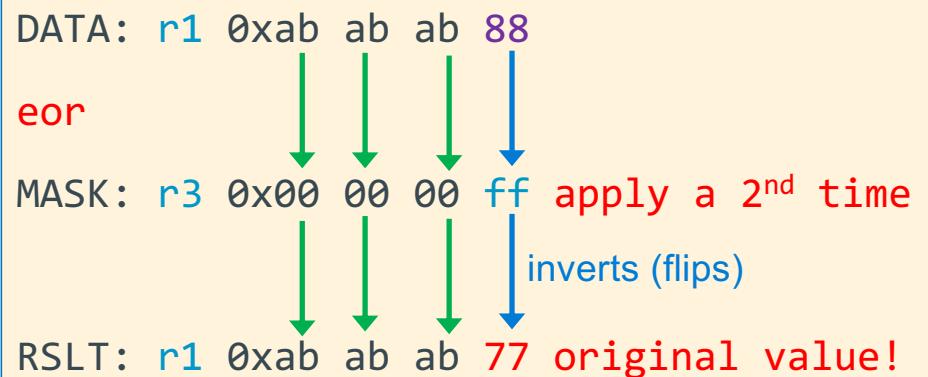
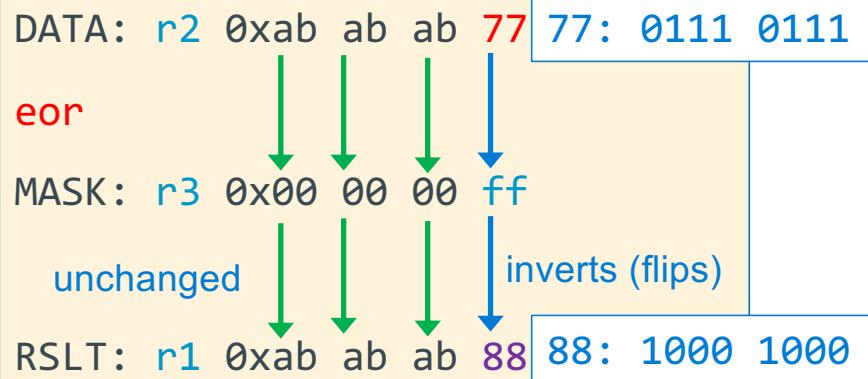
- 1 **will flip the bit**
- 0 to **let a bit through**
`eor r1, r2, r3`
- Observation: When applied twice, it returns the original value (symmetric encoding)
- With a mask of all 1's is a 1's compliment

Example: *flip* the lower 8-bits

`eor r1, r2, 0xff`

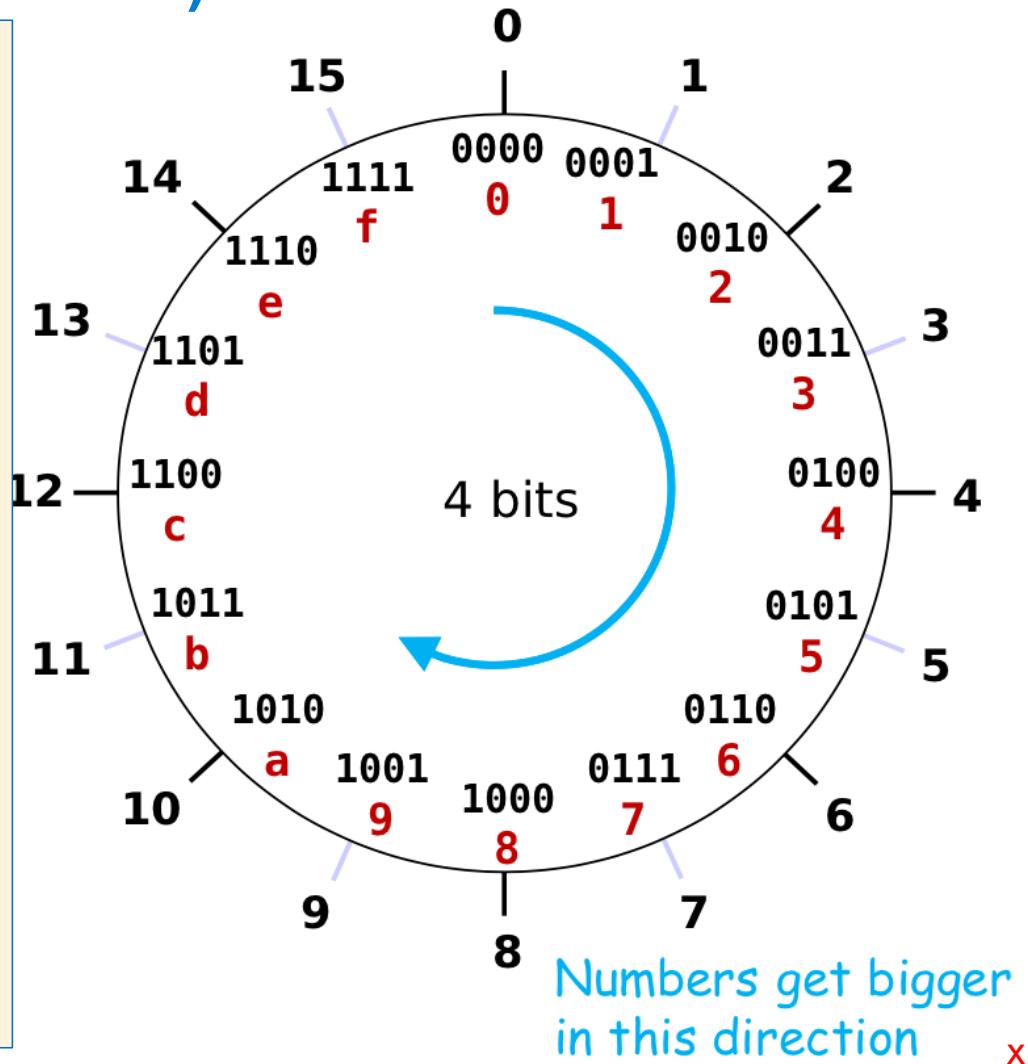
```
unsigned int r1, r2;  
r1 = r2 ^ 0xff;
```

Example: invert (*flip*) the lower 8-bits



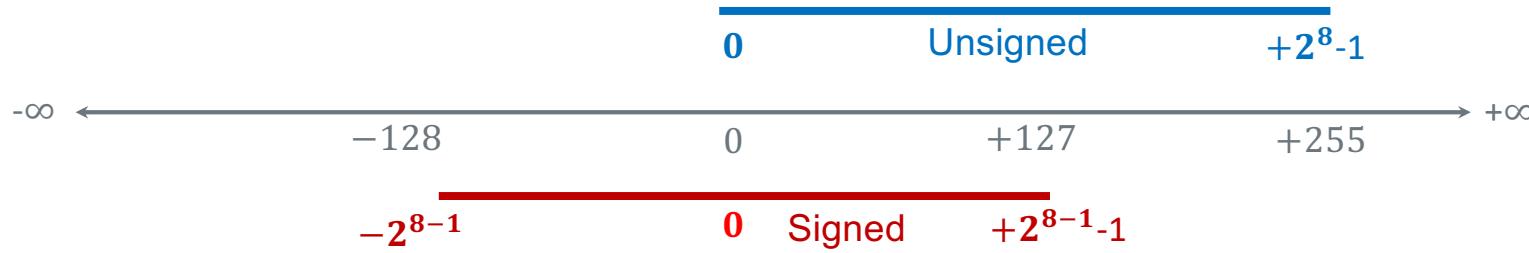
Unsigned Integers (positive numbers) with Fixed # of Bits

- 4 bits is $2^4 = \text{ONLY } 16$ distinct values
- Modular (C operator: `%`) or clock math
 - Numbers start at 0 and “wrap around” after 15 and go back to 0
- Keep adding 1
 - wraps (clockwise)
 $0000 \rightarrow 0001 \dots \rightarrow 1111 \rightarrow 0000$
- Keep subtracting 1
 - wraps (counter-clockwise)
 $1111 \rightarrow 1110 \dots \rightarrow 0000 \rightarrow 1111$
- Addition and subtraction use normal “carry” and “borrow” rules, just operate in binary



Problem: How to Encode Both Positive and Negative Integers

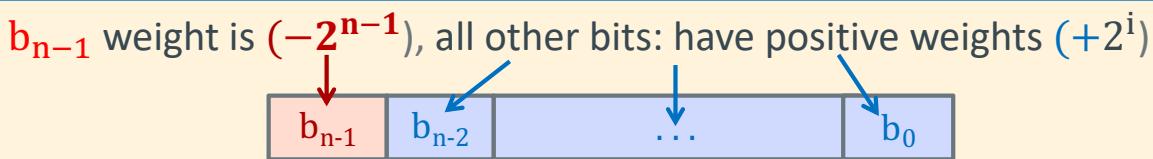
- How do we represent the negative numbers within a fixed number of bits?
 - Allocate some bit patterns to **negative** and others to **positive** numbers (and zero)
- 2^n distinct bit patterns to encode positive and negative values
- **Unsigned values:** $0 \dots 2^n - 1$ -1 comes from counting 0 as a "positive" number
- **Signed values:** $-2^{n-1} \dots 2^{n-1} - 1$ (dividing the range in ~ half including 0)
- **On a number line (below):** 8-bit integers – signed and unsigned (e.g., `char` in C)



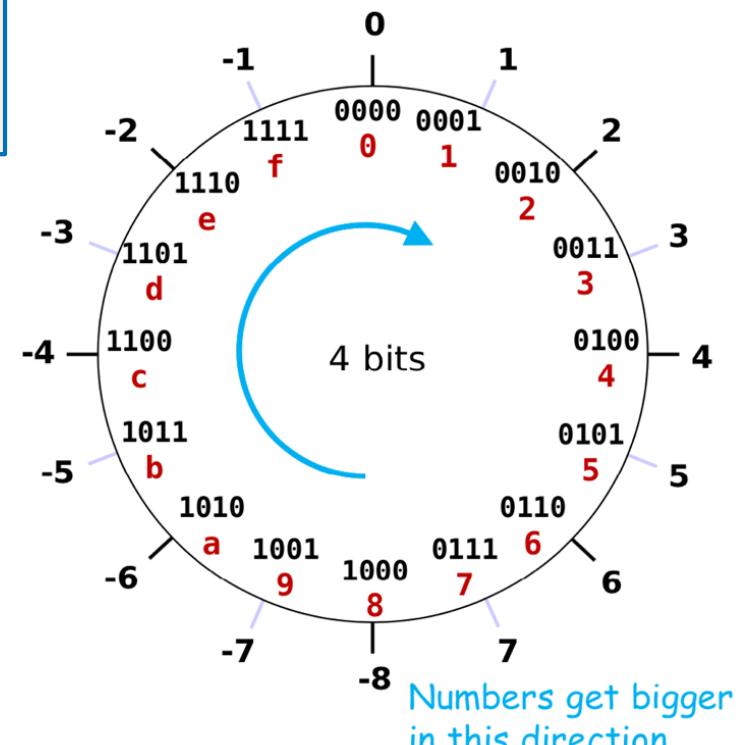
Same “width” (same number of encodings), just shifted in value

Two's Complement: The MSB Has a Negative Weight

$$2\text{'s Comp} = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_12^1 + b_02^0$$

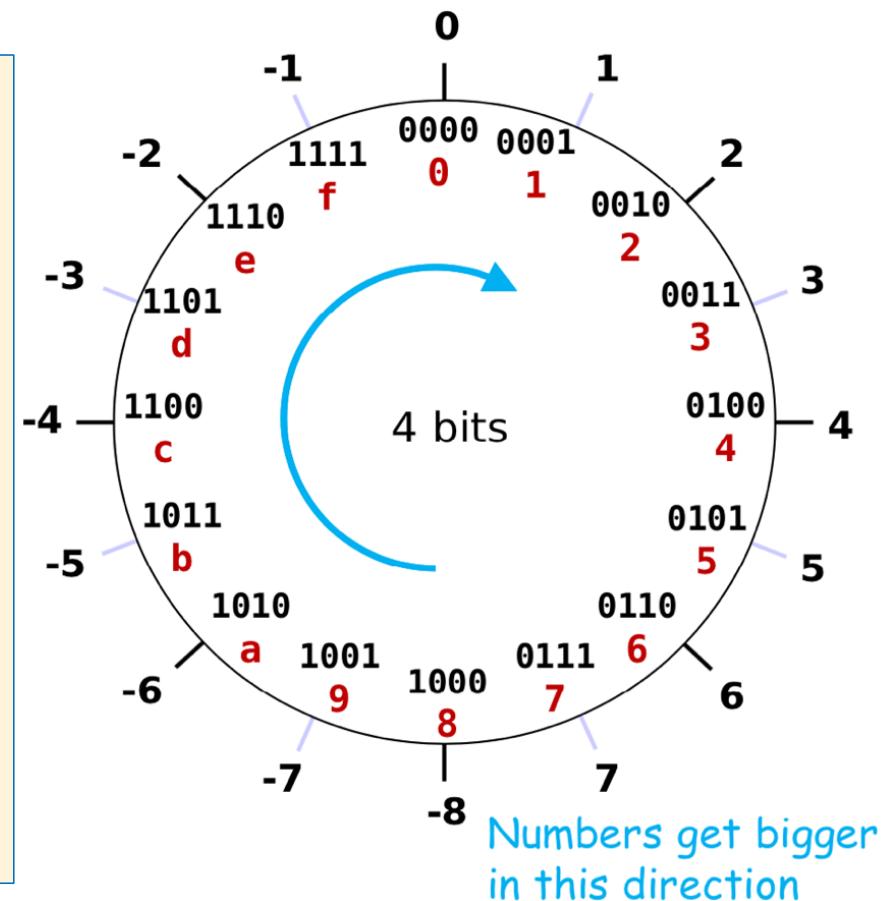


- 4-bit ($w = 4$) weight $= -2^{4-1} = -2^3 = -8$
 - 1010_2 **unsigned**:
 $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10$
 - 1010_2 **two's complement**:
 $-1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 2 = -6$
 - -8 in **two's complement**:
 $1000_2 = -2^3 + 0 = -8$
 - -1 in **two's complement**:
 $1111_2 = -2^3 + (2^3 - 1) = -8 + 7 = -1$



2's Complement Signed Integer Method

- Positive numbers encoded same as unsigned numbers
- All negative values have a one in the leftmost bit
- All positive values have a zero in the leftmost bit
 - This implies that 0 is a positive value
- Only one zero
- For n bits, Number range is $-(2^{n-1})$ to $+(2^{n-1} - 1)$
 - Negative values "go 1 further" than the positive values
- Example: the range for 8 bits:
-128, -127, ... 0, ... 126, +127
- Example the range for 32 bits:
-2147483648 .. 0, .. +2147483647
- Arithmetic is the same as with unsigned binary!*



Sign Extension (how type promotion works)

- Sometimes you need to work with integers encoded with different number of bits

8 bits (char) -> (16 bits) short -> (32 bits) int

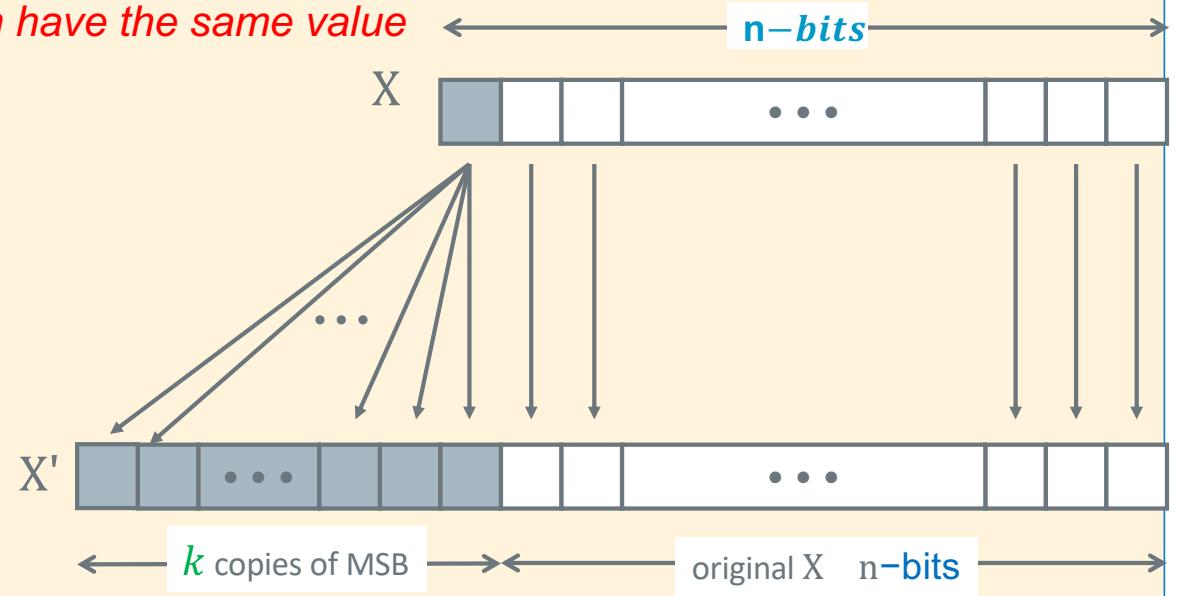
- Sign extension increases the number of bits: n -bit wide signed integer X, EXPANDS to a wider $n+k$ -bit + k -bit signed integer X' where both have the same value

Unsigned

- Just add leading zeroes to the left side

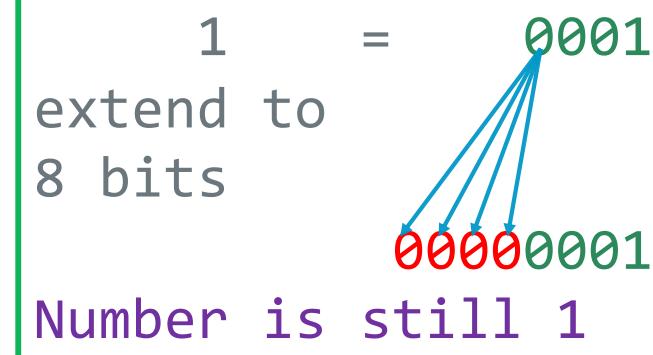
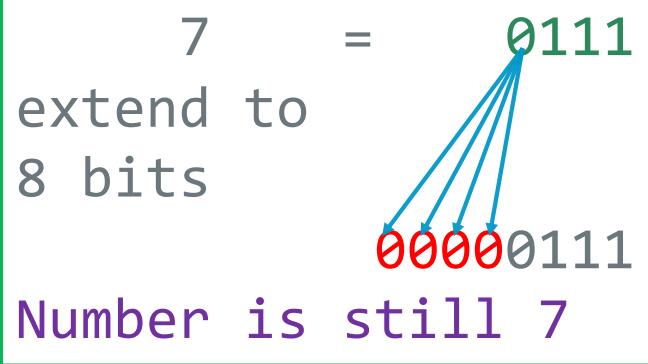
Two's Complement Signed:

- If positive, add leading zeroes on the left
 - Observe: Positive stay positive
- If negative, add leading ones on the left
 - Observe: Negative stays negative



Example: Two's Complement Sign or bit Extension - 1

- Adding 0's in front of a positive numbers does not change its value



Example: Two's Complement Sign or bit Extension -2

- Adding 1's if front of a negative number does not change its value

$$\begin{array}{rcl} 7 & = & 0111 \\ & & \downarrow \quad \downarrow \quad \downarrow \\ \text{invert} & = & 1000 \\ \text{add } 1 & + & \underline{1} \\ -7 & = & 1001 \end{array}$$

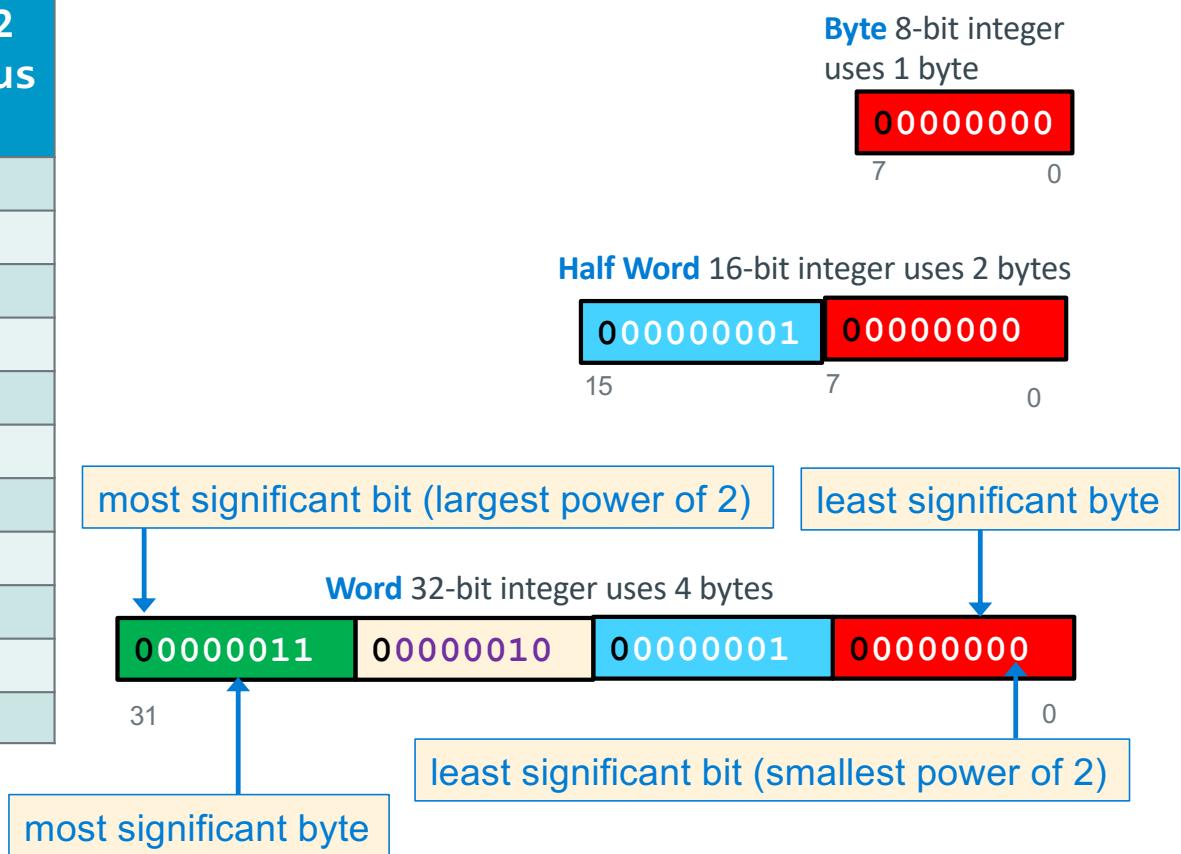
$$\begin{array}{rcl} -7 & = & 1001 \\ \text{extend to} & & \\ 8 \text{ bits} & & \\ & & \begin{array}{c} 1 \\ \diagup \quad \diagdown \\ 11111001 \end{array} \end{array}$$

$$\begin{aligned} 1001 &= -8 + 1 = -7 \\ 11111001 &= \\ (-128 + 64 + 32 + 16 + 8) &+ 1 \\ &= -8 + 1 = -7 \end{aligned}$$

$$\begin{array}{rcl} 7 & = & 00000111 \\ & & \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ \text{invert} & = & 11111000 \\ \text{add } 1 & + & \underline{1} \\ -7 & = & 11111001 \end{array}$$

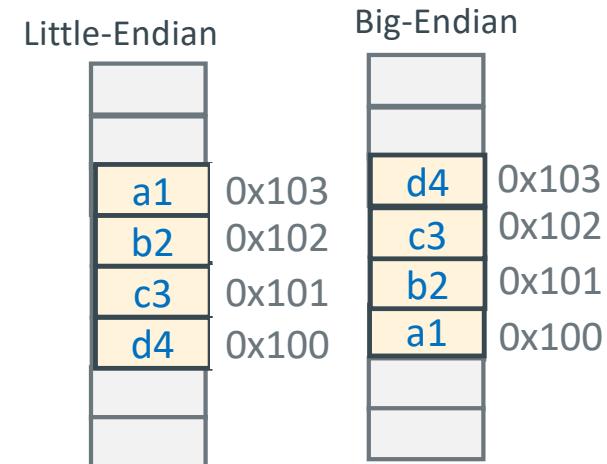
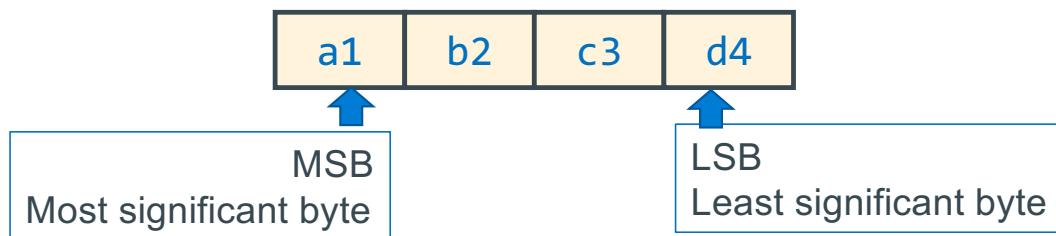
Different Type of Numbers each have a Fixed # of Bits Spanning one or more contiguous bytes of memory

C Data Type	AArch-32 contiguous Bytes
char (arm unsigned)	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
long long int	8
float	4
double	8
long double	8
pointer *	4



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



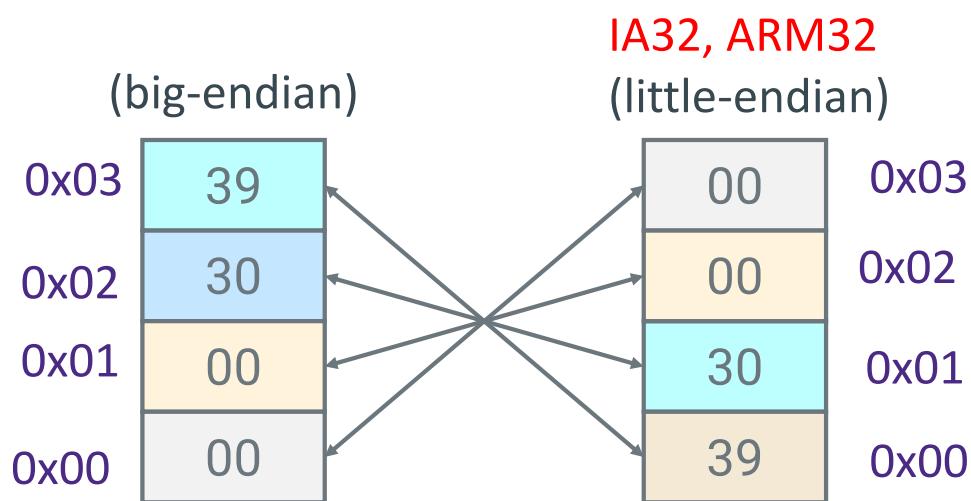
Byte Ordering Example

Decimal: 12345

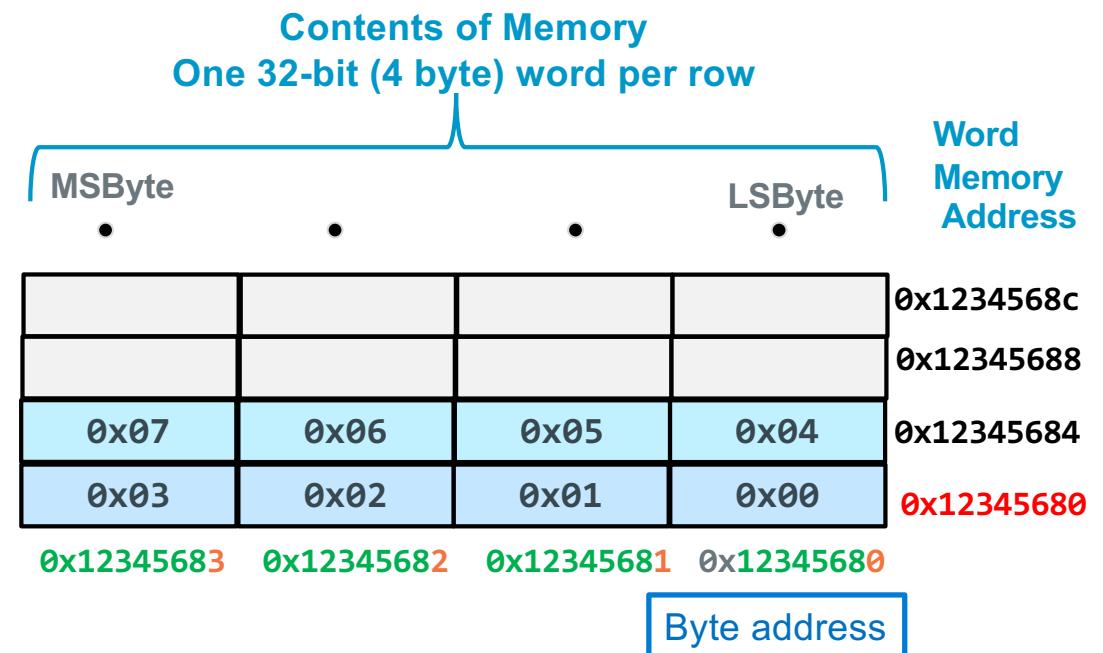
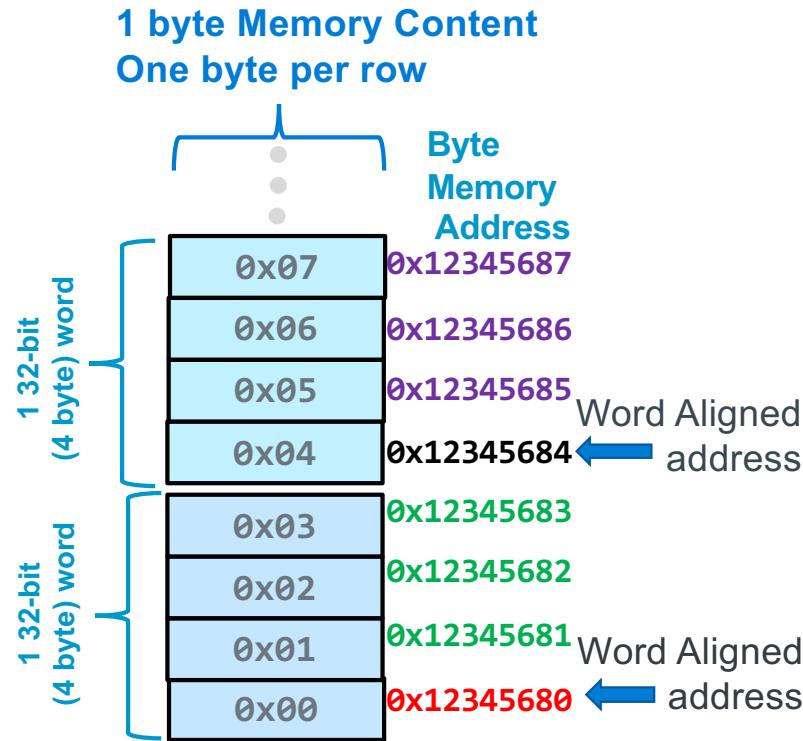
Binary: 0011 0000 0011 1001

Hex: 3 0 3 9

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

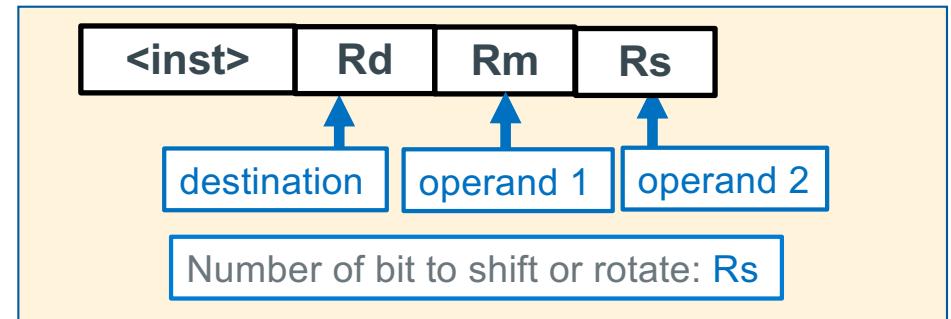
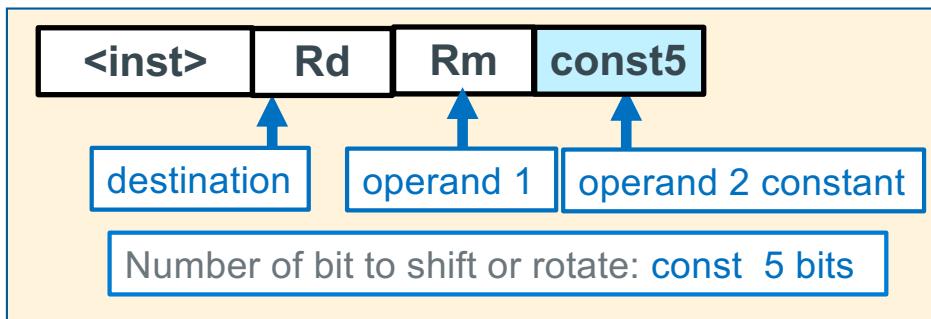


Byte Addressable Memory Shown as 32-bit words



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

Bit Shifting ARM and C Equivalents



<i>Instruction</i>	<i>ARM</i>	<i>C</i>	<i>Diagram</i>
Logical Shift Left 0 fills on right $0 \leq n \leq 32$ bits	lsl R_d, R_m, n lsl R_d, R_m, R_s	int x; or unsigned int x $x \ll n;$	A horizontal red bar representing a 32-bit register. Bit b31 is on the left and bit b0 is on the right. A blue arrow points from bit b31 to a red square labeled 'C'. A blue arrow points from bit b0 back to the register. A blue arrow also points from bit b0 to the number 0 at the far right.
Logical Shift Right 0 fills on left $0 \leq n \leq 32$ bits	lsr R_d, R_m, n lsr R_d, R_m, R_s	unsigned int x; $x \gg n;$	A horizontal red bar representing a 32-bit register. Bit b31 is on the left and bit b0 is on the right. A blue arrow points from the number 0 at the far left to bit b31. A blue arrow points from bit b0 to a red square labeled 'C' at the far right.
Arithmetic Shift Right Sign extends on left $0 \leq n \leq 32$ bits	asr R_d, R_m, n asr R_d, R_m, R_s	int x; $x \gg n;$	A horizontal red bar representing a 32-bit register. Bit b31 is on the left and bit b0 is on the right. A blue arrow points from bit b31 to a red square labeled 'C' at the far right. A blue arrow also points from bit b31 back to the register.
Rotate Right $0 \leq n \leq 32$ bits	ror R_d, R_m, n ror R_d, R_m, R_s	ror n in a 32-bit registers unsigned int x; $x = (x \gg n) (x \ll (32 - n));$	A horizontal red bar representing a 32-bit register. Bit b31 is on the left and bit b0 is on the right. A blue arrow points from bit b31 to a red square labeled 'C' at the far right. A blue arrow also points from bit b0 back to the register.

Shift Operations in C

- n is number of bits to shift a variable x of width w bits
- Shifts by $n < 0$ or $n \geq w$ are *undefined*
- Left shift ($x \ll n$) – **Multiplies by 2^N**
 - Shift N bits left, Fill with 0s on right
- In C: behavior of `>>` is determined by **compiler**
 - gcc: it depends on data type of x (signed/unsigned)
- Right shift ($x \gg n$) - **Divides by 2^N**
 - Logical shift (for unsigned variables)
 - Shift N bits right, Fill with 0s on left
 - Arithmetic shift (for signed variables) – Sign Extension
 - Shift N bits right while Replicating the most significant bit on left
 - Maintains sign of x
- In Java: logical shift is `>>>` and arithmetic shift is `>>`



Logical 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
```

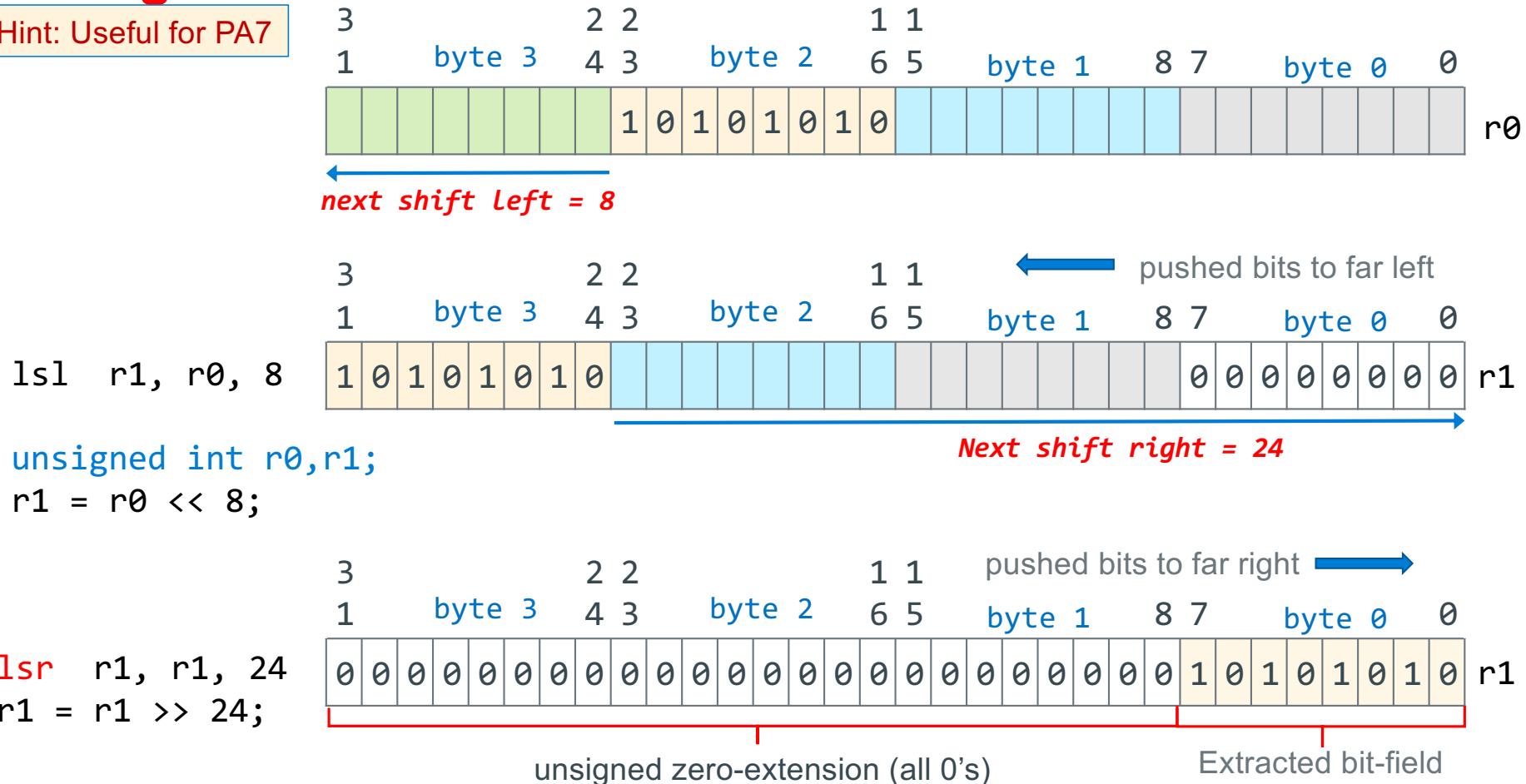


```
ror r2, r0, 8  
r0 0xab ab ab 77  
r2 0x77 ab ab ab
```

Extracting/Isolating Unsigned Bitfields

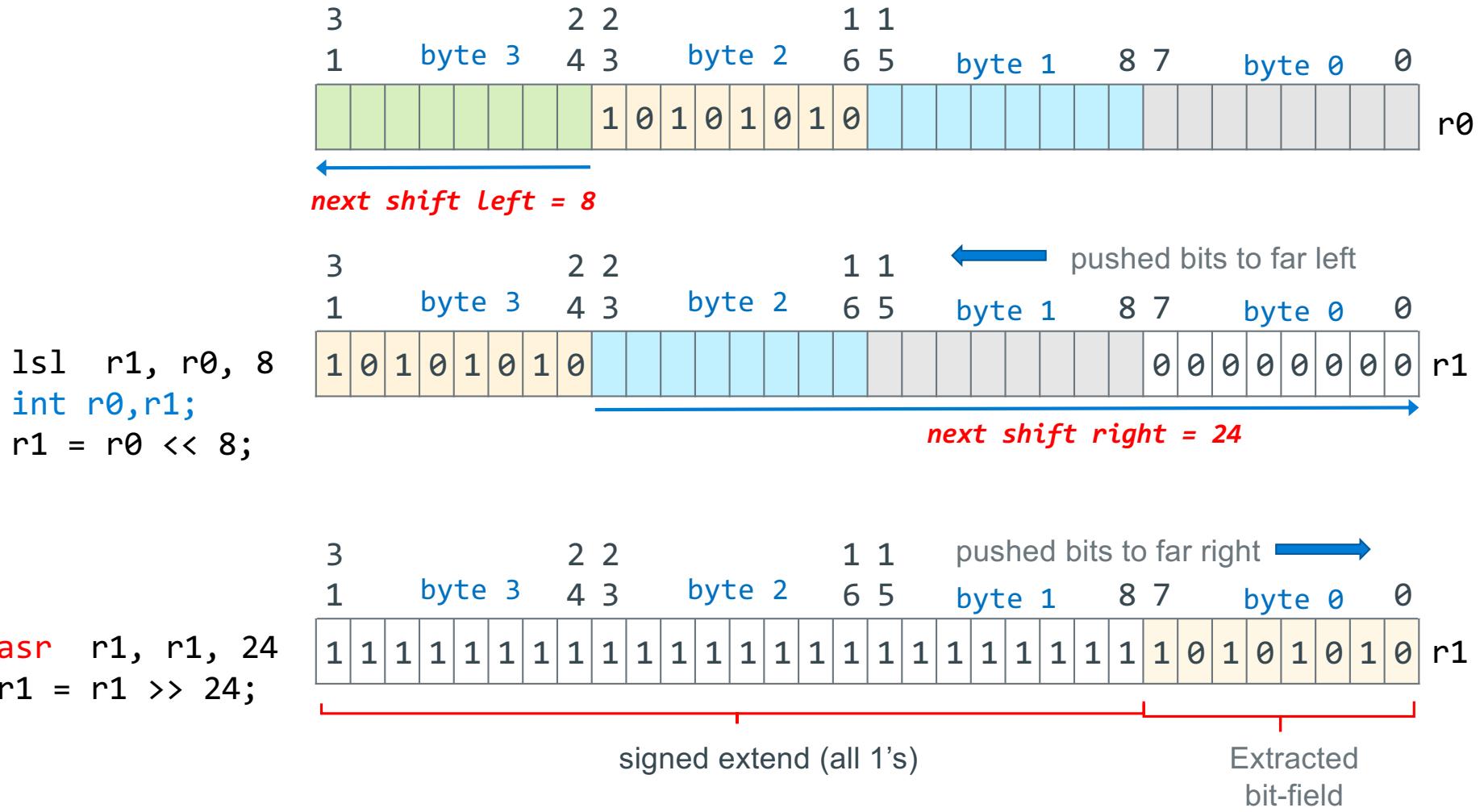
Hint: Useful for PA7

- Move byte 2 in r0 to byte 0 in r1



Extracting Signed Bitfields

- Move byte 2 in r0 to byte 0 in r1



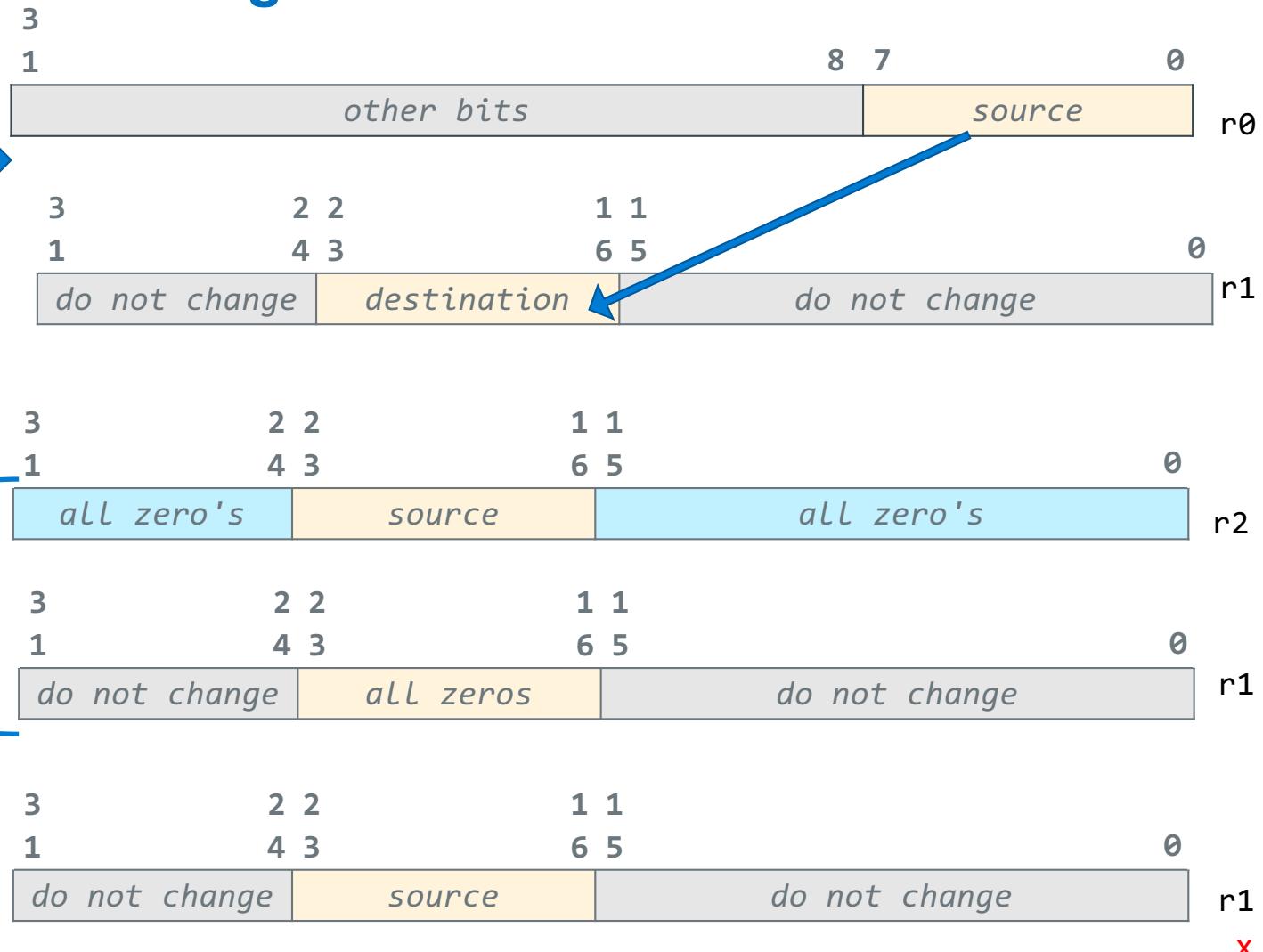
Inserting Bitfields – Inserting Source Field into Destination Field

Task: Insert source into destination

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

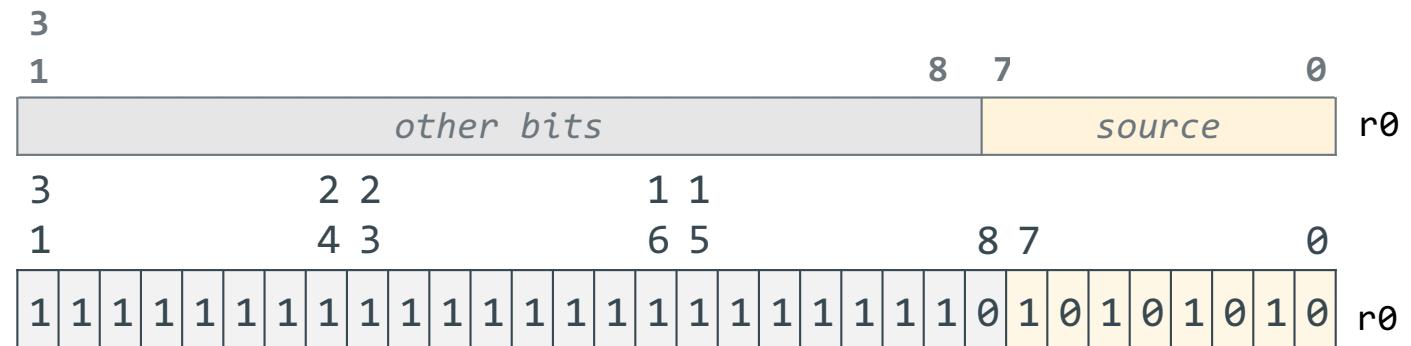
Approach
 (1) isolate source field
 (2) clear destination field
 (3) Bitwise **or** together

```
orr    r1, r1, r2
r1 = r1 | r2;
```



results in

Inserting Bitfields – Isolating the Source Field



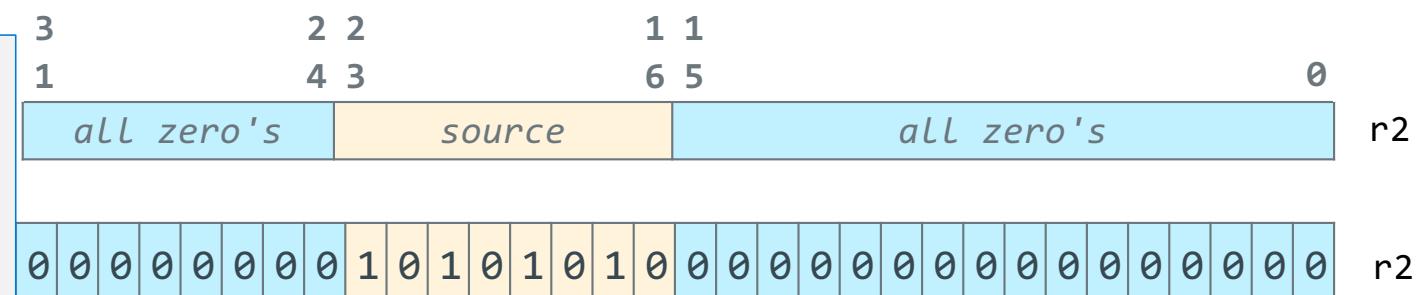
isolate source field

lsl r2, r0, 24

lsrc r2, r2, 8

```
r2 = r0 << 24;
```

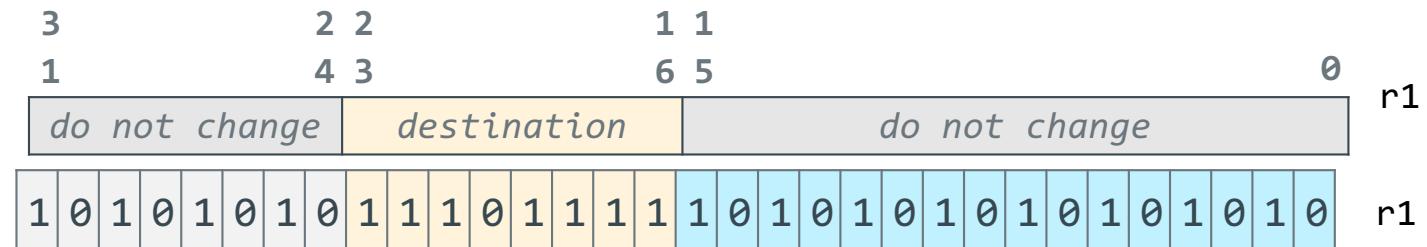
```
r2 = r2 >> 8;
```



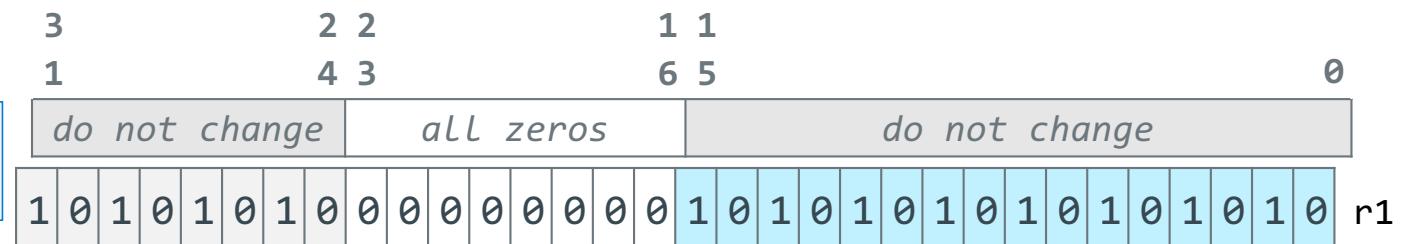
Inserting Bitfields – Clearing the Destination Field

```
clear the  
destination field  
ror    r1, r1, 24  
r1=(r1>>24)|(r1<<8);
```

```
lsl    r1, r1, 8  
r1 = r1 << 8;
```

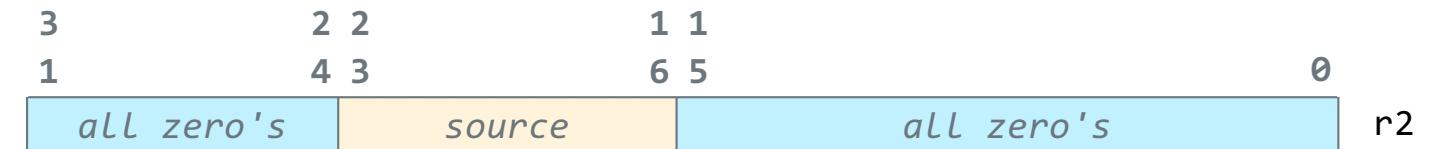


```
ror    r1, r1, 16  
r1= (r1>>16)|(r1<<16);
```

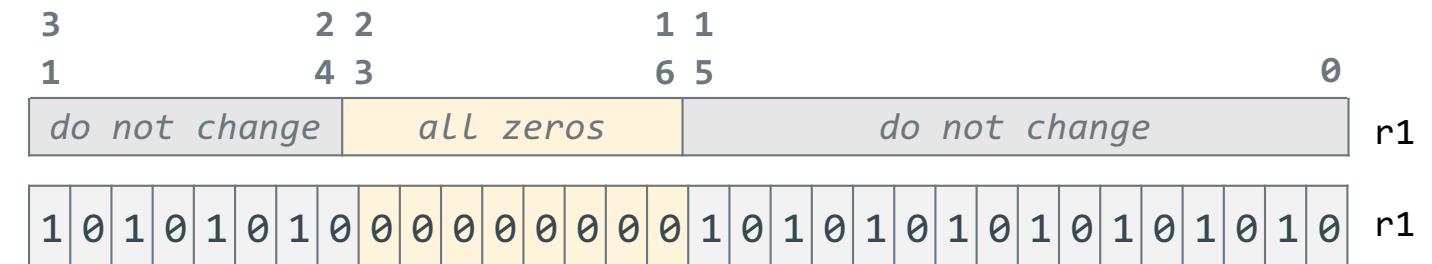


Inserting Bitfields – Combining Isolated Source and Cleared Destination

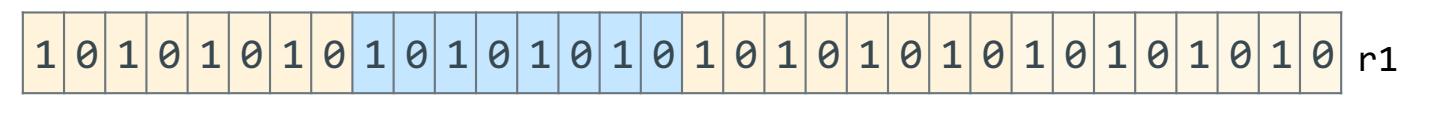
isolated source



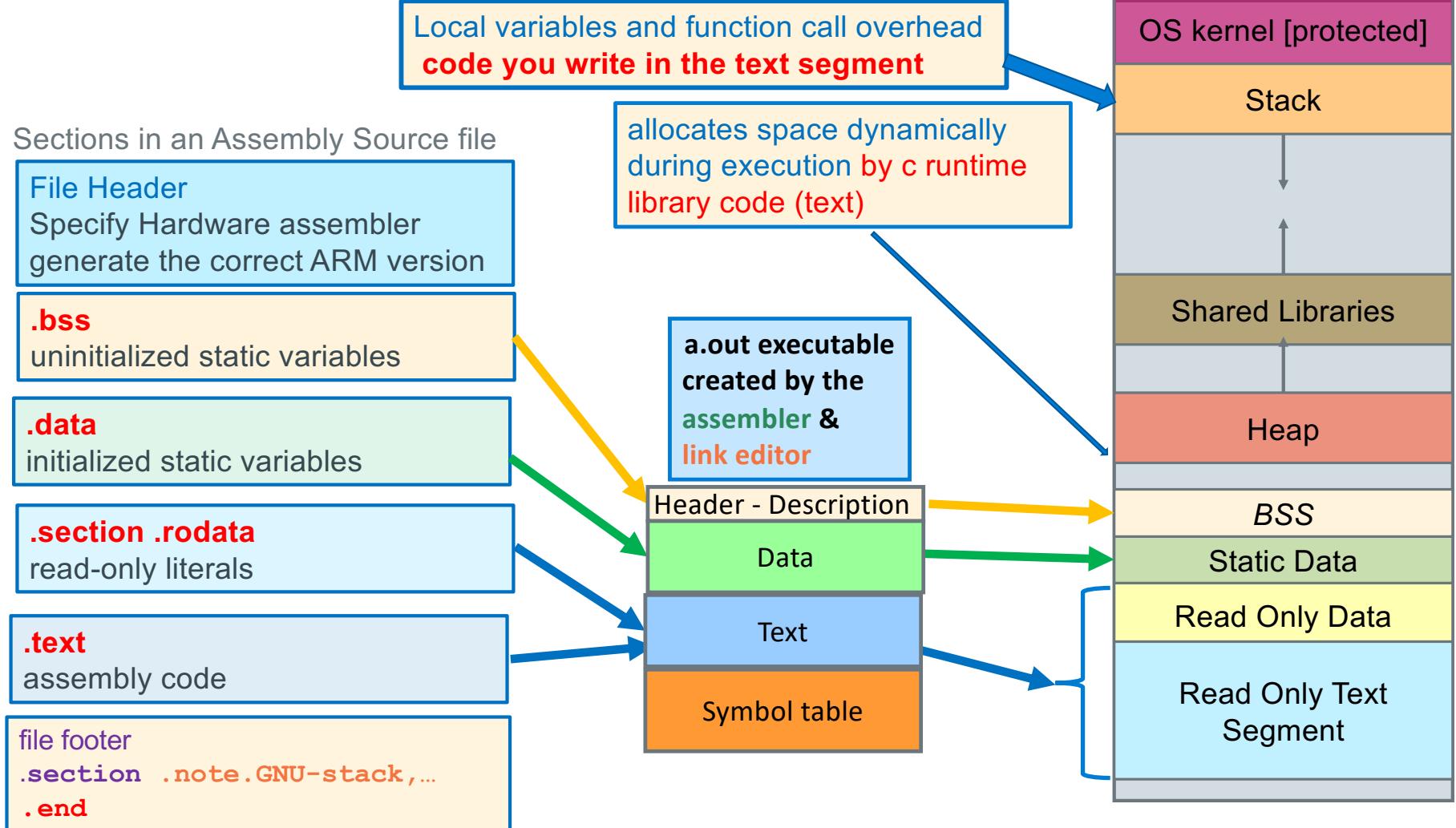
field cleared in
destination



```
inserted field  
orr      r1, r1, r0  
r1 = r1 | r0;
```



Assembly Source File to Executable to Linux Memory



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)
```

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 uninitialized globals)
    .bss
// Data Segment (only when you have initialized 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`

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

Assembler Directives: .equ and .equiv

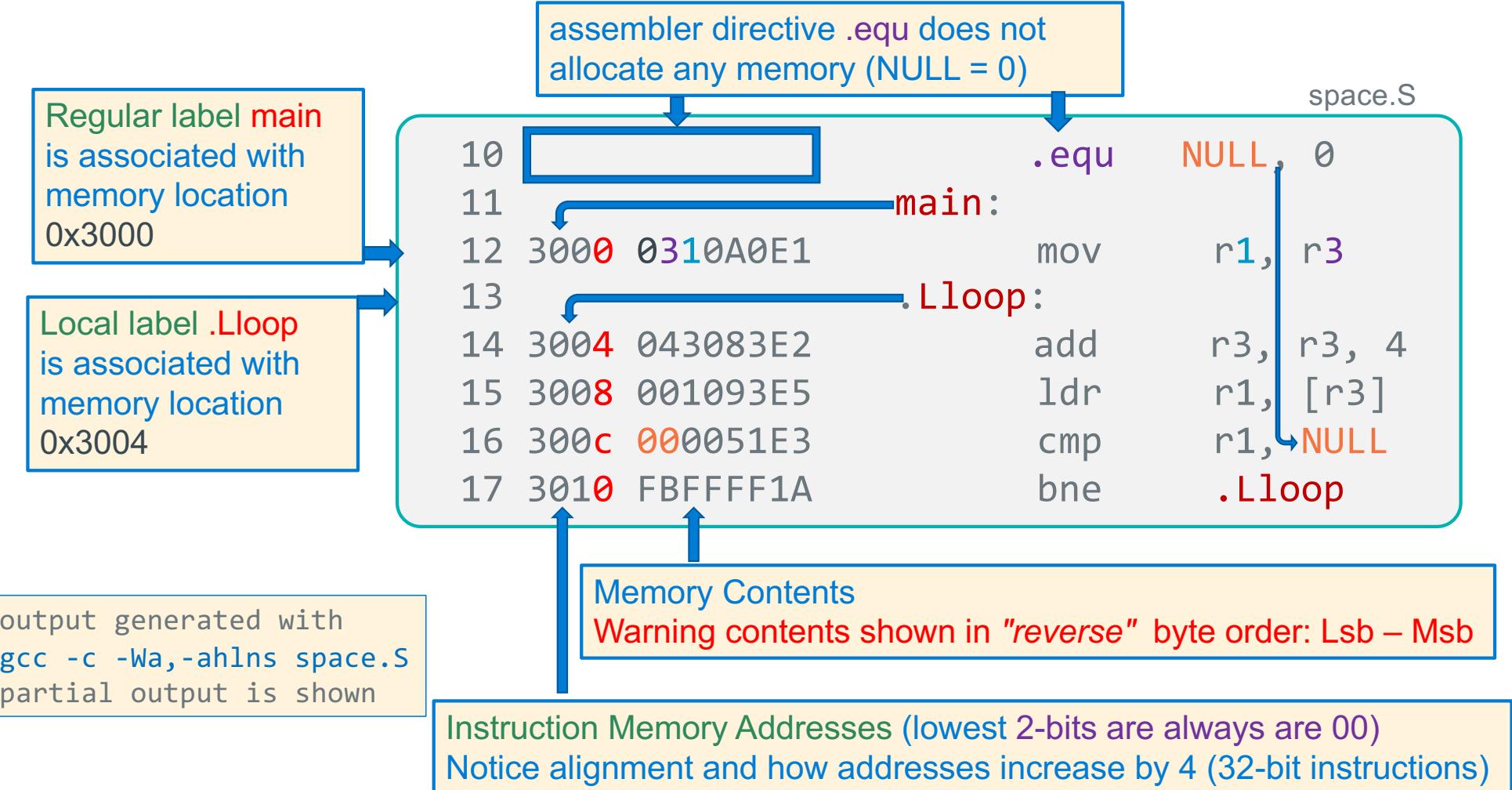
```
.equ    BLKSZ, 10240      // buffer size in bytes
.equ    BUFCNT, 100*4      // buffer for 100 ints
.equ    BLKSZ, STRSZ * 4   // redefine BLKSZ from here
```

.equ <symbol>, <expression>

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

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

Example: Assembler Directive and Instructions



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)**

.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_OFFSET, 4

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

.size name, bytes

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

.size name, (. - name)

```
.text
Function Header {
    .global myfunc           // make myfunc global for linking
    .type   myfunc, %function // define myfunc to be a function
    .equ    FP_OFFSET, 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)
```

Function Template

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

Function header { .global myfunc // make myfunc global for linking
                  .type   myfunc, %function // define myfunc to be a function
                  .equ    FP_OFF, 4        // fp offset in main stack frame

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

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

Function footer { .size myfunc, (. - myfunc)
```

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

Function Prologue creates stack frame

Function Epilogue removes stack frame

Preview: Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use	Register	Function Return Value Use
r0	1 st parameter	r0	8, 16 or 32-bit result, 32-bit address or least-significant half of a 64-bit result
r1	2 nd parameter	r1	most-significant half of a 64-bit result
r2	3 rd parameter		
r3	4 th parameter		

- 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

Assembler Directives: Label Scope Control (Normal Labels only)

- `.extern printf`
- `.extern fgets`
- `.extern strcpy`
- `.global fbuf`

`.extern <label>`

- **Imports** `label` (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

`.global <label>`

- **Exports** `label` (or `symbol`) to be visible outside the source file boundary (other assembly or c source)
- `label` is either a `function name` or a `global variable name`
- Only use with function names or static variables
- **Without** `.global`, `labels` are usually **local to the file** from the point where they are defined

Preview: Writing an ARM32 function

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

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

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

```
#ifndef SUM4_H
#define SUM4_H

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

#endif
```

```
#include "sum4.h"
.arch armv6
.arm
.fpu vfp
.syntax unified
.global sum4
.type sum4, %function
.equ FP_0FF, 28
// r0 = sum4(r0, r1, r2, r3)
sum4:
    push {r4-r9, fp, lr}
    add fp, sp, FP_0FF

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

    sub sp, fp, FP_0FF
    pop {r4-r9, fp, lr}
    bx lr
    .size sum4, (. - sum4)
    .section .note.GNU-stack,"",%progbits
.end
```

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

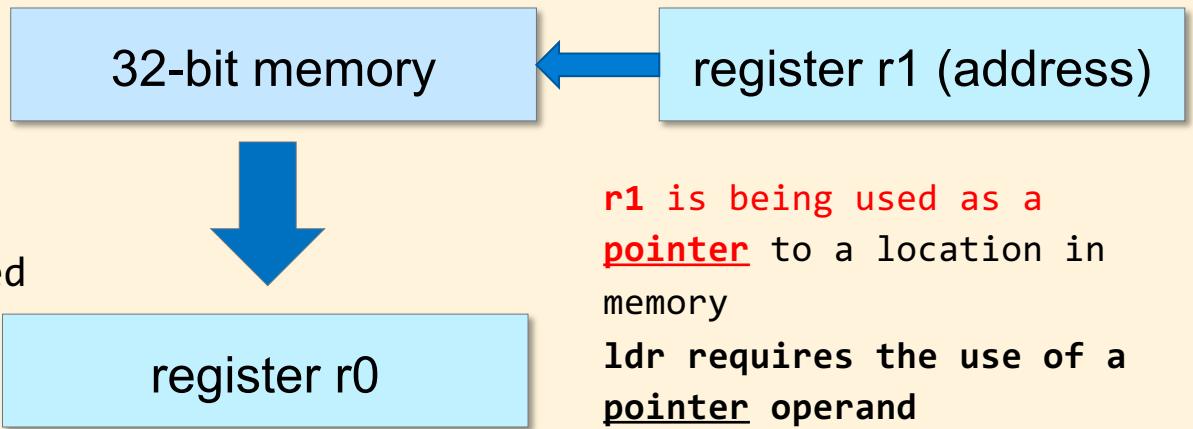
We will cover this
when we do stack frames

We will cover this
when we do stack frames

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



register r1 (address)

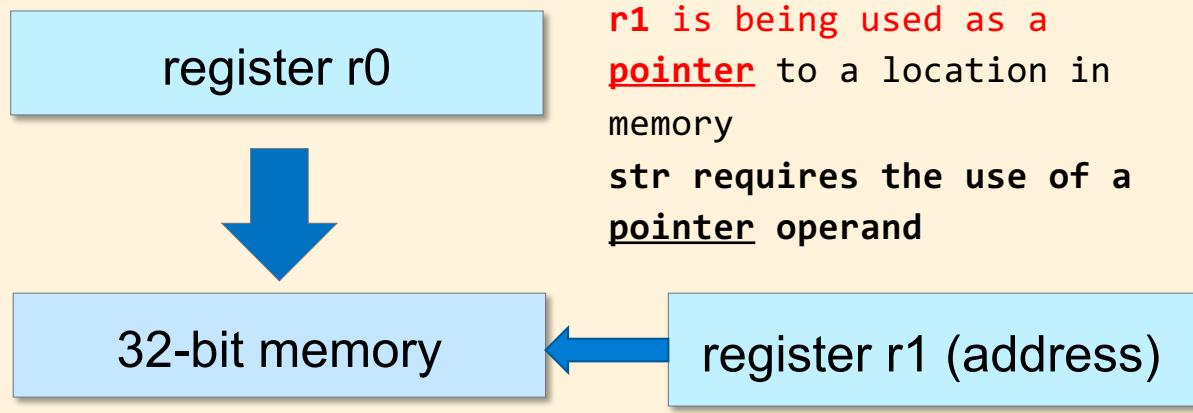
32-bit memory

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)



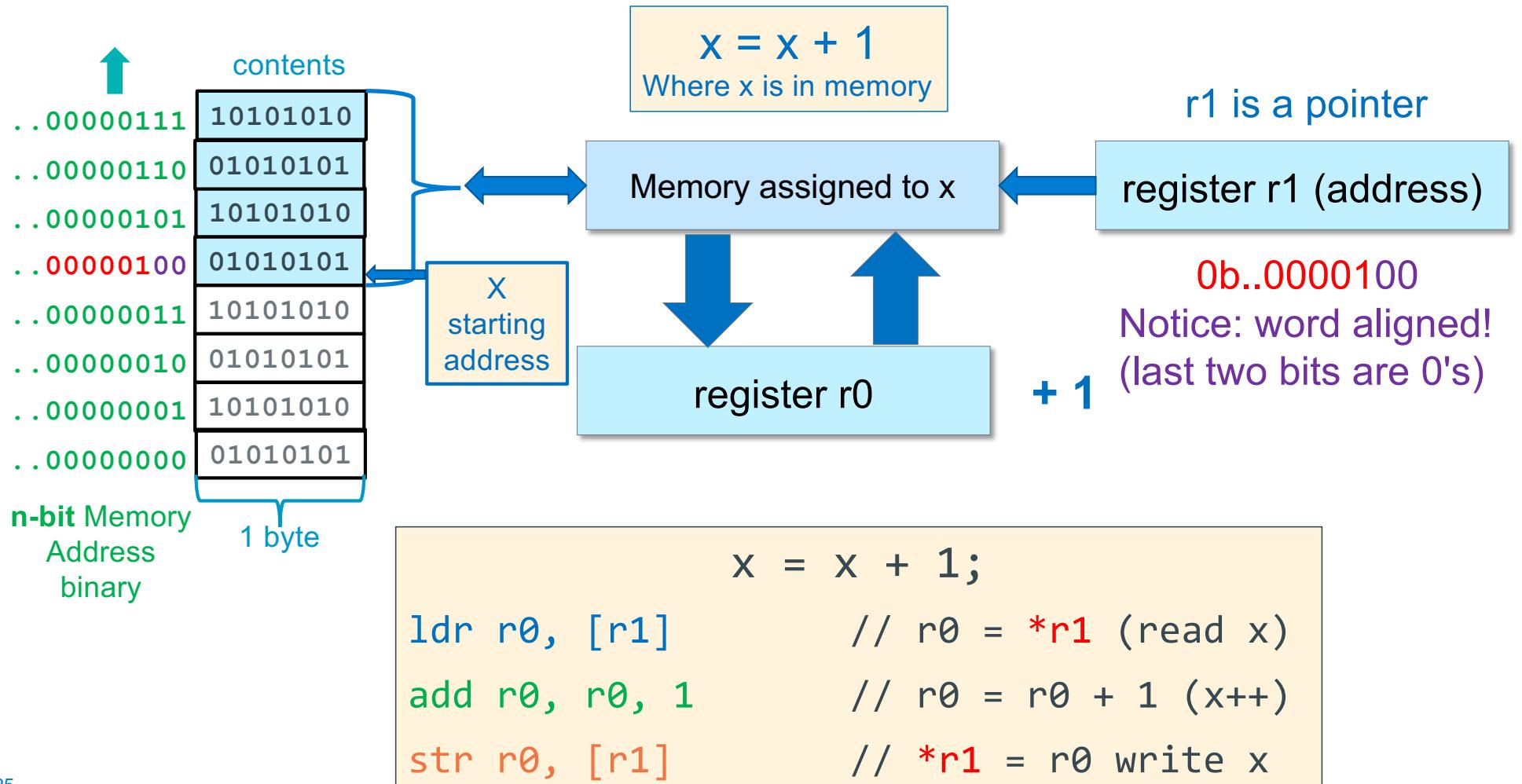
register r1 (address)

32-bit memory

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

str requires the use of a
pointer operand

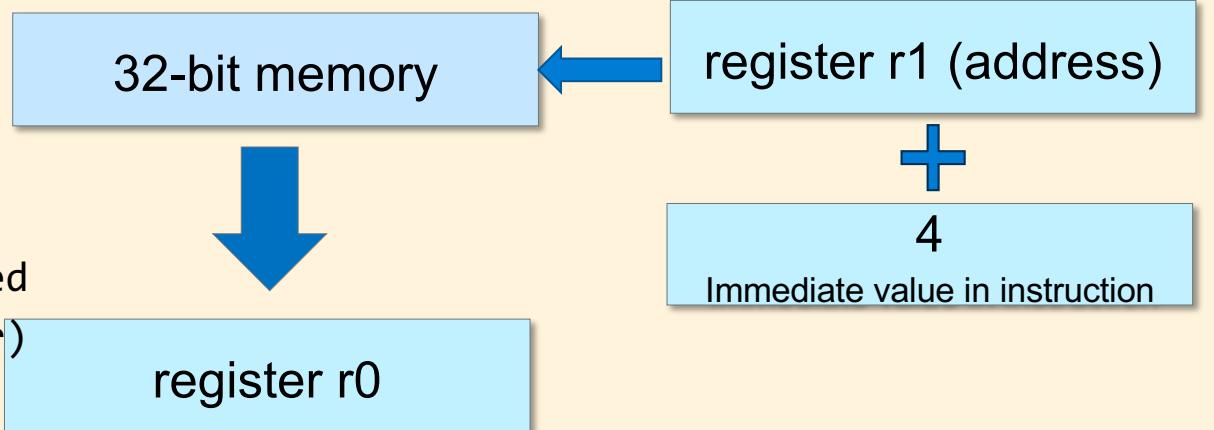
Example Base Register Addressing Load – Modify – Store



Load/Store: Register Base Addressing + Immediate

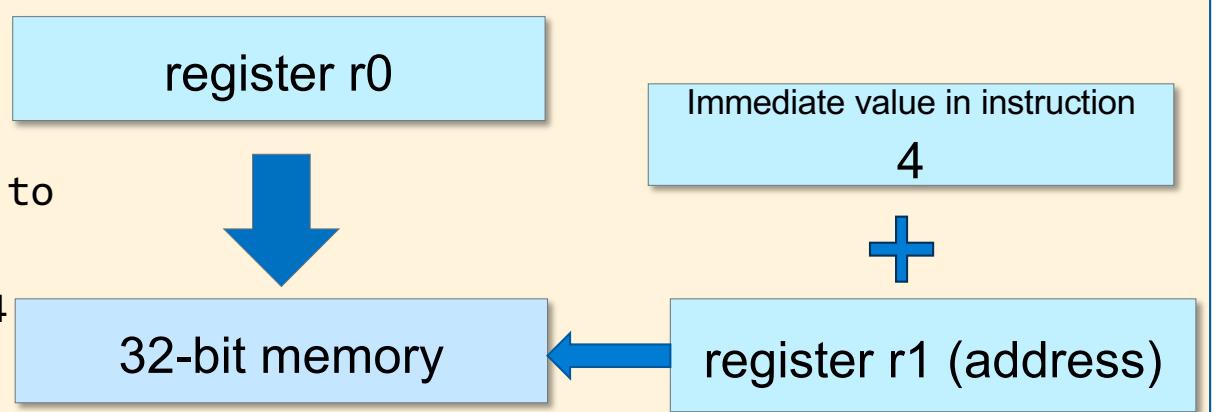
ldr r0, [r1, 4]

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

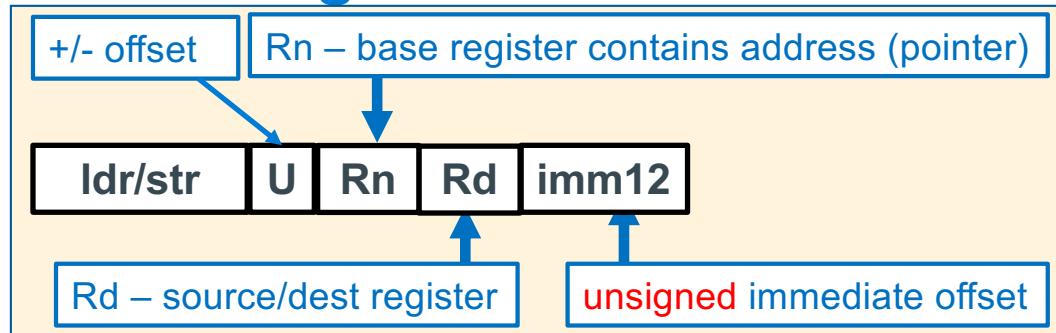


str r0, [r1, 4]

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

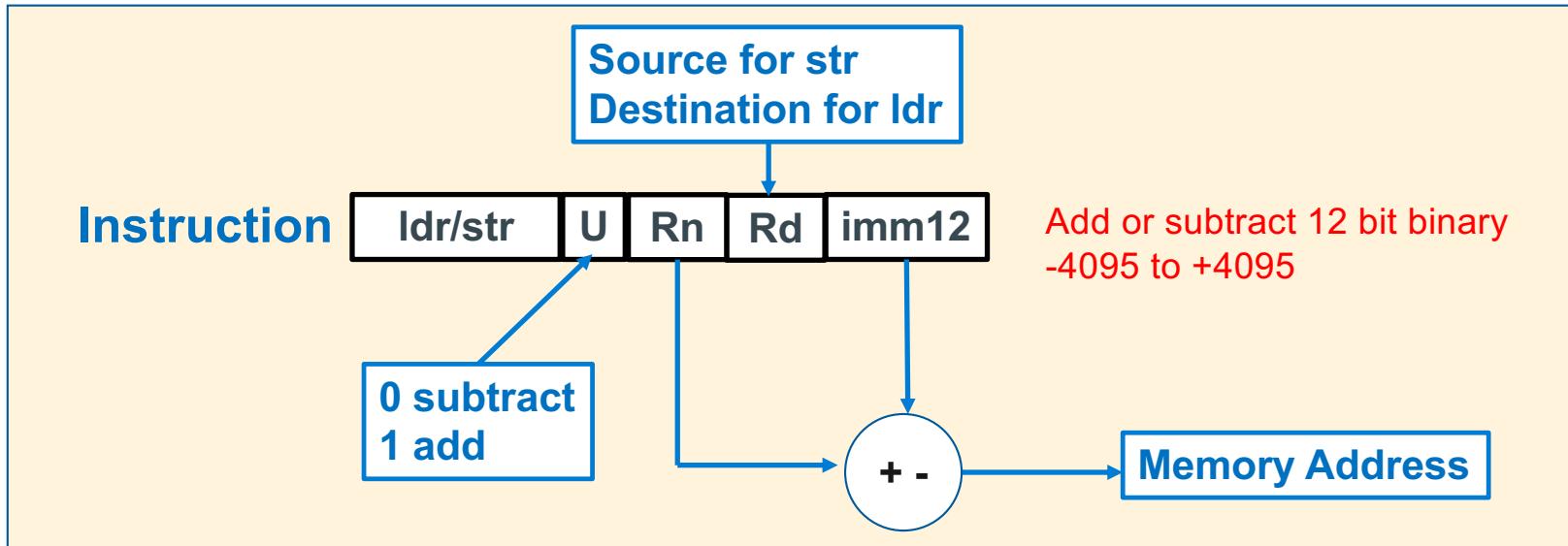


LDR/STR – Base Register + Immediate Offset Addressing



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

ldr/str Register Base + Immediate Offset Addressing



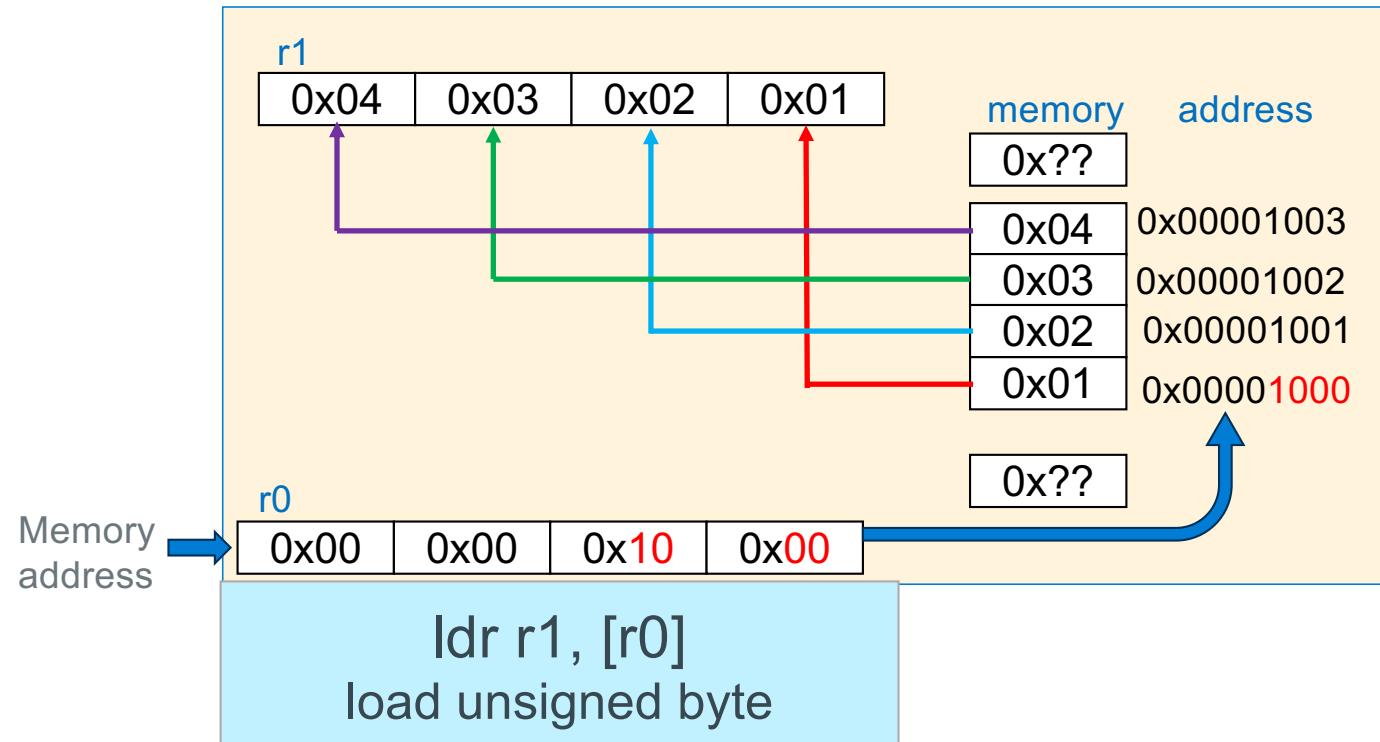
Syntax	Address	Examples
ldr/str Rd, [Rn, +/- constant] constant is in bytes ldr/str Rd, [Rn]	Rn + or - constant <i>same</i> →	ldr r0, [r5,100] str r1, [r5, 0] str r1, [r5]
		x

Loading and Storing: Variations List

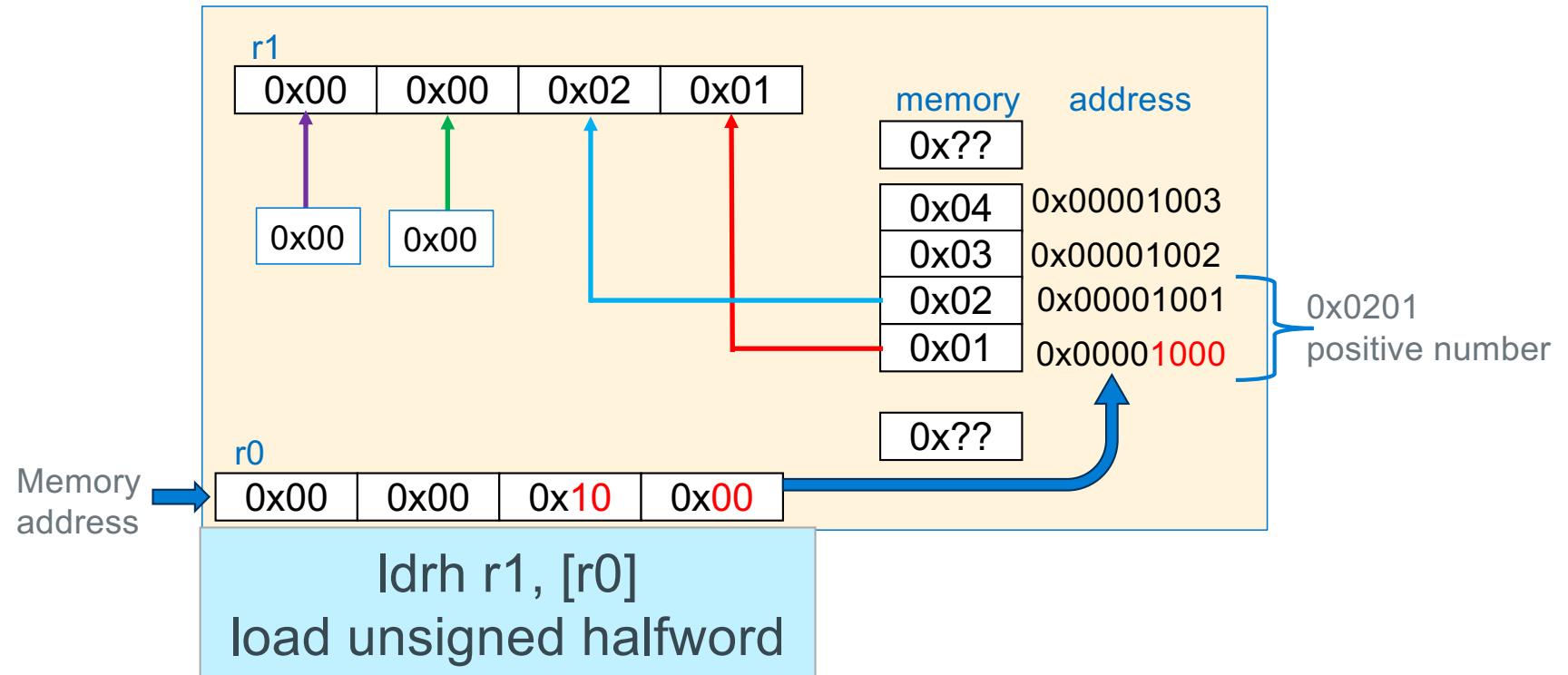
- Load and store have variations that move 8-bits, 16-bits and 32-bits
- Load into a register with less than 32-bits will set the upper bits not filled from memory differently depending on which variation of the load instruction is used
- Store will only select the lower 8-bit, lower 16-bits or all 32-bits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

Instruction	Meaning	Sign Extension	Memory Address Requirement
<code>ldr sb</code>	load signed byte	sign extension	none (any byte)
<code>ldr b</code>	load unsigned byte	zero fill (extension)	none (any byte)
<code>ldr sh</code>	load signed halfword	sign extension	halfword (2-byte aligned)
<code>ldr h</code>	load unsigned halfword	zero fill (extension)	halfword (2-byte aligned)
<code>ldr</code>	load word	---	word (4-byte aligned)
<code>strb</code>	store low byte (bits 0-7)	---	none (any byte)
<code>strh</code>	store halfword (bits 0-15)	---	halfword (2-byte aligned)
<code>str</code>	store word (bits 0-31)	---	word (4-byte aligned)

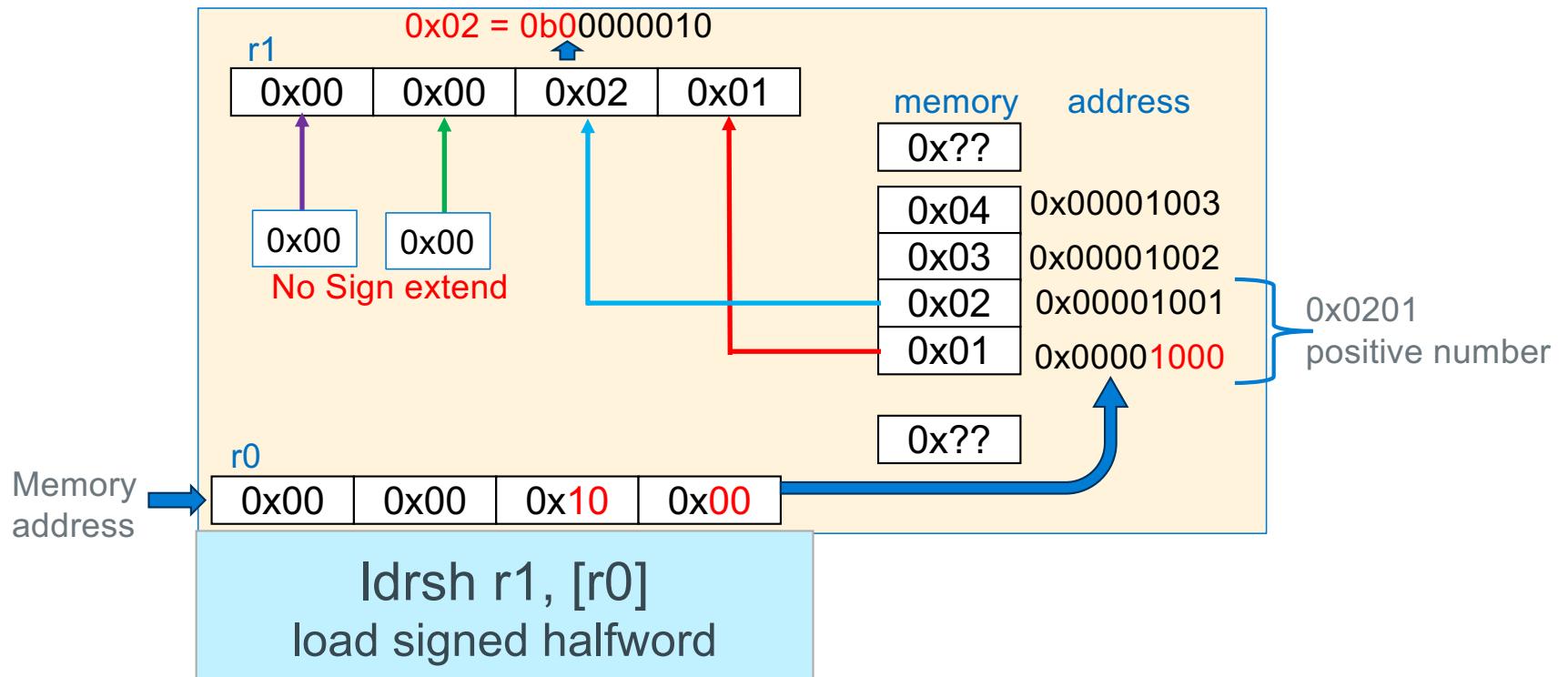
Loading 32-bit Registers From Memory, 32-bit



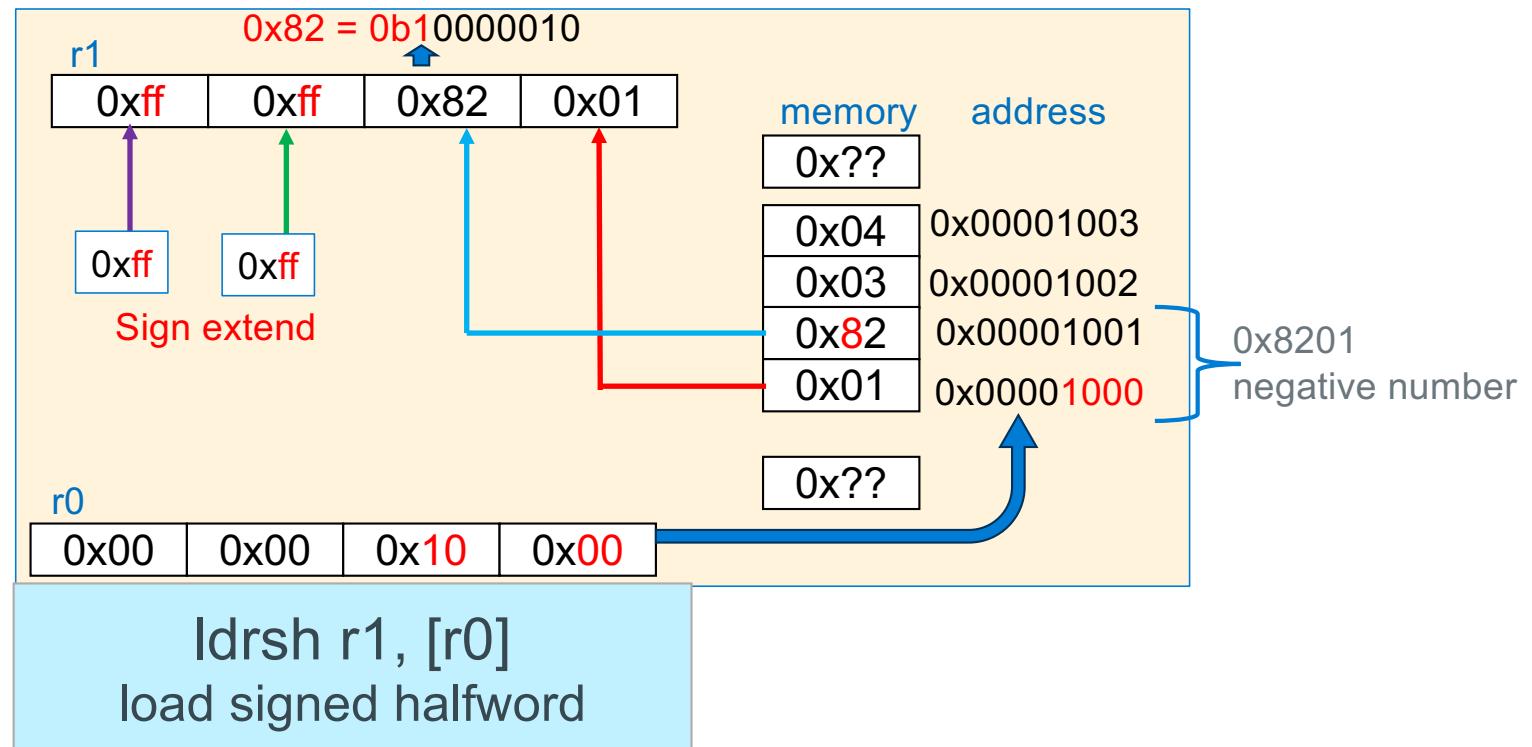
Loading 32-bit Registers From Memory, 16-bit



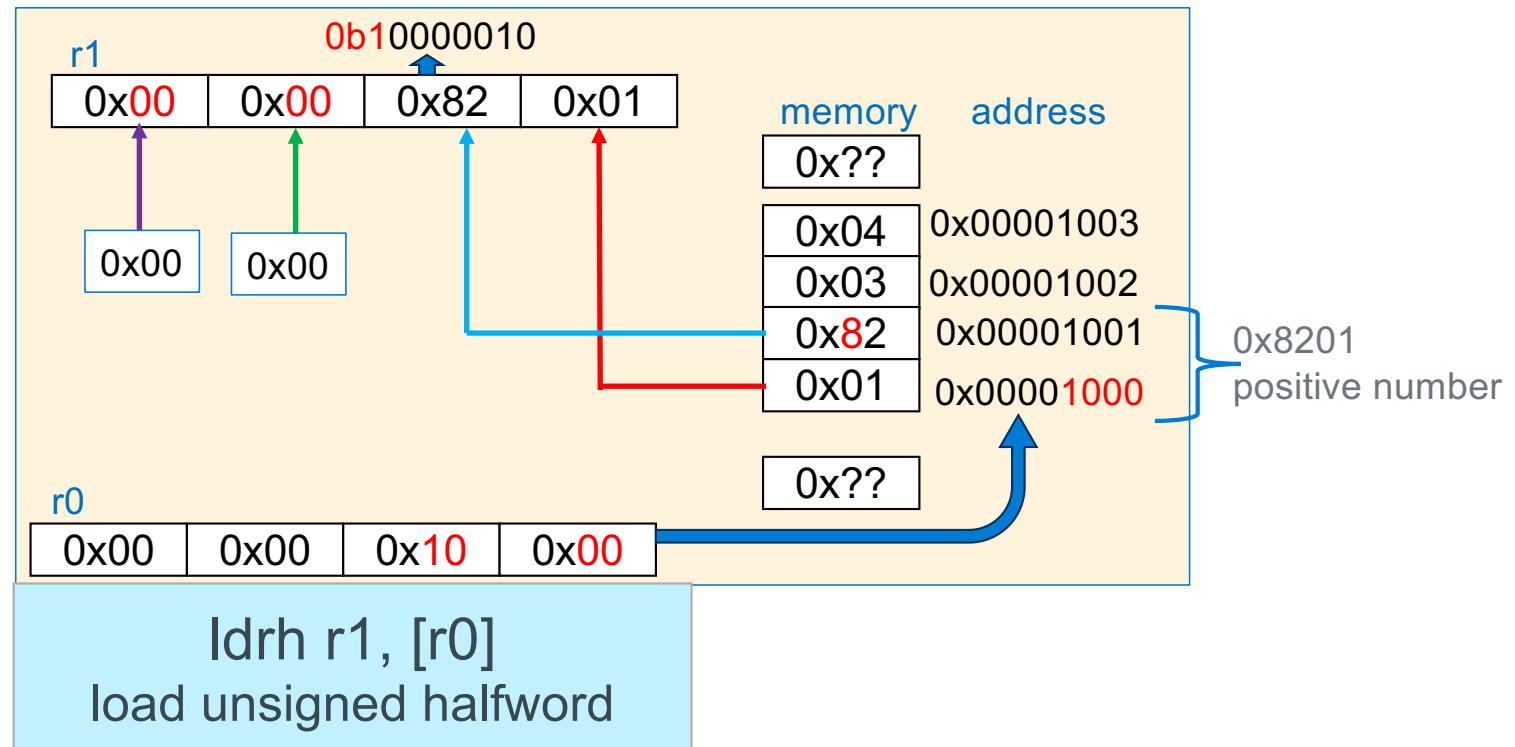
Loading 32-bit Registers From Memory, 16-bit



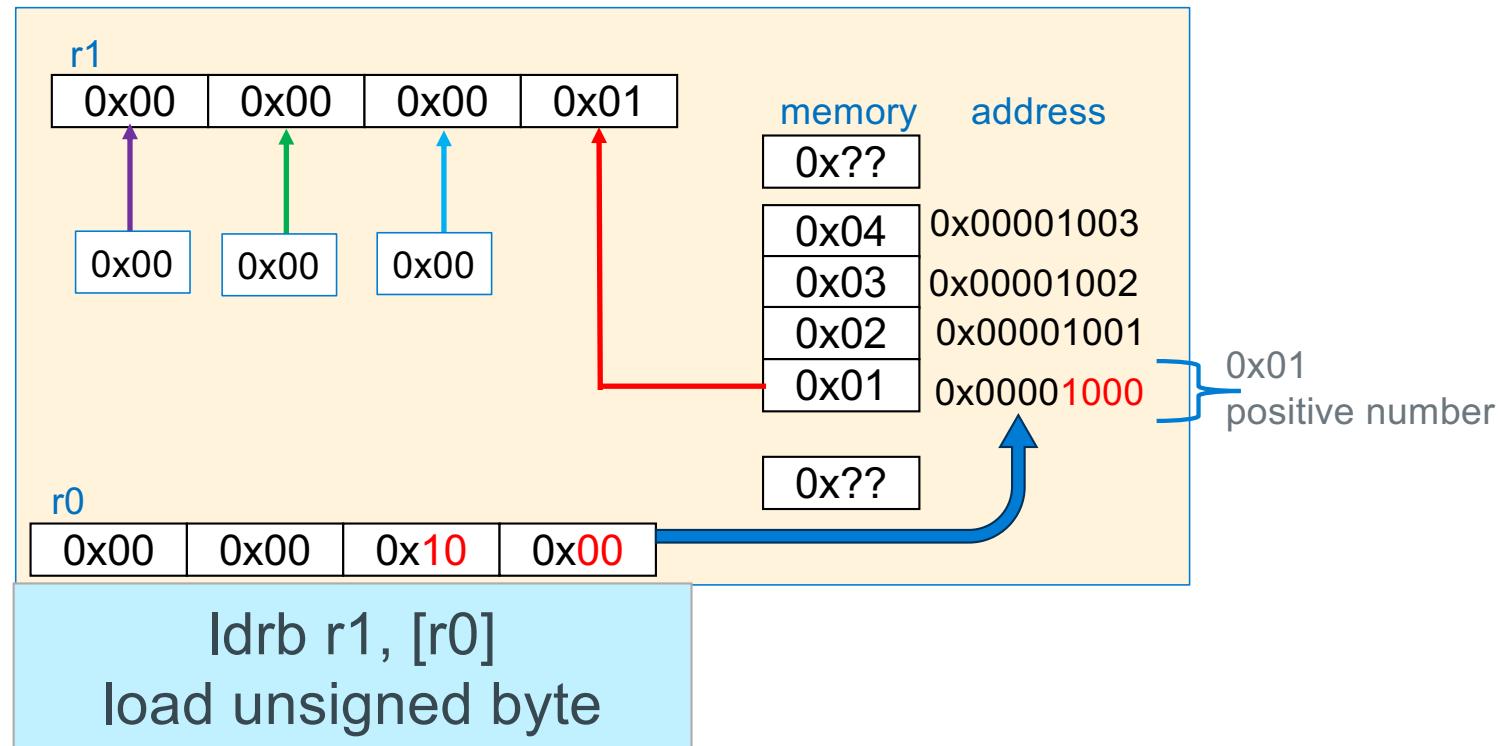
Loading 32-bit Registers From Memory, 16-bit Signed



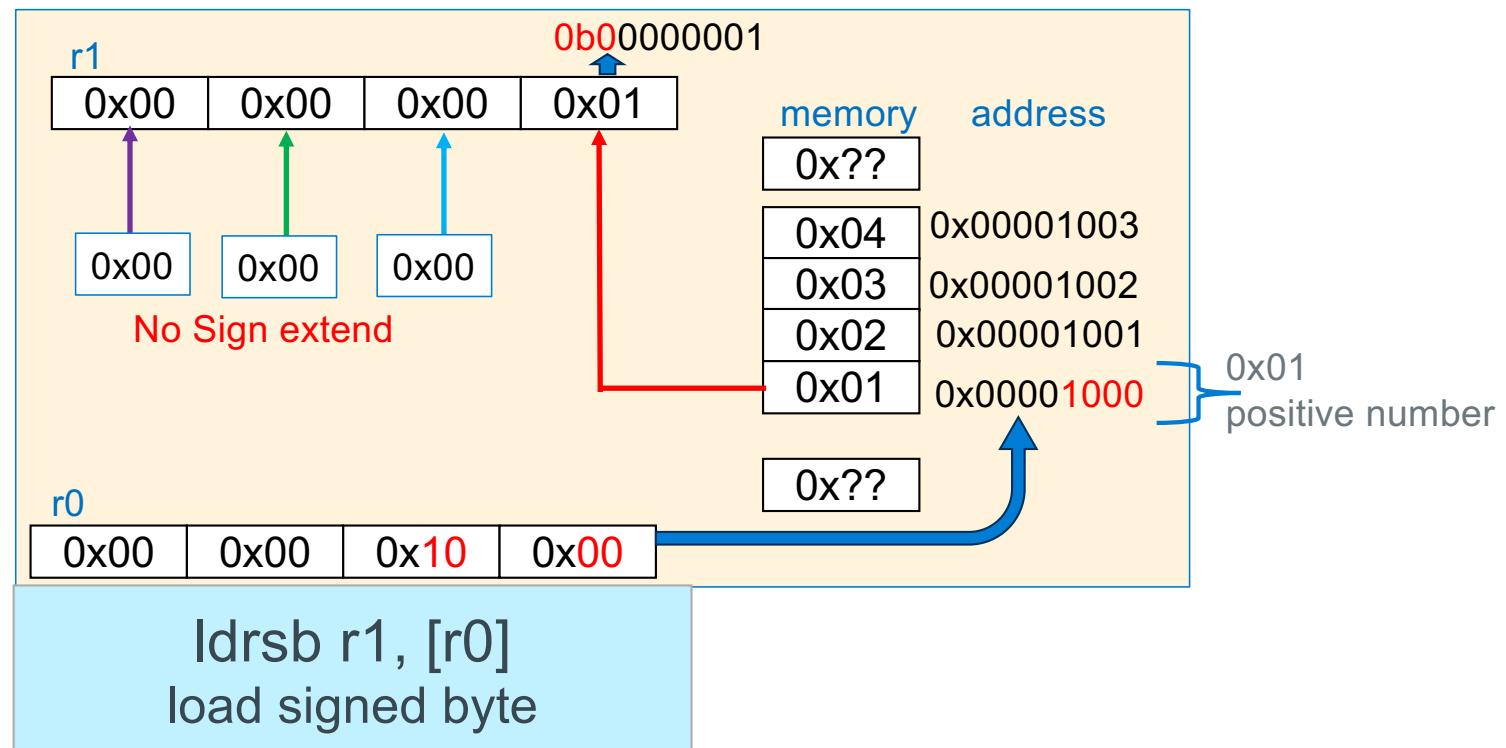
Loading 32-bit Registers From Memory, 16-bit Unsigned



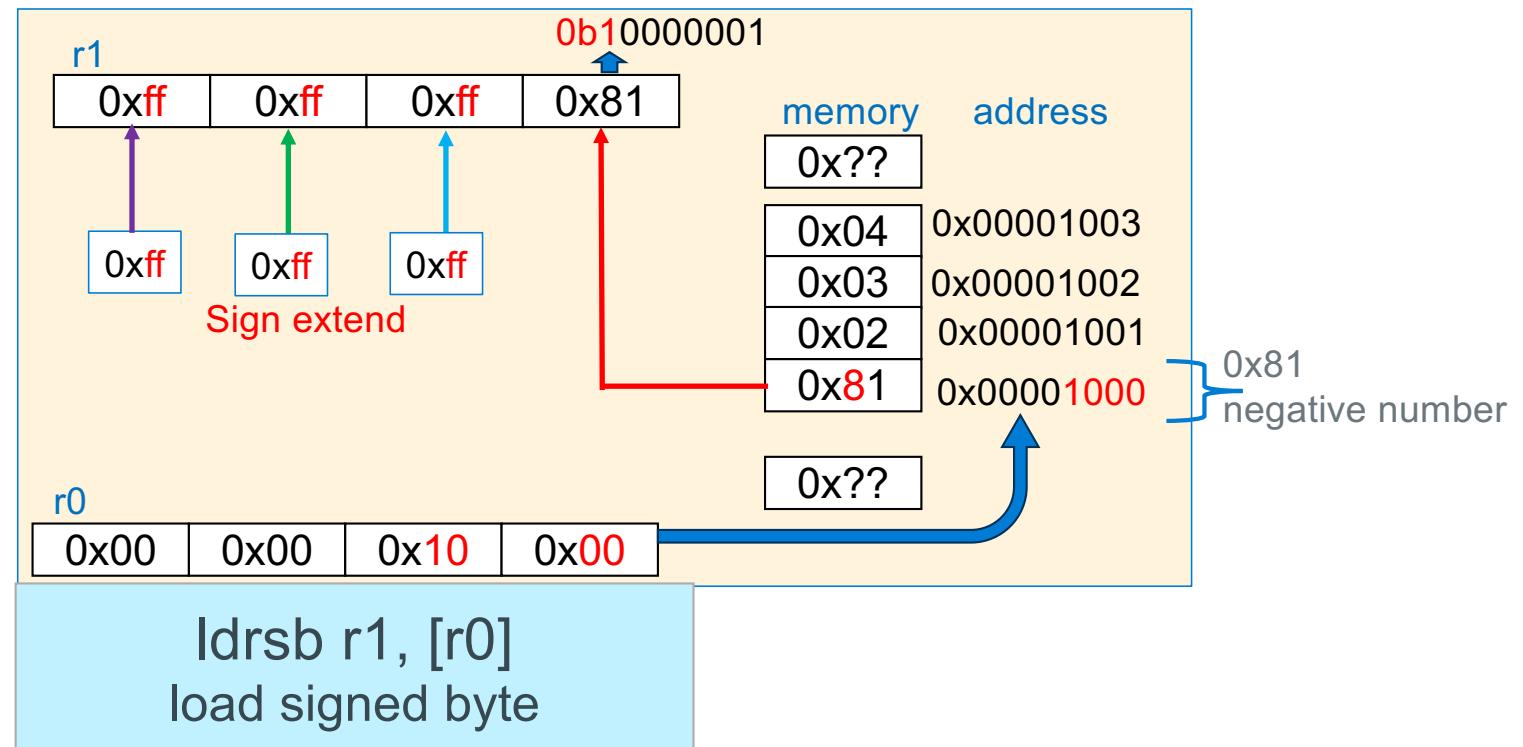
Loading 32-bit Registers From Memory, 8-bit



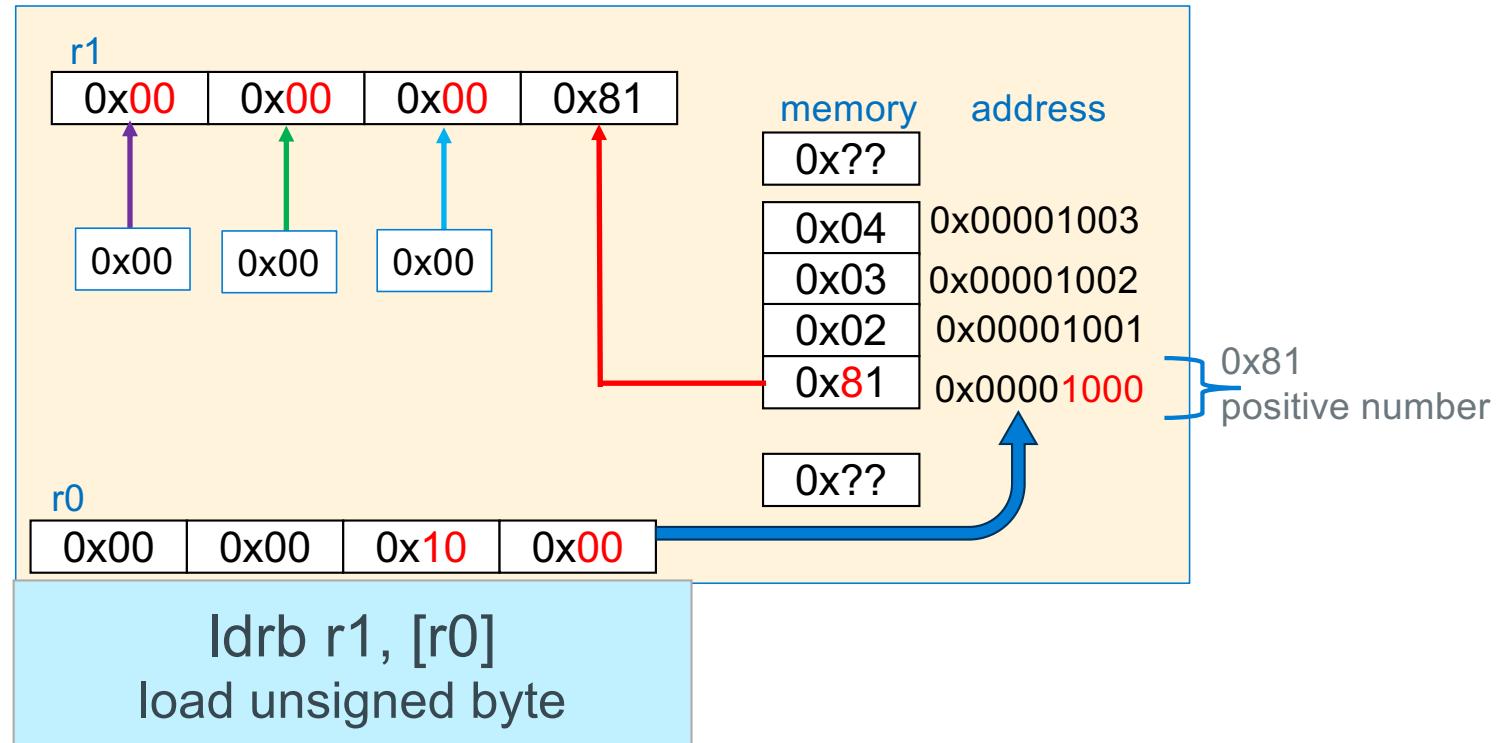
Loading 32-bit Registers From Memory, 8-bit



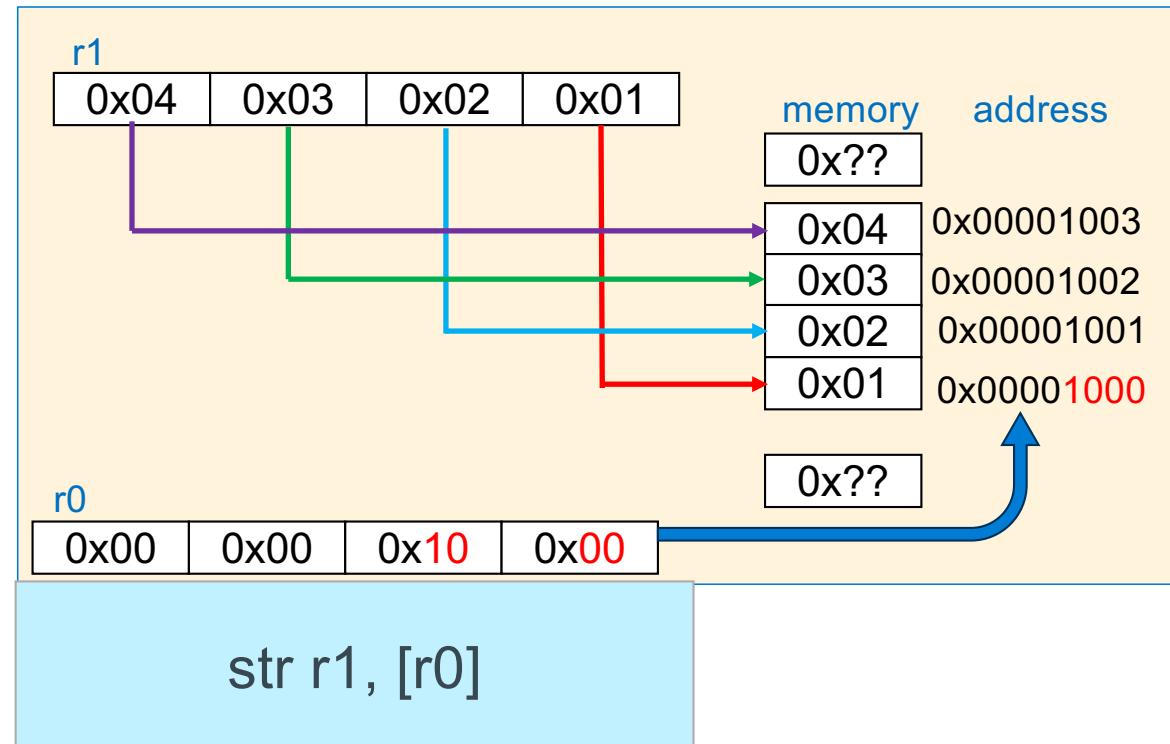
Loading 32-bit Registers From Memory, 8-bit Signed



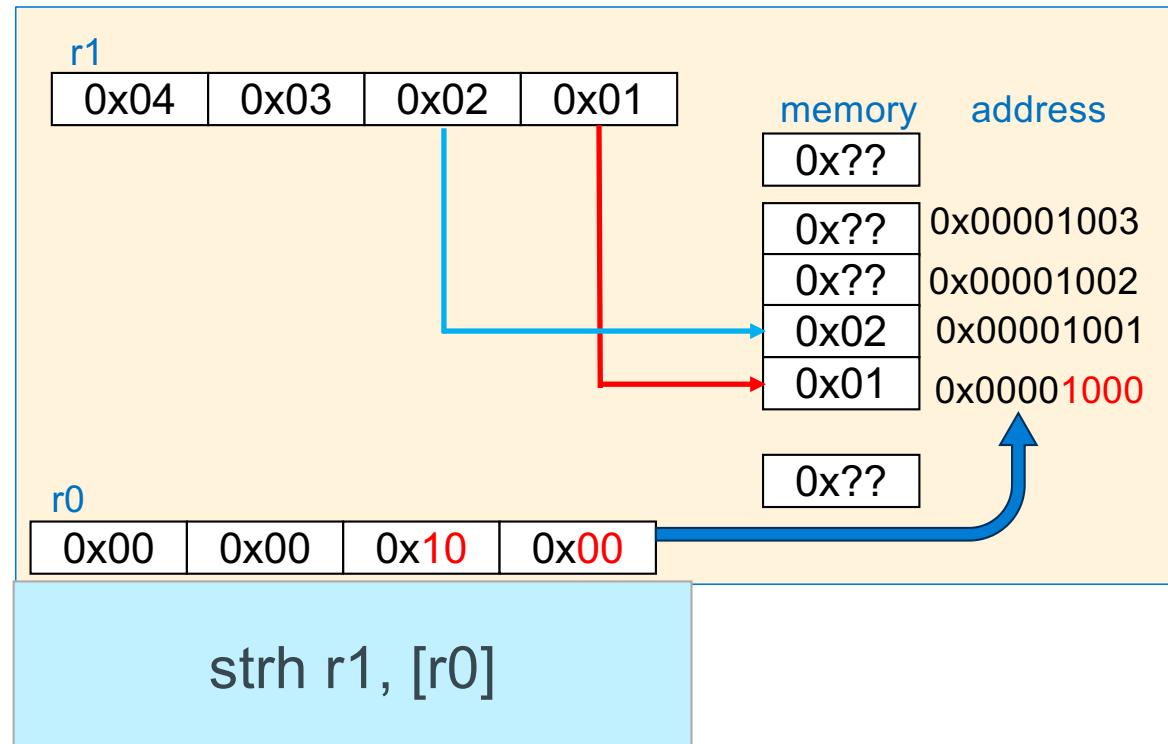
Loading 32-bit Registers From Memory, 8-bit Signed



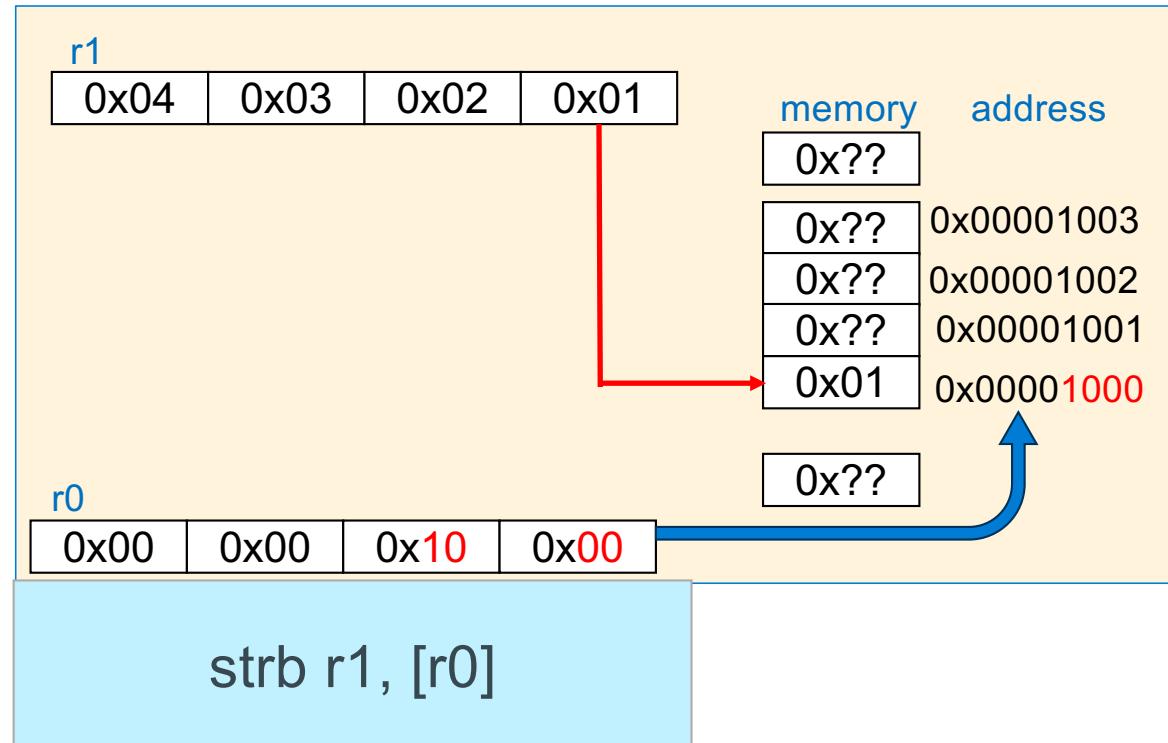
Storing 32-bit Registers To Memory, 32-bit



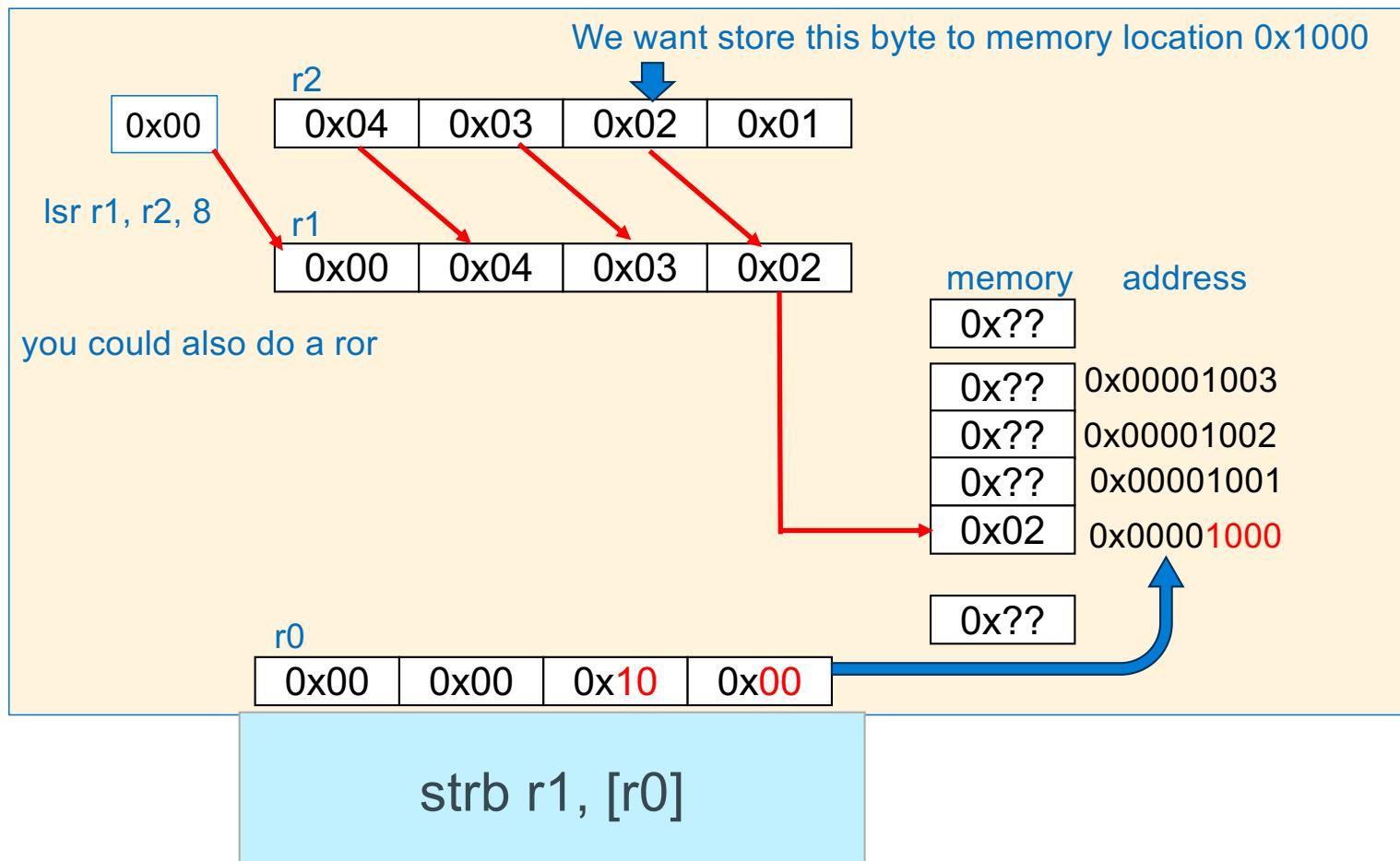
Storing 32-bit Registers To Memory, 16-bit



Storing 32-bit Registers To Memory, 8-bit

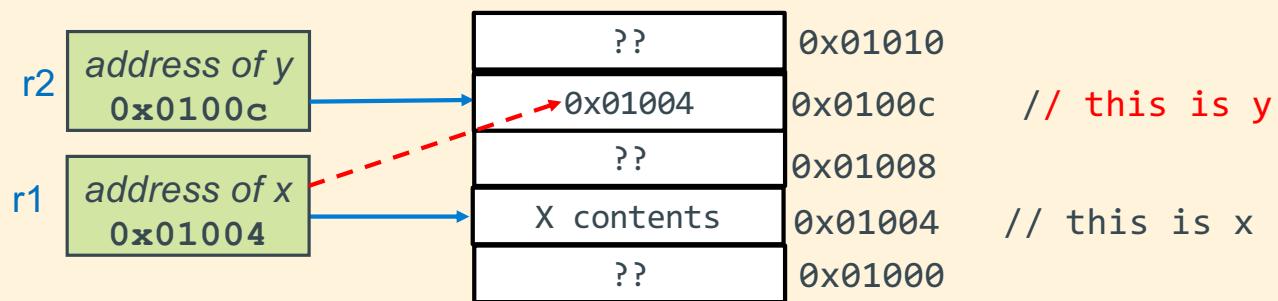


Storing 32-bit Registers To Memory, 8-bit – Storing different byte



ldr/str practice - 1

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



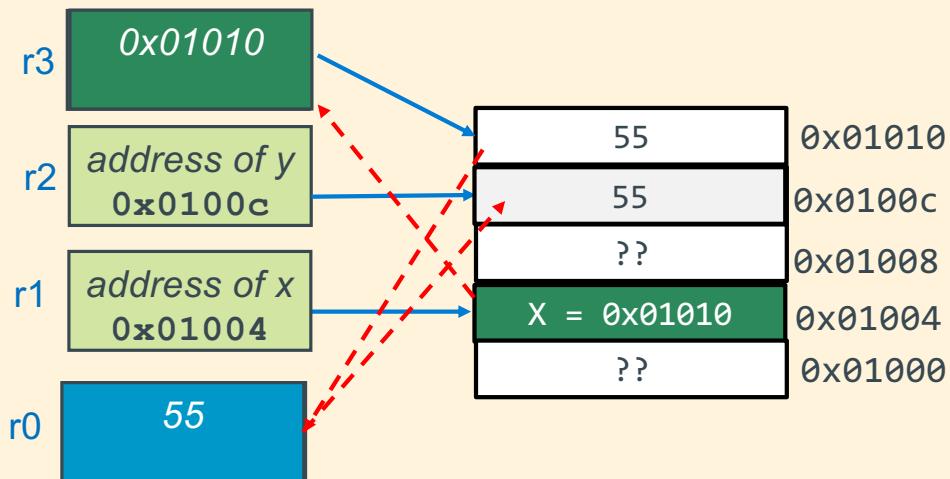
str r1, [r2] // y ← &x

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

using ldr/str: array copy

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

void icpy(int *, int *, int);

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

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

    return EXIT_SUCCESS;
}
```

```
void icpy(int *src, int *dst, int cnt)
{
    int *end = src + cnt;

    if (cnt <= 0)
        return;
    do {
        *dst++ = *src++;
    } while (src < end);
    return;
}
```

Base Register version

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global icpy
.type icpy, %function
.equ FP_OFFSET, 12

// r0 contains int *src
// r1 contains int *dst
// r2 contains int cnt
// r3 use as loop term pointer
// r4 use as temp

cpy:
    push {r4, r5, fp, lr}
    add fp, sp, FP_OFFSET
// see right -
    sub sp, fp, FP_OFFSET
    pop {r4, r5, fp, lr}
    bx lr
.size icpy, (. - icpy)
.end
```

```
        cmp r2, 0
        ble .Ldone
        pre loop guard

        lsl r2, r2, 2 //convert cnt to int size
        add r3, r0, r2 // loop term pointer

.Ldo:
        ldr r4, [r0] // load from src
        str r4, [r1] // store to dest

        add r0, r0, 4 // src++
        add r1, r1, 4 // dst++

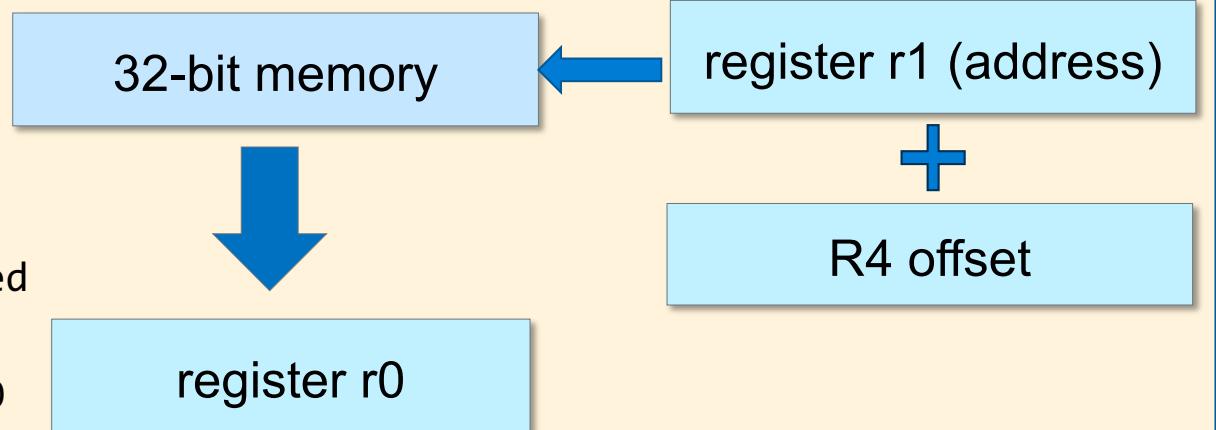
        cmp r0, r3 // src < term pointer?
        blt .Ldo
        loop guard

.Ldone:
```

Load/Store: Register Base Addressing + Register Offset

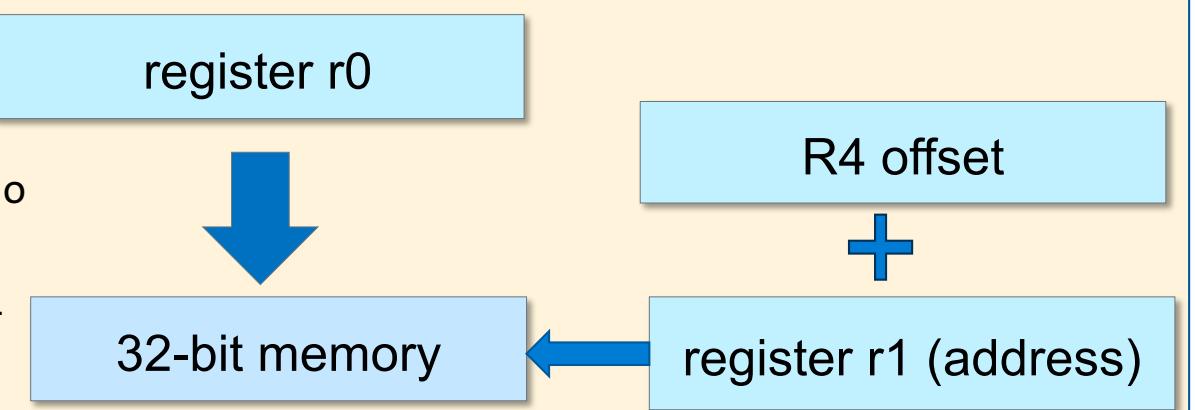
ldr r0, [r1, r4]

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

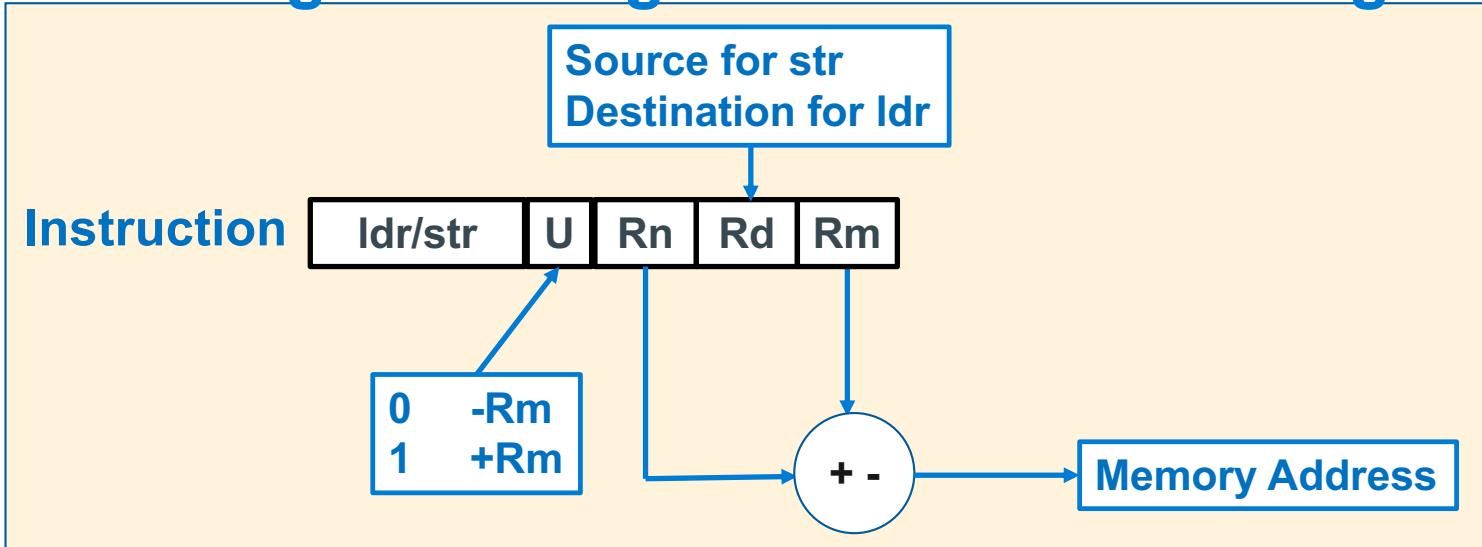


str r0, [r1, r4]

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



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**

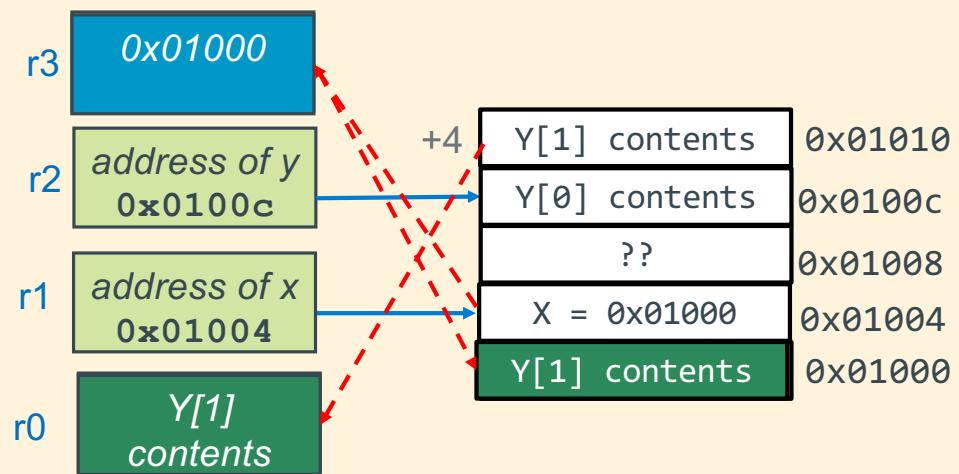
Syntax	Address	Examples
ldr/str Rd, [Rn +/- Rm]	Rn + or - Rm	ldr r0, [r5, r4] str r1, [r5, r4]

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]
```

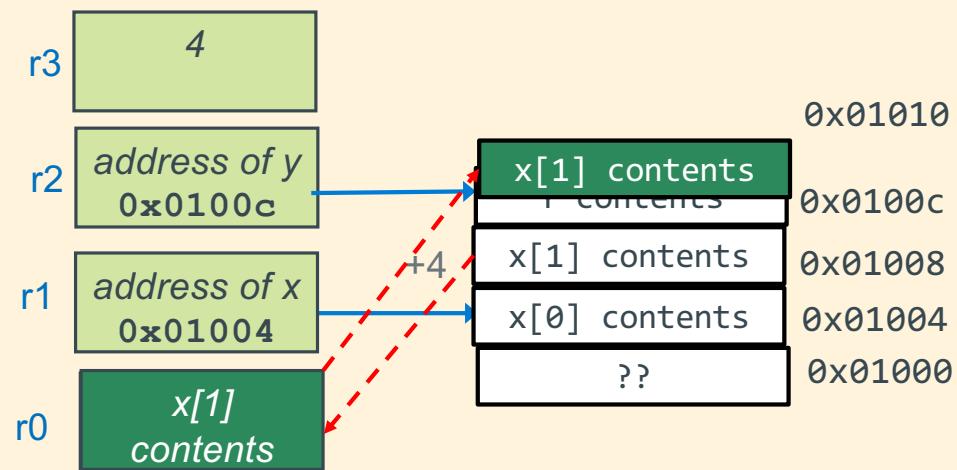
ldr/str practice - 4

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]

Base Register + Register Offset Version

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global icpy
.type icpy, %function
.equ FP_0FF, 12
// r0 contains int *src
// r1 contains int *dst
// r2 contains int cnt
// r3 use as loop counter
// r4 use as temp

cpy:
    push {r4, r5, fp, lr}
    add fp, sp, FP_0FF
// see right -
    sub sp, fp, FP_0FF
    pop {r4, r5, fp, lr}
    bx lr
.size icpy, (. - cpy)
.end
```

```
        cmp r2, 0
        ble .Ldone
        pre loop guard

        lsl r2, r2, 2      //convert cnt to int size
        mov r3, 0           // initialize counter

.Ldo:
        ldr r4, [r0, r3]   // load from src
        str r4, [r1, r3]   // store to dest
        add r3, r3, 4      // counter++
        cmp r3, r2         // count < r3
        blt .Ldo
        loop guard

.Ldone:
```

one increment
covers both arrays

Base Register + Register Offset With chars

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

    cpy(src, dst, SZ);
    printf("%s\n", dst);
    return EXIT_SUCCESS;
}
```

```
cmp    r2, 0
ble   .Ldone

mov    r3, 0          // initialize counter
.Ldo:
    ldrb   r4, [r0, r3]  // load from src
    strb   r4, [r1, r3]  // store to dest
    add    r3, r3, 1      // counter++
    cmp    r3, r2          // count < r3
    blt   .Ldo

.Ldone:
```

Reference: Addressing Mode Summary for use in CSE30

index Type	Example	Description
Pre-index immediate	ldr r1, [r0]	$r1 \leftarrow \text{memory}[r0]$ r0 is unchanged
Pre-index immediate	ldr r1, [r0, 4]	$r1 \leftarrow \text{memory}[r0 + 4]$ r0 is unchanged
Pre-index immediate	str r1, [r0]	$\text{memory}[r0] \leftarrow r1$ r0 is unchanged
Pre-index immediate	str r1, [r0, 4]	$\text{memory}[r0 + 4] \leftarrow r1$ r0 is unchanged
Pre-index register	ldr r1, [r0, +-r2]	$r1 \leftarrow \text{memory}[r0 +- r2]$ r0 is unchanged
Pre-index register	str r1, [r0, +-r2]	$\text{memory}[r0 +- r2] \leftarrow r1$ r0 is unchanged

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;
}
```

Base Register + Offset register

```
.arch armv6
.arm
.fpu vfp
.syntax unified
.text
.global count
.type count, %function
.equ FP_0FF, 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_0FF
// see right ->
    sub sp, fp, FP_0FF
    pop {r4, r5, fp, lr}
    bx lr
.size count, (. - count)
.end
```

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

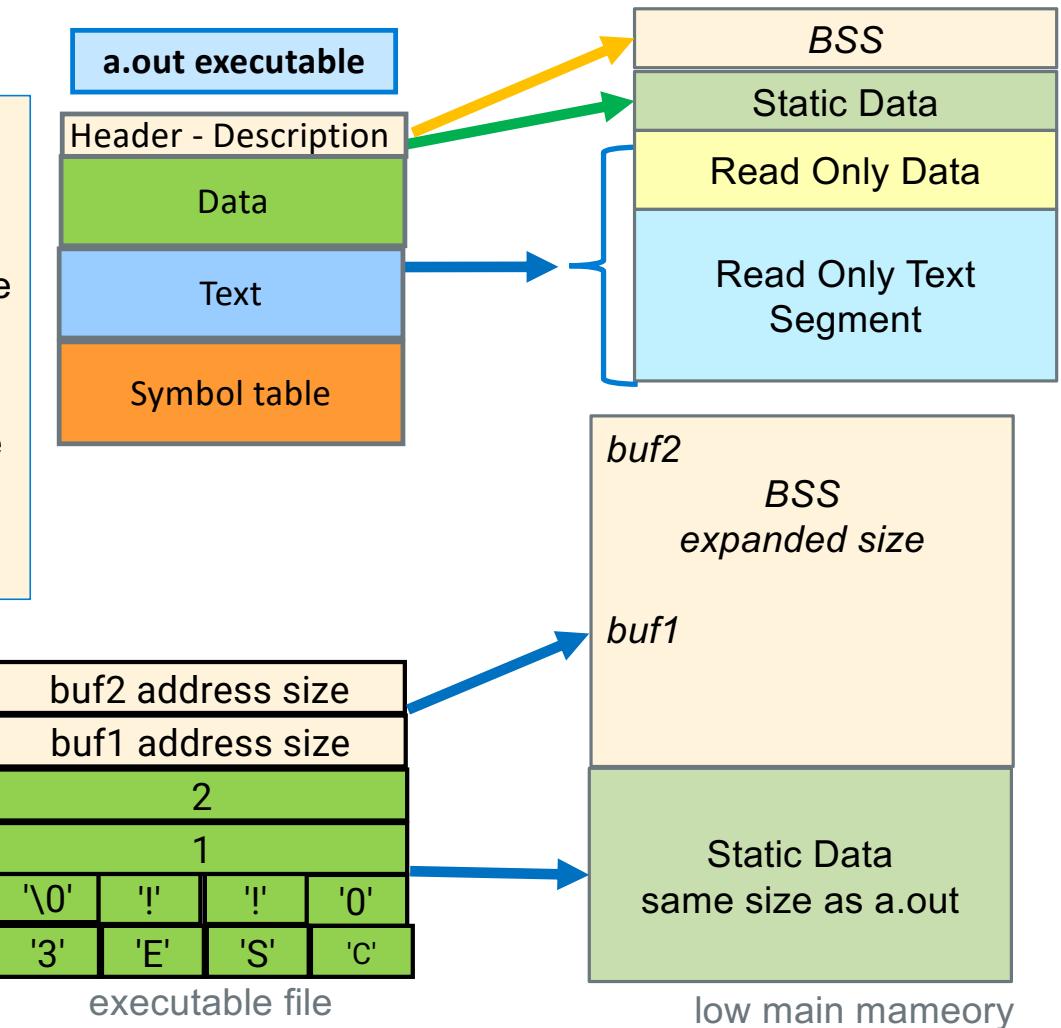
```
count:
    push {r4, r5, fp, lr}
    add fp, sp, FP_0FF
    mov r2, 0
    cmp r1, 0
    ble .Ldone
    mov r3, 0
.Lfor:
    cmp r3, r1
    bge .Ldone
    ldrb r4, [r0, r3]
    cmp r4, 'A'
    blt .Lendif
    cmp r4, 'Z'
    bgt .Lendif
    add r2, r2, 1
.Lendif:
    add r3, r3, 1
    b .Lfor
.Ldone:
    mov r0, r2
```

What is the conceptual difference between .bss and .data?

- All static variables that do not specify an initial value default to an initial value of 0 and are placed in .bss segment
- To save file system space in the executable file (the a.out file) the assembler collapses these .bss variables to a location and size "table"
- .data segment variables use the same space in the executable file as they have in memory
- .section .rodata is handled the same as .data

```
// these are .bss variables
int buf1[4096];
int buf2[4096]; just big enough for address, size

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



Variable Alignment In .data, .bss and .section .rodata

Use `.align` directive to force the assembler to align the address of the next variable defined after the `.align`

	char	1	any address
	short	2 bytes	addresses that end in 0b0
integer		4 bytes	addresses that end in 0b00

SIZE Alignment Requirements	Starting Address must end in	Align Directive
8-bit char - 1 byte	0b..0 or 0b..1	
16-bit int - 2 bytes	0b..0	<code>.align 1</code>
32-bit int - 4 bytes pointers, all arrays	0b..00	<code>.align 2</code>



Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	.align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		char chx = 'A' char string[] = {'A', 'B', 'C', 0};	chx: .byte 'A' string: .byte 'A', 'B', 0x42, 0
16-bit int (2 bytes)	.short	.align 1	short length = 0x55aa;	length: .short 0x55aa
32-bit int (4 bytes)	.word .long	.align 2	int dist = 5; int *distptr = &dist; unsigned int mask = 0xaa55; int array[] = {12, ~0x1, 0xCD, -1};	dist: .word 5 distptr: .word dist mask: .word 0xaa55 array: .word 12, ~0x1, 0xCD, -3
string with '\0'	.string		char class[] = "cse30";	class: .string "cse30"

Rule: Place the .align above the variable

```
.align 1
```

len: .short 0x55aa

Rule: use .align 2 before every array regardless of type

Rule: place variables with explicit initialized values in a .data segment

Rule: place variables with no explicit initiali value (default to 0) in .bss segment

Rule: place string literals in .section .rodata and use a local label (.Llabel:)

Loading Static variables into a register

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

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

- Tell the assembler load the contents into a register

```
ldr R0, [Rd] // Rd = address
```

Example to the right: y = x;

load a static **memory** variable

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

store to a static **memory** variable

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

```
.bss
y: .space 4

.data
x: .word 200

.text
// function header
main:

// load the address, then contents
// using r2

ldr r2, =x      // int *r2 = &x
ldr r2, [r2]    // r2 = *r2;

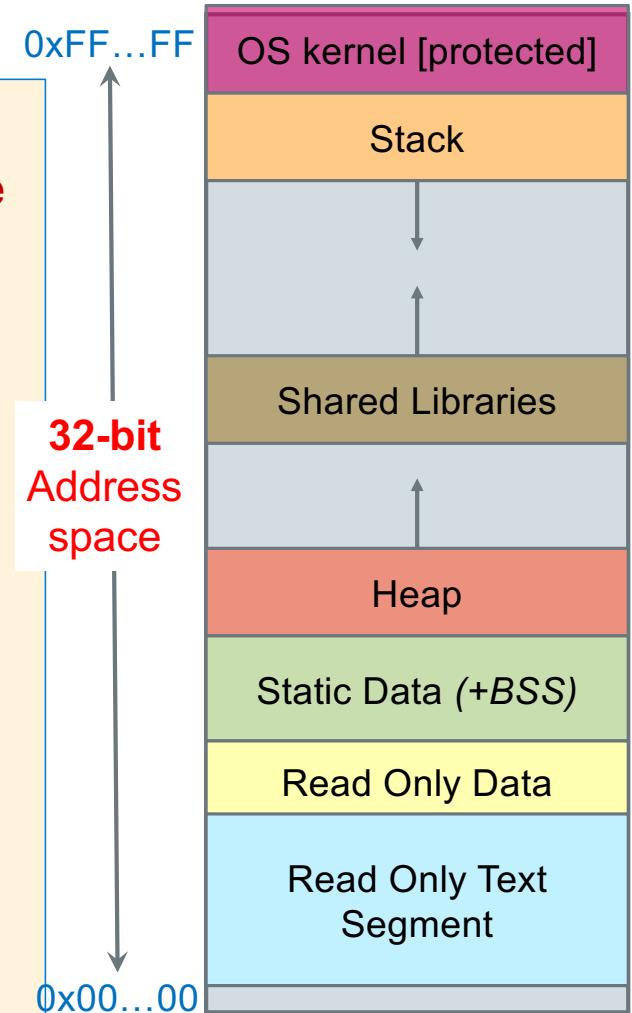
// &x was only needed once above
// Note: r2 was a pointer then an int
// no "type" checking in assembly!

// store the contents of r2

ldr r1, =y      // int *r1 = &y
str r2, [r1]    // *r1 = r2
```

Stack Segment: Support of Functions

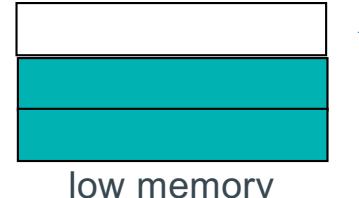
- The stack consists of a series of "*stack frames*" or "*activation frames*", one is **created** each time a function is called **at runtime**
- Each **frame** represents a function that is currently being **executed** and **has not yet completed** (why activation frame)
- A function's stack "frame" goes away when the function returns
- Specifically, a **new stack frame** is
 - allocated (**pushed** on the stack) for each function call (**contents are not implicitly zeroed**)
 - deallocated (**popped** from the stack) on function return
- **Stack frame** contains:
 - Local variables, parameters of function called
 - Where to return to which caller when the function completes (the return address)



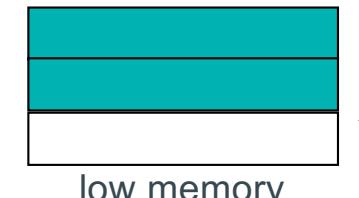
Stack types

- A Stack Implements a **last-in first-out** (LIFO) protocol
- Each time a **function is called**, a **stack frame is activated**
 - space is allocated by moving the stack pointer
 - push adds space, pop removes space
- Stack growth direction
 - **Ascending stack**: grows from low memory towards high memory ([adding to the sp to allocate memory](#))
 - **Descending stack**: grows from high memory towards low memory ([subtracting from the sp to allocate memory](#))
- Full versus empty stacks
 - **Empty stack**: **stack pointer** (sp) points at the **next word address** after the last item pushed on the stack
 - **Full stack**: **stack pointer** (sp) points at the **last item pushed on the stack**
- ARM on Linux uses a [full descending stack](#)

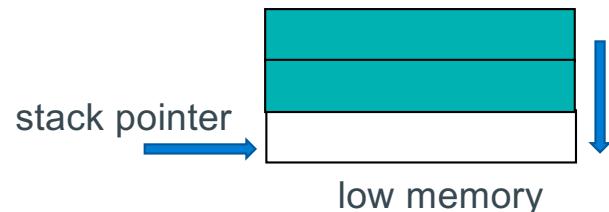
Ascending stack high memory



Descending stack high memory



Empty descending stack high memory



Full descending stack high memory



Ghost of Stack Frames Past.....

same stack frame
variable layout

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

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

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

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

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

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

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

Function Calls

Branch with Link (function call) instruction

bl **label** **bl** **imm24**

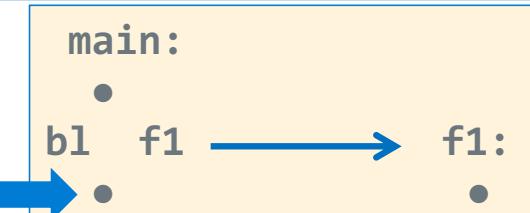
- Function call to the instruction with the address **label** (no local labels for functions)
 - **imm24** number of instructions from pc+8 (24-bits)
 - **label** **any function label** in the current file, **any function label that is defined as .global in any file that it is linked to**, **any C function that is not static**

Branch with Link Indirect (function call) instruction

blx **Rm** **blx** **Rm**

- Function call to the instruction whose address is stored in Rm (Rm is a function pointer)
- **bl and blx both save the address of the instruction immediately following the bl or blx instruction in register lr** (link register is also known as r14)
- **The contents of the link register is the return address in the calling function**

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



Function Call Return

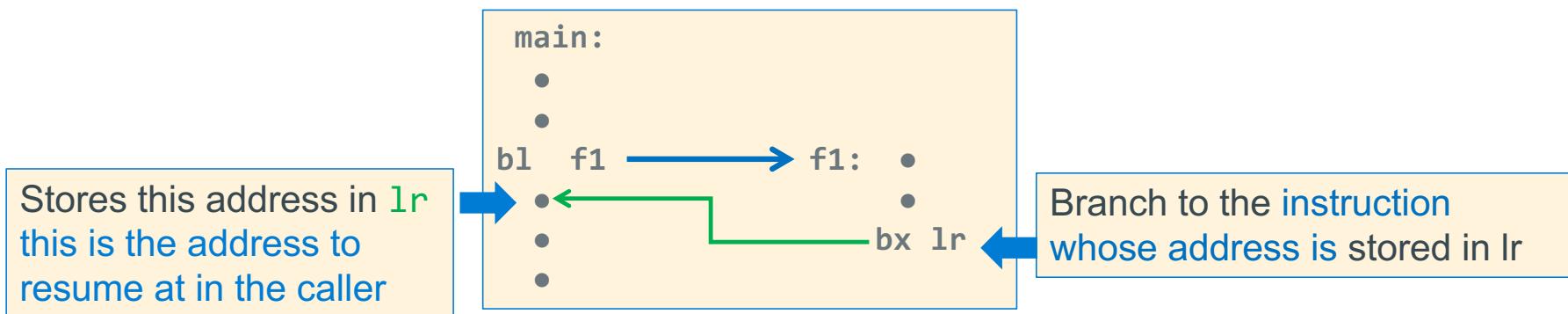
Branch & exchange (**function return**) instruction

`bx lr`

<code>bx</code>	<code>Rn</code>
-----------------	-----------------

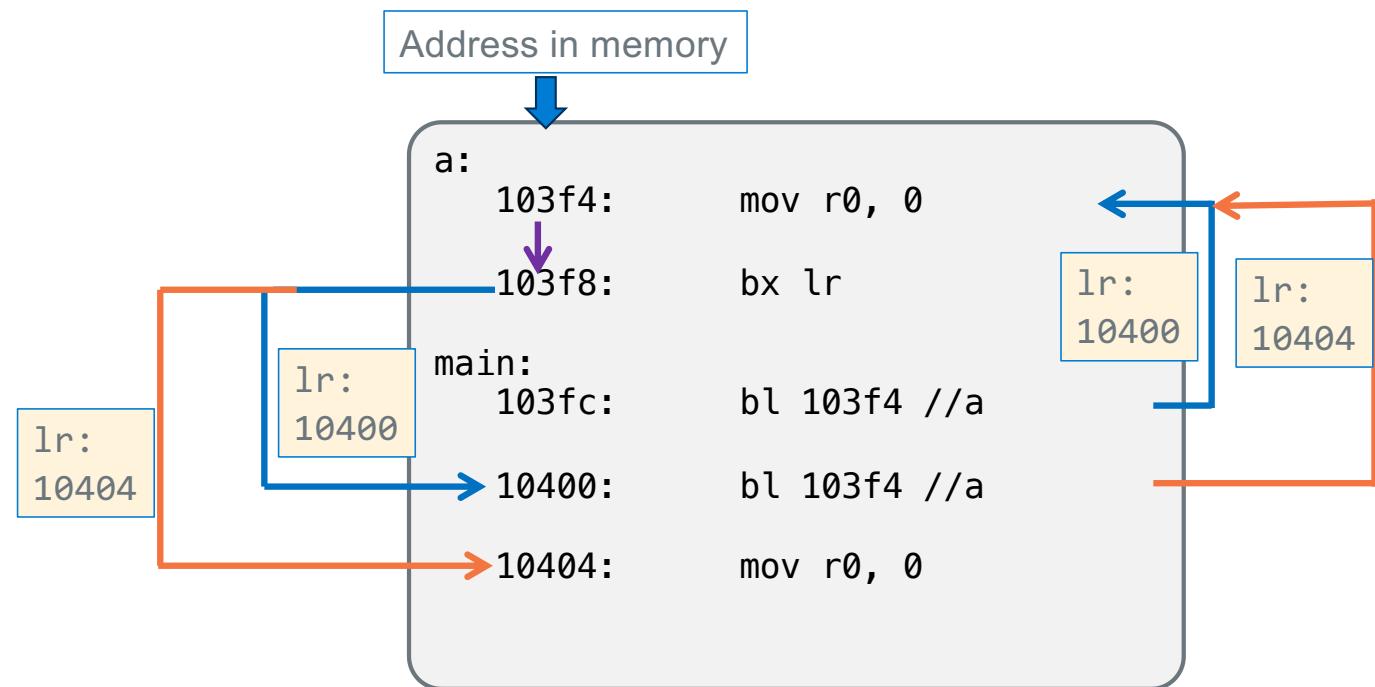
 // we will always use lr

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



Understanding bl and bx - 1

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



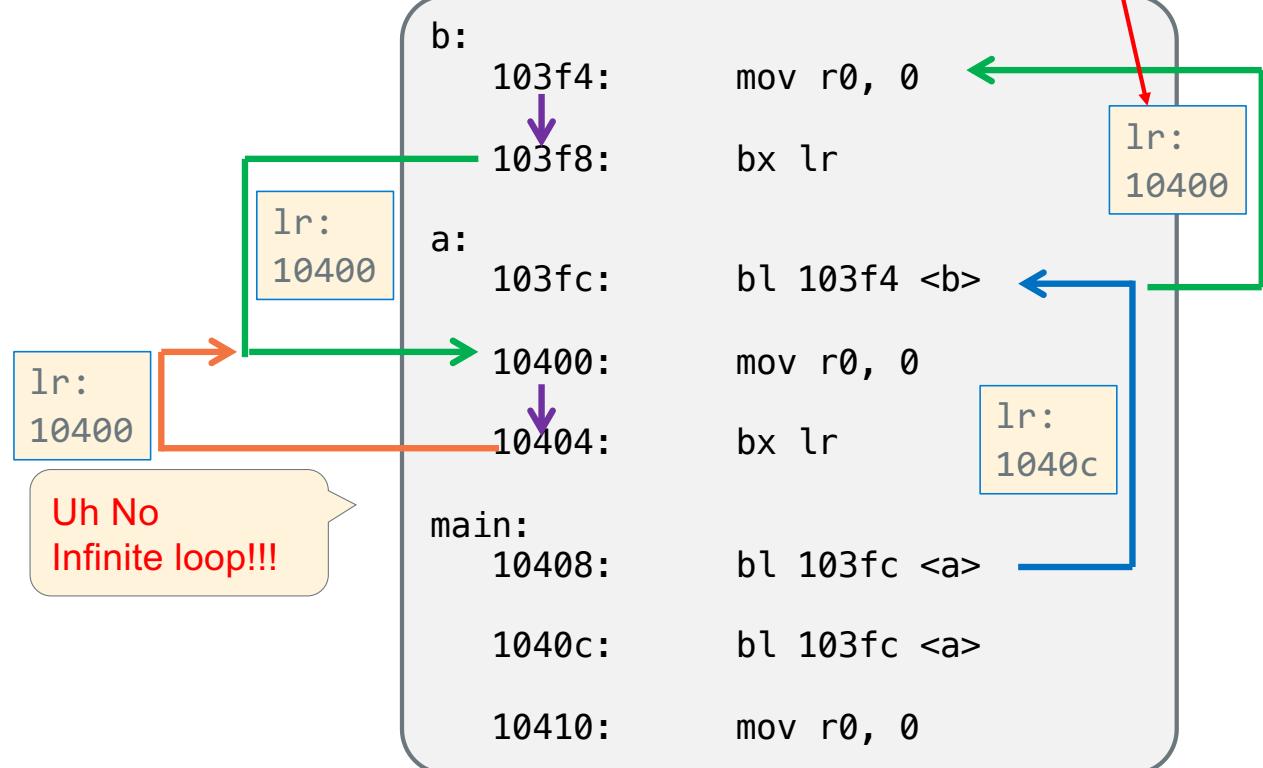
But there is a problem we must address here – next slide

Understanding bl and bx - 2

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

We need to preserve the lr!

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



Understanding bl and blx - 3

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

int (*func)() = a;

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

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

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

.text
.global a
.type a, %function
.equ FP_OFF, 4

a:
    mov r0, 0
    bx lr
    .size a, (. - a)

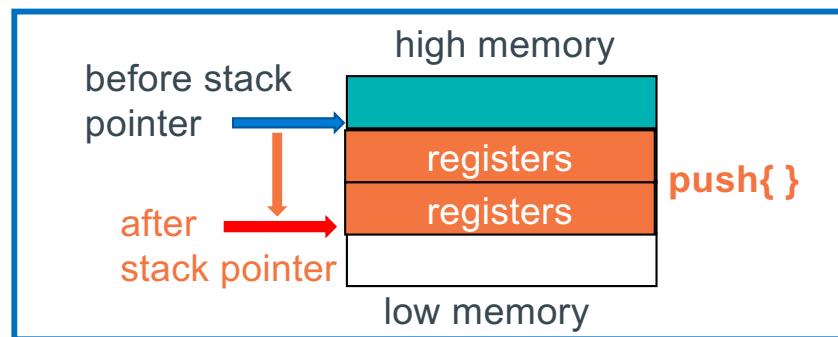
.global main
.type main, %function
.equ FP_OFF, 4

main:
    ldr r4, =func      // load address of func in r4
    ldr r4, [r4]        // load contents of func in r4
    blx r4             // we lose the lr for main!
    // not shown
    bx lr              // infinite loop!
```

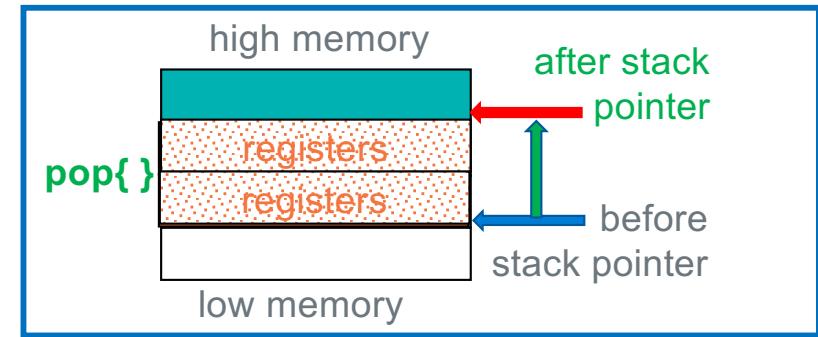
Preserving and Restoring Registers on the stack - 1

<i>Operation</i>	<i>Pseudo Instruction</i>	<i>Operation</i>
Push registers Function Entry	push {reg list}	$sp = sp - 4 \times \#registers$ Copy registers to mem[sp]
Pop registers Function Exit	pop {reg list}	Copy mem[sp] to registers, $sp = sp + 4 \times \#registers$

push (multiple register str to memory operation)



push (multiple register ldr from memory operation)



Preserving and Restoring Registers on the Stack - 2

<i>Operation</i>	<i>Pseudo Instruction</i>	<i>Operation</i>
Push registers Function Entry	<code>push {reg list}</code>	$sp = sp - 4 \times \#registers$ Copy registers to $mem[sp]$
Pop registers Function Exit	<code>pop {reg list}</code>	Copy $mem[sp]$ to registers, $sp = sp + 4 \times \#registers$

- `{reg list}` is a **list of registers in numerically increasing order, left to right**

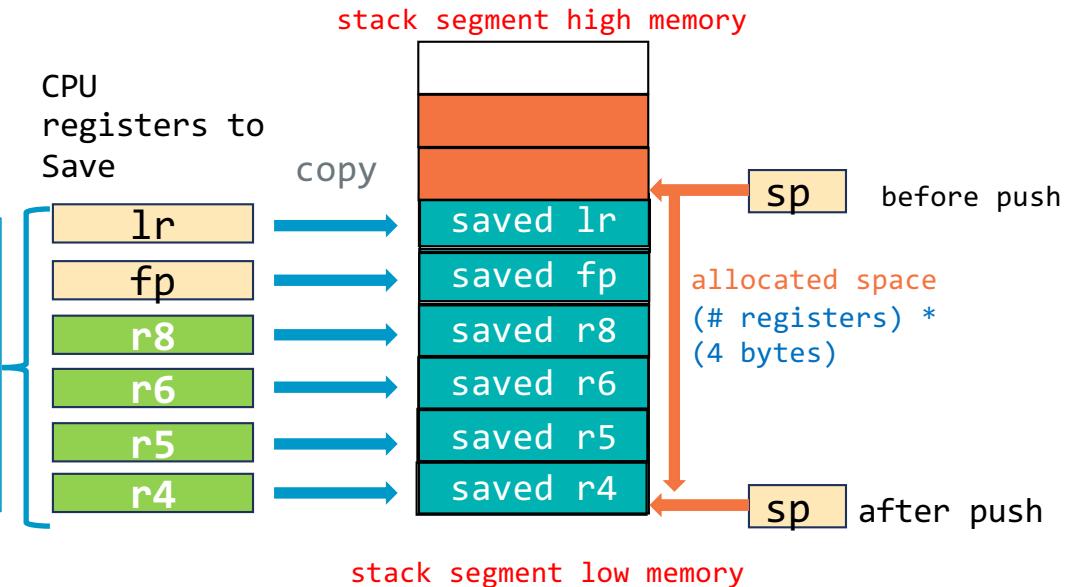
`push {r4-r10, fp, lr} // fp is r11, lr is r14`
- Registers **cannot be**:
 1. duplicated in the list
 2. listed out of increasing numeric order (left to right)
- Register ranges can be specified `{r4, r5, r8-r10, fp, lr}`
- **Never!** push/pop `r12, r13, or r15`
 - the top two registers on the stack must always be `fp, lr` // ARM function spec – later slides

push: Multiple Register Save to the stack

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

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

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

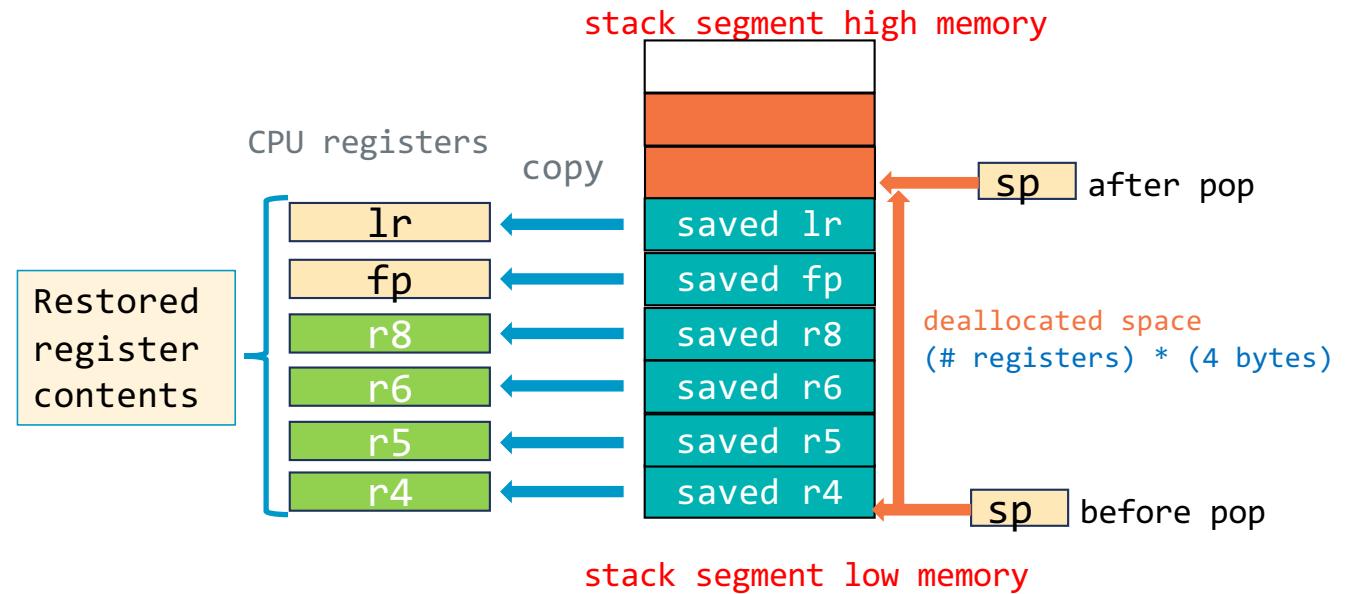


- **push** copies the contents of the **{reg list}** to stack segment memory
- **push** subtracts (# of registers saved) * (4 bytes) from the **sp** to **allocate** space on the stack
 - $sp = sp - (\# \text{ registers_saved} * 4)$
- **this must always be true:** **sp % 8 == 0**

pop: Multiple Register Restore from the stack

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

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



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

Registers: Rules For Use

<i>Register</i>	<i>Function Call Use</i>	<i>Function Body Use</i>	<i>Save before use Restore before return</i>
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No
r4-r10	preserved registers	contents preserved across function calls	Yes
r11 / fp	stack frame pointer	Use to locate variables on the stack	Yes
r12 / ip	may used by assembler with large text file	can be used as a scratch if really needed	No
r13 / sp	stack pointer	stack space allocation	Yes
r14 / lr	link register	contains return address for function calls	Yes
r15	Do not use	Do not use	No

Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use	Function Body Use	Save before use Restore before return
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No

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

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

- Each parameter and return value is limited to data that can fit in 4 bytes or less
- Calling function:
 - copy up to the first four parameters into these four registers before calling a function
 - MUST 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
- Called function:
 - you receive the first four parameters in these four registers (`r0 – r3`)

Return Value and Passing Parameters to Functions

(Four parameters or less)

Register	Function Call Use	Function Body Use	Save before use Restore before return
r0	arg1 and return value	scratch registers	No
r1-r3	arg2 to arg4	scratch registers	No

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

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

- For parameters, whose size is larger than 4 bytes, pass a pointer to the parameter (we will cover this later)
- One arg value per register!** – NO arrays across multiple registers
 - chars, shorts and ints are directly stored
 - Structs (not always), and arrays (always) are passed via a pointer
 - Pointers** passed as output parameters contain an address **that points at** the stack, BSS, data, or heap

What it means to be a Temporary/argument register

```
int a(void)
{
    // not shown
}
int main(void)
{
    int r0 = 0;
    int r1 = 1;
    int r2 = 2;
    int r3 = 3;
    r0 = a();
    // in C r1 and r3 would have the same values
    // after the call
```

```
// main()
// code not shown
mov r0, 0
mov r1, 1
mov r2, 2
mov r3, 3
bl a
// r0 = return value
// r1-r3 values are unknown as a() has right to change them as it wants
```

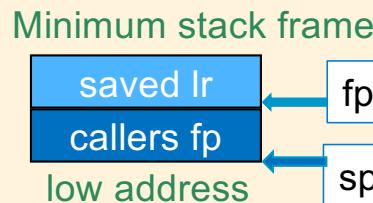
Preserved Registers

Register	Function Call Use	Function Body Use	Save before use Restore before return
r4-r10	preserved registers	contents preserved across function calls	Yes
r11/fp	stack frame pointer	Use to locate variables on the stack	Yes
r13/sp	stack pointer	stack space allocation	Yes
r14/lr	link register	contains return address for function calls	Yes

- 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)
- You use a preserved register when a function makes calls another function and you have:
 1. Local variables allocated to be in registers
 2. Parameters passed to you (in r0-r3) that you need to continue to use after calling another function

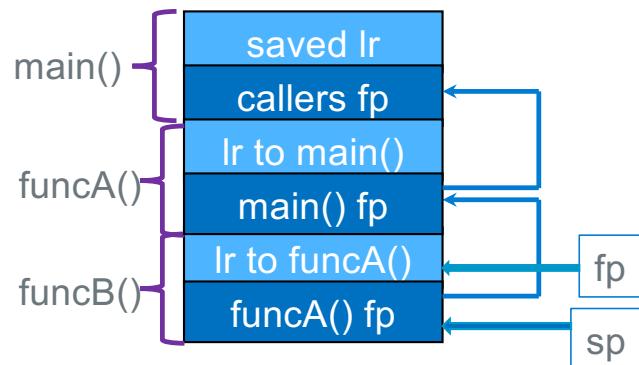
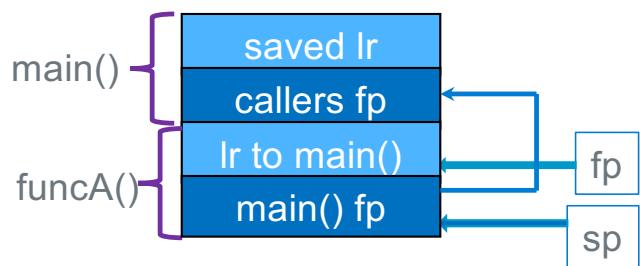
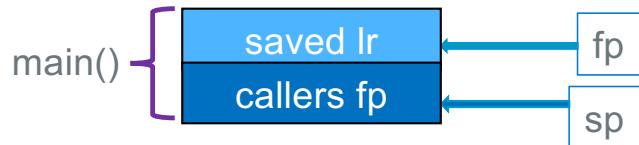
Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

- Minimal frame: allocating at function entry: `push {fp, lr}`

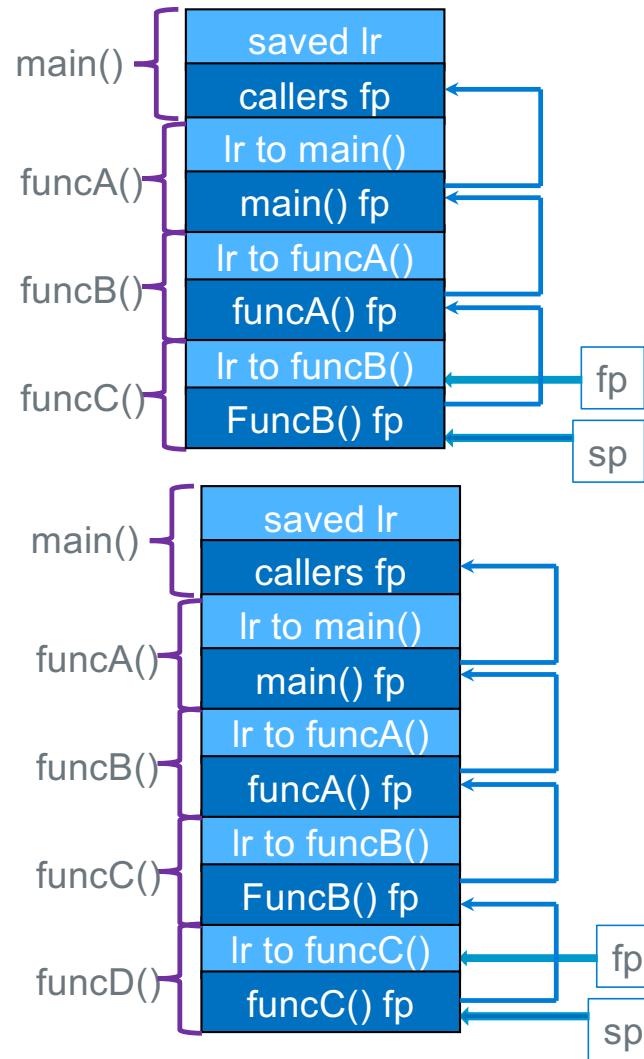


- `sp` always points at top element in the stack (lowest byte address)
- `fp` always points at the bottom element in the stack
 - Bottom element is always the saved `lr` (contains the return address of caller)
 - A saved copy of `callers fp` is always the next element below the `lr`
 - `fp` will be used later when referencing stack variables
- Minimal frame: deallocating at function exit: `pop {fp, lr}`
- On function entry: `sp` must be 8-byte aligned (`sp % 8 == 0`)

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



How gdb finds stack frames

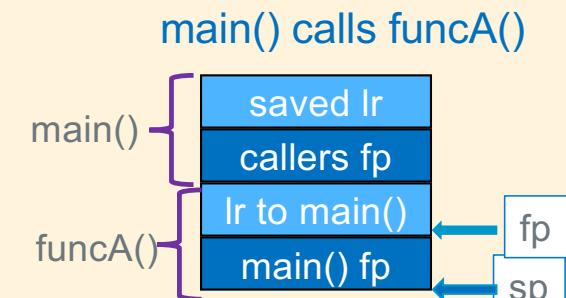


Minimum Stack Frame (Arm Arch32 Procedure Call Standards)

- Function entry (Function Prologue):

1. save lr and fp registers (push)
2. set fp to top entry in stack
3. allocate space for local vars – later slides

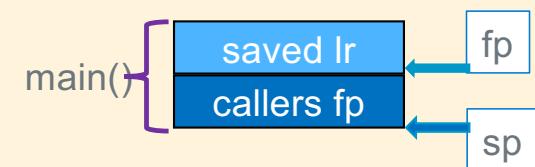
allocate stack space
 $SP = SP - \text{"space"}$
grows "down"



- Function return (Function Epilogue):

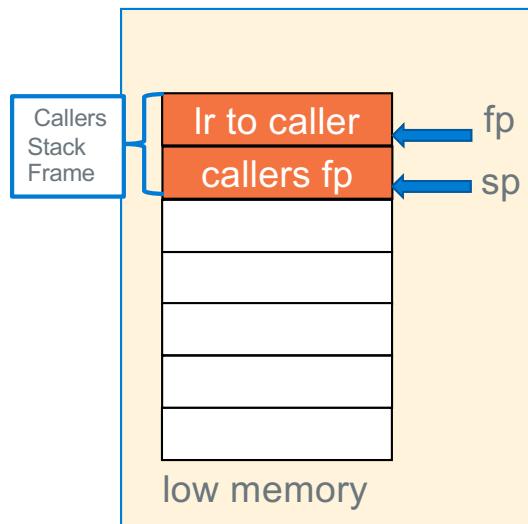
1. deallocate space for locals -later
2. restores lr and fp registers (pop)
3. Return To Caller

deallocate stack space
 $SP = SP + \text{"space"}$
shrinks "up"

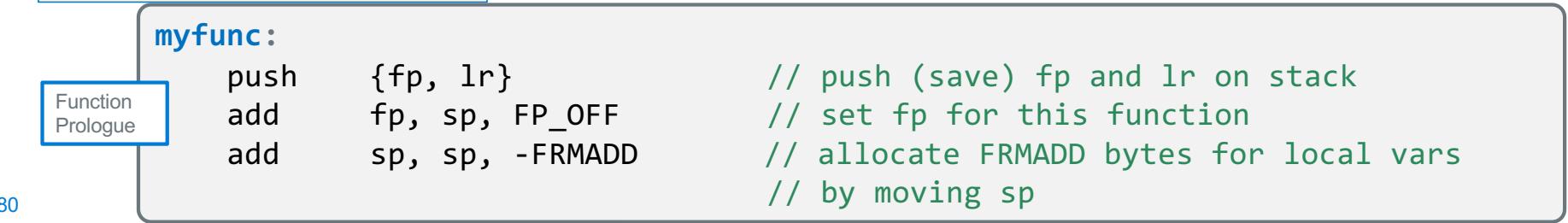


Function Prologue: Allocating the Stack Frame -1

at function entry

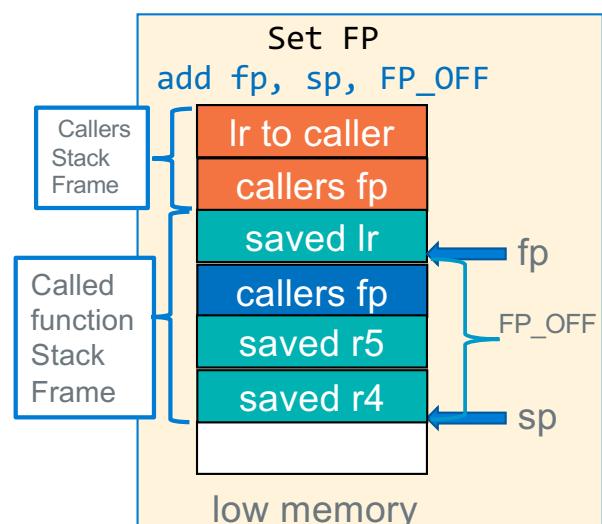


Function was just called this
how the stack looks
The orange blocks are part
of the caller's stack frame



using a **push**, save lr, fp and
those preserved registers it
wants to use on the stack

Prologue Step 2 of 3



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

myfunc:

Function Prologue	push {fp, lr} add fp, sp, FP_OFFSET add sp, sp, -FRMADD	<small>// push (save) fp and lr on stack</small> <small>// set fp for this function</small> <small>// allocate FRMADD bytes for local vars</small> <small>// by moving sp</small>
--------------------------	--	--

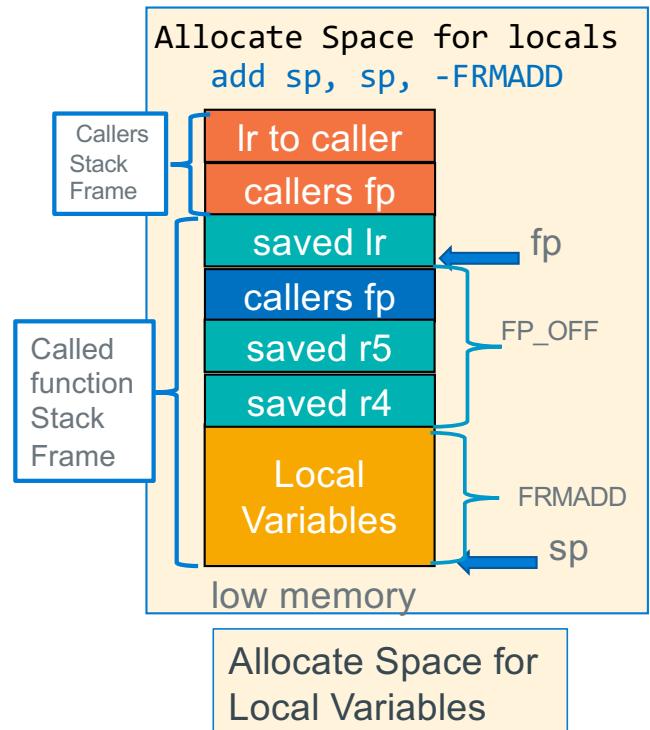
Function Prologue: Allocating the Stack Frame - 2

Prologue Step 3 of 3

- Space for local variables is allocated on the stack right below the lowest pushed register
- Add memory to the stack frame for local variables by moving the sp towards low memory**
- The amount moved is the total size of all local variables in bytes **plus** memory alignment padding

FRMADD = total local var space (bytes) + padding

- Allocate the space after the register push by
`add sp, sp, -FRMADD`
- fp (frame pointer) is used as a **pointer (base register)** to access all stack variables – later slides

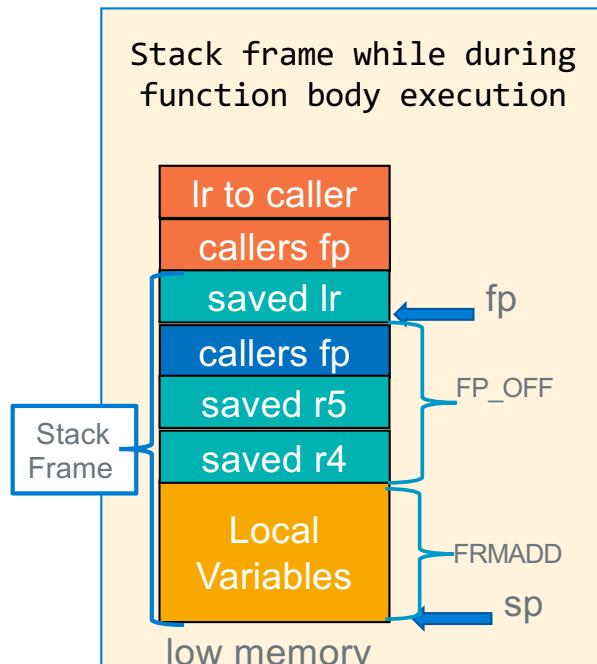


myFunc:

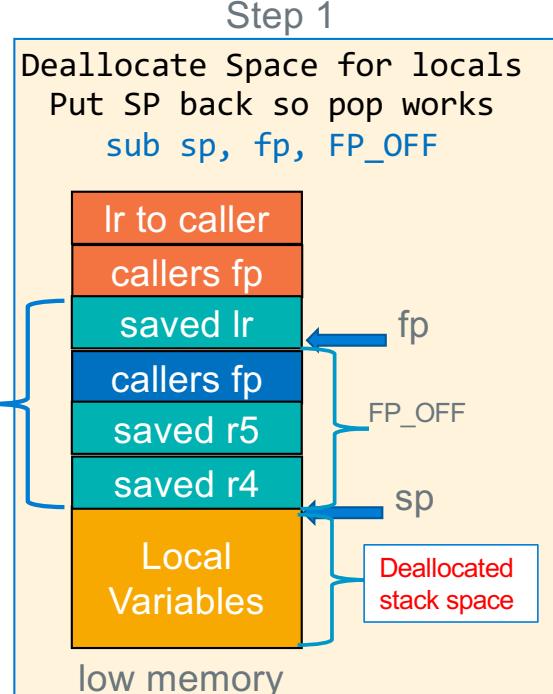
Function Prologue

```
push {fp, lr}           // push (save) fp and lr on stack
add fp, sp, FP_OFFSET // set fp for this function
add sp, sp, -FRMADD   // allocate FRMADD bytes for local vars
                      // by moving sp
```

Function Epilogue: Deallocating the Stack Frame - 1



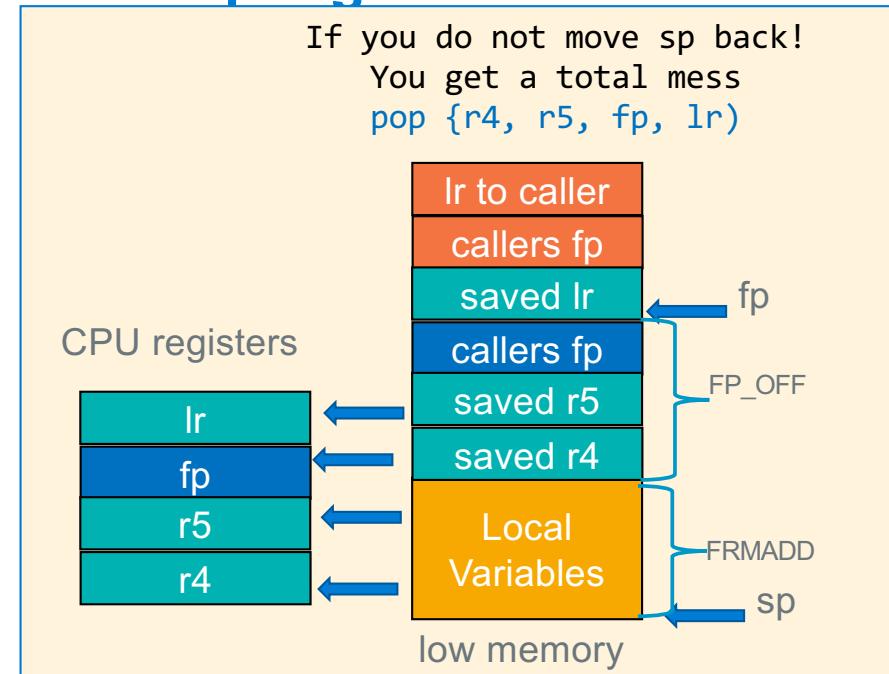
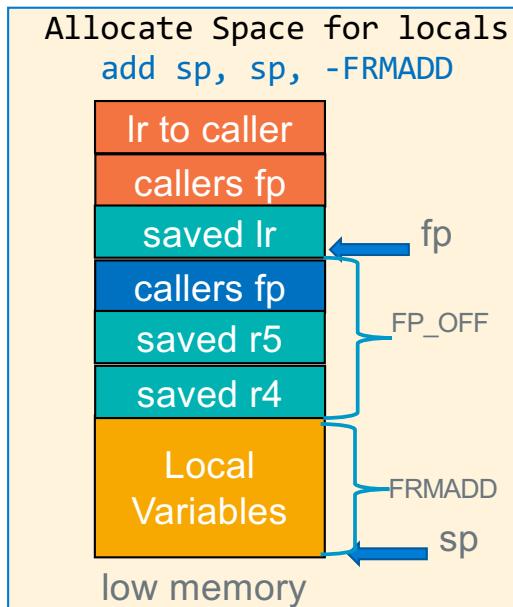
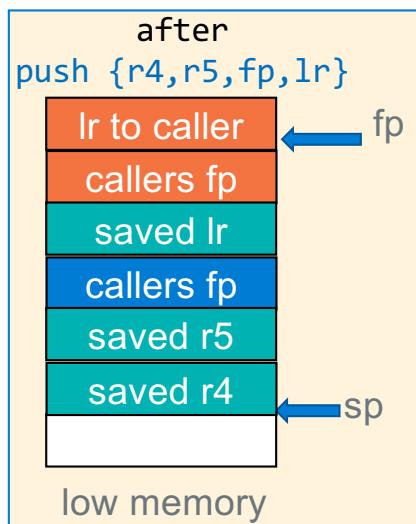
Use fp as a pointer to find local variables on the stack



Move SP back to where it was after the push in the prologue.
So, pop works properly (this also deallocates the local variables)

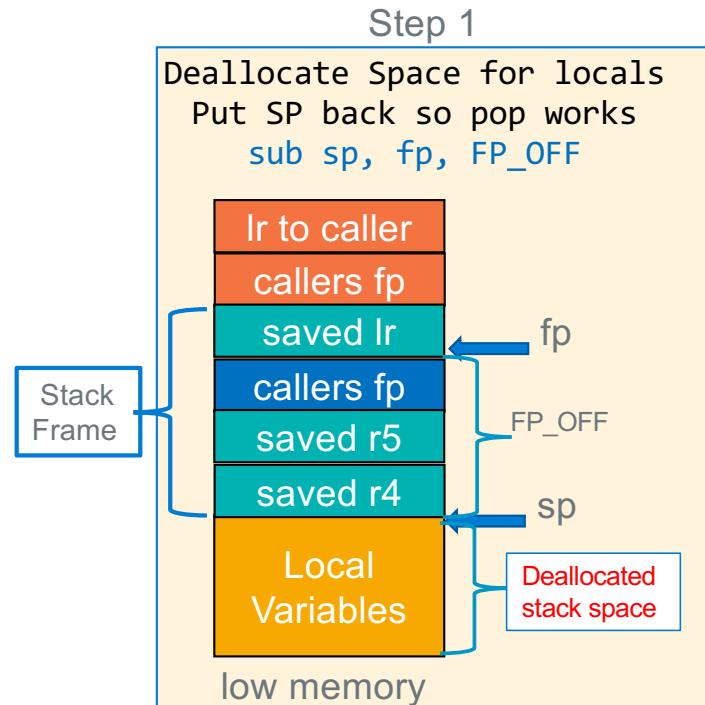
function Epilogue	<code>sub sp, fp, FP_OFFSET</code>	// deallocate local variables by moving sp
	<code>pop {fp, lr}</code>	// pop (restore) fp and lr from stack
	<code>bx lr</code>	// return to caller

Why You must move SP before POP in the Epilogue

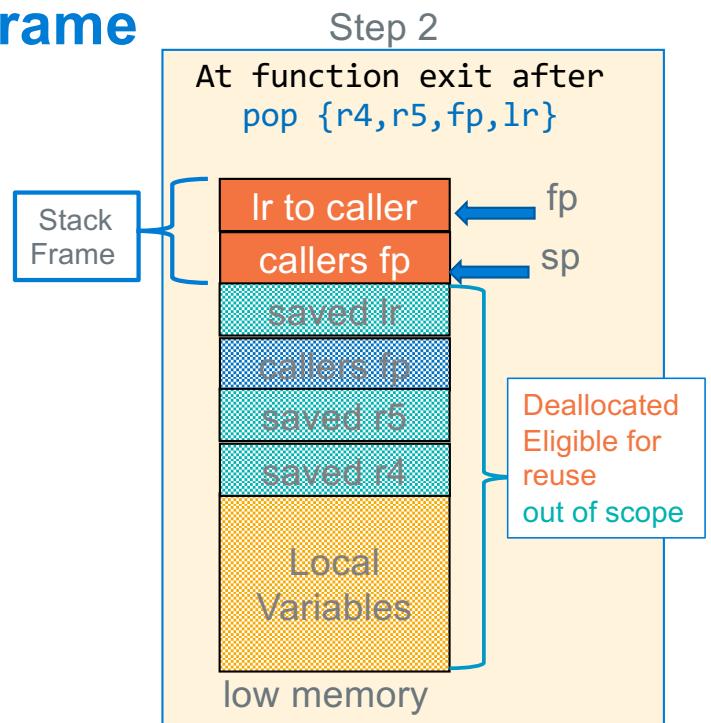


function Epilogue	sub sp, fp, FP_OFFSET	// deallocate local variables by moving sp
	pop {fp, lr}	// pop (restore) fp and lr from stack
	bx lr	// return to caller

Function Epilogue: Deallocating the Stack Frame



Move SP back to where it was after the push in the prologue.
So, pop works properly (this also deallocates the local variables)



Use `pop` to restore the registers to the values they had at function entry

function
Epilogue

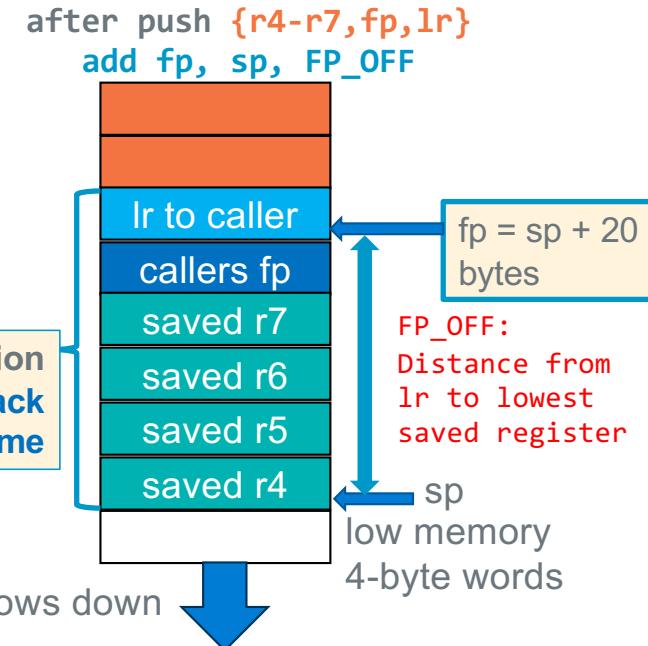
```
sub    sp, fp, FP_OFF      // deallocate local variables by moving sp
pop    {fp, lr}            // pop (restore) fp and lr from stack
bx    lr                  // return to caller
```

How to Set FP

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

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

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



# regs saved	FP_OFFSET in Bytes Distance from lr to lowest saved register
2	4
3	8
4	12
5	16
6	20
7	24
8	28
9	32

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

Reference Table: Global Variable access

<i>var</i>	<i>global variable address into r0 (lside)</i>	<i>global variable contents into r0 (rside)</i>	<i>contents of r0 into global variable</i>
x	ldr r0, =x	ldr r0, [r0]	ldr r1, =x str r0, [r1]
*x	ldr r0, =x ldr r0, [r0]	ldr r0, =x ldr r0, [r0] ldr r0, [r0]	ldr r1, =x ldr r1, [r1] str r0, [r1]
**x	ldr r0, =x ldr r0, [r0] ldr r0, [r0]	ldr r0, =x ldr r0, [r0] ldr r0, [r0] ldr r0, [r0]	ldr r1, =x ldr r1, [r1] ldr r1, [r1] ldr r1, [r1] str r0, [r1]
stderr	ldr r0, =stderr	ldr r0, =stderr ldr r0, [r0]	<do not write unless you really know what you are doing>
.Lstr	ldr r0, =.Lstr	ldr r0, =.Lstr ldrb r0, [r0]	<read only>

stdin, stdout and stderr are global variables

```
.bss // from Libc
stderr: .space 4 // FILE *
```

```
.data
x: .data y //x = &y
```

```
.section .rodata
.Lstr: .string "HI\n"
```

Passing global variables as a parameter: printf()

- `r0 = function(r0, r1, r2, r3)`
`printf(stderr, "arg2", arg3, arg4)`
- create a literal string for arg2 which tells printf() how to interpret the remaining arguments
- stdin, stdout, stderr are all **global variable** and are part of libc
 - these **names are their lside (label names)**
 - get their **contents** and pass that to printf(), fread(), fwrite()

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

    c = a + b;
    printf(stderr, "c=%d\n", c); ←
    r0, r1, r2
    return EXIT_SUCCESS;
}
```

We are going to
put these
variables in
temporary
registers

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

// part of the **text segment** below

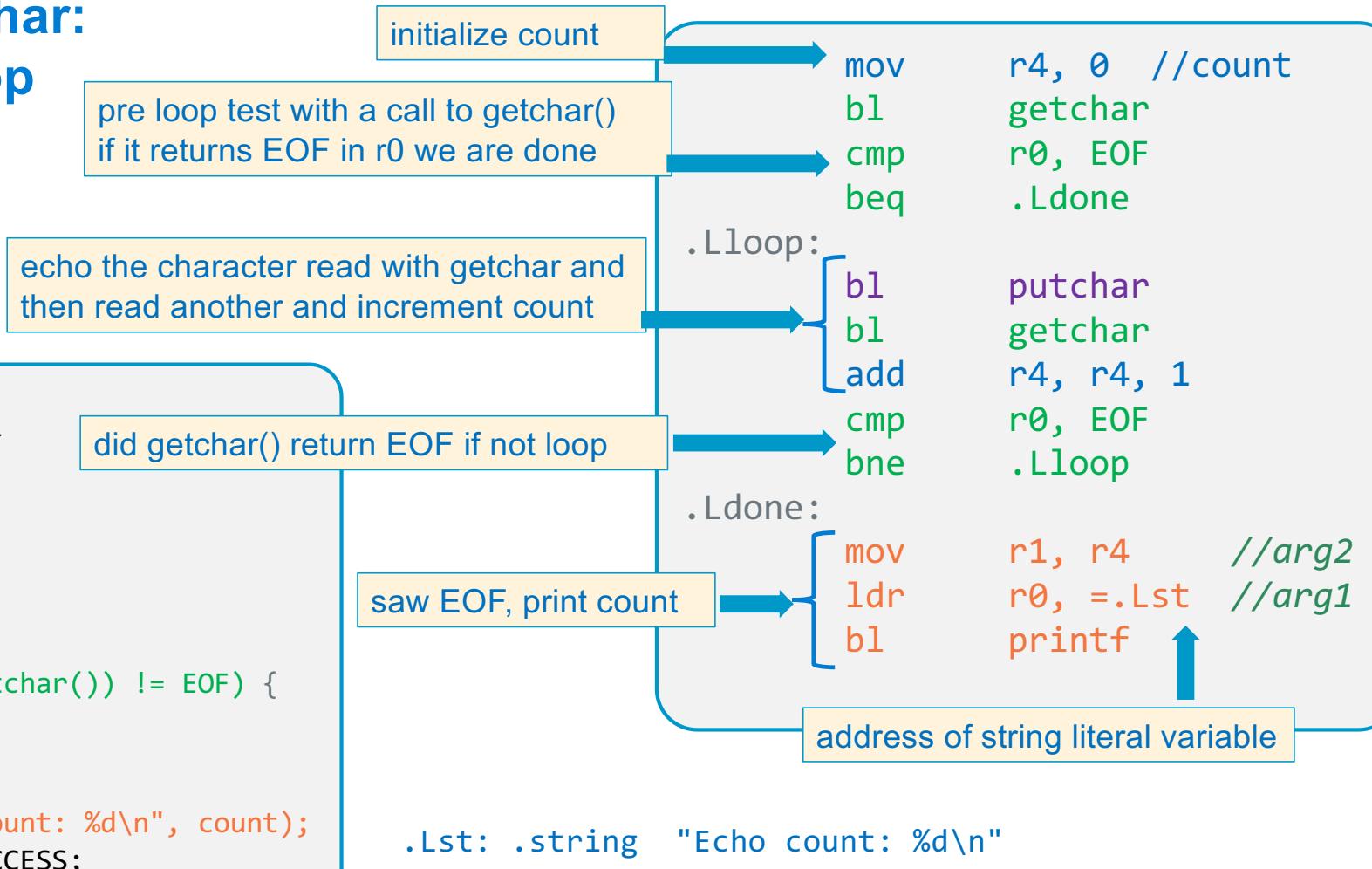
```
mov   r2, 2           // int a = 2;
mov   r3, 3           // int b = 3;
add   r2, r2, r3     // arg 3: int c = a + b;
ldr   r0, =stderr      // get stderr address
ldr   r0, [r0]         // arg 1: get stderr contents
ldr   r1, =.Lfst       // arg 2: =literal address
bl    printf
```

three passed
args in this
use of printf

Putchar/getchar: The while loop

```
#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;
}
```



File header and footers are not shown

Accessing Pointers (argv) in ARM assembly

```

.extern printf
.extern stderr
.section .rodata
.Lstr: .string "argv[%d] = %s\n"
.text
.global main // main(r0=argc, r1=argv)
.type main, %function
.equ FP_OFF, 20
main:
    push {r4-r7, fp, lr}
    add fp, sp, FP_OFFSET
    mov r7, r1 // save argv!
    ldr r4, =stderr // get the address of stderr
    ldr r4, [r4] // get the contents of stderr
    ldr r5, =.Lstr // get the address of .Lstr
    mov r6, 0 // set indx = 0;

// see next slide

.Ldone:
    mov r0, 0
    sub sp, fp, FP_OFFSET
    pop {r4-r7, fp, lr}
    bx lr

```

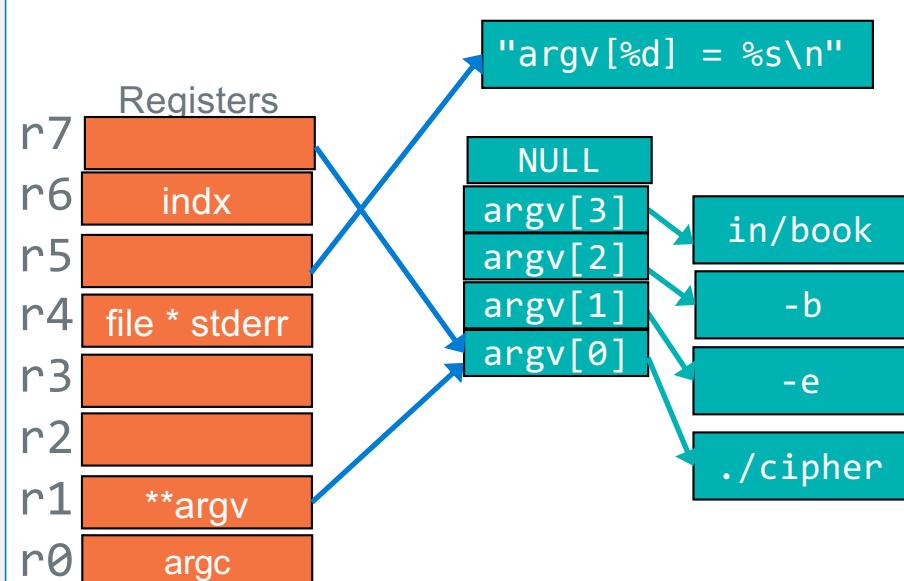
need to save r1 as we are calling a function - fprintf

```

% ./cipher -e -b in/book
argv[0] = ./cipher
argv[1] = -e
argv[2] = -b
argv[3] = in/book

```

r0-r3 lost due to fprintf call



fprintf(stderr, "argv[%d] = %s\n", indx, *argv);

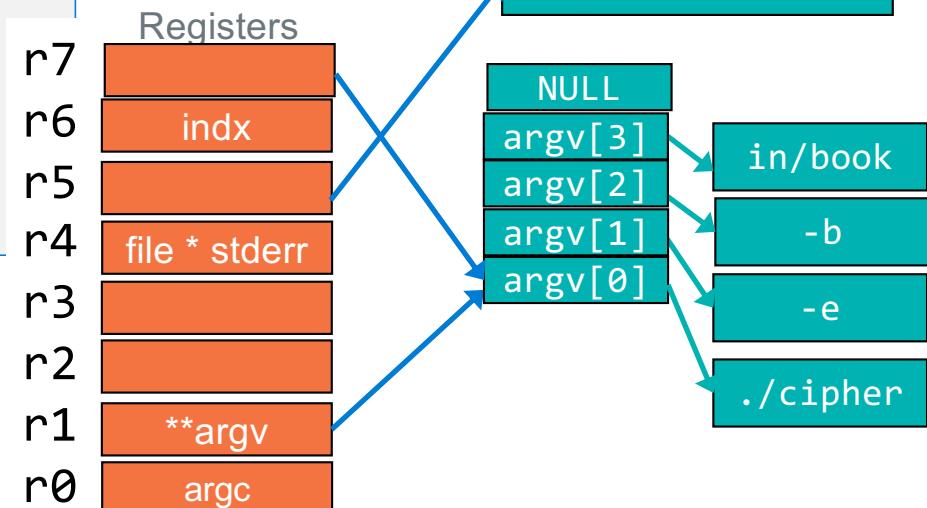
Accessing Pointers (argv) in ARM assembly

```
.Lloop:  
    // fprintf(stderr, "argv[%d] = %s\n", idx, *argv)  
    ldr    r3, [r7]          // arg 4: *argv  
    cmp    r3, 0             // check *argv == NULL  
    beq    .Ldone            // if so done  
    mov    r2, r6              // arg 3: idx  
    mov    r1, r5              // arg 2: "argv[%d] = %s\n"  
    mov    r0, r4              // arg 1: stderr  
    bl     fprintf  
    add    r6, r6, 1           // idx++ for printing  
    add    r7, r7, 4           // argv++ pointer  
    b     .Lloop  
.Ldone:
```

observe the
different
increment sizes

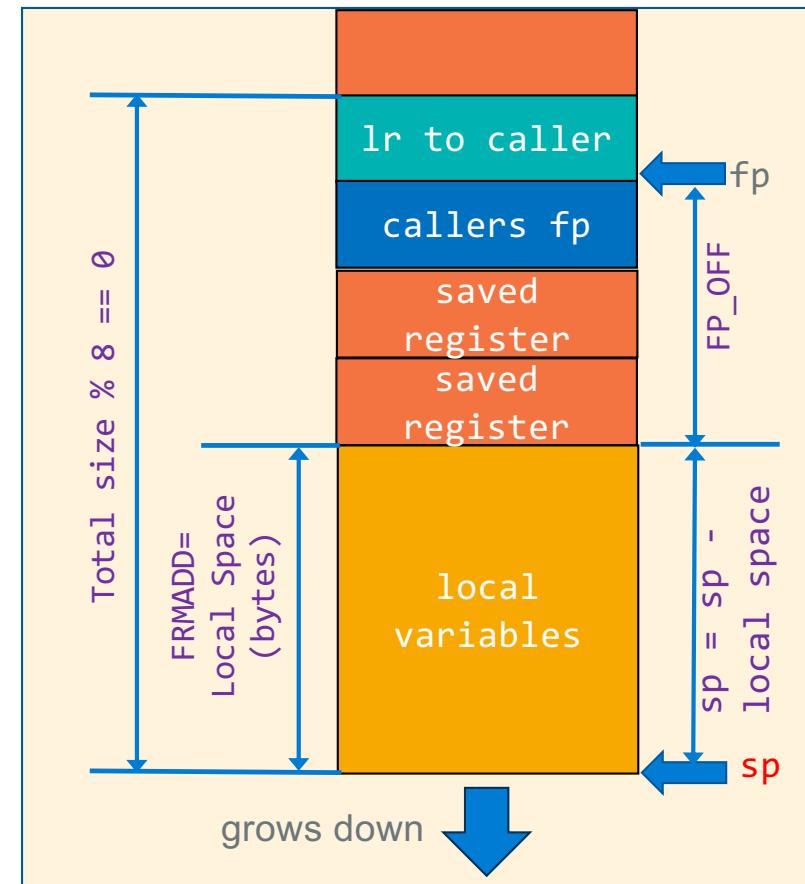
```
% ./cipher -e -b in/B00K  
argv[0] = ./cipher  
argv[1] = -e  
argv[2] = -b  
argv[3] = in/B00K
```

r0-r3 lost due to fprintf call



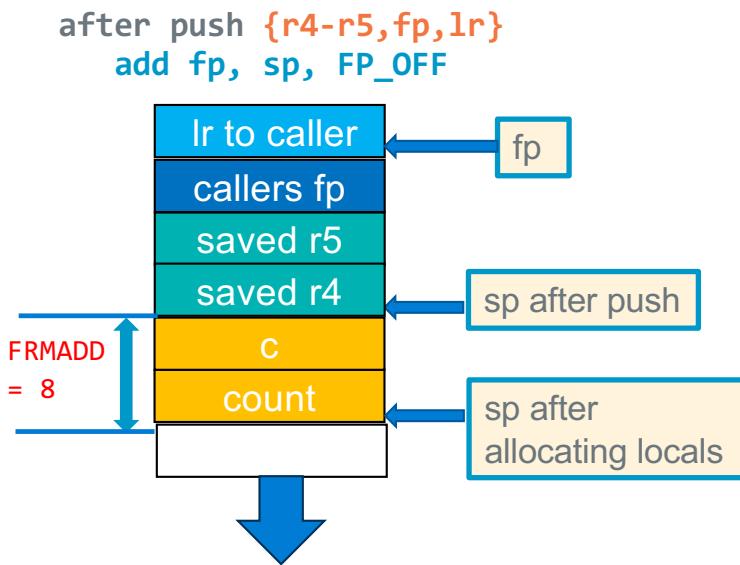
Allocating Space For Locals on the Stack

- Space for local variables is allocated on the stack right below the lowest pushed register
 - Move the **sp** towards low memory by the total size of all local variables in bytes **plus padding**
 $\text{FRMADD} = \text{total local var space (bytes)} + \text{padding}$
- Allocate the space after the register push by
`add sp, sp, -FRMADD`
- Requirement:** on function entry, **sp** is always 8-byte aligned
 sp \% 8 == 0
- Padding (as required):**
 - Additional space between variables on the stack to meet memory alignment requirements
 - Additional space so the frame size is evenly divisible by 8
- fp** (frame pointer) is used as a **pointer (base register)** to access all stack variables – later slides



Local Variables on the stack

```
int main(void)
{
    int c;
    int count = 0;
    // rest of code
}
```



```
.text
.type main, %function
.global main
.equ FP_OFF, 12
.equ FRMADD, 8
main:
    push {r4, r5, fp, lr}
    add fp, sp, FP_OFF
    add sp, sp, -FRMADD
// but we are not done yet!
```

// when FRMADD values fail to assemble
ldr r3, =-FRMADD
add sp, sp, r3

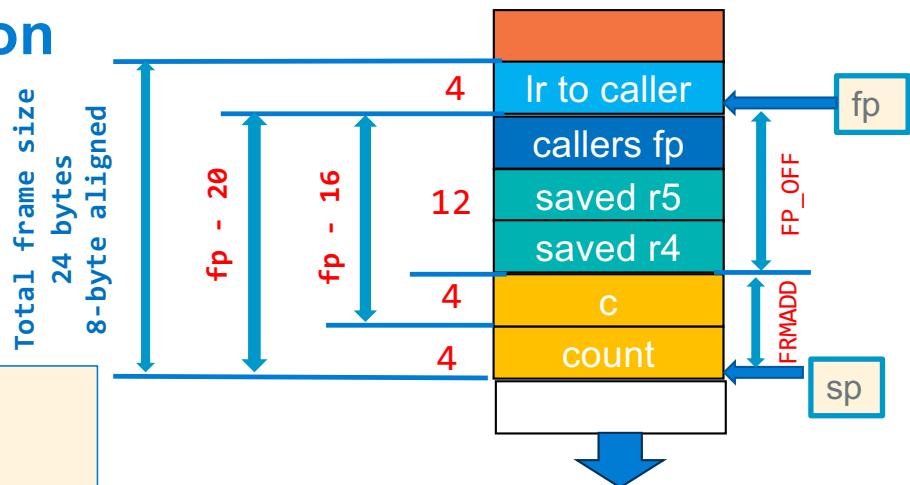
- In this example we are **allocating two variables on the stack**
- When writing assembly functions, in many situations **you may choose to allocate these to registers instead**

- Add space on the stack for each local
 - we will allocate space in same order the locals are listed the C function shown from high to low stack address
 - gcc compiler allocates from low to high stack addresses
 - Order does not matter for our use

Accessing Stack Variables: Introduction

```
int main(void)
{
    int c;
    int count = 0;
    // rest of code
}
```

- To Access data stored in the stack
 - use the `ldr/str` instructions
- Use register **fp** with offset (**distance in bytes**) addressing (use either register offset or immediate offset)
- **No matter what address the stack frame is at, fp always points at saved lr, so you can find a local stack variable by using an offset address from the contents of fp**

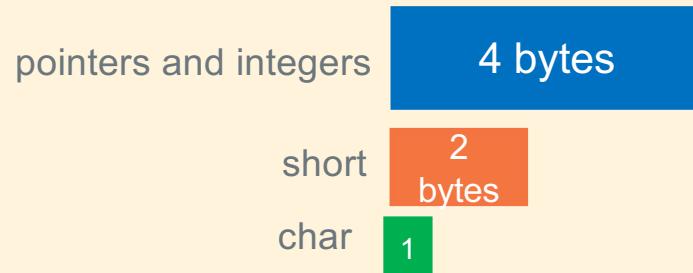


```
.text
.type main, %function
.global main
.equ FP_OFFSET, 12
.equ FRMADD, 8
main:
    push {r4, r5, fp, lr}
    add fp, sp, FP_OFFSET
    add sp, sp, -FRMADD
// but we are not done yet!
```

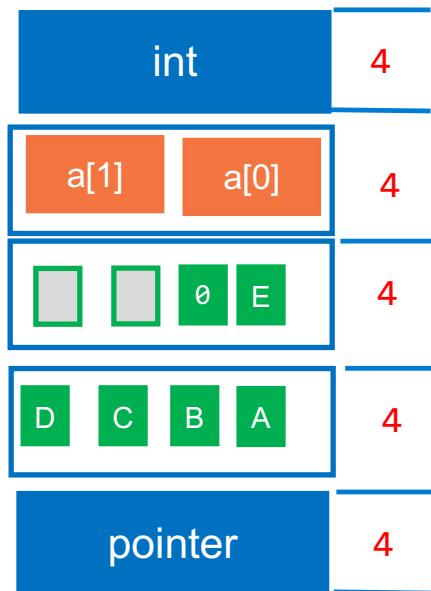
Variable	distance from fp	Read variable	Write Variable
<code>int c</code>	-16	<code>ldr r0, [fp, -16]</code>	<code>str r0, [fp, -16]</code>
<code>int count</code>	-20	<code>ldr r0, [fp, -20]</code>	<code>str r0, [fp, -20]</code>

Stack Frame Design – Local Variables

- When writing an ARM equivalent for a C program, for CSE30 we will not re-arrange the order of the variables to optimize space (covered in the compiler course)
- Arrays start at a 4-byte boundary (even arrays with only 1 element)
 - Exception: double arrays [] start at an 8-byte boundary
 - struct arrays are aligned to the requirements of largest member
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- Padding may be required (see next slide)

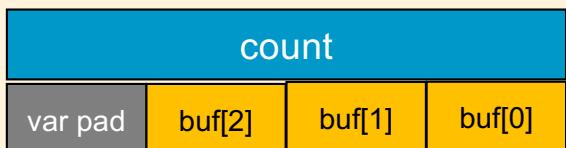


Rule: When the function is entered the stack is already 8-byte aligned

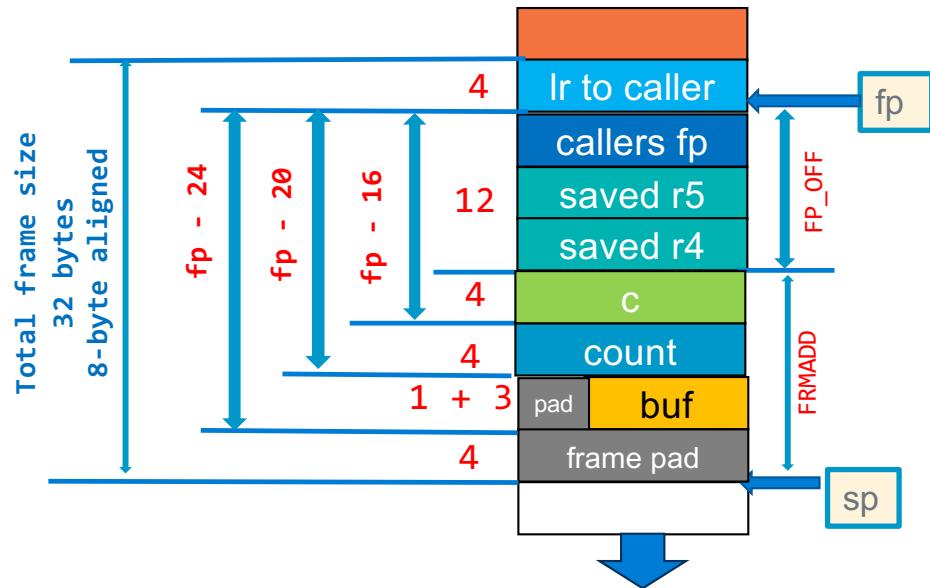


Stack Variables: Padding

- **Variable padding** – start arrays at 4-byte boundary and **leave unused space at end** (high side address) before the variable higher on the stack



- **Frame padding** – add space below the last local variable to keep 8-byte alignment



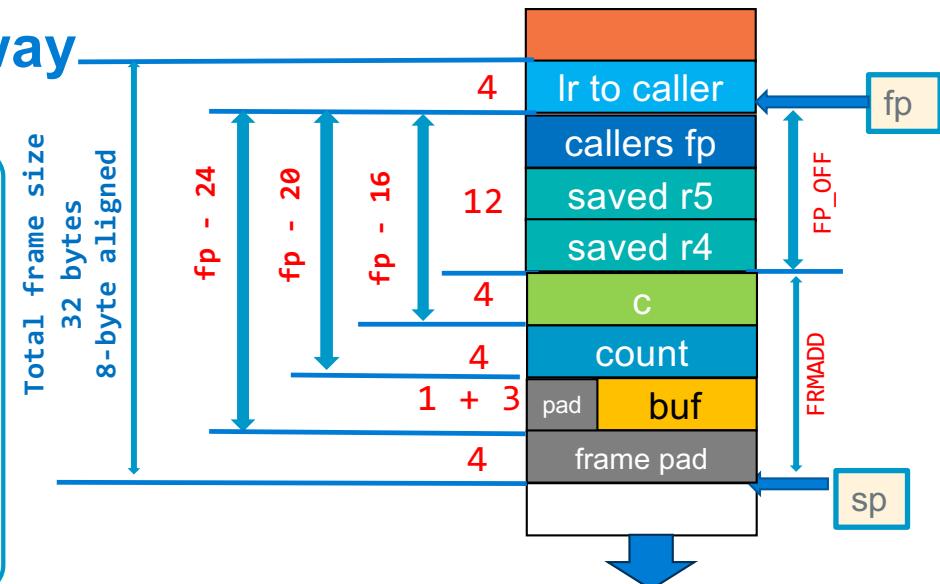
```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
.text
.type    main, %function
.global   main
.equ     FP_OFF,      12
.equ     FRMADD,      16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
// but we are not done yet!
```

Accessing Stack Variables, the hard way

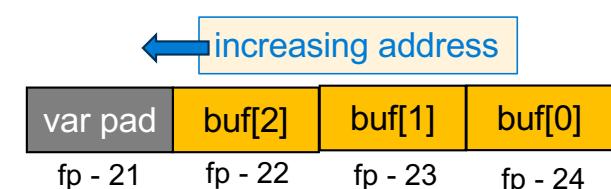
```
int main(void)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    // rest of code
}
```

```
.text
.type    main, %function
.global   main
.equ     FP_OFF,      12
.equ     FRMADD,      16
main:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    add     sp, sp, -FRMADD
// but we are not done yet!
```



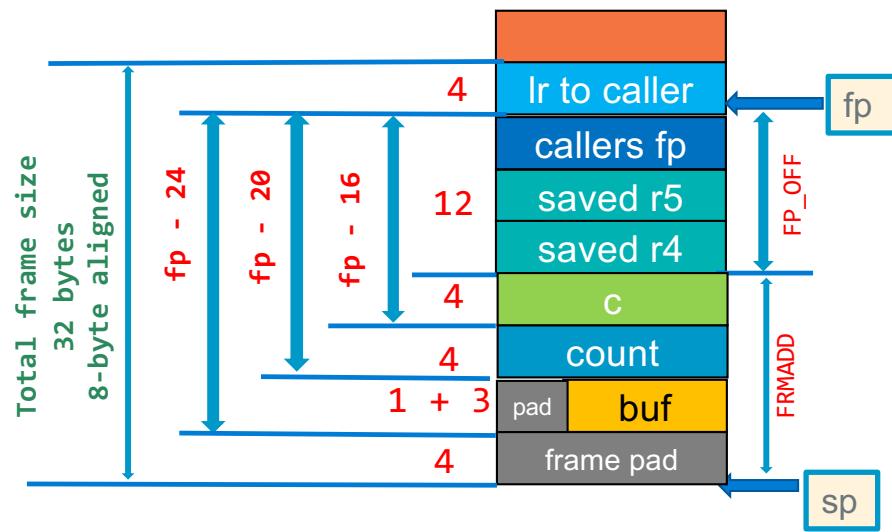
char buf[] by usage with ASCII chars we will use strb (or make it unsigned char)

Variable	distance from fp	Read variable	Write Variable
int c	16	ldr r0, [fp, -16]	str r0, [fp, -16]
int count	20	ldr r0, [fp, -20]	str r0, [fp, -20]
char buf[0]	24	ldrb r0, [fp, -24]	strb r0, [fp, -24]
char buf[1]	23	ldrb r0, [fp, -23]	strb r0, [fp, -23]
char buf[2]	22	ldrb r0, [fp, -22]	strb r0, [fp, -22]



- Calculating offsets is a lot of work to get it correct
- It is also hard to debug
- There is a better way!

Best Practice: Assembler Generated FP Distance Table



FP Distance Table one For each function

```
.type main, %function
.global main
    pushed reg fp distance // pushed reg fp distance

.equ FP_OFF, 12 // Prior allocation distance
variable size in bytes

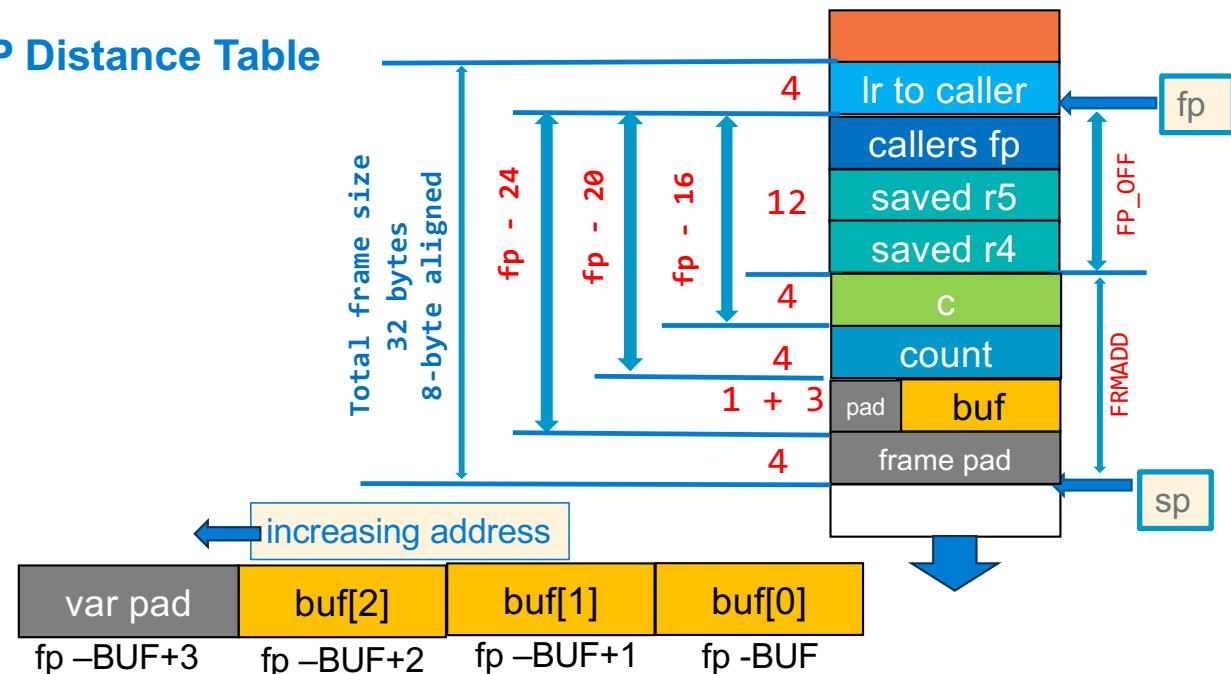
.equ C, 4 + FP_OFF
.equ COUNT, 4 + C
.equ BUF, 4 + COUNT
.equ PAD, 4 + BUF
.equ FRMADD, PAD - FP_OFF
// FRMADD = 28 - 12 = 16
```

- For each stack variable create a **.equ symbol** whose **value** is the **distance in bytes** from the **FP** after the **prologue**
- After the last variable add a name **PAD** for the size of the frame padding (if any). if no padding, PAD will be set to the same value as the variable above it
- The **value of the symbol** is an **expression that calculates the distance from the FP** based on the distance of the variable above it on the stack. The first variable will use **SP_OFF** as the starting distance
.equ VAR, size_of var + variable_padding + previous_var_symbol // previous_var_symbol distance of the var above
- Calculate the size of the local variable area that needs to be added to the **sp** in bytes
FRMADD = distance PAD minus distance of the SP to the FP (FP_OFF) after the prologue push

Best Practice: Assembler Generated FP Distance Table

FP Distance Table For each function

```
.type main, %function
.global main
.equ FP_OFF, 12
.equ C, 4 + FP_OFF
.equ COUNT, 4 + C
.equ BUF, 4 + COUNT
.equ PAD, 4 + BUF
.equ FRMADD, PAD - FP_OFF
// FRMADD = 28 - 12 = 16
```



Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add r0, fp, -BUF+1	ldrb r0, [fp, -BUF+1]	strb r0, [fp, -BUF+1]
char buf[2]	BUF-2	add r0, fp, -BUF+2	ldrb r0, [fp, -BUF+2]	strb r0, [fp, -BUF+2]

Initializing and Accessing Stack variables

```

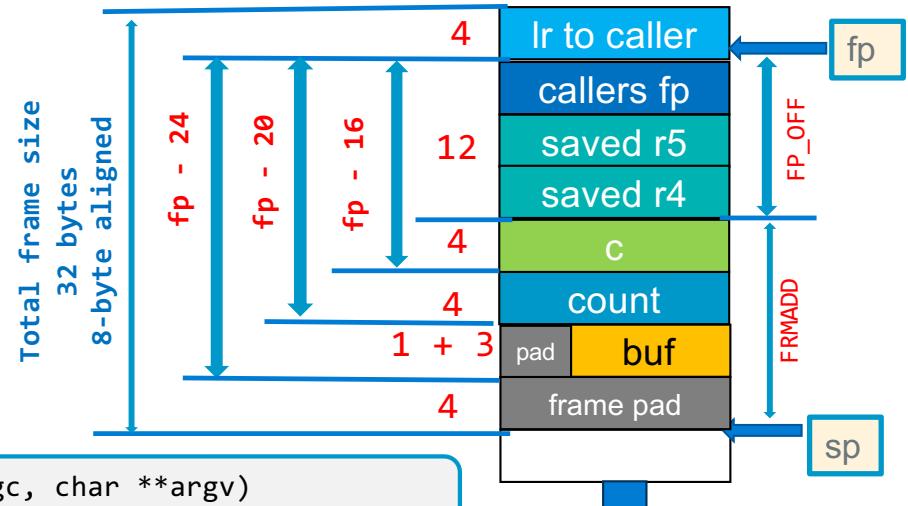
.section .rodata
.Lmess: .string "%d %d %s\n"
.extern printf

main:
    push {r4, r5, fp, lr}
    add fp, sp, FP_OFF
    add sp, sp, -FRMADD
    // nothing to do for C
    mov r2, 0
    str r2, [fp, -COUNT]
    strb r2, [fp, -BUF+2]
    mov r2, 'h'
    strb r2, [fp, -BUF]
    mov r2, 'i'
    strb r2, [fp, -BUF+1]

    ldr r0, =.Lmess      // arg1
    ldr r1, [fp, -C]     // arg2
    ldr r2, [fp, -COUNT] // arg3
    add r3, fp, -BUF    // arg4
    bl printf

    passes address of a stack variable buf
    passes contents of stack var C and COUNT

```



```

int main(int argc, char **argv)
{
    int c;
    int count = 0;
    char buf[] = "hi";
    printf("%d %d %s\n", c, count, buf);
    // rest of code
    pass stack address

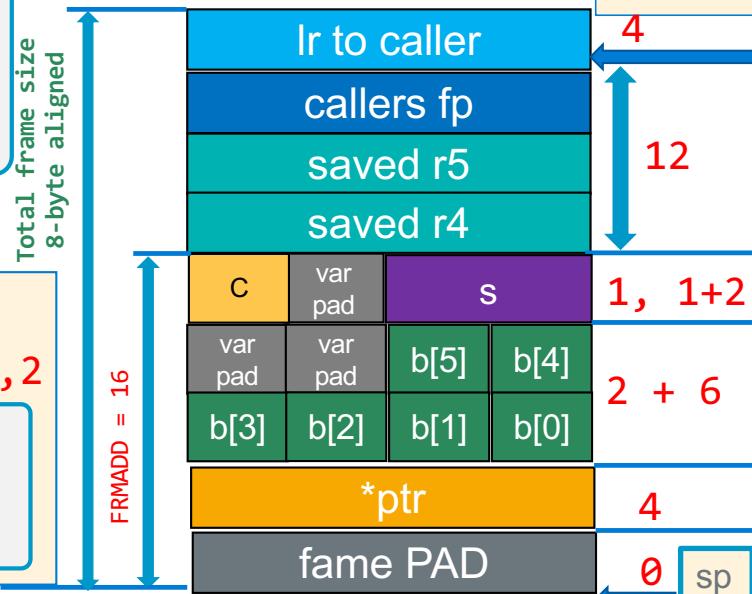
```

./a.out
-136572160 0 hi

Variable	distance from fp	Address on Stack	Read variable	Write Variable
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	COUNT	add r0, fp, -COUNT	ldr r0, [fp, -COUNT]	str r0, [fp, -COUNT]
char buf[0]	BUF	add r0, fp, -BUF	ldrb r0, [fp, -BUF]	strb r0, [fp, -BUF]
char buf[1]	BUF-1	add r0, fp, -BUF+1	ldrb r0, [fp, -BUF+1]	strb r0, [fp, -BUF+1]
char buf[2]	BUF-2	add r0, fp, -BUF+2	ldrb r0, [fp, -BUF+2]	strb r0, [fp, -BUF+2]

Stack Frame Design Practice

```
void func(void)
{
    signed char c;
    signed short s;
    unsigned char b[] = "Stack";
    unsigned char *ptr = &b;
    // rest of code
}
```



Alternative design



```
.equ FP_OFF, 12
.equ C, 2 + FP_OFF
.equ S, 2 + C
...
...
```

1. Write the variables in C
2. Draw a picture of the stack frame
3. Write the code to generate the offsets
4. create the distance table to the variables

```
.equ FP_OFF, 12
.equ C, 1 + FP_OFF
.equ S, 3 + C
.equ B, 8 + S
.equ PTR, 4 + B
.equ PAD, 0 + PTR
.equ FRMADD, PAD - FP_OFF
// FRMADD = 28 - 12 = 16
```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
signed char c	C	add r0, fp, -C	ldr sb r0, [fp, -C]	str sb r0, [fp, -C]
signed short s	S	add r0, fp, -S	ldr sh r0, [fp, -S]	str sh r0, [fp, -S]
unsigned char b[0]	B	add r0, fp, -B	ldr b r0, [fp, -B]	str b r0, [fp, -B]
unsigned char *ptr	PTR	add r0, fp, -PTR	ldr r0, [fp, -PTR]	str r0, [fp, -PTR]

Working with Pointers on the stack

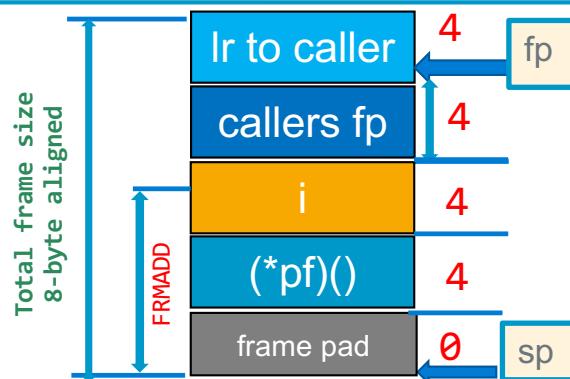
```
int sum(int j, int k)
{
    return j + k;
}
void testp(int j, int k, int (*func)(int, int), int *i)
{
    *i = func(j,k);
    return;
}
int main()
{
    int i;                      // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum;   // pf could be in a register

    testp(1, 2, pf, &i);      ← Output Parameters (like i) you
    printf("%d\n", i);         pass a pointer to them, must be
    return EXIT_SUCCESS;       on the stack!
}
```

Working with Pointers on the stack

```
int main()
{
    int i; // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

    testp(1, 2, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```



```
.section .rodata
.Lmess: .string "%d\n"
.extern printf
.text
.global main
.type main, %function
.equ FP_OFF, 4
.equ I, 4 + FP_OFF
.equ PF, 4 + I
.equ PAD, 0 + PF
.equ FRMADD, PAD - FP_OFF
// FRMADD = 12 - 4 = 8
```

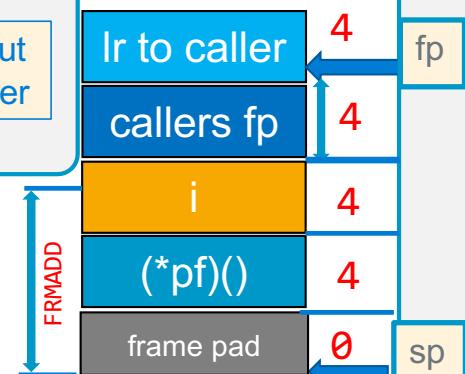
Variable	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]

Working with Pointers on the stack

```
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, pf, &i) ← I is Output Parameter
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

```
.section .rodata
.Lmess: .string "%d\n"
.extern printf
.text
.global main
.type main, %function
.equ FP_OFF, 4
.equ I, 4 + FP_OFF
.equ PF, 4 + I
.equ PAD, 0 + PF
.equ FRMADD, PAD - FP_OFF
// FRMADD = 12 - 4 = 8
```



main:

```

push {fp, lr}
add fp, sp, FP_OFFSET
add sp, sp, -FRMADD

ldr r2, =sum           // func address
str r2, [fp, -PF]      // store in pf

mov r0, 1               // arg 1: 1
mov r1, 2               // arg 2: 2
ldr r2, [fp, -PF]       // arg 3: (*pf)()
add r3, fp, -I          // arg 4: &I
bl

testp

ldr r0, =.Lmess         // arg 1: "%d\n"
ldr r1, [fp, -I]         // arg 2: I
printf

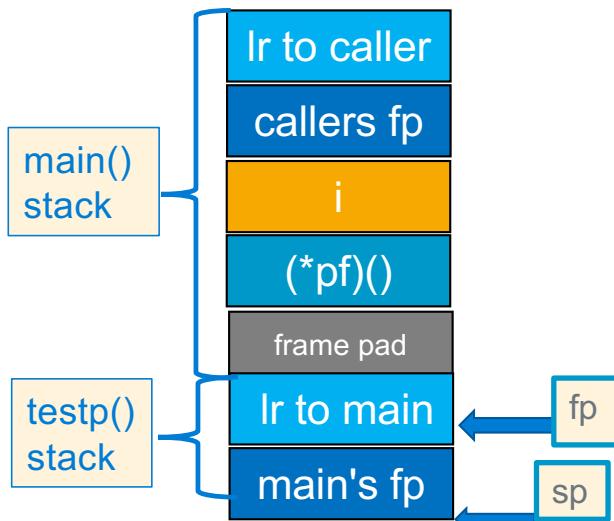
sub sp, fp, FP_OFFSET
pop {fp, lr}
bx lr

```

Variable	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]

Working with Pointers on the stack

```
void  
testp(int j, int k, int (*func)(int, int), int *i)  
{  
    *i = func(j, k);  
    return;  
}
```



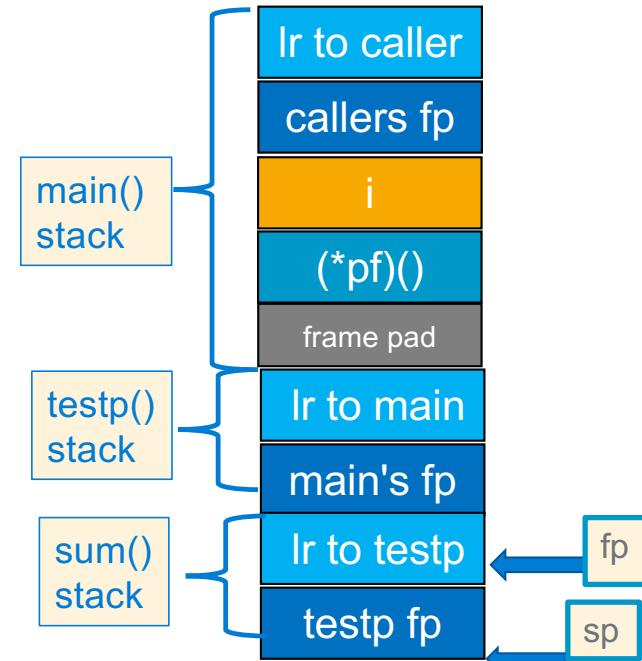
r0,r1,r2 already set

```
.global testp  
.type testp, %function  
.equ FP_0FF, 12  
testp:  
    push {r4, r5, fp, lr}  
    add fp, sp, FP_0FF  
    mov r4, r3 // save i  
    blx r2 // r0=func(r0,r1)  
    str r0, [r4] // *i =r0  
    sub sp, fp, FP_0FF  
    pop {r4, r5, fp, lr}  
    bx lr  
.size testp, (. - testp)
```

Working with Pointers on the stack

```
int
sum(int j, int k)
{
    return j + k;
}
```

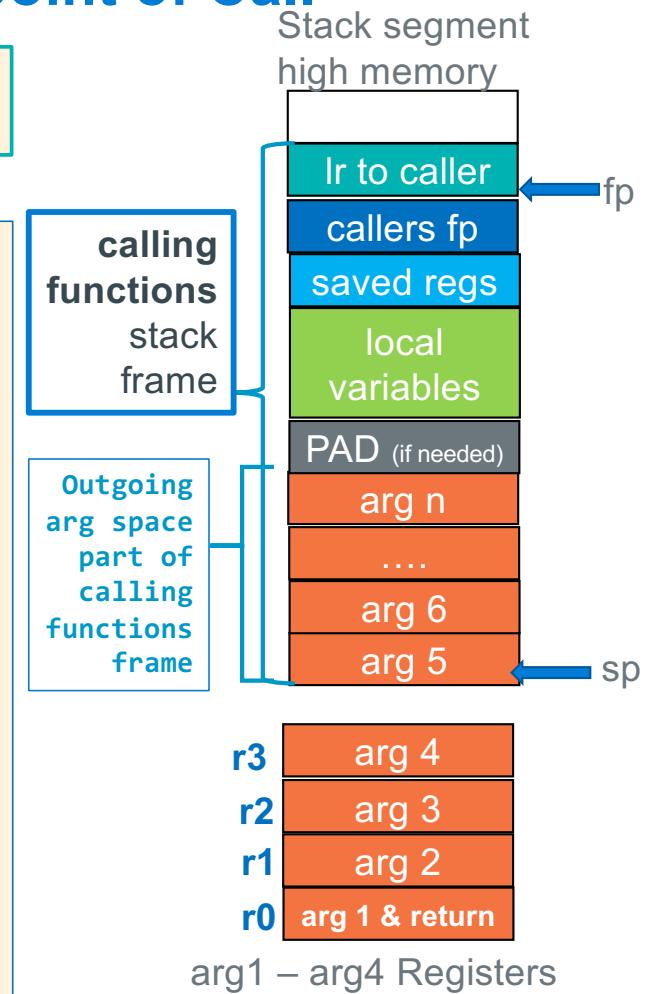
```
.global sum
.type   sum, %function
.equ    FP_OFF, 4
sum:
    push   {fp, lr}
    add    fp, sp, FP_OFFSET
    add    r0, r0, r1
    sub    sp, fp, FP_OFFSET
    pop   {fp, lr}
    bx     lr
.size sum, (. - sum)
```



Passing More Than Four Arguments – At the point of Call

```
r0 = function(r0, r1, r2, r3, arg5, arg6, ... argn)
      arg1, arg2, arg3, arg4, ...
```

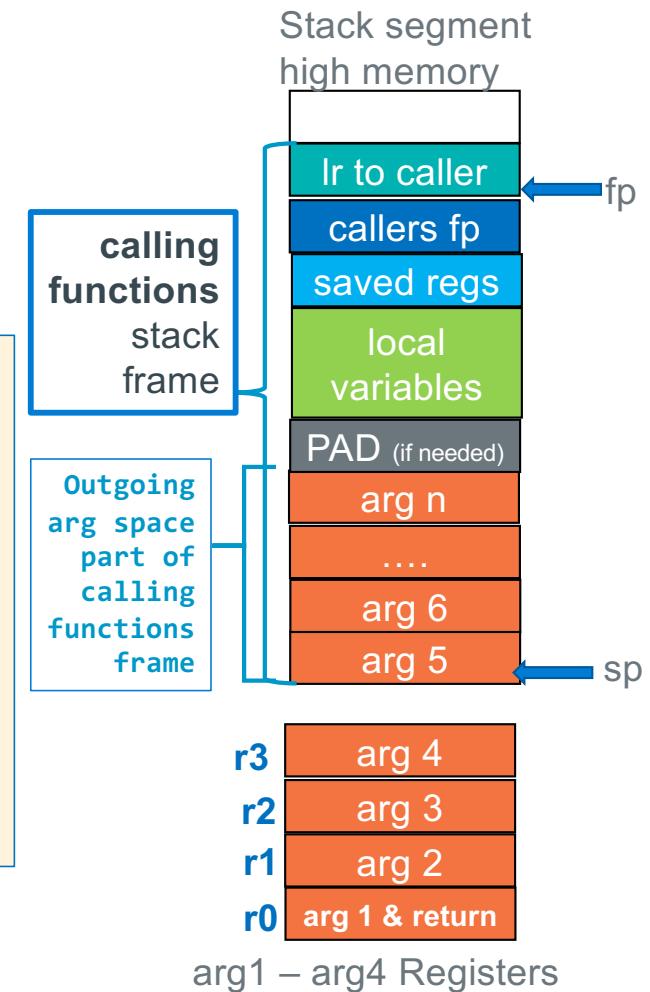
- Approach: Increase stack frame size to include space for args# > 4
 - Arg5 and above are in caller's stack frame at the **bottom of the stack**
- Arg5 is always at the **bottom (at sp)**, arg6 and greater are above it
- **One arg value per slot!** – NO arrays across multiple slots
 - chars, shorts and ints are directly stored
 - Structs (not always), and arrays (always) are passed via a pointer
- Output parameters contain an address **that points at** the stack, **BSS**, **data**, or **heap**
- Prior to any function call (and obviously at the start of the called function):
 1. sp must point at arg5
 2. sp and therefore **arg5 must be at an 8-byte boundary**,
 3. **Add padding to force arg5 alignment if needed is placed above the last argument the called function is expecting**



Passing More Than Four Arguments – At the point of Call

```
r0 = function(r0, r1, r2, r3, arg5, arg6, ... argn)  
      arg1, arg2, arg3, arg4, ...
```

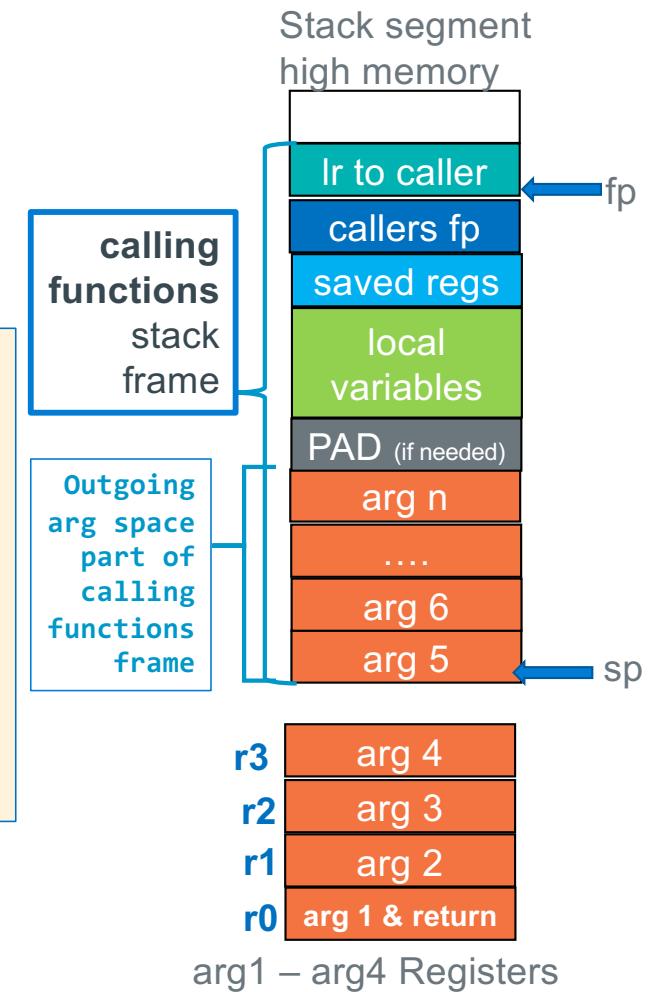
- Called functions have the **right to change stack args** just like they can change the register args!
 - Caller **must always assume all args including ones on the stack are changed by the caller**
- Calling function prior to making the call you must
 - Evaluate **first four args**: place the resulting **values** in r0-r3
 - Evaluate Arg 5 and greater and place the resulting values on the stack



Passing More Than Four Arguments – At the point of Call

```
r0 = function(r0, r1, r2, r3, arg5, arg6, ... argn)  
      arg1, arg2, arg3, arg4, ...
```

- **Approach:** Extend the stack frame to include enough space for stack arguments for the called function that has the greatest number of args
 1. Examine every function call in the body of a function
 2. Find the function call with greatest arg count, this determines space needed for outgoing args
 3. Add the greatest arg count space as needed to the frame layout
 4. Adjust PAD as required to keep the sp 8-byte aligned



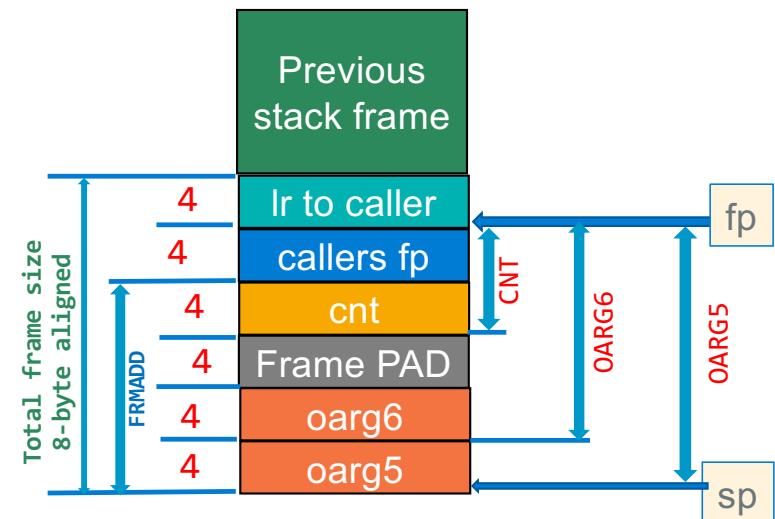
Calling Function Stack Frame: Pass ARG 5 and higher

Rules: At point of call

1. OARG5 must be pointed at by sp
2. SP must be 8-byte aligned at function call

```
int cnt;
r0 = func(r0, r1, r2, r3, OARG5, OARG6);
```

```
.equ FP_OFF, 4
.equ CNT,        4 + FP_OFF      // int cnt
.equ PAD,        4 + CNT        // added as needed
.equ OARG6,      4 + PAD        // 4 bytes
.equ OARG5,      4 + OARG6      // 4 bytes
.equ FRMADD     OARG5 - FP_OFF
// FRMADD = 20 - 4 = 16
```



Variable	distance from fp	Address on Stack	Read variable	Write Variable
int cnt	CNT	add r0, fp, -CNT	ldr r0, [fp, -CNT]	str r0, [fp, -CNT]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

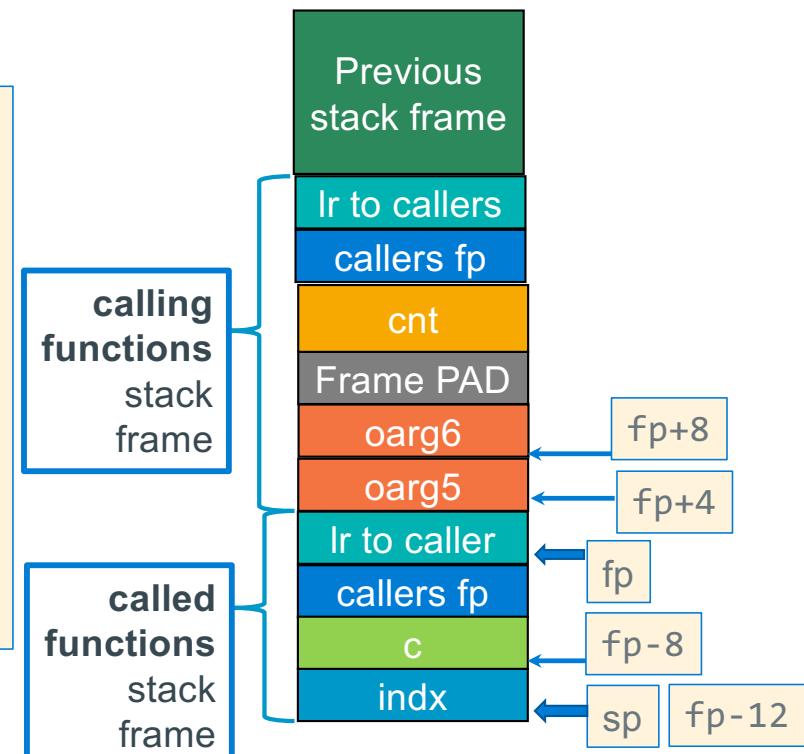
Called Function: Retrieving Args From the Stack

```
r0 = func(r0, r1, r2, r3, ARG5, ARG6);
```

- At function start and before the push{} the **sp** is at an 8-byte boundary
- **Args > 4 in caller's stack frame and arg 5 always starts at fp+4**
 - Additional args are higher up the stack, with one “slot” every 4-bytes
 - .equ ARGN, (N-4)*4 // where n must be > 4
- This “algorithm” for finding args was designed to enable **variable** arg count functions like printf("conversion list", arg0, ... argn);
- No limit to the number of args (except running out of stack space)

Rule:

Called functions always access stack args using a **positive offset to the fp**



Called Function: Retrieving Args From the Stack

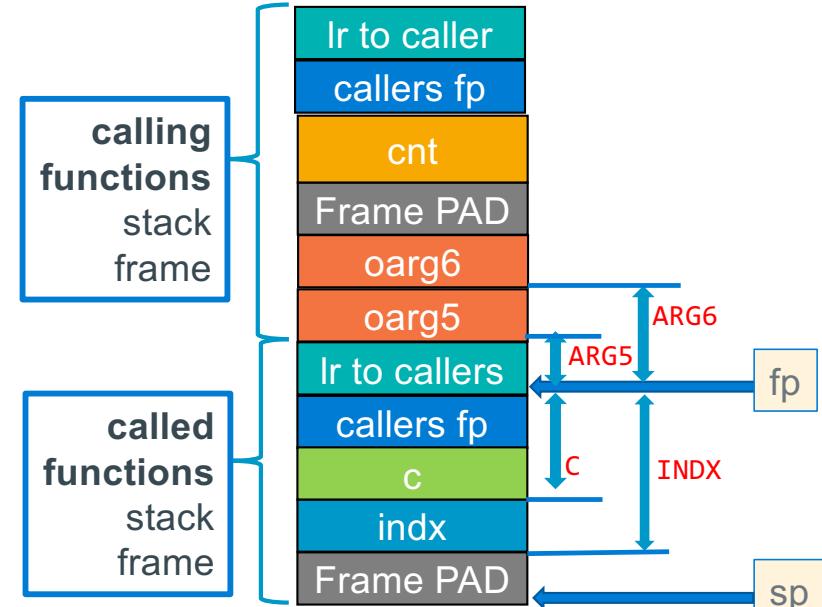
```

.equ    FP_OFF,      4
.equ    C,           4 + FP_OFF
.equ    INDX,        4 + C
.equ    PAD,          0 + INDX
.equ    FRMADD,     PAD - FP_OFF
// below are distances into the caller's stack frame
.equ    ARG6,         8
.equ    ARG5,         4

```

r0 = func(r0, r1, r2, r3, r4, ARG5, ARG6);

Rule:
Called functions always access stack args
using a **positive offset to the fp**



Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
int arg6	ARG6	add r0, fp, ARG6	ldr r0, [fp, ARG6]	str r0, [fp, ARG6]
int arg5	ARG5	add r0, fp, ARG5	ldr r0, [fp, ARG5]	str r0, [fp, ARG5]
int c	C	add r0, fp, -C	ldr r0, [fp, -C]	str r0, [fp, -C]
int count	INDX	add r0, fp, -INDX	ldr r0, [fp, -INDX]	str r0, [fp, -INDX]

Observe the positive offsets

Example: Passing Stack Args, Calling Function

```
int sum(int j, int k)
{
    return j + k;
}

void arg1    arg2    arg3    arg4      arg5          arg6
testp(int j, int k, int l, int m, int (*func)(int, int), int *i)
{
    *i = func(j,k) + func(l, m); // notice two func() calls

    return;
}

int main()
{
    int i; // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);

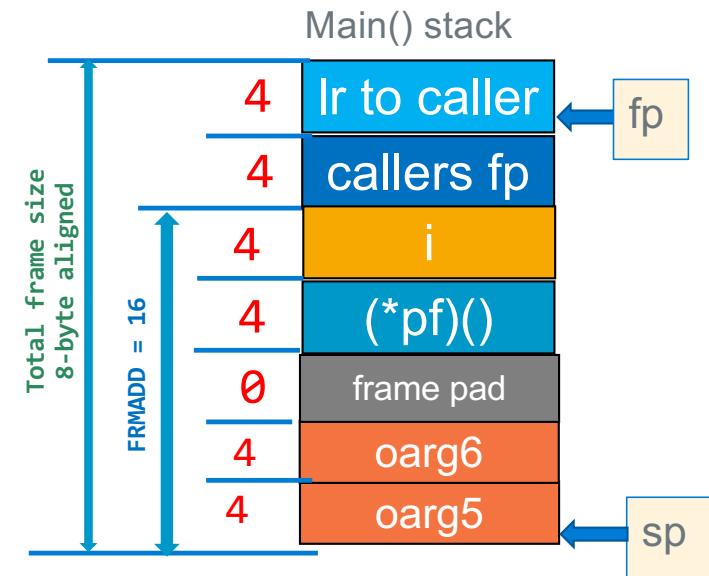
    return EXIT_SUCCESS;
}
```

Example: Passing Stack Args, Calling Function

```
int main()
{
    int i; // NOTICE: i must be on stack as you pass the address!
    int (*pf)(int, int) = sum; // pf could be in a register

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

```
.equ    FP_OFF, 4
.equ    I,      4 + FP_OFF
.equ    PF,     4 + I
.equ    PAD,    0 + PF
.equ    OARG6,  4 + PAD
.equ    OARG5,  4 + OARG6
.equ    FRMADD, OARG5 - FP_OFF
// FRMADD = 20 - 4 = 16
```



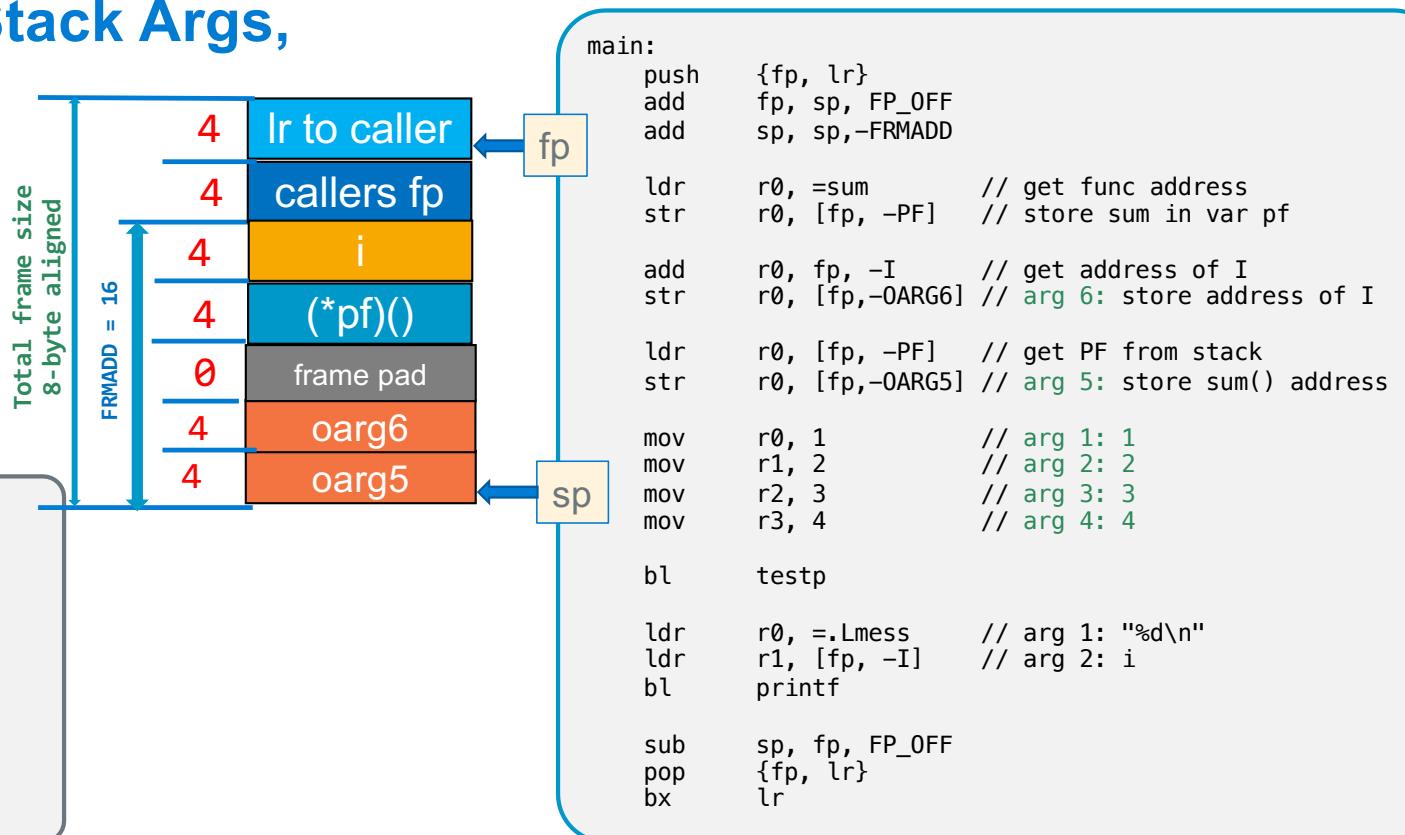
Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

Example: Passing Stack Args, Calling Function

```
int main()
{
    int i;
    int (*pf)(int, int) = sum;

    testp(1, 2, 3, 4, pf, &i);
    printf("%d\n", i);
    return EXIT_SUCCESS;
}
```

```
.section .rodata
.Lmess: .string "%d\n"
// other stuff not shown
.equ FP_OFFSET, 4
.equ I, 4 + FP_OFFSET
.equ PF, 4 + I
.equ PAD, 0 + PF
.equ OARG6, 4 + PAD
.equ OARG5, 4 + OARG6
.equ FRMADD, OARG5 - FP_OFFSET
// FRMADD = 20 - 4 = 16
```



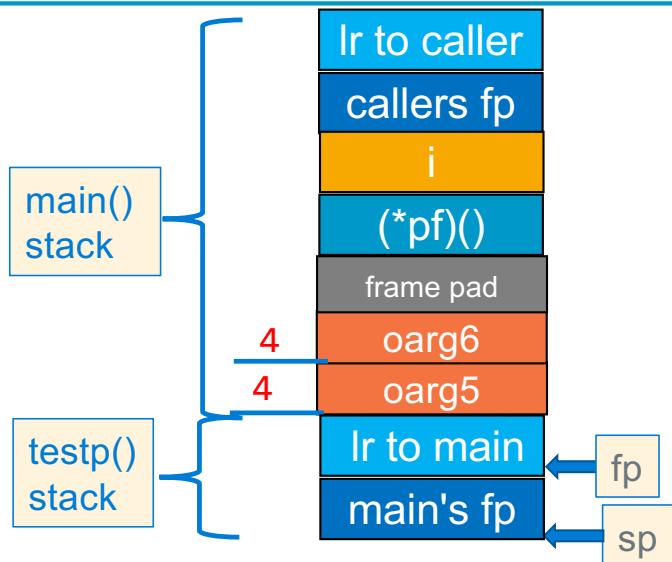
Variable or Argument	distance from fp	Address on Stack	Read variable	Write Variable
int i	I	add r0, fp, -I	ldr r0, [fp, -I]	str r0, [fp, -I]
int (*pf)()	PF	add r0, fp, -PF	ldr r0, [fp, -PF]	str r0, [fp, -PF]
int oarg6	OARG6	add r0, fp, -OARG6	ldr r0, [fp, -OARG6]	str r0, [fp, -OARG6]
int oarg5	OARG5	add r0, fp, -OARG5	ldr r0, [fp, -OARG5]	str r0, [fp, -OARG5]

Example: Passing Stack Args, Called Function

```

void testp(int j, int k, int l, int m, int (*func)(int, int), int *i)
{
    *i = func(j, k) + func(l, m);
    return;
}
  
```

short circuit: make this call first



```

.equ FP_0FF, 20
.equ ARG6, 8
.equ ARG5, 4
testp:
    push {r4-r7, fp, lr}
    add fp, sp, FP_0FF

    mov r4, r2 // save l
    mov r5, r3 // save m
    ldr r6, [fp, ARG5] // load func
    ldr r7, [fp, ARG6] // load i
    blx r6 // r0 = func(j, k)

    mov r1, r5 // arg 2 saved m
    mov r5, r0 // save func return value
    mov r0, r4 // arg 1 saved l
    blx r6 // r0 = func(l, m)
    add r0, r0, r5 // func(l,m) + func(j,k)
    str r0, [r7] // store sum to *i

    sub sp, fp, FP_0FF
    pop {r4-r7, fp, lr}
    bx lr
  
```

Argument	distance	Address on Stack	Read variable	Write Variable
int *i	ARG6	add r0, fp, ARG6	ldr r0, [fp, ARG6]	str r0, [fp, ARG6]
int (*pf)()	ARG5	add r0, fp, ARG5	ldr r0, [fp, ARG5]	str r0, [fp, ARG5]