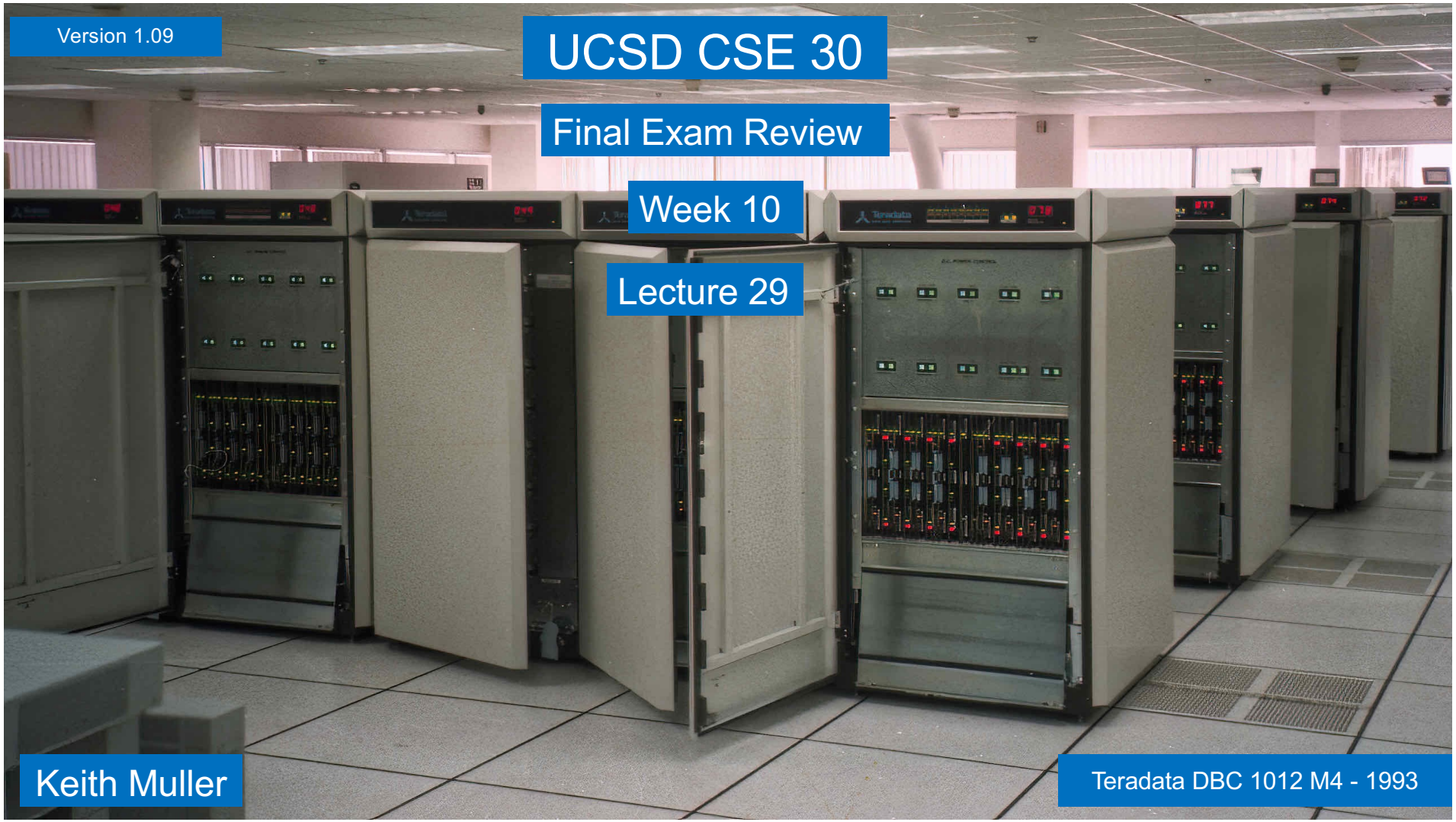
Version 1.09

UCSD CSE 30

Final Exam Review

Week 10

Lecture 29



Keith Muller

Teradata DBC 1012 M4 - 1993

Exam Logistics

- **Make sure you study the most recent versions of the slides!**
- **The final exam will be as scheduled Thursday June 9 11:30 AM – 2:29 PM**
- **The final will be in two rooms**
  - **The Jeannie**
  - **Mosaic 0113**
- We will be assigning students across the two rooms to keep students spread out (we should be under 50% occupancy for each room).
- **The room/seat will be e-mailed to you on Wed June 8.**
  - If you do not receive your assignment (or you forget) , no worries, we will seat you at the exam.
  - Just find me or one of the proctors for help. I will be outside of Jeannie prior to the exam.
- **The exam is no electronic devices** (please leave them at home or turned OFF).
  - Exam questions are being designed to focus on concepts and to minimize the potential for math errors).

Exam Logistics

- Bring pencil(s) and a good eraser
- The exam is mostly multiple choice (fill in a bubble) with a few fill in the blanks
- The exam is open notes. To keep things fair for all students, notes are defined to be:
- **Paper size (one of):**
  - US Standard 8 1/2 inch x 11 inch paper
  - A4 (210 x 297 mm)

  US standard 9 inch x 12 inch paper
- **Page limits**
  - Hand-written (by your hand not printed): 20 sheets of paper (both sides - total of 40 sides)
  - Printed: 10 sheets of paper (both sides - total of 20 sides) - including lecture slides
  - If you have a combination of printed and hand-written sheets:
    - each printed sheet is equal to two handwritten sheets.
- We will provide the arm instruction list (green card), with the exam and a C precedence chart

Help During Final Week

- Edstem/email – I will answer questions as time permits (it may take a couple of hours during the day until I get the final finished and printed)
  - edstem is preferred
- Tuesday Zoom office hours 4-5:30PM
- Check Canvas calendar for other office hours

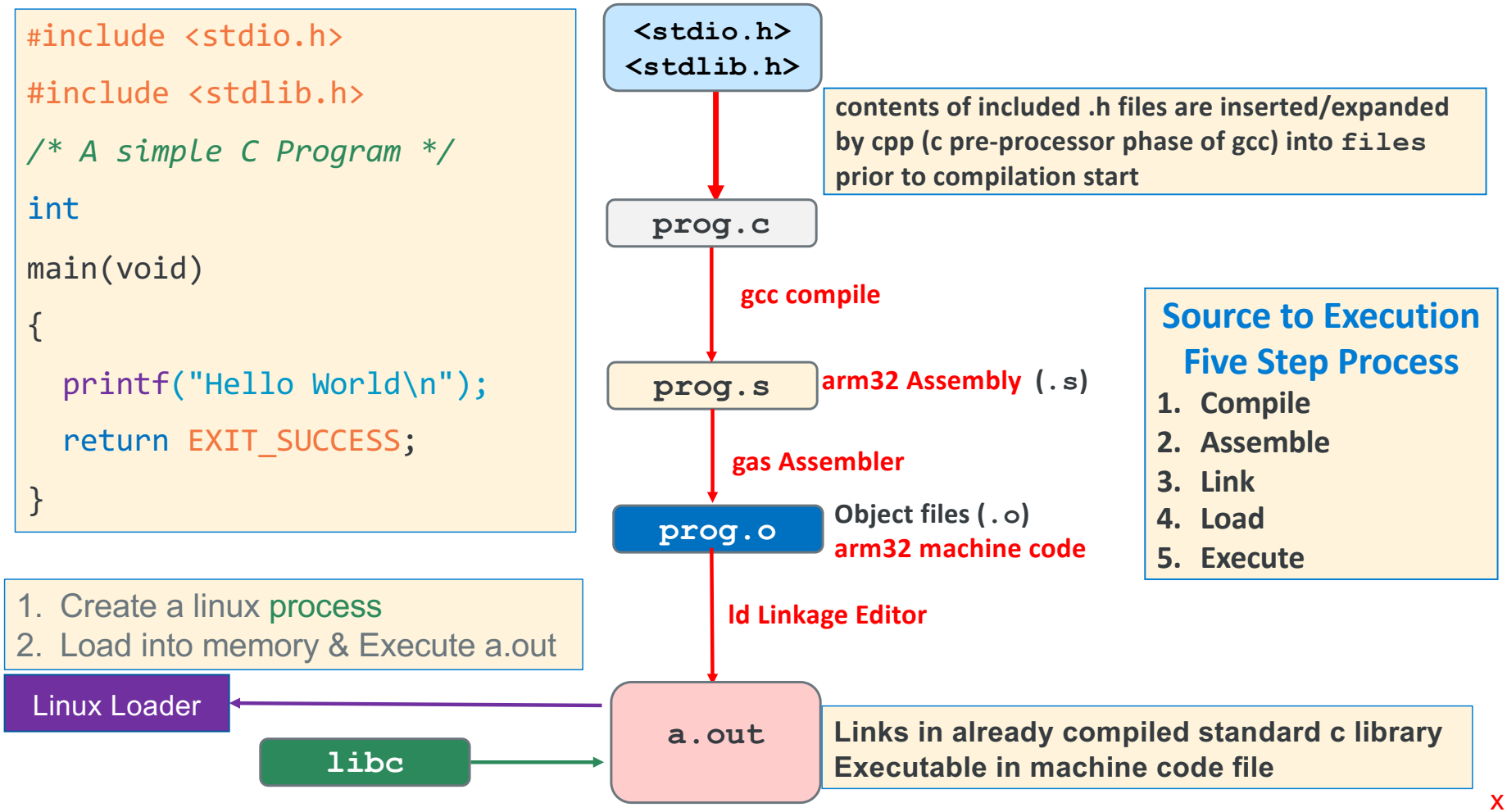
Exam Logistics

- Bring pencil(s) and a good eraser
- The exam is mostly multiple choice (fill in a bubble) with a few fill in the blanks
- The exam is open notes. To keep things fair for all students, notes are defined to be:
- **Paper size (one of):**
  - US Standard 8 1/2 inch x 11 inch paper
  - A4 (210 x 297 mm)

  US standard 9 inch x 12 inch paper
- **Page limits**
  - Hand-written (by your hand not printed): 20 sheets of paper (both sides - total of 40 sides)
  - Printed: 10 sheets of paper (both sides - total of 20 sides) - including lecture slides
  - If you have a combination of printed and hand-written sheets:
    - each printed sheet is equal to two handwritten sheets.
- We will provide the arm instruction list (green card), with the exam and a C precedence chart

Five Step Workflow in the Linux Environment, Single Source File

```c
#include <stdio.h>
#include <stdlib.h>
/* A simple C Program */
int
main(void)
{
  printf("Hello World\n");
  return EXIT_SUCCESS;
}
```



Source to Execution Five Step Process

1. Compile
2. Assemble
3. Link
4. Load
5. Execute

C Library Function API : Simple Character I/O

Operation	Usage Examples
Write a char	`int status; int c;` `status = putchar(c);  /* Writes to screen stdout */`
Read a char	`int c;` `c = getchar();  /* Reads from keyboard stdin */`

```
#include <stdio.h>  // import the API declarations
```

```
int putchar(int c);
```

- writes c (converted to a char) **to** `stdout`
- returns either: c on success *OR* EOF (a macro often defined as -1) on failure
- see man 3 putchar

```
int getchar(void);
```

- returns the next input character (if present) **converted to an int** read **from** `stdin`
- see man 3 getchar

- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition – see later slides** (-1 is not a valid char)

x

C Library Function API : Simple Character I/O

Operation	Usage Examples
Write a char	`int status; int c;` `status = putchar(c);  /* Writes to screen stdout */`
Read a char	`int c;` `c = getchar();  /* Reads from keyboard stdin */`

```
#include <stdio.h>  // import the API declarations
```

`int putchar(int c);`

- writes c (converted to a char) **to `stdout`**
- returns either: c on success *OR* EOF (a macro often defined as -1) on failure
- see man 3 putchar

`int getchar(void);`

- returns the next input character (if present) **converted to an int** read **from `stdin`**
- see man 3 getchar

- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition – see later slides** (-1 is not a valid char)

x

What is a Definition in C?

- **Definition**: creates an instance of a *thing*
- There **must be exactly one** definition of each *function or variable* (no duplicates)
- In C you must define a variable or a function **before first use** in your code

- **Function definitions**
  1. **creates code** as specified by the body
  2. **allocates** memory for the code
  3. **binds** the function name to the allocated memory
- **Variable definitions**
  1. **allocates memory:** at compile time (global vars) or at runtime (local vars)
  2. **initialize memory:** (global vars) or create code to initialize the memory (local vars)
  3. **binds (or associates)** the variable name to the allocated memory

x

What is a Declaration in C?

- **Declaration**: describes a *thing* – specifies types, does not create an instance
- **Function prototype** describes
  - The return type
  - The types of the parameters
- **Variable declaration** describes
  - The variable is defined elsewhere
- **Derived and defined type description**
  - Later in course:(struct, arrays, unions, enums)
- In C, you must **declare a function or variable before you use it**
  - Use before declaration will implicitly default to int (and a compiler warning/error – not good)
- An **identifier** can be **declared multiple times**, but **only defined once**
- A **definition is also a declaration in C**

x

Programming Language Concepts: Scope

- **Scope**: Range (or the extent) of instructions over which a name/identifier can be referenced with C instructions
  1. **File Scope:** Within a single source file (also called a translation unit)
  2. **Block Scope:** Within an enclosing block (variables only)
  3. **Function Scope:** Within the enclosing function (goto label)

```
int global0;            /* global variable file scope */
void                    /* function with file scope */
foo(int parm)           /* parameter parm block scope to function foo */
{
    int i, j = 5;                    /*  block scope – entire function */
    for (int i = 0; i < 10; i++) { // I has block scope for; hides outer i
       label:                       /* function scope not block scope */
 }
}
```


x

Programming Language Concepts: Lifetime

- **Lifetime (Or Storage Duration)**
  - **Duration** in terms of program execution is where the contents of a variable is valid to reference in a C statement (by C language specs – not the OS!)
- **Important:** Linux may allow access to a memory location *even though* the language says you cannot reference the variable (local variables in particular) – later in course when we talk about ***aliases***
  - ***This is core concept behind many security exploits***

x

C Storage Durations

- C variables have one of the following lifetimes (durations)
  1. **Static Storage Lifetime:** valid while program is executing
     - Storage allocated and is initialized prior to runtime (implicit default is 0)
  2. **Automatic Storage Lifetime:** valid while enclosing block is activated
     - Storage allocated and is not implicitly initialized (garbage) by executing code when entering scope
  3. **Allocated Storage Lifetime:** valid from point of allocation until freed
     - Storage allocated by call to an allocator function (malloc() etc.) at runtime and is not implicitly initialized (garbage) - one allocator does initialize to zero at runtime calloc() – later in course
  4. **Thread Storage Lifetime:** valid while thread is executing (not CSE 30)

x

C Variable Definitions

- **Global variables:** Defined **outside a function body**
  - Scope: valid from **point of definition** to end of file
  - **Static storage lifetime -** valid while program is executing
- **Local Variables & Function Parameters**: Defined **within a block**
  - Scope: valid from **point of definition** to the end of the code block where defined
  - **Lifetime: Automatic storage lifetime -** valid while enclosing block is activated

```
int global0;              /* global variable default initial value is 0 */
int global1 = 1;          /* global variable explicitly set to 1 */
void foo(int parm)        /* automatic parameter parm is "local" to foo */
{
    int i, j = 5;        /* automatics i initial value is unknown, j is 5 */
    const int n = 5;     /* automatic value of n cannot change */
    static int s;        /* block scope, static lifetime initial value: 0 */
    // body of code
}
```


x

Static Storage Duration

- Variables defined **with the storage class specifier** ***static*** have static storage duration including variables with block scope
- Global variables **without the storage class specifier** ***static*** also have static storage duration
- All variables with static storage duration are allocated space and initialized before execution starts (default is 0)

```
% ./a.out
2 3 4 5 6
%
```

```
#include <stdio.h>
#include <stdlib.h>
int unused; //global static storage duration
int
foo(void)
{
    static int s=1; //static storage duration
    return s += 1;
}

int main(void)
{
    for (int i = 0; i < 5; i++)
        printf("%d ", foo());
    return EXIT_SUCCESS;
}
```


x

C Function Definitions - 1

- **C Functions are not methods**
  - no classes, no objects
- **C function definition**
  - returns a value of `returnType`
  - zero or more ***typed*** parameters
- Every program must have a main() function
- main() is the first function in your code to run/execute
  - main() is **not the first code** to run in a Linux process, it is called by the C runtime startup code
    - later in course

```
returnType fname(type param1, …, type paramN)
{
   // statements
   return value;
}
```

function definition

```
// returns: sum of integers from 1 to max
int
sum(int max)        // function definition
{
  int i, sum = 0;   // variable def

  for (i = 1; i <= max; i++) {
    sum += i;
  }

  return sum;
}
```


x

Full Example: Function Prototypes

```c
// sum of integers from 1 to MAX
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int sum(int max)
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}

int main(void)
{
  printf("sum(%d): %d\n", MAX, sum(MAX));
  return EXIT_SUCCESS;
}
```

```c
// sum of integers from 1 to MAX
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int sum(int);  // func prototype

int main(void)
{
  printf("sum(%d): %d\n", MAX, sum(MAX));
  return EXIT_SUCCESS;
}

int sum(int max)
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

Using prototypes, the order of definition does not matter

x

Function Ordering In a Source File: Must be Declared Before Use (You can define in another file)

sum() is defined and declared here

Definitions and declarations are valid

1. Only in the file (also called a translation unit) where they are located **and**
2. From the point of use to the end of the file (translation unit)

sum() is used here

```c
// sum of integers from 1 to max
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int sum(int max)
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}

int main(void)
{
  printf("sum(8): %d\n", sum(MAX));
  return EXIT_SUCCESS;
}
```

Relaxing Ordering Restrictions: Function Prototype Declarations

```
returnType fname(type_1, …, type_n);
```

function prototype

- A **function prototype declaration** is
  - function header, followed by a semicolon (;) instead of **a code block**
  - It specifies the function type (the return type) and parameter types)
- Declarations enable usage at a point in the file when it is
  - defined later (after) in the **same** source file **or**
  - defined in a **different** source file

sum() declared here

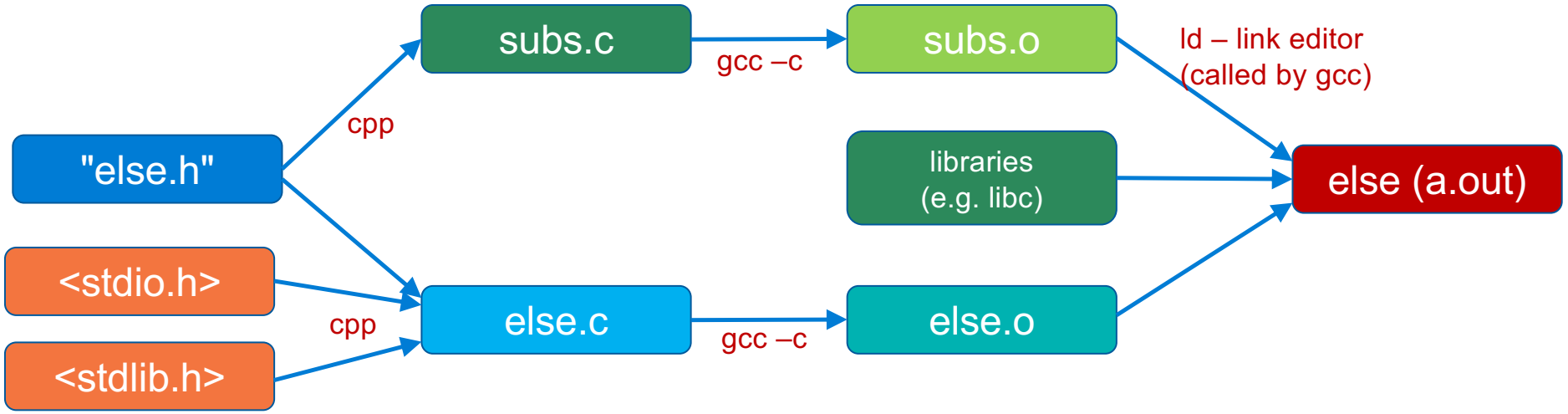
sum() defined here

code block

```
int sum(int);       // function prototype - declaration
// other stuff
int sum(int max)    // function definition
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```


x

Compiling Multi-File Programs



1. compile each .c file independently to a .o object file (incomplete machine code)

```
gcc -Wall -Wextra -Werror -c subs.c  # creates subs.o
gcc -Wall -Wextra -Werror -c else.c  # creates else.o
```

2. link all the .o objects files and library's (aggregation of multiple .o files) to produce an executable file (complete machine code) (gcc calls ld, the linker)

```
gcc -Wall -Wextra -Werror else.o subs.o -o else
```


x

Controlling Linkage Across Files in Multi-File C Programs

- Linkage determines whether an object (like a variable or a function) can be referenced outside the file it defined in
- External Linkage: function and variables with external linkage can be referenced anywhere in the entire program
  - Global variables and functions have external linkage by **default**
- Internal Linkage: function and global variables with internal linkage can only be referenced in the same file
  - Global variables and functions can be changed to internal linkage by using the keyword static
- No Linkage across files: function parameters, variables defined inside a function (without a keyword extern - declaration)

x

Reference Slide: Scope and Lifetime Summary

where defined	definition type	definition preceded with ***static*** keyword	name exported for linking to other files	definition category	scope	lifetime (of variables)	Default Initial Value	Where located in system memory
inside function or block	variable	no	no	local automatic variable (no linkage)	enclosing function or block	enclosing function or block activation	garbage (if initial value given, done by code at runtime)	stack segment
inside function or block	variable	yes	no	function/block private static variable (no linkage)	enclosing function or block	process lifetime	zero; Once before program start	data segment if initial value specified, bss otherwise
outside of function	variable	no	yes	global static variable (external linkage)	file; extendable to other files up to program	process lifetime	zero; Once before program start	data segment if initial value specified, bss otherwise
outside function	variable	yes	no	file private global static variable (internal linkage)	file	process lifetime	zero; Once before program start	data segment if initial value specified, bss otherwise
	function	no	yes	function (external linkage)	file; extendable to other files (up to program)			text
	function	yes	no	file private (internal linkage)	file			text

Representing DFA states: C Enumerated Data Type

```
enum tag {enum_0, enum_1, …, enum_n};      // defines just the type
enum tag var;                              // defines a variable instance
```

```
enum suit {CLUBS, DIAMOND, HEARTS, SPADES};  // CLUBS=0, DIAMOND=1, etc
enum suit card = HEARTS;                     // card defined & initialized to HEARTS
```

- **Enumeration identifiers** are really integer constants and can be used as such
- **Default:** compiler assigns a sequential integer value to each name, starting with 0 and incrementing from there 0, 1, 2, 3, …
- **Over-ride** the default value by specifying a unique integer value for each enumeration name

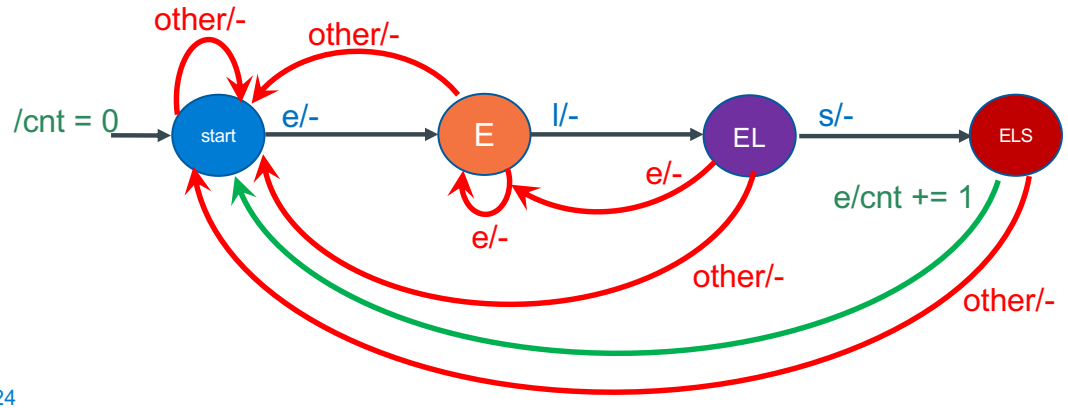
```
enum compass {NORTH=0, EAST=90, SOUTH=180, WEST=270};
enum compass direction = WEST; // direction defined & initialized to WEST
```


x

Programming a Deterministic Finite Automaton - 2

```c
#include <stdio.h>
#include <stdlib.h>
enum typestate {START, E, EL, ELS};
/* prototypes here */
enum typestate STARTst(int);
enum typestate Est(int);
enum typestate ELst(int);

int main(void)
{
    int c;
    int cnt = 0; // else counter
    enum typestate state = START;
```

file: else.c
(more next slide)

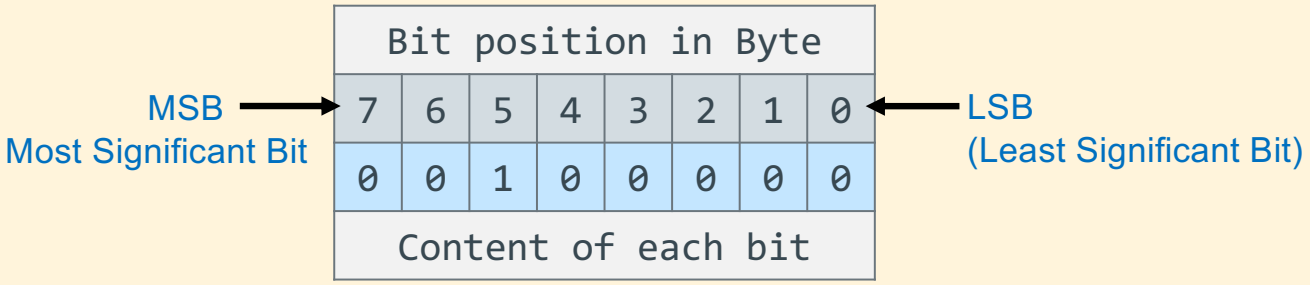



```c
    while ((c = getchar()) != EOF) {
        switch (state) {
        case START:  // nothing
            state = STARTst(c);
            break;
        case E: // saw an e
            state = Est(c);
            break;
        case EL: // saw an el
            state = Elst(c);
            break;
        case ELS: //saw an els
            state = START;
            if (c == 'e')
                cnt += 1;
            break;
        }
    }
    printf("cnt = %d\n", cnt);
    if (cnt > 0)
        return EXIT_SUCCESS;
    return EXIT_FAILURE;
}
```

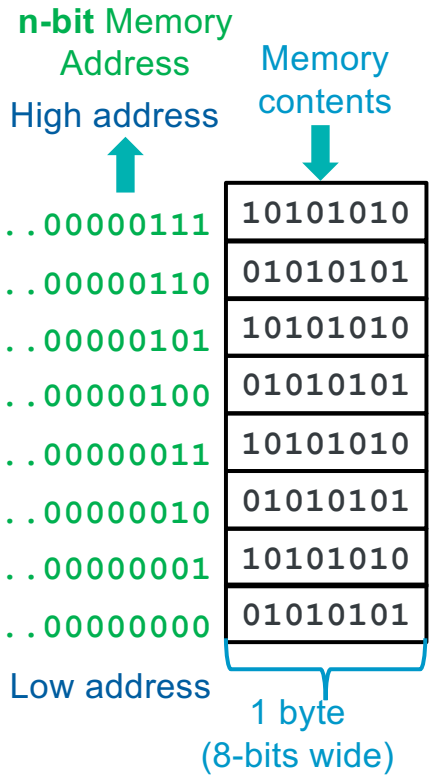
x

Memory Review: Organized in Units of Bytes

- One bit (digit) of storage (in memory) has two possible **states**: 0 or 1
- Memory is organized into a **fixed unit** of 8 bits, called a **byte**



- Conceptually, memory is a single, **large array** of **bytes**, **where each byte** has a unique *address (byte addressable memory)*
- An address is an **unsigned** (positive #) *fixed-length* n-bit binary value
  - Range (domain) of possible addresses = *address space*
- Each byte in memory can be **individually accessed** and operated on given its **unique address**

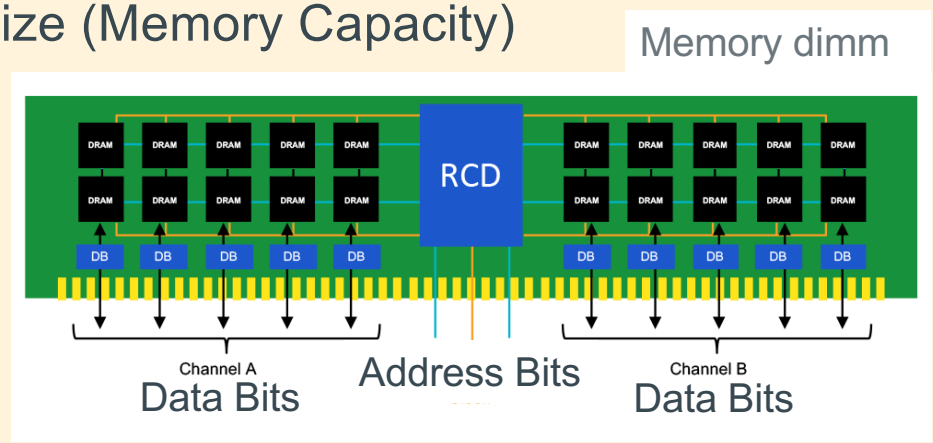


x

Memory Size

- **Since memory addresses are implemented in hardware using binary**
  - The **Size (number of byte sized cells)** of Memory is specified in **powers of 2**
- Memory size/capacity in **bytes** is specified by the "Number of bits" in an address
  - 32 bits of address = 2^{32} = 4,294,967,296
  - Address Range is 0 to $2^{32} - 1$ (unsigned)

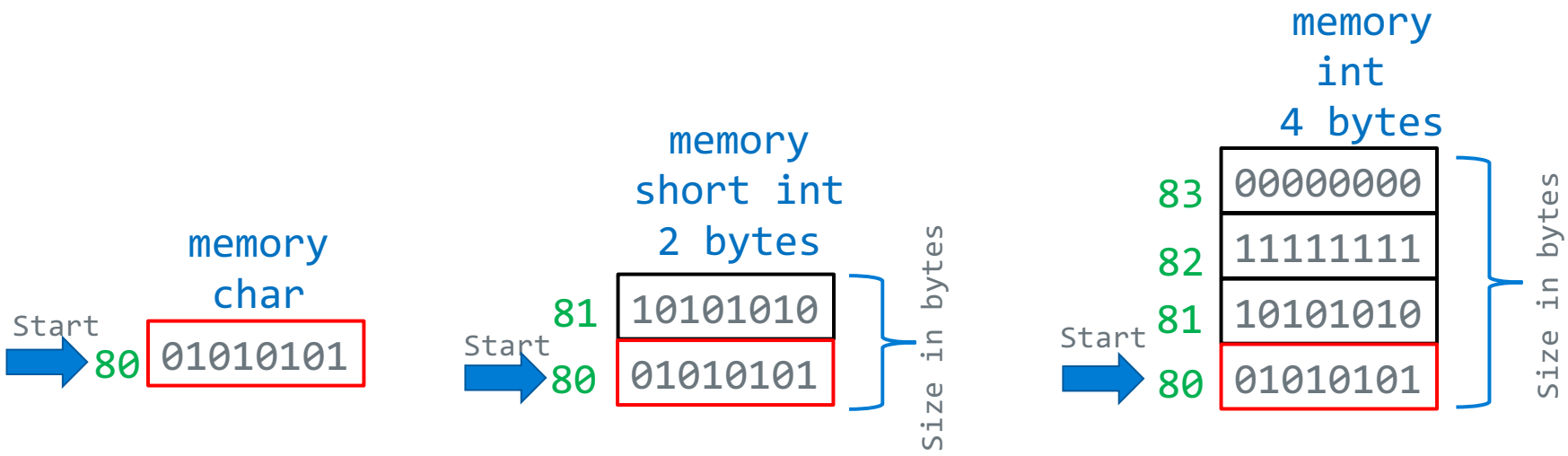
- Shorthand notation for address size (Memory Capacity)
  - KB = 2^{10} (K=1024) kilobyte
  - MB = 2^{20} megabyte
  - GB = 2^{30} gigabyte
  - TB = 2^{40} terabyte
  - PB = 2^{50} petabyte



x

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



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

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

sizeof() operator returns:

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:

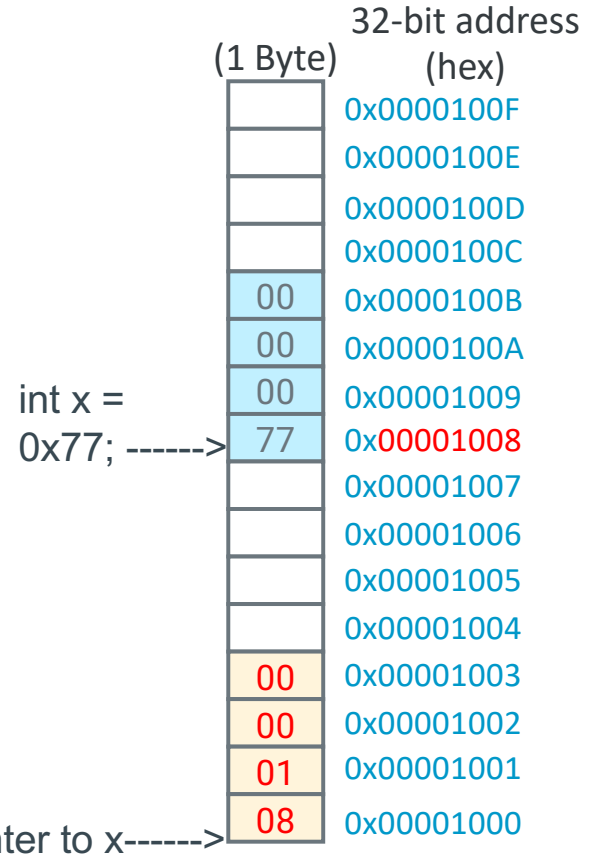
```
size = sizeof(int * 10);
```

  - reads as:
    - number of bytes required to store **10 integers (an array of [10])**

x

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 0x00001008
- The variable pt is at memory location 0x00001000
- The contents of pt is the address of x 0x00001008



x

Variables: Size

- **Integer types**
  - **`char`**, **`int`**
- **Floating Point**
  - **`float`**, **`double`**
- Modifiers for each base type
  - **`short`** [int]
  - **`long`** [int, double]
  - **`signed`** [char, int]
  - **`unsigned`** [char, int]
  - **`const`**: variable read only
- **char type**
  - One byte in a byte addressable memory
  - **Signed** vs **Unsigned** Char implementations
  - **Be careful** char is unsigned on arm and signed on other HW like intel

C Data Type	AArch-32 contiguous Bytes	AArch-64 contiguous Bytes	printf specification
char (arm unsigned)	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	8	16	%Lf
pointer *	4	8	%p

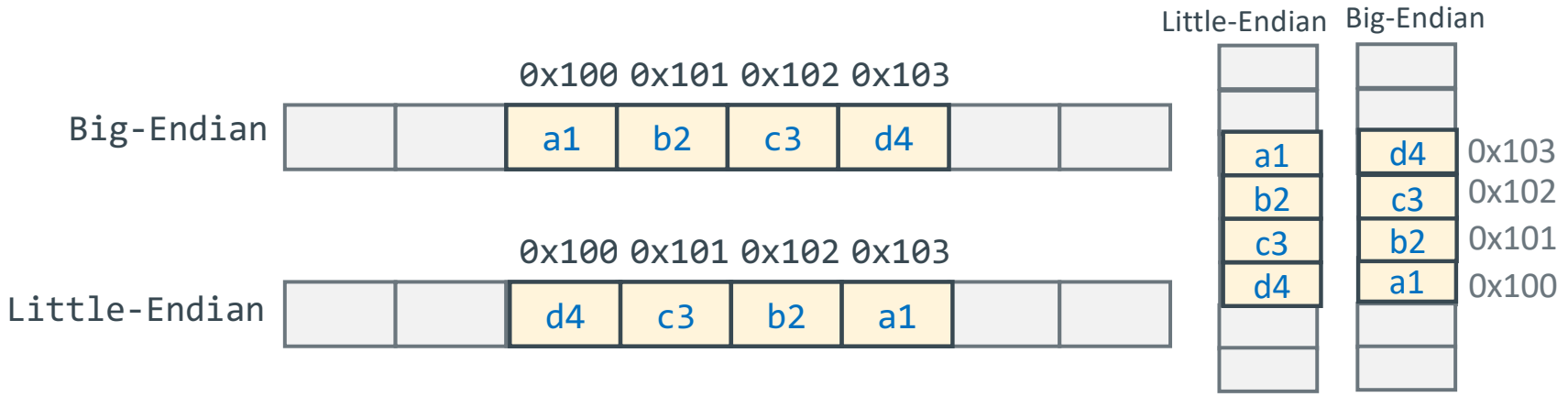
x

Byte Ordering of Numbers In Memory: Endianness

- Two different ways to place multi-byte integers in a byte addressable memory
- Big-endian: Most Significant Byte ("big end") starts at the *lowest (starting)* address
- Little-endian: Least Significant Byte ("little end") starts at the *lowest (starting)* address
- Example: 32-bit integer with 4-byte data `0x a1 b2 c3 d4 at 0x100`

MSB
Most significant byte

LSB
Least significant byte

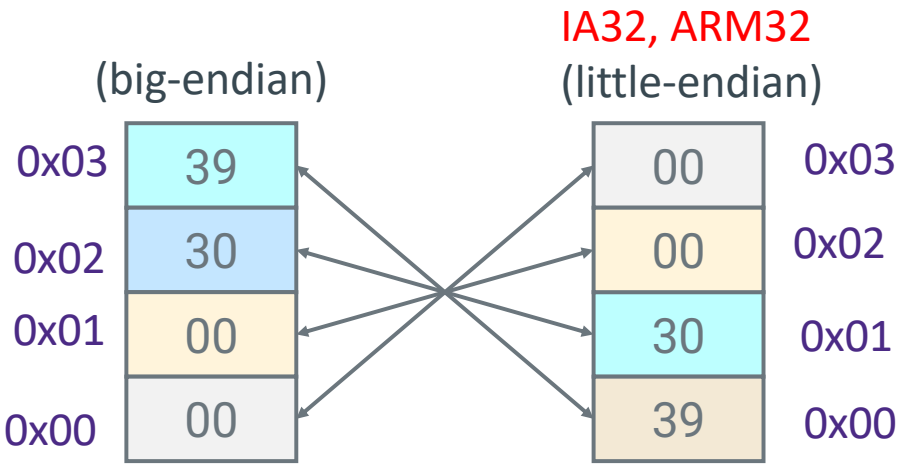


x

Byte Ordering Example

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

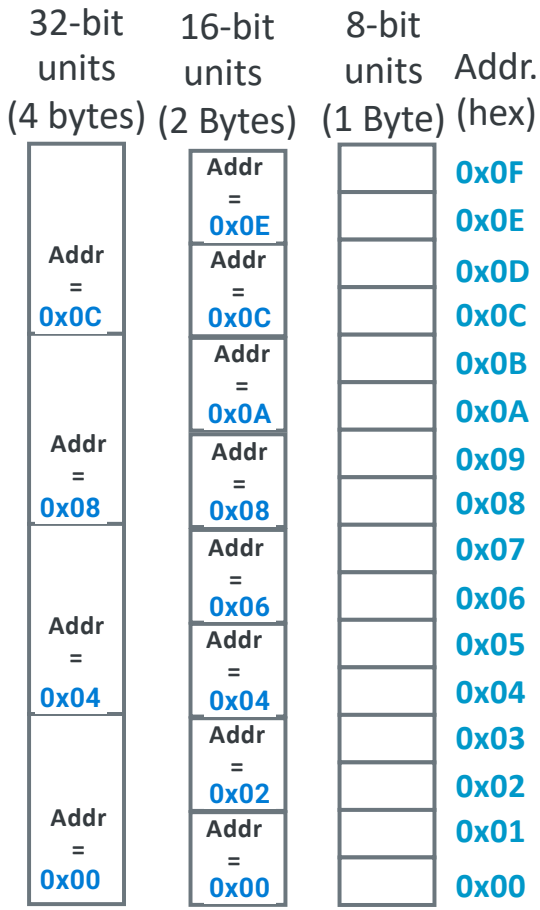
```
int x = 12345;
// or x = 0x3039;
```



x

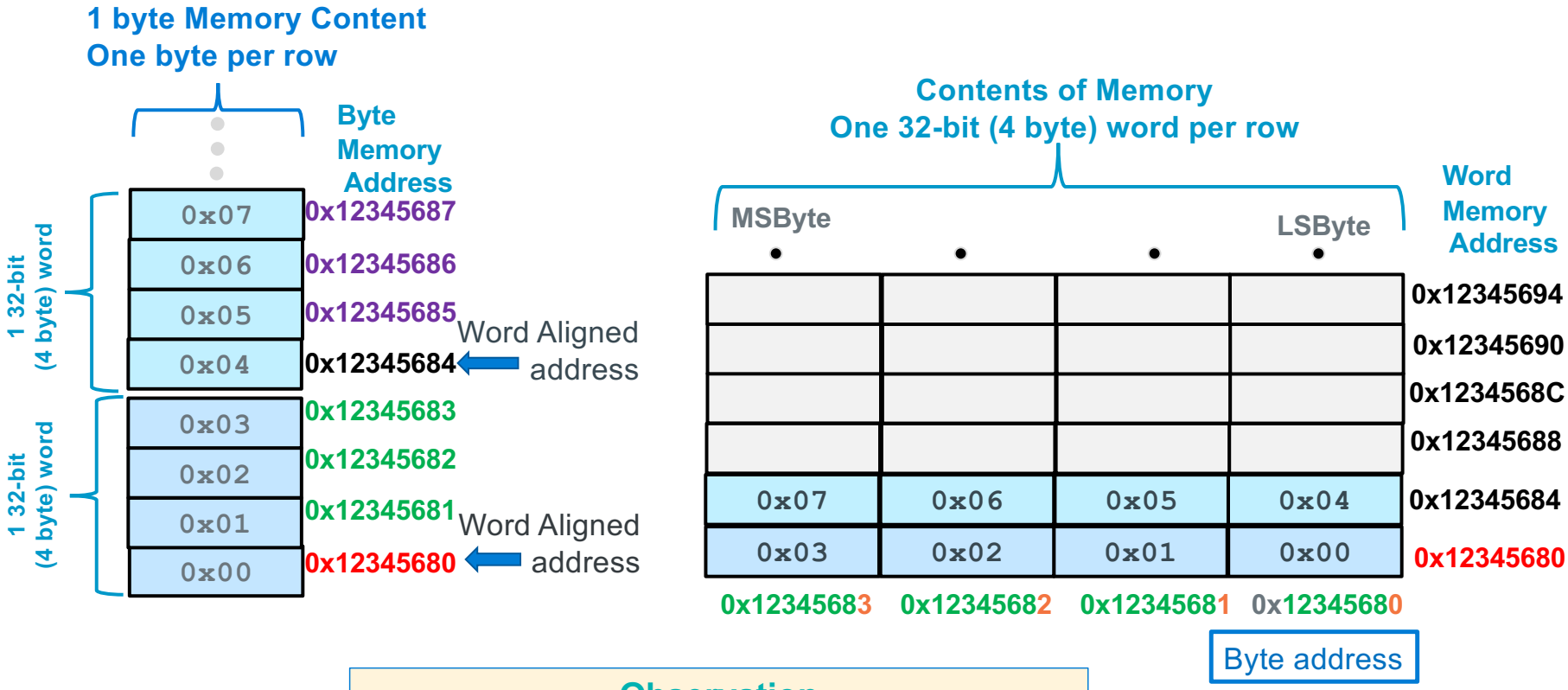
Memory Alignment of Variable and Words

- Variables are *"aligned"* to specific starting addresses based on size
- **Word** is the number of bytes necessary to store an address (32-bits on Pi-cluster) – hardware defined
- The address of *any sized* unit of memory is always the address of the **first byte**
- All variable types **align** to **address 0x00**
  - Successive instances of the same size all adhere to the same alignment (think of an array)
  - `char (1 byte)` can start at any address
  - `int (4 bytes)` can only start at address 0x00, 0x04, 0x08,…



x

Byte Addressable Memory Shown as 32-bit words

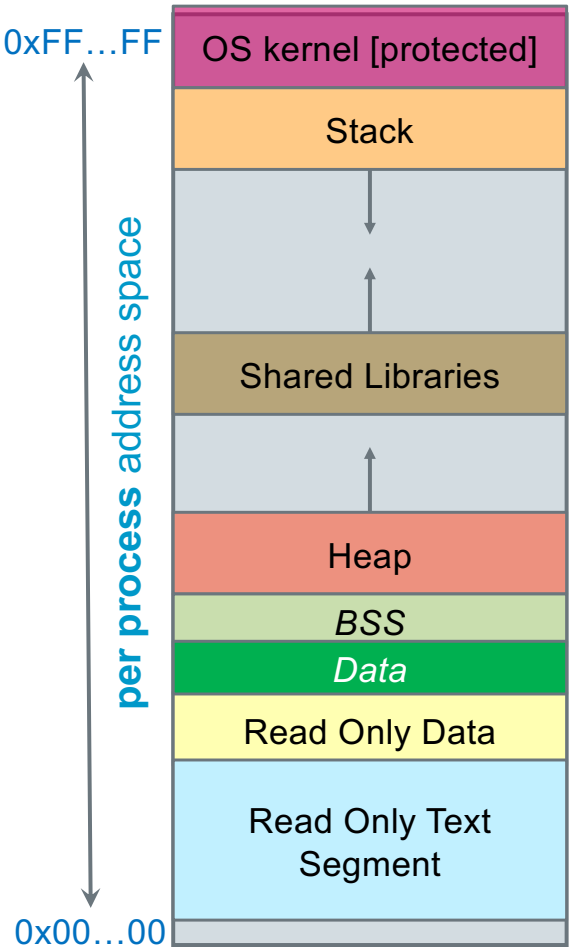


Observation
32-bit aligned addresses
rightmost 2 bits of the address are always 0

x

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



x

Where Variables Reside in Memory

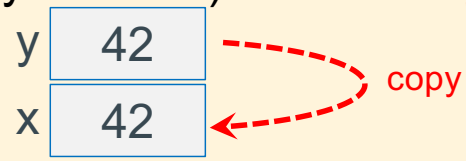
```
int global0 = 1;           // data segment
int global1[100];          // bss segment
static int global2;        // bss segment
int funcA(int b)           // text segment for code in funcA()
{                          // b may be in stack or a CPU register - later
    int x = 3;             // stack segment
    int s;                 // stack segment
    static int z;          // bss segment
    static int w = 1;     // data segment
    for (int j = 0; j < MAX; j++) {    // j in stack segment
        int w;             // stack segment
        printf("Hi\n");    // "Hi\n" literal is in read-only data
    }
/* …. rest of code …  */
```


x

Memory Addresses & Memory Content

- A **variable name** *(by itself)* in a C statement evaluates to either:
  - **Lvalue:** when on the left side (Lside or Left value) of the = sign is the address where it is stored in memory – a constant
  - **Rvalue:** on the right side (Rside or Right value) of an = sign is the contents or value stored in the variable (at its memory address) – a memory read

```
x = y;      // Lvalue = Rvalue
```



- **x** on left side (**Lside**) of the assignment operator = evaluates to:
  - The address of the memory assigned to the x – this is x's **Lvalue**
- **y** on right side (**Rside**) of the assignment operator = evaluates to:
  - READ the contents of the memory assigned to the variable y (type determines length) - this is y's **Rvalue**
- Read memory at y (**Rvalue**); write it to memory at x's address (**Lvalue**)

x

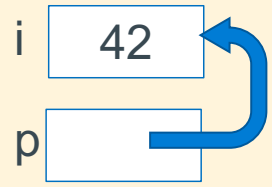
Introduction: Pointer Variables - 1

- In C, there is a *variable* ***type*** for **storing an address**: a ***pointer***
  - **Contents** of a pointer is an **unsigned** (0+ positive numbers) **memory address**
- When the **Rside of a variable** contains a **memory address**, (it **evaluates** to an **address**) the variable is called a **pointer variable**

```
type *name;  // defines a pointer; name contains address of a variable of type
```

- A pointer is defined by placing a ***star (***or ***asterisk)*** **(*)** ***before*** the identifier (name)
- You also must specify the type of variable to which the pointer points

```
int i = 42;
int *p = &i;  /* p "points at" i (assign address of i to p) */
```



- Recommended: be careful when defining multiple pointers on the same line:

```
int *p1, p2;
```

is not the same as

```
int *p1, *p2;
```

Use instead:

```
int *p1;
int *p2;
```


x

Introduction: Pointer Variables - 2

- **Pointers are typed**! Why?
  - Tells the compiler the size (sizeof()) of the data **you are pointing at** (number of bytes to access)
- A pointer definition:

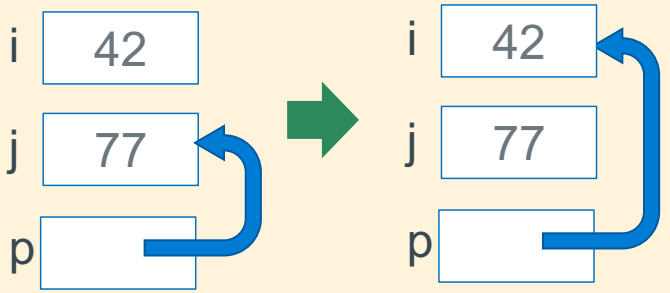
```
int *p = &i;  /* p points at i (assign address i to p) */
```

- Is the same as writing the following definition and assignment statements

```
int *p;       /* p is defined (not initialized) */
p = &i;       /* p points at i (assign address i to p */
```

- The * is part of the definition of p and is not part of the variable name
  - The name of the variable is simply p, not *p
- As with any variable, its value can be changed

```
p = &j;       /* p now points at j */
p = &i;       /* p now points at i */
```



x

Introduction: Address Operator: &

- Unary ***address operator*** (&) produces the **address** of where an identifier is in memory
- Requirement: **identifier must have a Lvalue**
  - Cannot be used with constants (e.g., 12) or expressions (e.g., x + y)
  - `&12` does not have an *Lvalue*, so &12 is **not** a legal expression
- How can I get an address on the **Rside**?
  - **&var** (any variable identifier or name)
  - **function_name** (name of a function, not func()); **&funct_name** is equivalent
  - **array_name** (name of the array like array_name[5]); &array_name is equivalent
- Example: this might print:

  *the **value** of g is: 42*

  *the **address** of g is: 0x71a0a0*

  (the address will vary)

```
int g = 42;
int main(void)
{
    printf("the value of g is: %d\n", g);
    printf("the address of g is: %p\n", &g);
}
```

- *Tip*: The `printf()` format specifier to display an address/pointer (in hex) is "%p"

x

The NULL Pointer

- **NULL** is a pointer **value** to represent that the pointer points to "nothing"
- If the value of a pointer is:
  - Unknown at the time of definition, or
  - When the pointer points at a memory location that becomes *invalid (we will discuss later)*
  - Then it is good style to assign the value of NULL to the pointer
- A pointer with a value of NULL is often called a "NULL pointer" (not a valid address!)
- Many functions return NULL to indicate an error has occurred

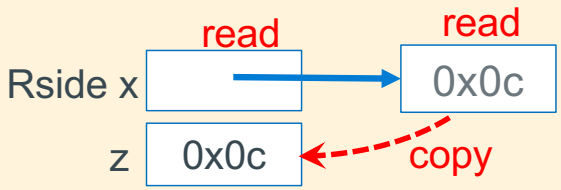
```
int *p = NULL;
int *p = (int *)0;   // cast 0 to a pointer type
int *p = (void *)0;  // automatically gets converted to the correct type
```


x

Introduction: Indirection Operator - 2

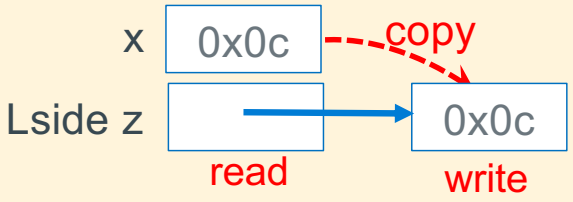
- * on the Rside: read the contents of the variable to get an address and then **read** and return the contents at that address (requires two reads of memory on the Rside)

```
z = *x; // copy the contents of memory pointed at by x to z
```



- * on the Lside: read the contents of the variable to get an address and then **write** the evaluation of the Rside expression to that address (requires one read of memory and one write of memory on the Lside)

```
*z = x; // copy the value of x to the memory pointed at by z
```

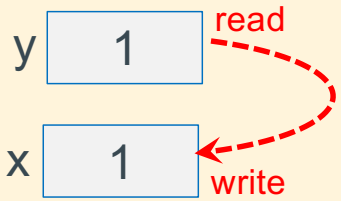


x

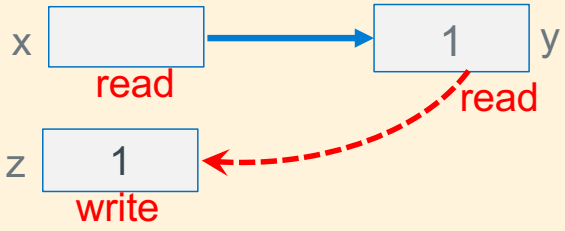
Introduction: Indirection Operator - 3

- Each * when used as a dereference operator in a statement (Lside and Rside) generates an additional read

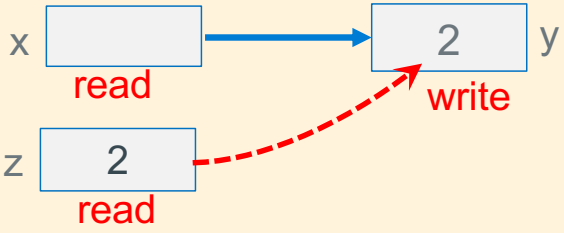
```
int x = 2, y = 1;
x = y;
```



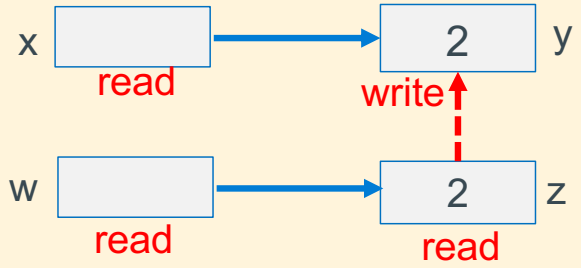

```
int z = 2, y = 1;
int *x = &y;
z = *x;
```




```
int z = 2, y = 1;
int *x = &y;
*x = z;
```




```
int z = 2, y = 1;
int *x = &y;
int *w = &z;
*x = *w;
```

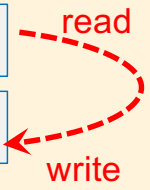


x

Recap: Lside, Rside, Lvalue, Rvalue

```
int x = 2, y = 1;
x = y;
```

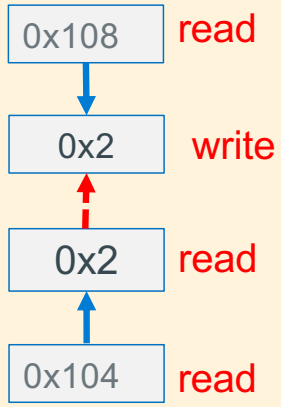
Constant Var Name	Lvalue address	Rvalue Contents
y	0x108	0x1
x	0x104	0x1




```
int z = 2, y = 1;
int *x = &y;
int *w = &z;
*x = *w;
```

```
*x on Lside is   0x10c
w  on Rside is   0x100
*w on Rside is   2
```

Constant Var Name	Lvalue address	Rvalue Contents
x	0x10c	0x108
y	0x108	0x2
z	0x104	0x2
w	0x100	0x104



x

Pointer to Pointers (Double, Triple and … Indirection

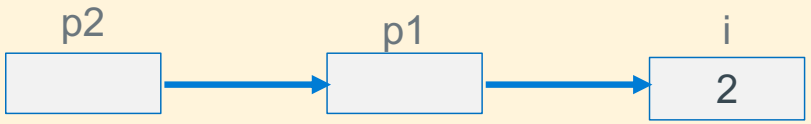
- A pointer **cannot** point at itself, why?

```
int *p = &p; /* is not legal – type mismatch */
```

  - p is defined as (int *), a pointer to an int, **but**
  - the type of &p is (int **), a pointer to a pointer to an int

- Define a pointer to a pointer (p2 below)

```
int i = 2;
int *p1;
int **p2;
p1 = &i;
p2 = &p1;
printf("%d\n", **p2 * **p2);
```



number of * in the definition tells you how many reads it takes to get to the base type
reads = number of * + 1
e.g., int **p2 requires 3 reads to get to the int

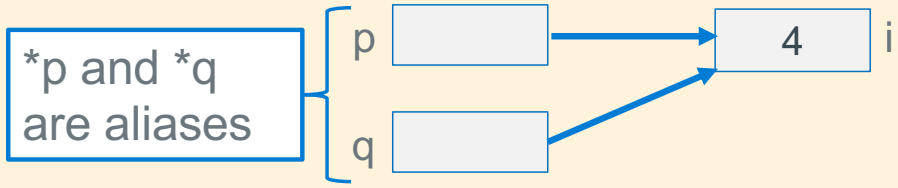
- C allows any number of pointer indirections
  - more than three levels is very uncommon in real applications as it reduces readability and generates at lot of memory reads

x

What is Aliasing?

- Two or more variables are aliases of each other when they all reference the same memory
- When one pointer is copied to another pointer it *creates an **alias***
- ***Side effect***: Changing one variable changes the value for another variable
  - Multiple variables all read and write the **same** memory location
  - Side effects can be by accident (coding errors) or deliberate (careful: readability)

```
int i = 5;
int *p = &i;
int *q;

q = p;   // *p & *q are aliases
*q = 4;  // changes i
```



x

Background: Different Ways to Pass Parameters - 1

- **Call-by-reference (or pass by reference) (Not in C, Java, or C++)**
  - Parameter in the called function is an ***alias*** (references the same memory location) for the supplied argument
  - Modifying the parameter modifies the calling argument

- **Call-by-value** (or pass by value) (C, Java, C++)
  - **Calling** Function does:
    - Evaluates the parameter *expressions* for each argument to obtain the *value*
    - Makes a **copy** of the evaluated parameter (the ***value*** in call by **value**)
    - The copy is used by the called function
    - **Implementation**: The caller allocates memory for the copy and the *value of the whole expression* is stored in the contents of that memory location

x

Background: Different Ways to Pass Parameters - 2

Call-by-value (or pass by value) (C, Java, C++)

- What **Called** Function Does
  - Passed Parameters are used like local variables
  - Modifying the passed parameter in the function is allowed just like a local variable
  - So, writing to the parameter, ***only*** changes the ***copy***
- The return value from a function in C is **by value**
- The memory (or location) of the return value (the expression with the return statement) is considered part of the caller's scope
  - very implementation dependent
  - Example: ARM32 uses registers (later in the course)

x

Function Output Parameters: Passing Pointers

- 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**
- Enables additional *values to be returned (besides the return)* from a function call

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

- With a pointer to x, inc() can change x in main()
  - This is called a *side-effect*
- inc() can also change the *value* of p, the copy, just like any other parameter

- 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

x

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
}
```


x

Arrays in C - 1

Definition: `type name[count]`

- ***"Compound"*** data type where each value in an array is an element of type
- Above allocates **name** with a *fixed* count array elements of type **type**
- **Arrays are indexed starting with 0**
- Allocates (`count * sizeof(type)`) bytes of ***contiguous memory***
- Common usage is to specify a compile-time constant for `count`

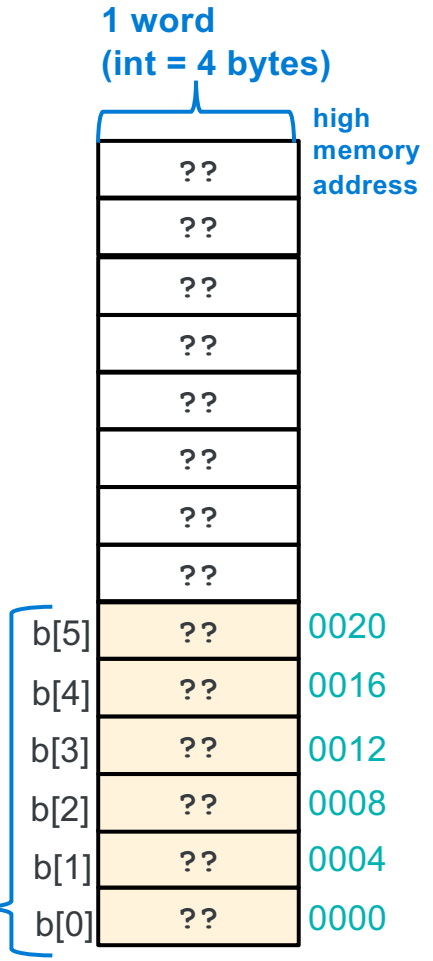
```
#define BSZ   6
int b[BSZ];
```

BSZ is a macro replaced by the C preprocessor before compilation starts

- **Size (bytes or element count) of an array** is not stored anywhere!!!!!!
  - **An array does not know its own size!**
  - `sizeof(array)` **only works** in **scope** of array variable definition
- automatic (only) variable-length arrays (sized at runtime):

```
/* VLA only in block scope – automatics */
int func (int n) {
    int scores[n];  // these are not widely used!
```

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



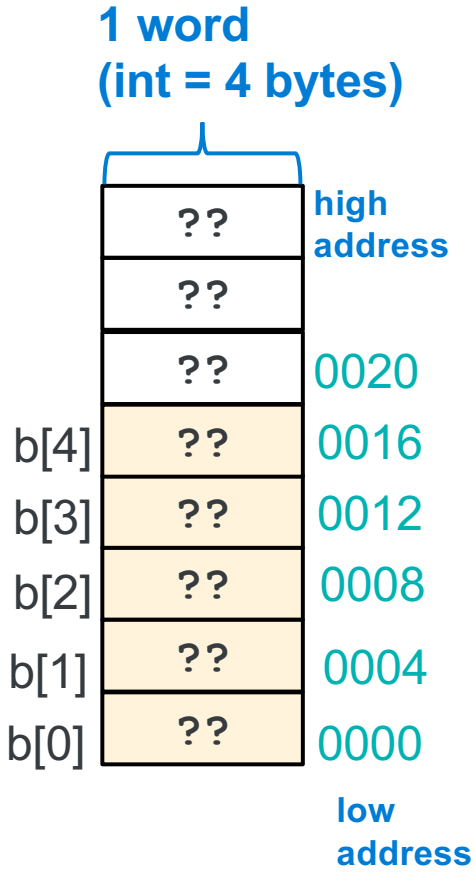
Arrays In C - 2

- **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[5];
int b[5];
```

- Array name (by itself with no [ ]) on the Rside evaluates to the address of the first element of the array
- Array **names are constants (like all variable names)** and cannot be assigned (cannot appear on the Lside by themself)

```
a = b;        // invalid does not copy the array
              // copy arrays element by element
```



x

Arrays in C - 3

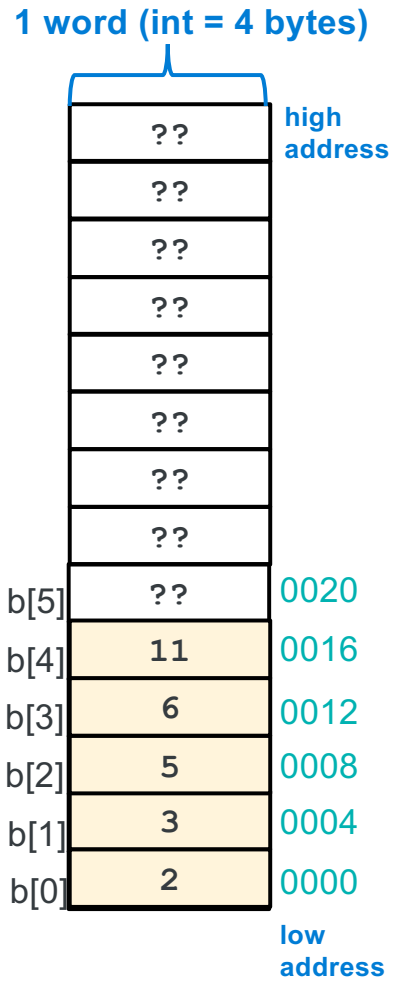
- Initialization: `type name[count] = {val0,…,valN};`
  - `{ }` *(optional)* initialization list can *only* be used at **time** of **definition**
  - If no `count` supplied, `count` is determined by compiler using the number of array initializers

    no initialization values given; then elements are initialized to 0
  - `int block[20] = {}; //only works with constant size arrays`
    - defines an **array of 20 integers** each element filled with zeros
    - Performance comment: do not zero automatic arrays unless really needed!
  - When a **count** is given:
    - **extra** *initialization values* are **ignored**
    - **missing** *initialization values* are set to **zero**

```
int block[5] = {2, 3, 5, 6, 11, 13};
```

not needed and if used **may** truncate initialization list

6 initialization values given, **only 5 are used**



So, How Big is My Array?

```
// defining array with a fixed size use a #define to eliminate embedded "magic" numbers
#define SZ 6
int szblock[SZ];                        // manual: you specify the array has SZ elements
int indx; // use when SZ is defined

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

- Programmatically (and safely) determining the element count in a compiler calculated array

`sizeof(array) / sizeof(of just one element in the array)`

Remember: `sizeof(array)` **only works** in **scope** of the array variable definition

```
#include <stddef.h>
int block[] = {2, 3, 5, 6, 11, 13};      // automatic: compiler calculates array size
int cnt = (int)(sizeof(block) / sizeof(block[0]));       // in this case cnt = 6

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


x

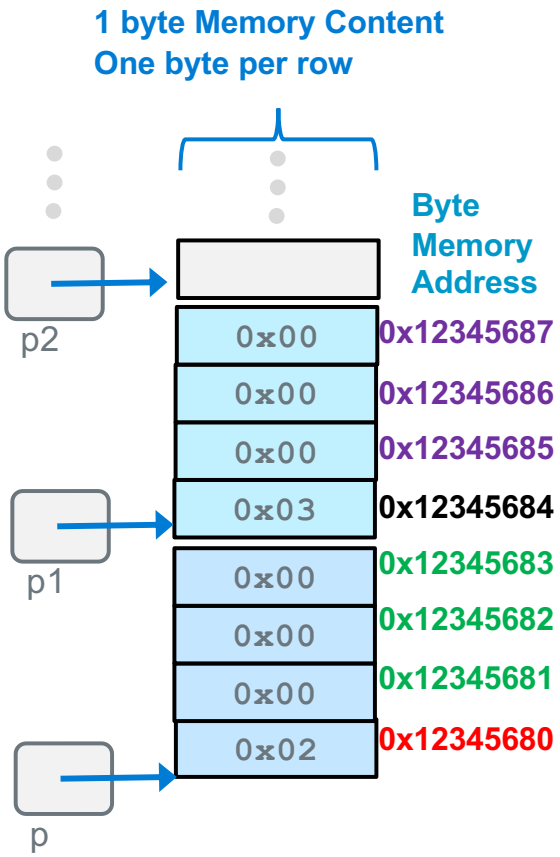
Pointer and Arrays - 1

- 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 point at the first array element

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

*p = *p + 10;
*p1 = *p1 + 10;        // {12, 13, 5, 6, 11}
```



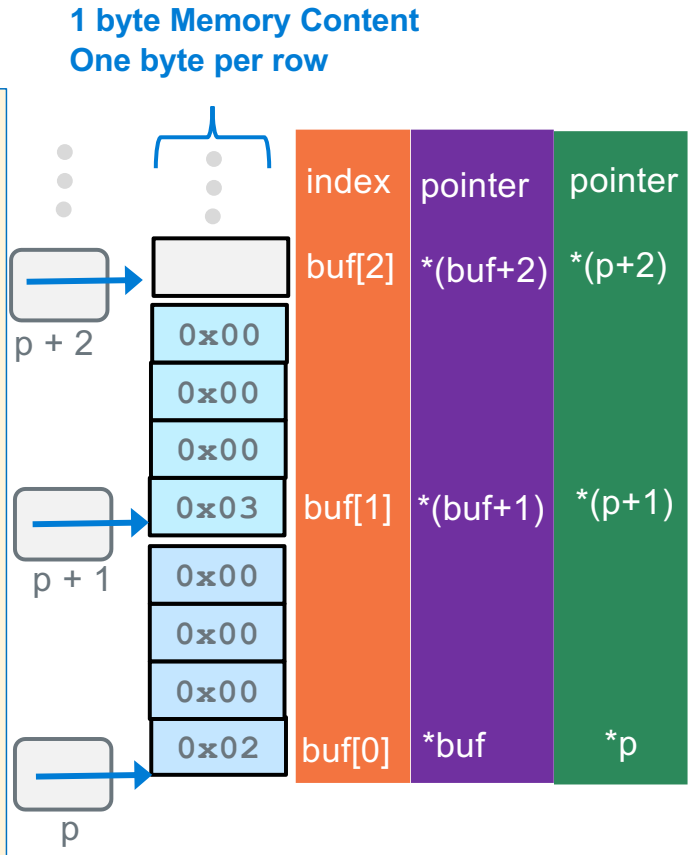
x

Pointer and Arrays - 2

When p is a pointer, the actual value of `(p+1)` **depends on the type** that pointer p points at

- `(p+1)` adds `1 x sizeof(what p points at)` bytes to p
  - Comment: `++p` is equivalent to `p = p + 1`
- Using pointer arithmetic to find array elements:
  - Address of the second element `&buf[1]` is `(buf + 1)`
  - It can be referenced as `*(buf + 1)` **or** `buf[1]`

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

*p = *p + 10;
*(p + 1) = *(p + 1) + 10; // {12, 13, 5, 6, 11}
```

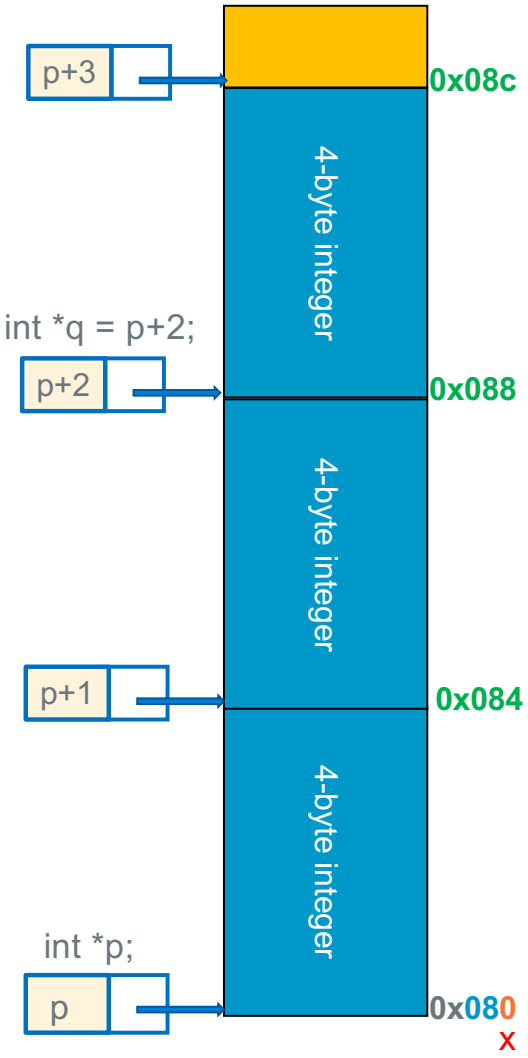


x

Pointer Arithmetic

- **You cannot add two pointers** *(what is the reason?)*
- A pointer q can be subtracted from another pointer p when the pointers are the same type – best done only within arrays!
- The value of `(p-q)` is the number of **elements between** the two pointers
  - Using memory address arithmetic (p and q Rside are both byte addresses):

```
distance in elements = (p - q)bytes/sizeof(*p)bytes

(p + 3) - p = 3 = (0x08c - 0x080)/4 = 3
```



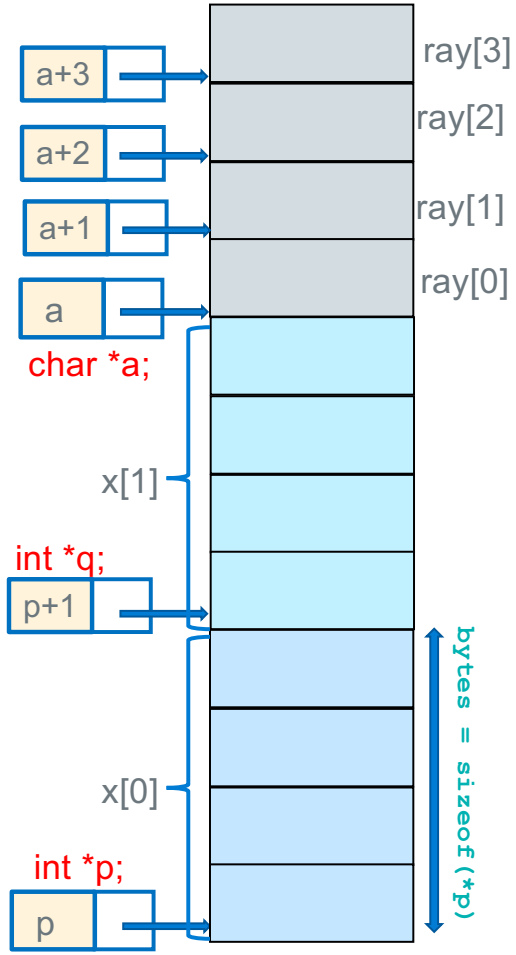
Pointer Arithmetic Use With Arrays

```
char ray[4];
char *a = ray;
int x[2];
int *p = x;
int *q = &x[1];
```

- Remember how ***`sizeof()`*** works:
  - `sizeof(p)` is the size of the **pointer**
  - `sizeof(*p)` evaluates to the size of **what p points at**
- Adding an integer i to a pointer p, the memory address computed by (p + i) in C is calculated with ***memory address arithmetic***

  `memory_address = p + (i × sizeof(*p))`
- Subtracting an integer i from a pointer `(p - i)`

  `memory_address = p - (i × sizeof(*p))`
- Number of element between two pointers p and q pointing at the same array
  - Caution: C only checks types, not if they are pointing at the same array

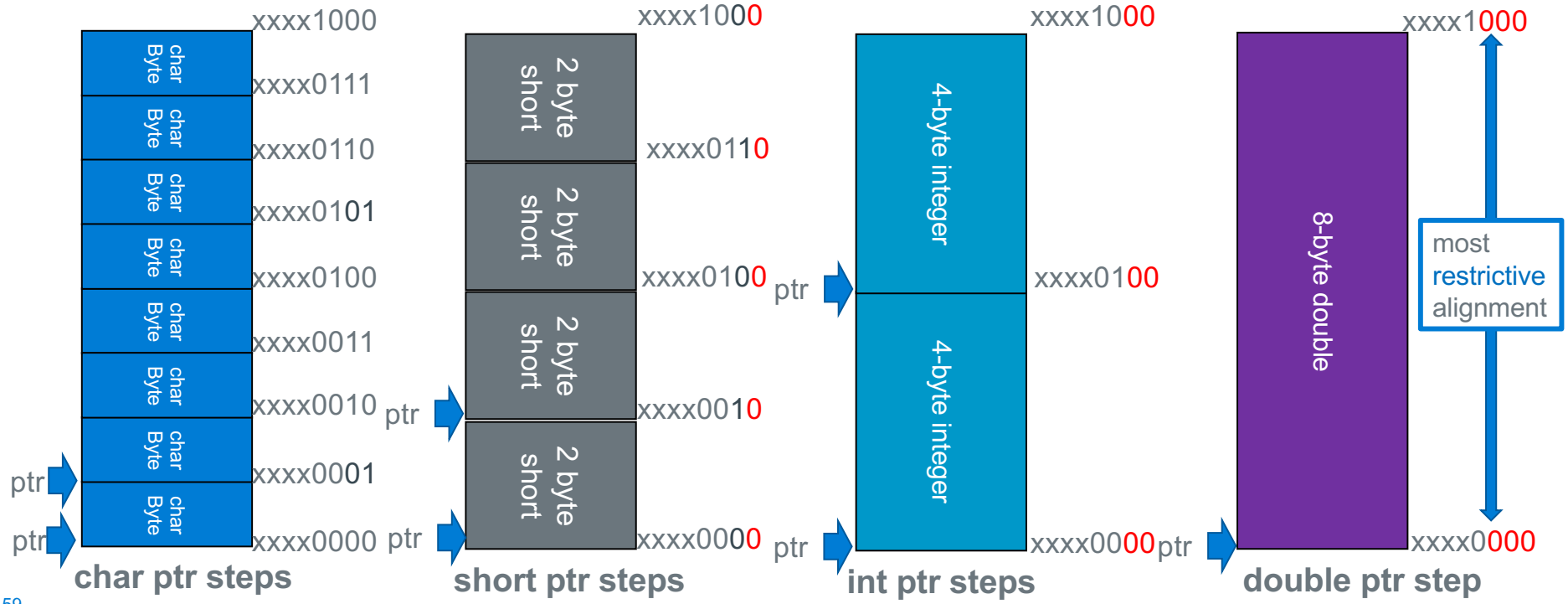


x

Starting Address Alignment Requirements and Pointer Math

For each type, a block represents the change in the actual memory (byte) address stored in the pointer when adding 1 in a C expression

variable address alignment : (**red** address digits) specificizes limitations on starting address by type



x

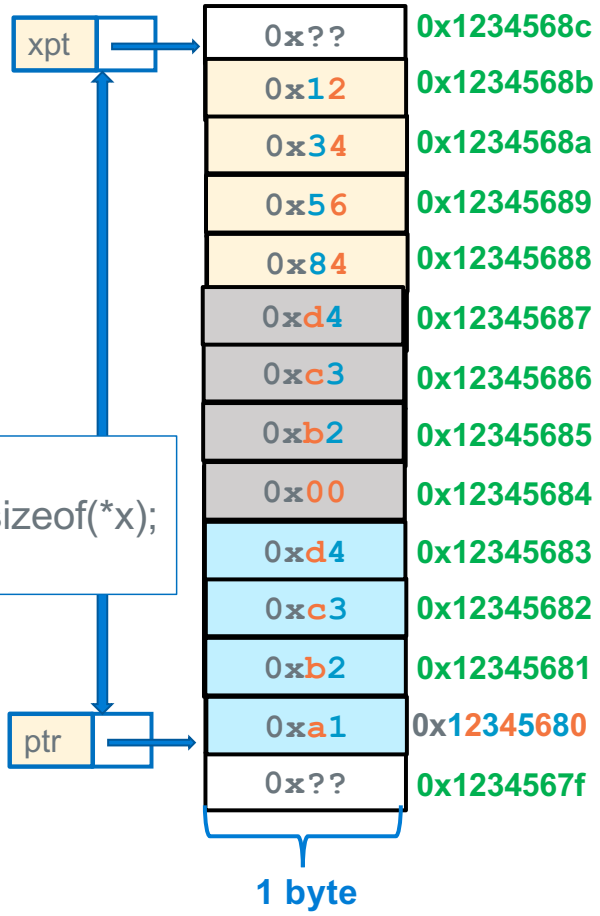
"Walking" an Array with Pointers

```
int x[] = {0xd4c3b2a1, 0xd4c3b200, 0x12345684};
int cnt = (int)(sizeof(x) / sizeof(*x));

int *ptr = x; //or &x[0]

int *xpt = ptr + cnt;

while (ptr < xpt) {
    printf("%#x\n", *ptr);
    ptr++;
}
```

xpt is a loop **limit pointer** points 1 element past the end of the array

```
cnt   = 3;
bytes = cnt * sizeof(*x);
      = 12
```

```
 % ./a.out
0xd4c3b2a1
0xd4c3b200
0x12345684
```



x

C Precedence and Pointers

- ++ -- pre and post increment combined with pointers will create code that is complex, hard to read and difficult to maintain, so be careful!
- My advice: Always Use () to improve readability

```
int array[] = {2, 5, 7, 9, 11, 13};
int *ptr = array;
int x;
```

```
x = 1 + (*ptr++)++; // yuck!!
```

2 1 3

```
/* Same as the one line above */
x = 1 + *ptr;     // x = 1 + *orig_ptr (2) = 3;

*ptr = *ptr + 1; //(*orig_ptr)++ is array[0]= 3;

ptr = 1 + ptr;    // ptr = &array[1] = points 5
```

Operator	Description	Precedence level	Associativity
( ) [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Dot operator (Member selection via object name) Arrow operator(Member selection via pointer) Postfix increment/decrement	1 highest	Left to Right
+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Prefix increment/decrement Logical NOT One's complement Indirection Address (of operand) Type cast Determine size in bytes on this implementation	2	Right to Left
* / %	Multiplication Division Modulus	3	Left to Right
+ -	Addition Subtraction	4	Left to Right
<< >>	Left shift Right shift	5	Left to Right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left to Right
== !=	Equal to Not equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
|	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
||	Logical OR	12	Left to Right
?:	Conditional operator	13	Right to Left
= *= /= %= += -= &= ^= != <<= >>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

x

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***

```c
int main(void)
{
  int numbers[] = {9, 8, 1, 9, 5};

  passa(numbers);
  printf("numbers size:%lu\n", sizeof(numbers)); // 20
  return EXIT_SUCCESS;
}
```

```c
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)

- 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)

x

Arrays As Parameters: What is the size of the array?

- It's tricky to use arrays as parameters, as **they are passed as pointers to the start of the array**
  - In C, **Arrays do not know their own size** and at runtime there is no "bounds" checking on indexes

```
int sumAll(int a[]);

int main(void)
{
  int numb[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numb);

  return EXIT_SUCCESS;
}

int sumAll(int a[])
{
  int i, sum = 0;
  int sz = (int) (sizeof(a)/sizeof(*a));
  for (i = 0; i < sz; i++) // this does not work
      sum += a[i];
  }
}
```

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

"inside" the body of sumAll(), the question is: how big is that array? all I have is a POINTER to the first element…..
sz is a 1 on 32 bit arm

x

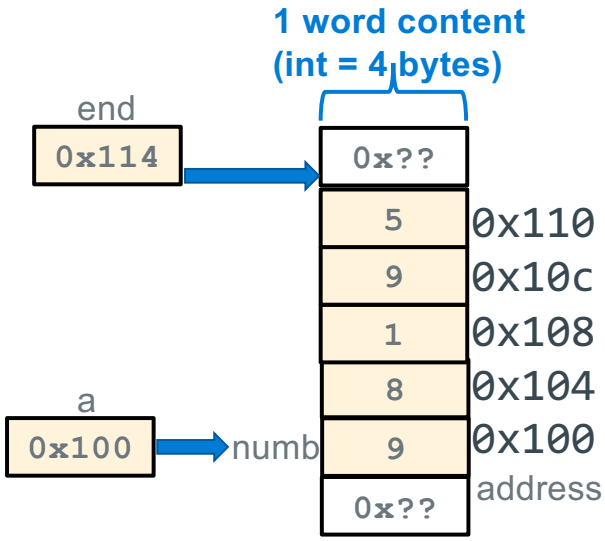
Arrays As Parameters, Approach 1: Pass the size

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 = sizeof(numb)/sizeof(numb[0]);

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




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

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

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

  for (int i= 0; i < size; i++)
      sum += a[i]; // *(a + i)
  return sum;
}
```

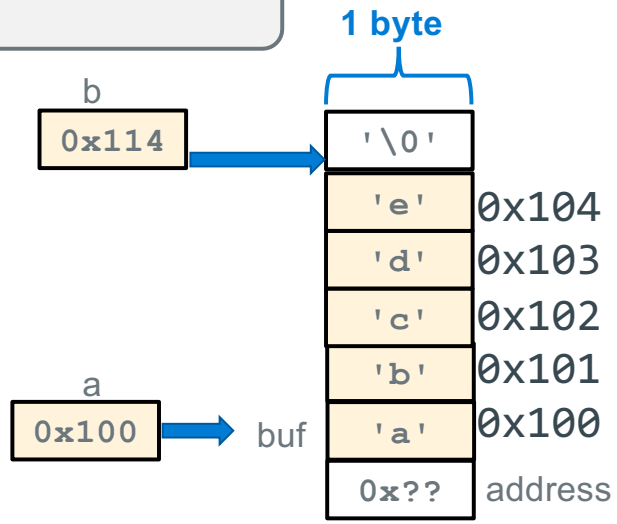

x

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). For example: '\0', NULL

```
int my_strlen(char *a);
int main(void)
{
  char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // string

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

```
int my_strlen(char *a)
{
  char *b = a;

  if (a == NULL)  // check for NULL pointer
    return 0;
  while (*b++ != '\0')
      ;
  return (b - a - 1);
}
```



x

Returning Arrays; Array as an Output Parameter

This is very bad
you return the address
of an automatic variable

```
int *copyArray(int src[], int size)
{
  int i, dst[size];   // dynamic array

  for (i = 0; i < size; i++)
    dst[i] = src[i];
  return dst;  // no compiler error, but wrong!
}
```

- Option 1: Use an array either defined in the caller or valid in the caller's scope
  - Then pass a pointer to the array as an output parameter
- Option 2: use allocated storage: malloc()

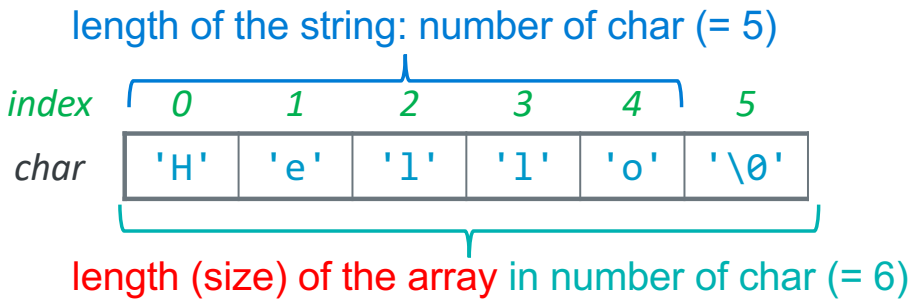
```
#define SZ 5
…
int orig[SZ] = {9, 8, 1, 9, 5};
int copy[SZ];

copyArray(orig, copy, SZ);
…
```

```
void copyArray(int *src, int *dst, int size)
/* assumes dst array is same or larger */
{
  int *end = src + size;
  while (src < end)
    *dst++ = *src++;
}
```


x

C Strings - 1

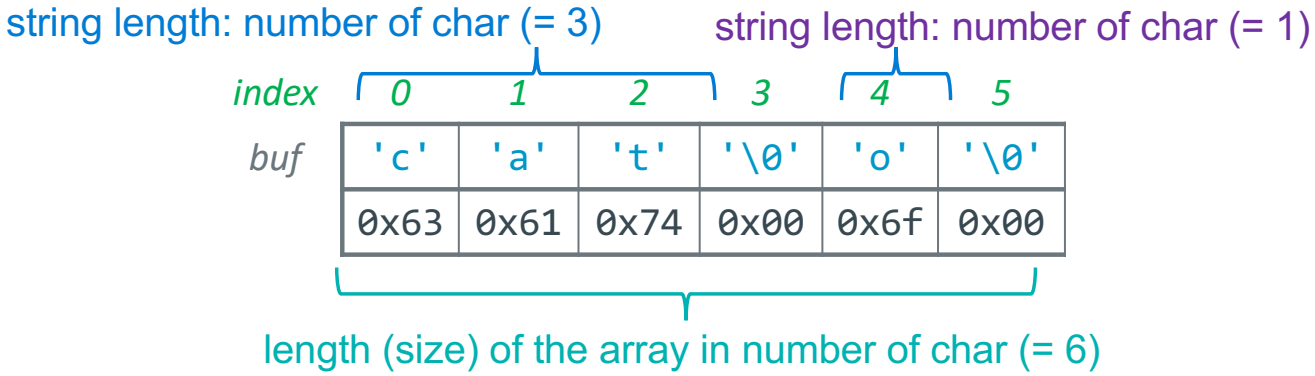
- **C does not** have a **dedicated type** for strings
- **Strings are** an **array of characters terminated by** a sentinel termination **character**
- `'\0'` is the **Null termination character;** has the **value of zero (do not confuse with '0')**
- An **array of chars** contains **a string only when** 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 end



x

C Strings - 2

- **First `'\0'` encountered from the start of the string** always indicates the end of a string
- The **'\0' does not have to be** in the **last element in the space allocated to the array**
  - String length is always less than the size of the array it is contained in
- In the example below, the array buf contains two strings
  - One string starts at &(buf[0]) is "cat" with a string length of 3
  - The other string starts at &(b[4]) is "o" with a string length of 1
  - "o" has two bytes: 'o' and '\0'



x

String Literals (Read-Only) in Expressions

- When strings in quotations (*e.g.,* "string") are **part of** an **expression** (*i.e., not* part of an *array initialization*) they are called ***string literals***

```
printf("literal\n");
printf("literal %s\n", "another literal");
```

- What is a ***string literal:***
  - Is a null-terminated string in a **`const char array`**
  - Located in the **read-only data** segment of memory
  - Is not assigned a variable name by the compiler, so it is only accessible by the location in memory where it is stored
- **String literals** are a type of ***anonymous variable***
  - Memory containing data without a name bound to them (only the address is known)
- Code above, the *string literal* in the printf()'s, are replaced with the starting address of the corresponding array (first or [0] element) when the code is compiled

x

Be Careful with C Strings and Arrays of Chars

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

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

```
char *mess2 = "Hello World";  // "Hello World" is a string literal
                              // mess2 is a pointer NOT an array!
*mess = 'h';                  // undefined in C, linux seg fault
mess2 = mess1;
```

- mess2 **pointer** to an **immutable** array with enough space to hold the string + '\0'
  - you **cannot change** array contents, but you can change what mess2 points at

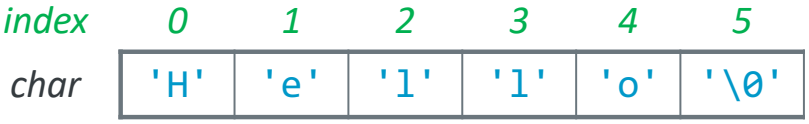
```
char mess3[] = {'H','e','l','l','o',' ','W','o','r','l','d'\'};
```

- mess3 is an array but does not contain a '\0' SO IT IS NOT A VALID STRING

x

Copying Strings: Use the Sentinel; libc: strcpy(), strncpy()

- To copy an array, you must copy each character from source to destination array
- Watch overwrites: strcpy assumes the target array size is equal or larger than source array




```
char str1[80];
strcpy(str1, "hello");
```

```
// strncpy adds a length limit on copy
char str1[6];
strncpy(str1, "hello", 5); // \0 not copied
str1[5] = '\0'; // make sure \0 terminated
```

```
char *strcpy(char *s0, char *s1)
{
    char *str = s0;

    if ((s0 == NULL) || (s1 == NULL))
       return NULL;
    while (*s0++ = *s1++)
       ;
    return str;
}
```

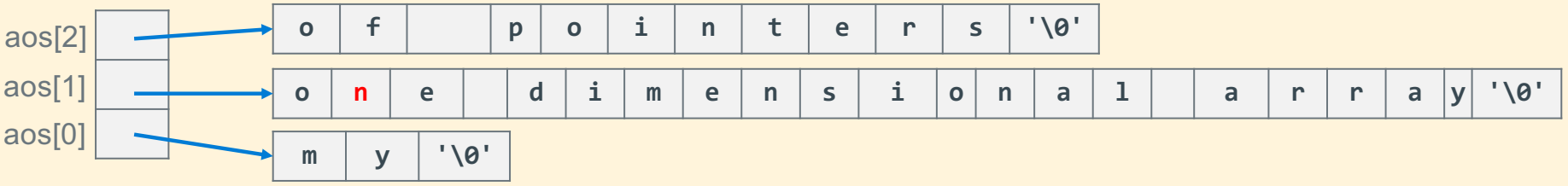
```
char *strncpy(char *s0, char *s1, int len)
{
    char *str = s0;
    if ((s0 == NULL) || (s1 == NULL))
       return NULL;

    while ((*s0++ = *s1++) && --len)
       ;
    return str;
}
```


x

2D Pointer Array to Strings

- 2D char arrays are an inefficient way to store strings (wastes memory) unless all the strings are similar lengths, so 2D char arrays *are rarely used* with string elements
- An 2D **array of pointers** is common for strings as *"rows"* can very in length

```
char *aos[] = {"my", "one dimensional array", "of pointers"};
```



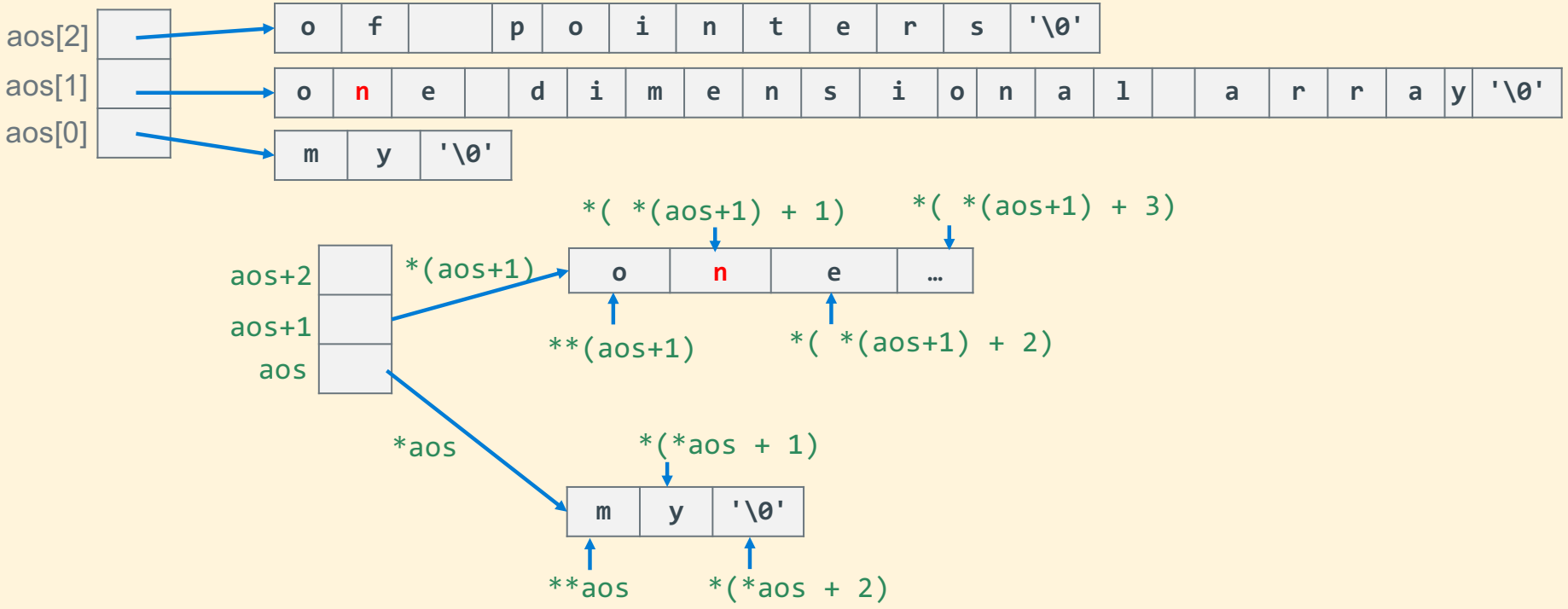
- `aos` is an array of pointers, with each pointer pointing to a read only string literal
  - Not a 2D array, but any char can be accessed as if it was in a 2D array of chars

x

2D Pointer Array to Strings

```
char *aos[] = {"my", "one dimensional array", "of pointers"};
```

For example. `aos[1][1]` is `*(*(aos + 1) + 1)` which contains '**n**'



x

main() Command line arguments: argc, argv

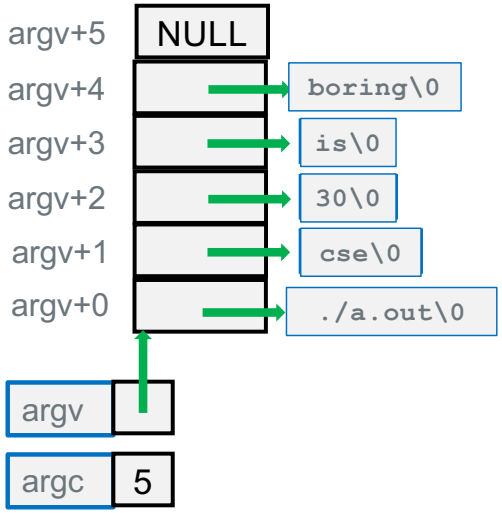
- Arguments are passed to main() as a pointer to an array of pointers `(**argv or *argv[])`

  `Conceptually: % *argv[0] *argv[1] *argv[2] ….`

  `% vim file.c`

- `argc` is the number of elements in the array or args
- `*argv (argv[0]) is` **usually** is the name of the executable file (% ./vim file.c)
- `*(argv + argc)` always contains a NULL (0) sentinel
- `*argv[]` elements point at **mutable strings**!

```
% ./a.out cse 30 is boring
./a.out
cse
30
is
boring
```



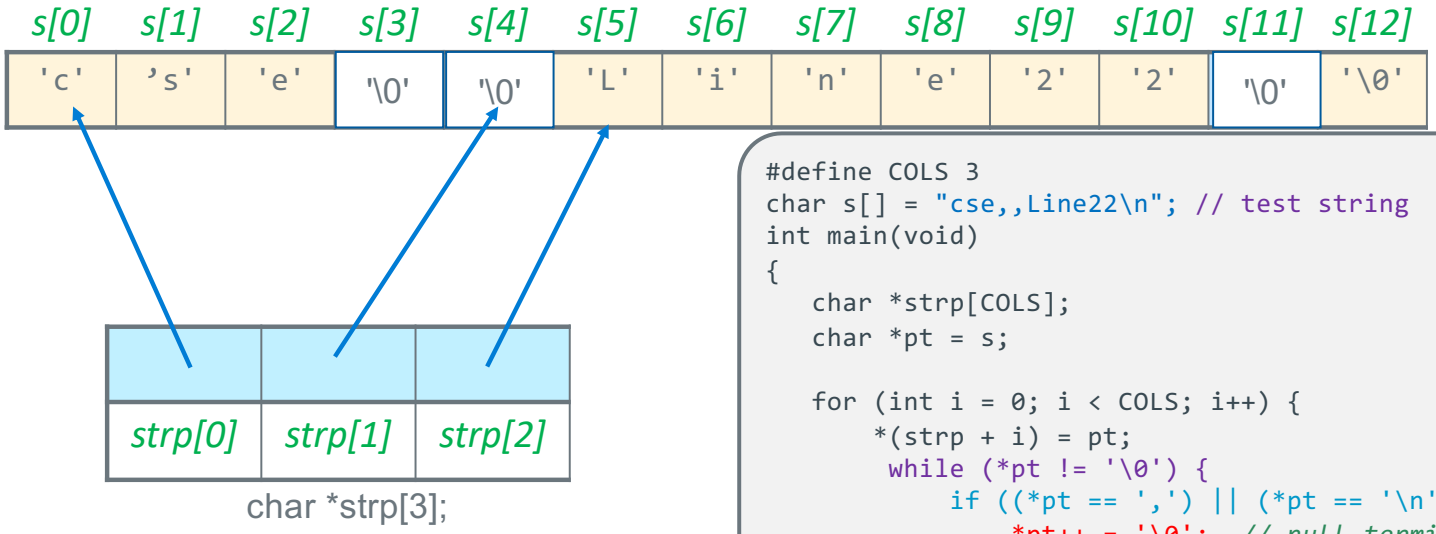

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    while (*argv != NULL)
        printf("%s\n", *argv++);
    }
    return EXIT_SUCCESS;
}
```


x

PA2: Creating an Array of Tokens in a String

Break up a string at each space into smaller strings and link each substring into an array of string pointers

Sample code assumes exactly 3 columns



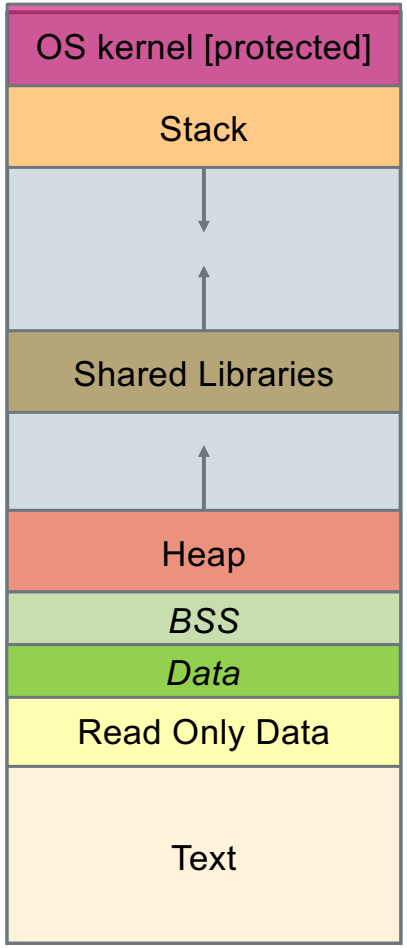

```
% ./a.out
[0]: cse
[1]:
[2]: Line22
```

```c
#define COLS 3
char s[] = "cse,,Line22\n"; // test string
int main(void)
{
   char *strp[COLS];
   char *pt = s;

   for (int i = 0; i < COLS; i++) {
       *(strp + i) = pt;
        while (*pt != '\0') {
            if ((*pt == ',') || (*pt == '\n')) {
                *pt++ = '\0';  // null terminate
                break;         // break out of while loop
            }
            pt++;
        }
    }
    for (int i = 0; i < COLS; i++)
        printf("[%d]: %s\n",i, *(strp + i));
    return EXIT_SUCCESS;
}
```


x

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



x

Heap Dynamic Memory Allocation Library Functions

#include <stdlib.h>	args	Clears memory
void *malloc(…)	size_t size	no
void *calloc(…)	size_t nmemb, size_t memsize	yes
void *realloc(…)	void *ptr, size_size	no
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 (not allowed in C, but allowed in gcc). The assignment to a typed pointer *"converts"* it from a void *
- **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**

x

Heap Allocation Routine Summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- `NULL` on failure, so check return value
- Memory is returned is contiguous
- it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location
- If you `realloc`/`free` non-heap address

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

- **Always use** `sizeof()` it makes your **code more portable**

```
int *ptr = malloc(n * sizeof(*ptr));
```

```
#include <stdlib.h>                    // need this for malloc() etc
    int col_cnt = 10;
    char *bufptr;
    /* ALWAYS CHECK THE RETURN VALUE FROM MALLOC!!!! */
    if ((bufptr = malloc(col_cnt * sizeof(*bufptr))) == NULL) {
        fprintf(stderr, "Unable to malloc memory");
        return NULL
    }
    return bufptr;
```


x

Calloc()

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

`calloc()` variant of **`malloc()`** but zeros out every byte of memory 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

x

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 allocation (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(col_cnt * sizeof(*bufptr))) == NULL) {
     fprintf(stderr, "Unable to malloc memory");
     return EXIT_FAILURE;
 }
 for (int j = 0; j < col_cnt; j++)
    *(bufptr + j) = 'a';                 // fill each array element with 'a'
 free(bufptr);                          // returns memory to the heap
 bufptr = NULL;                         // set bufptr to NULL
```


x

Mis-Use of Free()

- Call **`free()`** only with only the same memory returned from the heap
  - It is NOT an error to pass **`free()`** a pointer to NULL
- 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()`**, **`realloc()`**, **`calloc()`**) may fail or seg fault

```
char *bytes = malloc(1024 * sizeof(*bytes));
char *ptr = "cse30";
            …
    /* some code */
    free(bytes + 5);        // not ok
    free(ptr);              /* not memory on the heap */
```

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


x

Heap Memory "Leaks"

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

```
void
leaky_memory (void)
{
    char *bytes = malloc(BLKSZ * sizeof(*bytes));
…
    /* code that never passes the pointer in bytes to anything */
    return;
}
```

- Your program is responsible for cleaning up any memory it allocates but no longer needs
  - 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
  - Make sure you free memory when you no longer need it
- Valgrind is a tool for finding memory leaks (not pre-installed in all linux distributions though!)

x

Dangling Pointers

- When a 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

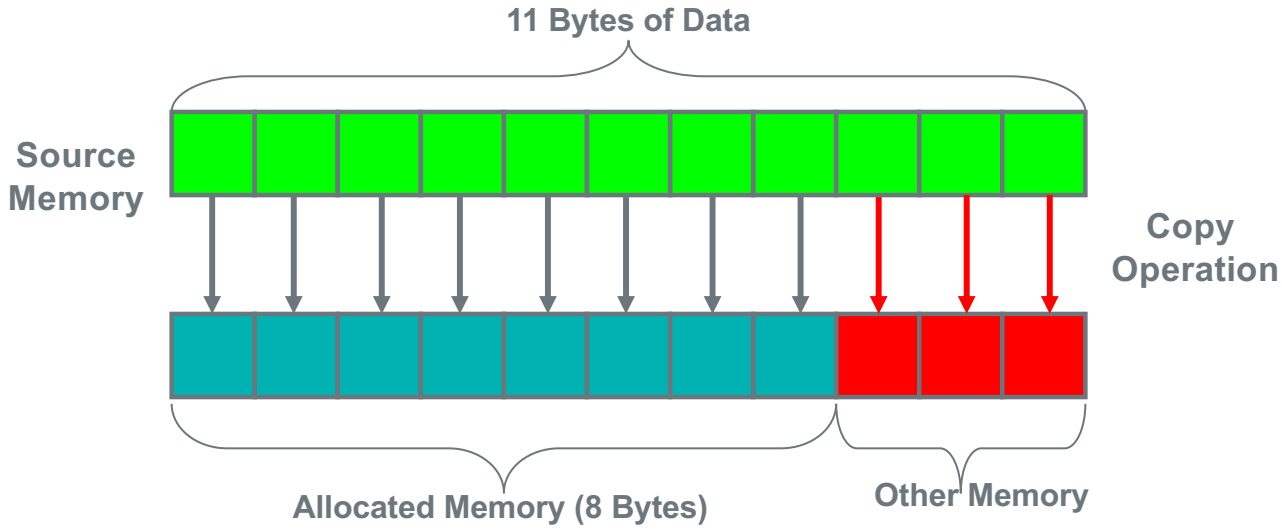
```
char *dangling_freed_heap(void)
{
    char *buff = malloc(BLKSZ * sizeof(*buff));
…
    free(buff);
    return buff;
}
```

- `dangling_freed_heap()` type code often causes the allocators (**`malloc()`**and friends) to **seg fault**
  - Because it corrupts data structures the heap code uses to manage the memory pool

x

string buffer overflow: common security flaw

- A buffer overflow occurs when data is written outside the boundaries of the memory allocated to target variable (or target buffer)
- `strcpy()` is a very *common source of buffer overrun security flaws*:
  - always ensure that the destination array is **large enough** (and don't forget the null terminator)
- `strcpy()` can cause problems when the ***destination*** and **source** regions ***overlap***



X

Struct Variable Definitions

- Variable definitions like any other data type:

```
struct vehicle name1, *pn, ar[3];
```

type: "struct vehicle"

single variable instance

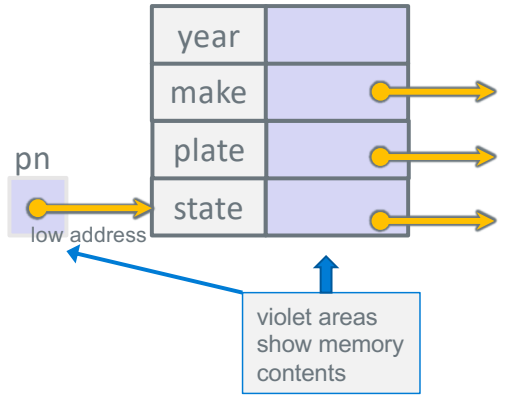
pointer

array

- Can combine struct and variable definition:
  - This syntax can be harder to read, though

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
} name1, *pn = &name1, ar[3];
```

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
};
struct vehicle name1;
struct vehicle *pn;
struct vehicle ar[3];
pn = &name1;
```



x

Accessing members of a struct

- Like arrays, struct variables are aggregated contiguous objects in memory
- the . structure operator which *"selects"* the requested field or member

```
struct date {// defining struct type
    int month;
    int day; // members date struct
};
```

- Now create a *pointer* to a struct

```
struct date *ptr = &bday;
```

```
struct date bday; // struct instance
bday.month = 1;
bday.day = 24;

// shorter initializer syntax
struct date new_years_eve = {12, 31};
struct date final = {.day= 24, .month= 1};
```

day	24
month	1

- Two options to reference a member via a struct pointer (. is higher precedence than *):
- Use * and . operators:

```
(*ptr).month = 11;
```

- Use -> operator for shorthand:

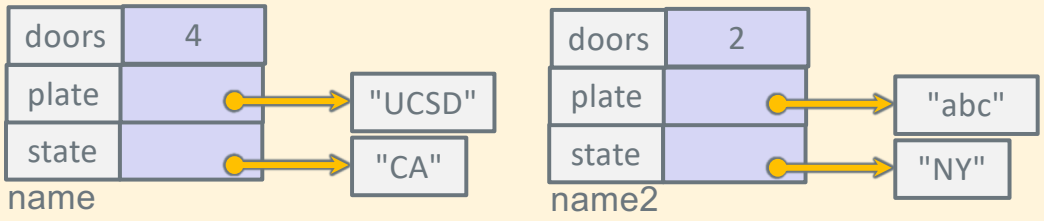
```
ptr->month = 11;
```


x

Comparing Two Structs

- You cannot compare entire structs, you must compare them one member at a time

```
struct vehicle {
  char *state;
  char *plate;
  int doors;
};
```




```
struct vehicle name  = {"CA", "UCSD", 4};
struct vehicle name2 = {(char []){"NY"}, (char []){"abc"}, 2};
```

```
if ((strcmp(name.state, name2.state) == 0) &&
    (strcmp(name.plate, name2.plate) == 0) &&
    (name.doors == name2.doors)) {
    printf("Same\n");
} else {
    printf("Different\n");
}
```


x

Struct: Arrays and Dynamic Allocation

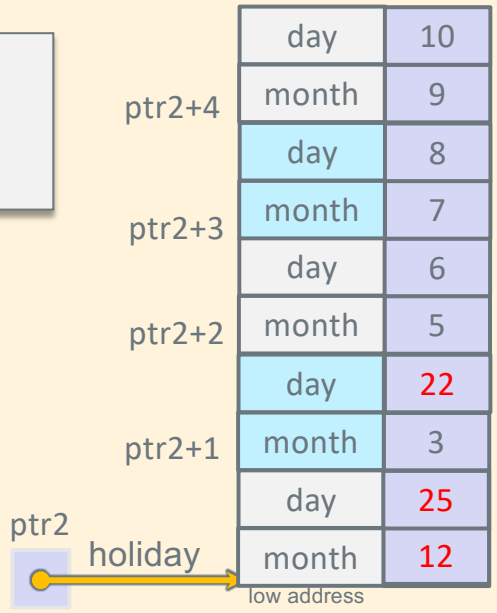
- Like any other type in C, you can create an array of structs

```
struct date holiday[] = {{1,2}, {3,4}, {5,6}, {7,8}, {9,10}};
int cnt = sizeof(holiday)/sizeof(*holiday);  // cnt = 5
```

- Allocate individual structs and arrays of structs using malloc()
  - Remember . is higher precedence than *:

```
#define HOLIDAY 5
struct date *pt1 = malloc(sizeof(*pt1));
struct date *pt2 = malloc(sizeof(*pt2) * HOLIDAY);
```

```
pt2->month = 12;
pt2->day = 25;
(pt2+1)->day = 22;   //or (*(pt2+1)).month
free(pt1);
pt1 = NULL;
free(pt2);
pt2 = NULL;
```

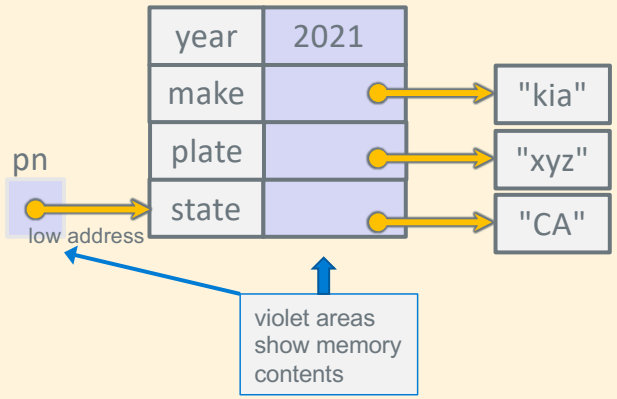


x

Struct Definition with Pointer Members

- You must allocate anything that is pointed at by a struct member independently (they are not part of the struct, only the pointers are)

```
struct vehicle {
  char *state;
  char *plate;
  char *make;
  int year;
};
struct vehicle name1;
pn = &name1;
```

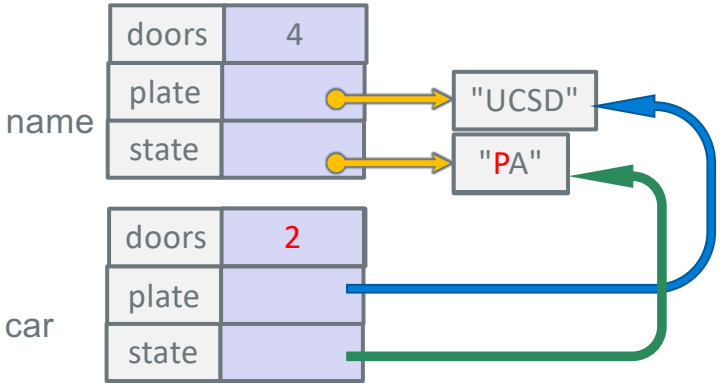



```
name1.state = strdup("CA");
pn->plate = strdup("xyz");
pn->make = strdup("kia");
```

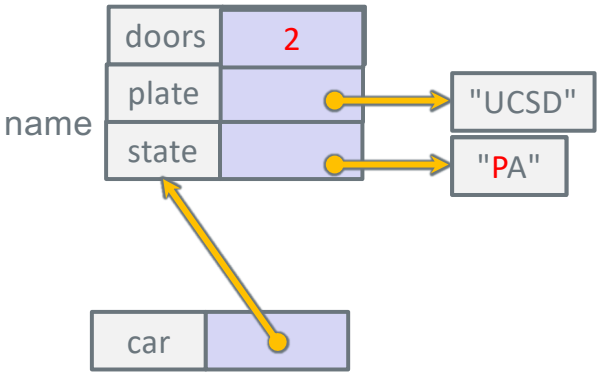

x

Struct as a Parameter to Functions

```
void change1(struct vehicle car)
{
    car.door = 2;
   *(car.state) = "P";
}
…
change1(name);
```




```
void change2(struct vehicle *car)
{
    car->door = 2;
   *(car->state) = "P";
}
…
change2(name);
```

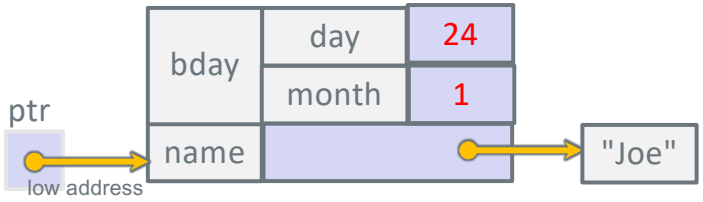


x

Struct as an Output Parameter

```
struct date {
    int month;
    int day;
};
```

```
struct person {
    char *name;
    struct date bday;
};
```




```
int fill(struct person *ptr, char *name, int month, int day)
{
    if ((ptr->name = strdup(name)) == NULL)
        return -1;
    ptr->bday.month = month;
    ptr->bday.day = day;
    return 0;
}
…
    struct person first;
    if (fill(&first, "joe", 1, 24) == 0)
        printf("%s %d %d\n", first.name,
         first.bday.month, first.bday.day);
…
```


x

Review: Singly Linked Linked List - 1

- Is a linear collection of nodes whose order is not specified by their relative location in memory, like an array
- Each node consists of a payload and a pointer to the next node in the list
  - The pointer in the last node in the list is NULL (or 0)
  - The head pointer points at the first node in the list (the head is not part of the list)
- Nodes are easy to insert and delete from any position without having to re-organize the entire data structure
- Advantages of a linked list:
  - Length can easily be changed (expand and contract) at execution time
  - Length does not need to be known in advance (like at compile time)
  - List can continue to expand while there is memory available

x

Linked List Using Self-Referential Structs

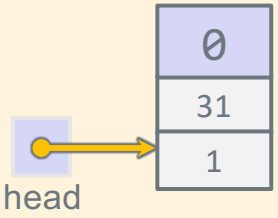
- A **self-referential struct** is a struct that has one or more members that are **pointers** to a **struct variable of the same type**
- Self-referential member
  - points to same type – itself

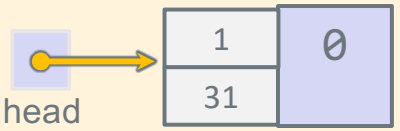
```
struct node {
  int data;
  struct node *next;
};
```

- There can be multiple struct members that make up the payload

```
struct node {
  int month;
  int day;
  struct node *next;
} x;
x.month = 1;
x.day = 31;
x.next = NULL;
```

```
struct node *head;  // head pointer
head = &x;
```





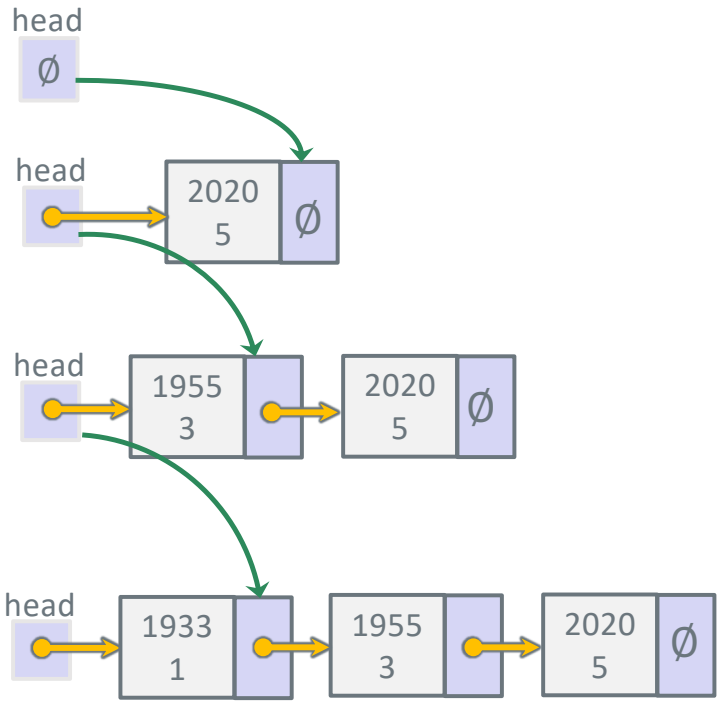
x

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

```
// create node; insert at front when passed head
struct node *creatNode(int data1, int data2,
       struct node *link)
{
    struct node *ptr = malloc(sizeof(*ptr));
    if (ptr != NULL) {
        ptr->data1 = data1;
        ptr->data2 = data2;
        ptr->next = link;
    }
    return ptr;
```

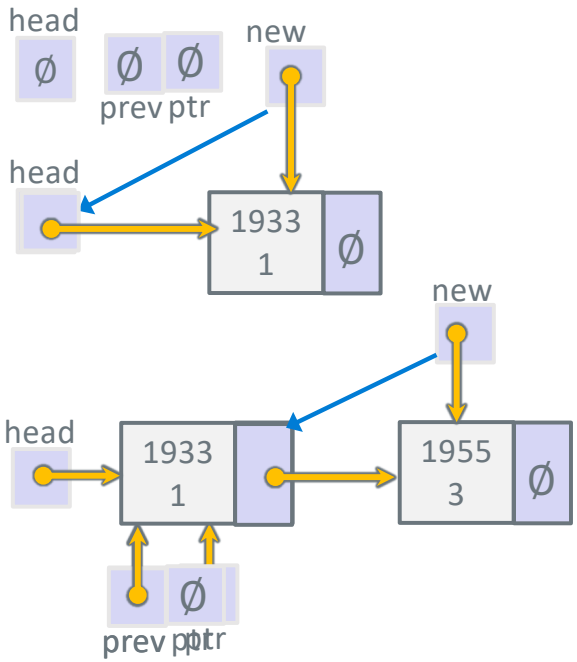
```
struct node *head = NULL; // insert at front
struct node *ptr;

if ((ptr = creatNode(2020, 5, head)) != NULL)
    head = ptr; // error handling not shown
if ((ptr = creatNode(1955, 3, head)) != NULL)
    head = ptr;
if ((ptr = creatNode(1933, 1, head)) != NULL)
    head = ptr;
```

```
struct node {
  int data1;
  int data2;
  struct node *next;
};
```



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

```
struct node *
insertEnd(int data1, int data2,struct node *head)
{
    struct node *ptr = head;
    struct node *prev = head;
    struct node *new;

    if ((new = creatNode(data1, data2, NULL)) == NULL)
        return NULL;

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

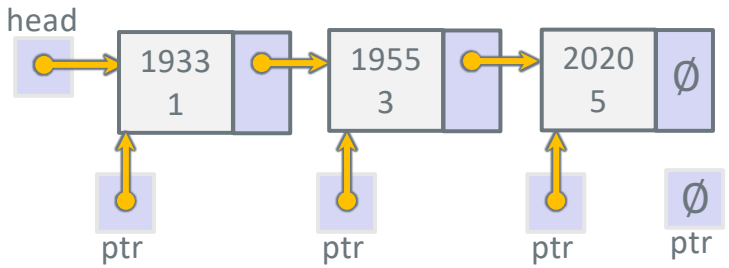



```
struct node *head = NULL;  // insert at end
struct node *ptr;
if ((ptr = insertEnd(1933, 1, head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, 3, head)) != NULL)
    head = ptr;
```


x

"Dumping" the Linked List

"walk the list from head to tail"



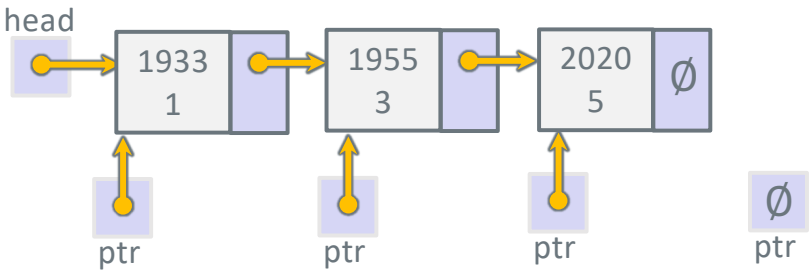

```
struct node *head;
struct node *ptr;
…
printf("\nDumping All Data\n");
ptr = head;
while (ptr != NULL) {
    printf("data1: %d data2: %d\n", ptr->data1, ptr->data2);
    ptr = ptr->next;
}
```

```
Dumping All Data
data1: 1933 data2: 1
data1: 1955 data2: 3
data1: 2020 data2: 5
```


x

Finding A Node Containing a Specific Payload Value

```
struct node {
  int data1;
  int data2;
  struct node *next;
};
```



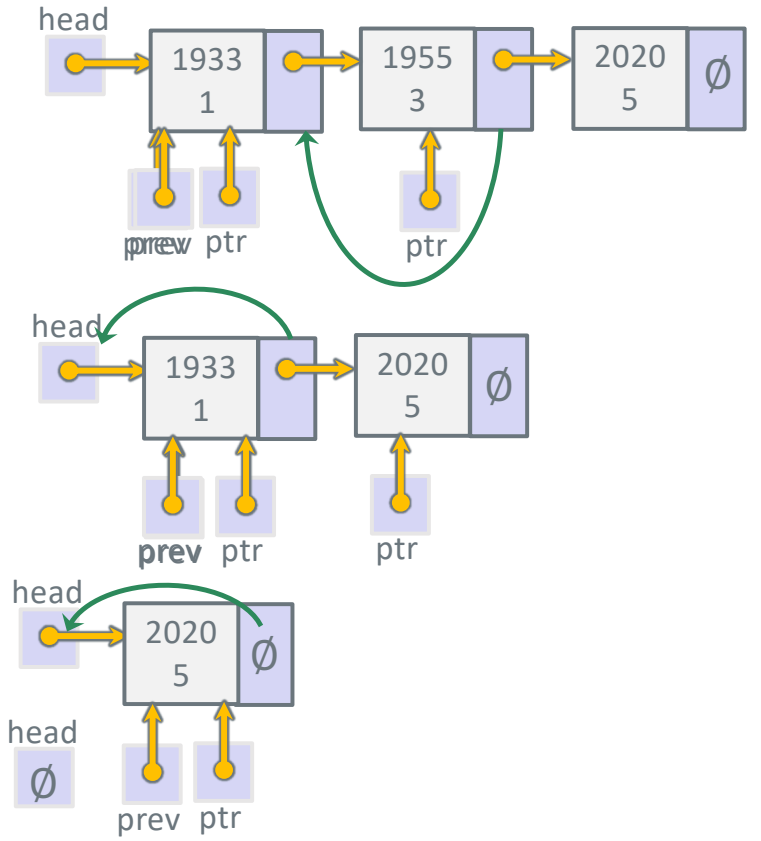

```
struct node *findNode(int data, struct node *ptr)
{
    while (ptr != NULL) {
        if (ptr->data1 == data)
            break;
        ptr = ptr->next;
    }
    return ptr;
}
```

```
struct node *found;

if ((found = findNode(2005, head)) != NULL)
    printf("data1: %d data2: %d\n", found->data1, found->data2);

if ((found = findNode(2020, head)) != NULL)
    printf("data1: %d data2: %d\n", found->data1, found->data2);
```


x

Deleting a Node in a Linked List

```
struct node *deleteNode(int data1, struct node *head)
{
    struct node *ptr = head;
    struct node *prev = head;

    while (ptr != NULL) {
        if (ptr->data1 == data1)
            break;
        prev = ptr;
        ptr = ptr->next;
    }
    if (ptr == NULL) // not found return head
        return head;
    if (ptr == head)
        head = ptr->next;
    else
        prev->next = ptr->next;
    free(ptr);
    return head;
}
```

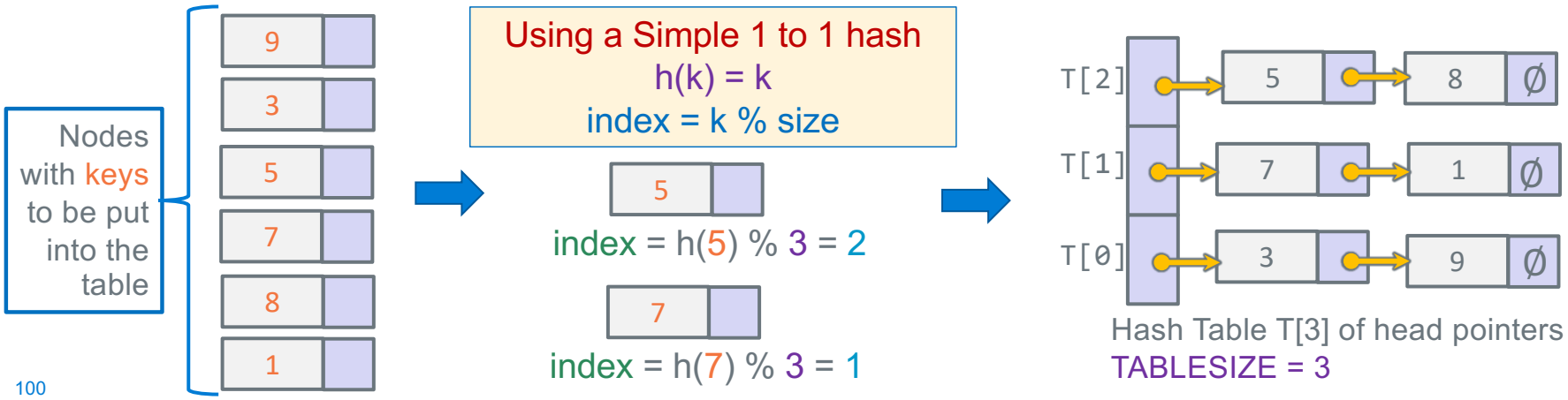



```
struct node *head = NULL;
head = deleteNode(1955, head);
head = deleteNode(1933, head);
head = deleteNode(2020, head);
```

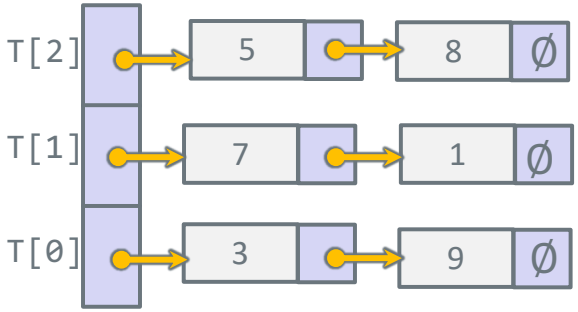
x

Hashing

- Hash table is an array of head pointers to different linked lists (called hash chains)
- **Each data item** must have a **unique key** that **identifies it (e.g., auto license plate)**
  - *h*(*k*) is the ***hash value*** of key *k* *to encode the key k into* ***an integer***
- Hash value is used as an index into the hash table array T[index] of size TABLESIZE
  - Index = *h*(*k*) % TABLESIZE (mod operator % maps a keys hash value to table index)
- Keys that hash to the same array index (*collide*) are then put on a linked list to resolve
  - After hashing a key, you then traverse the selected linked list to find the entry



Hash Table With Collision Chaining (multiple linked lists)



Hash Table T[3]
TABLESIZE = 3

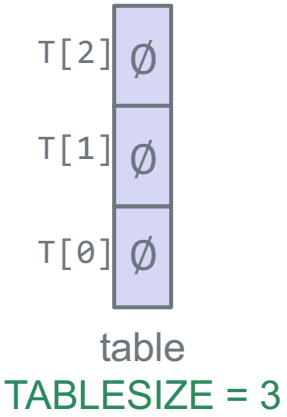
- Make TABLESIZE ***prime*** *as* keys are typically not randomly distributed, and have a *pattern*
  - Mostly even, mostly multiples of 10, etc.
  - In general: mostly multiples of some k
- If k is a factor of TABLESIZE, then only (TABLESIZE/k) slots will ever be used!

- Find the correct chain. Calculate index `i = hash(key) % TABLESIZE`
  - Go to array element `i`, i.e., `T[i]` that contains the head pointer for collision chain
  - Walk the linked list for element, add element, remove element, etc. from the linked list
  - New items added to the hash table are usually added at the front or at the end of the *collision chain* linked list *(when multiple keys hash to same index .. they **collide**)*
- **What about keys that are text "strings" ?**
  - **Hash arrays** need an **index number to select a chain**, so if we have a string, we must first convert to a number

x

Allocating the Hash Table (collision chain head pointers)

```
#define TBSZ 3
int main(void)
{
    struct node *ptr;
    struct node **table;
    uint32_t index;

    if ((table = calloc(TBSZ, sizeof(*table))) == NULL) {
        fprintf(stderr,"Cannot allocate hash table\n");
        return EXIT_FAILURE;
    }
…
```

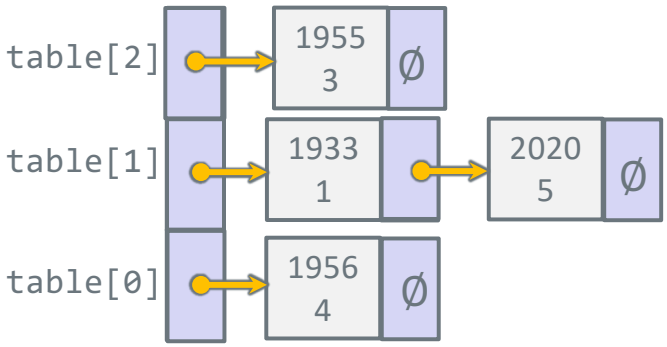


x

Inserting Nodes into the Hash Table (at the end)

```
index = 1933 % TBSZ; // really is hash(1933) % TBSZ
if ((ptr = insertEnd(1933, 1, table[index])) != NULL)
    table[index] = ptr;

index = 1955 % TBSZ;
if ((ptr = insertEnd(1955, 3, table[index])) != NULL)
    table[index] = ptr;

index = 2020 % TBSZ;
if ((ptr = insertEnd(2020, 5, table[index])) != NULL)
    table[index] = ptr;

index = 1956 % TBSZ;
if ((ptr = insertEnd(1956, 4, table[index])) != NULL)
    table[index] = ptr;
```

In this example: hash function is
h(k) = k
so, the hash function is not shown

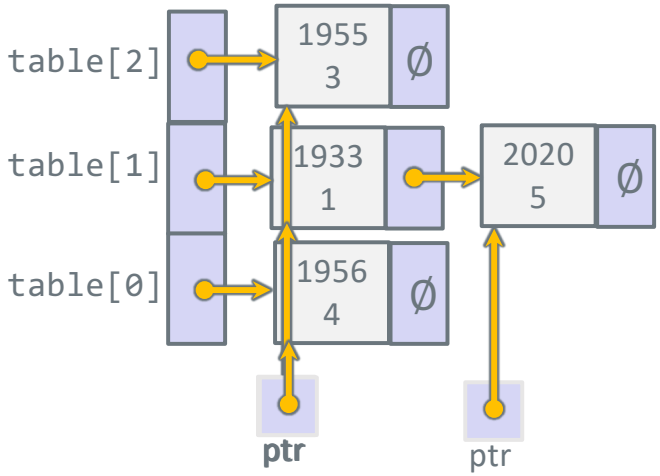


Notice

Substitute **createNode()** for **insertEnd()** to insert nodes at the **front** of the collision chain **instead** of at the **end** of the collision chain

x

"Dumping" the Hash Table (traversing all Nodes)

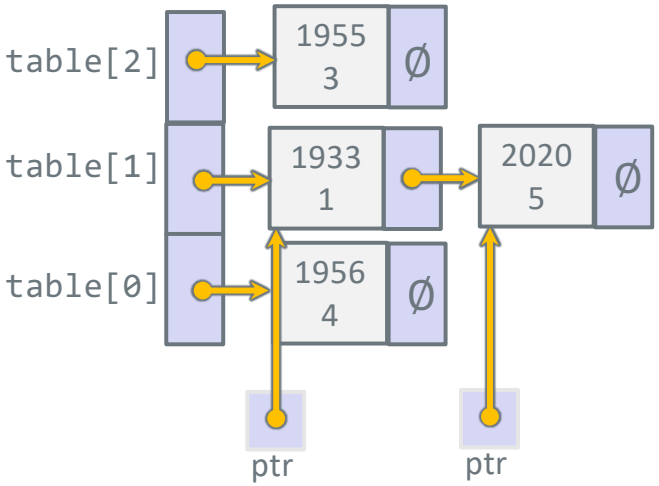



```
Dumping All Data
chain: 0
data1: 1956 data2: 4
chain: 1
data1: 1933 data2: 1
data1: 2020 data2: 5
chain: 2
data1: 1955 data2: 3
```

```
printf("\nDumping All Data\n");
for (index = 0U; index < TBSZ; index++) {
    ptr = table[index];
    printf("chain: %d\n", index);

    while (ptr != NULL) {
        printf("data1: %d data2: %d\n", ptr->data1, ptr->data2);
        ptr = ptr->next;
    }
}
```


x

Finding a Node with a Specific Payload Value



In this example the hash function is
h(k) = k
so, the hash function is not shown

```
// same routine as shown in a previous slide
struct node *findNode(int data, struct node *ptr)
{
    while (ptr != NULL) {
        if (ptr->data1 == data)
            break;
        ptr = ptr->next;
    }
    return ptr;
}
```

```
index = 2020 % TBSZ;
if ((ptr = findNode(2020, table[index])) != NULL)
    printf("Found data1: %d data2: %d\n", ptr->data1, ptr->data2);
else
    printf("Not Found 2020\n");
```


x

Number Base Overview (as written in C)

- Decimal is base 10, Hexadecimal is base 16, and octal is base 8
- **Octal digits** have 8 values 0 – 7 (written in C as **0**0 – **0**7, careful **0**73 is octal = 59 in decimal)
- **Hex digits** have 16 values 0 - 9 a - f (written in C as 0x0 – 0xf)
- No standard prefix in C for binary (most use hex) – gcc (compiler) allows 0b prefix others might not

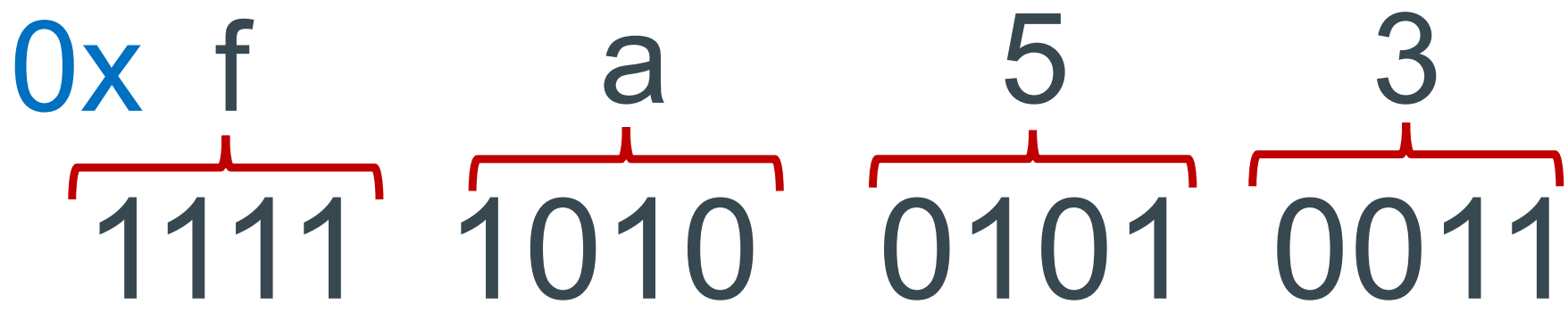
Hex digit Octal digit	0x0 00	0x1 01	0x2 02	0x3 03	0x4 04	0x5 05	0x6 06	0x7 07
Decimal value	0	1	2	3	4	5	6	7
Binary value	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111

Hex digit Octal digit	0x8 010	0x9 011	0xa 012	0xb 013	0xc 014	0xd 015	0xe 016	0xf 017
Decimal value	8	9	10	11	12	13	14	15
Binary value	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111

x

Hex to Binary (group 4 bits per digit from the right)

- Each Hex digit is 4 bits in base 2 $16^1 = 2^4$



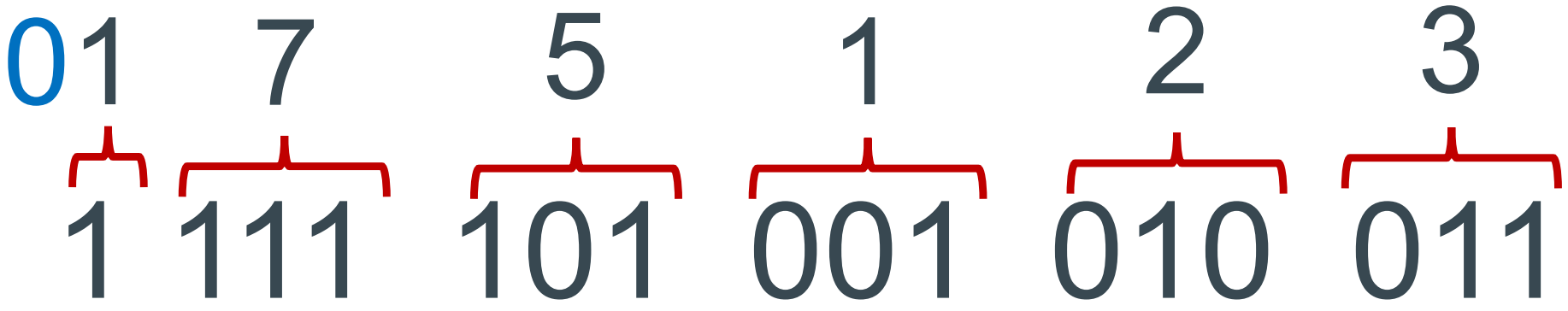
0b1111101001010011

binary start with a 0b in C

x

Octal to Binary (group 3 bits per digit from the right)

- One Octal digit is three binary digits $2^3 = 8^1$



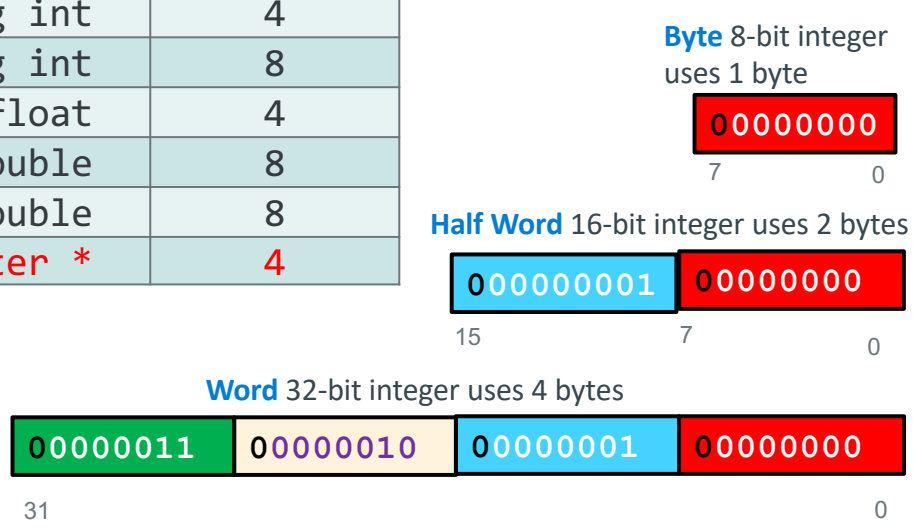
0b1111101001010011

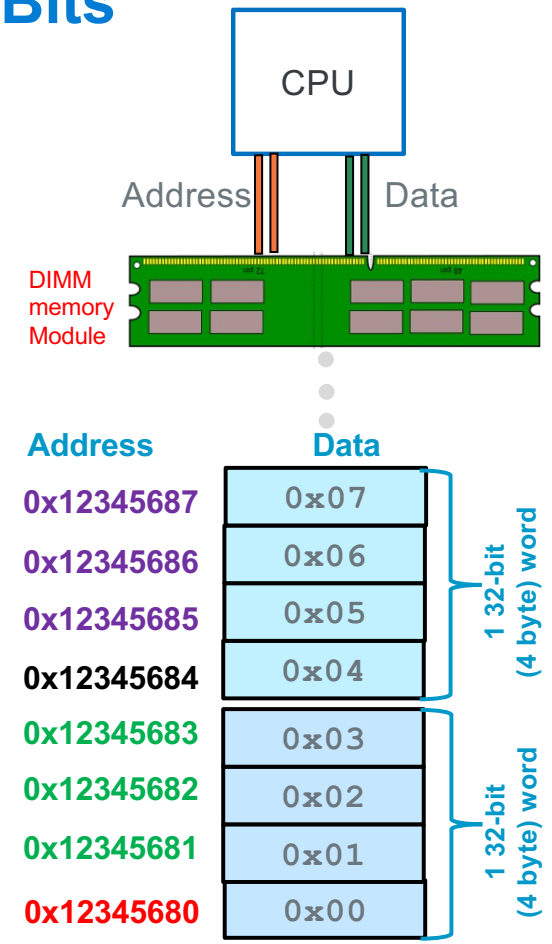
binary start with a 0b in C

x

Numbers Are Implemented with a Fixed # of Bits

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





x

Unsigned Decimal to Unsigned Binary Conversion

dividend 249	Quotient	Remainder	Bit Position
249/2	124	1	b0
124/2	62	0	b1
62/2	31	0	b2
31/2	15	1	b3
15/2	7	1	b4
7/2	3	1	b5
3/2	1	1	b6
1/2	0	1	b7

249(base 10) = $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1 b_0$ = $0b11111001$

11111001 = (1x128) + (1x64) + (1x32) + (1x16) + (1x8) + 1 = 249

x

Unsigned Binary to Unsigned Decimal Conversion

What is $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$
0 1 1 0 0 1 0 1$_{(base\ 2)}$ in decimal (N)?

Product Shift Left	Addend	Bit Position	Product
0	+ 0	b7	0
2 x 0 = 0 (shift left)	+ 1	b6	1
2 x 1 = 2	+ 1	b5	3
2 x 3 = 6	+ 0	b4	6
2 x 6 = 12	+ 0	b3	12
2 x 12 = 24	+ 1	b2	25
2 x 25 = 50	+ 0	b1	50
2 x 50 = 100	+ 1	b0	101

$101_{(base\ 10)}$ = (1x64) + (1x32) + (1x4) + 1 (checking the conversion)

x

Unsigned Integers (positive numbers) with a Fixed # of Bits

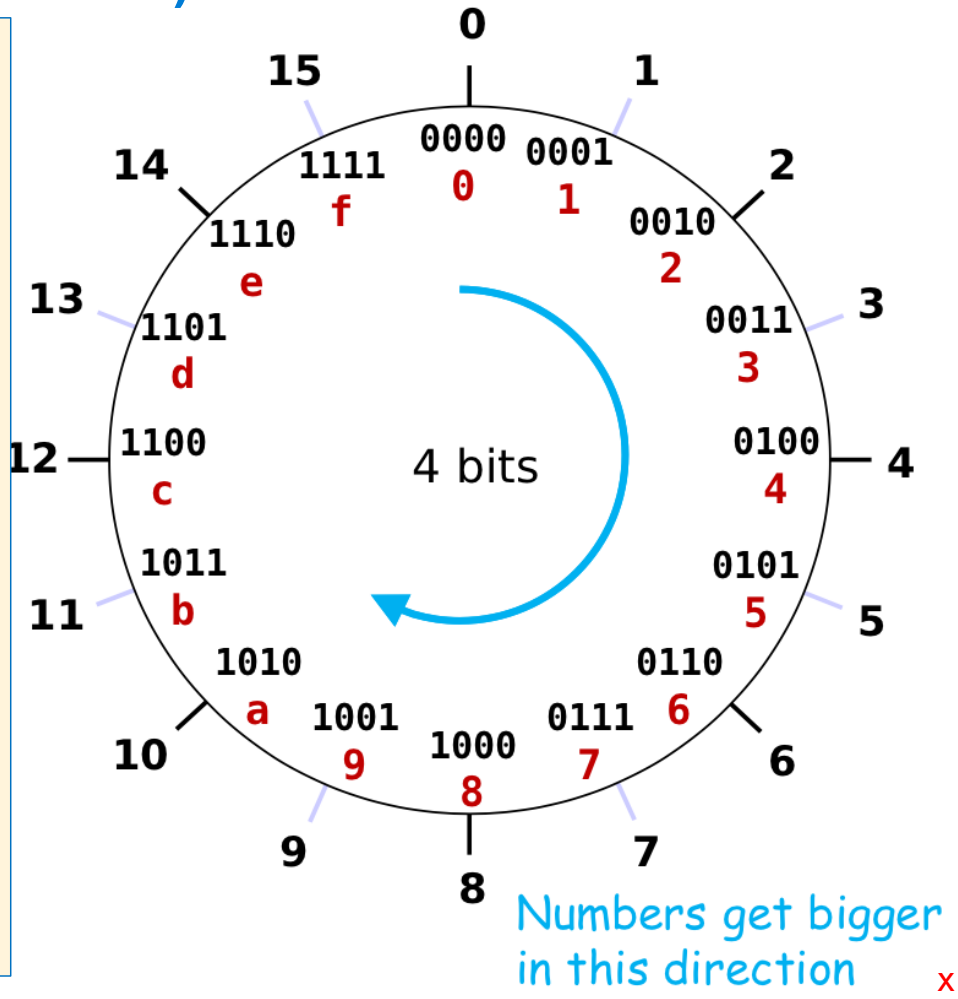
- Example 4 bits is 2^4 = 16 distinct values
- **Mod**ular (C operator: %) or clock math
  - Numbers start at 0 and "wrap around" after 15 and go back to 0
- Keep adding 1

  wraps (clockwise)

  0000 -> 0001 … -> 1111 -> 0000
- Keep subtracting 1

  wraps (counter-clockwise)

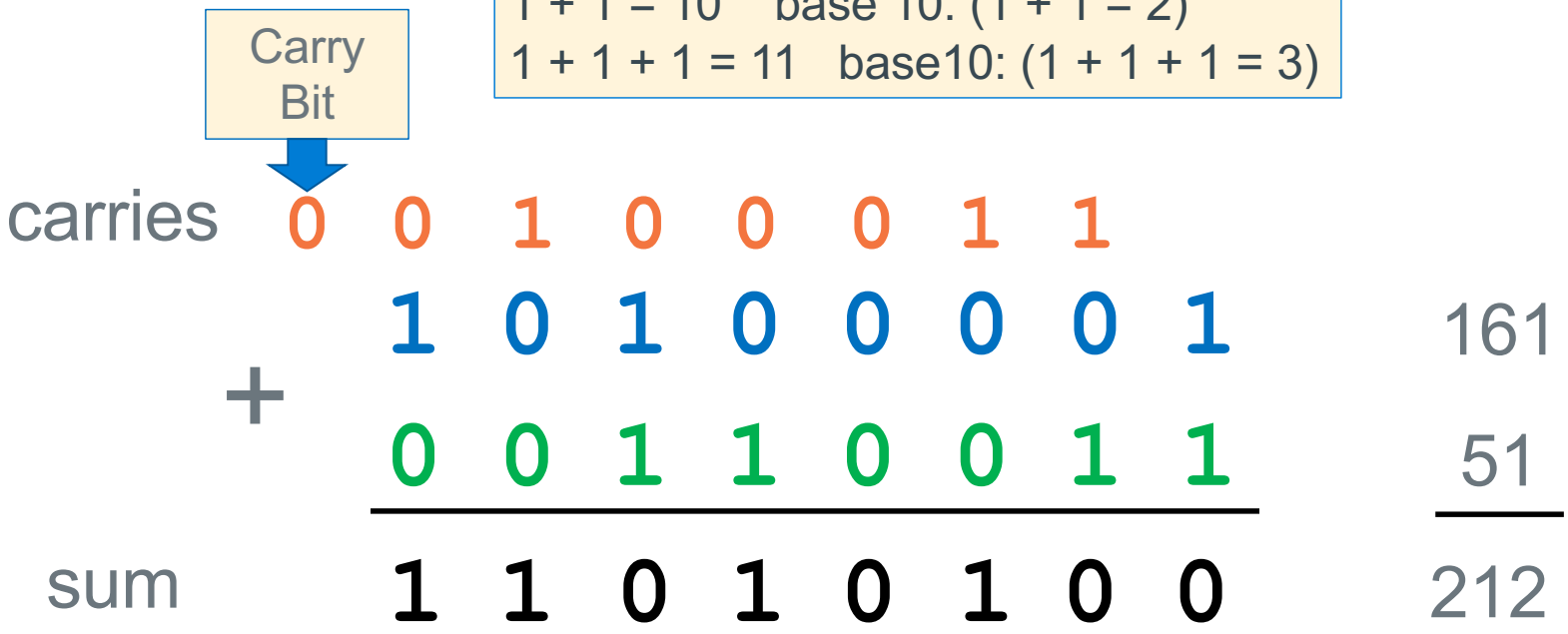
  1111 -> 1110 … -> 0000 -> 1111
- Addition and subtraction use normal "carry" and "borrow" rules, just operate in binary



x

Unsigned Binary Number: Addition in FIXED 8 bits

Be Aware in Binary
1 + 1 = 10 base 10: (1 + 1 = 2)
1 + 1 + 1 = 11 base10: (1 + 1 + 1 = 3)



x

Unsigned Binary Number: Subtraction in FIXED 8 bits

borrows		
	1 0 1 0 0 0 0 1	161
-	0 0 1 1 0 0 1 1	- 51
difference		

Be Aware in Binary

1 - 1 = 0

10 - 1 = 1 base 10: (2 – 1 = 1)

x

Unsigned Binary Number: Subtraction in FIXED 8 bits

borrows	10 10 10 10 10 10	
	0 1 0 1 1 1 0 1	161
-	0 0 1 1 0 0 1 1	- 51
difference	0 1 1 0 1 1 1 0	110

Be Aware in Binary

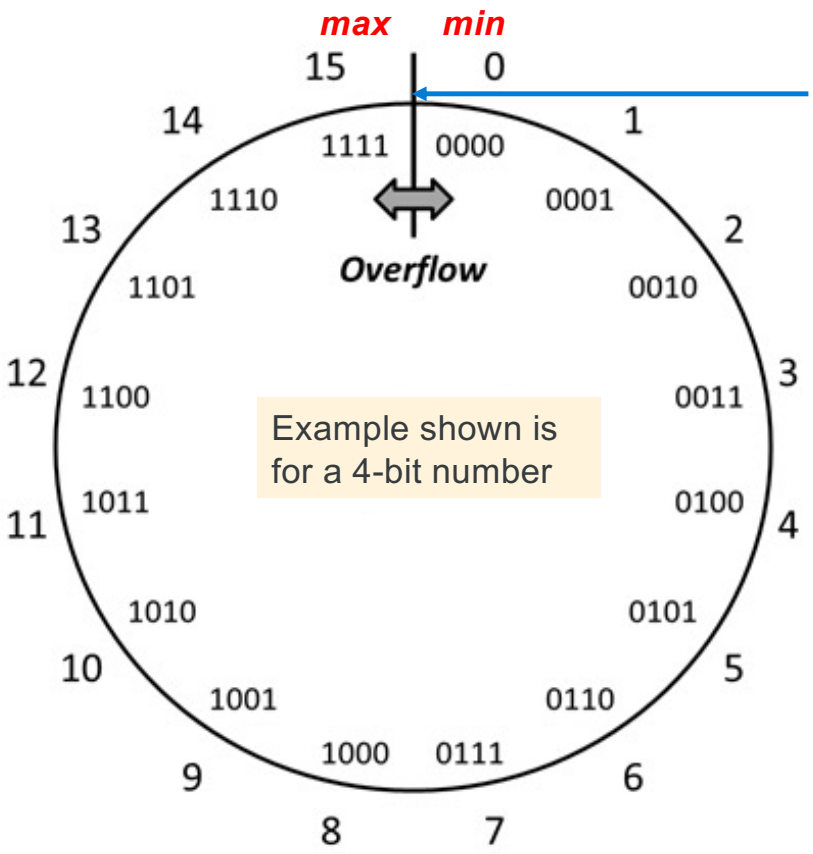
1 - 1 = 0

10 - 1 = 1 base 10: (2 – 1 = 1)

slide has powerpoint builds

x

Overflow: Going Past the Boundary Between max and min



Overflow: Occurs when an arithmetic result (from addition or subtraction for example) is is more than **min** or **max** limits

C (and Java) ignore overflow exceptions

- You end up with a bad value in your program and absolutely no warning or indication… happy debugging!….

x

Overflow: Unsigned Values 4-bit limit

Overflow: Occurs when an arithmetic result is **exceeds** the min or max limits

Addition Overflow: hardware drops carry

```
  15
+  2
  17
```

only 4 bits for numbers in this example

carry bit is always dropped from result

```
  1111
+ 0010
 10001
oops 1
```

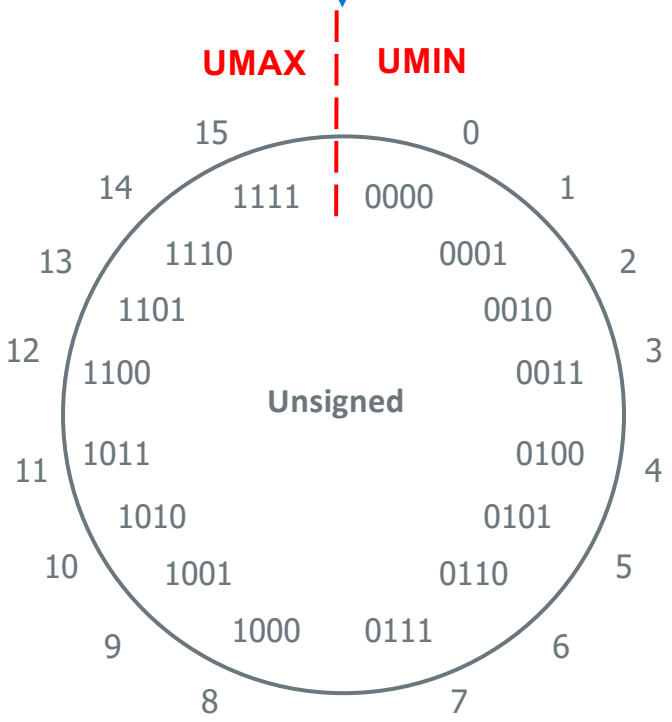
Subtraction Overflow: drops the borrow

```
   1
-  2
  -1
```

only 4 bits for numbers in this example

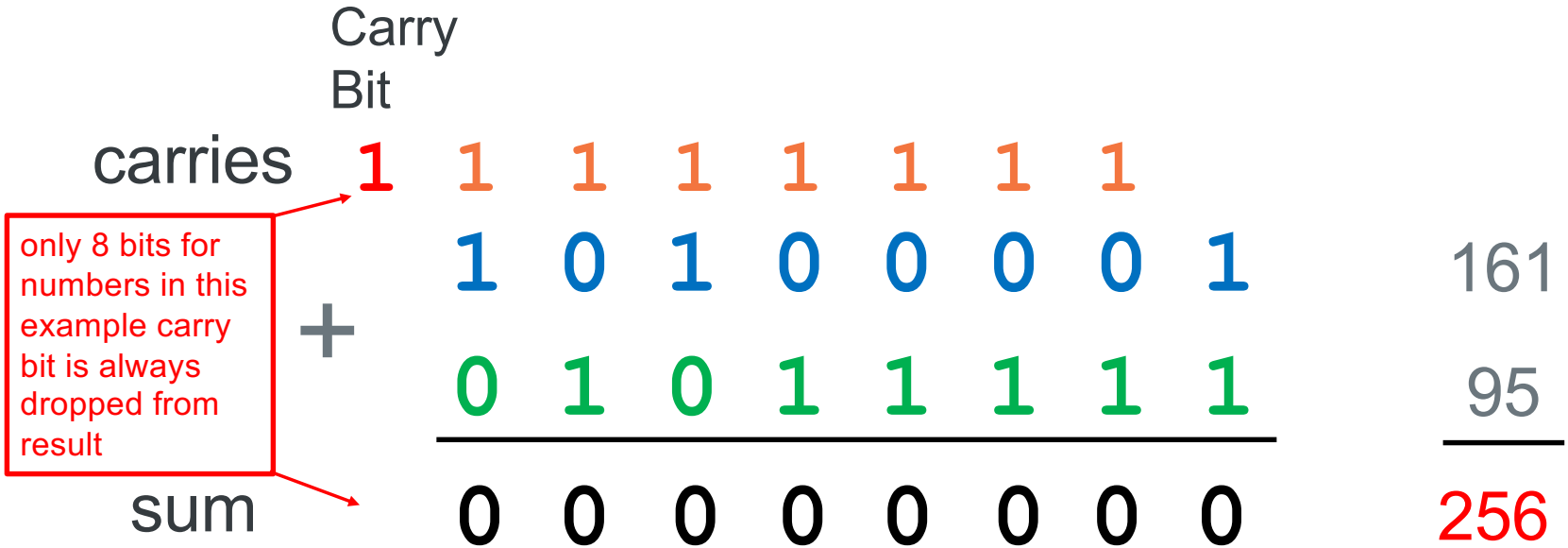
carry bit is always dropped from result

```
 10001
- 0010
  1111
oops 15
```



x

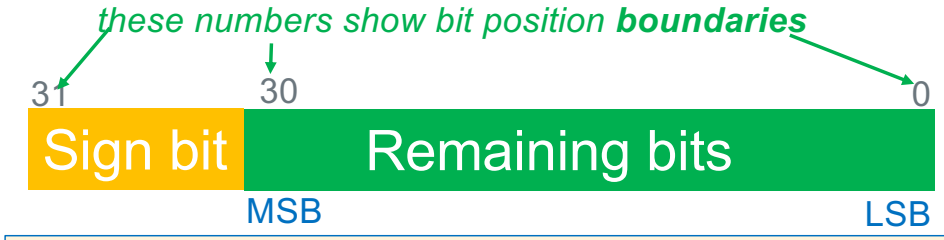
Unsigned Integer Number Overflow: Addition in 8 bits



Rule: When Carry Bit != 0, overflow has occurred for unsigned integers!

x

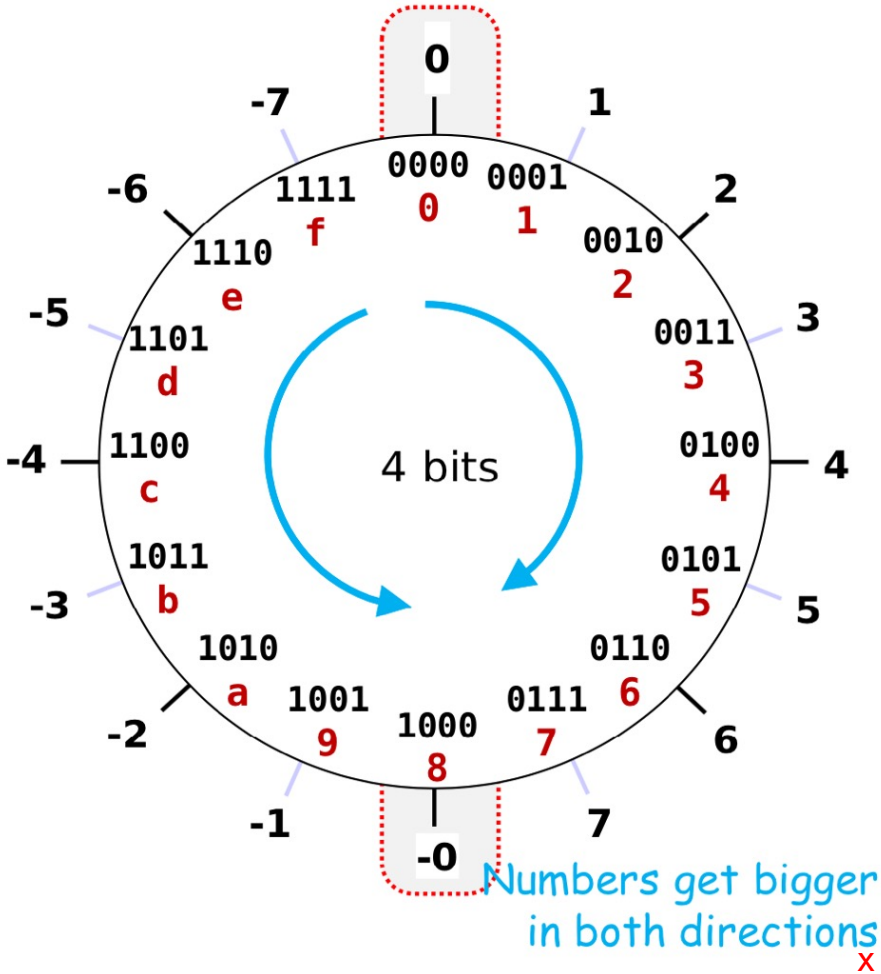
Negative Integer Numbers: Sign + Magnitude Method



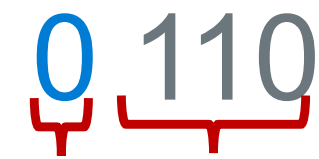
- Use the Most Significant Bit as a sign bit
  - 0 as the MSB represents positive numbers
  - 1 as the MSB represents negative numbers
- Two (oops) representations for zero: 0000, 1000
- Tricky Math (must handle sign bit independently)

```
  4        0100        4        0100
- 3      - 0011      + -3     + 1011
  1        0001        -7       1111
           ✓                    ✗
```

- With Simple math, Positive and Negatives "*increment*" (+1) in the opposite directions!

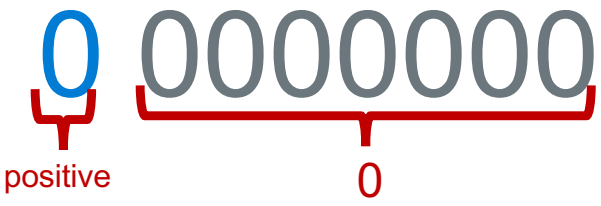


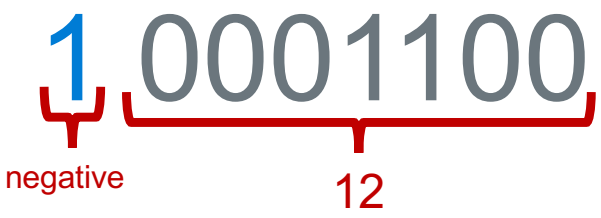
Signed Magnitude Examples (Sign bit is always MSB)



Examples (4 bits):

1 000 = -0?	0 000 = 0?
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7





Examples Using Hex notation (8 bits):

0x00 = 0b00000000 is positive, because the sign bit is 0

0x7F = 0b01111111 is positive ($+127_{10}$)

0x85 = 0b10000101 is negative (-5_{10})

0x80 = 0b10000000 is negative… also zero

x

Excess Bias Encoding (As used in floating point numbers)

- Given a number in E bits, to divide the range in about 1/2 the following is used:

  excess N bias = $(2^{E-1} - 1)$ *(this is just one of many bias formulas)*

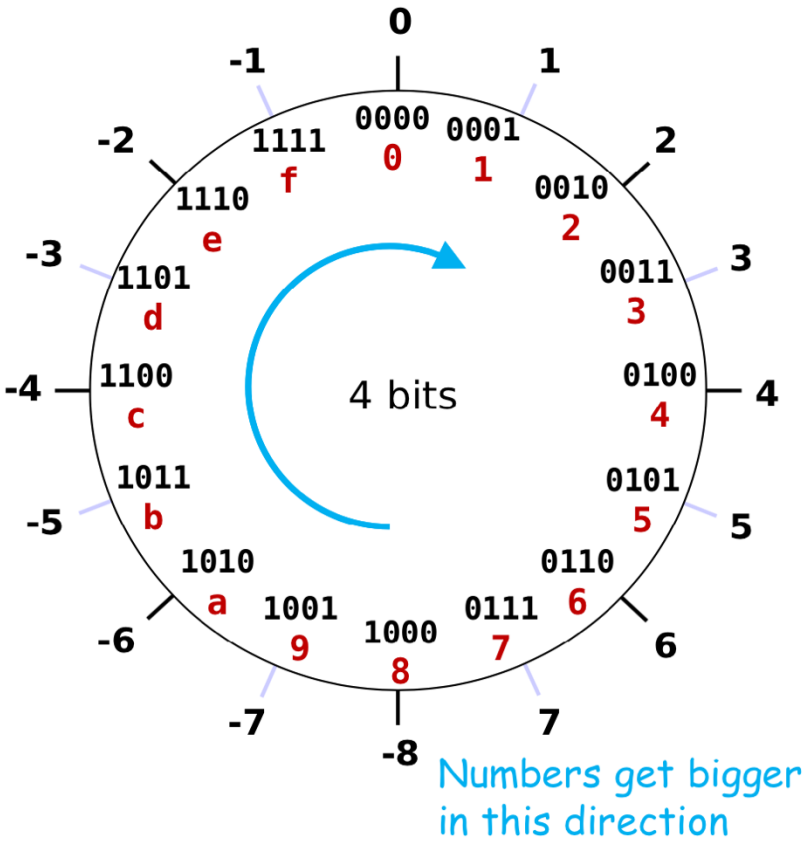
- **With this excess N Bias approach**: actual numbers range from most negative to most positive is: **-(bias) to bias+1**
- **So, for a number that is limited to** 4 bits (0 to 15 unsigned)
  - Then excess N bias = $2^{4-1} - 1 = 2^3 - 1$ = a bias of +7

actual	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
bias	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7
bias encoded	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

x

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 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!*



x

Two's Complement: The MSB Has a *Negative Weight*

$$2's\ Comp = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0$$

b_{n-1} weight is (-2^{n-1}), all other bits: have positive weights $(+2^i)$

- 4-bit (w = 4) weight = -2^{4-1} = -2^3 = -8
  - 1010_2 **unsigned**:
    $1x2^3 + 0x2^2 + 1x2^1 + 0x2^0$ = **10**
  - 1010_2 **two's complement**:
    $-1x2^3 + 0x2^2 + 1x2^1 + 0x2^0$ = -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**

x

Another Way to Look at 2's Complement Encoding

- A 2's compliment value can be thought of as using a slightly different **bias encoding** for negative numbers only (more negative values): -2^{w-1}
- The leftmost bit is then interpreted as a decision to apply the bias (if 1) or not (if 0)
  - 1 apply the bias
  - 0 do not apply the bias
- For example, for a 4-bit number (w = 4) , the negative number bias weight would be $= -2^{4-1} = -2^3 = -8$

2's	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
3 bit	000	001	010	011	100	101	110	111	000	001	010	011	100	101	110	111
decimal	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
+Bias	-8	-8	-8	-8	-8	-8	-8	-8	0	0	0	0	0	0	0	0
Actual	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7

Observe: adding +1 makes the number more positive for both negative and positive numbers

x

Summary: Min, Max Values: Unsigned and Two's Complement

Two's Complement → Unsigned for n bits

- **Unsigned Value Range**

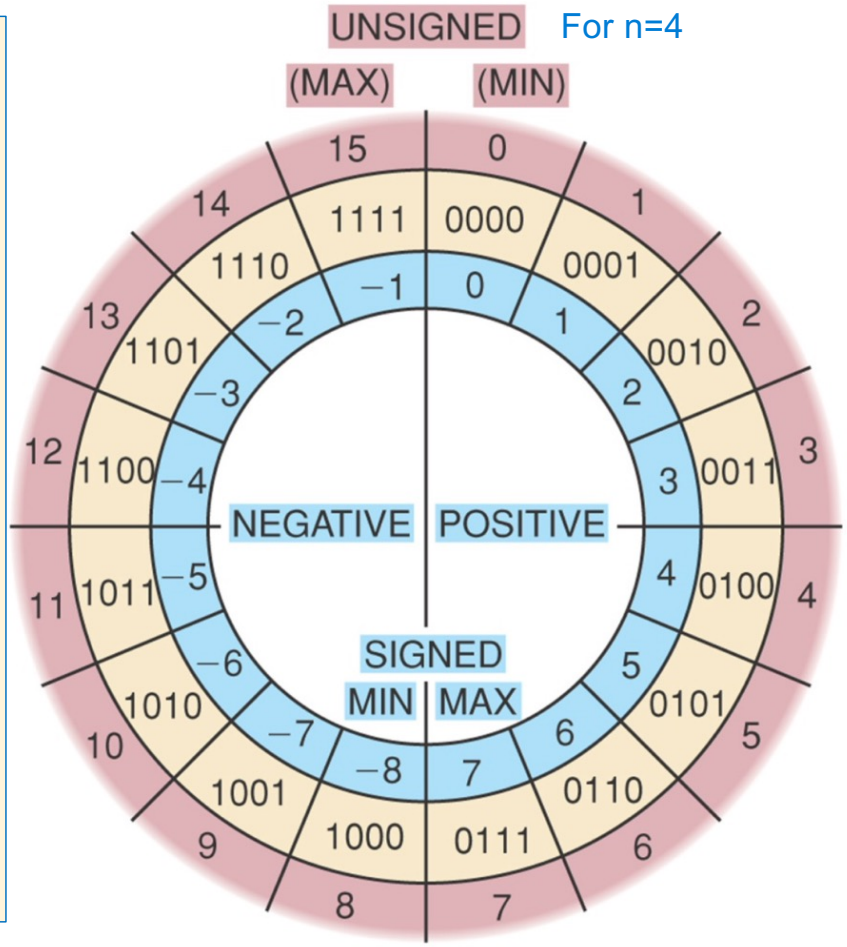
  UMin = 0b00…00
  = **0**

  UMax = 0b11…11
  = $2^n - 1$

- **Two's Complement Range**

  SMin = 0b10…00
  = -2^{n-1}

  SMax = 0b01…11
  = $2^{n-1} - 1$



x

Negation Of a Two's Complement Number (Method 1)

```
     7 = 0111
         ↓↓↓↓
invert = 1000
add 1  +    1
    -7   1001
```

```
    -7 = 1001
         ↓↓↓↓
invert = 0110
add 1  +    1
     7   0111
```

```
-x == ~x + 1;
```

```
 7 =          0111
-7 =     +    1001
(discard carry) 0000
```

```
     1 = 0001
         ↓↓↓↓
invert = 1110
add 1  +    1
    -1   1111
```

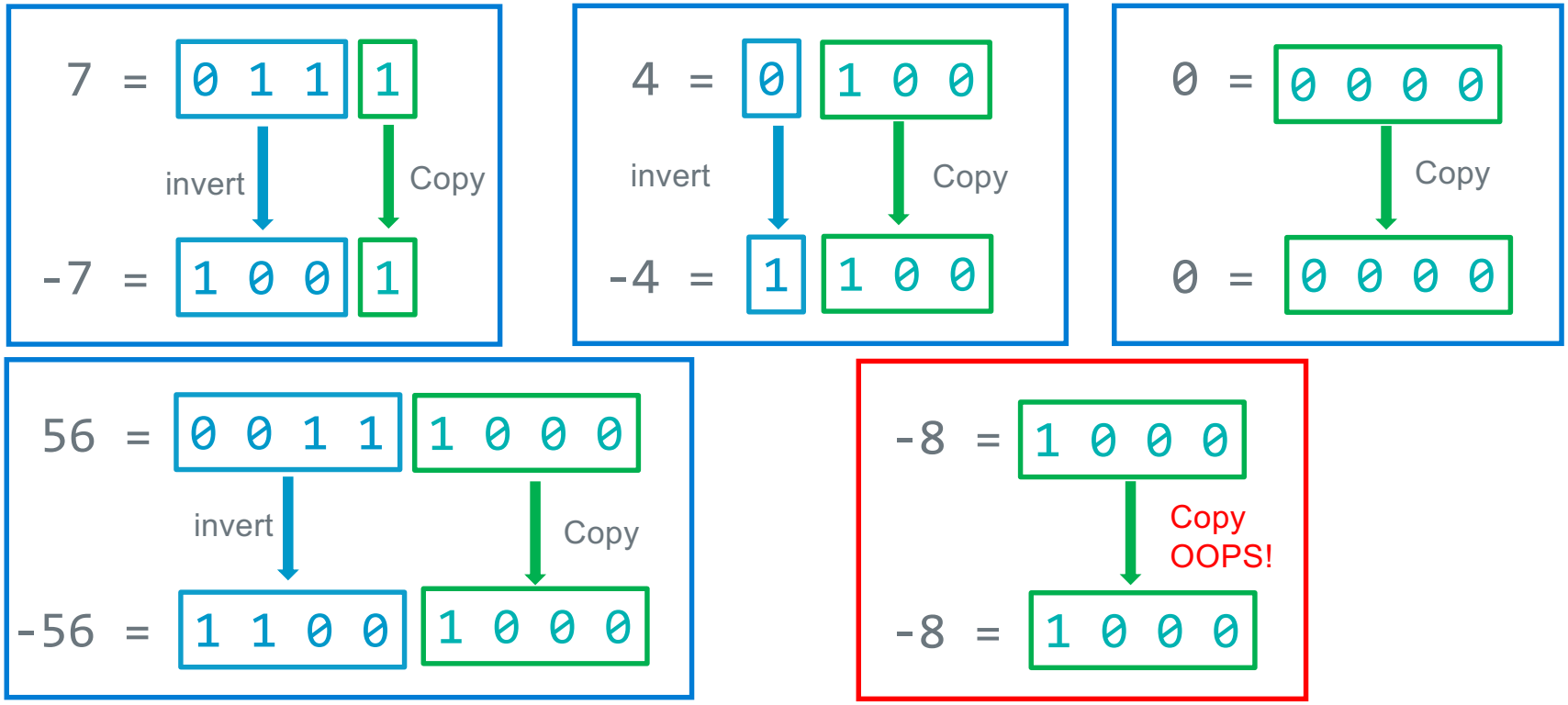
```
    -1 = 1111
         ↓↓↓↓
invert = 0000
add 1  +    1
     1   0001
```

```
    -8 = 1000
         ↓↓↓↓
invert = 0111
add 1  +    1
    -8   1000 oops!
```


x

Negation of a Two's Complement Number (Method 2)

1. **copy unchanged** right most bit containing a 1 and all the 0's to its right
2. Invert all the bits to the left of the right-most 1



x

Signed Decimal to Two's Complement Conversion

dividend -102	Quotient	Remainder	Bit Position
102/2	51	0	b0
51/2	25	1	b1
25/2	12	1	b2
12/2	6	0	b3
6/2	3	0	b4
3/2	1	1	b5
1/2	0	1	b6
0/2	0	0	b7

102(base 10) = $b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1 b_0$ = $0b0110\ 0110$

Get the two complement of 01100110 is 10011010

x

Two's Complement to Signed Decimal Conversion - Positive

What is $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ 0 1 1 0 0 1 0 1 (base 2) in decimal (N)?

Signed Bit Bias	Bit	Bit Position	Bias
$-2^{W-1} = -2^{8-1} = -128$	x 0	b7	0
Product Shift Left	Addend	Bit Position	Product
2 x 0 = 0 (shift left)	+ 1	~~b6~~	1
2 x 1 = 2	+ 1	~~b5~~	3
2 x 3 = 6	+ 0	~~b4~~	6
2 x 6 = 12	+ 0	~~b3~~	12
2 x 12 = 24	+ 1	~~b2~~	25
2 x 25 = 50	+ 0	~~b1~~	50
2 x 50 = 100	+ 1	b0	SUM = 101
		Bias + SUM:	0 + 101 = 101

x

Two's Complement to Signed Decimal Conversion - Negative

What is $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ **1 1 1 0 0 1 0 1**$_{(base\ 2)}$ in decimal (N)?

Signed Bit Bias	Bit	Bit Position	Bias
$-2^{W-1} = -2^{8-1} = -128$	x 1	b7	-128
Product Shift Left	Addend	Bit Position	Product
2 x 0 = 0 (shift left)	+ 1	~~b6~~	1
2 x 1 = 2	+ 1	~~b5~~	3
2 x 3 = 6	+ 0	~~b4~~	6
2 x 6 = 12	+ 0	~~b3~~	12
2 x 12 = 24	+ 1	~~b2~~	25
2 x 25 = 50	+ 0	~~b1~~	50
2 x 50 = 100	+ 1	b0	SUM = 101
		Bias + SUM:	-128 + 101 = -27

x

Two's Complement Addition and Subtraction

- **Addition:** just add the two number directly
- **Subtraction:** you can convert to addition: **difference = minuend – subtrahend**
  difference = minuend + 2's complement (subtrahend)

```
        Cout
          0  0  0  0  0  0  1  1
     x =  0  1  0  1  0  0  1  1
     y =  0  0  0  0  1  0  1  1
x + y =  0  1  0  1  1  1  1  0
```

```
     x =  0  1  0  1  0  0  1  1
     y =  0  0  0  0  1  0  1  1
x-y   =  0  1  0  0  1  0  0  0
```

2's complement first and then add

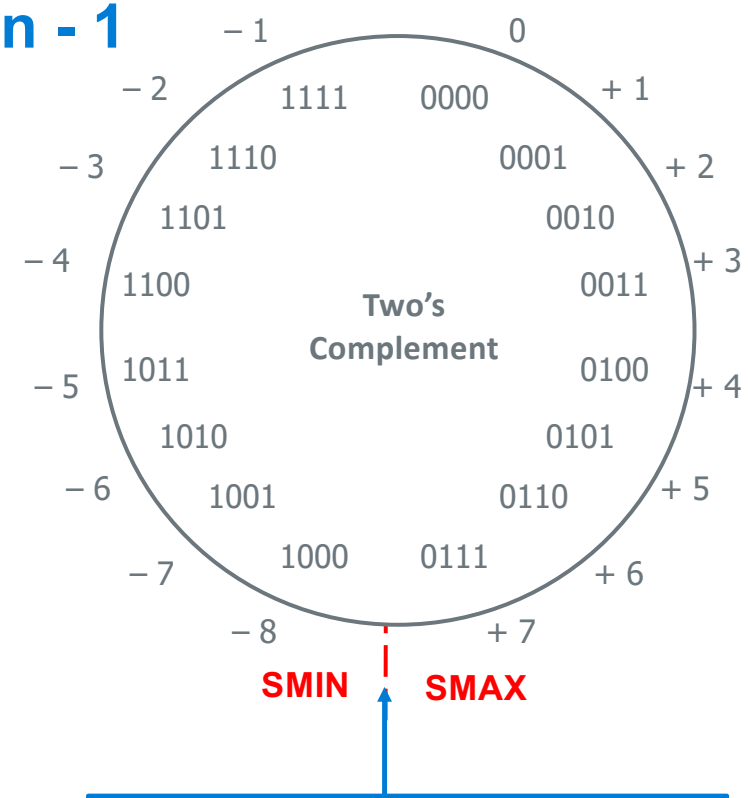
```
          x =  0  1  0  1  0  0  1  1
+ (-y)  =  1  1  1  1  0  1  0  1
x - y = x +(-y) =  0  1  0  0  1  0  0  0
```


x

Two's Complement Overflow Detection - 1

- When adding two positive numbers or two negative numbers
- **4-bit** Two's complement numbers (positive overflow)

```
Cout Cin
 0    1   0   0
     0   1   0   1          5
   + 0   1   1   0          6
   ---------------------------
     1   0   1   1   -5   != 11
```



Overflow: Occurs when an arithmetic result is beyond the min or max limits

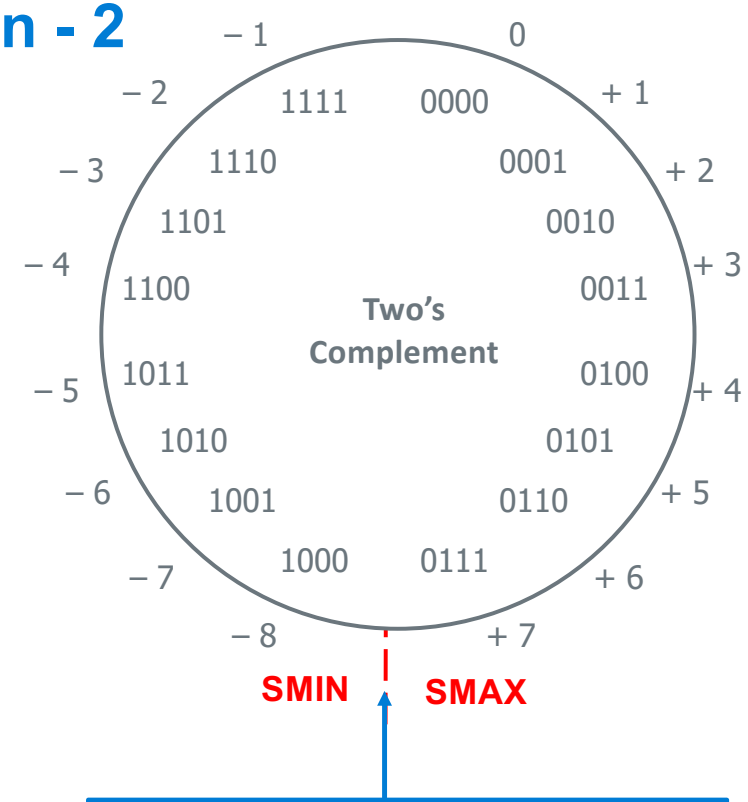
x

Two's Complement Overflow Detection - 2

- When adding two positive numbers or two negative numbers
- **4-bit** Two's complement numbers (negative overflow)

carry bit is dropped from result

```
Cout Cin
 1    0  1  1
      1  0  0  1        -7
   +  1  0  1  1        -5
   ------------------------
      0  1  0  0   +4   != -12
```



Result is correct **ONLY** when the **carry into** the sign bit position (MSB) equals the **carry out** of the sign bit position (MSB)

Overflow: Occurs when an arithmetic result is beyond the min or max limits

x

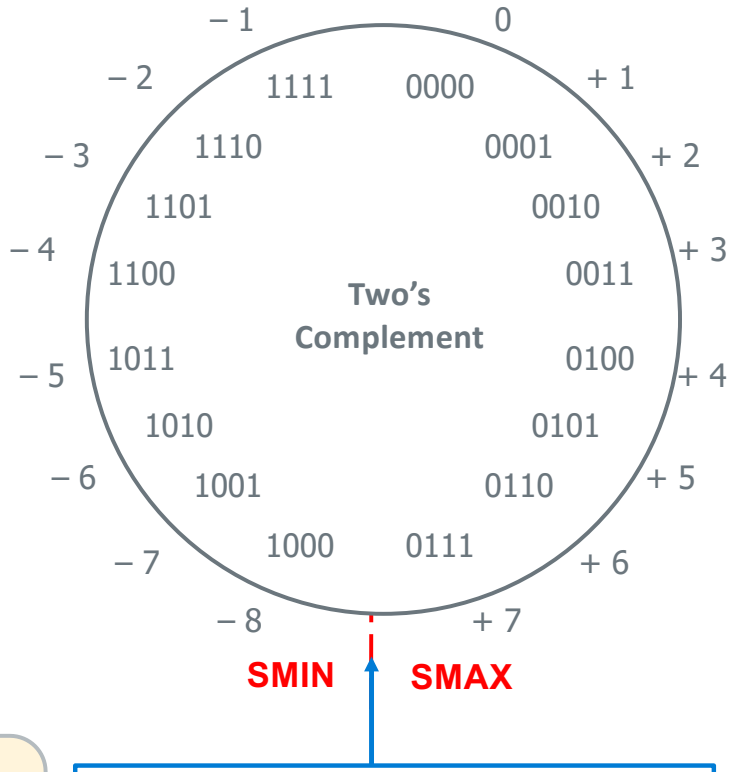
Two's Complement Alternative Overflow Detection

- **Addition:** (+) + (+) = (–) huh?

```
   6        0110
 + 3      + 0011
 ---      ------
   9        1001
          oops -7
```

- **Subtraction:** (–) + (–) = (+) huh?

```
  -7        1001
 - 3      + 1101
 ---      ------
 -10        0110
          oops 6
```



Another Way to look at it for signed numbers:
overflow occurs if
operands have same sign and result's sign is different

Overflow: Occurs when an arithmetic result is beyond the min or max limits

x

Summary: When Does Overflow Occur

Operand 1

+ Operand 2

Result

Operand 1 Sign	Operand 2 Sign	Is overflow Possible?
+	+	YES
–	–	YES
+	–	NO
–	+	NO

x

Sign Extension 2's complement number

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

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

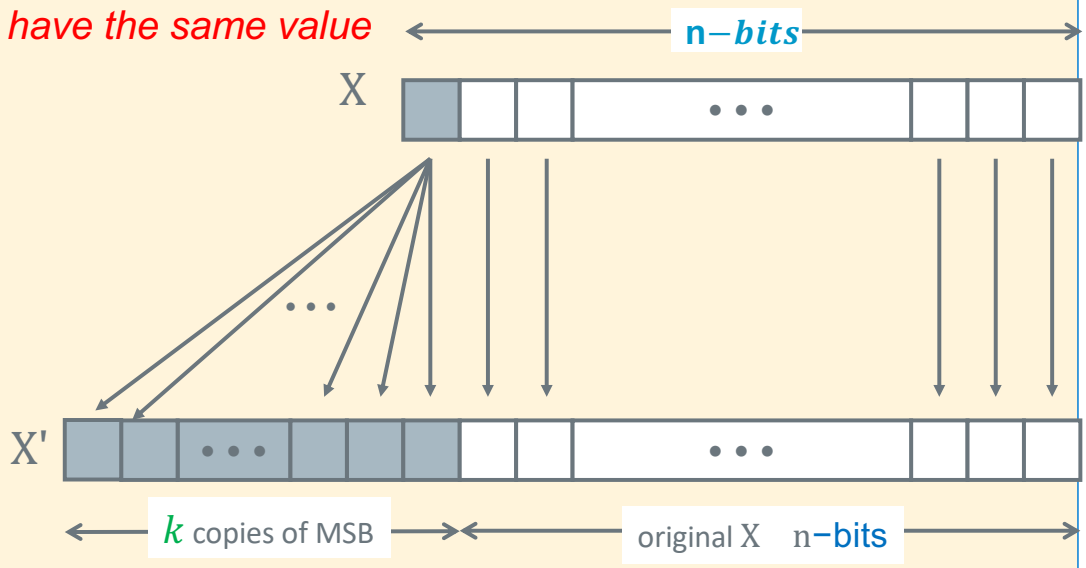
- **Sign extension increases the number of bits: n-bit** wide signed integer X, ***EXPANDS*** to a ***wider*** n−bit + k-bit signed integer X′ where *both have the same value*

Unsigned

- Just add leading zeroes to the left side

Two's Complement Signed:

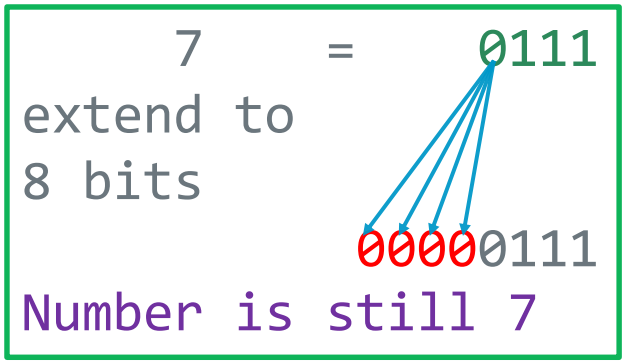
- If positive, add leading zeroes on the left
  - Observe: Positive stay positive
- If negative, add leading ones on the left
  - Observe: Negative stays negative

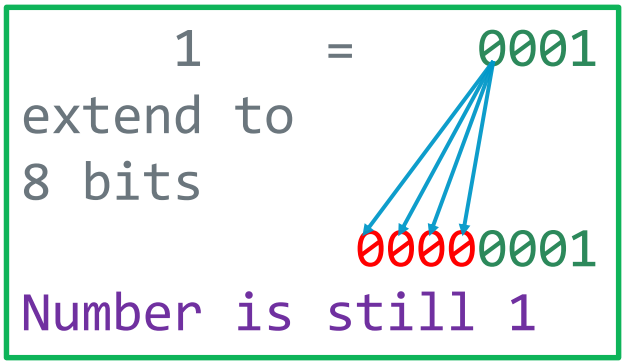


x

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

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





x

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

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

```
      7 = 0111
          ||||
invert = 1000
add 1  +    1
     -7   1001
```

```
    -7    =    1001
extend to
8 bits
             11111001
```

```
      7 = 00000111
          ||||||||
invert = 11111000
add 1  +        1
     -7   11111001
```


x

Example: Two's Complement Sign or bit Extension - 3

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

```
     1 = 0001
         ||||
invert = 1110
add 1  +    1
    -1   1111
```

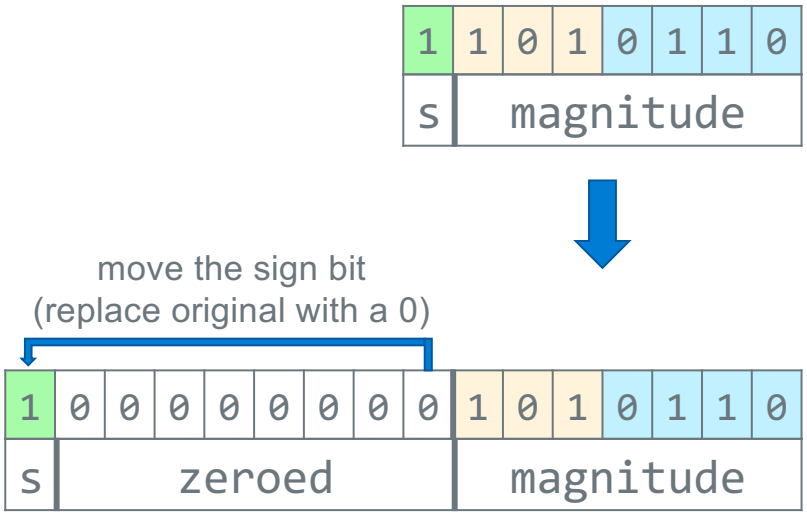
```
    -1    =          11111111
extend to
16 bits
          11111111 11111111
```

```
    -1    =     1111
extend to
8 bits
          11111111
```


x

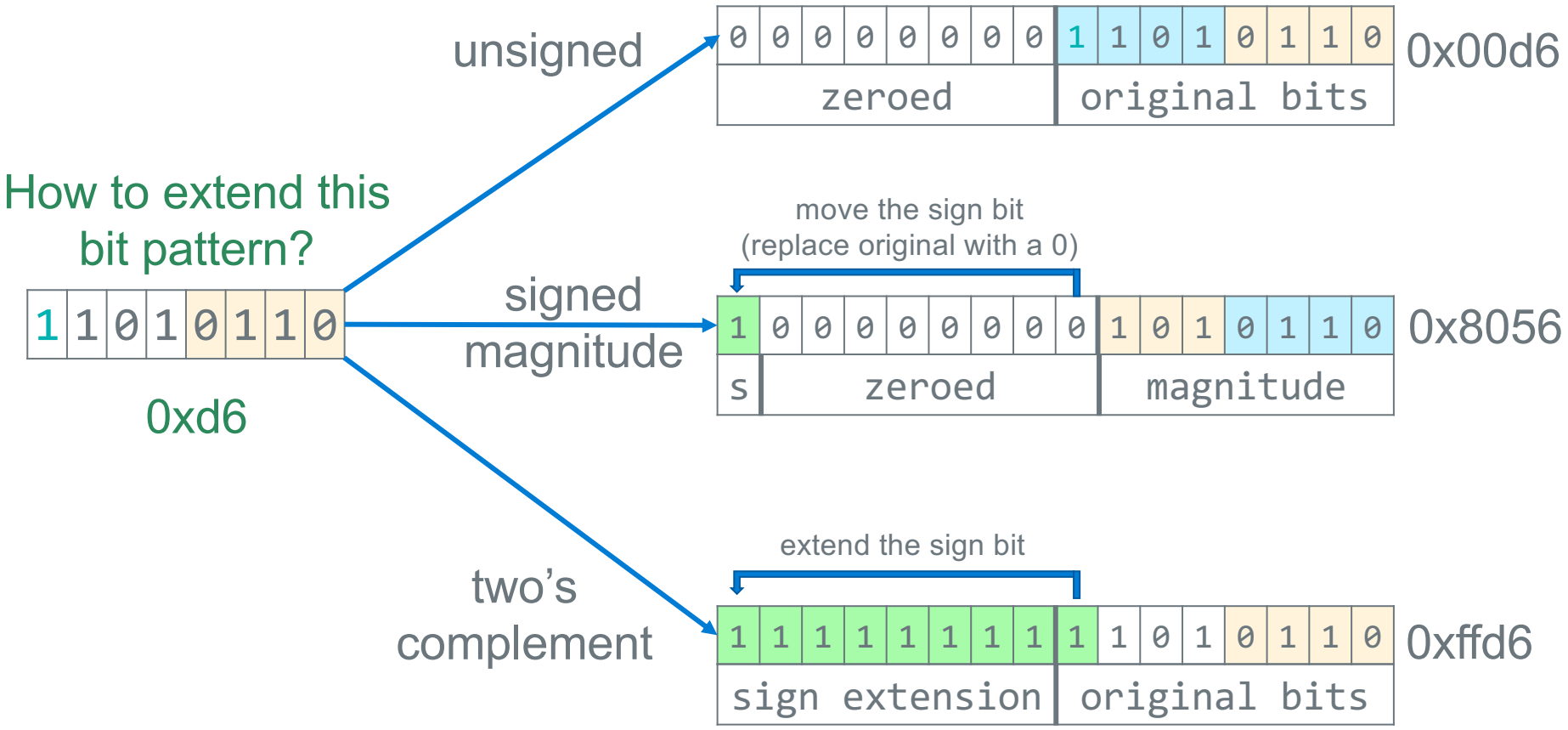
Sign Extension Signed Magnitude number

- Just move the sig bit and expand the magnitude with zeros to the left



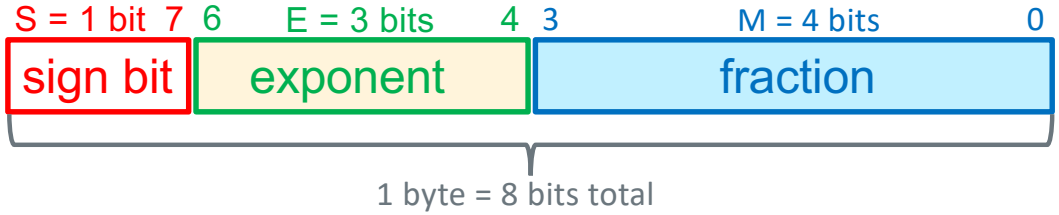
x

Interpreting and extending with Different representations



x

Floating Point Number in a Byte (Not A Real Format)

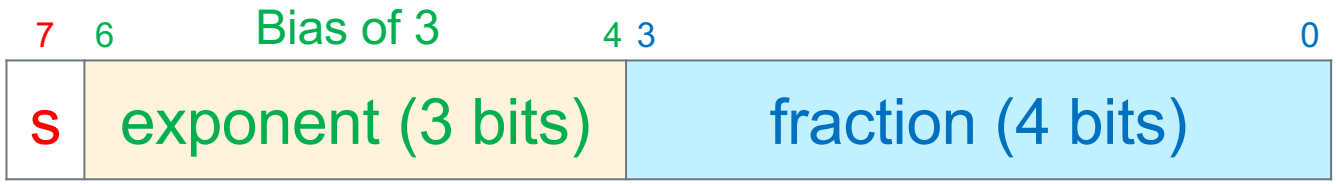


- **Mantissa encoding: = 1.[xxxx] encoded as an unsigned value**
- **Exponent encoding:** 3 bits encoded as an unsigned value using bias encoding
  - **Use the following variation of Bias encoding = $(2^{E-1} - 1)$**
  - 3 bits for the bias we have $2^{3-1} - 1 = 2^2 - 1$ = a bias of 3
  - **With a Bias of 3**: positive and negative numbers range: small to large is: 2^{-3} to 2^4

Actual	-3	-2	-1	0	1	2	3	4
Bias	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+3
Biased	0	1	2	3	4	5	6	7

x

Decimal to Float



Binary	Decimal
2^{-2}	0.25
2^{-3}	0.125

Step 1: convert from base 10 to binary (absolute value)

$-0.375_{(decimal)} = 0000.0110_{base\ 2}$

Step 2: Find out how many places to shift to get the number into the normalized 1.xxxx mantissa format

$0000.0110_2 = 1.1000 \times (2^{-2})_{base\ 10}$

exponent: $-2_{10} + bias\ of\ 3_{10} = 1_{10} = 0b001$ for the exponent (after adding the bias)

Step 3: Use as many digits as possible to the right of the decimal point in the fractional .xxxx part

1.1000

Step 4: Sign bit

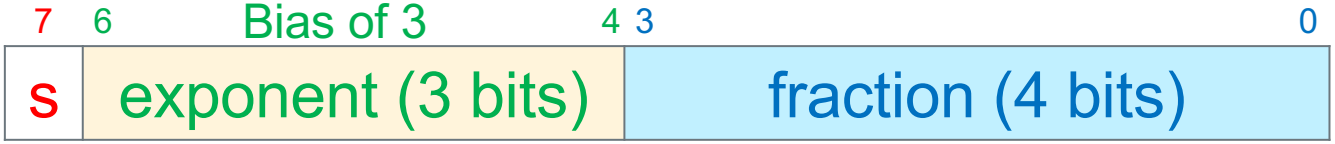
positive sign bit is 0

negative sign bit is 1

s	exponent	fraction
1	0b001	0b1000
0x9		0x8

$= 0x98$

Float to Decimal



Step 1: Break into binary fields

0x45 =

0x4		0x5
s	exponent	fraction
0	0b100	0b0101

Step 2: Extract the unbiased exponent

$0b100 = 4_{base\ 10}$ - bias of $3_{10} = 1_{10}$ for the exponent (bias removed)

Step 3: Express the mantissa (restore the hidden bit)

1.0101

Step 4: Apply the **unbiased** exponent

$1.0101_{base\ 2}$ x $(2^1)_{base\ 10} = 10.101$

Step 5: Convert to decimal

$10.101 = 2.625_{base\ 10}$

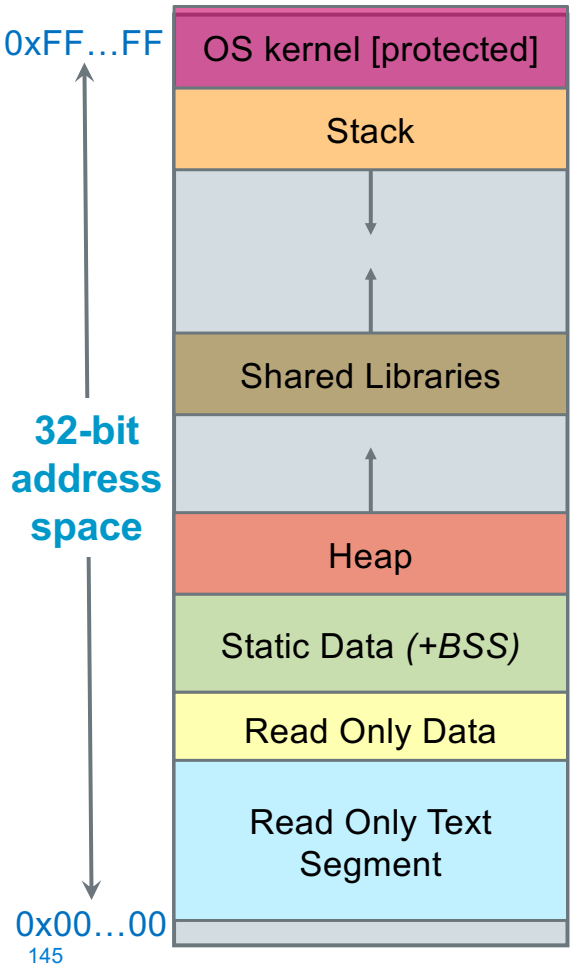
Step 6: Apply the Sign

+ $2.625_{base\ 10}$

Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625

x

Assembly and Machine Code



- Machine Language (or code): Set of instructions the CPU executes are encoded in memory using patterns of ones and zeros
- Assembly language is a symbolic version of the machine language
- Each assembly statement (called an **Instruction**), executes exactly **one** from a list of simple commands
  - Instructions describe operations (e.g., =, +, -, *)
- Each line of arm32 assembly code contains at most one instruction
- Assembler (gnu as) translates assembly to machine code

Memory Address	word (4-bytes) contents	Assembly Language
1040c:	e28db004	add fp, sp, 4
10410:	e59f0010	ldr r0, [pc, 16]
10414:	ebffffb3	bl 102e8 <printf>
10418:	e3a00000	mov r0, 0
1041c:	e24bd004	sub sp, fp, 4

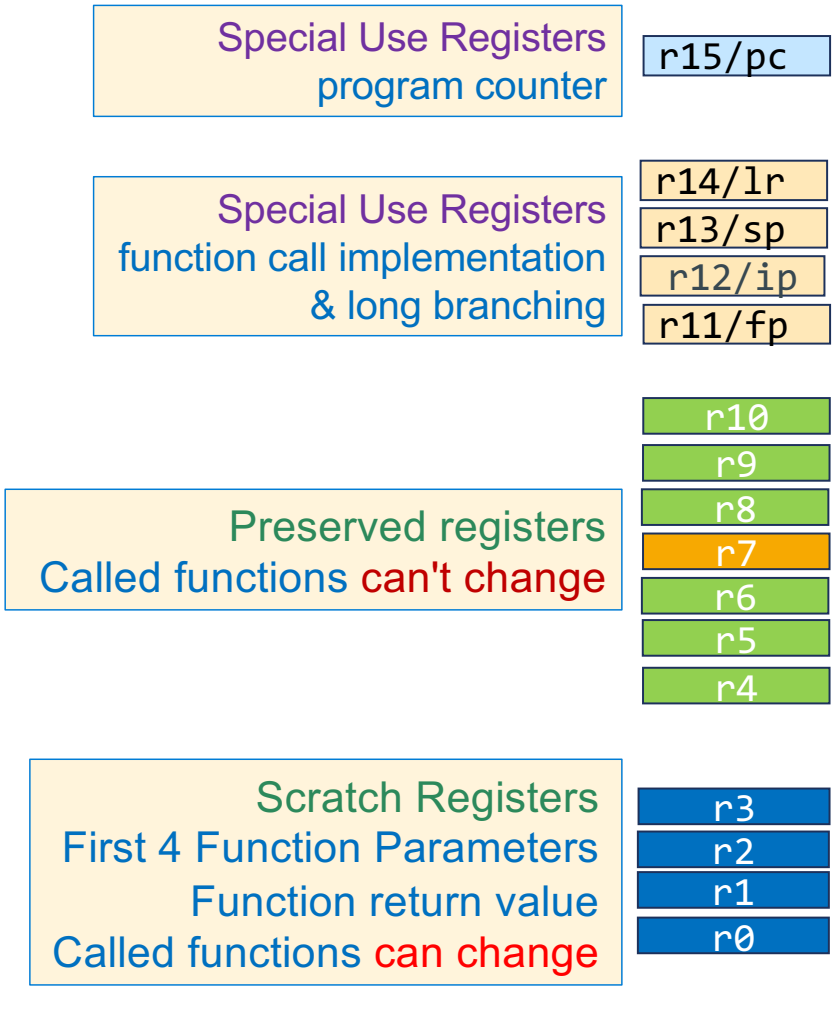
Machine Code

high <- low bytes

x

Using Aarch32 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: dedicated hardware use (like r15 the pc) or special use when used with certain instructions (like r13 & r14)
- r15/pc is the program counter that contains the address of an instruction being executed (not exactly … later)



x

How r1 (pc) is used: CPU Operational Overview

Everything has a **memory address**: **instructions** & **data**

- Machine code uses addresses for loops, branches, function calls, variables, etc.

1 Fetch

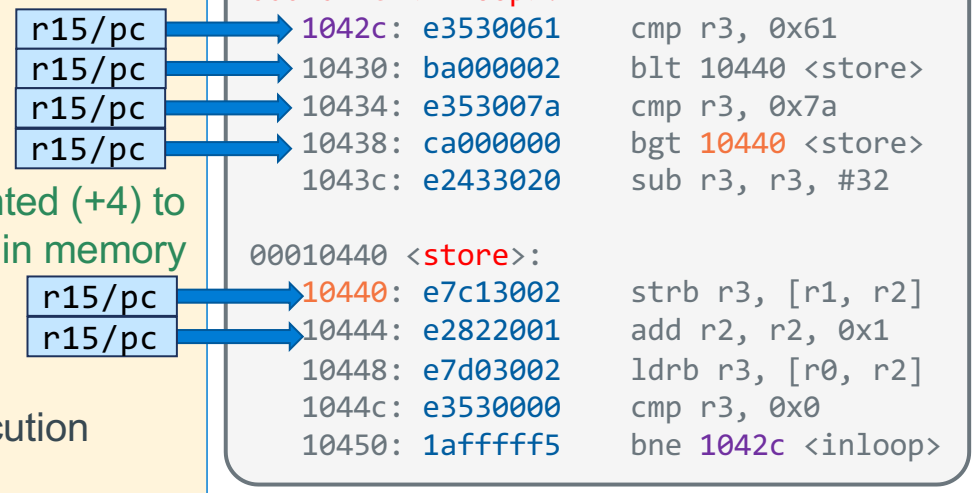
- read the instruction into memory ***(fetch)***
- program counter is automatically incremented (+4) to contain the address of the next instruction in memory
  Instructions are 32 bits

2 Decode

- ***Decodes the instruction*** and sets up execution

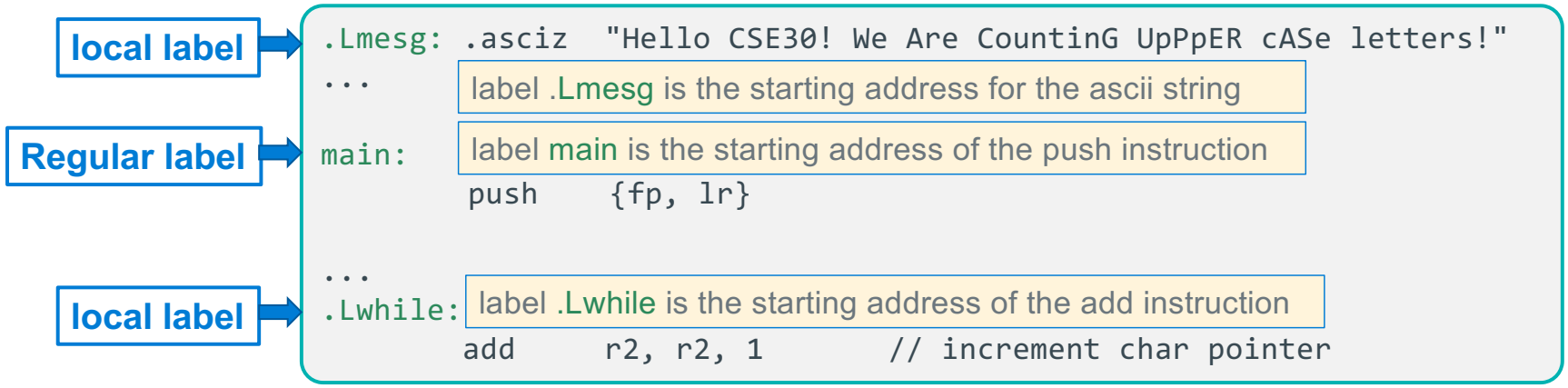
3 Execute

- CPU completes the ***execution*** of the instruction
- Execution may alter the pc to take branches, etc.
- Go to **fetch**



x

Labels in Arm Assembly



- Remember, a Label associates a name with memory location
- **Regular Label:**
  - Used with a Function name (label) or for 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 (string not foo in `char *foo = "anonymous variable"`)
  3. Read only literals when allocated in the text segment – special case)

x

Assembler Directives: .equ and .equiv

```
        .equ    BLKSZ,  10240      // buffer size in bytes
        .equ    BUFCNT, 100*4      // buffer for 100 ints
        .equiv  STRSZ,  128        // buffer for 128 bytes
        .equiv  STRSZ,  1280       // ERROR! already defined!
        .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

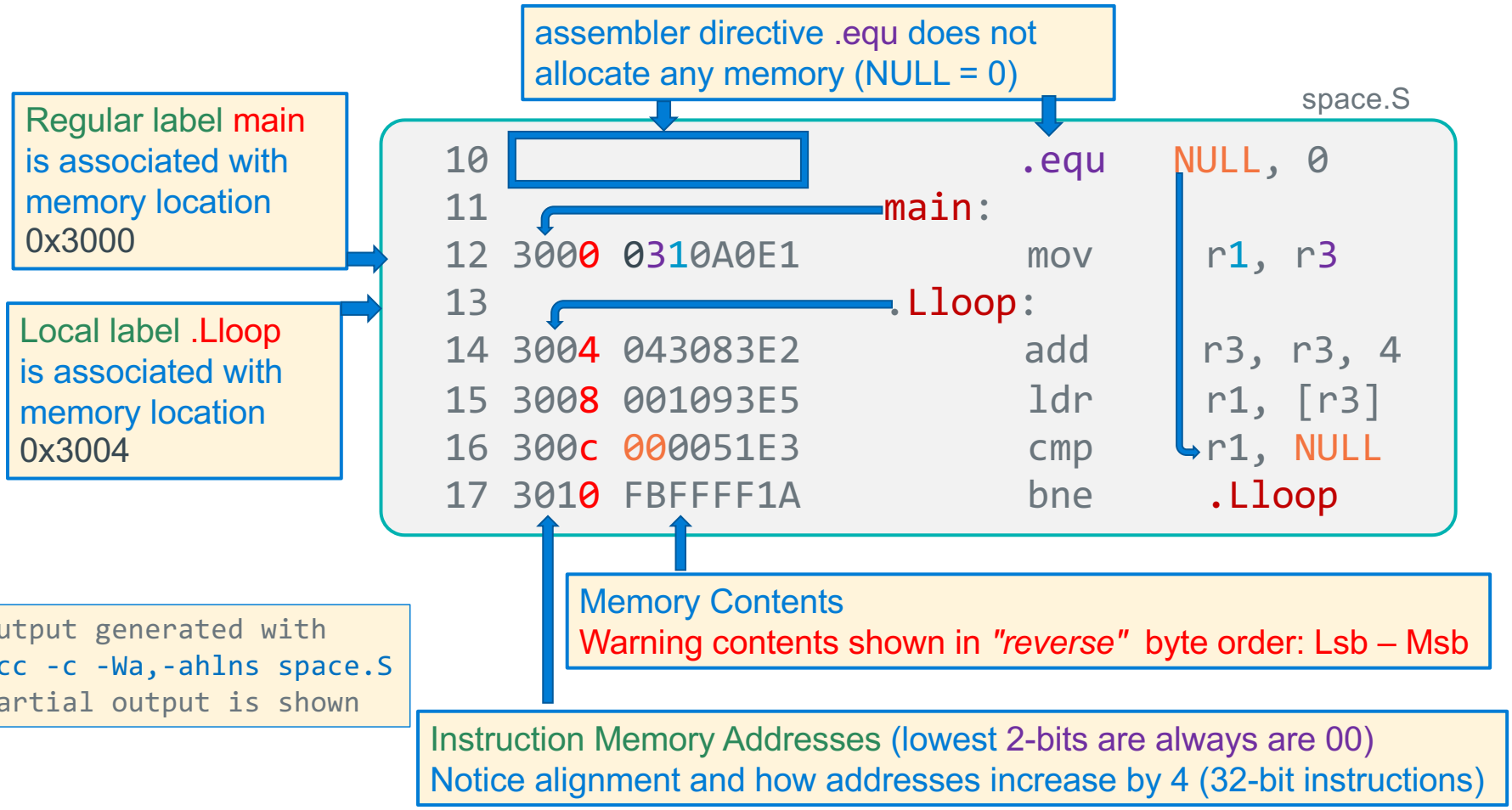
```
        .equ    BLKSZ,  10240      // buffer size in bytes
        // other lines
        .equ    BLKSZ,  1024       // buffer size in bytes
```

`.equiv <symbol>, <expression>`

- **`.equiv`** directive is like **`.equ`** except that the assembler will signal an error if symbol is already defined

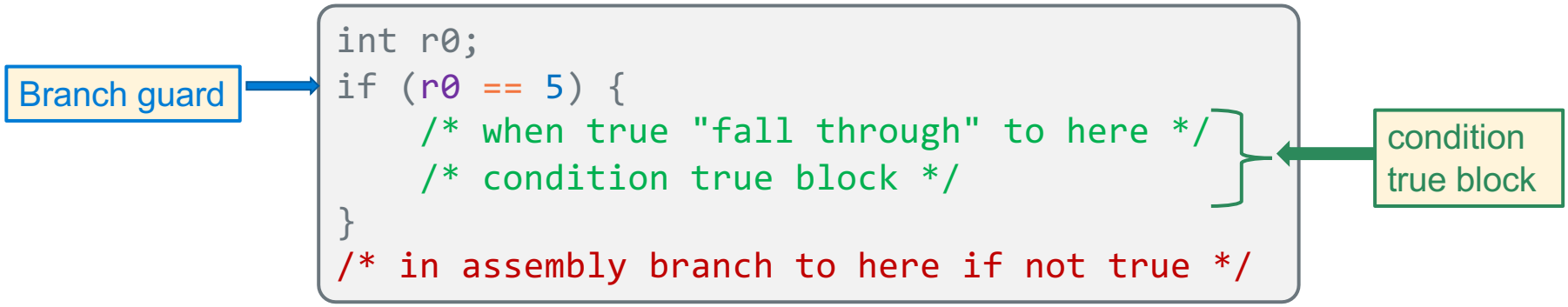
x

Example: Assembler Directive and Instructions



x

Program Flow: Keeping the same *"Block Order"* as C



- In ARM32, you either **fall through** (execute the *next instruction in sequence*) or **branch to a specific instruction** and then *resume* sequential instruction execution
- In order to keep the **same block order** as the **C version** that says: ***fall through*** to the condition **true** block when the branch guard evaluates to be **true**
  - Assembly: ***invert*** the **condition test** to **branch around** the ***condition true*** block
- Summary: In ARM32 use a **condition test** that **specifies the opposite of the condition used in C , then branch around** the **condition true** block

x

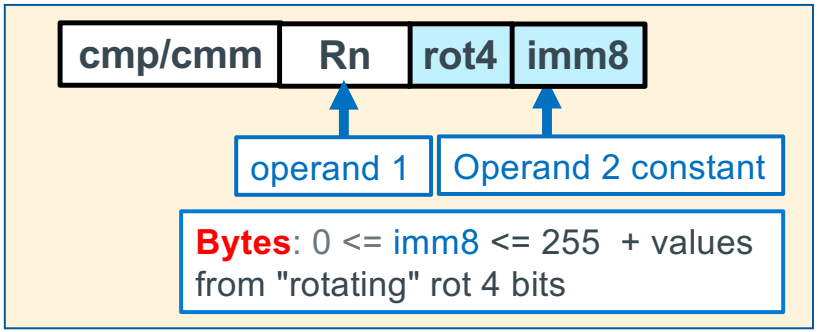
Examples: Guards (Conditional Tests) and their Inverse

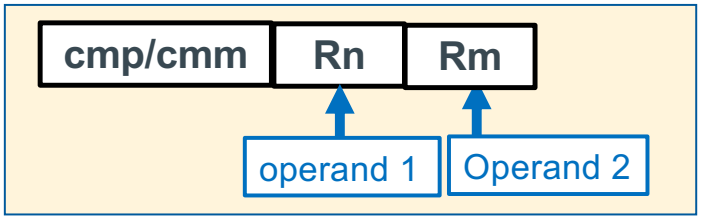
Compare in C	*"Inverse"* Compare in C
==	!=
!=	==
>	<=
>=	<
<	>=
<=	>

- Changing the conditional test (guard) to its inverse, allows you to swap the order of the blocks in an if else statement

x

cmp/cmm – Making Conditional Tests






```
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 **cmn** 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
```


x

Quick Overview of the Condition Bits/Flags

- The CSPR is a special register (like the other registers) in the CPU
- The four bits at the left are called the Condition Code flags
  - **Summarize** the **result** of a previous instruction
  - Not all instruction will change the CC bits
- **Specifically, Condition Code flags are set** by cmm/cmp (and others)

```
Example:            cmp    r4, r3
```

- **N** (Negative) **flag**: Set to 1 when the result of r4 – r3 is negative, set to 0 otherwise
- **Z (Zero) flag:** Set to 1 when the results of r4 – r3 is 0, set to 0 otherwise
- **C (Carry bit) flag:** Set to 1 when r4 – r3 does not have a borrow, set to 0 otherwise
- **V flag** (oVerflow): Set to 1 when r4 – r3 causes an overflow, set to 0 otherwise

x

Conditional Branch: Changing the Next Instruction to Execute

Branch instruction

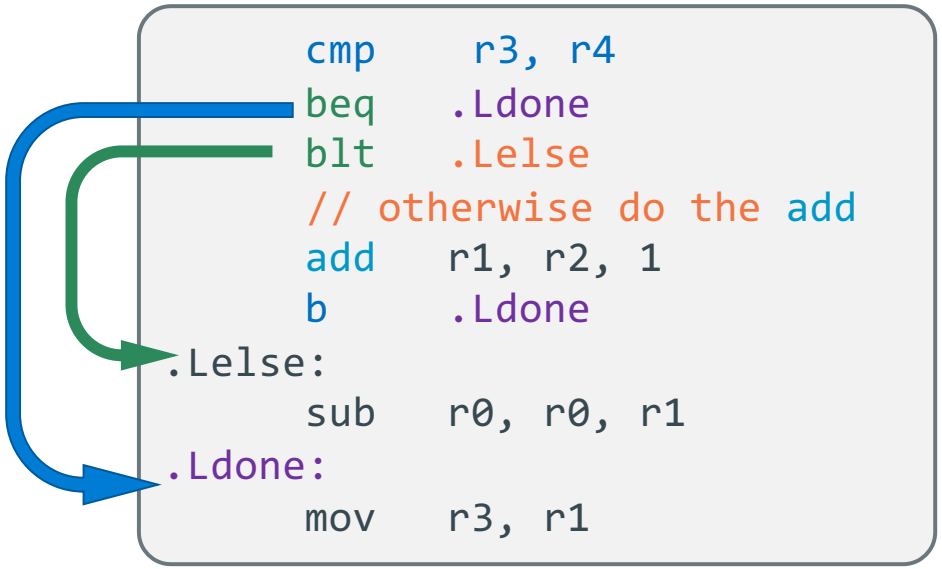
```
bsuffix .Llabel
```

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

Condition	*Meaning*	*Flag Checked*
BEQ	Equal	Z = 1
BNE	Not equal	Z = 0
BGE	Signed ≥ ("Greater than or Equal")	N = V
BLT	Signed < ("Less Than")	N ≠ V
BGT	Signed > ("Greater Than")	Z = 0 && N = V
BLE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
BHS	Unsigned ≥ ("Higher or Same") or Carry Set	C = 1
BLO	Unsigned < ("Lower") or Carry Clear	C = 0
BHI	Unsigned > ("Higher")	C = 1 && Z = 0
BLS	Unsigned ≤ ("Lower or Same")	C = 0 || Z = 1
BMI	Minus/negative	N = 1
BPL	Plus - positive or zero (non-negative)	N = 0
BVS	Overflow	V = 1
BVC	No overflow	V = 0
B (BAL)	Always (unconditional)	

x

Conditional Branch: Changing the Next Instruction to Execute



```
        cmp    r3, r4
        beq    .Ldone
        blt    .Lelse
        // otherwise do the add
        add    r1, r2, 1
        b      .Ldone
.Lelse:
        sub    r0, r0, r1
.Ldone:
        mov    r3, r1
```

Conditi on	*Meaning*	*Flag Checked*
BEQ	Equal	Z = 1
BLT	Signed < ("Less Than")	N ≠ V
B	Always (unconditional)	

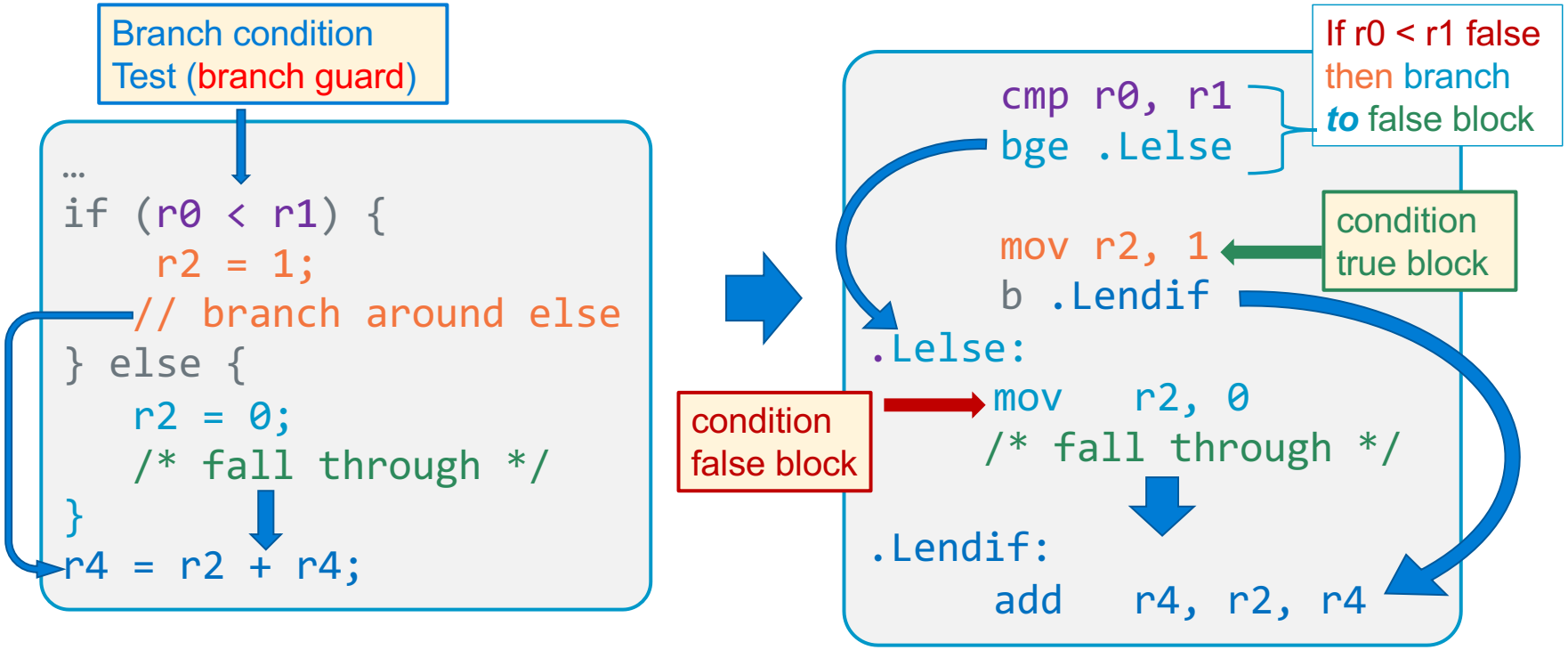
```
cmp    r3, r4 // r3 – r4
// if r3 == r4 sets Z = 1
// if r3 < r4 sets N and V; is N == V?
```

Two steps to do a Conditional Branch: Use a **cmp/cmm** instruction to set the condition bits

1. Follow the **cmp/cmm with one or more variants of the conditional branch instruction Conditional branch instructions** if evaluate to true (bases on the CC bits set) will go to the instruction with the branch label. Otherwise, it executes the instruction that follows
2. You can have one or more conditional branches after a single cmp/cmm

x

If with an Else Examples



x

Branching: Use Fall through!

Avoid goto like structure

- Do not use unnecessary branches (sometimes called a “goto” like structure) when a “fall through” works
- You can see this by structures that have a **conditional branch around an unconditional branch that immediately follows it**

Do not do the following:

```
    cmp r0, 0
    beq .Lthen
    b .Lendif
.Lthen:
    add r1, r1, 1
.Lendif:
    add r1, r1, 2
```

Not good!
Two adjacent branches

Do the following:

```
    cmp r0, 0
    bne .Lendif  // fall through
    add r1, r1, 1
.Lendif:
    add r1, r1, 2
```


x

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 (x || ++x) // true block always executed: ++x!
    printf("%d\n", x);
```

- 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();
```


x

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 */
}
r4 = r3;
```

```
    cmp r0, 5  // test 1
    bne .Lelse

    cmp r1, 3  // test 2
    ble .Lelse

    mov r2, r5 // true block
    // branch around else
    b .Lendif
.Lelse:
    mov r5, r2 //false block
    // fall through
.Lendif:
   mov r4, r3
```

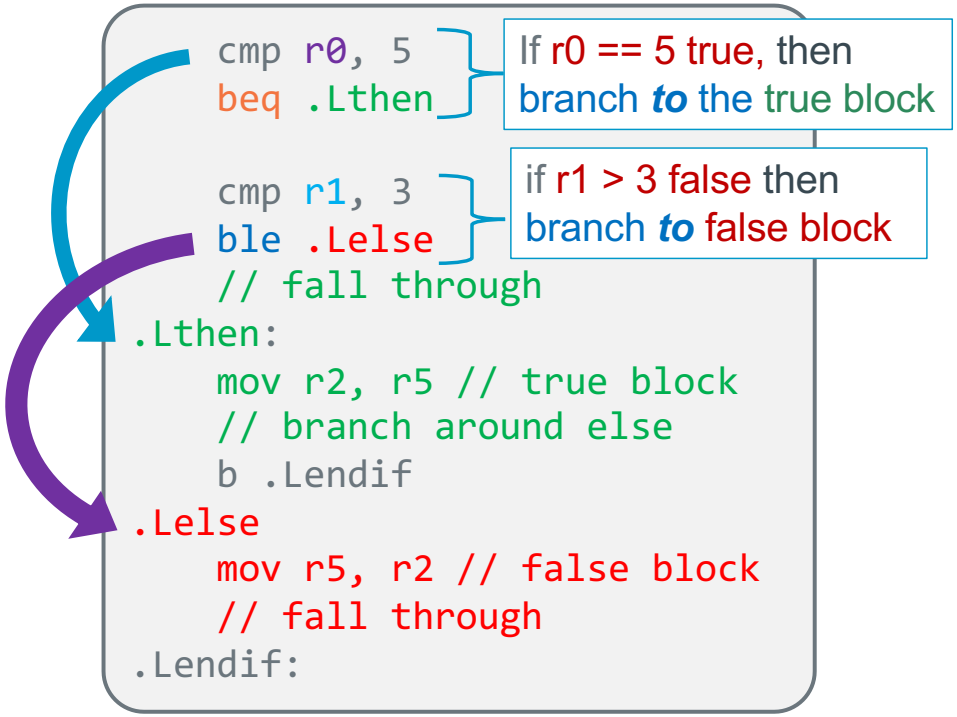
if r0 == 5 false then short circuit branch ***to*** the false block

if r1 > 3 false then branch ***to*** the false block

x

Program Flow – If statements || compound tests - 2

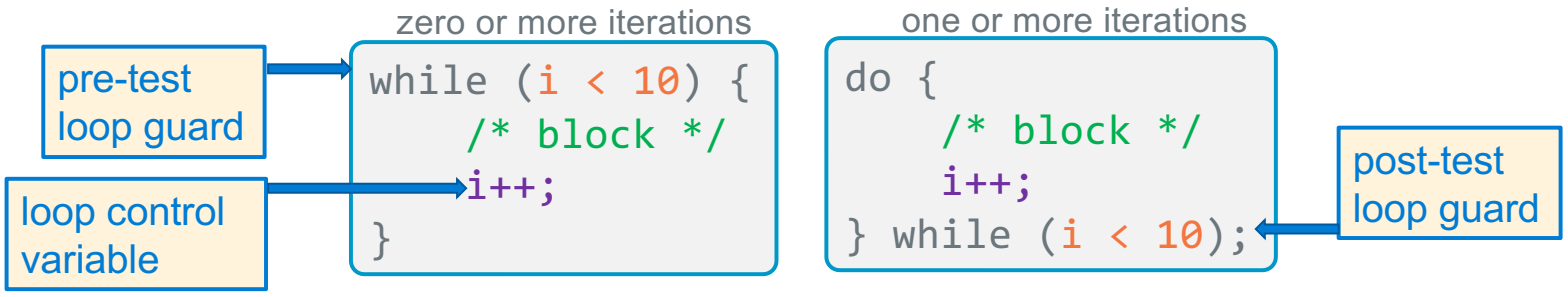
```
if ((r0 == 5) || (r1 > 3)) {
    r2 = r5; // true block
    /* branch around else */
} else {
    r5 = r2; // false block
    /* fall through */
}
```



x

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 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

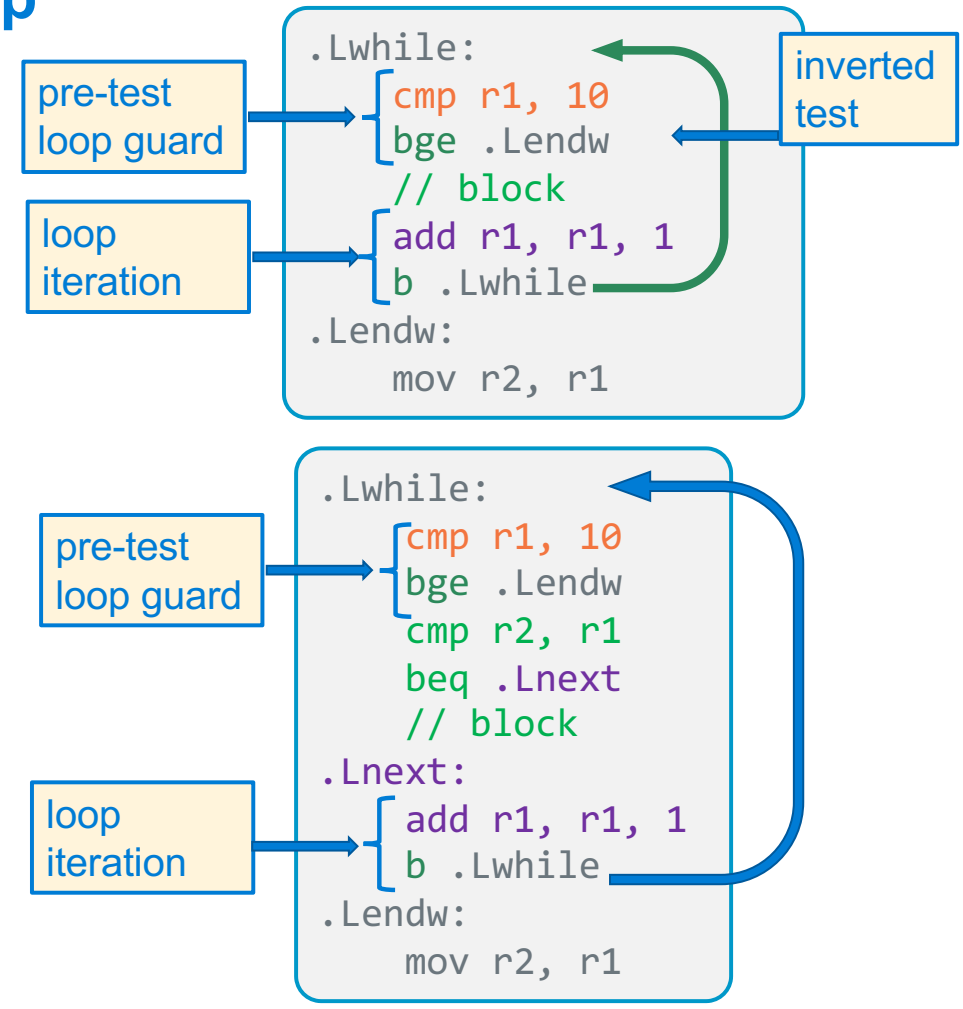


x

Pre-Test Guards - While Loop

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

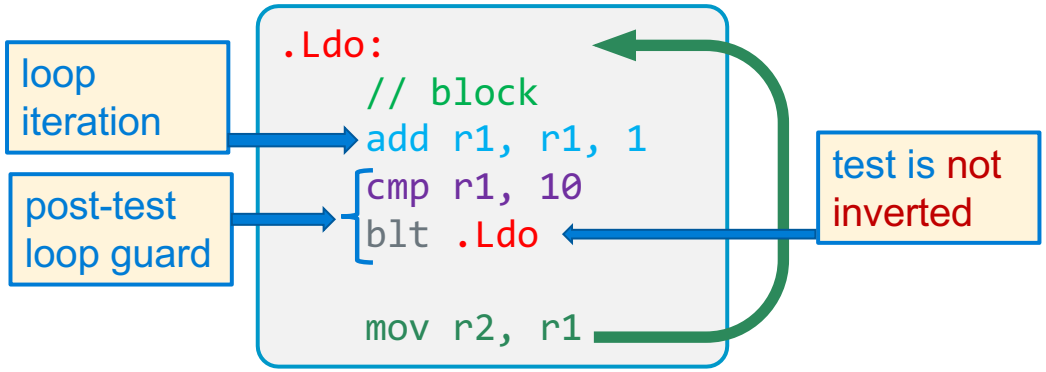
```
while (r1 < 10) {
    if (r2 != r1) {
        /* block */
    }
    r1++;
}
r2 = r1;
```



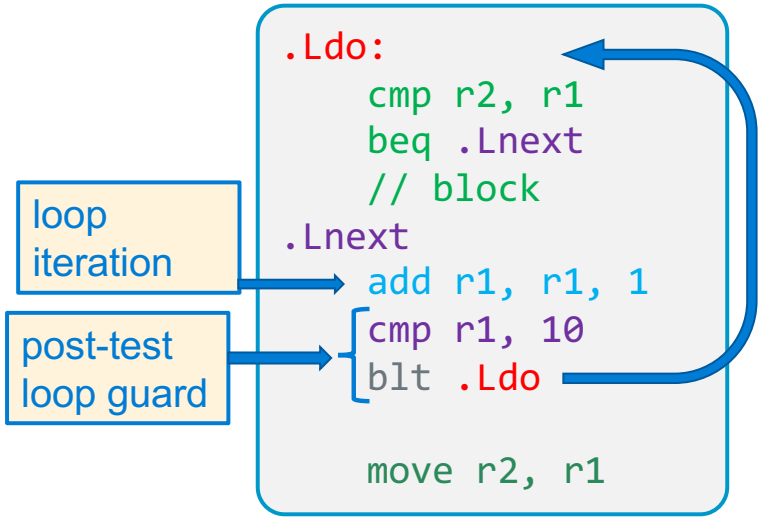
x

Post-Test Guards – Do While Loop

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

r2 = r1;
```

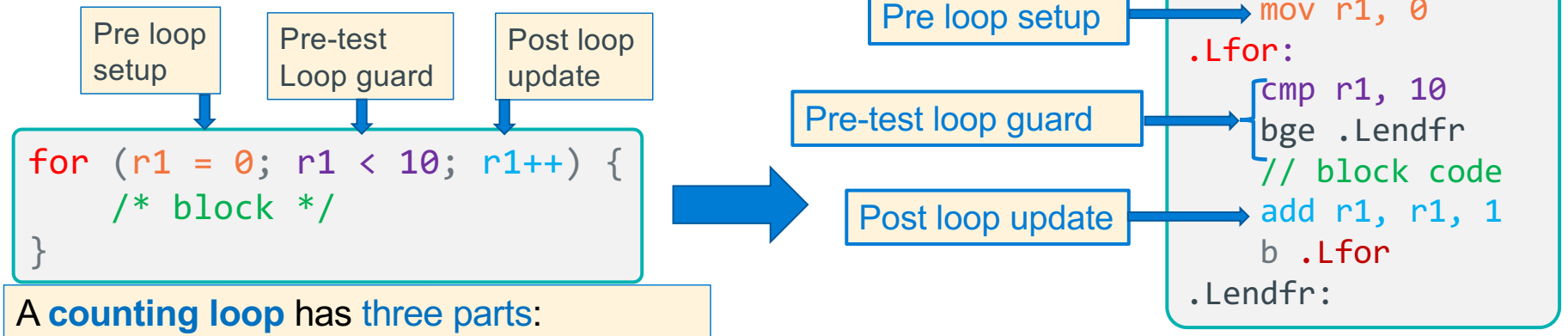



```
do {
    if (r2 != r1) {
        /* block */
    }
    r1++;
} while (r1 < 10);

r2 = r1;
```



x

Program Flow – Counting (For) Loop

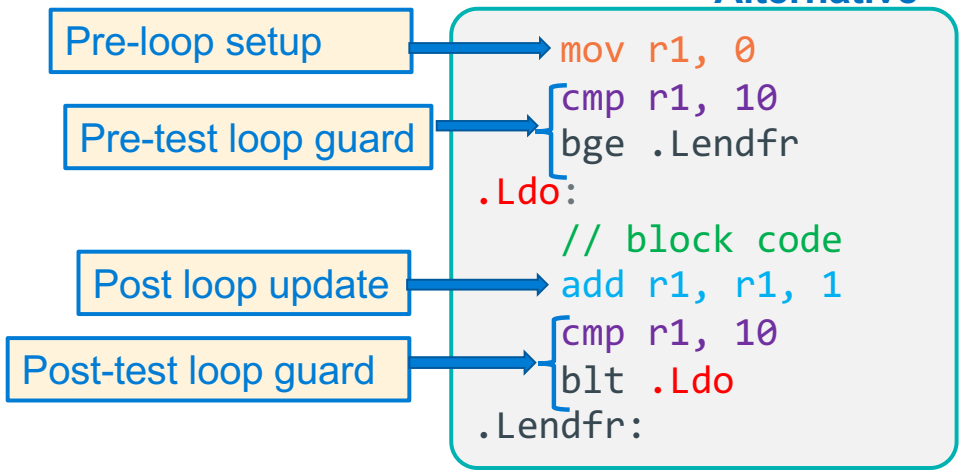


A **counting loop** has three parts:

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

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

Alternative



x

Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	Align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		char chx = 'A' char string[] = {'A','B','C', 0};	chx: .byte 'A' string: .byte 'A','B',0x42,0
16-bit int (2 bytes)	.hword .short	1	short length = 0x55aa;	length: .hword 0x55aa

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

strings '\0' term	.string		char class[] = "cse30";	class: .string "cse30"

```
int num;
int *ptr = &num;
char msg[] = "123";
char *lit = "456";
```

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


x

Defining Array Static Variables

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

```
 In C:      int int_buf[100];
            int array[] = {1, 2, 3, 4, 5};
            char buffer[100];
.bss
int_buf:    .space 400   // convert 100 to 400 bytes
char_buf:   .space 100
.data
array:      .word 1, 2, 3, 4, 5
one_buf:    .space 100, 1  // 100 bytes each byte filled with 1
```

`.space size, fill`

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

x

Static Variable Alignment: Using .align

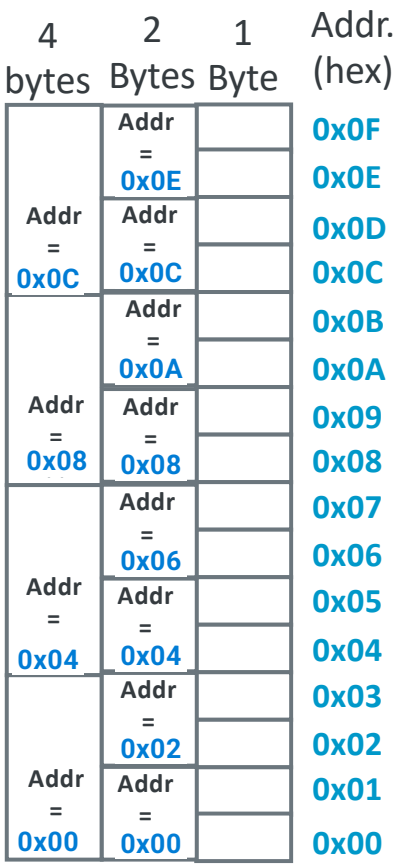
Accessing **address aligned** memory based on data type has the best performance

integer: 4 bytes

short: 2 bytes

char: 1

SIZE	Directive	Address ends in	Align Directive
8-bit char -1 byte	.byte	0b..0 or 0b..1	
16-bit int -2 bytes	.hword .short	0b..0	.align 1
32-bit int -4 bytes	.word .long	0b..00	.align 2



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

- Tells the assembler **the next line that allocates memory** must **start** at the next higher memory address **where** the lower **n** address bits are zero
- Assembler may adjust the starting address of the **next variable if needed**
- At the **first use of any Segment directive**, alignment **starts at an 8-byte aligned address** (for doubles)

x

Static Variable Alignment Example

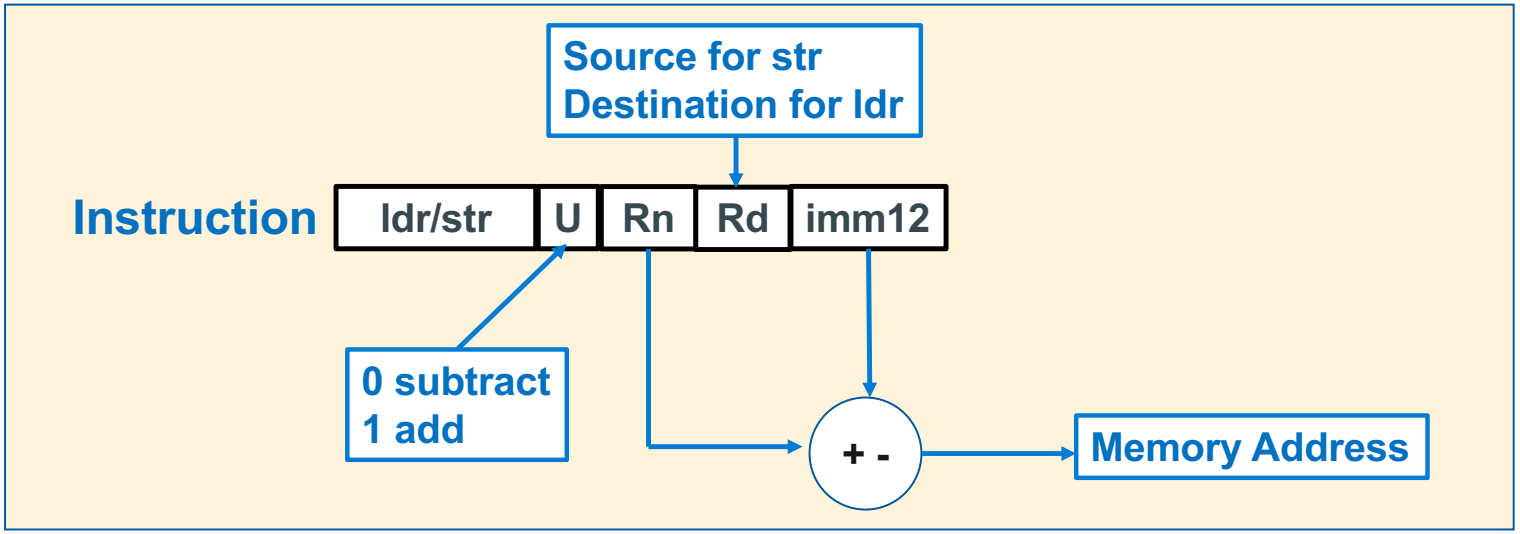
`.align n` use is **for the next address allocated** immediately following the `.align` directive

1. Examine both the size and alignment of the variable defined above; determine the address it ends at
2. Add a `.align` if the **next variable** would not be properly aligned based on its type (it is ok if the .align is **not** needed)

```
ch:     .byte 'A','B','C','D','E'
        .align 1    // next 2 bytes
ary:    .hword 0, 1
        .align 2    //next 4 bytes
x:      .word 55

str:    .string "HI"
```

x

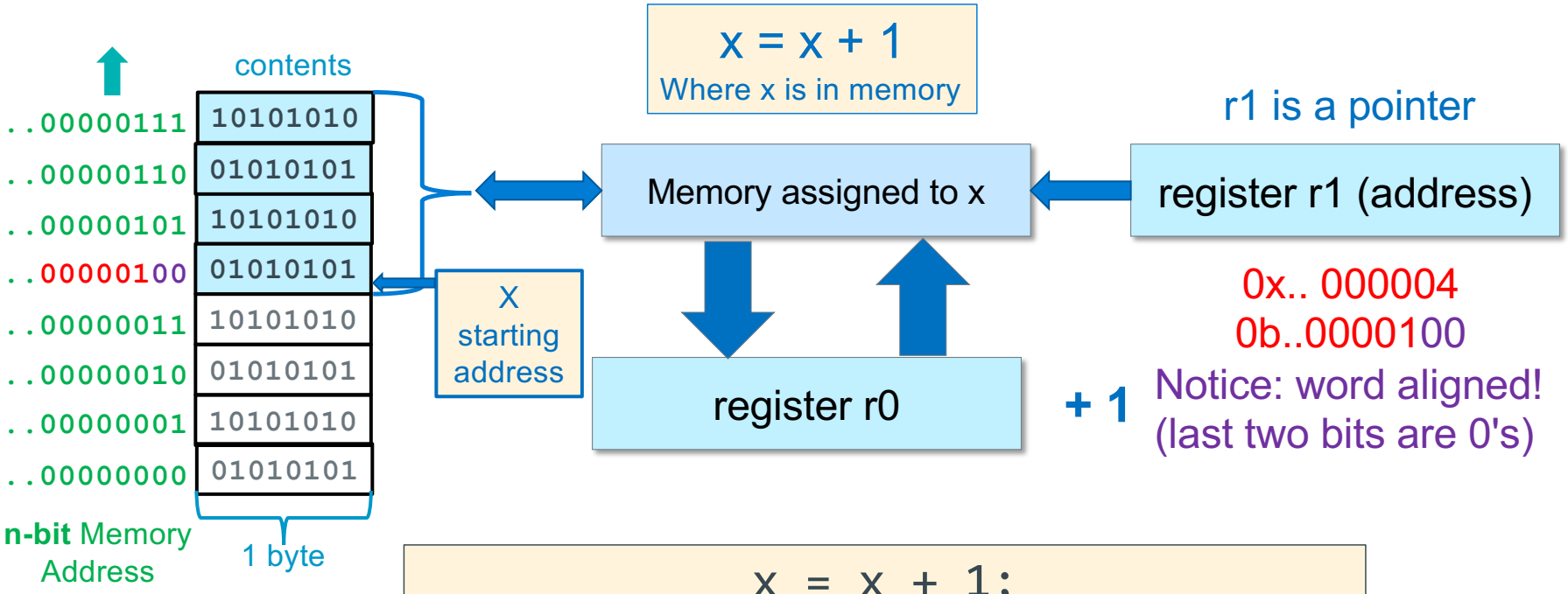
ldr/str Register Base and Register + Immediate Offset Addressing



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

x

Example Base Register Addressing Load – Modify – Store

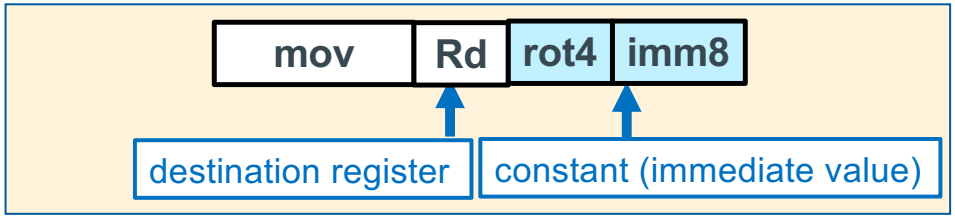



```
                x = x + 1;
ldr r0, [r1]        // r0 = *r1 (read x)
add r0, r0, 1       // r0 = r0 + 1 (x++)
str r0, [r1]        // *r1 = r0 write x
```

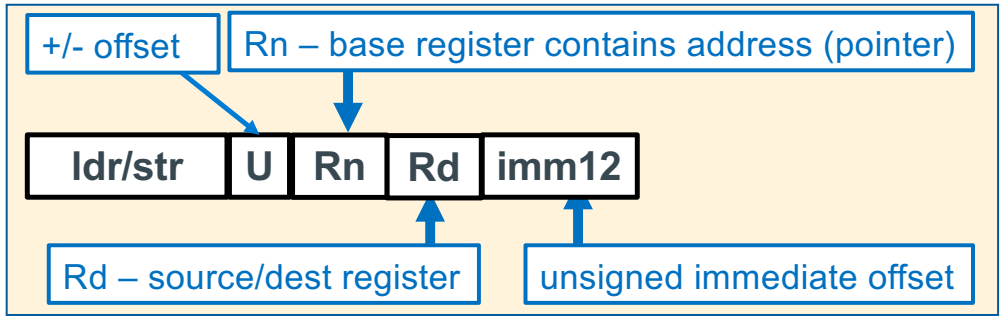

x

Review From Earlier week: How to Access Memory?

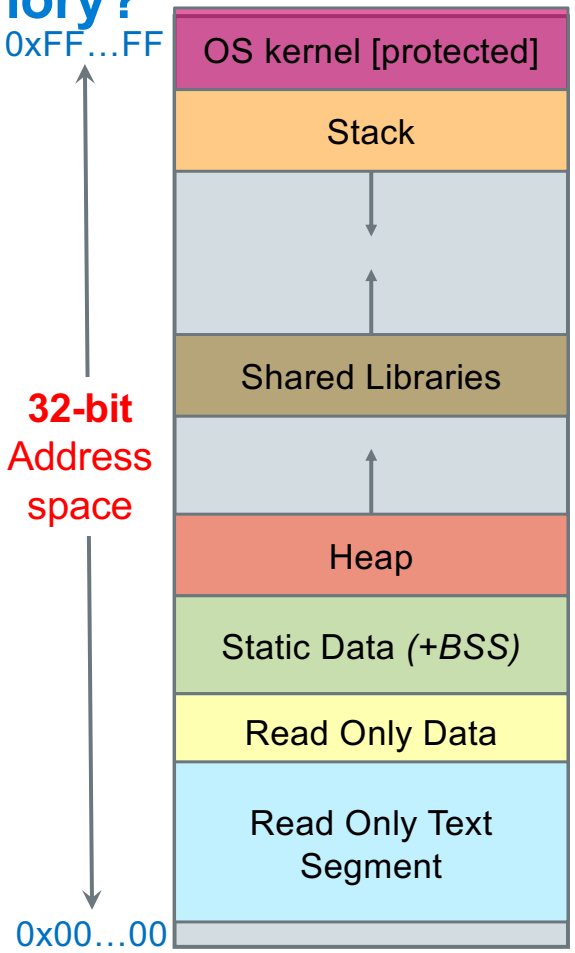
- Address space is 32 bits wide – POINTERS in registers



rot4/imm8 is too small



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



x

How to Access variables in a Data Segment

```
ldr/str  Rd,  [Rn, +- imm12]
```

- How do you get the **address into the base register Rn** for a Labeled location in memory?
- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
  - It is accessed using **the pc as the base register**
  - Each entry contains a **32-bit Label address**
- How to access this table to get a pointer:

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

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

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

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

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

assembly source file ex.S

```
        .bss
y:      .space 4

        .data
x:      .word 200

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

        .text
        // function header
main:

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

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

x

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

- In data processing instructions, the field **imm8 + rotate 4 bits** is too small to store many numbers outside of the range of -256 to 255, how do you get larger immediate values into a register?

fails ➡ `mov r0, 1023`

xxx.s:24: Error: invalid constant (3ff) after fixup

replacement ➡ `ldr r0, =1023`

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

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


x

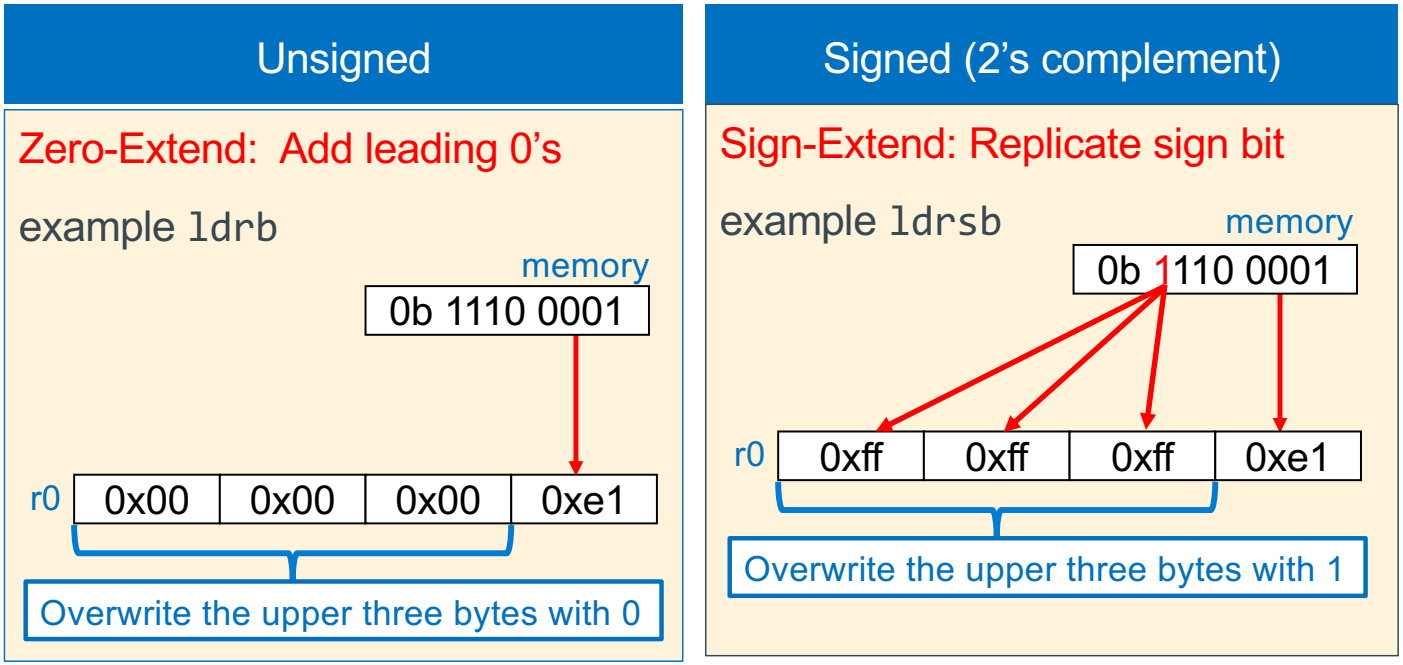
Loading and Storing: Variations List

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

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

x

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

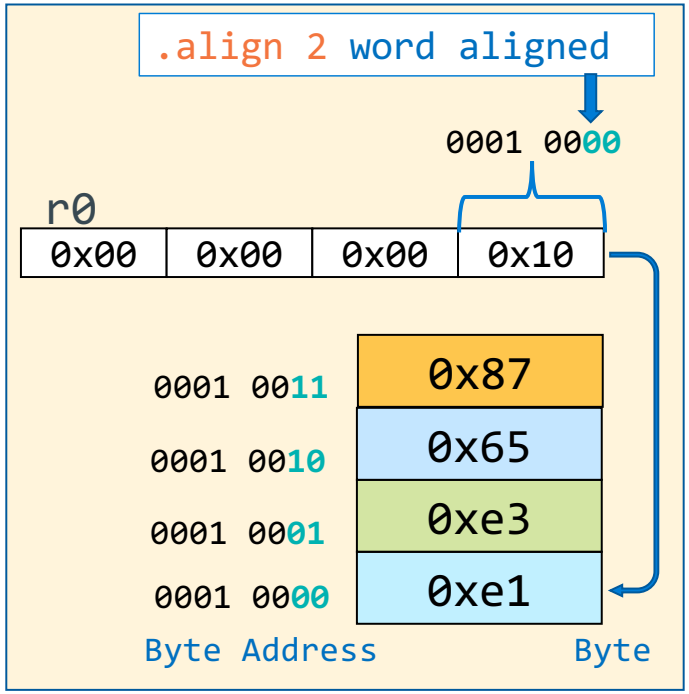


Instructions that zero-extend:
ldrb, ldrh

Instructions that sign-extend:
ldrsb, ldrsh

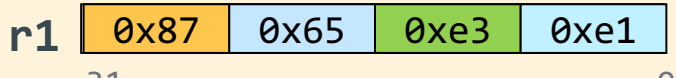
x

Load a Byte, Half-word, Word



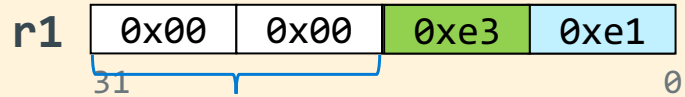
Load a word

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



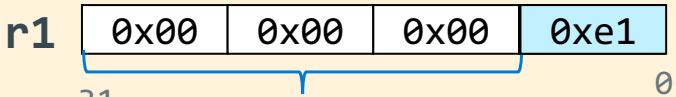
Load a halfword

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



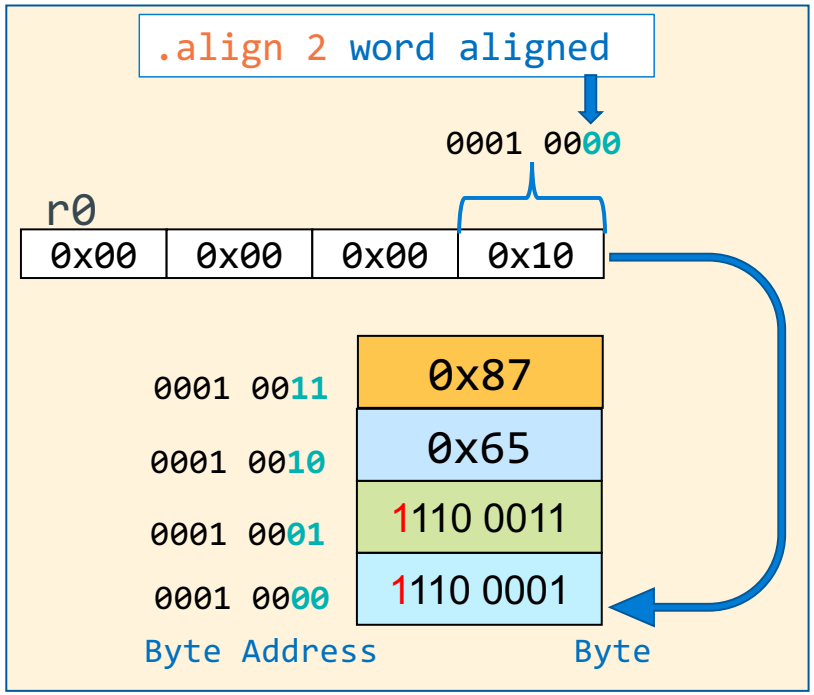
Load a byte

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



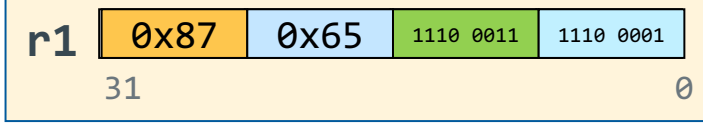
x

Signed Load a Byte, Half-word, Word



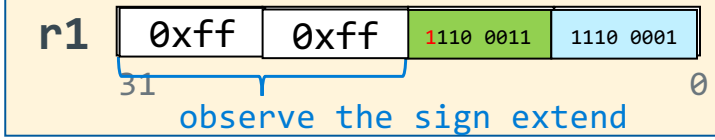
Load a word (no change)

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



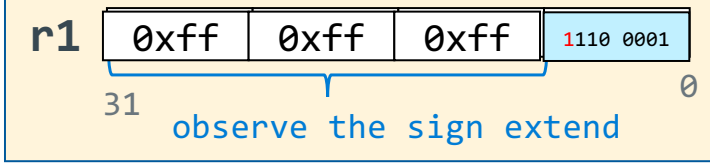
Load a halfword

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



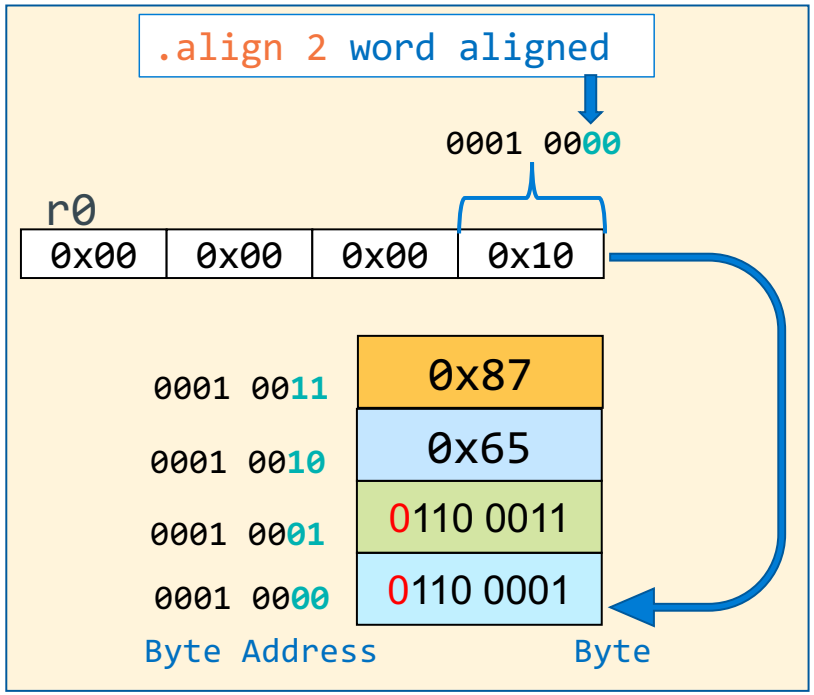
Load a byte

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



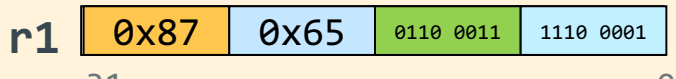
x

Signed Load a Byte, Half-word, Word



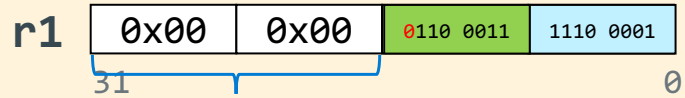
Load a word (no change)

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



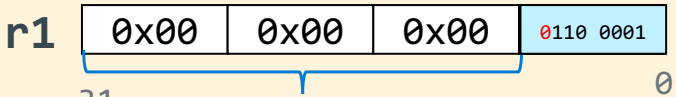
Load a halfword

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



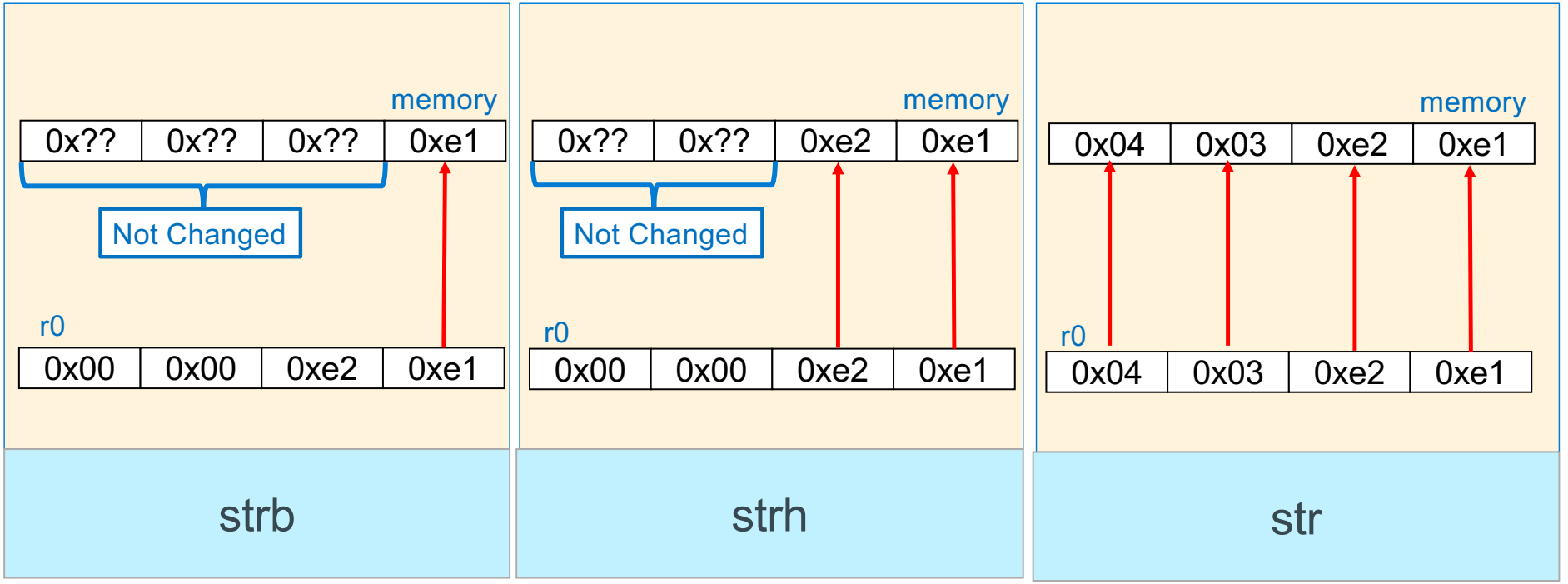
Load a byte

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



x

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



x

Store a Byte, Half-word, Word

initial value in r0

0x20	0x00	0x00	0x00

Store a byte

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

r1 | 0x87 | 0x65 | 0xe3 | 0xe1 |

31 ... 0

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

observe other bytes NOT altered

Store a halfword

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

r1 | 0x87 | 0x65 | 0xe3 | 0xe1 |

31 ... 0

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

Store a word

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

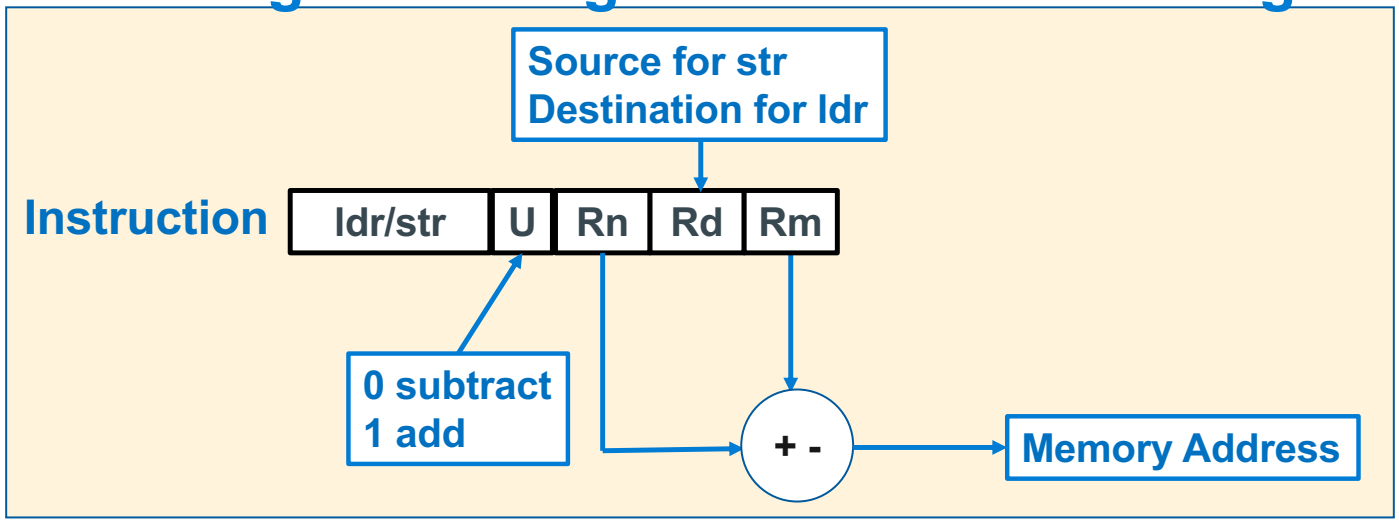
r1 | 0x87 | 0x65 | 0xe3 | 0xe1 |

31 ... 0

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

x

ldr/str Base Register + Register Offset Addressing



Pointer Address = Base Register + Register Offset

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

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

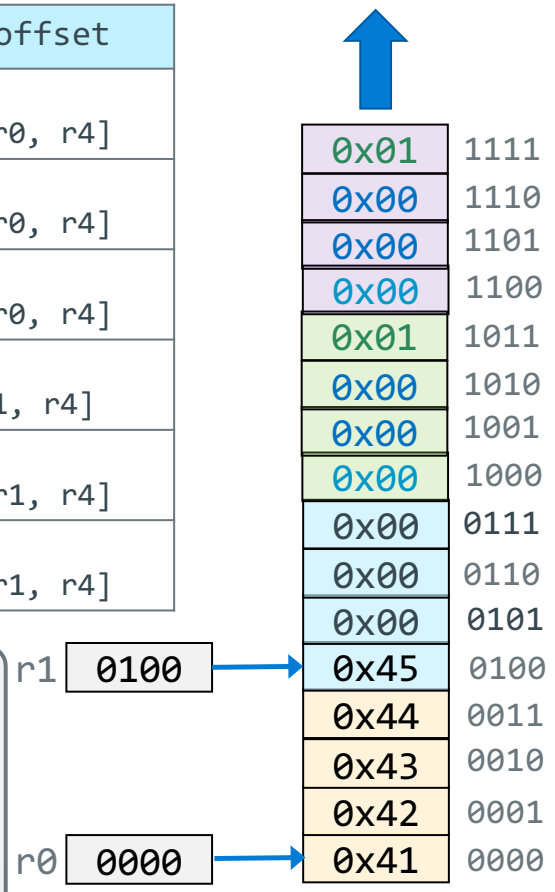
x

Array addressing with ldr/str

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

table rows are independent instructions

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

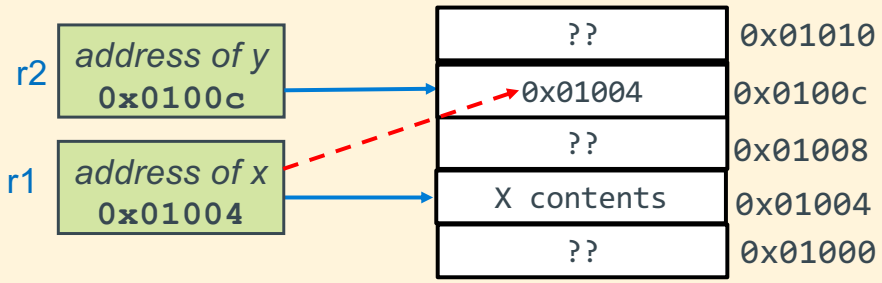


ldr/str practice - 1

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

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

write Y = &X;




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

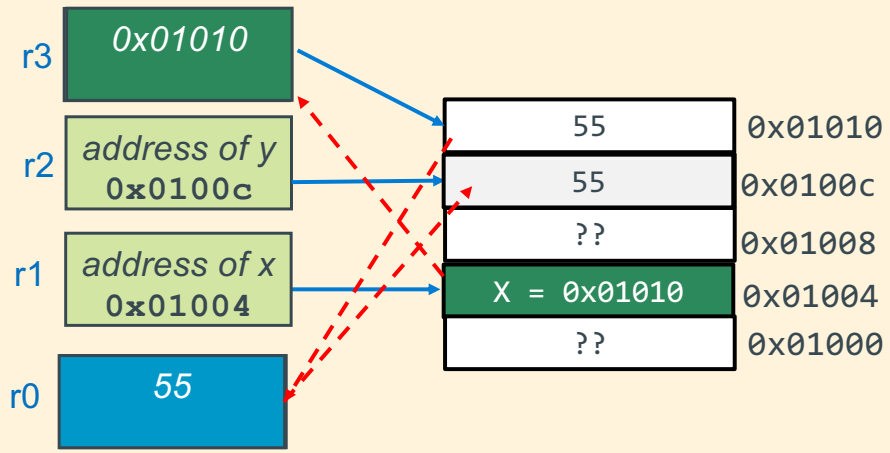

x

ldr/str practice - 2

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

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

write Y = *X;

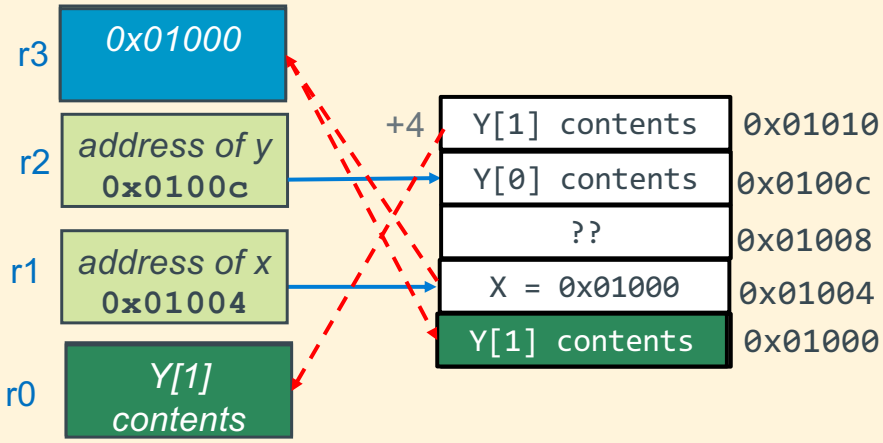



```
ldr   r3, [r1]  // r3 ← x (read 1)
ldr   r0, [r3]  // r0 ← *x (read 2)
str   r0, [r2]  // y ← *x
```


x

ldr/str practice - 3

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




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


x

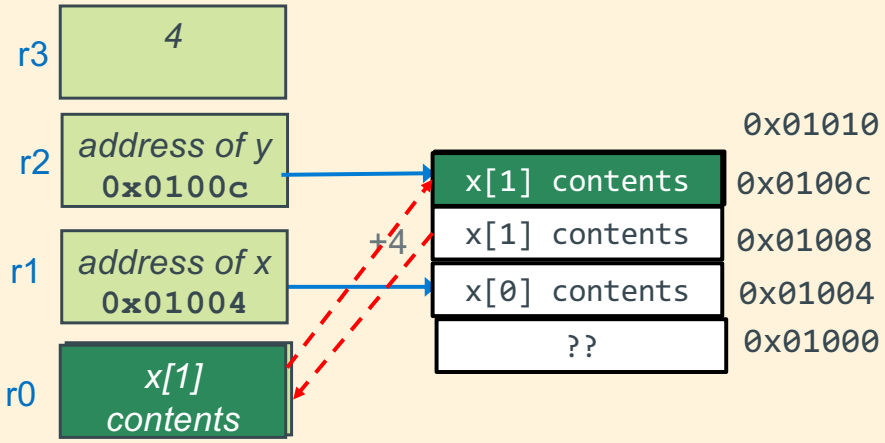
ldr/str practice - 4

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

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

r3 contains a 4

write Y = X[1];

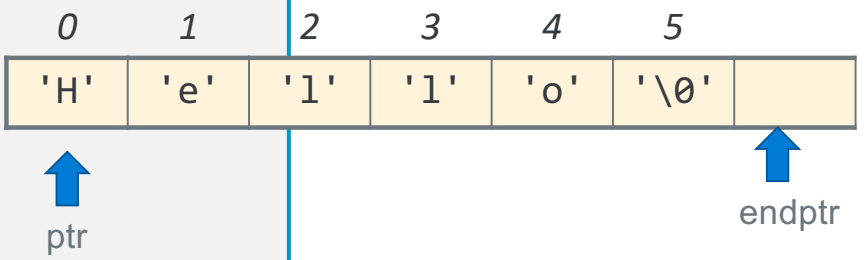



```
ldr   r0, [r1, r3]  // r0 ← x[1]
str   r0, [r2]    // y ← x[1]
```


x

Example: Base Register Addressing with Arrays

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

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

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

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



x

Example: Base Register Addressing with Arrays

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


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

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


x

Example: Base Register + Offset Register

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

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

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

Using Base register pointer with an end pointer

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

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

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

Using Base register pointer + Offset register

x

Example: Base Register + Register Offset Two Buffers

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

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

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


    return EXIT_SUCCESS;
}
```

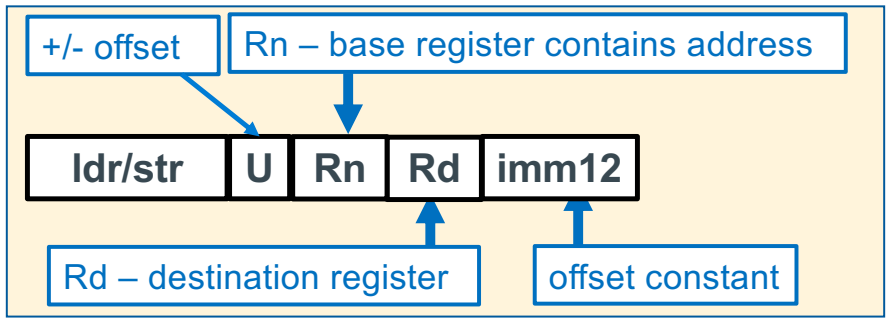
- Make sure to index by bytes and increment the index register by sizeof(int) = 4

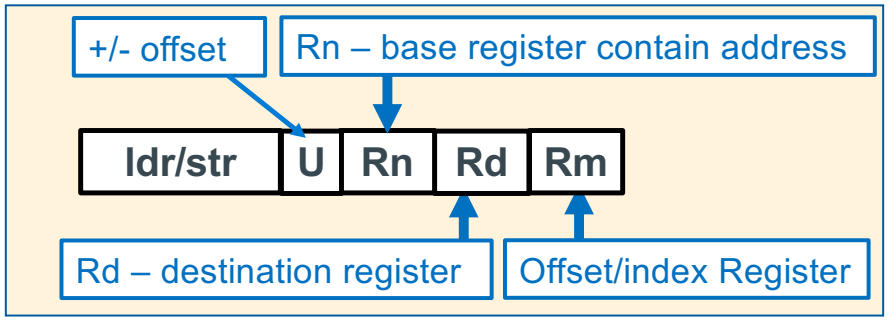
```
    .data       // segment
src:.word       1, 3, 5, 7, 9, 11
    .equ        SRCSZ, (. - src)  // bytes msg
dest:.space     SRCSZ
    .equ        INT_STEP, 4
…

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

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

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

one increment covers both arrays

x

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






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

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


x

Reference: Addressing Mode Summary for use in CSE30

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

x

Function Calls, Parameters and Locals: Requirements

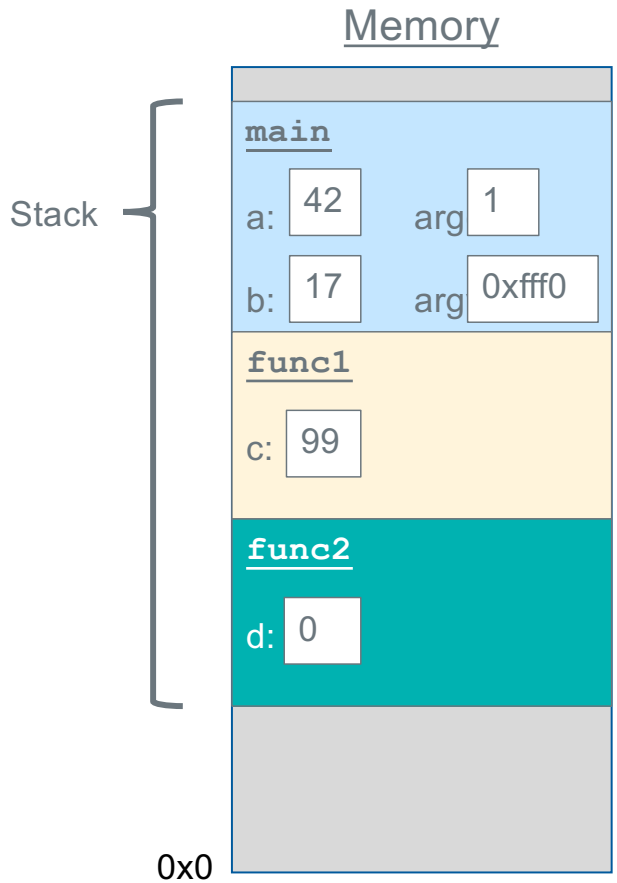
```
int
main(int argc, char *argv[])
{
    int x, z = 4;

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

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

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

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

x

The Stack

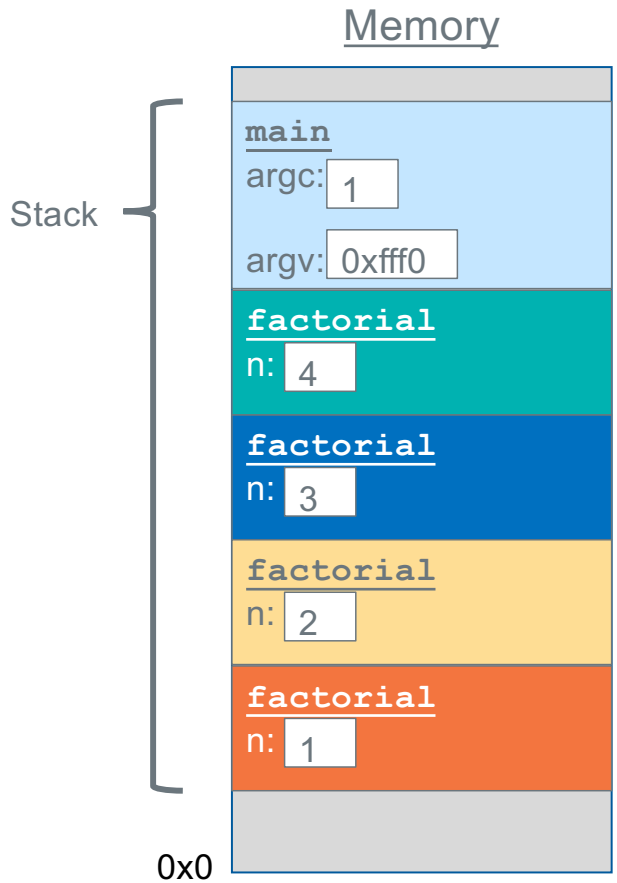
```
void func2() {
    int d = 0;
}

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

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



The Stack

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

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

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



Support For Function Calls and Function Call Return - 1

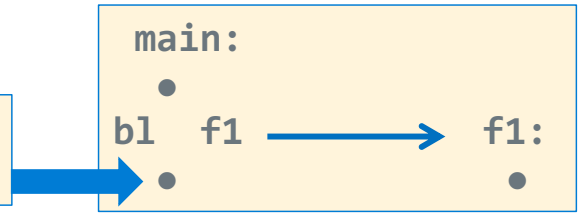
bl	imm24

Branch with Link (function call) instruction

```
bl label
```

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

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



x

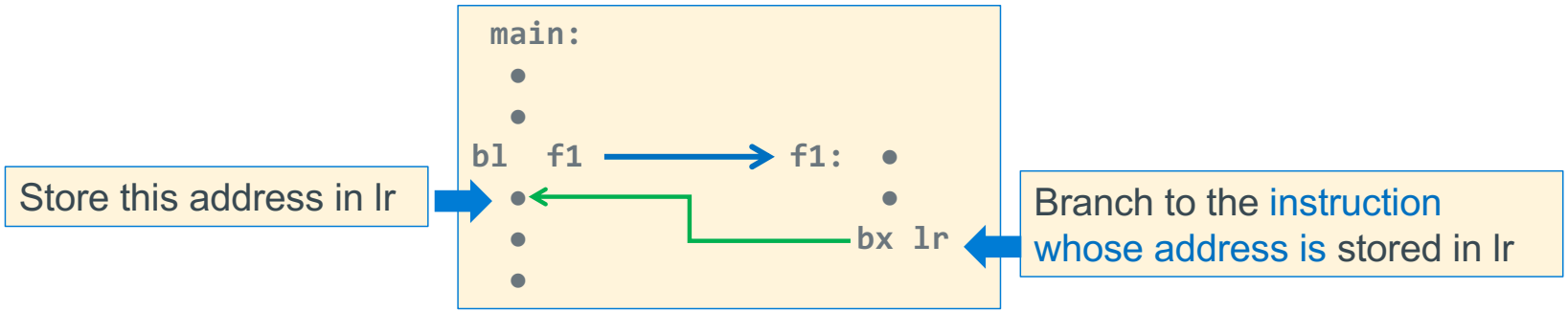
Support For Function Calls and Function Call Return - 2

bx	Rn

Branch & exchange (function return) instruction

```
bx lr
```

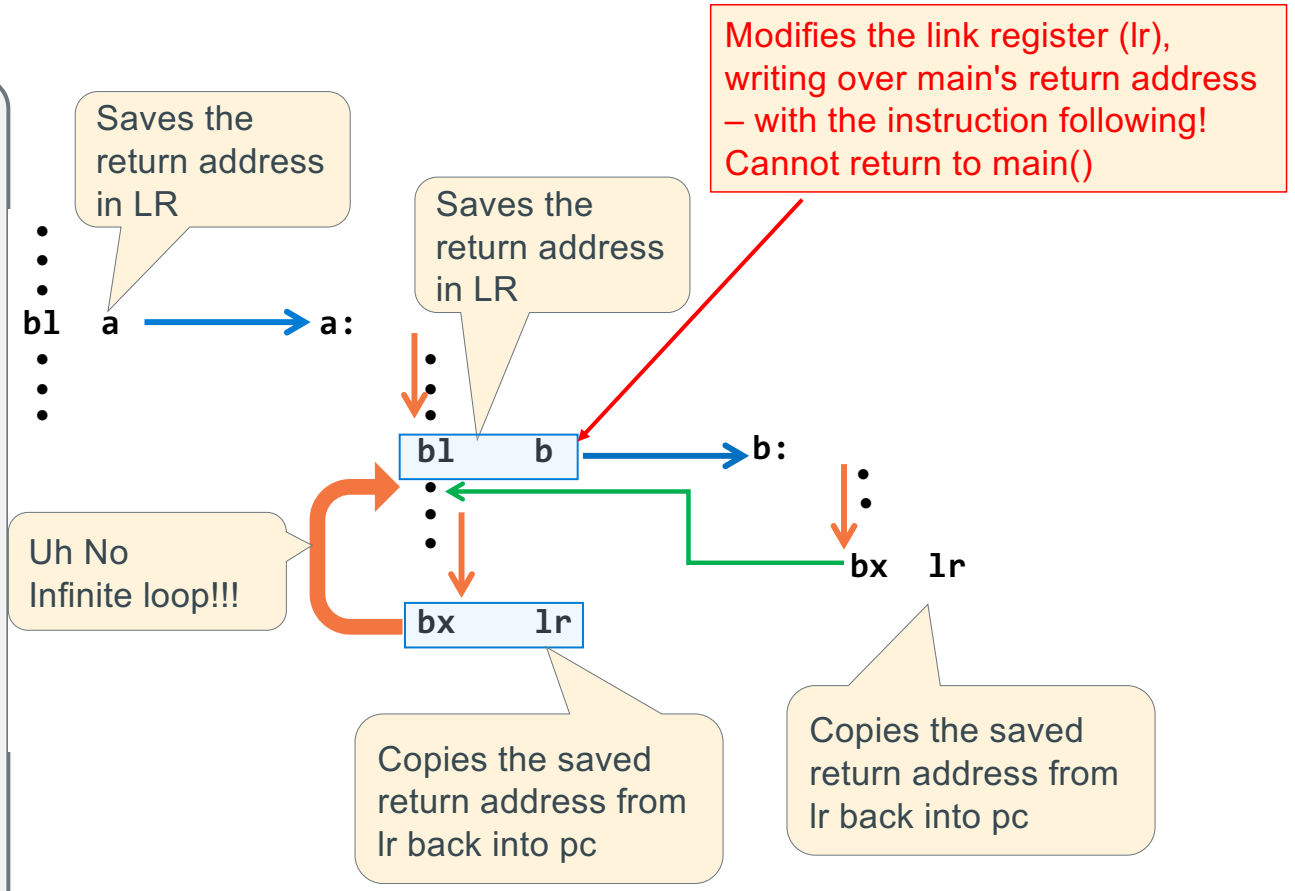
- Causes a branch to the instruction **whose address is stored** in register <`lr`>
  - It copies `lr` to the PC
- This is often used to implement a return from a function call (exactly like a C return) when the function is called using `bl label`



x

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

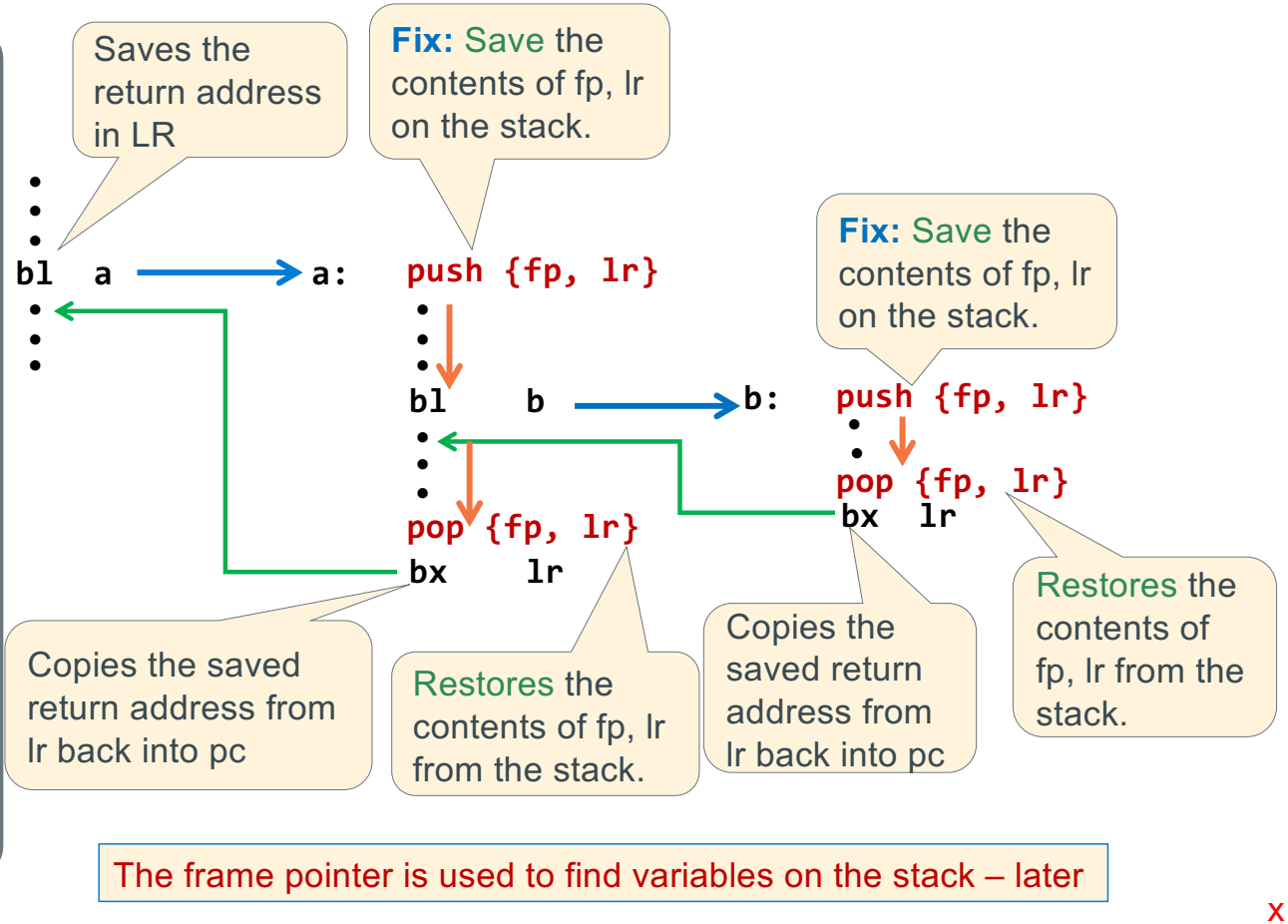
```
int main(void)
{
     a();
     /* other code */
     return EXIT_SUCCESS;
}

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

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



x

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

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

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

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



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

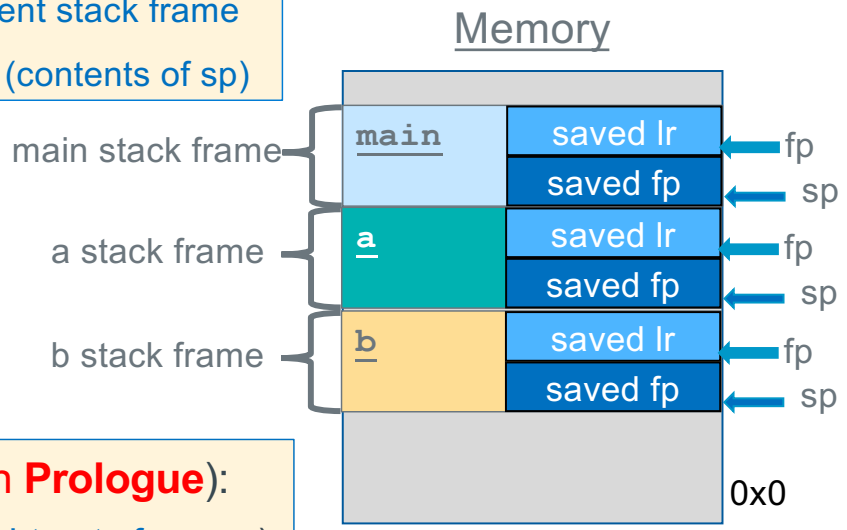
x

Minimal Stack Frame (Arm Arch32 Procedure Call Standards)

Requirements

- `sp` points at top element in the stack (lowest byte address)
- `fp` points at the `lr` copy stored in the current stack frame
- **Stack frames align to 8-byte addresses** (contents of sp)

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



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

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

x

Review Return Value and Passing Parameters to Functions

(Four parameters or less)

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

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

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

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

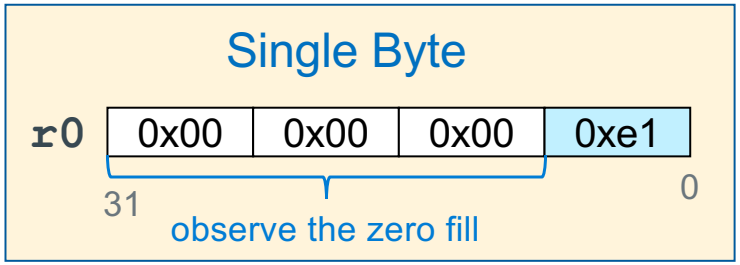
- Each **parameter and return value is limited to data that can fit in 4 bytes or less**
- You receive up to the first four parameters in these four registers
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes,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

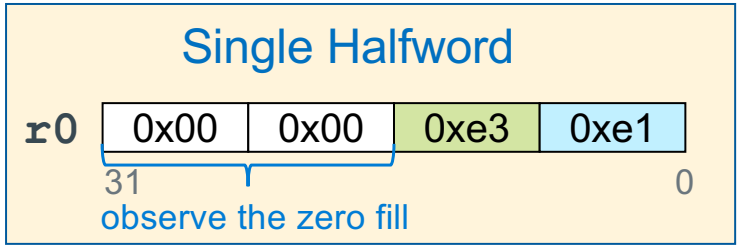
x

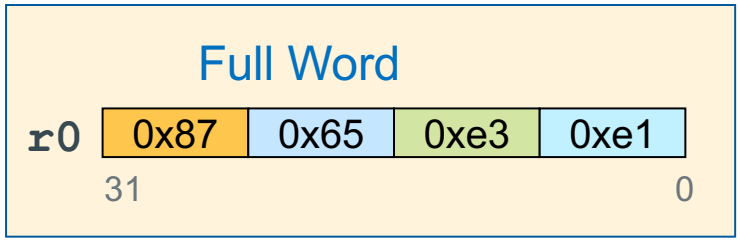
Argument and Return Value Requirements

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

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







Preserved Registers: Protocols for Use

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

- **Function Call Spec**:

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

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

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

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

x

Preserved Registers: When to Use?

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

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

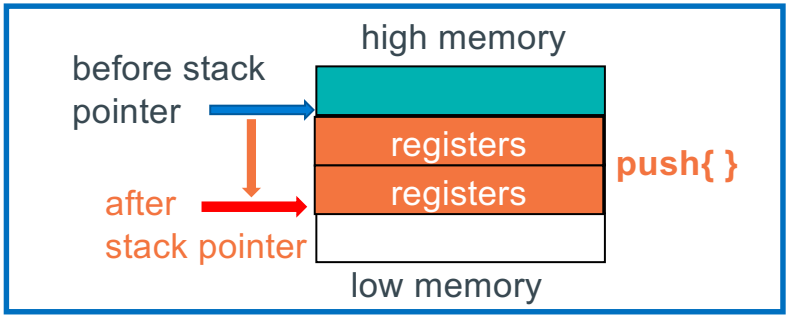
1. Values that you want to protect from being changed by a function call
   a) Local variables stored in registers
   b) Parameters passed to you (in `r0-r3`) that you need to continue to use after calling another function
2. Need more than `r0-r3` whether you call another function or not

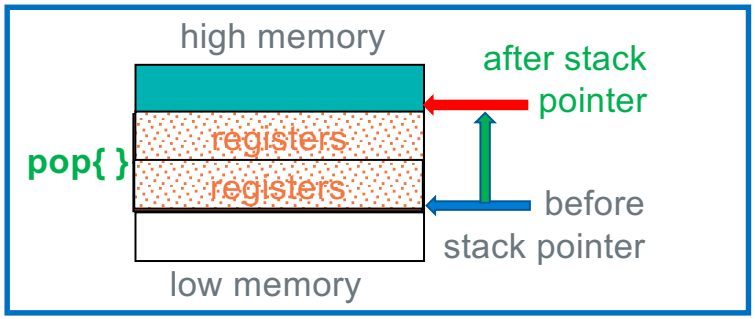
   Options are:
   a) preserved register *or*
   b) stack local variable (later slides)

x

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

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





x

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

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

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

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

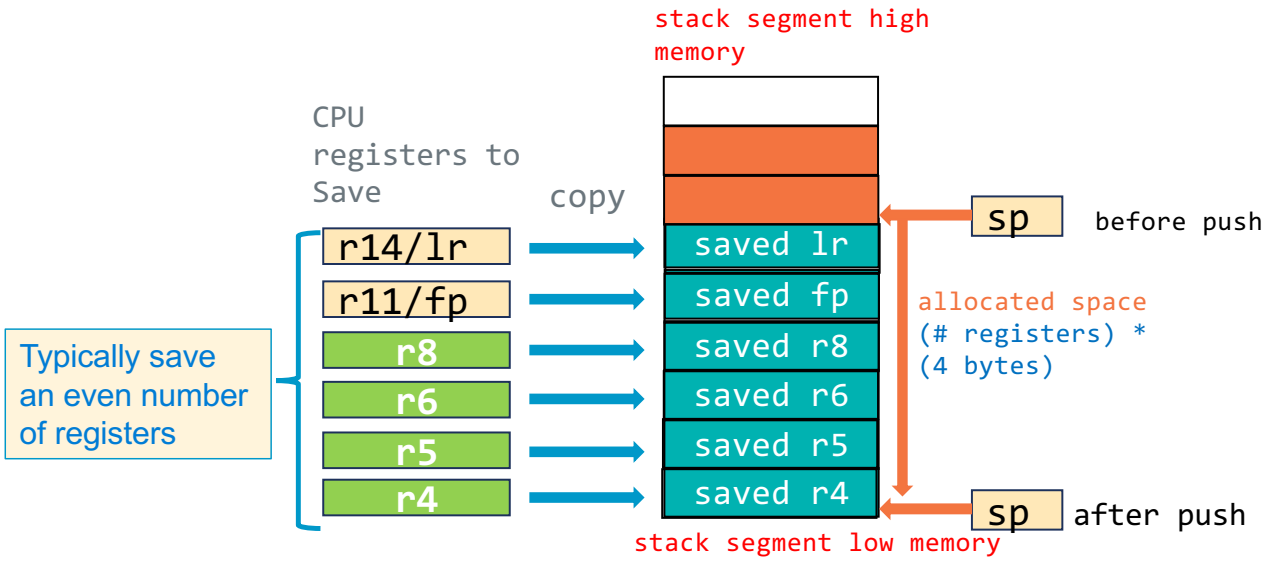
- Registers cannot be: (1) duplicated in the list, nor be (2) listed out of numeric order
- Register ranges can be specified `{r4, r5, r8-r11, fp, lr}`

x

push: Multiple Register Save

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

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



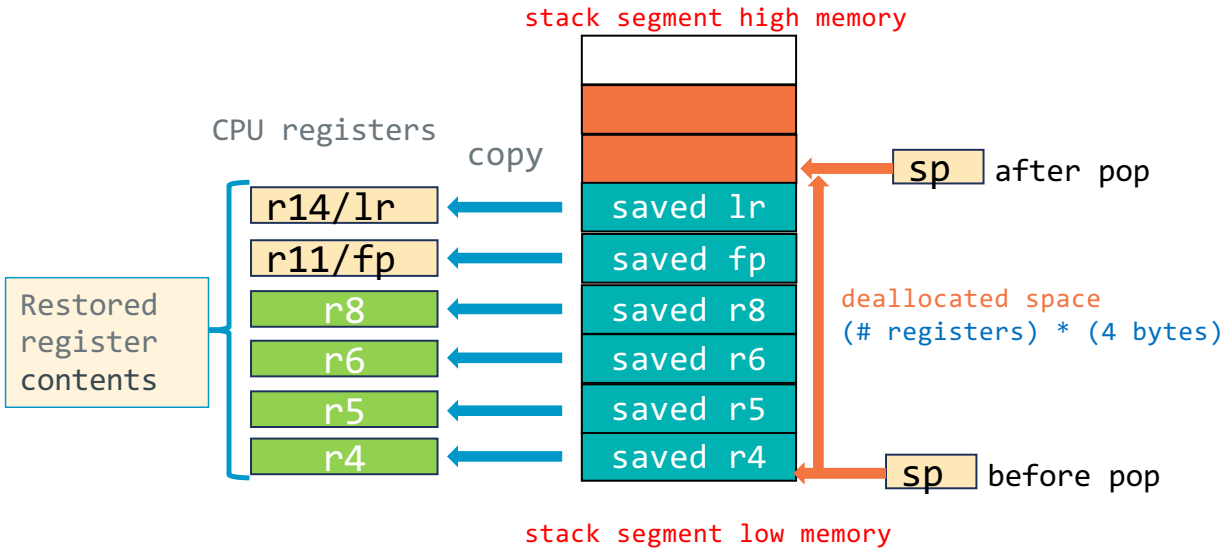
- **`push`** copies the contents of the **`{reg list}`** to stack segment memory
- **`push`** **Also** subtracts (# of registers saved) * (4 bytes) from the **`sp`** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)

x

pop: Multiple Register Restore

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

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

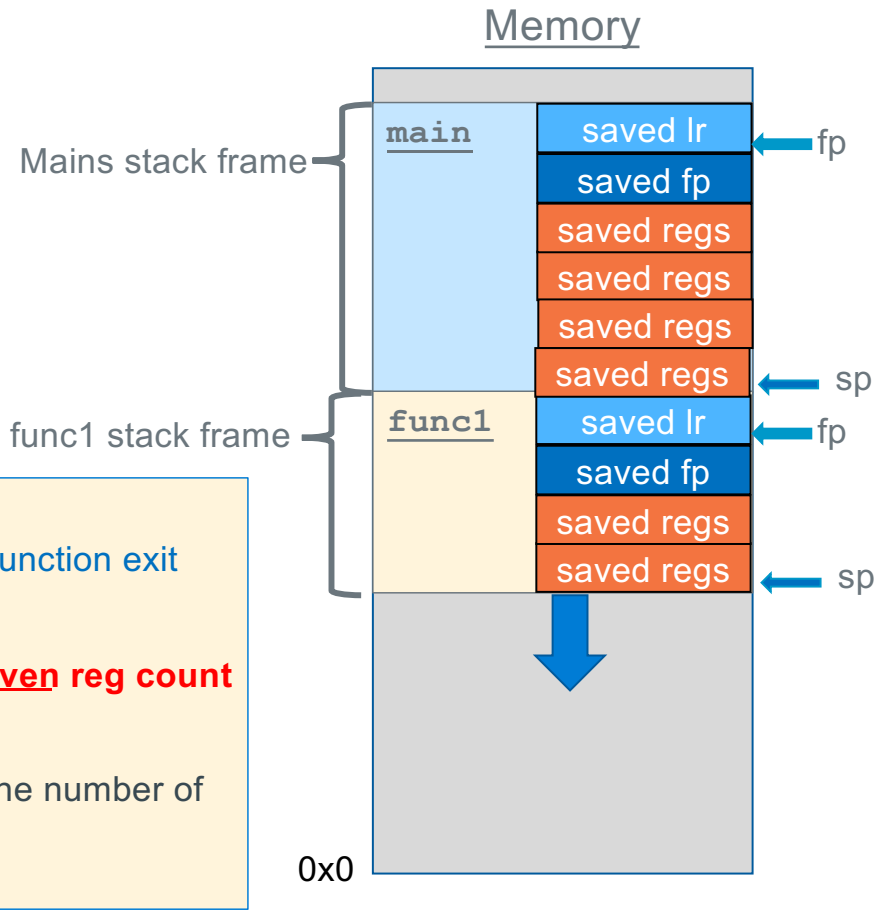


- `pop` copies the contents of stack segment memory to the `{reg list}`
- `pop` **adds:** (# of registers restored) * (4 bytes) to `sp` to ***deallocate*** space on the stack
  - sp = sp + (# registers restored * 4)
- **Remember:** `{reg list}` must be the same in both the `push` and the corresponding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

x

Basic Stack Frames (Arm Arch32 Procedure Call Standards)

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

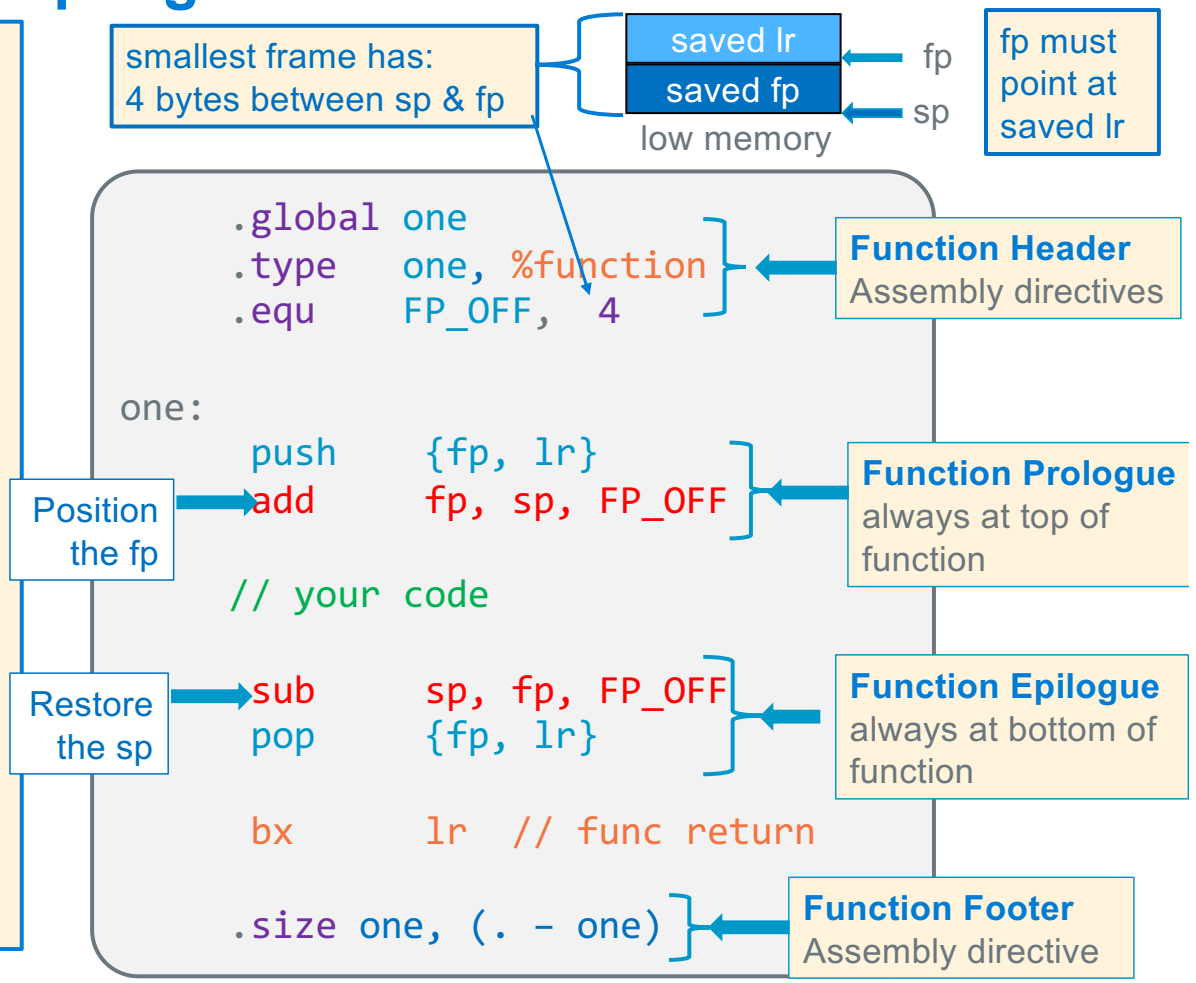


- **On each function call start (entry)**
  - Preserved registers: push at function entry and pop at function exit
- **Rules**
  - Keep `sp` 8-byte aligned strategy: `{reg list}` has an **even reg count**
  - **Remember** `fp` must always points at the saved `lr`
- Issue: number of registers saved on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

x

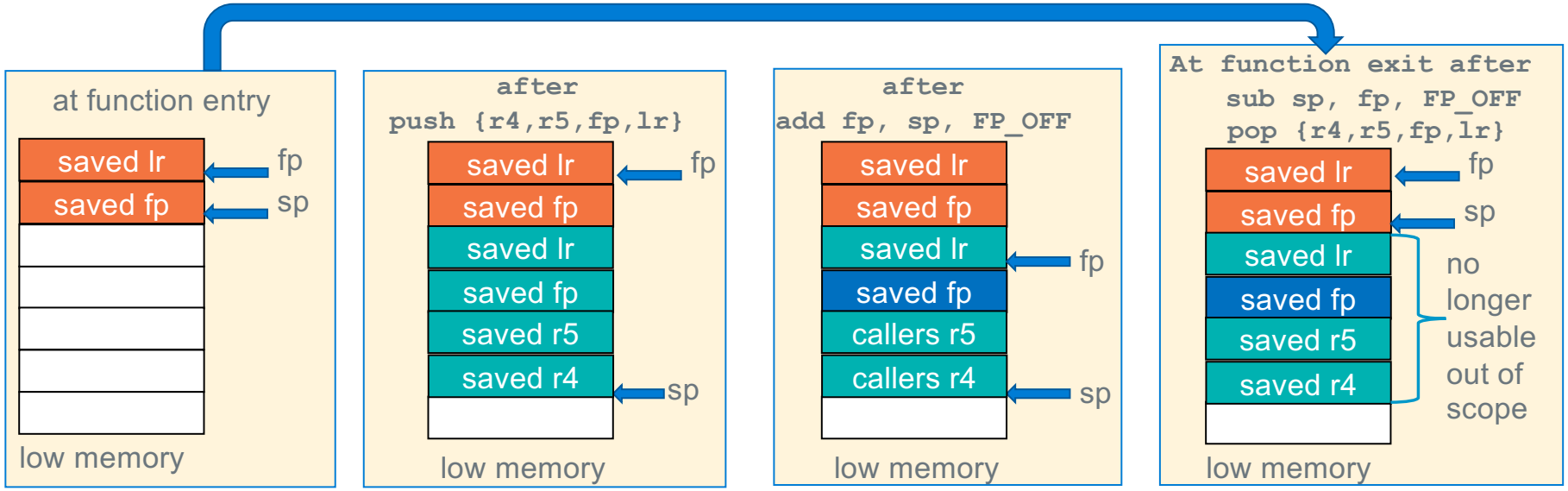
Function Prologue and Epilogue: Minimum Stack Frame

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



x

Saving/Restoring Preserved Registers At Function entry/exit



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

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

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

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

Part of function prologue

Part of function epilogue

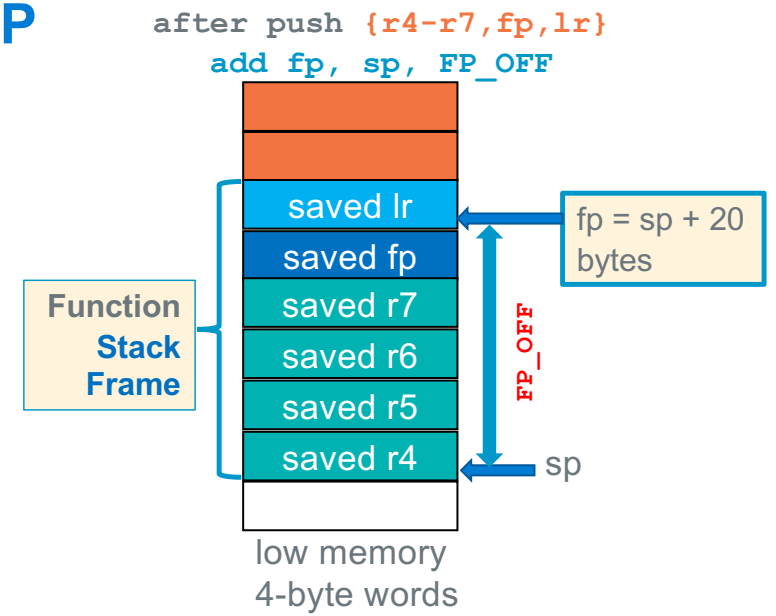
x

Setting FP_OFF: Distance from FP to SP

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

always at top of function saves regs and sets fp

always at bottom of function restores regs including the sp



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

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

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

Means Caution, odd number of regs!

x

Stack Creation Overview

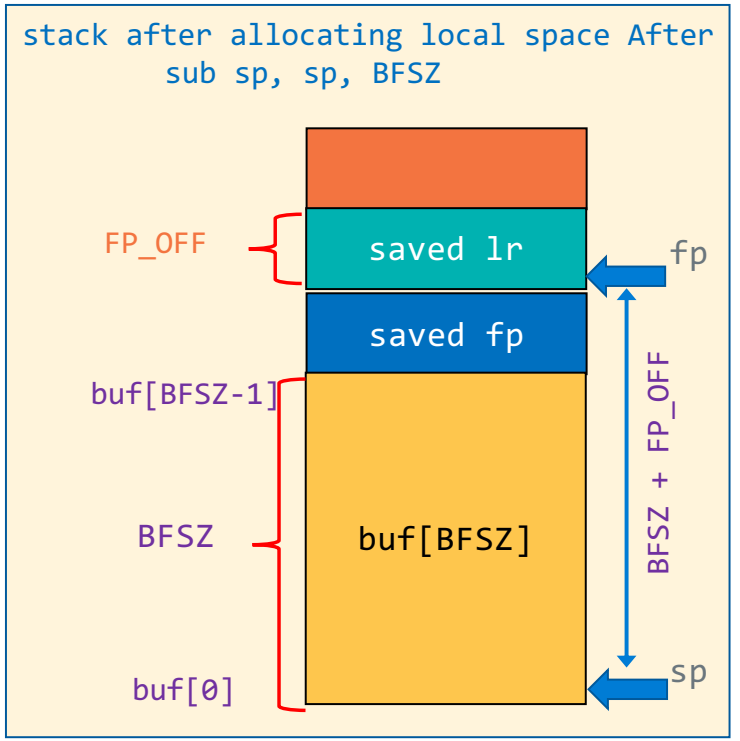
1. Calculate how much additional space is needed by local variables
2. **After the push, Subtract from the sp** the required byte count (+ padding - later slides)
3. If the variable has an initial value specified: add code to set the initial value
   a) mov and str are useful for initializing simple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

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

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

Function Prologue Extended

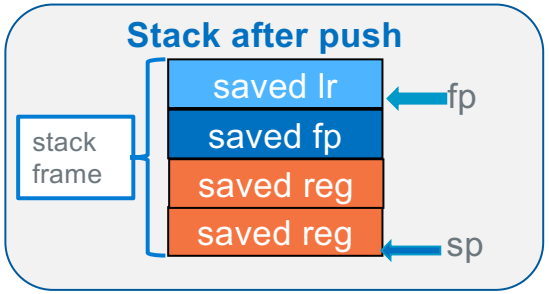
allocate space for buf[256]

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



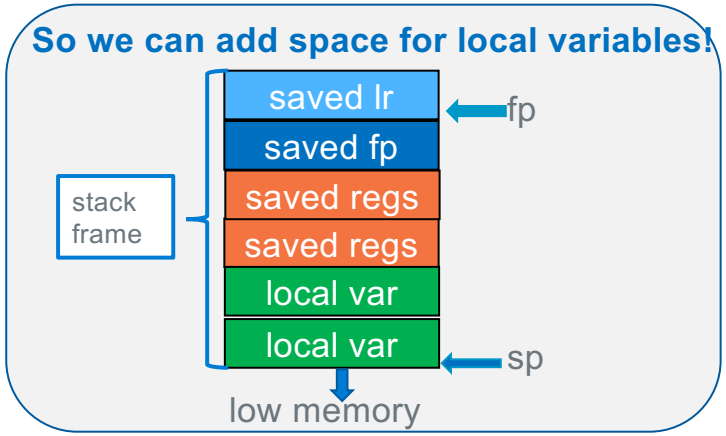
x

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




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

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




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

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

        bx      lr  // func return
```

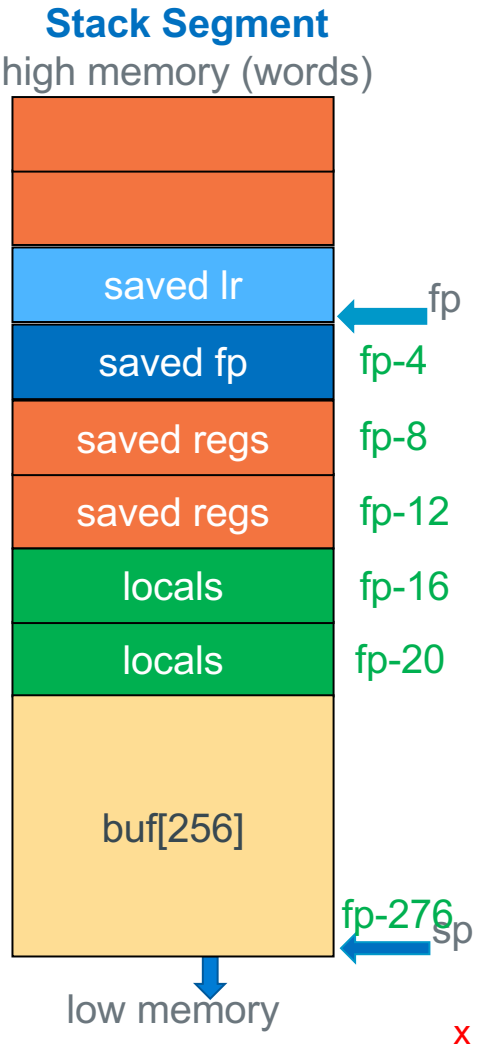
- force the `sp` (using the `fp`) to contain the same address it had after the push operation
  `sub sp, fp, FP_OFF`

x

Accessing the Stack Variables Overview

- Access data stored in the stack use `ldr/str` instructions
- **Use base register `fp` with offset addressing** (either register offset or immediate offset)
- No matter where the stack frame starts on the stack, `fp` always points at the same place in every stack (points at saved `lr`)
- *"hand calculated offset constants and sizes"* like -16 and -20 to access items is easy to get wrong, there is an easier way!

```
.equ   BFSZ, 256            // char array
ldr    r0, [fp, -16]
str    r3, [fp, -20]
sub    r0, fp, 256+20      // r0 = &(buf[0]);
ldr    r1, [r0]            // r1 = buf[0];
str    r3, [r0, 2]         // buf[2] = r3;
```



Variable Alignment on Stack

integer/pointer: 4 bytes

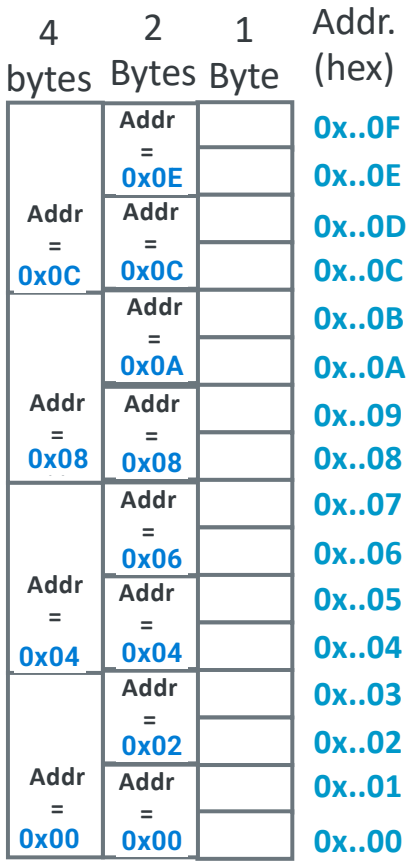
short: 2 bytes

char: 1

Variable Type/Size	Address ends in
8-bit char -1 byte	0b..0 or 0b..1
16-bit int -2 bytes	0b..0
32-bit int -4 bytes	0b..00
32-bit pointer -4 bytes	0b..00

- Starting address alignment requirements for local variables stored on the stack is just like static variables
- sp must be aligned to 8-bytes at function entry & exit
  - contents of sp always ends in 0b..000 at function entry
- Approach we will take (also what compilers often do): allocate all the local variable space as part of the function prologue
  - Aside: You cannot use .align as assembly directives are for fixed address

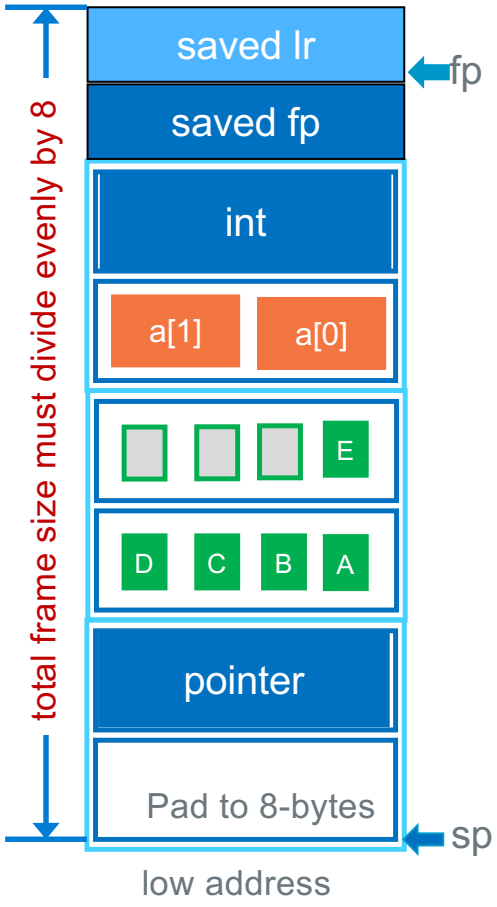
Starting address by size



x

Overview: Stack Frame Alignment Rules

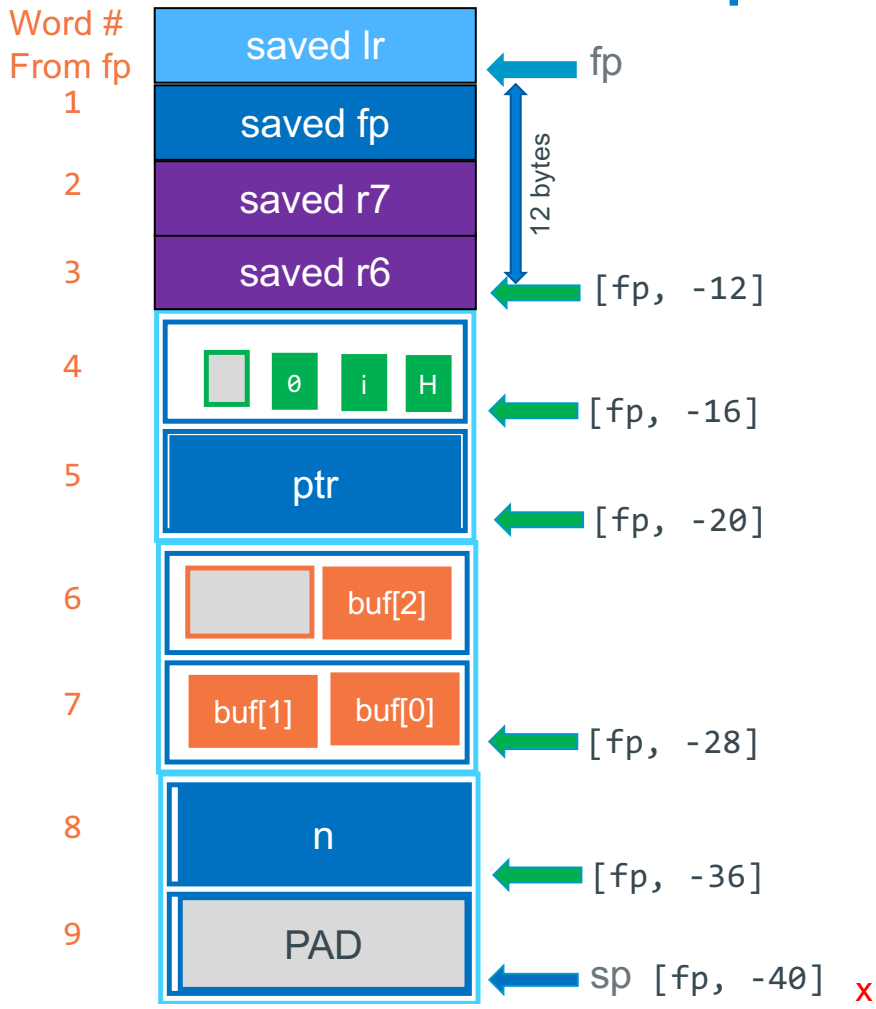
- Goal: minimize stack frame size
- Arrays start at a 4-byte boundary (even arrays with only 1 element)
  - Exception: double arrays [ ] start at an 8-byte boundary
  - struct arrays are aligned to the requirements of largest member
- Space padding when necessary is added at the high address end of a variables allocated space, so the next variable is aligned
- Single chars (and shorts) can be grouped together in same 4-byte word (following the alignment for the short)
- After all the variables have been allocated, add padding at stack frame bottom (low memory) so the total stack frame size (including all saved registers) is a multiple of 8 when the prologue is finished



x

Stack Frame Design – Step 3 Generate the Offsets from fp

- Word offset is a way to visualize the distance from fp for calculating offset values
- Better to have the assembler to generate readable offsets for use with `str` and `ldr`
  1. Easy to add and remove variable allocations from the design
  2. Creates well documented names for each variable: `ldr r0, [fp -20]` is hard to read
  3. Automatically calculates the total size of the stack frame used by local variables



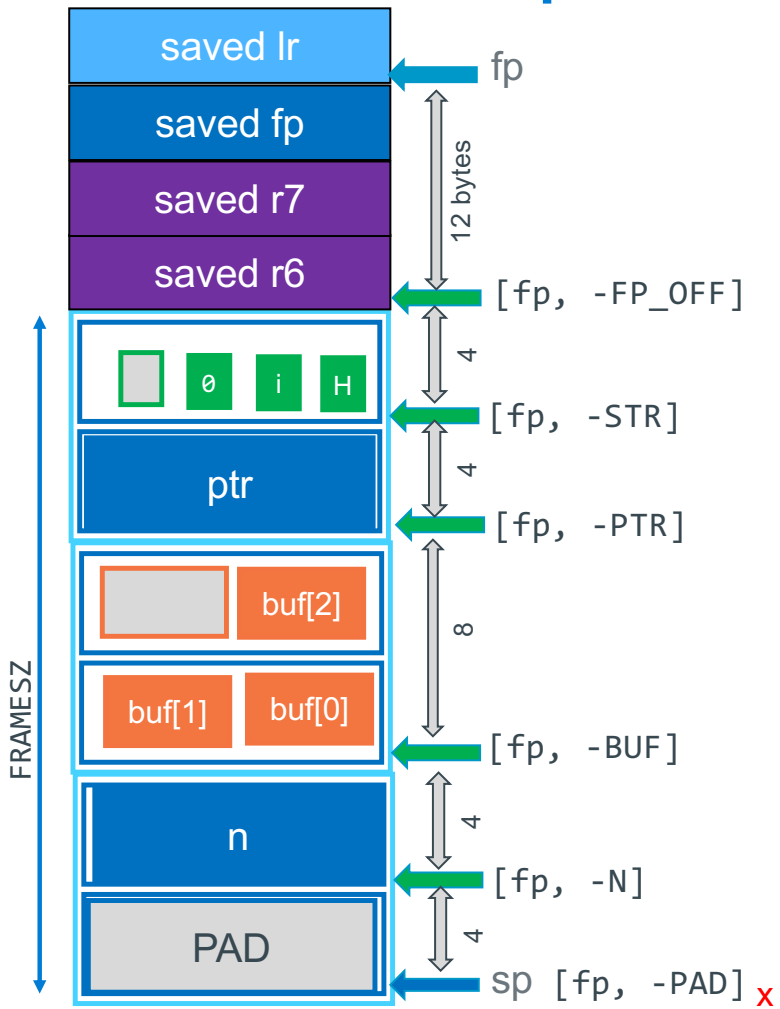
Stack Frame Design – Step 3 Generate the offsets from fp

Variable name	Initial Value	Size bytes	Alignment pad to next	Total Size
char str[]	"Hi"	3	1	4
char *ptr	str	4	0	4
short buf[3]		3 * 2	2	8
int n	0	4	0	4

Allocation Type	Total
FP_OFF + 4 = 12 + 4 = 16	16
Pad to get to 8-byte boundary	4
FRAMESZ space for Locals (PAD – FP_OFF)	24

```
        .equ   FP_OFF,      12  // local base
           // NAME,         SIZE + prev_name
        .equ   STR,         4 + FP_OFF
        .equ   PTR,         4 + STR
        .equ   BUF,         8 + PTR
        .equ   N,           4 + BUF
        .equ   PAD,         4 + N
        .equ   FRAMESZ      PAD - FP_OFF
```

Distance Offsets from fp



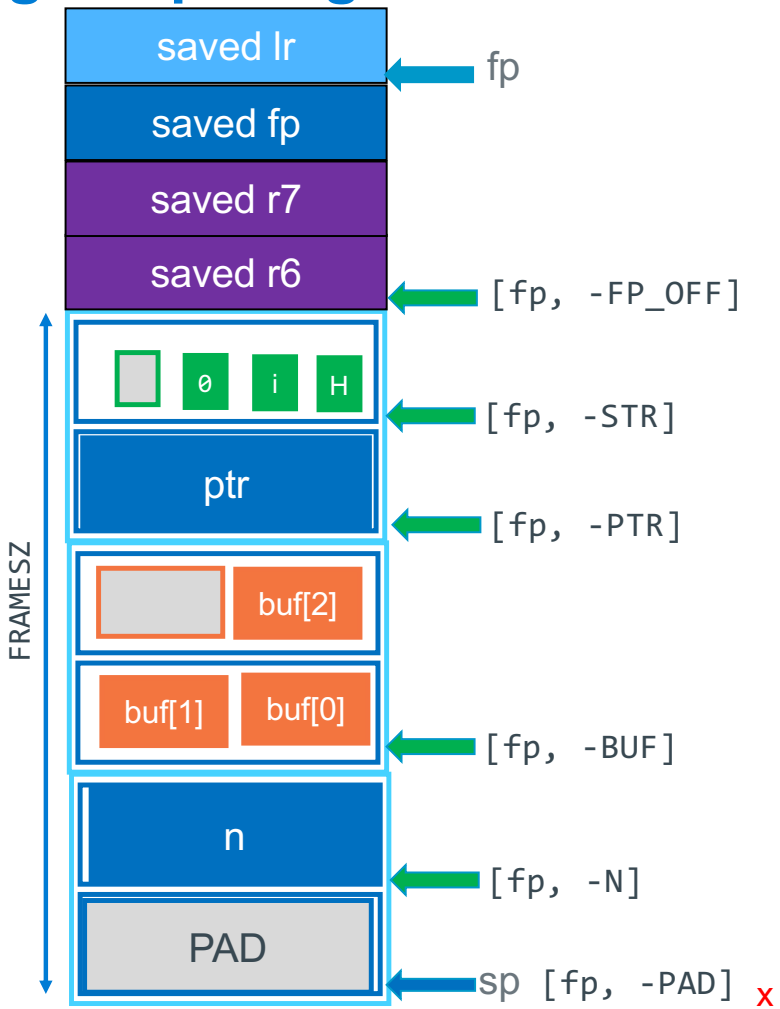
Stack Frame Design – Step 4 Modifying the prologue

```
        .equ   FP_OFF,      12  // local base
           // NAME,         SIZE + prev_name
        .equ   STR,         4 + FP_OFF
        .equ   PTR,         4 + STR
        .equ   BUF,         8 + PTR
        .equ   N,           4 + BUF
        .equ   PAD,         4 + N
        .equ   FRAMESZ      PAD - FP_OFF
```

Distance Offsets from fp

```
main:
   push    {r6, r7, fp, lr}
   add     fp, sp, FP_OFF
   sub     sp, sp, FRAMESZ // add for locals
  // no change to epilogue
```

variable	arm ldr/str statement examples
n	ldr/str r0, [fp, -N]
buf[1]	ldrh/strh r0, [fp, -BUF + 2]
&(str[0])	sub r0, fp, STR

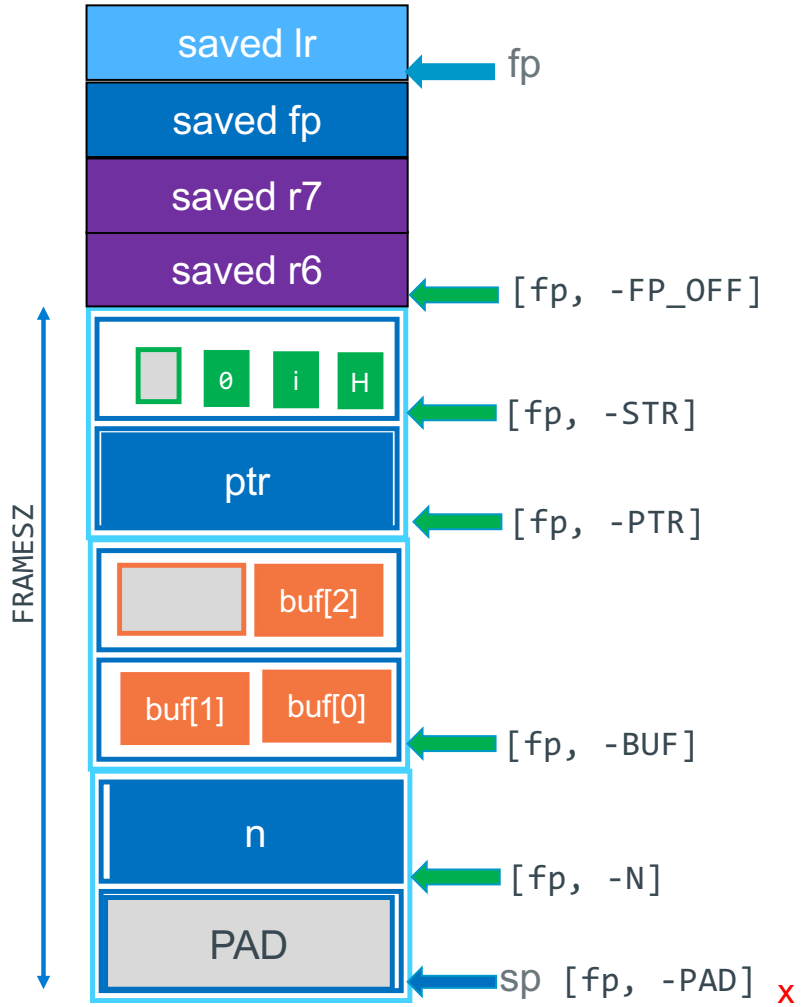


Stack Frame Design – Step 5 Initialize the variables

```
char str[] = "Hi";
char *ptr = str;
short buf[3];
// other code
int n = 0;
// other code
```

```
main:
        push    {r6, r7, fp, lr}
        add     fp, sp, FP_OFF
        sub     sp, sp, FRAMESZ
        sub     r6, fp, STR     // &(str[0])
        str     r6, [fp, -PTR]
        mov     r6, 'H'
        strb    r6, [fp, -STR]
        mov     r6, 'i'
        strb    r6, [fp, -STR+1]
        mov     r6, 0
        strb    r6, [fp, -STR+2]
        // other code
        mov     r6, 0
        str     r6, [fp, -N]
```

Used in PA5



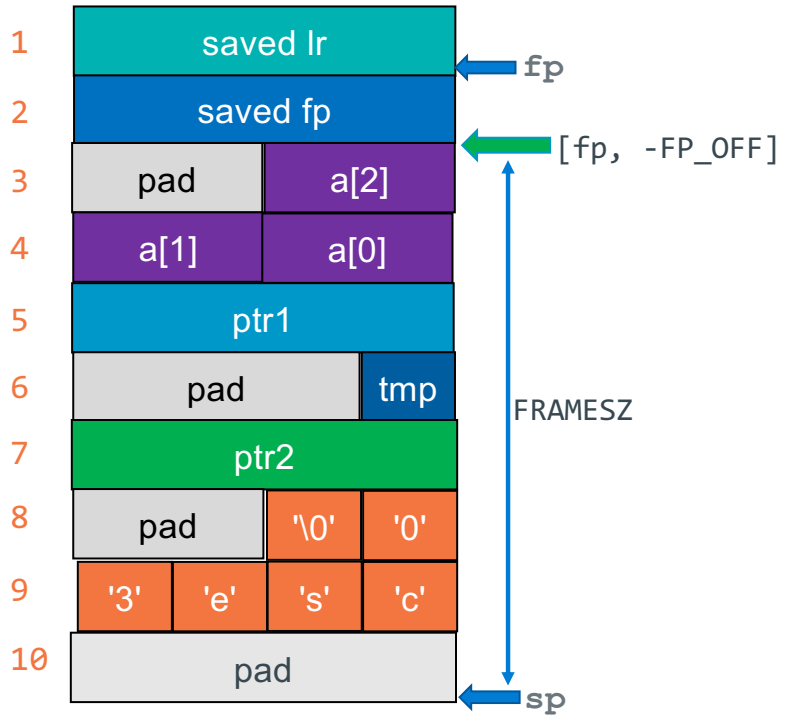
x

Local Variables: Stack Frame Design Practice

Example shows allocation **without reordering** variables to optimize space

```
short a[3];
short *ptr1;
char tmp;
char *ptr2;
char nm[] = "cse30";
```

```
.equ   FP_OFF,       4  // Local base
// NAME,            SIZE + prev_name
.equ   A,            8 + FP_OFF
.equ   PTR1,         4 + A
.equ   TMP,          4 + PTR1
.equ   PTR2,         4 + TMP
.equ   NM,           8 + PTR2
.equ   PAD,          4 + NM
.equ   FRAMESZ       PAD - FP_OFF // for locals
```



When writing real code, you do not have to put all locals on the stack

- Place locals in registers if they fit, are accessed often, **and**
- You do not need their address (they are not an output variable in a function call)

x

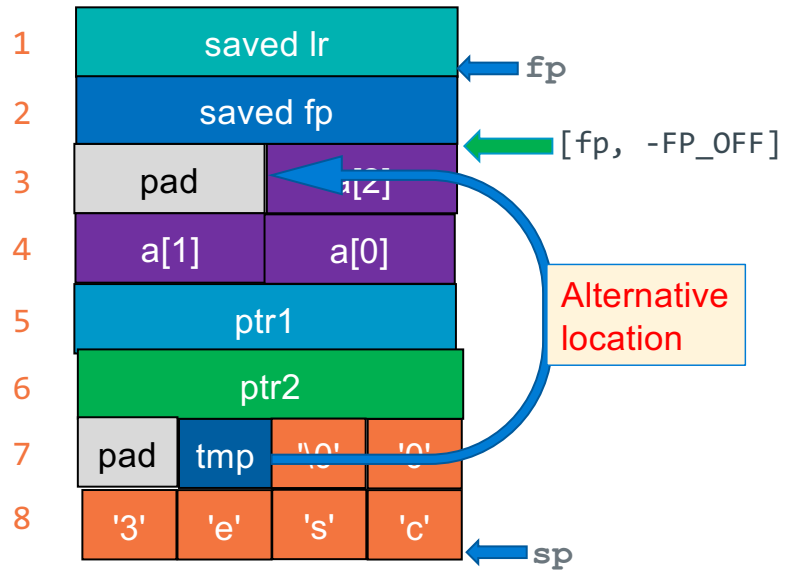
Local Variables: Stack Frame Design Reordering

Example shows allocation **with reordering** variables to optimize space

```
short a[3];
short *ptr1;
char *ptr2;
char tmp;
char nm[] = "cse30";
```

```
.equ   FP_OFF,       4  // Local base
// NAME,           SIZE + prev_name
.equ   A,            8 + FP_OFF
.equ   PTR1,         4 + A
.equ   PTR2,         4 + PTR1
.equ   TMP,          2 + PTR2
.equ   NM,           6 + TMP
.equ   PAD,          0 + NM // not needed
.equ   FRAMESZ       PAD - FP_OFF
```

size change

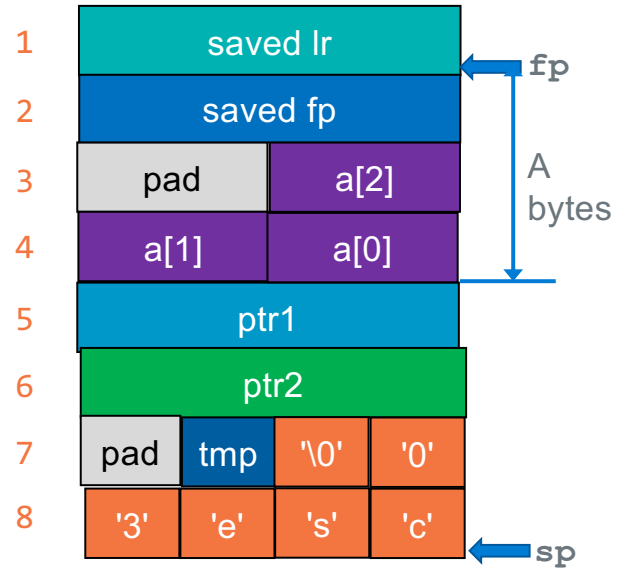


When writing real code, you do not have to put all locals on the stack

- Place locals in registers if they fit, are accessed often, **and**
- You do not need their address (they are not an output variable in a function call)

x

Entire source file



	Evaluated into r0
&(a[1])	sub r0, fp, A - 2
&(a[1])	add r0, fp, -A + 2
&(nm[1])	add r0, fp, -NM + 1
ptr2	add r0, fp, -PTR2

```
        .arch armv6
        .arm
        .fpu vfp
        .syntax unified
         // globals etc here
        .text
        .type       doit, %function
        .global     doit
        .equ        EXIT_SUCCESS, 0
        .equ        FP_OFF,   4  // Local base
        .equ        A,        8 + FP_OFF
        .equ        PTR1,     4 + A
        .equ        PTR2,     4 + PTR1
        .equ        TMP,      2 + PTR2
        .equ        NM,       6 + TMP
        .equ        PAD,      0 + NM
        .equ        FRAMESZ   PAD - FP_OFF
doit:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        sub     sp, sp, FRAMESZ
        // doit() code goes here
        mov     r0, EXIT_SUCCESS

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

With large frames you may need to use ldr if the immediate value FRAMESZ does not fit in imm8 (r3 is not a parameter in this example)

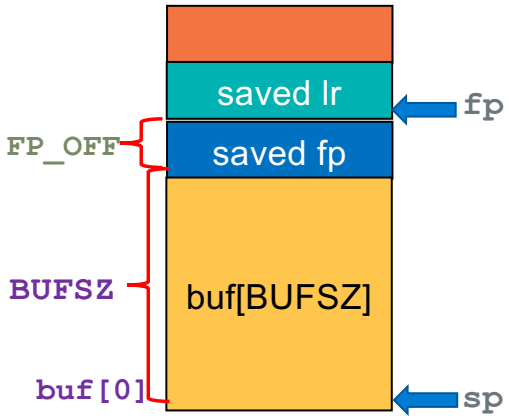
```
ldr     r3, =FRAMESZ
sub     sp, sp, r3
```


x

Passing an Output Parameter

```
#define BUFSZ 256
int main(void)
{
  char buf[BUFSZ];
  if (fgets(buf, BUFSZ, stdin) != NULL)
    printf("%s", buf);
  return EXIT_SUCCESS;
}
```

```
char *fgets(char *s, int size, FILE *stream);
```
returns *s or NULL r0, r1, r2




```
if the immediate value
of BUF does not fit in
imm8
ldr     r0, =BUF
sub     r0, fp, r0

if the immediate value
of BUFSZ does not fit in
imm8
ldr     r1, =BUFSZ
```

```
        .extern printf
        .extern fgets
        .extern stdin
        .section .rodata
.Lpfstr .string  "%s"
        .text
         // function header stuff not shown
        .equ    BUFSZ,      256
        .equ    FP_OFF,     4
        .equ    BUF,        BUFSZ + FP_OFF
        .equ    FRAMESZ,    BUF - FP_OFF
main:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        sub     sp, sp, FRAMESZ
        sub     r0, fp, BUF
        mov     r1, BUFSZ
        ldr     r2, =stdin
        ldr     r2, [r2]
        bl      fgets
        cmp     r0, NULL
        beq     .Ldone
        mov     r1, r0
        ldr     r0, =.Lpfstr
        bl      printf
.Ldone: // rest of file not shown
```

stdin is a global variable pointer *FILE

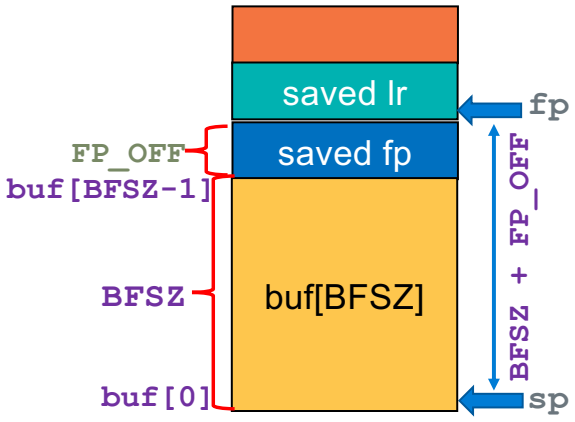
```
r0 = &(buf[0]);
r1 = BUFSZ;
r2 = stdin
```

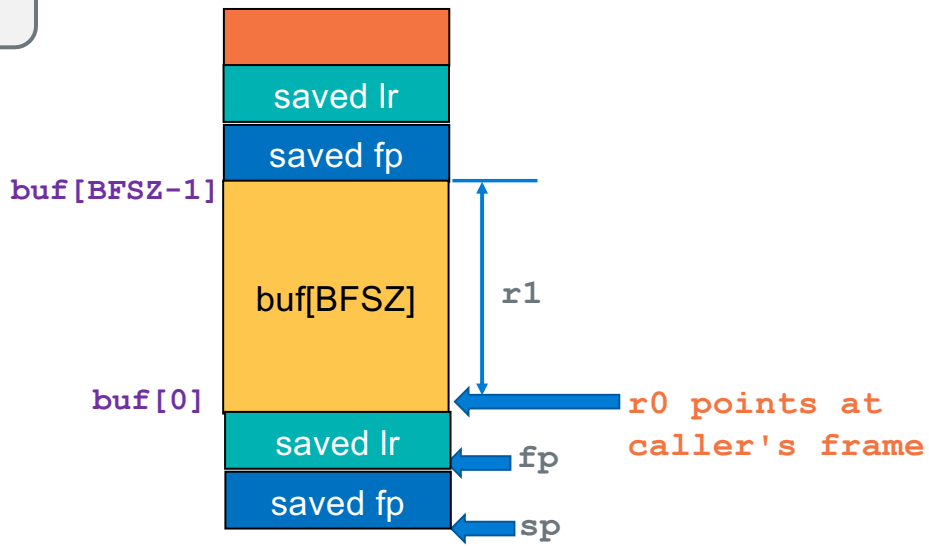

x

Writing Functions: Receiving an Output Parameter - 1

```
#define BFSZ 256
void fillbuf(char *s, int len, char fill);
int main(void)   r0,       r1,       r2
{
  char buf[BFSZ];
  fillbuf(buf, BFSZ, 'A');
  return EXIT_SUCCESS;
}
```

```
void fillbuf(char *s, int len, char fill)
{    r0,       r1,       r2
    char enptr = s + len;
    while (*s < enptr)
       *(s++) = fill;
}
```

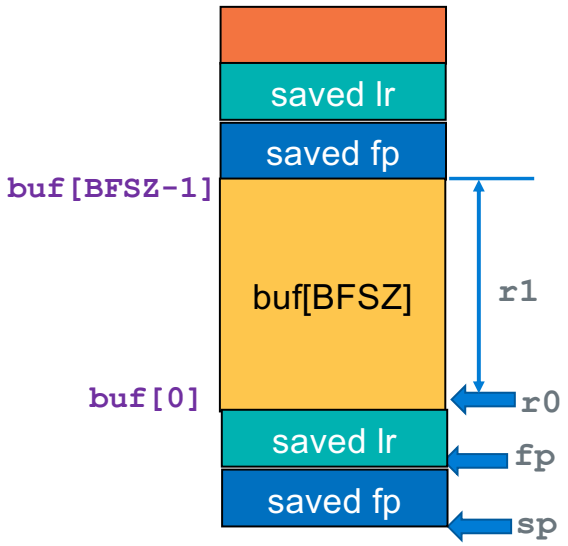




x

Writing Function: Receiving an Output Parameter - 2

```
void        r0,         r1,         r2
fillbuf(char *s, int len, char fill)
{
    char enptr = s + len;
    while (s < enptr)
        *(s++) = fill;
}
```



Using r1 for endptr

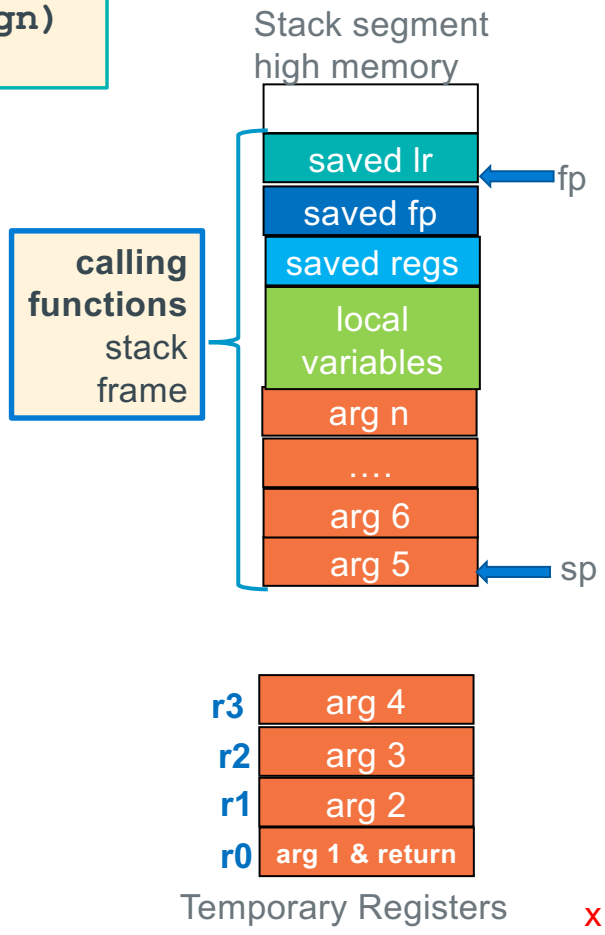
```
fillbuf:
    push    {fp, lr}        // stack frame
    add     fp, sp, FP_OFF  // set fp to base

    add     r1, r1, r0      // copy up to r1 = bufpt + cnt
    cmp     r0, r1          // are there any chars to fill?
    bge     .Ldone          // nope we are done

.Ldowhile:
    strb    r2, [r0]        // store the char in the buffer
    add     r0, 1           // point to next char
    cmp     r0, r1          // have we reached the end?
    blt     .Ldowhile       // if not continue to fill

.Ldone:
    sub     sp, fp, FP_OFF  // restore stack frame top
    pop     {fp, lr}        // restore registers
    bx      lr              // return to caller
```


x

Passing More Than Four Arguments - 1

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

- Each argument is a value that must fit in 32-bits
- **Args > 4 are in the caller's stack frame and arg 5 always starts at fp+4**
  - At the function call (bl) sp points at arg5
  - Additional args are higher up the stack, with one argument "slot" every 4-bytes
- Called functions have the right to change stack args just like they can change the register args!
- Caller must assume all args including ones on the stack are changed by the caller



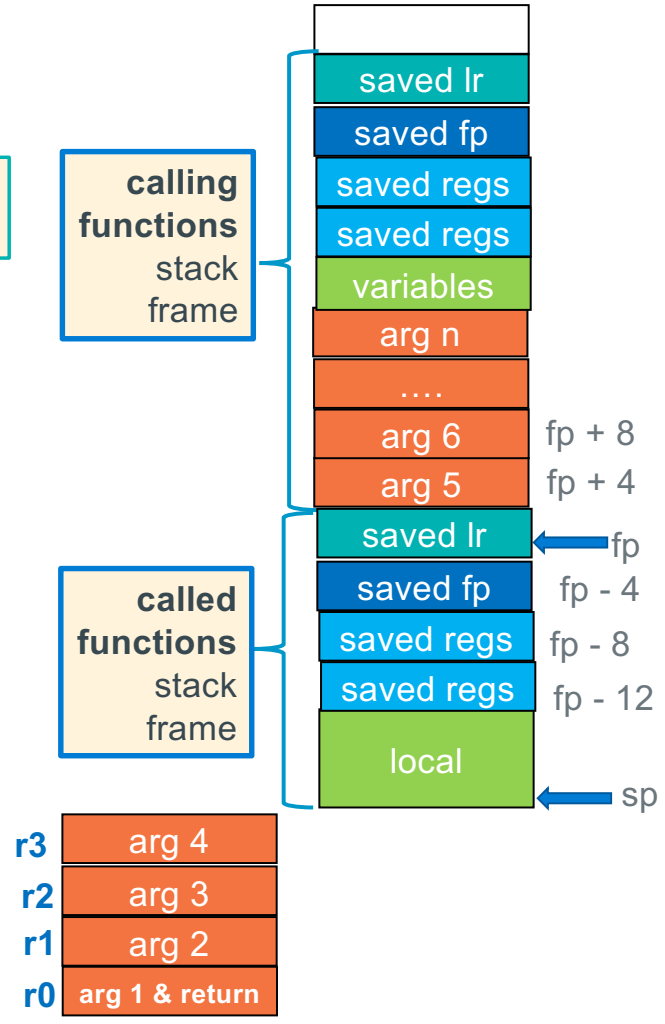
x

Passing More Than Four Arguments - 2

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

- **Addressing rules**
  - Adding to fp to get arg address in caller's frame
  - Subtracting from fp are addresses in called frame
- Why does it work this way?
- This "algorithm" for finding args was designed to enable languages to have variable argument count functions like:

```
printf("conversion list", arg0, … argn);
```

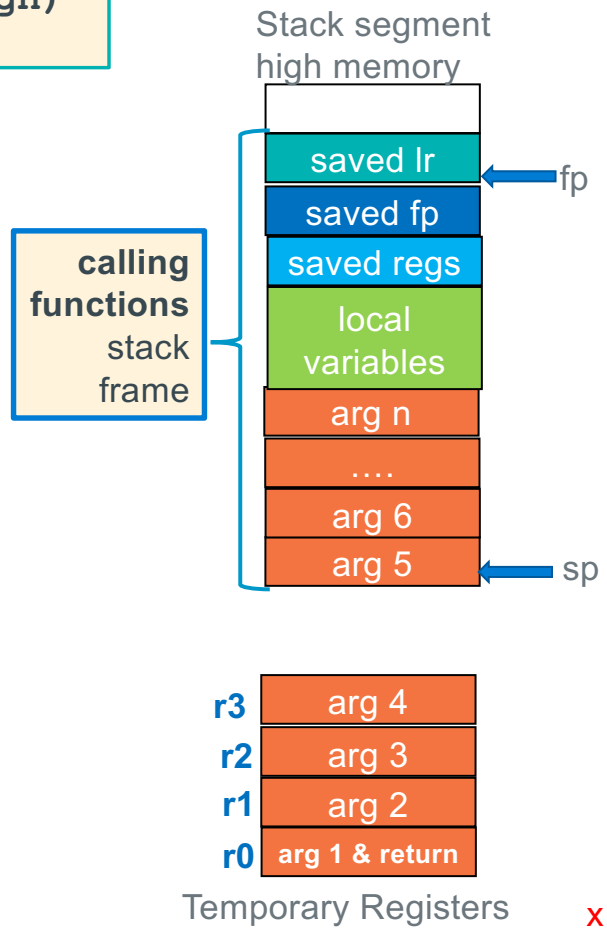


x

Passing More Than Four Arguments – Calling Function

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

- Calling function prior to making the call
  1. Evaluate first four args: place resulting values in r0-r3
  2. Arg 5 and greater are evaluated
  3. Store Arg 5 and greater parameter values on the stack
- **One arg value per slot!** – NO arrays in a slot
- chars, shorts and ints are directly stored
- Structs (not always), arrays are passed via a pointer
- **Pointers** passed as output parameters usually contain an address ***that points at*** the stack, BSS, data, or heap



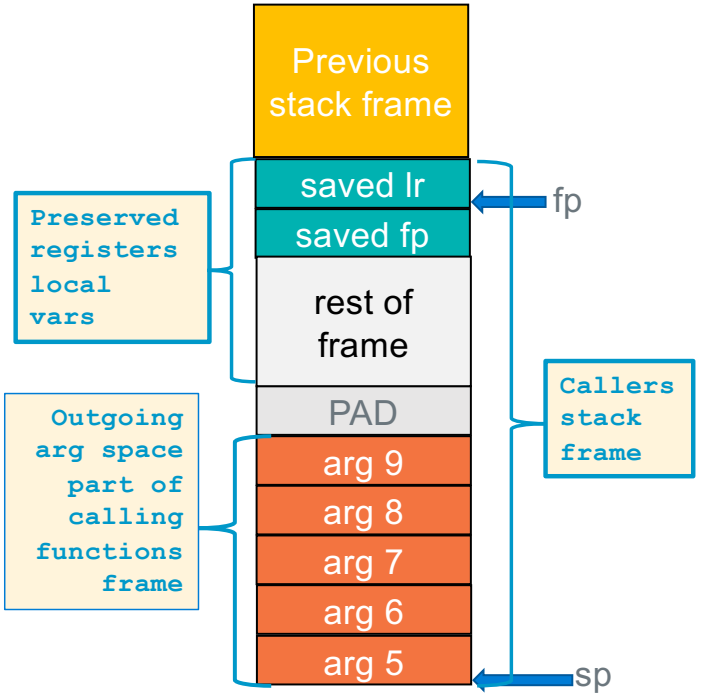
Calling Function: Allocating Stack Parameter Space

At the point of a function call (and obviously at the start of the called function):

1. sp must point at arg5
2. arg5 **must be at an 8-byte boundary**,
   a) **padding** to force arg5 alignment is **placed above** the last **argument the called function is expecting**

Approach: Extend the stack frame to include enough space for stack arguments function with the greatest arg count

1. Examine every function call in the body of a function
2. Find the function call with greatest arg count, Determines space needed for outgoing args
3. Add the space needed to the frame layout



Rules: At point of call

1. **arg5 must be pointed at by sp**
2. **SP must be 8-byte aligned**

x

Determining the Passed Parameter Area on The Stack

- Find the function called by main with the largest number of parameters
- That function determines the size of the Passed Parameter allocation on the stack

```
int main(void)
{
    /* code not shown */
    a(g, h);

   /* code not shown */
    sixsum(a1, a2, a3, a4, a5, a6);

   /* code not shown */

   b(q, w, e, r);
   /* code not shown */
}
```

largest arg count is 6
allocate space for 6 - 4 = 2 arg slots

x

Passing More than Four Args – Six Arg Example

- Problem: Write and call a function that receives six integers and returns the sum
- First 4 parameters are in register r0 - r3 and the remaining argument are on the stack
- For this example, we will put all the locals on the stack

```
int main(void)
{
    int cnt = sixsum(1, 2, 3, 4, 5, 6);

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

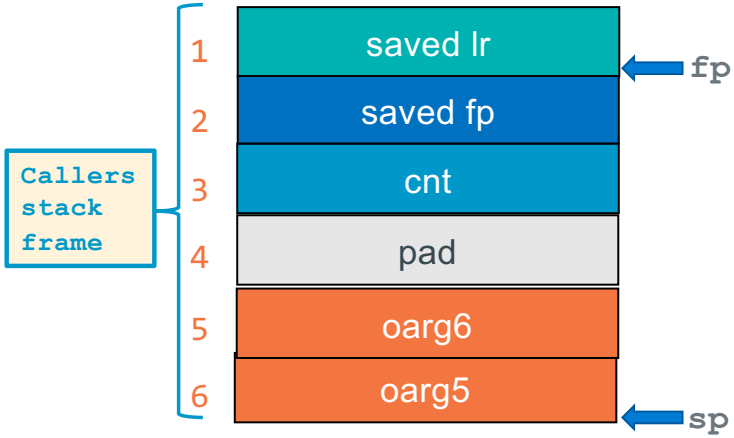
```
int
sixsum(int a1, int a2, int a3, int a4, int a5, int a6)
{
    return a1 + a2 + a3 + a4 + a5 + a6;
}
```


x

Calling Function > 4 Args - 1

```
int cnt = sixsum(1, 2, 3, 4, 5, 6);
```

```
        .equ    FP_OFF,         4  // local base
           // NAME,             SIZE + prev_name
        .equ    CNT,            4 + FP_OFF
        .equ    PAD,            4 + CNT
        .equ    OARG6,          4 + PAD
        .equ    OARG5,          4 + OARG6
        .equ    FRAMESZ         OARG5 - FP_OFF
```

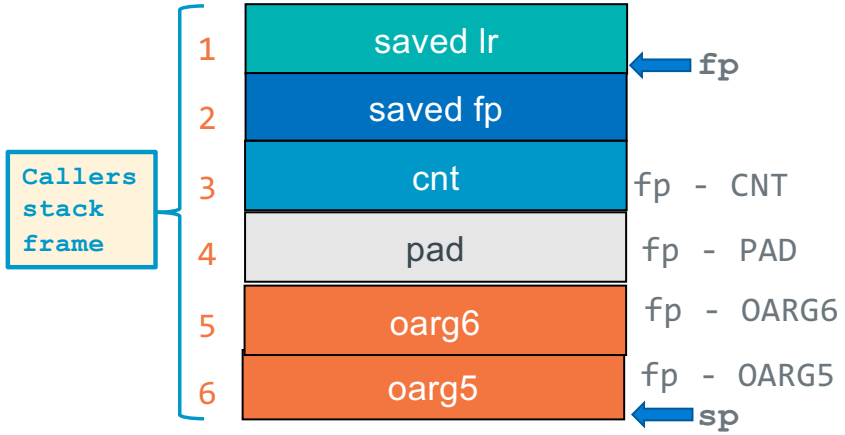


x

Calling Function > 4 Args - 2

```
int cnt = sixsum(1, 2, 3, 4, 5, 6);
```

```
.equ   FP_OFF,      4
.equ   CNT,         4 + FP_OFF
.equ   PAD,         4 + CNT
.equ   OARG6,       4 + PAD
.equ   OARG5,       4 + OARG6
.equ   FRAMESZ      OARG5 - FP_OFF
```




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

    mov     r0, 6
    str     r0, [fp, -OARG6]
    mov     r0, 5
    str     r0, [fp, -OARG5]
    mov     r3, 4
    mov     r2, 3
    mov     r1, 2
    mov     r0, 1
    bl      sixsum
    str     r0, [fp, -CNT]
    mov     r1, r0
    ldr     r0, =.Lpfstr
    bl      printf

    mov     r0, EXIT_SUCCESS
    sub     sp, fp, FP_OFF
    pop     {fp, lr}
    bx      lr
```

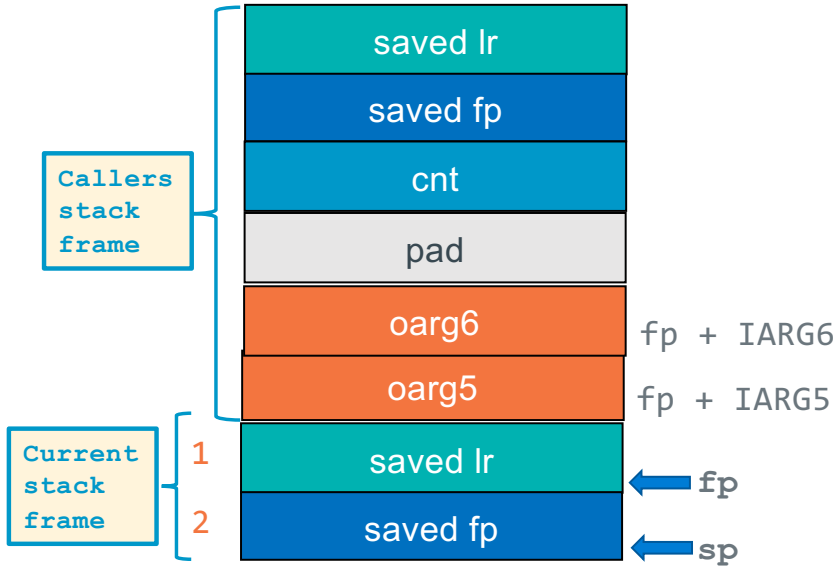

x

Called Function > 4 Args

```
int sixsum(int a1, int a2, int a3, int a4, int a5, int a6)
    return a1 + a2 + a3 + a4 + a5 + a6;
```

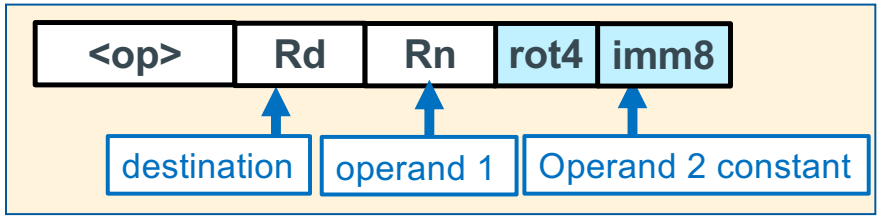
```
.equ   IARG6,       8 // offset into caller's frame
.equ   IARG5,       4 // offset into caller's frame
.equ   FP_OFF,      4 // local base
```

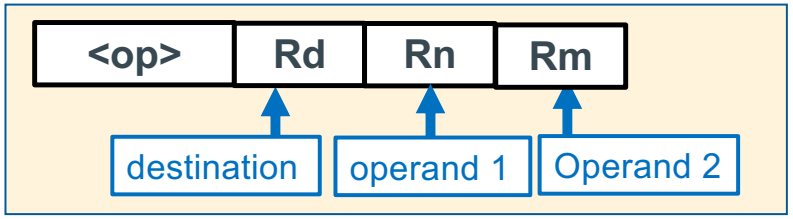
```
sixsum:
        push    {fp, lr}
        add     fp, sp, FP_OFF
        add     r0, r0, r1
        add     r0, r0, r2
        add     r0, r0, r3
        ldr     r1, [fp, IARG5]
        add     r0, r0, r1
        ldr     r1, [fp, IARG6]
        add     r0, r0, r1
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
```



x

Bitwise Instructions






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

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

Bitwise <op> description	*<op> Syntax*	*Operation*
Bitwise **AND**	`and` R_d, R_n, Op2	$R_d \leftarrow R_n$ & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	`bic` R_d, R_n, Op2	$R_d \leftarrow R_n$ & ~Op2
Bitwise **OR**	`orr` R_d, R_n, Op2	$R_d \leftarrow R_n$ | Op2
Exclusive **OR**	`eor` R_d, R_n, Op2	$R_d \leftarrow R_n$ ^ Op2

X

Bit Masks: Masking - 1

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

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

```
orr  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 ff ff lower half to 1
RSLT: r1 0xab ab ff ff
```

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

```
eor  r1, r2, r3
DATA: r2 0xab ab ab 77
MASK: r3 0x00 00 00 ff flip LSB bits
RSLT: r1 0xab ab ab 88

MASK: r3 0x00 00 00 ff apply a 2nd time
RSLT: r1 0xab ab ab 77 original value!
```

x

Bit Masks: Masking - 2

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

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

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

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

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

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

clear bit 5 to a 0 without changing the other bits

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


x

Bit Masks: Masking - 3

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

(like mvn)

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

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

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

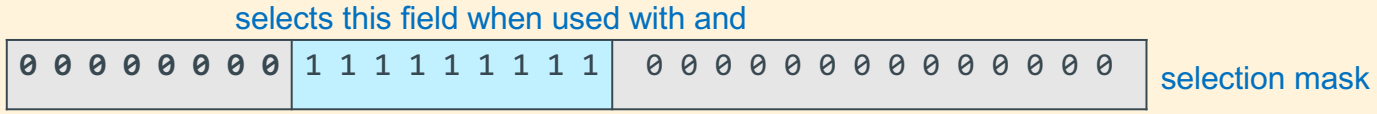
remainder (mod): num % d where n ≥ 0 and d = 2^k

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

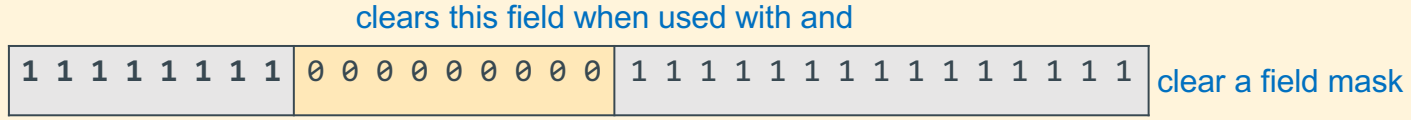

x

Masking Summary

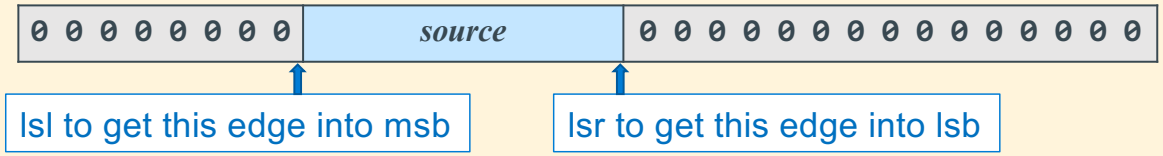
Select a field: Use **`and`** with a mask of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero



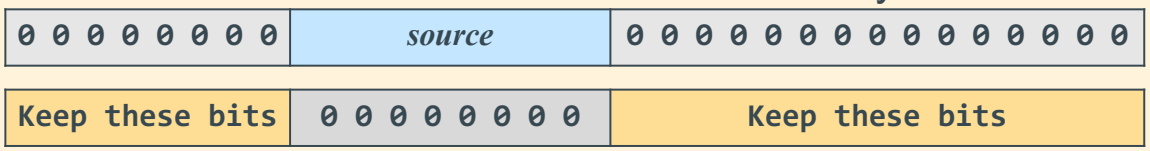
Clear a field: Use **`and`** with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero



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

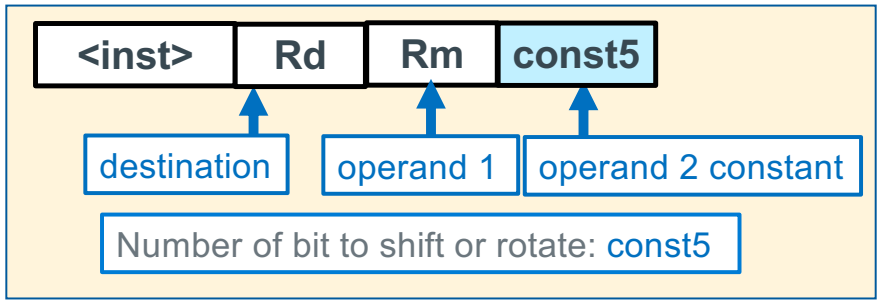


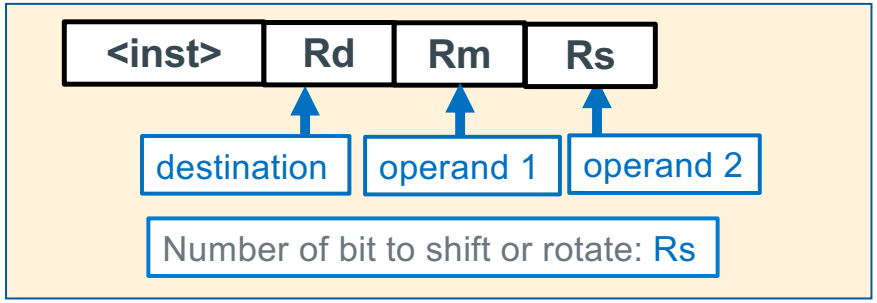
Insert a field: Use **`orr`** with fields surrounded by zeros



x

Shift and Rotate Instructions



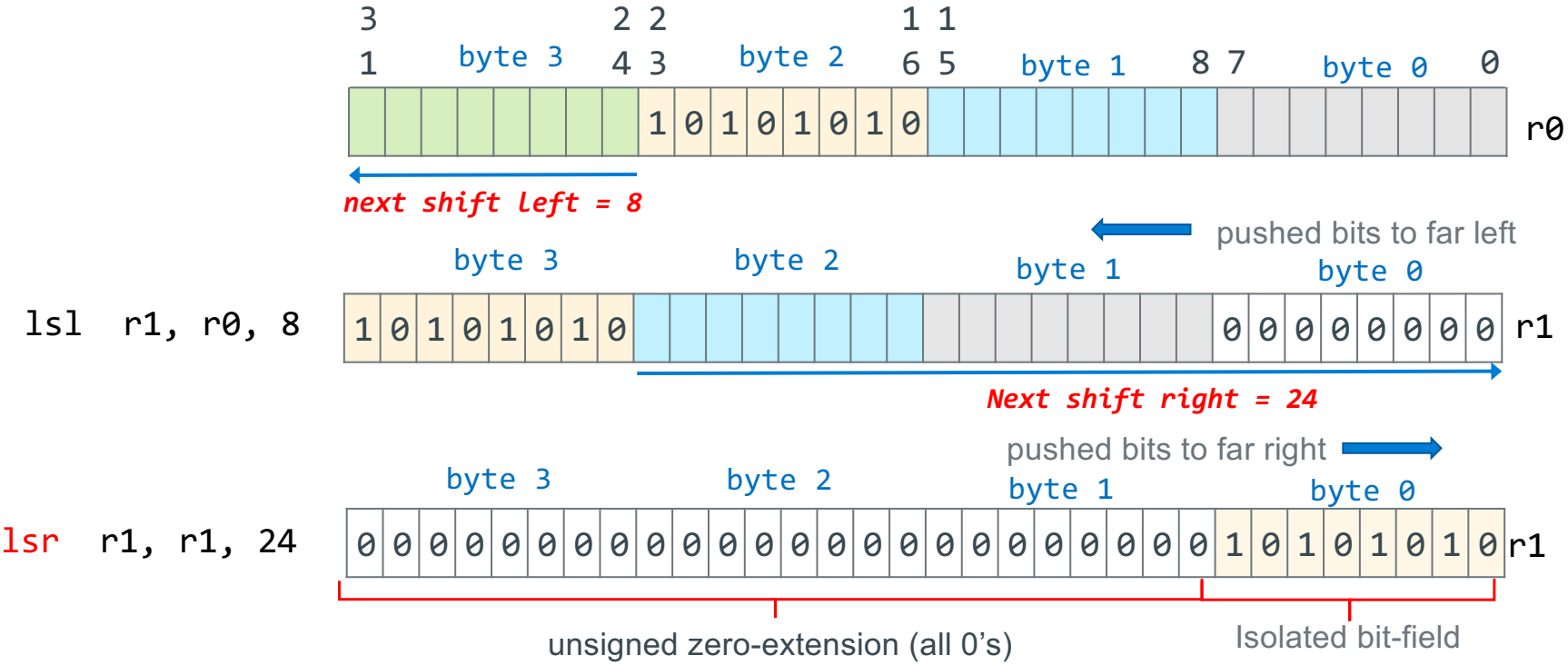


Instruction	*Syntax*	*Operation*	*Notes*	*Diagram*
Logical Shift Left	LSL R_d, R_m, *const5* LSL R_d, R_m, R_s	$R_d \leftarrow R_m$ << *const5* $R_d \leftarrow R_m$ << R_s	Zero fills shift: 0 - 31	
Logical Shift Right	LSR R_d, R_m, *const5* LSR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow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

x

Isolating Unsigned Bitfields

Hint: Useful for PA5



- You can use ror to move the field to the desired location
- Alternative: If you can create an immediate value mask with a data operation like: movn, mov, add, or sub that is often faster

x