Version 1.01

UCSD CSE 30

Computer Organization and Systems Programming

Review Session

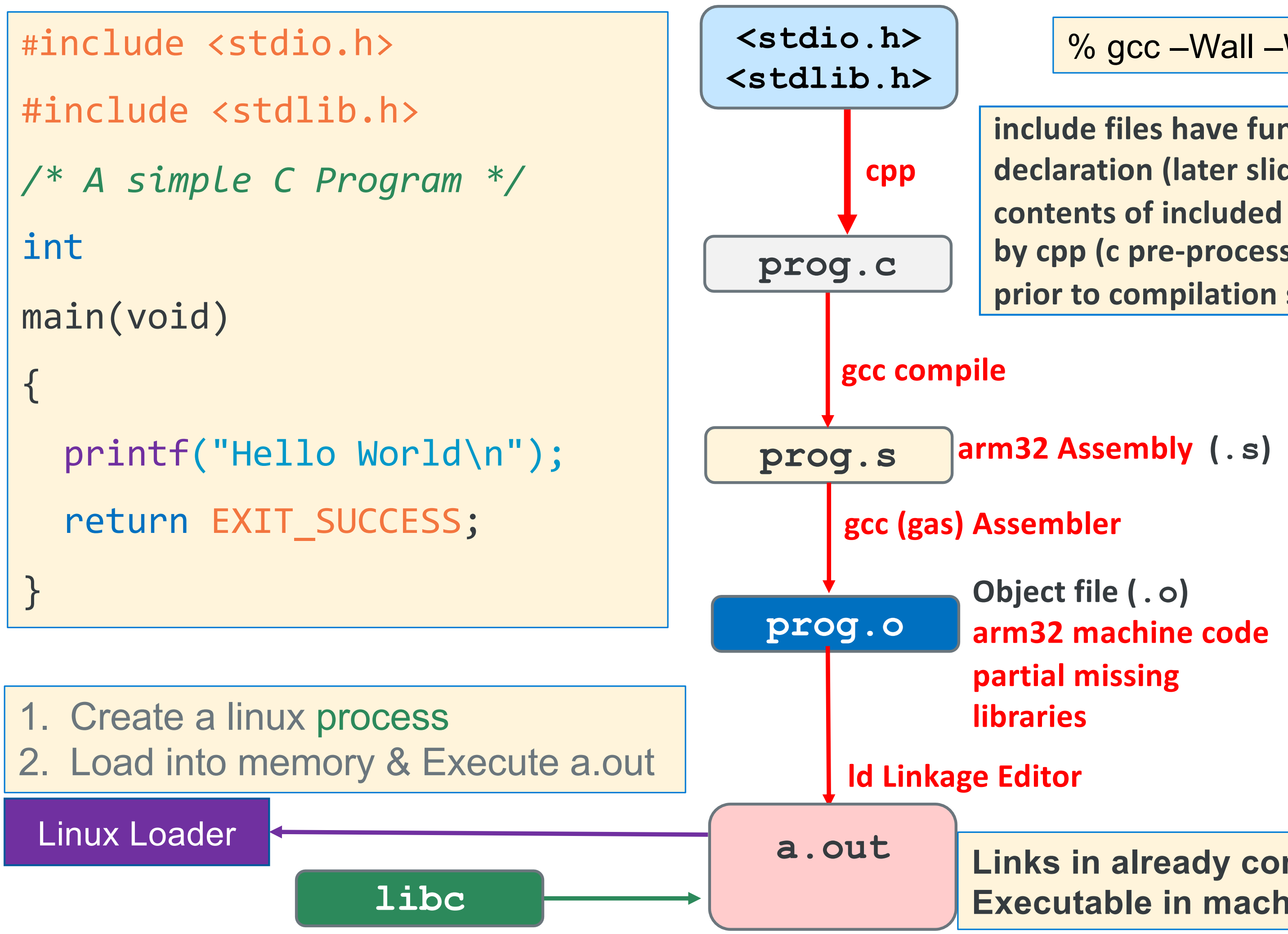
Keith Muller

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;
}
```

% gcc –Wall –Wextra –Werror prog.c

include files have function prototype and other declaration (later slides)
contents of included .h files are inserted/expanded by cpp (c pre-processor phase of gcc) into `files` prior to compilation start at their position in the file



Source to Execution Five Step Process

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

1. Create a linux process
2. Load into memory & Execute a.out

Links in already compiled standard c library
Executable in machine code file

X

Background: What is a Definition in a program language?

- **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 definition (compiler actions)**
  1. **creates code** you wrote in the functions body
  2. **allocates** memory to store the code
  3. **binds** the function name to the allocated memory
- **Variable definitions (compiler actions)**
  1. **allocates memory:** generate code to allocate space for local variables
  2. **initialize memory:** generate code to initialize the memory for local variables
  3. **binds (or associates)** the variable name to the allocated memory

x

Background What is a Declaration in programming language?

- **Declaration**: describes a *thing* – specifies types, does not create an instance
- **Function prototype** describes (more in a few slides …)
  - The type of the function return value
  - The types of each of the parameters
- **Variable declaration** describes
  - The type of a variable that is defined elsewhere
- **Derived and defined type description**
  - Later slides:(enums, struct, arrays, unions)
- In C, you must **declare a function or variable <u>before</u> 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

C Library Function API : Simple Character I/O – Used in PA3

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 (demoted 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) **promoted to an int** read **from** **`stdin`**
- see % man 3 getchar

- Make sure you use int variables with putchar() and putchar()
- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition (**-1) is outside the range of valid characters

Why is character I/O using an int?

Answer: Needs to indicate an EOF (-1) condition that is not a valid char

X

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

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

$$0x\ \underbrace{1111}_{f}\ \underbrace{1010}_{a}\ \underbrace{0101}_{5}\ \underbrace{0011}_{3}$$

$\boxed{0b1111101001010011}$

binary start with a 0b in C

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

Product Shift Left	Addend	Bit Position	Product
0	+ 0	b7	0
0	+ 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

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

x

Unsigned Integers (positive numbers) Impact of Fixed # of Bits

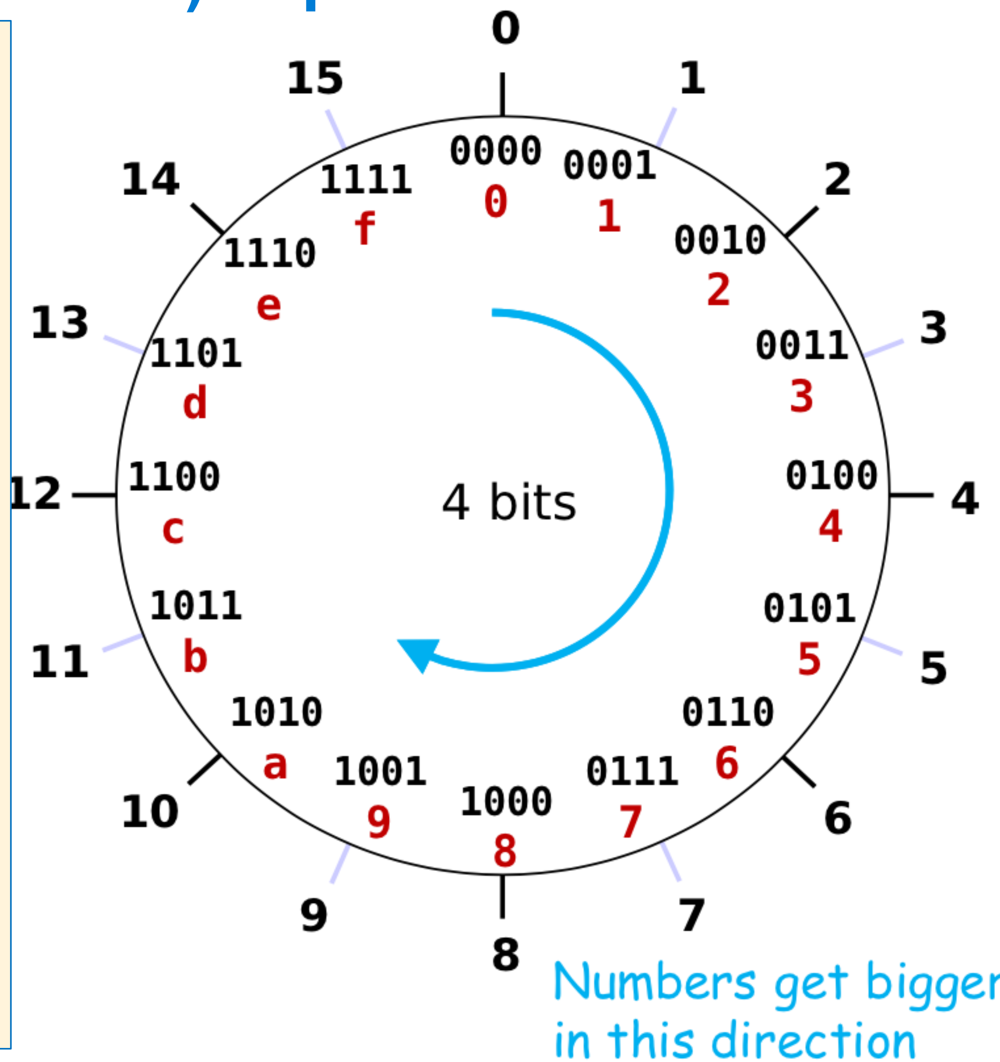
- 4 bits is 2^4 = ONLY 16 distinct values
- **Modular** (C operator: %) or clock math
  - Numbers start at 0 and “wrap around” after 15 and go back to 0
- Keep adding 1

  wraps (clockwise)

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

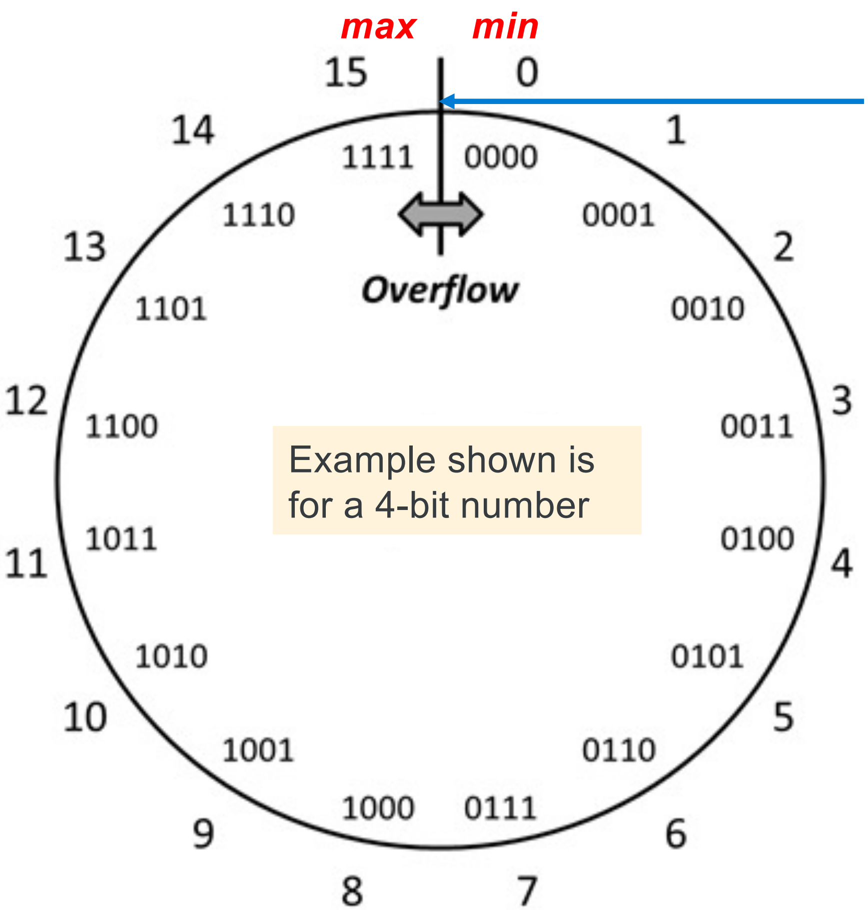
  wraps (counter-clockwise)

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



x

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

Unsigned Integer Number Overflow: Addition in 8 bits

	Carry Bit									
carries	1	1	1	1	1	1	1	1		
		1	0	1	0	0	0	0	1	161
+		0	1	0	1	1	1	1	1	95
sum		0	0	0	0	0	0	0	0	256

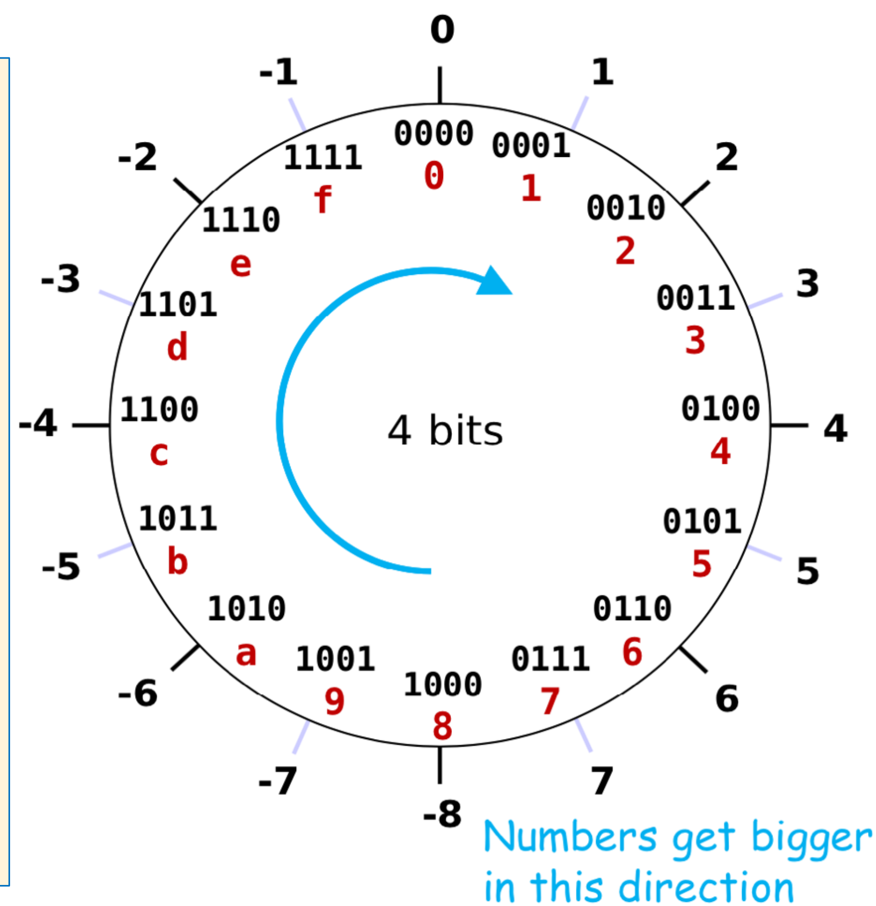
only 8 bits for numbers in this example carry bit is always dropped from result

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

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 1 further" than the positive values
- Example: the range for 8 bits:
  -128, -127, .. 0, .. 126, **+127**
- Example the range for 32 bits:
  -2147483648 .. 0, .. **+2147483647**
- *Arithmetic is the same as with unsigned binary!*



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 $(+2^i)$

b_{n-1}	b_{n-2}	. . .	b_0

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

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

Two's Complement → Unsigned for n bits

- **Unsigned Value Range**

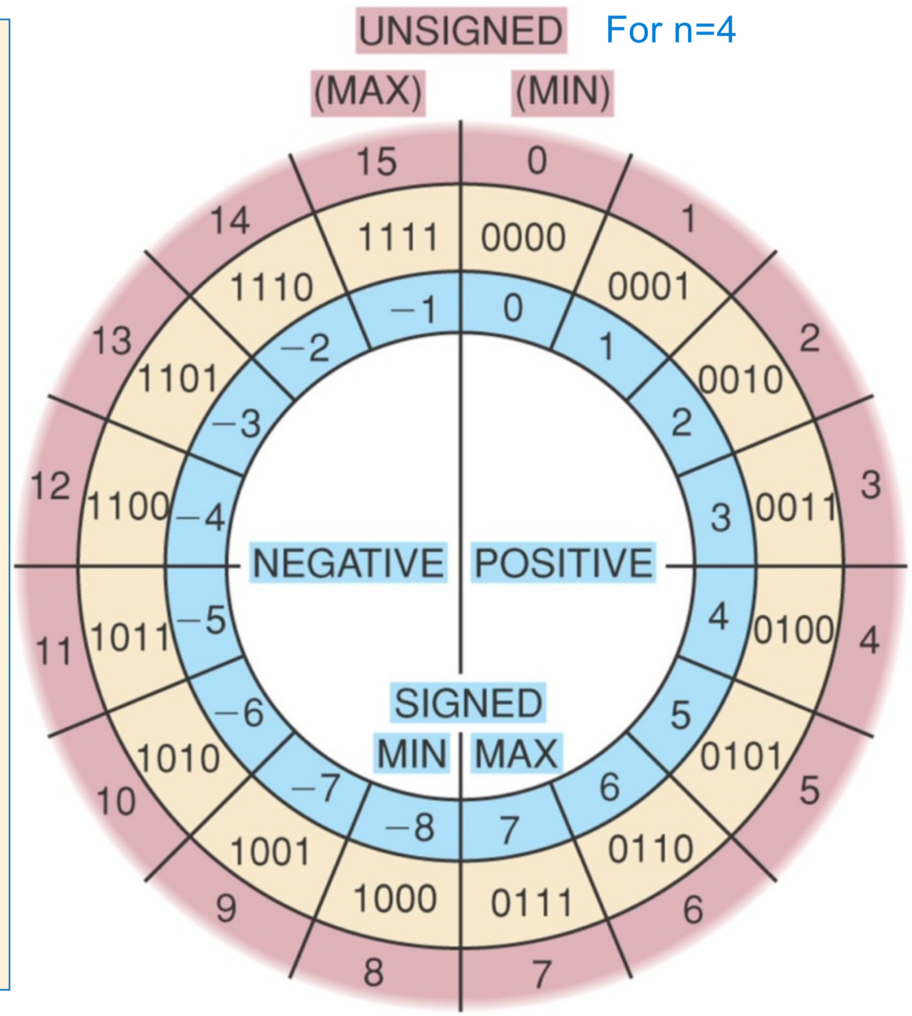
UMin = 0b00…00
= $\mathbf{0}$

UMax = 0b11…11
= $\mathbf{2^n - 1}$

- **Two's Complement Range**

SMin = 0b10…00
= $\mathbf{-2^{n-1}}$

SMax = 0b01…11
= $\mathbf{2^{n-1} - 1}$



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

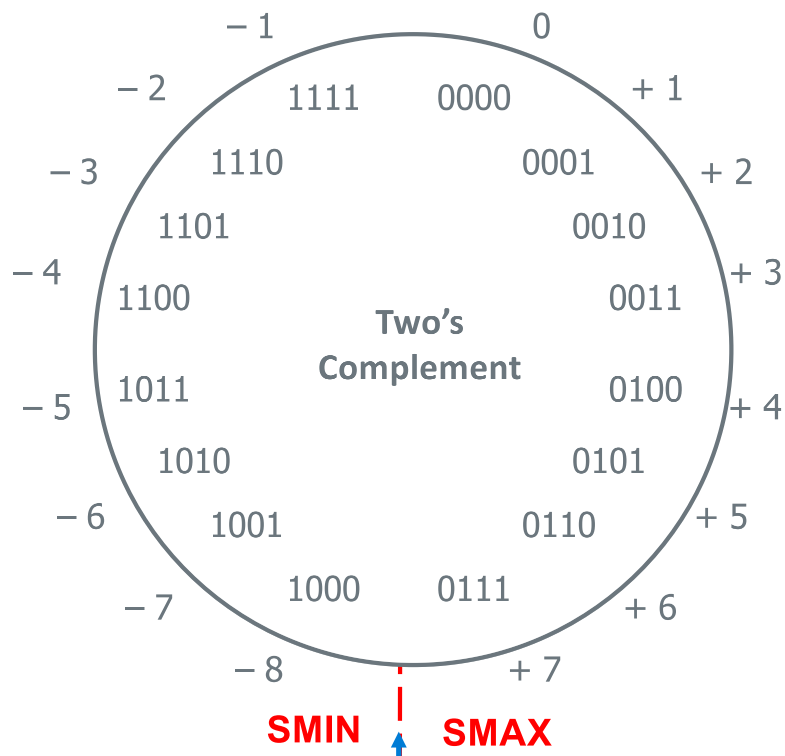
Signed Bit Bias	Bit	Bit Position	Bias
$-2^{W-1} = -2^{8-1} = -128$	x 0	b7	0
Product Shift Left	Addend	Bit Position	Product
0	+ 1	b6	1
2 x 1 = 2	+ 1	b5	3
2 x 3 = 6	+ 0	b4	6
2 x 6 = 12	+ 0	b3	12
2 x 12 = 24	+ 1	b2	25
2 x 25 = 50	+ 0	b1	50
2 x 50 = 100	+ 1	b0	SUM = 101
		Bias + SUM:	0 + 101 = 101

x

Two’s Complement Positive Overflow

- **4-bit** Two’s complement numbers (positive overflow)

Cout					
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

signed numbers: overflow occurs if
operands have same sign and result’s sign is different

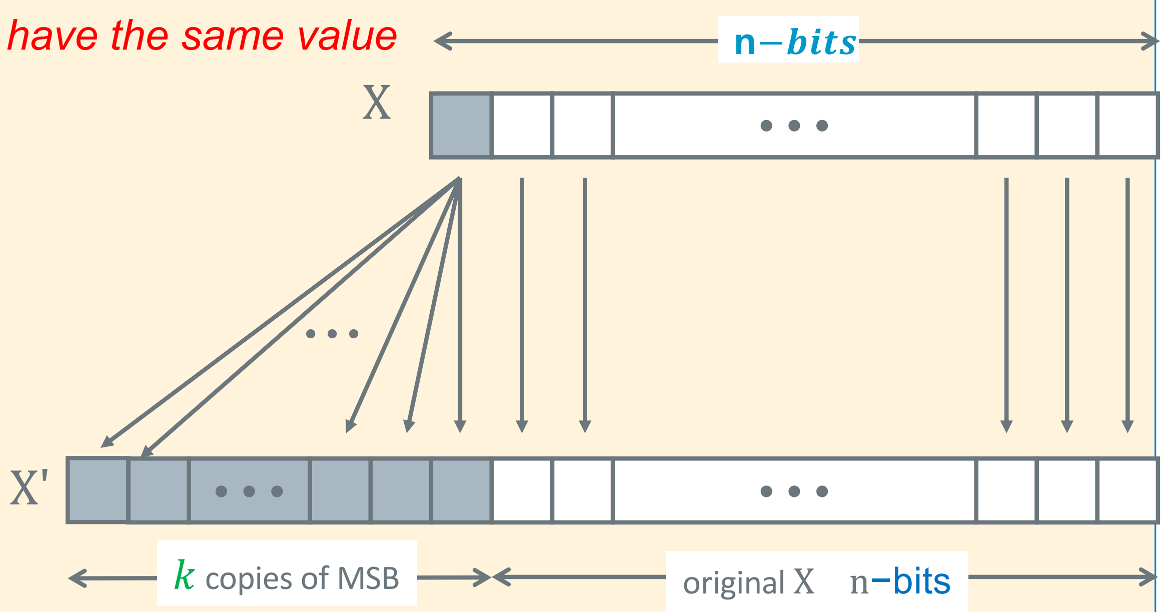
x

Sign Extension (how type promotion works)

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

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

- **Sign extension increases the number of bits: n-bit** wide signed integer X, ***EXPANDS*** to a ***wider*** n−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

```
    7     =     0111
extend to
8 bits
            00000111
Number is still 7
```

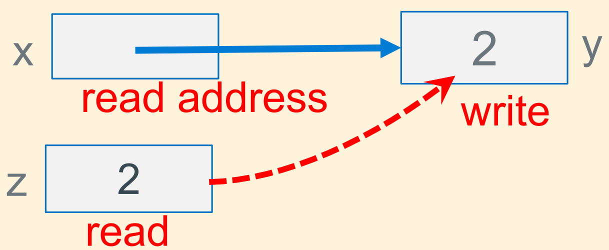
```
    1     =     0001
extend to
8 bits
            00000001
Number is still 1
```


x

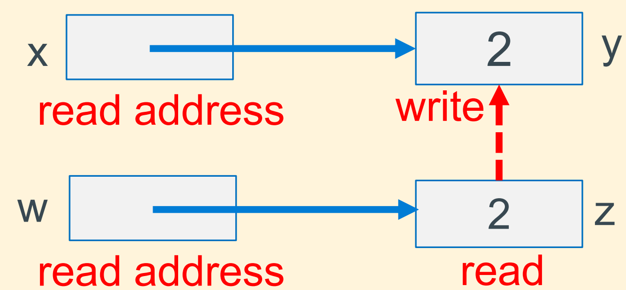
Each use of a * operator results in one additional read -2

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

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




```
int z = 2, y = 1;
int *x;
int *w;
x = &y;
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	read
x	0x104	0x1	write

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

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

```
*x on Lside is   0x108
 w on Rside is   0x100
*w on Rside is   2
```


x

Background: Different Ways to Pass Parameters

- **Call-by-reference (or pass by reference)**
  - 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)

  - 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**

x

Example Using Output Parameters

```
void inc(int *p);
int
main(void)
{
    int x = 5;
    inc(&x);
    printf("%d\n", x);
    return EXIT_SUCCESS;
}

void
inc(int *p)
{
    if (p != NULL)
        *p += 1;    // or (*p)++
}
```

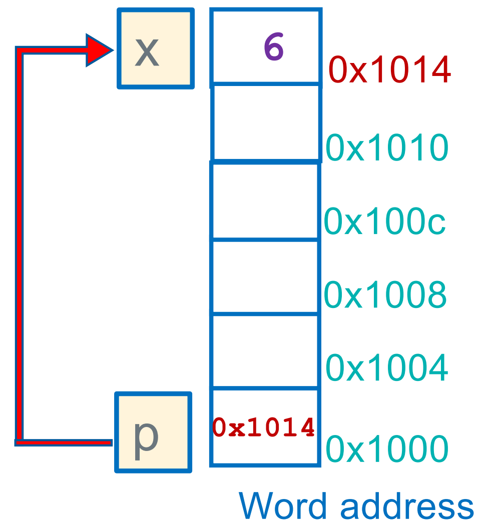
Pass the address of x (&x)

Receive an address copy (int *p)

Write to the output variable (*p)

At the Call to inc() in main()

1. Allocate space for p
2. Copy x's address into p



With a pointer to X,

inc() can change x in main()

this is called a side effect

p just like any other local variable

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, **<u>Arrays do not know their own size</u>** 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

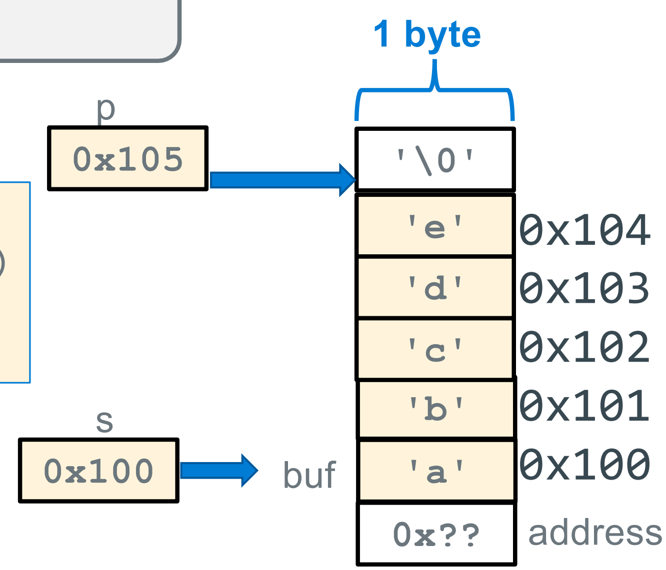
Arrays As Parameters, Approach 2: Use a sentinel element

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

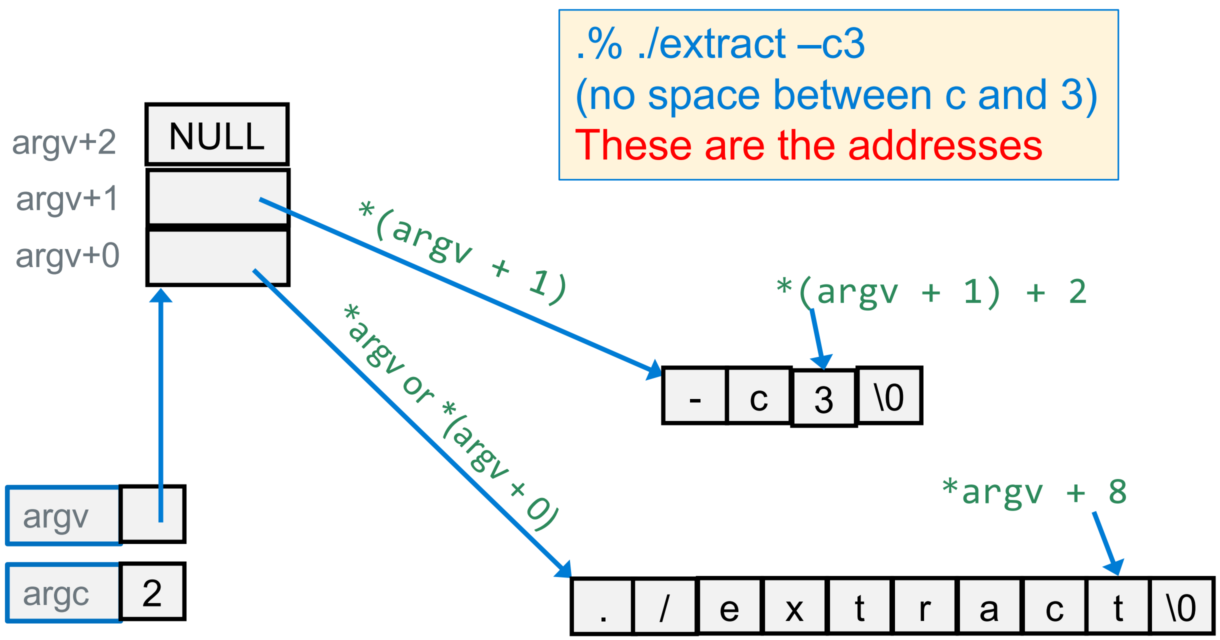
```c
int strlen(char *a);
int main(void)
{
  char buf[] = {'a', 'b', 'c', 'd', 'e', '\0'}; // string

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

```c
/* Assumes parameter is a terminated string */
int strlen(char *s)
{
   char *p = s;
   if (p == NULL)
       return 0;
   while (*p++)
       ;
    return (p - s - 1);
}
```

```c
same as:
while (*p != '\0')
    p = p + 1;
return (p - s);
```

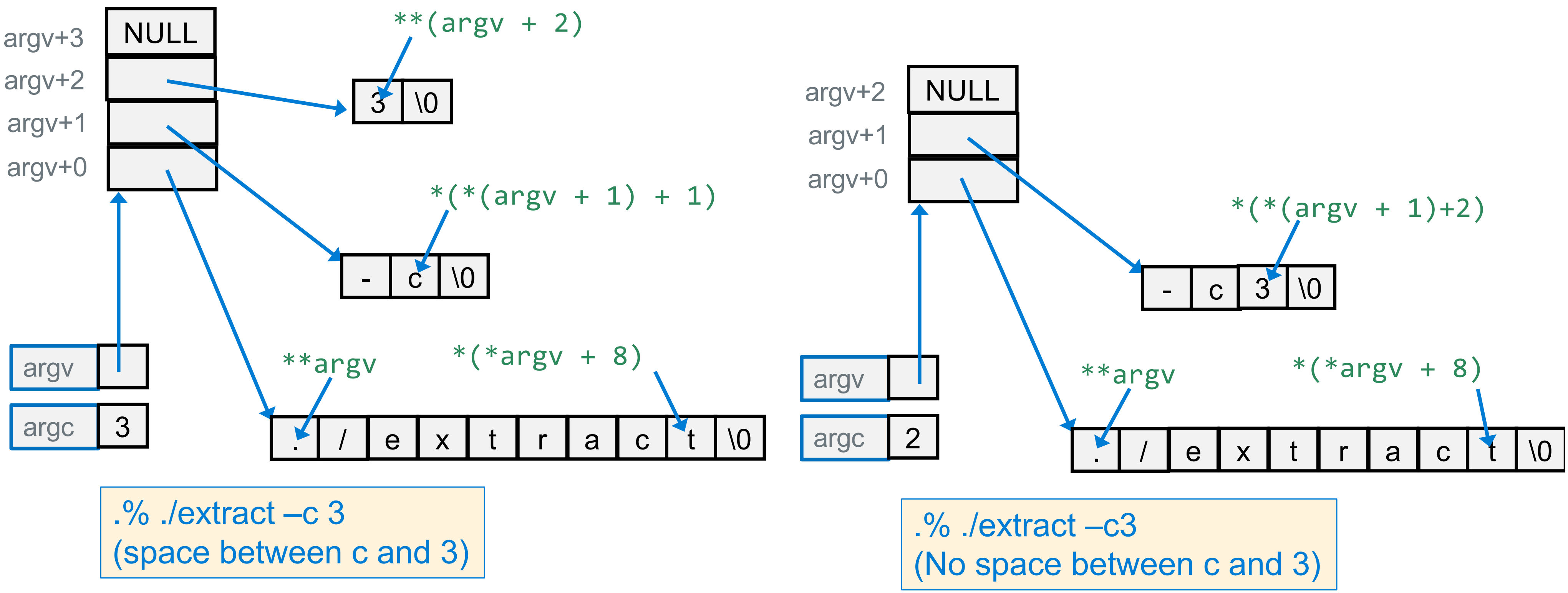


Array of Pointers: main() : argc, argv Character Address



x

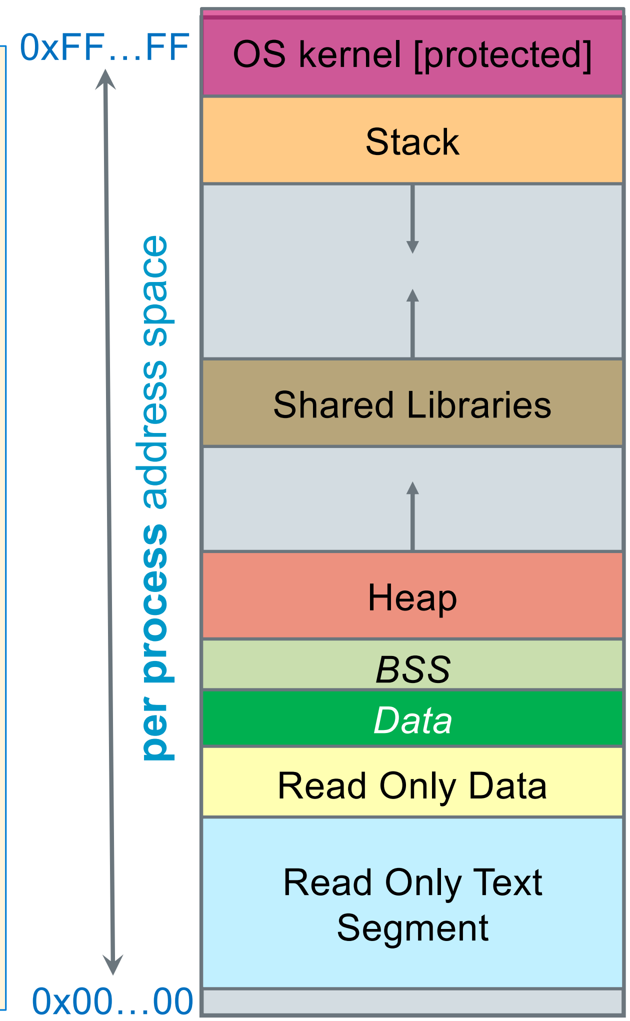
main() Command line arguments: argc, argv



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

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

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

More Dangling Pointers: Reusing "freed" memory

- 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;
}
```

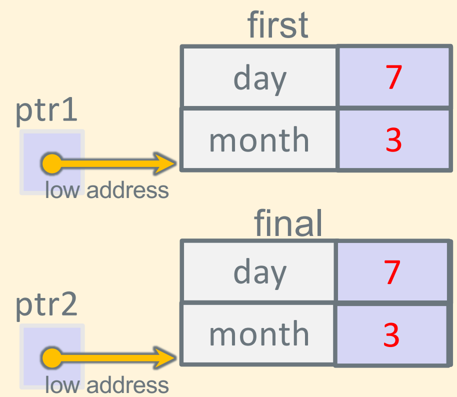
- `dangling_freed_heap()` type code often causes the allocators (**`malloc()`**and friends) to **seg fault**
  - Because it corrupts data structures the heap code uses to manage the memory pool

x

Copying Structs

- You can assign the member value(s) of the whole struct from a struct of the same type – *this copies the entire contents!*
- Individual fields can also be copied

```
struct date first = {1, 1};
struct date final = {.day= 31, .month= 12};

struct date *pt1 = &first;
struct date *pt2 = &final;

final.day = first.day; // both day are 1
final = first;         // copies whole struct

pt2->month = 3;
*pt1 = *pt2;           // copies whole struct
pt2->day = 7;
pt1->day = pt2->day;  // both days are now 7
```

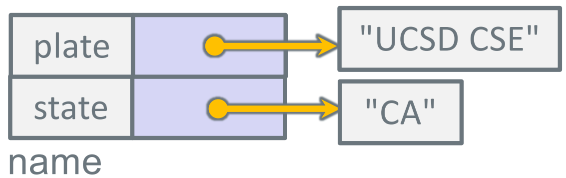


x

Struct: Copy and Member Pointers

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

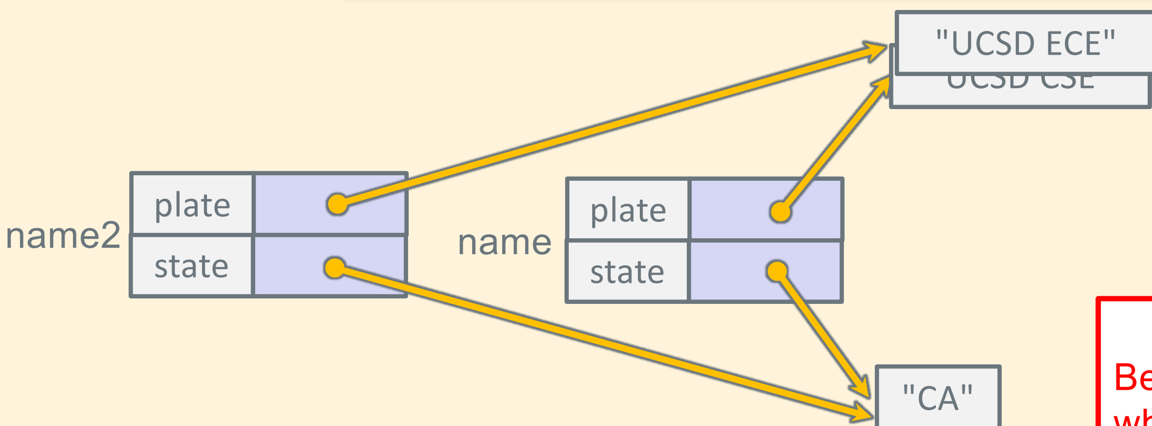
```
struct vehicle name  = {"CA", "UCSD CSE"};
struct vehicle name2;
```



immutable strings
(read only data)

- When you assign one struct to another it just copies the member fields!

```
name2 = name;   // copies members Only
```




```
name2.plate = "UCSD ECE";
```

Warning
Be **very careful** with "shallow copies" in C when pointers are involved

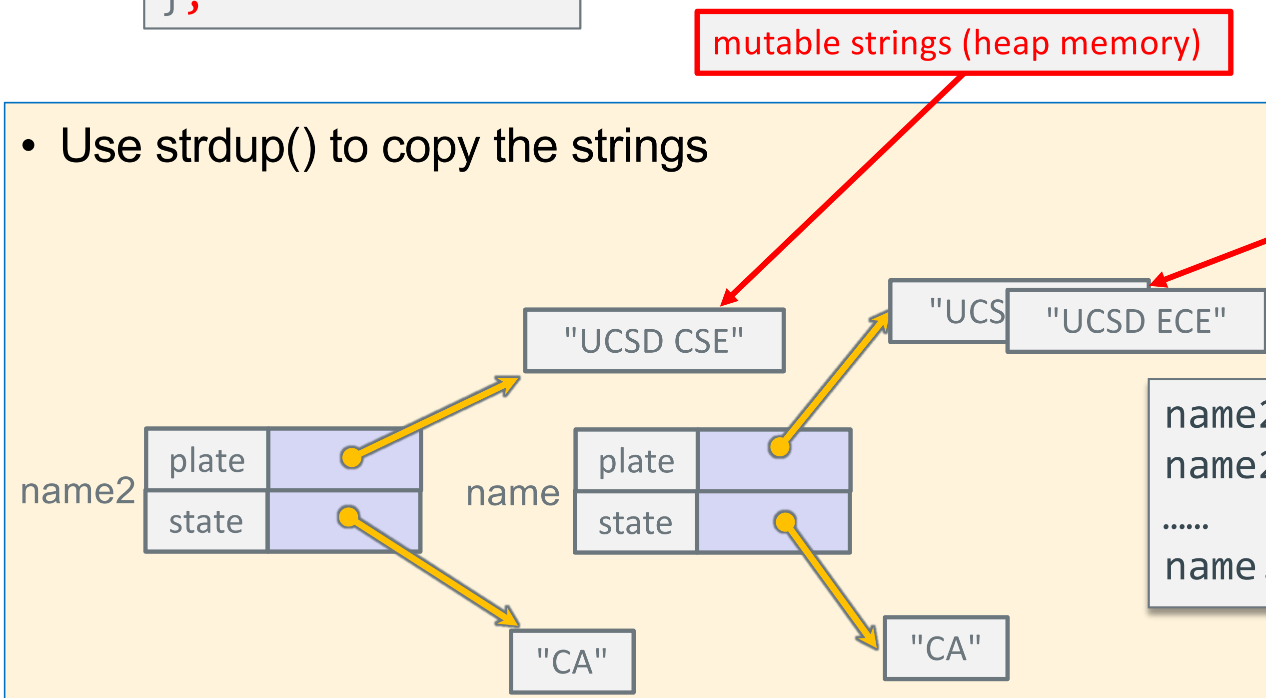
x

Struct: Copy and Member Pointers --- "Deep Copy"

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

```
struct vehicle name  = {"CA", "UCSD CSE"};
struct vehicle name2;
```

- Use strdup() to copy the strings




```
name2.plate = strdup(name.plate);
name2.state = strdup(name.state);
……
name.plate = "UCSD ECE";
```


x

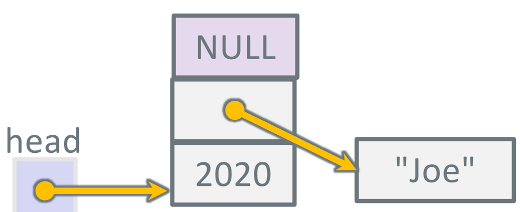
Creating a Node & Inserting it at the Front of the List

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

Must duplicate the string because of buffer reuse

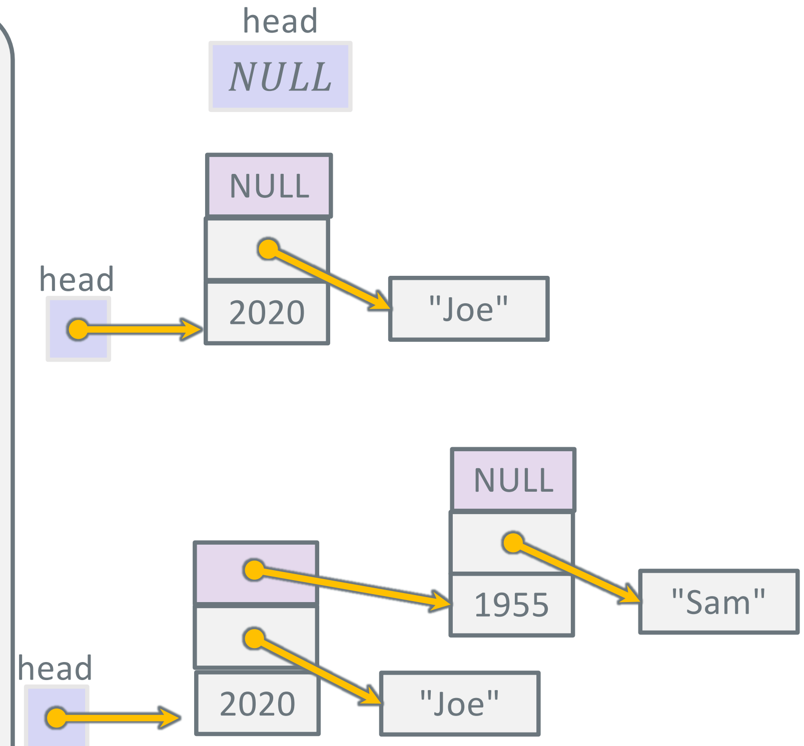
```
struct node {
    int year;
    char *name;
    struct node *next;
};
```

```
// calling function body
struct node *head = NULL; // insert at front
struct node *ptr;
char buf[BUFSZ];

if (fgets(buf, BUFSZ, stdin) != NULL) { // reads joe
    if ((ptr = creatNode(2020, buf, head)) != NULL) {
        head = ptr; // error handling not shown
    }
}
if (fgets(buf, BUFSZ, stdin) != NULL) {  // reads sam
    if ((ptr = creatNode(1955, buf, head)) != NULL) {
        head = ptr; // error handling not shown
    }
}
```



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

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

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

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

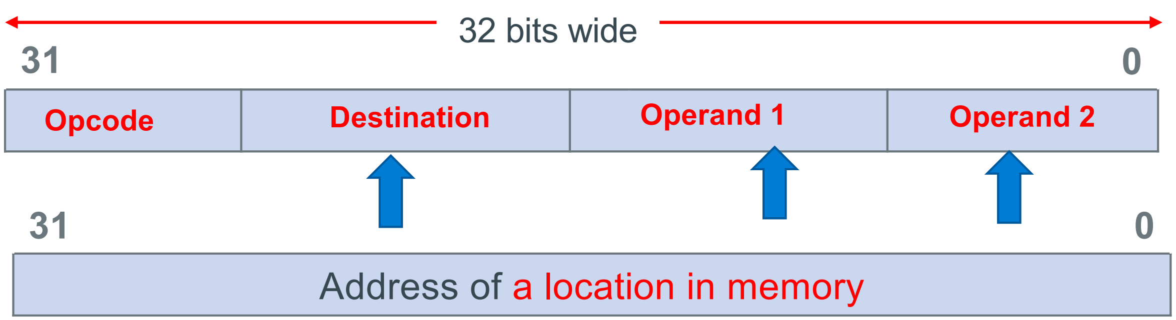


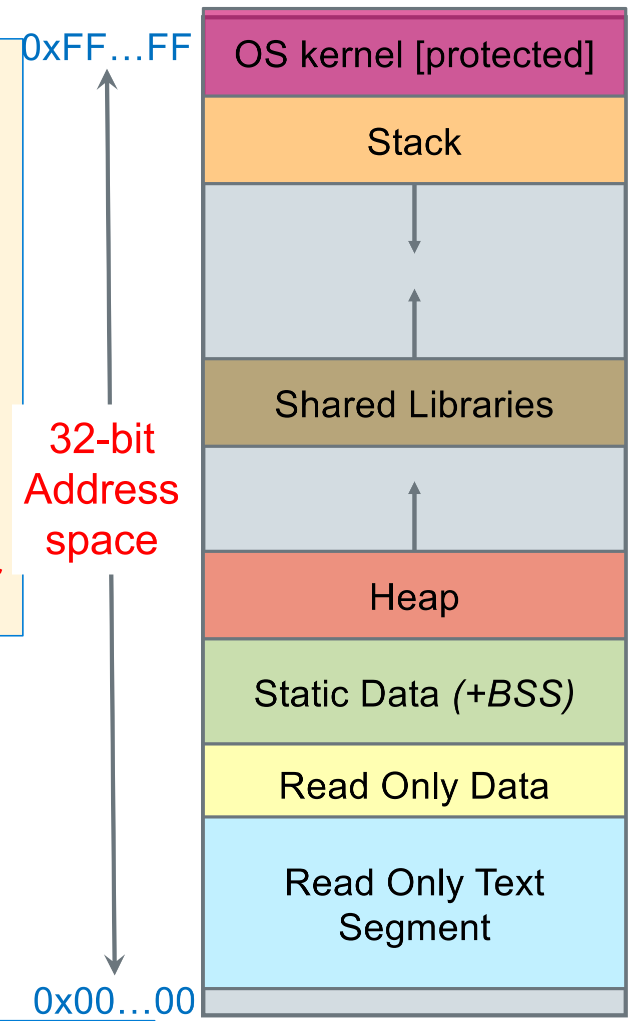

```c
struct node *head = NULL;  // insert at end
struct node *ptr;
if ((ptr = insertEnd(2020, "Joe", head)) != NULL)
    head = ptr;
if ((ptr = insertEnd(1955, "Sam", head)) != NULL)
    head = ptr;
```


x

How to Access Memory?

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





NOT ENOUGH BITS for FULL Addresses to be stored in the instruction

x

R (register) Type Data Processing: Machine Code

- Instructions that process data using three-register arguments
- The general instruction format is (not all fields will be in every instruction)

```
opcode   Rd (destination), Rn (operand 1), Rm (operand 2)
```

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	0	Rn	Rd	00000000	Rm

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

```
add r0, r1, r3
```

is encoded as

1110 0000 1000 0001 0000 0000 0000 0011

in hex is

0xe0810003

x

Arm Register Summary

- 16 Named registers r0 – r15
- The operands of almost all instructions are registers
- To operate on a variable in memory do the following:
  1. Load the value(s) from memory into a register
  2. Execute the instruction
  3. Store the result back into memory (only if needed!)
- Going to/from memory is expensive
  - 4X to 20X+ **slower** than accessing a register
- Strategy: Keep variables in registers as much as possible

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

Program Flow: Simple If statement, No Else

Approach: adjust the conditional test then **branch around** the **true block**

Use a **conditional test** that **specifies the <u>inverse</u> of the condition used in C**

C source Code	*Incorrect Assembly*	*Correct Assembly*

`int r0;`
`if (r0 > 10)`
	`cmp r0, 10`
	`bgt .Lendif`
	
	`.Lendif:`
		`cmp r0, 10`
		`ble .Lendif`
		
		`.Lendif:`

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

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

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

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

x

If with an Else Examples

Branch condition
Test (branch guard)

```
…
if (r0 < r1) {
     r2 = 1;
   // branch around else
} else {
   r2 = 0;
   /* fall through */
}
r4 = r2 + r4;
```

If r0 < r1 false
then branch
to false block

```
        cmp r0, r1
        bge .Lelse

        mov r2, 1
        b .Lendif
.Lelse:
        mov    r2, 0
       /* fall through */

.Lendif:
        add    r4, r2, r4
```

condition
true block

condition
false block

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 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 */
}
```

```
    cmp r0, 5
    beq .Lthen
    
    cmp r1, 3
    ble .Lelse
    // fall through
.Lthen:
    mov r2, r5 // true block
    // branch around else
    b .Lendif
.Lelse
    mov r5, r2 // false block
    // fall through
.Lendif:
```

If r0 == 5 true, then branch ***to*** the true block

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

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

pre-test loop guard

loop control variable

zero or more iterations

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

one or more iterations

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

post-test loop guard

Pre-Test Guards - While Loop

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

pre-test loop guard

loop iteration

inverted test

```
.Lwhile:
    cmp r1, 10
    bge .Lendw
    // block
    add r1, r1, 1
    b .Lwhile
.Lendw:
    mov r2, r1
```

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

pre-test loop guard

if stmt guard

loop iteration

```
.Lwhile:
    cmp r1, 10
    bge .Lendw
    cmp r2, r1
    beq .Lnext
    // block
.Lnext:
    add r1, r1, 1
    b .Lwhile
.Lendw:
    mov r2, r1
```


x

Post-Test Guards – Do While Loop

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

r2 = r1;
```

loop iteration

post-test loop guard

```
.Ldo:
    // block
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo

    mov r2, r1
```

test is not inverted

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

r2 = r1;
```

loop iteration

post-test loop guard

```
.Ldo:
    cmp r2, r1
    beq .Lnext
    // block
.Lnext
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo

    move r2, r1
```


x

Program Flow – Counting (For) Loop

Pre loop setup | Pre-test Loop guard | Post loop update

```
for (r1 = 0; r1 < 10; r1++) {
    /* block */
}
```

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**

Pre loop setup

Pre-test loop guard

Post loop update

```
    mov r1, 0
.Lfor:
    cmp r1, 10
    bge .Lendfr
    // block code
    add r1, r1, 1
    b .Lfor
.Lendfr:
```

Alternative

Pre-loop setup

Pre-test loop guard

Post loop update

Post-test loop guard

```
    mov r1, 0
    cmp r1, 10
    bge .Lendfr
.Ldo:
    // block code
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo
.Lendfr:
```


x

Nested loops

```
for (r3 = 0; r3 < 10; r3++) {
    r0 = 0;

    do {
        r0 = r0 + r1++;
    } while (r1 < 10);

    // fall through
    r2 = r2 + r1;
}
r5 = r0;
```

- Nest loop blocks as you would in C or Java
- Do not branch into the middle of a loop, this is hard to read and is prone to errors

```
    mov r3, 0
.Lfor:
    cmp r3, 10      // loop guard
    bge .Lendfor

    mov r0, 0

.Ldo:
    add r0, r0, r1
    add r1, r1, 1

    cmp r1, 10   // loop guard
    blt .Ldo

    // fall through
    add r2, r2, r1

    add r3, r3, 1 // loop iteration
    b .Lfor
.Lendfor:
    mov r5, r0
```


x

Bitwise Not (vs Boolean Not)

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

a	~a
0	1
1	0

Bitwise NOT

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

	Bitwise Not
number	0101 1010 0101 1010 1111 0000 1001 0110
~number	1010 0101 1010 0101 0000 1111 0110 1001

Meaning	*Operator*	*Operator*	*Meaning*
Boolean NOT	!b	~b	Bitwise NOT

Boolean operators act on the entire value not the individual bits

Type	*Operation*	result
bitwise	~0x01	1111 1111 1111 1111 1111 1111 1111 1110
Boolean	!0x01	0000 0000 0000 0000 0000 0000 0000 0000

x

Bitwise versus C Boolean Operators

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

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

& versus &&

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

`! versus ~`

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


x

Bitwise Instructions

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

<op>	Rd	Rn	Rm
	destination	operand 1	Operand 2

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

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

Bitwise <op> description	C Syntax	***Arm <op> Syntax Op2: either register or constant value***	***Operation***
Bitwise **AND**	`a & b`	`and` R_d, R_n, Op2	R_d = R_n & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	`a & ~b`	`bic` R_d, R_n, Op2	R_d = R_n & ~Op2
Bitwise **OR**	`a | b`	`orr` R_d, R_n, Op2	R_d = R_n | Op2
Exclusive **OR**	`a ^ b`	`eor` R_d, R_n, Op2	R_d = R_n ^ Op2

x

The act (operation) of *Masking*

a =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
	<op>	<op>	<op>	<op>	<op>	<op>	<op>	<op>
b =	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
result =	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

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

x

Extracting (Isolate) a Field of Bits with a mask

extract top 8 bits of r2 into r1

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

```
and  r1, r2, r3
```

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

unchanged

forces to a 0

```
RSLT: r1 0xab 00 00 00
```

extract top 8 bits of r2 into r1

```
DATA: r2 0xab ab ab 77

and  r1, r2, 0xff000000

RSLT: r1 0xab 00 00 00

r1 = r2 & 0xff000000;  // in C
```

MOD %<power of 2>

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

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

```
and  r1, r2, r3
```

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

```
Example: Mod 16
DATA: r2 0xab ab ab 77
and
MASK: r3 0x00 00 00 0f (mod 16)
 forces to a 0          unchanged
RSLT: r1 0xab 00 00 07
```


x

Flipping bits: bit toggle
Used in PA8

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

- 1 **will flip the bit**
- 0 to let a **bit through**

```
eor  r1, r2, r3
```

- Observation: When applied twice, it returns the original value (symmetric encoding)
- With a mask of all 1's is a 1's compliment

```
Example: flip the lower 8-bits

    eor  r1, r2, 0xff
```

```
unsigned int r1, r2;

r1 = r2 ^ 0xff;
```

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

```
DATA: r2 0xab ab ab 77    77: 0111 0111

eor

MASK: r3 0x00 00 00 ff

  unchanged                inverts (flips)

RSLT: r1 0xab ab ab 88    88: 1000 1000
```

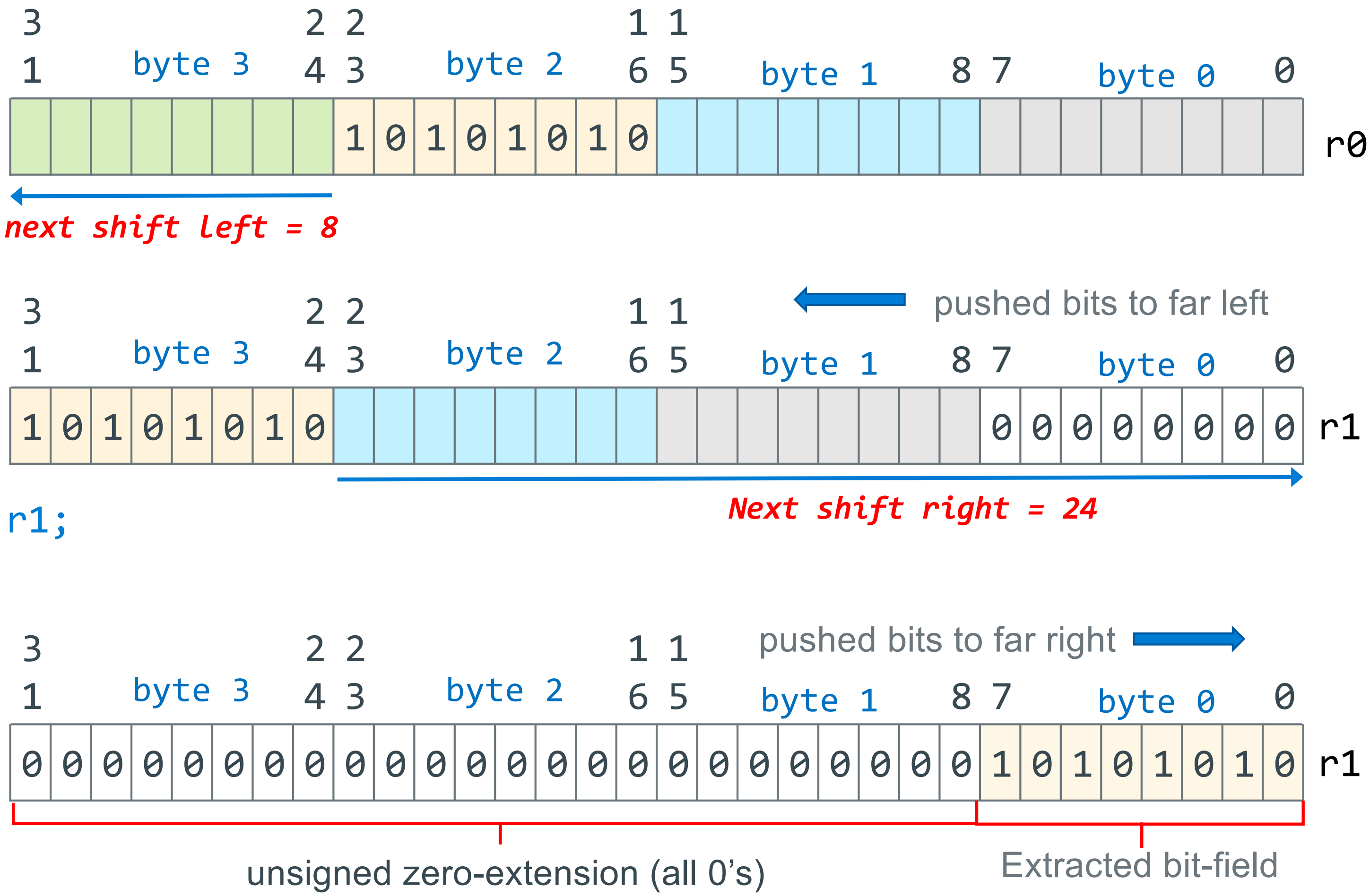
```
DATA: r1 0xab ab ab 88

eor

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

                       inverts (flips)

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


x

Extracting/Isolating Unsigned Bitfields

Hint: Useful for PA8

- Move byte 2 in r0 to byte 0 in r1




```
lsl   r1, r0, 8
```

```
unsigned int r0,r1;
r1 = r0 << 8;
```

```
lsr   r1, r1, 24
r1 = r1 >> 24;
```


x

Inserting Bitfields – Inserting Source Field into Destination Field

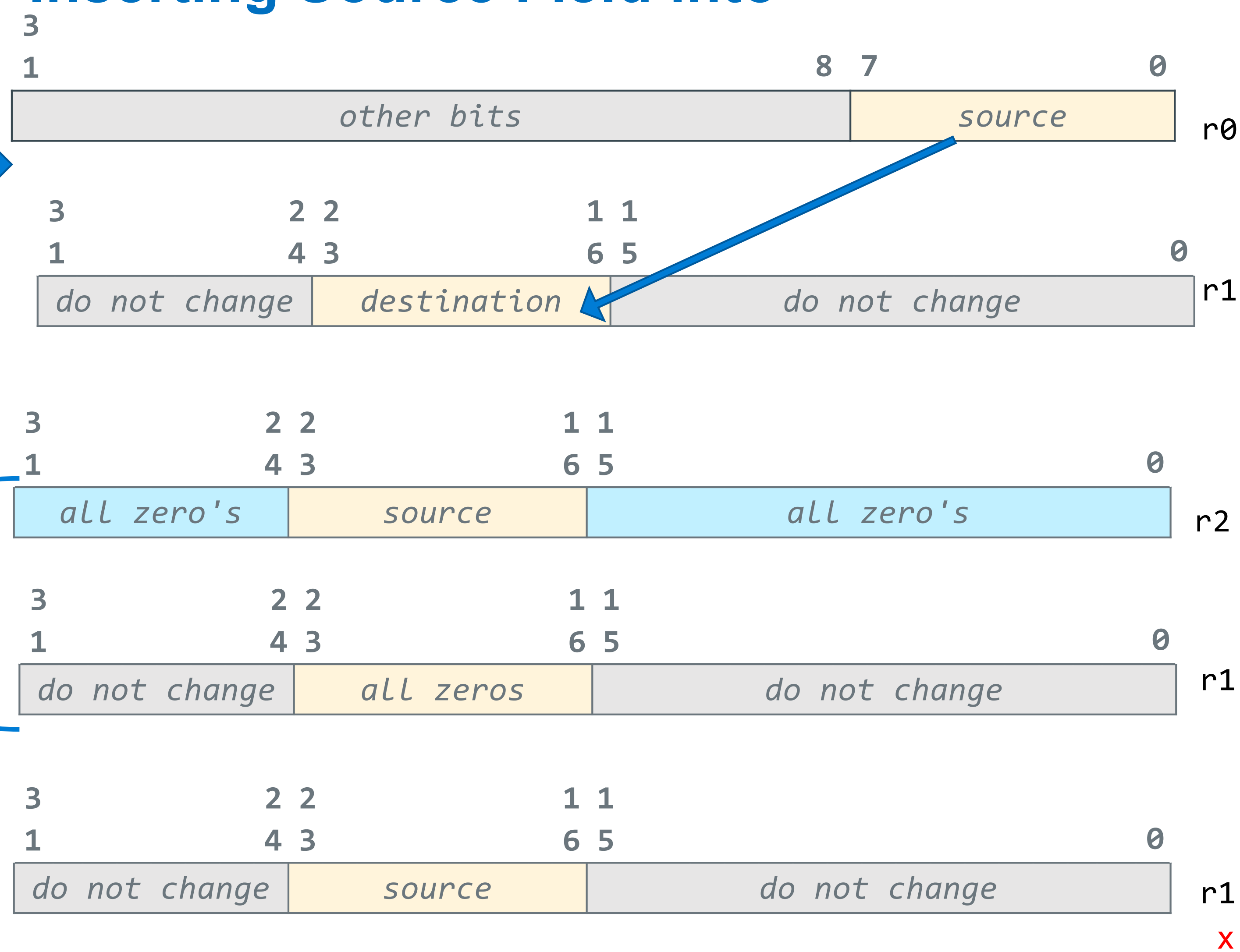
Task: Insert source into destination

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

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

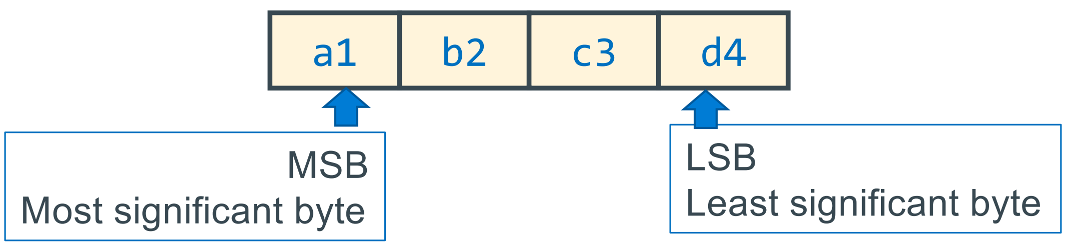
```
orr    r1, r1, r2
r1 =   r1 | r2;
```

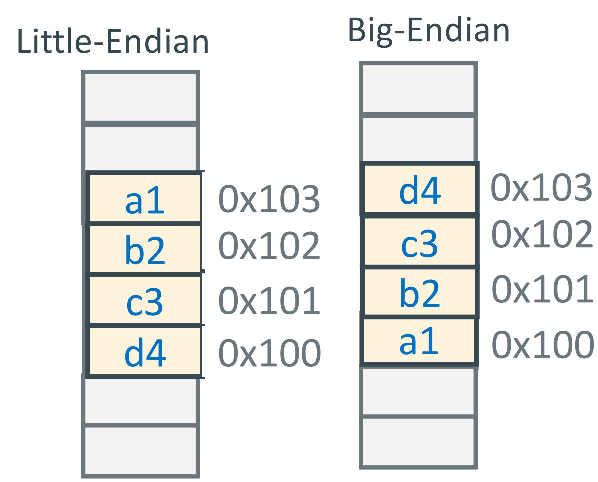
results in



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



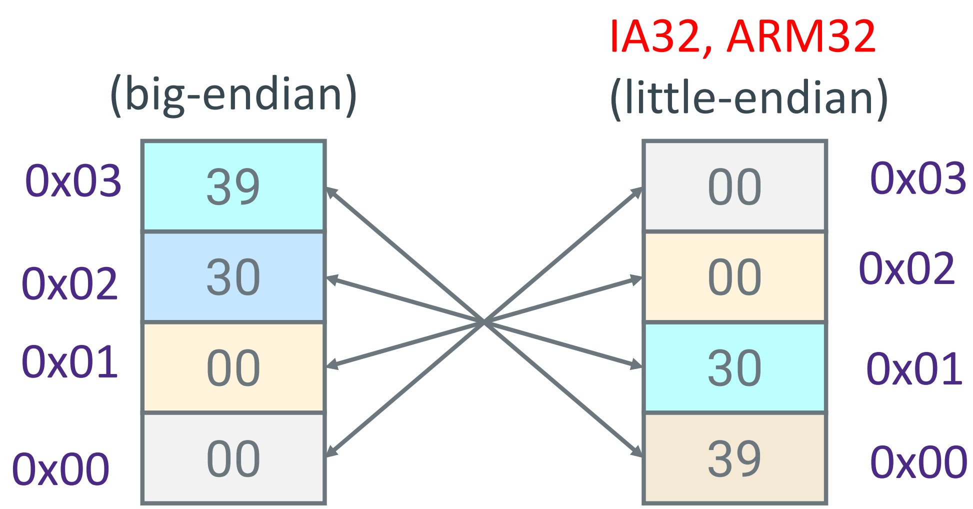


x

Byte Ordering Example

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

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



x

Loading and Storing: Variations List

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

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

x

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

Unsigned

Zero-Extend: Add leading 0's

example `ldrb`

memory

0b 1110 0001

r0				
r0	0x00	0x00	0x00	0xe1

Overwrite the upper three bytes with 0

Signed (2's complement)

Sign-Extend: Replicate sign bit

example `ldrsb`

memory

0b 1110 0001

r0	0xff	0xff	0xff	0xe1

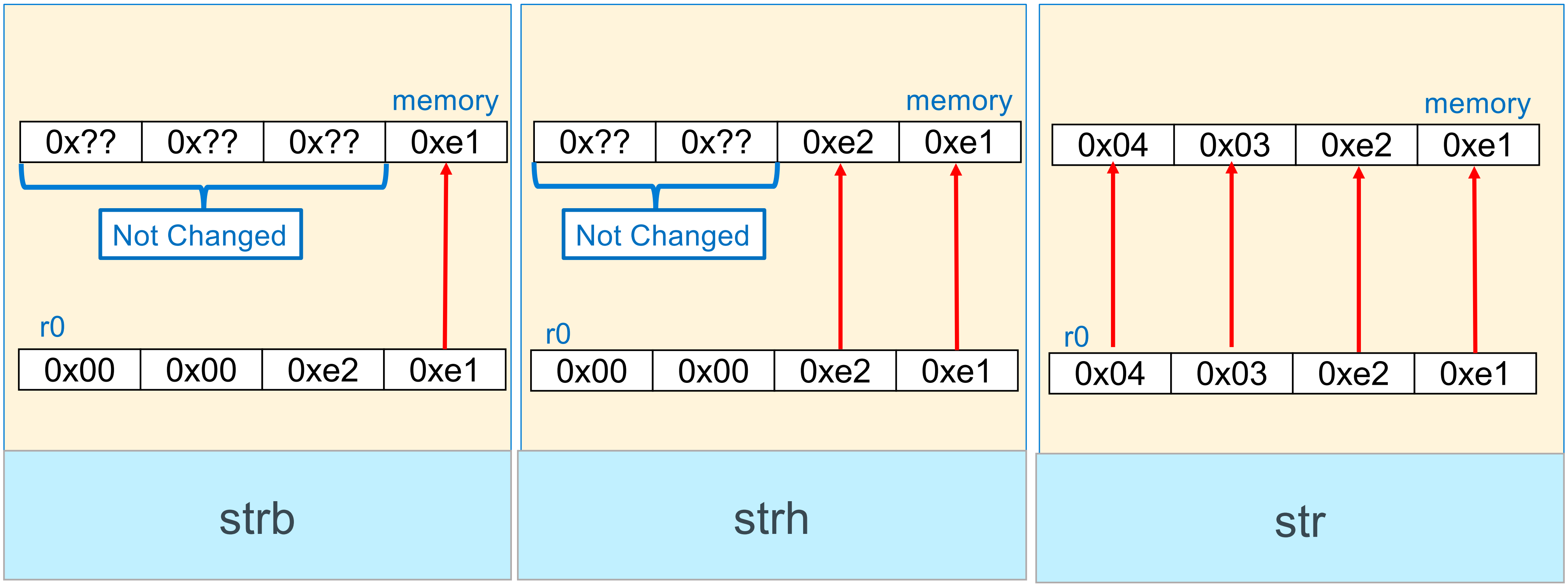
Overwrite the upper three bytes with 1

Instructions that zero-extend:
ldrb, ldrh

Instructions that sign-extend:
ldrsb, ldrsh

x

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

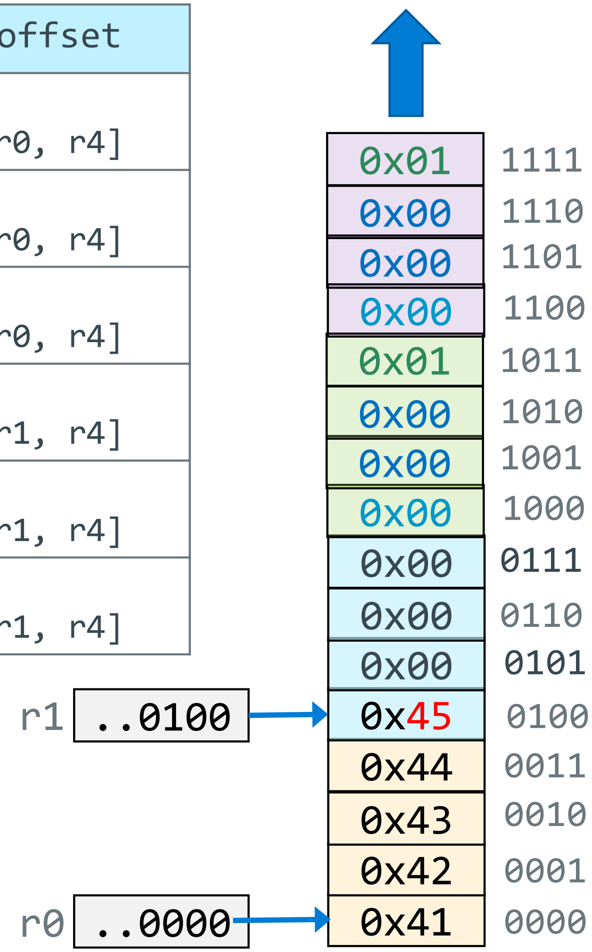


x

Array addressing with ldr/str

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

table rows are
independent instructions not a sequence

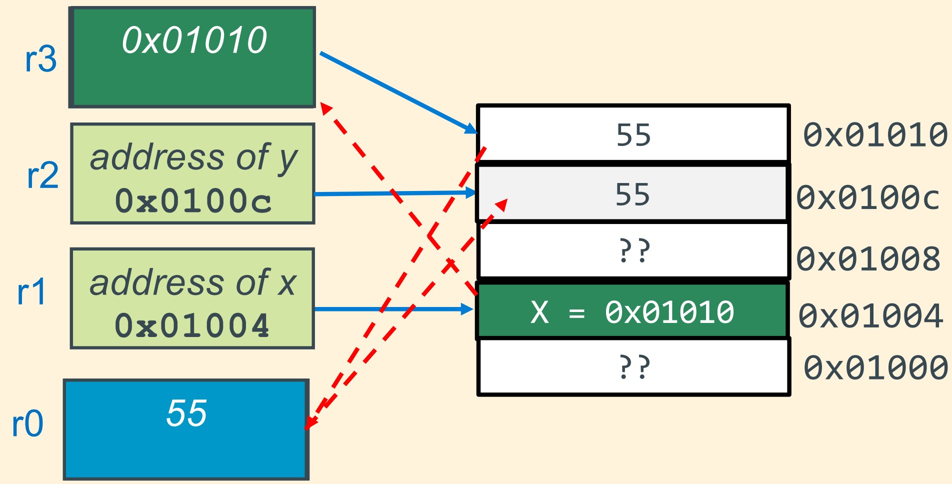


ldr/str practice - 2

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

Preview: Return Value and Passing Parameters to Functions

(Four parameters or less)

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

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

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

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

- Each **parameter and return value is limited to data that can fit in 4 bytes or less**
- You receive up to the first four parameters in these four registers
- You copy up to the first four parameters into these four registers before calling a function
- For parameter values using more than 4 bytes, a pointer to the parameter is passed (we will cover this later)
- **You MUST ALWAYS assume** that the called function will **alter the contents of all four registers: r0-r3**
  - **In terms of C runtime support, these registers contain the copies given to the called function**
  - **C allows the copies to be changed in any way by the called function**

X

Using the literal table to fix:

Error: invalid constant (3ff) after fixup

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

mov	Rd	rot4	imm8

fails → `mov r0, 1023`

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

replacement → `ldr r0, =1023`

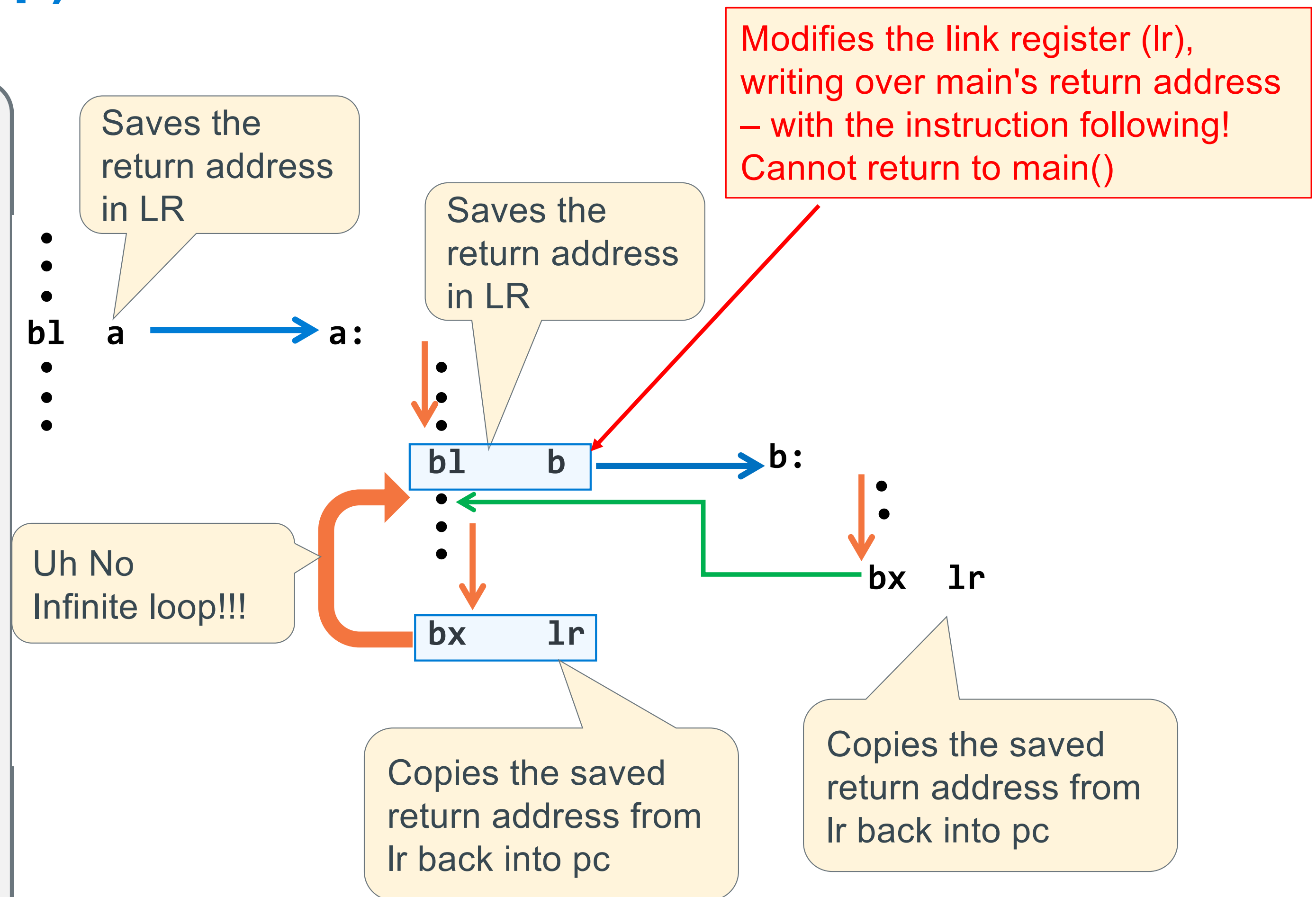
- Answer: use **`ldr`** instruction with the constant as an operand: **`=constant`**
- Assembler creates a **literal table entry** with the **constant**

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


x

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

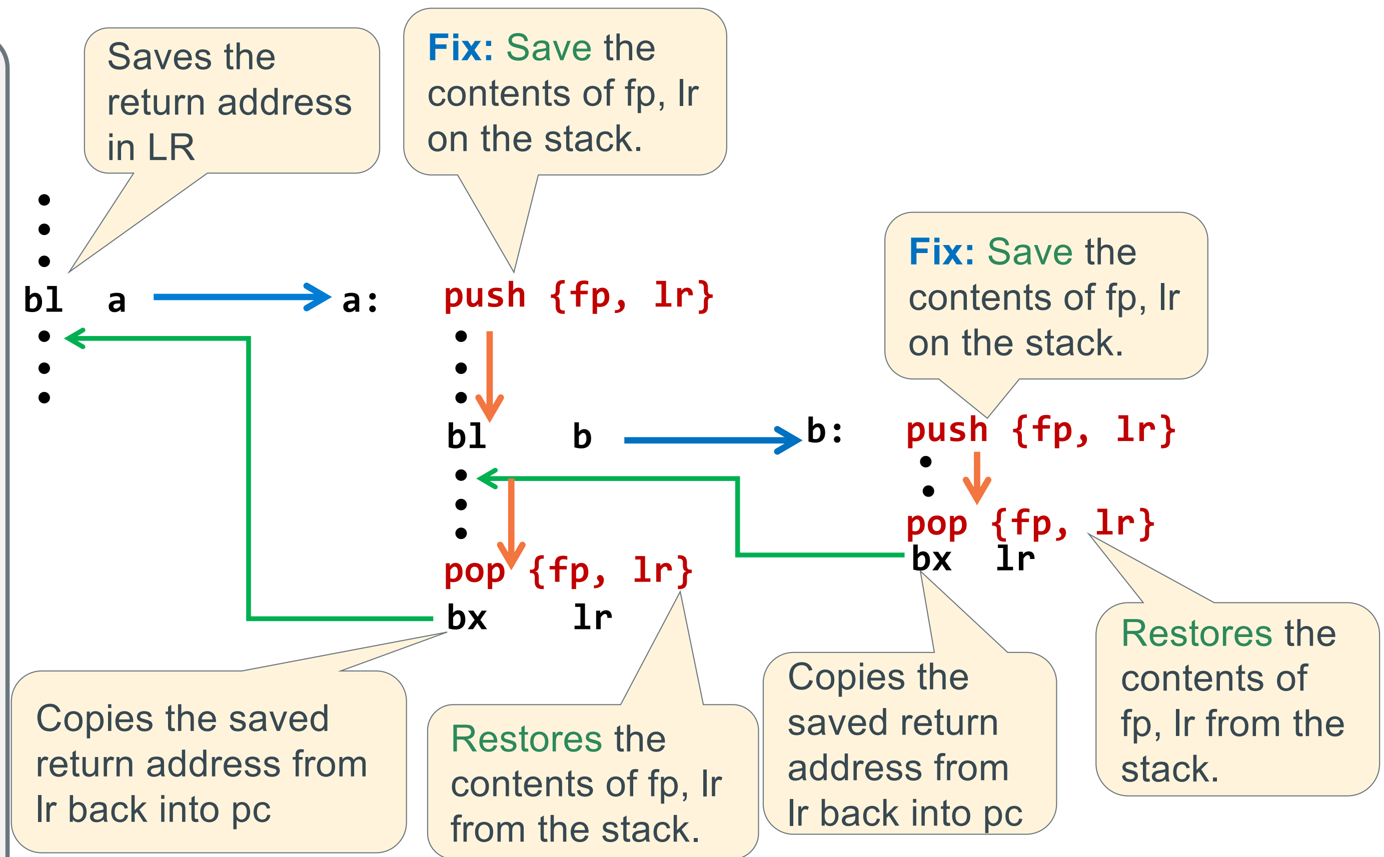
```
int main(void)
{
     a();
     /* other code */
     return EXIT_SUCCESS;
}

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

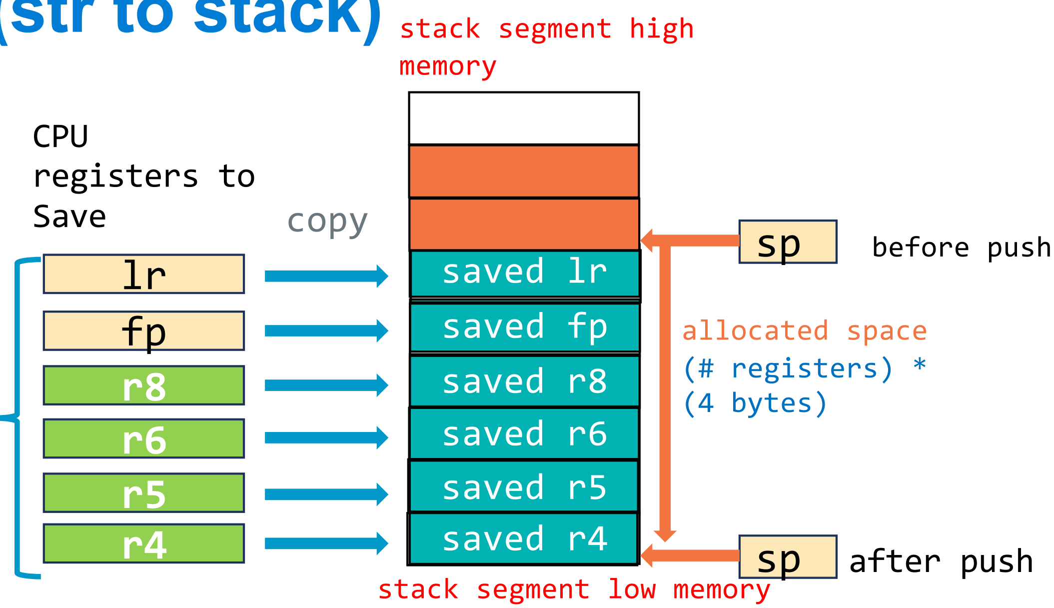
x

push: Multiple Register Save (str to stack)

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

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

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



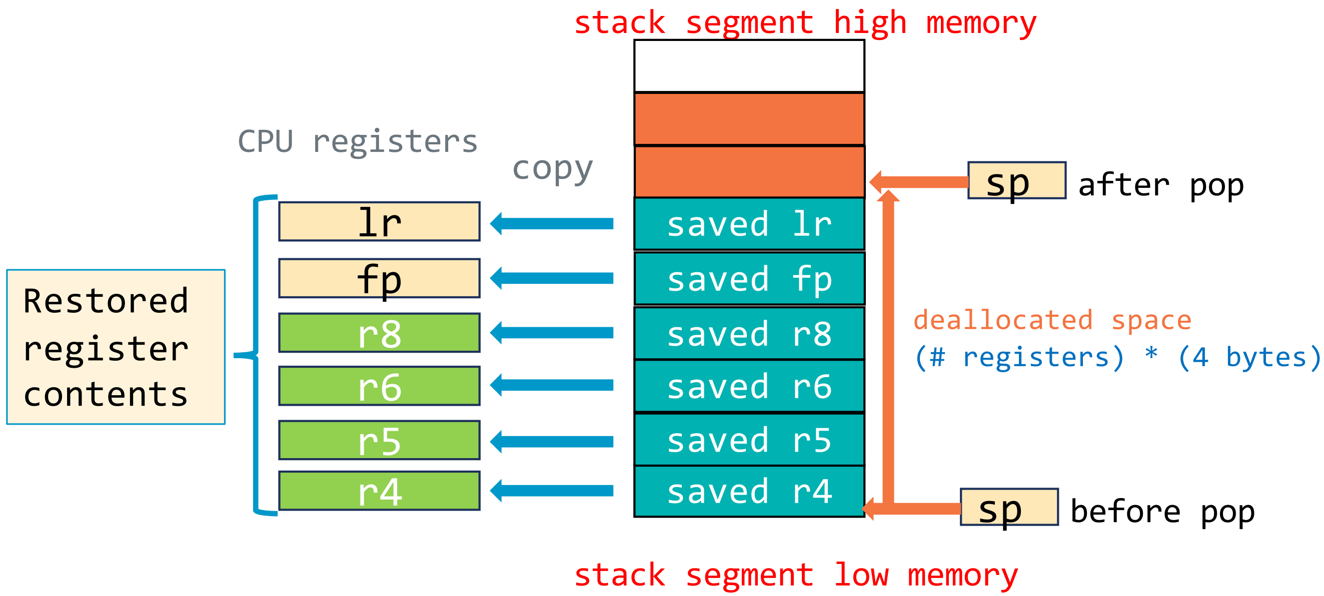
- **`push`** copies the contents of the **`{reg list}`** to stack segment memory
- **`push`** **Also** <u>subtracts</u> (# of registers saved) * (4 bytes) from the **`sp`** to ***allocate*** space on the stack
  - sp = sp – (# registers_saved * 4)
- **this must always be true: sp % 8 == 0**

x

pop: Multiple Register Restore (ldr from stack)

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

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



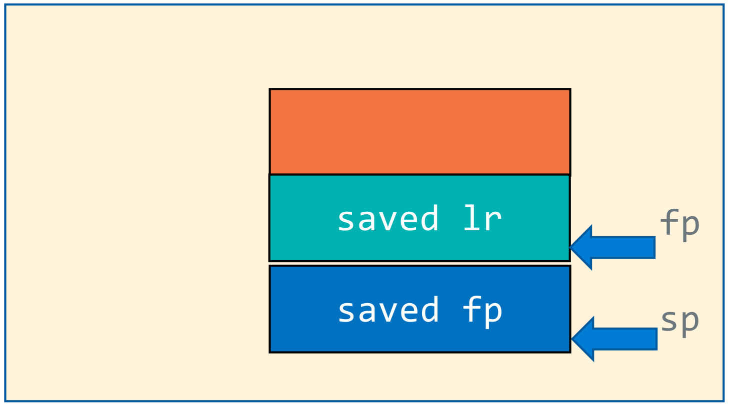
- `pop` copies the contents of stack segment memory to the `{reg list}`
- `pop` **adds:** (# of registers restored) * (4 bytes) to `sp` to ***deallocate*** space on the stack
  - sp = sp + (# registers restored * 4)
- **`Remember:`** `{reg list}` must be the same in both the `push` and the corresponding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

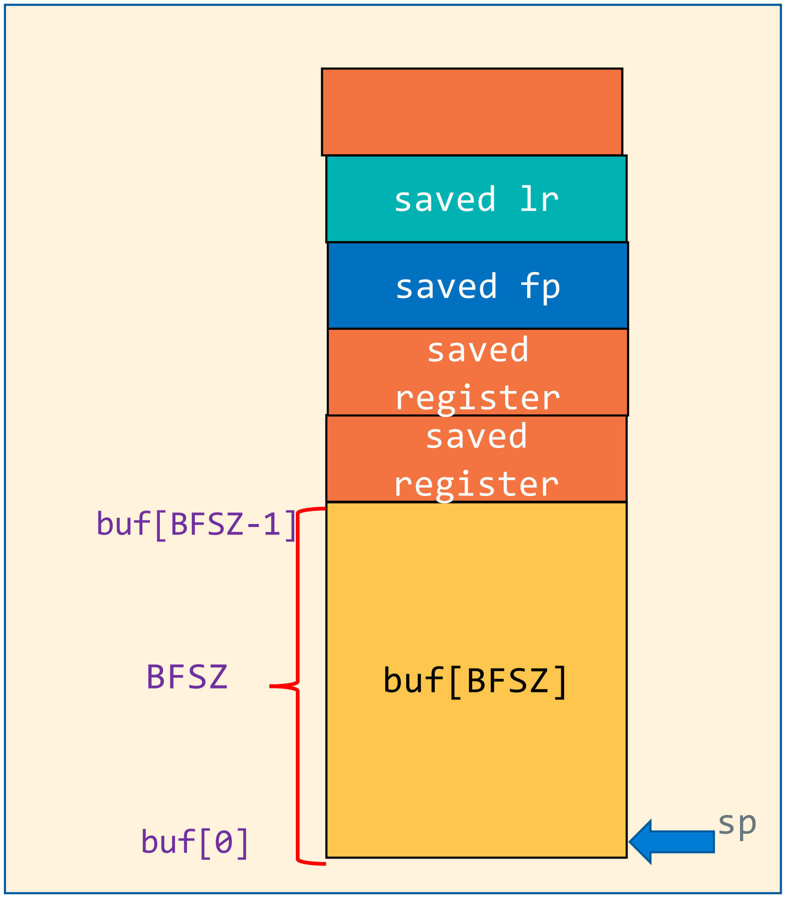
x

Local Variables are Part of Each Stack Frame

- Local variables are on the stack below the lowest numbered saved (pushed) register

```
#define BFSZ 4
int main(void)
{
  char buf[BFSZ]; // BFSZ bytes
...
```



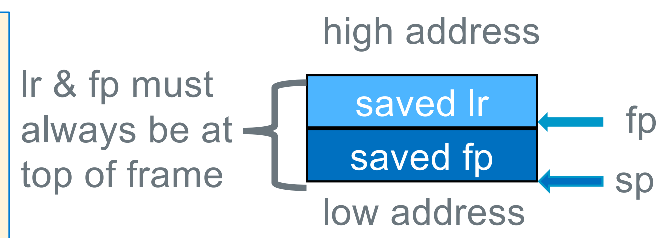


x

Stack Frame (Arm Arch32 Procedure Call Standards)

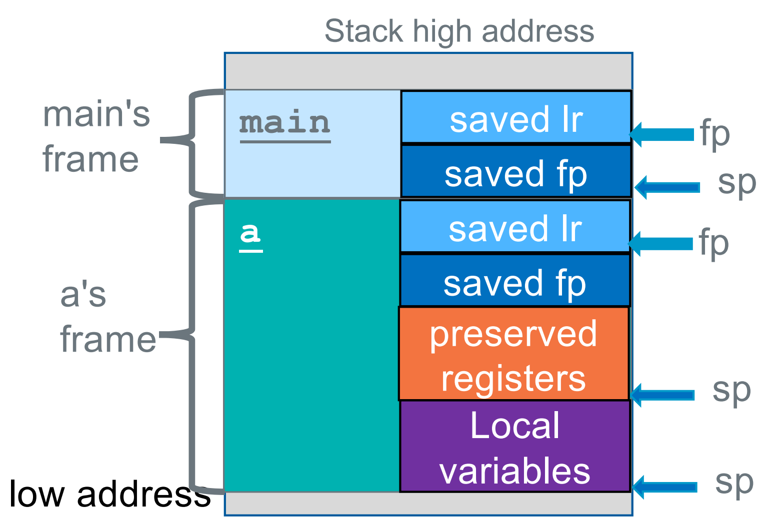
Stack Frame Requirements

- **Minimal frame: at function entry** `push {fp, lr}`
- The top two entries in a stack frame are always (1) saved lr, (2) saved fp
- `sp` points at top element in the stack (lowest byte address)
- `fp` points at the `lr` copy stored in the current stack frame
- **Stack frames MUST ALWAYS BE aligned to 8-byte addresses**
  - So, this must always be true: sp % 8 == 0



minimal frame above
Always save at least fp and lr and set fp at saved lr

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



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

deallocate stack space
SP = SP + "space"
shrinks "up"

Note slide has builds

x

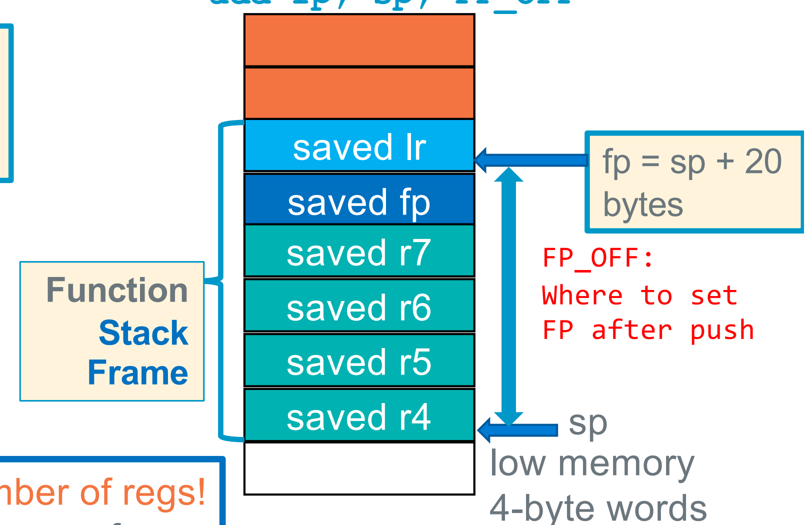
FP_OFF: Distance from FP to SP
Used to set FP at push and SP before pop

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

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

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

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



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

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

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

IMPORTANT: FP_OFF has **two** uses:
1. Where to set fp after prologue push (remember sp position)
2. Restore sp (deallocate locals) right before epilogue pop

Function Prologue and Epilogue: Minimum Stack Frame

- Each function has only one Prologue at the top of the function body and only one Epilogue at the bottom of the function body
- When you want to exit the function, set the return value in r0, and then branch (or fall through) to the epilogue
- Function entry (Function **Prologue**):
  1. save preserved registers
  2. set the fp to point at saved lr
  3. allocate space for locals (subtracts from sp)
- Function return (Function **Epilogue**):
  1. deallocate space for locals (adds to sp)
  2. restores preserved registers
  3. return to caller

smallest frame has:
4 bytes between sp & fp

saved lr	← fp
saved fp	← sp

low memory

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

one:
    push    {fp, lr}
    add     fp, sp, FP_OFF
    // add space for locals
    // your code

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

    bx      lr  // func return

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

Position the fp → add

Position the sp → sub

Function Header
Assembly directives

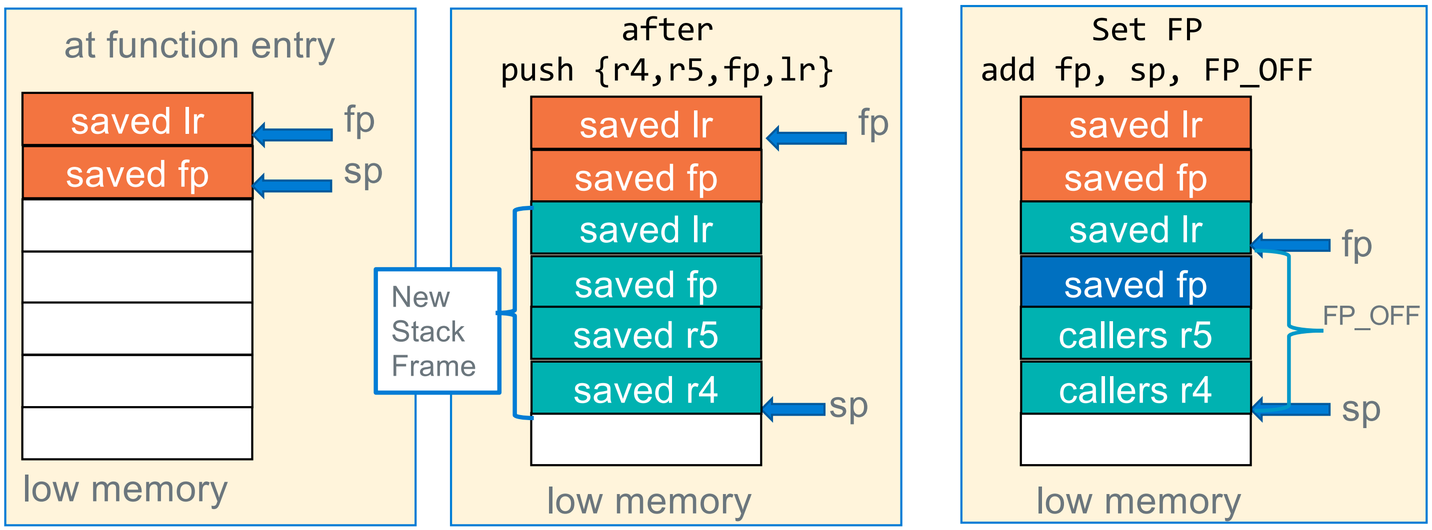
Function Prologue
always at top of function

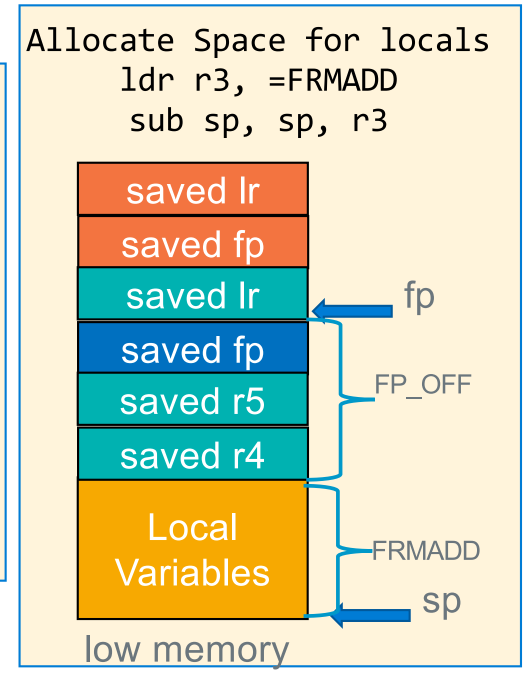
Function Epilogue
always at bottom of function

Function Footer
Assembly directive

x

Function Prologue: Allocating the Stack Frame





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
Do not push r12 or r13

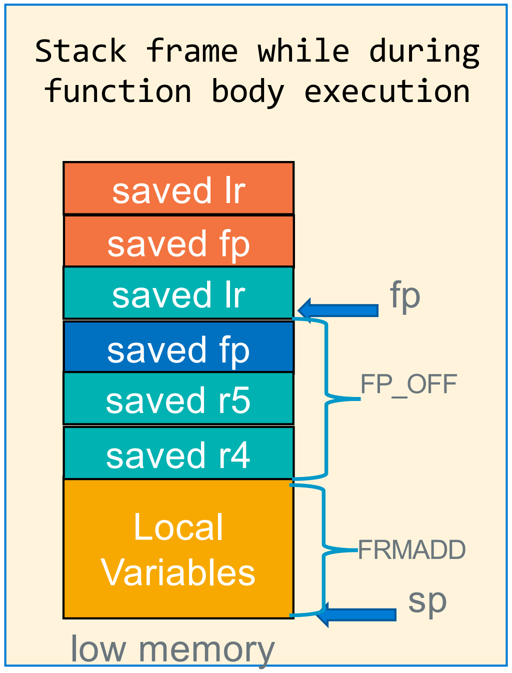
Function moves the fp to point at the saved lr as required by the Aarch32 spec

Allocate Space for Local Variables

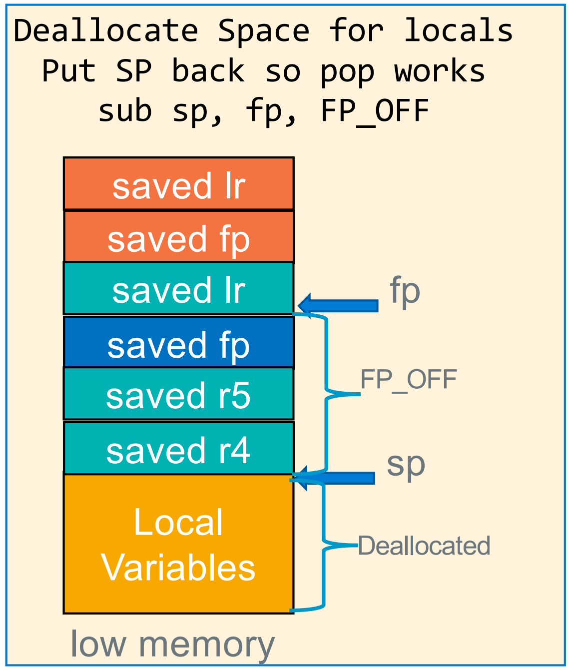
Part of function prologue

x

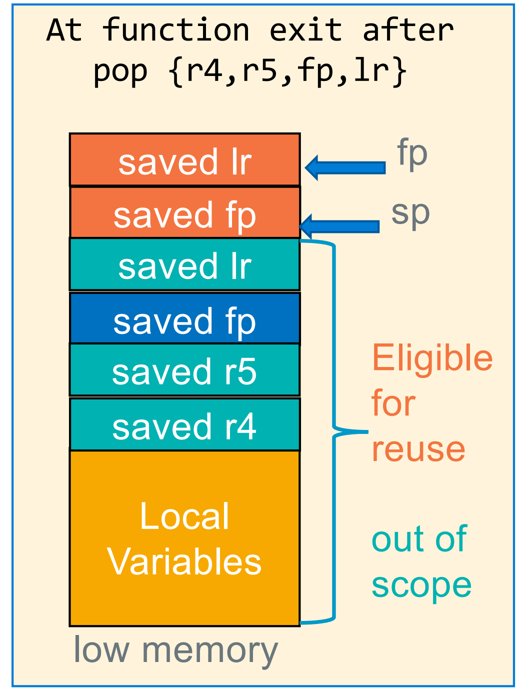
Function Epilogue: Deallocating the Stack Frame



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



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



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 epilogue

x

Review Return Value and Passing Parameters to Functions

(Four parameters or less)

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

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

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

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

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

X

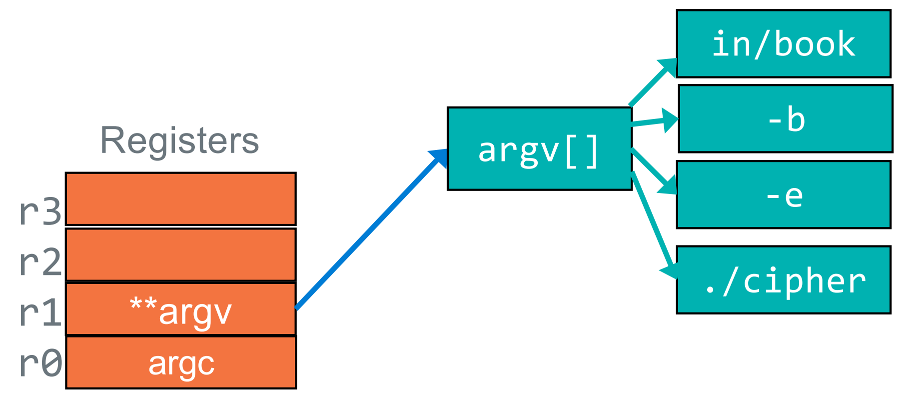
Accessing argv from Assembly (stderr version)

```
    .extern printf
    .extern stderr
    .section .rodata
.Lstr:  .string "argv[%d] = %s\n"
    .text
    .global main    // main(r0=argc, r1=argv)
    .type   main, %function
    .equ    FP_OFF,     20
main:
    push    {r4-r7, fp, lr}
    add     fp, sp, FP_OFF
    ldr     r4, =stderr      // get the address of stderr
    ldr     r4, [r4]         // get the contents of stderr
    ldr     r5, =.Lstr       // get the address of .Lstr
    mov     r6, 0            // set indx = 0;
    mov     r7, r1           // save argv
.Lloop:
    // fprintf(stderr, "argv[%d] = %s\n", indx, argv[indx])
    ldr     r3, [r7]        // argv[indx]
    cmp     r3, 0           // check argv[indx]==NULL
    beq     .Ldone          // if so done
    mov     r2, r6          // indx
    mov     r1, r5          // "argv[%d] = %s\n"
    mov     r0, r4          // stderr
    bl      fprintf
    add     r6, r6, 1       // indx++
    add     r7, r7, 4       // argv++
    b       .Lloop
.Ldone:
    mov     r0, 0
    sub     sp, fp, FP_OFF
    pop     {r4-r7, fp, lr}
    bx      lr
```

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

Function Epilogue
always at bottom of function Branch to this to exit the function
restores regs including the sp

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



Local Variables on the Stack

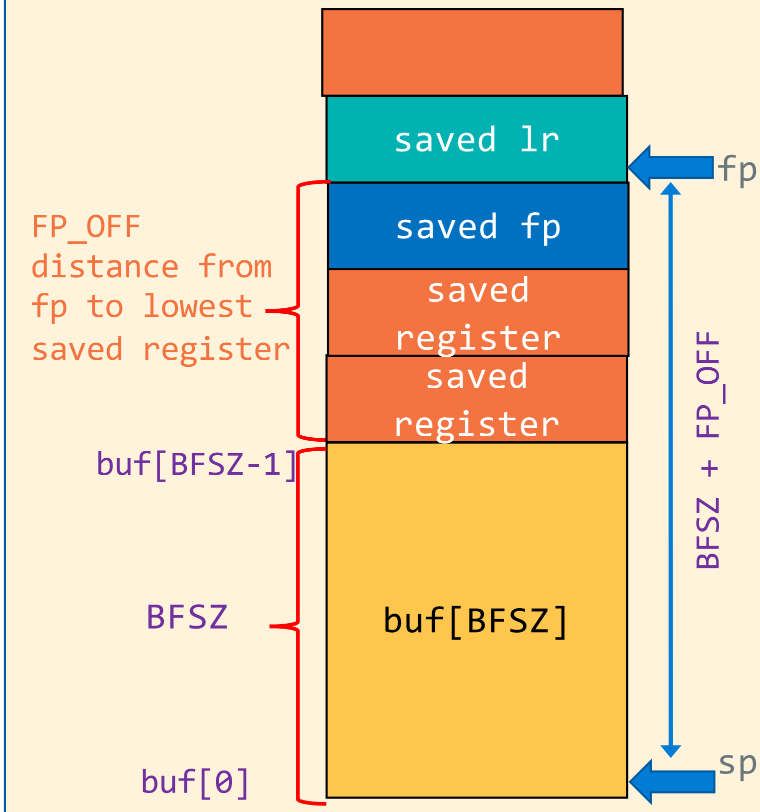
- Local variables are on the stack below the lowest numbered saved register
- frame pointer is used as a **pointer** to stack variables
- fp is the base register in ldr and str instructions
- Example load buf[0] into r4

```
#define BFSZ 4
int main(void)
{
  char buf[BFSZ]; // BFSZ bytes
...
```

- `FP_OFF = 12, BUFSZ = 4`
- `Distance from FP is buf[0] is 12 + 4 = 16`

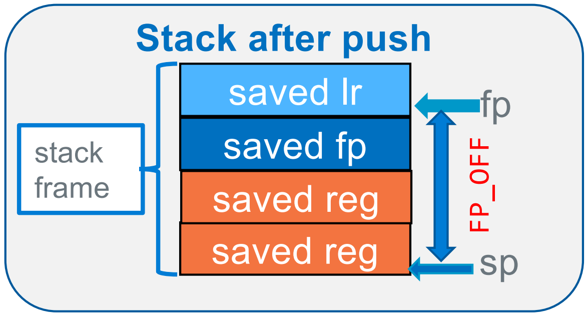
```
ldrb r4, [fp, -16]
```

1. Calculate how much additional space is needed by all the local variables
2. **After the register save push, Subtract from the sp** the size of the variable in bytes (+ padding - later slides)

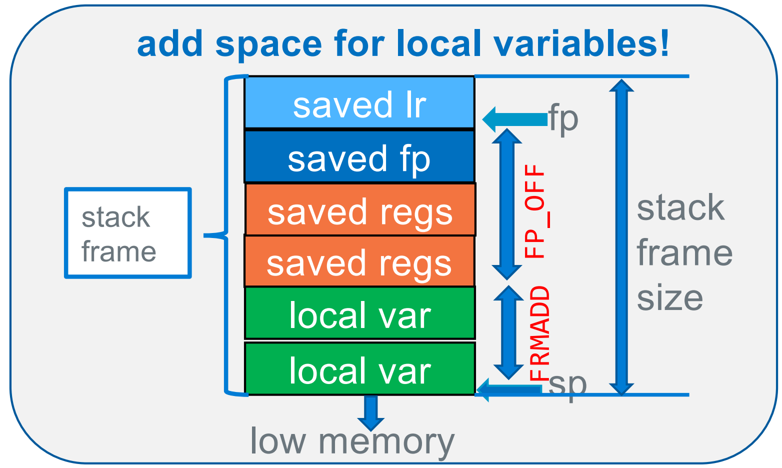


x

Function prologue with local variables




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

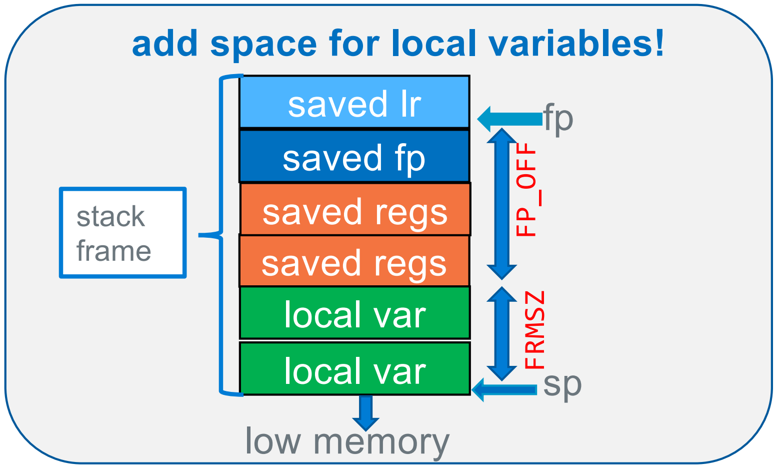


- move the sp to allocate space on the stack for local variables and outgoing parameters (later)

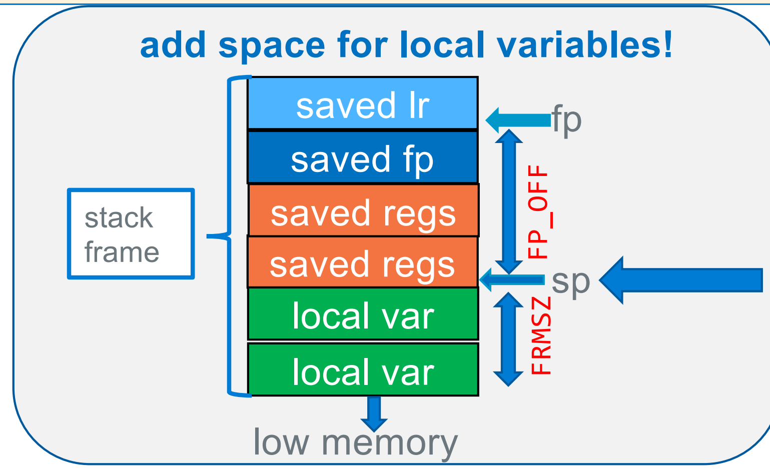
```
    .equ    FRMADD, 8
    push    {fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD // frames may be large
    sub     sp, sp, r3
// your code
```


x

Function epilogue with local variables



- For pop to restore the registers correctly:
  - sp must point at the last saved preserved register put on the stack by the save register operation: the push



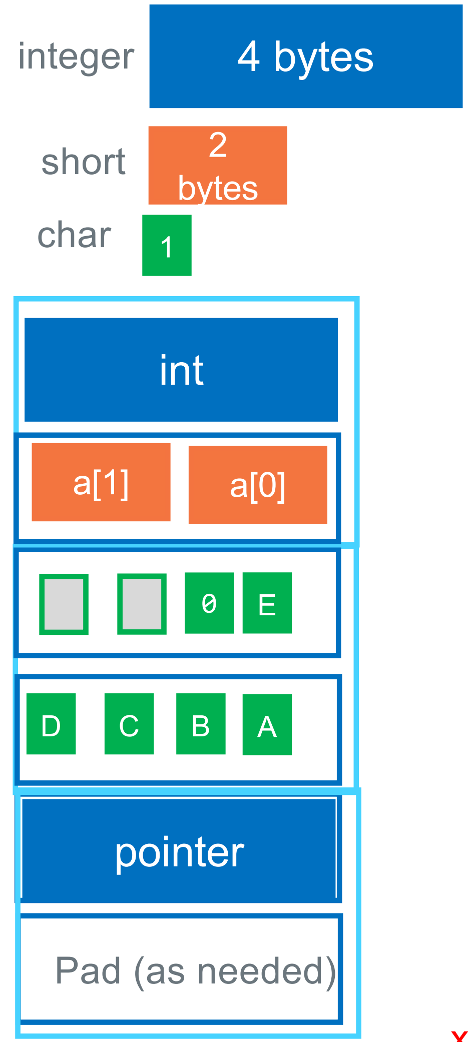

```
.equu    FRMADD, 8
push     {fp, lr}
add      fp, sp, FP_OFF
ldr      r3, =FRMADD
sub      sp, sp, r3
  // your code

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

bx       lr  // func return
```

- Return the **sp** (using the **fp**) to the same address it had after the push operation
  sub sp, fp, FP_OFF
- this works no matter how much space was allocated in the prologue

x

Stack Frame Design – Local Variables

- 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 (0 or 4 bytes) when necessary is added at the high address end of a variables allocated space, based on the variable's alignment and the requirements of variable below it on the stack
- 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

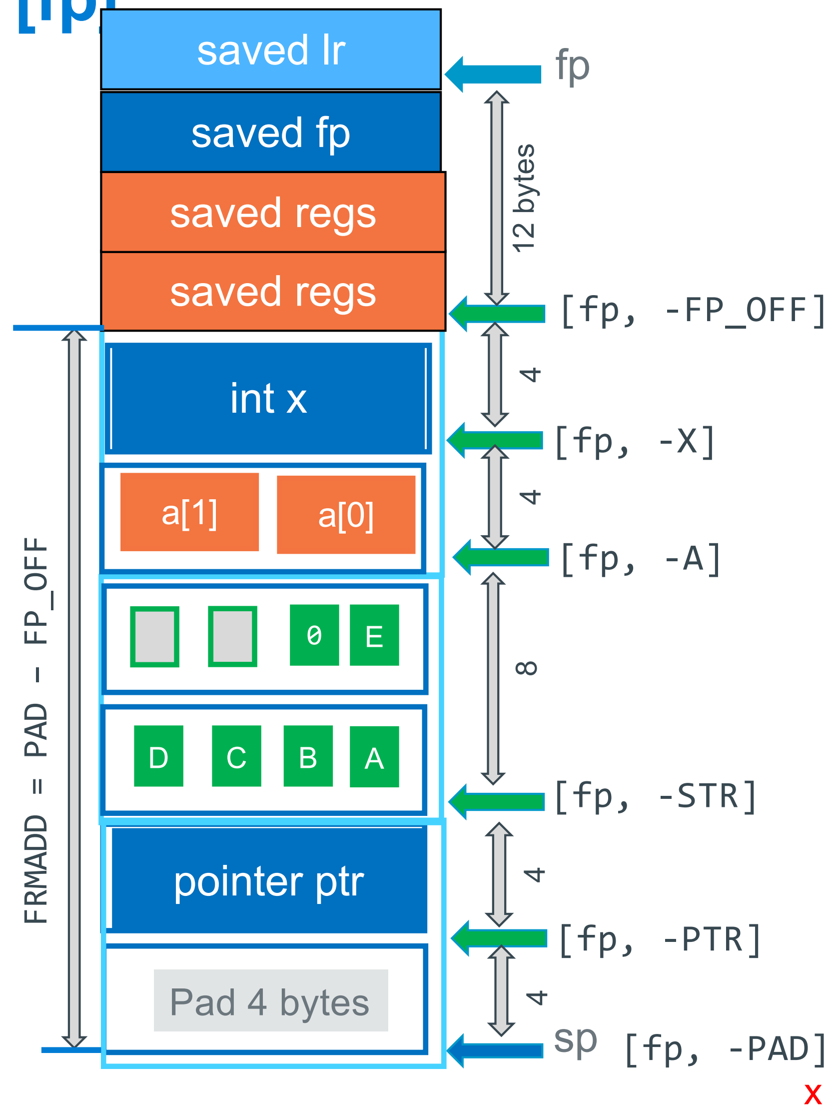
Step 2 Generate Distance offsets from [fp]

- Use the assembler to calculate the **distance** from the address contained in fp `[fp, -offset]`

```
.equ FP_OFF, 12
.equ X, 4+FP_OFF // X = 16
.equ A, 4+X      // A = 20
```

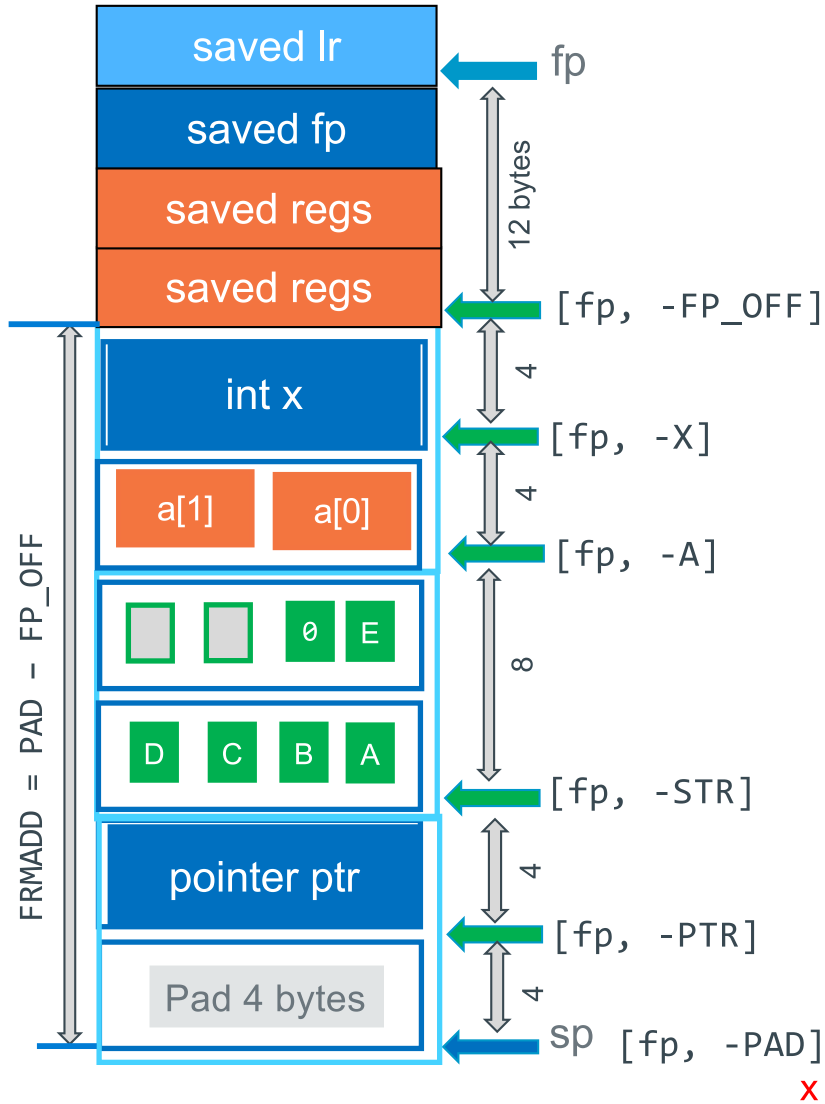
- Assign label names for each local variable
  - Each name is .equ to be the offset from fp

Variable name	Size	Name	expression size + prev	Distance from fp
Pushed regs-1	12	FP_OFF		12
int x	4	X	4 + FP_OFF	16
short a[]	4	A	4 + X	20
char str[]	8	STR	8 + A	28
char *ptr	4	PTR	4 + STR	32
PAD Added	4	PAD	4 + PTR	36
FRMADD		FRMADD	PAD-FP_OFF	24



Step 3 Allocate Space in the Prologue

```
    .global func
    .type   func, %function
    .equ       FP_OFF,        12
    .equ       X,             4 + FP_OFF
    .equ       A,             4 + X
    .equ       STR,           8 + A
    .equ       PTR,           4 + STR
    .equ       PAD,           4 + PTR
    .equ       FRMADD         PAD - FP_OFF
func:
    push    {r4, r5, fp, lr}
    add     fp, sp, FP_OFF
    ldr     r3, =FRMADD //frames can be large
    sub     sp, sp, r3 // add space for locals
    // rest of function code
    // no change to epilogue
    sub     sp, fp, FP_OFF  // deallocate locals
    pop     {r4, r5, fp, lr}
    bx      lr
    .size   func, (. - func)
```



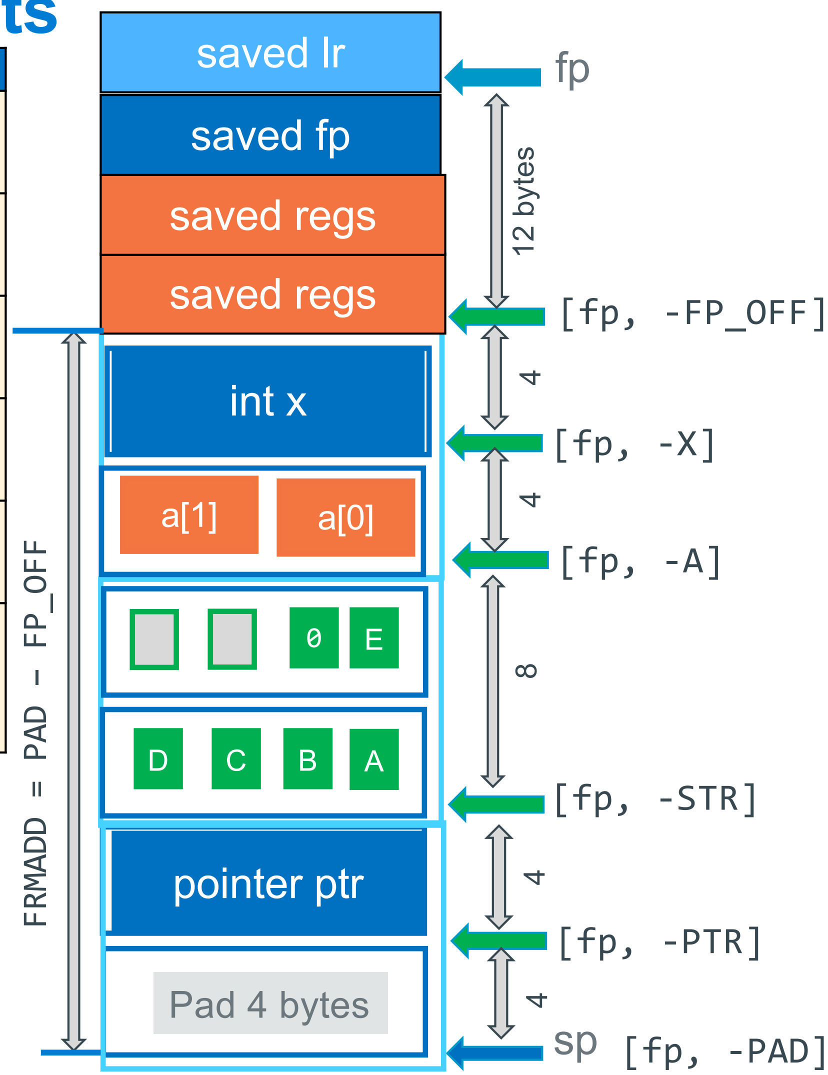
Accessing Stack using distance offsets

var	stack variable address into r0	stack variable contents into r0
x	`ldr r0, =X` `sub r0, fp, r0`	`ldr r0, =X` `ldr r0, [fp, -r0]`
a[0]	`ldr r0, =A` `sub r0, fp, r0`	`ldr r0, =A` `ldrsh r0, [fp, -r0]`
a[1]	`ldr r0, =A - 2` `sub r0, fp, r0`	`ldr r0, =A - 2` `ldrsh r0, [fp, -r0]`
str[1]	`ldr r0, =STR - 1` `sub r0, fp, r0`	`ldr r0, =STR - 1` `ldrb r0, [fp, -r0]`
ptr	`ldr r0, =PTR` `sub r0, fp, r0`	`ldr r0, =PTR` `ldr r0, [fp, -r0]`

*ptr	`ldr r0, =PTR`
	`sub r0, fp, r0`
	`ldr r0, [r0]`
		`ldr r0, =PTR`
		`ldr r0, [fp, -r0]`
		`ldr r0, [r0]`

var	write contents of r0 to stack variable
ptr	`ldr r1, =PTR` `str r0, [fp, -r1]`

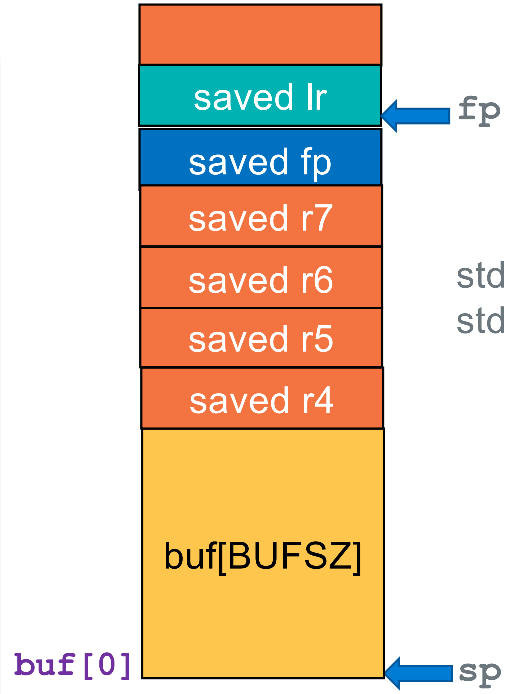
*ptr	`ldr r1, =PTR`
	`ldr r1, [fp, -r1]`
	`str r0, [r1]`

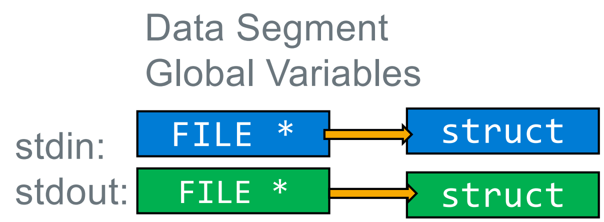


x

Passing Pointers to Stack Variables

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define BUFSZ 4096
// copies input to output
int
main(void) {
    char buf[BUFSZ];
    size_t cnt;   // assign to a register only

    // read from stdin, up to BUFSZ bytes
    // and store them in buf
    // Number of bytes read is in cnt
    while ((cnt = fread(buf, 1, BUFSZ, stdin)) > 0) {
         // write cnt bytes from buf to stdout
        if (fwrite(buf, 1, cnt, stdout) != cnt) {
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```



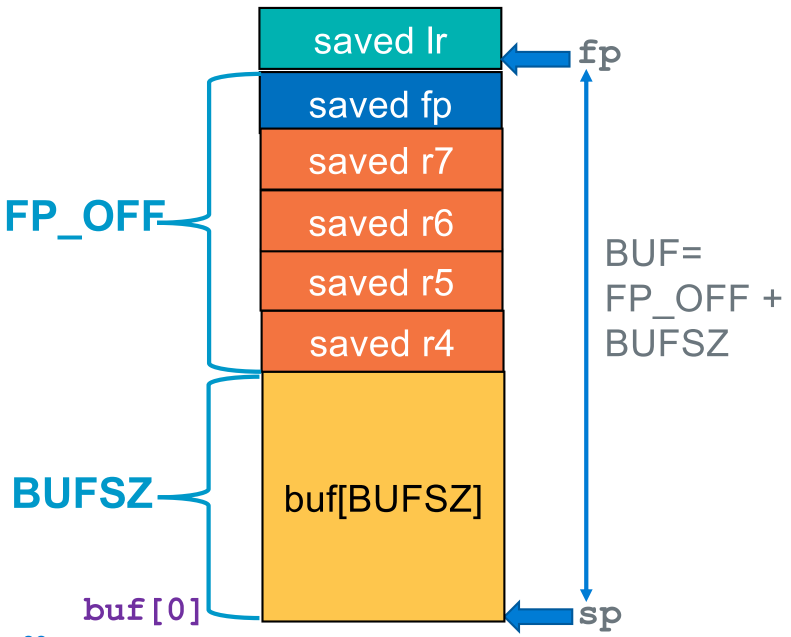



```
    .text
    .global main
    .type   main, %function    // stack frame below
    .equ    BUFSZ,      4096
    .equ    FP_OFF,     20          // fp offset in main stack frame
    .equ    BUF,        BUFSZ+FP_OFF// buffer
    .equ    PAD,        0+BUF       // Stack frame PAD
    .equ    FRMADD,     PAD-FP_OFF  // space for locals+passed args
```

Reading and Writing bytes using C library routines fread() and fwrite()

```
    .text
    .global main
    .type   main, %function    // stack frame below, distances from fp
    .equ    BUFSZ,      4096
    .equ    FP_OFF,     20          // fp offset in main stack frame
    .equ    BUF,        BUFSZ+FP_OFF// buffer
    .equ    PAD,        0+BUF       // Stack frame PAD
    .equ    FRMADD,     PAD-FP_OFF  // space for locals+passed args
```

```
    // save values in preserved registers
    ldr     r4, =BUF      // distance from fp
    sub     r4, fp, r4    // pointer to buffer
    ldr     r5, =stdin    // standard input global
    ldr     r5, [r5]
    ldr     r6, =stdout   // standard output global
    ldr     r6, [r6]
```




```
    // fread(buffer, element_size, number of elements, FILE *)
    // fread(r0=buf, r1=1, r2=BUFSZ, r3=stdin)
    mov     r0, r4              // buf
    mov     r1, 1               // bytes
    mov     r2, BUFSZ           // cnt (or ldr r2, =BUFSZ)
    mov     r3, r5              // stdin
    bl      fread
    cmp     r0, 0               // check return value from fread
```

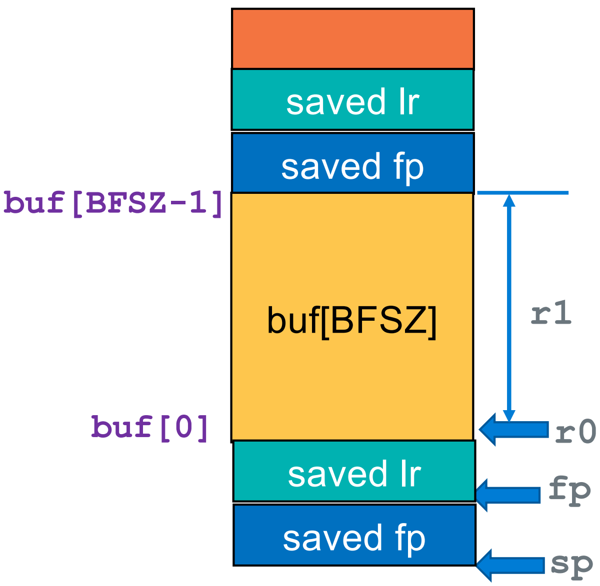
```
    // fwrite(buffer, element_size, number of elements, FILE *)
    // fwrite(r0=buf, r1=1, r2=cnt, r3=stdout)
    mov     r0, r4              // buf
    mov     r1, 1               // bytes
    mov     r2, r7              // cnt
    mov     r3, r6              // stdout
    bl      fwrite
    cmp     r0, r7              // check return value from fwrite
```

Writing Function: Receiving a Pointer 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
```

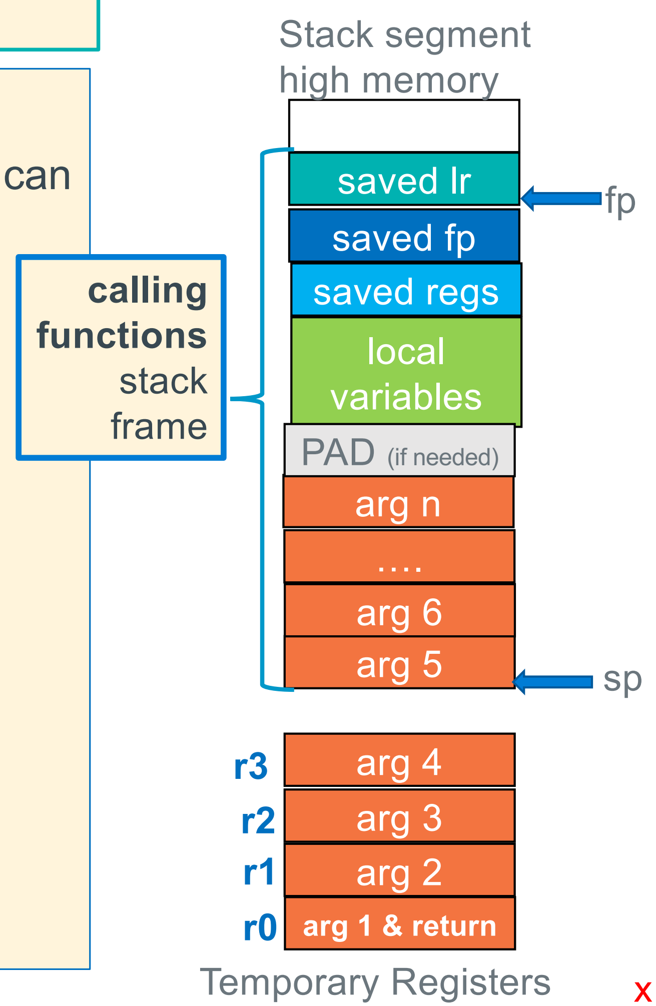


x

Passing More Than Four Arguments – At the point of Call

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

- **Args > 4 are in the caller's stack frame at SP (argv5), an up**
- 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
- Calling function prior to making the call
  1. Evaluate first four args: place resulting values in r0-r3
  2. Store Arg 5 and greater parameter values on the stack
- **One arg value per slot!** – NO arrays across multiple slots
  - chars, shorts and ints are directly stored
  - Structs (not always), and arrays are passed via a pointer
  - **Pointers** passed as output parameters usually contain an address ***that points at*** the stack, BSS, data, or heap



x

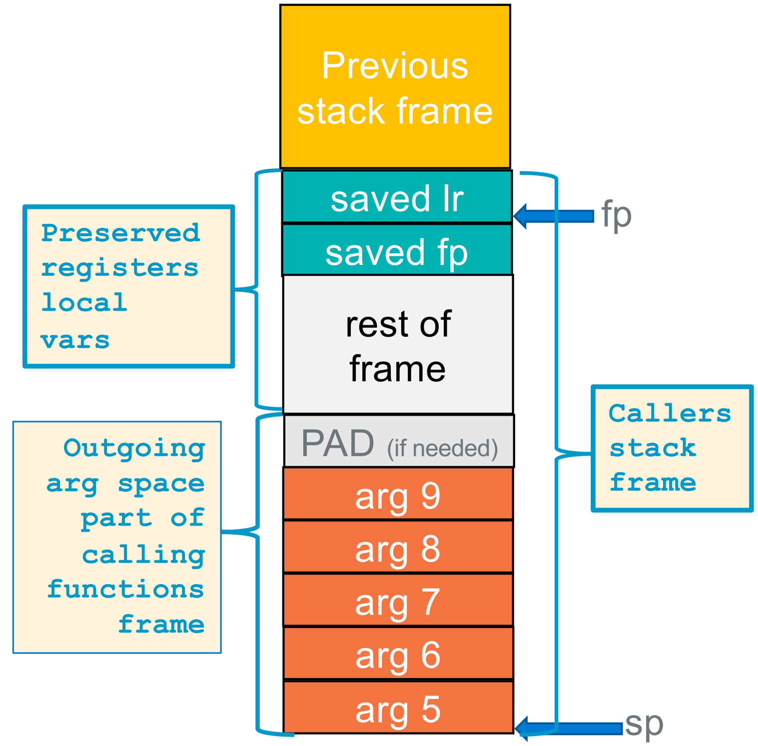
Calling Function: Allocating Stack Parameter Space

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**

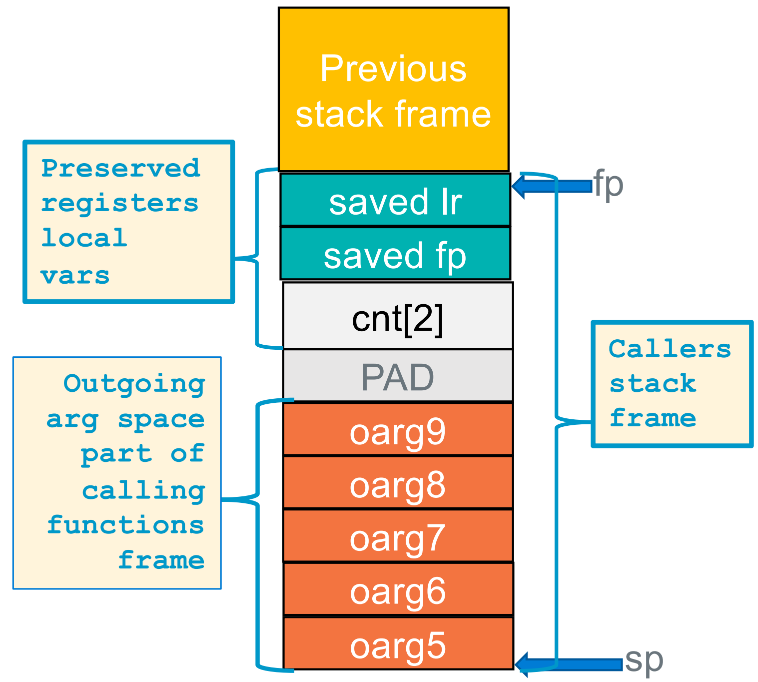
x

Calling Function: Pass ARGS 5 and higher

```
.equ    FP_OFF,4
.equ    CNT,          8 + FP_OFF        // int cnt[2];
.equ    PAD,          4 + CNT        // added as needed
.equ    OARG9,        4 + PAD
.equ    OARG8,        4 + OARG9
.equ    OARG7,        4 + OARG8
.equ    OARG6,        4 + OARG7
.equ    OARG5,        4 + OARG6
.equ    FRMADD        OARG5 – FP_OFF
```

var	*write contents*
OARG5 = r1	`ldr     r0, =OARG5          //distance` `str     r1, [fp, -r0]`

OARG6 = &cnt	`ldr     r2, =CNT            //distance`
	`sub     r2, fp, r2          // &cnt`
	
	`ldr     r0, =OARG6          //distance`
	`str     r2, [fp, -r0]`



Rules: At point of call

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

X

<u>Called Function</u>: Retrieving Args From the Stack

- At function start and before the push{} the sp is at an 8-byte boundary
- **Args are in the <u>caller's stack frame</u> and arg 5 always starts at fp+4**
  - Additional args are higher up the stack, with one “slot” every 4-bytes
- This "algorithm" for finding args was designed to enable variable arg count functions like printf("conversion list", arg0, … argn);

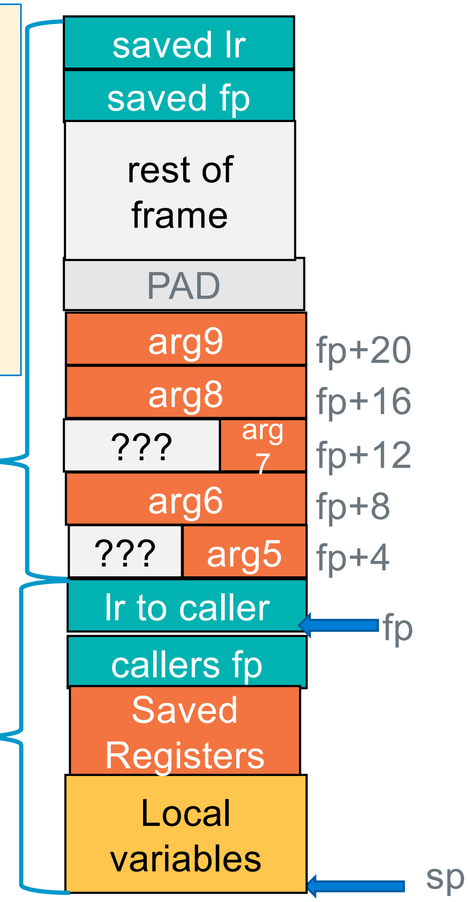
```
int func(int a1, int a2, int a3, int a4,
         short a5, int a6, char a7, int a8, int a9)
```

Constant	*Offset*	*arm ldr /str statement*
ARGN	(N-4)*4	`ldr  r4, [fp, ARGN]`
ARG9	20	`ldr  r4, [fp, ARG9]`
ARG8	16	`ldr  r4, [fp, ARG8]`
ARG7	12	`ldrb r4, [fp, ARG7]`
ARG6	8	`ldr  r4, [fp, ARG6]`
ARG5	4	`ldrh r4, [fp, ARG5]`

```
.equ ARG9,  20
.equ ARG8,  16
.equ ARG7,  12
.equ ARG6,   8
.equ ARG5,   4
```

Callers Stack frame
no defined limit to number of args, keep going up stack 4 bytes at a time

Current Stack Frame



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

x