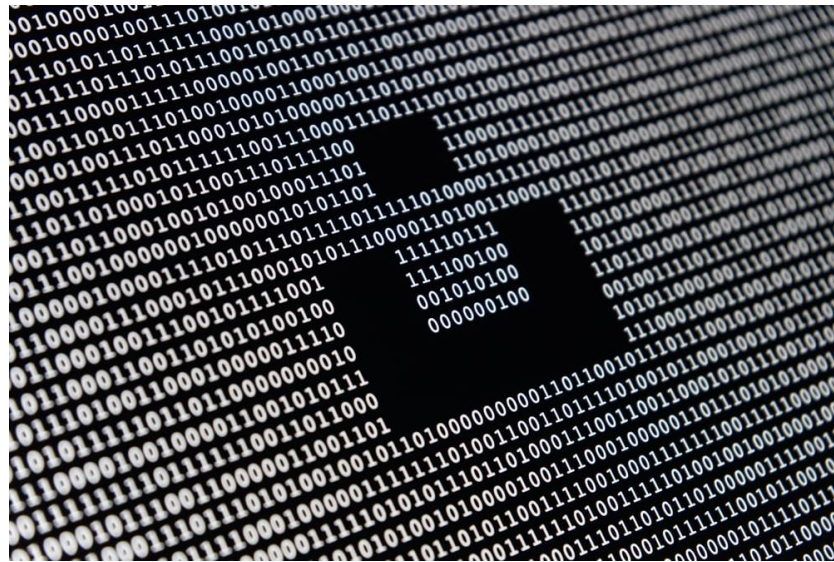


# COS 217: Introduction to Programming Systems

## Machine Language



PRINCETON UNIVERSITY



# Instruction Set Architecture (ISA)

There are many kinds of computer chips out there:

ARM (AArch64)

Intel x86 series

IBM PowerPC

RISC-V

MIPS

(and, in the old days, dozens more)

Each of these different “machine architectures” understands a different *machine language* – binary encoding of instructions



# Machine Language

Today we'll cover:

- A motivating example from Assignment 6: Buffer Overrun
- The AARCH64 machine language

Next time (our last lecture 🥲) we'll cover:

- The assembly and linking processes



# Flashback to last lecture ...



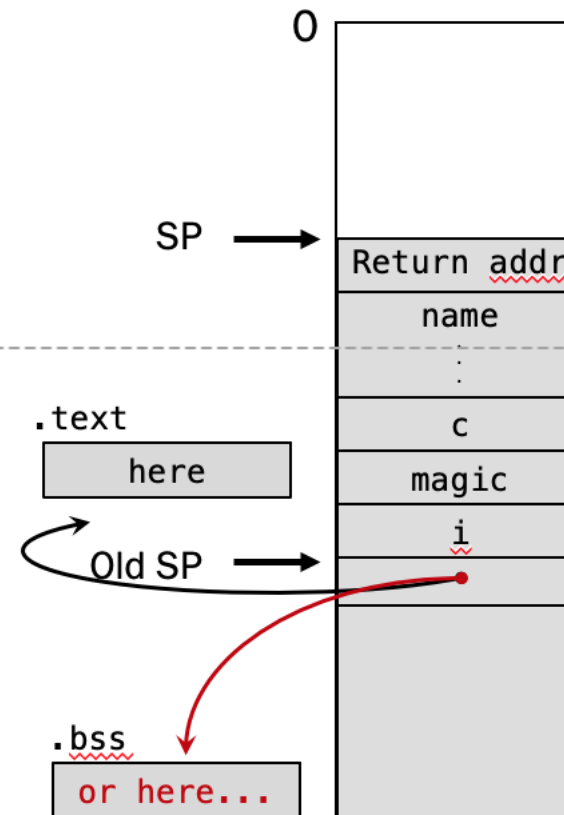
## It Gets **Much, Much** Worse...

Buffer overrun can overwrite return address of a previous stack frame!

- Value can be an invalid address, leading to a segfault, or it can cleverly cause unintended control flow, **or even cause arbitrary malicious code to execute!**

```
#include <stdio.h>
int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while ((c = getchar()) != '\n')
        name[i++] = c;
    name[i] = '\0';
    printf("Thank you, %s.\n", name);
    printf("The answer to life, the universe, "
           "and everything is %d\n", magic);
    return 0;
}
```

10





# Assignment 6: Attack the “Grader” Program

```
/* Prompt for name and read it */  
void getName() {  
    printf("What is your name?\n");  
    readString();  
}
```

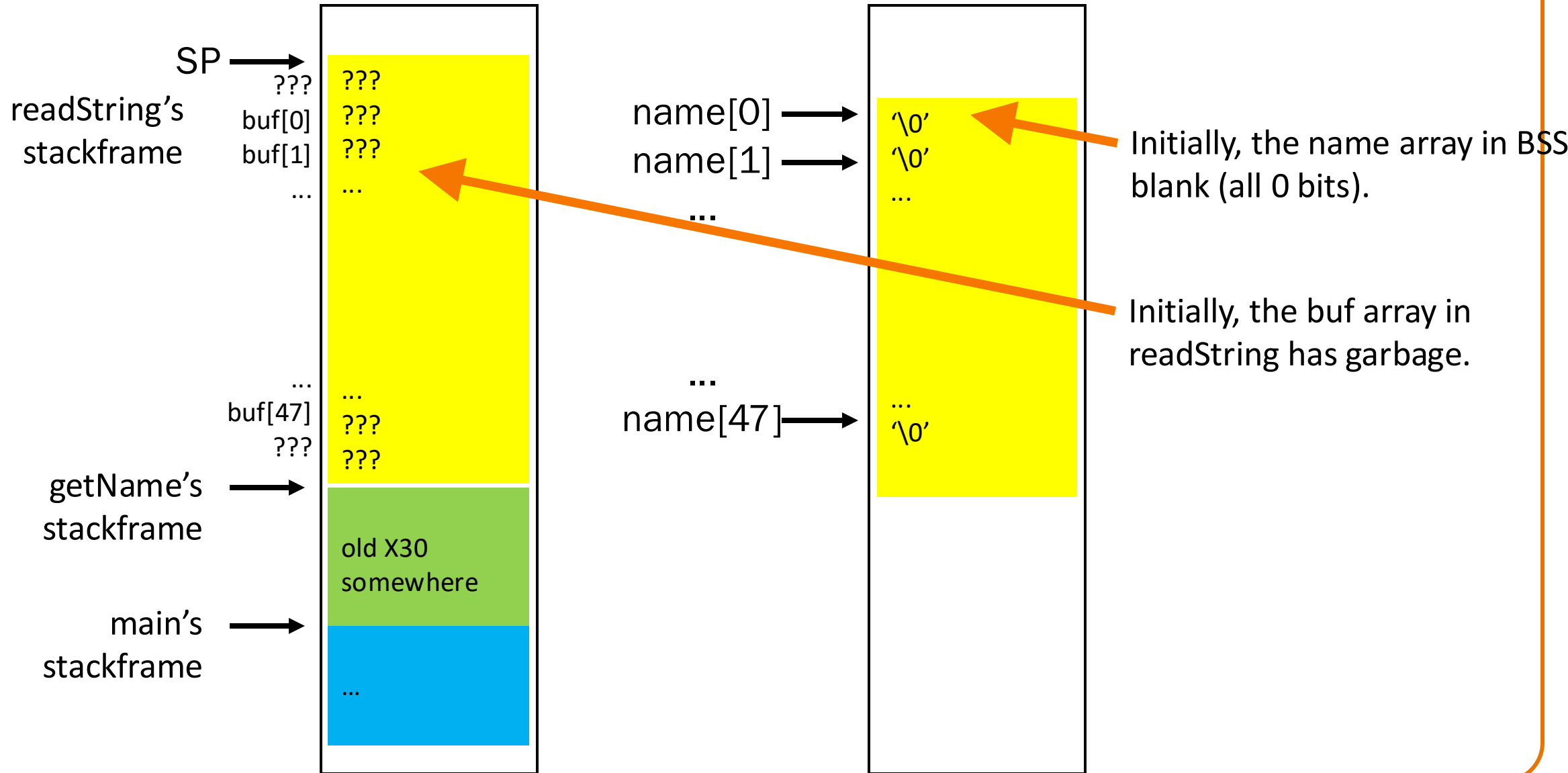
Unchecked  
write to  
buffer!

Opportunity to inject  
instructions into  
persistent memory!

```
/* Read a string into name */  
void readString() {  
    char buf[BUFSIZE];  
    int i = 0;  
    int c;  
  
    /* Read string into buf[] */  
    for (;;) {  
        c = fgetc(stdin);  
        if (c == EOF || c == '\n')  
            break;  
        buf[i] = c;  
        i++;  
    }  
    buf[i] = '\0';  
  
    /* Copy buf[] to name[] */  
    for (i = 0; i < BUFSIZE; i++)  
        name[i] = buf[i];  
}
```

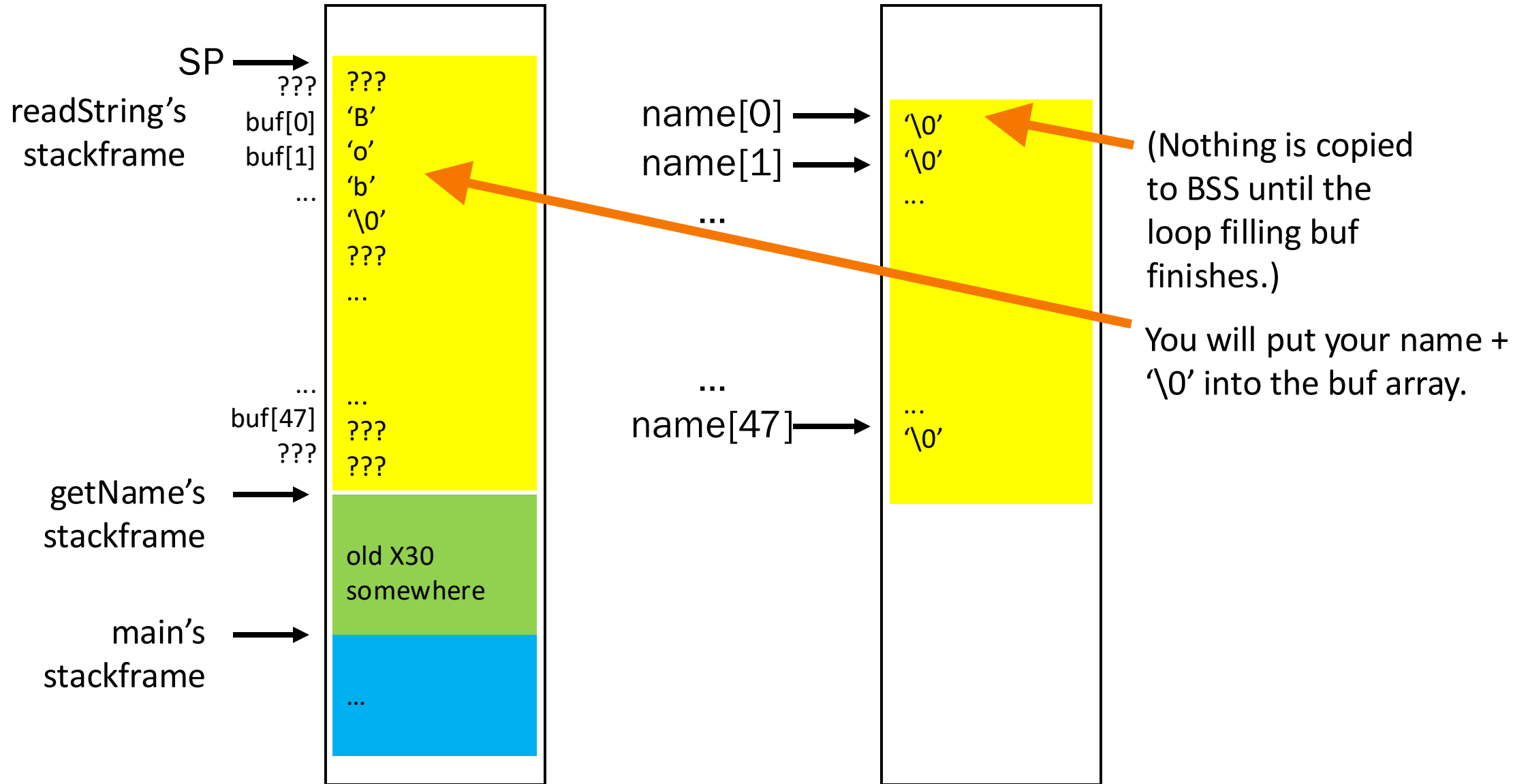


# Memory Map of Stack and BSS Section



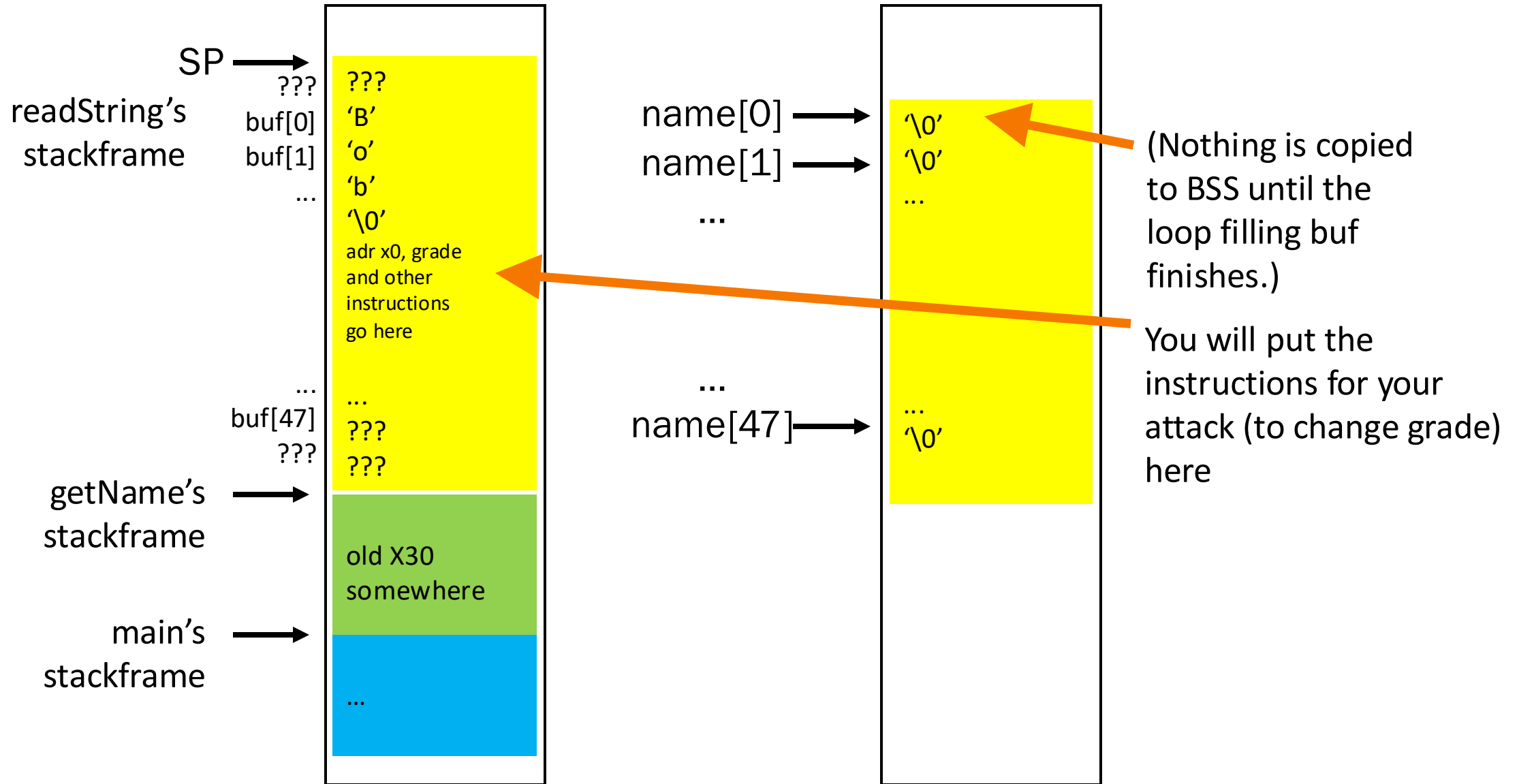


# Memory Map of Stack and BSS Section





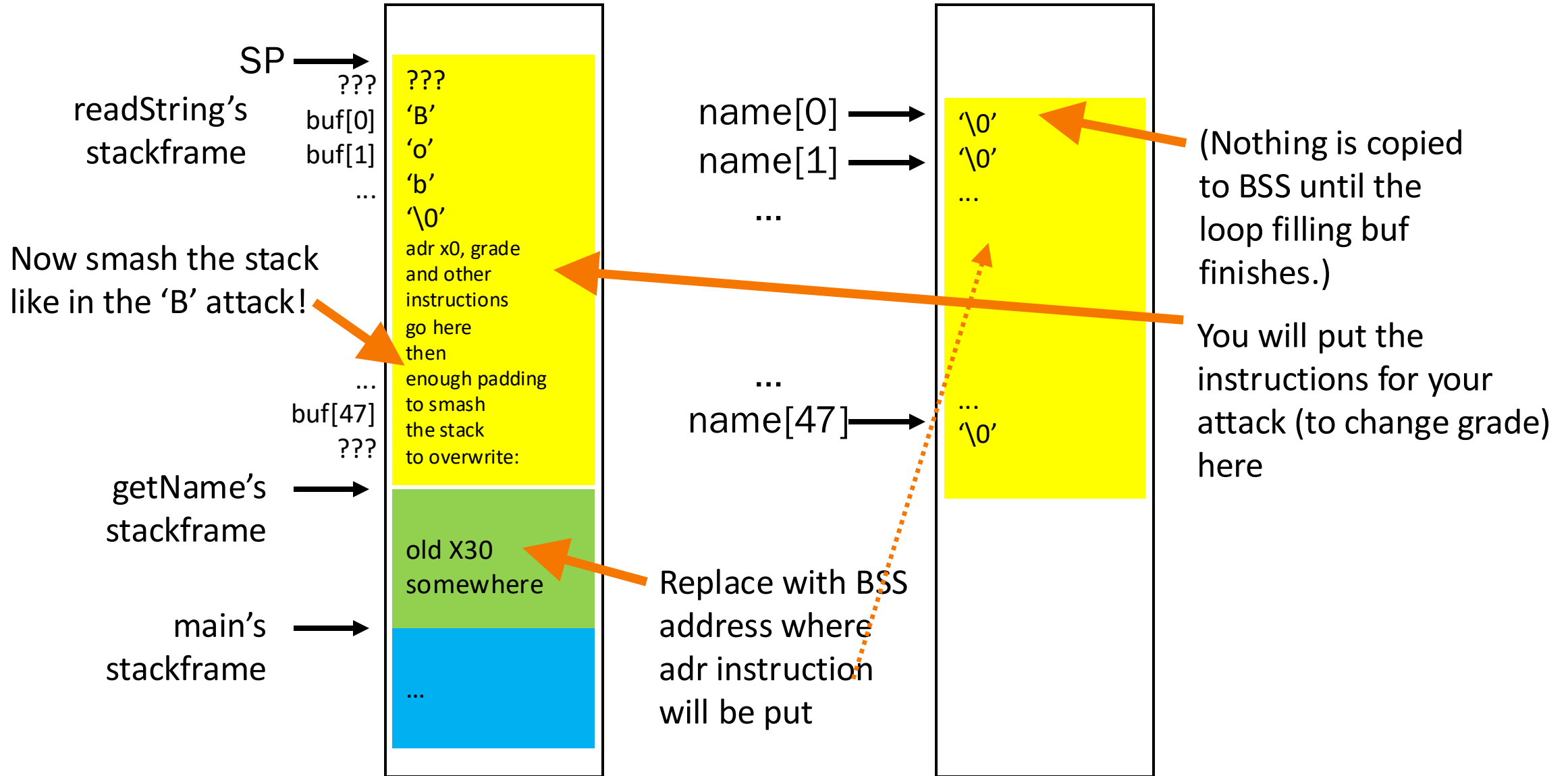
# Memory Map of Stack and BSS Section





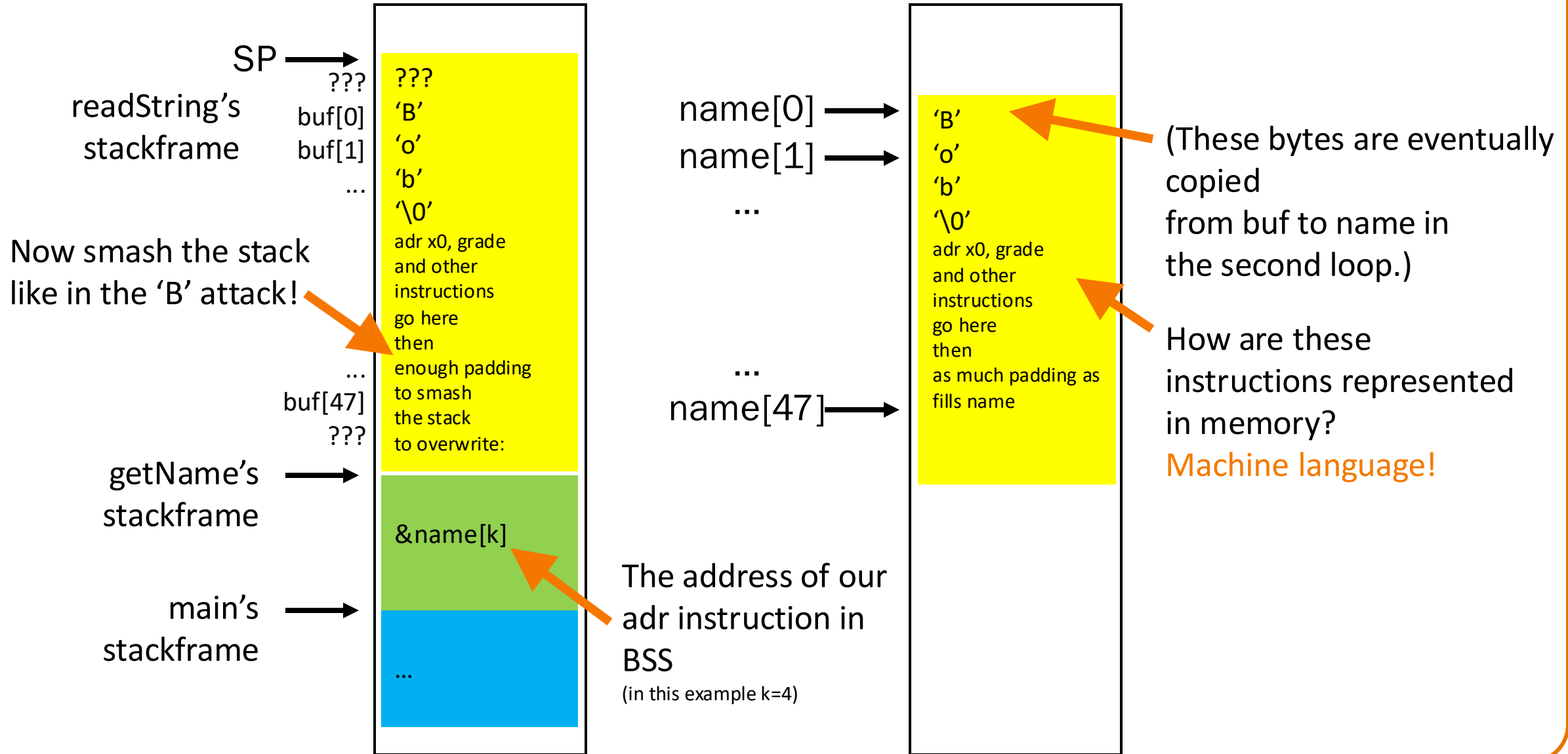


# Memory Map of Stack and BSS Section





# Memory Map of Stack and BSS Section



# Agenda



A6 “A” Attack

AARCH64 Machine Language

Assembly Language: add x1, x2, x3

Machine Language: 1000 1011 0000 0011 0000 0000 0100 0001

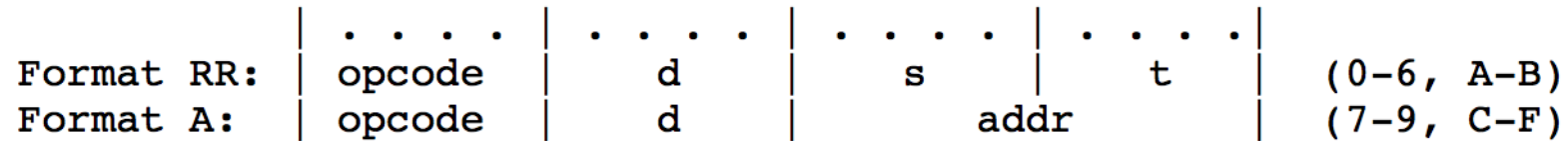


# Machine Language: TOY → AARCH64

## INSTRUCTION FORMATS

Remember TOY?

ARM is more complex, but the same ideas!



## AARCH64 machine language

- All instructions are 32 bits long, 4-byte aligned
- Some bits allocated to *opcode*: what kind of instruction is this?
- Other bits specify register(s)
- Depending on instruction, other bits may be used for an immediate value, a memory offset, an offset to jump to, etc.

## Instruction formats

- Variety of ways different instructions are encoded
- We'll go over quickly in class, to give you a flavor
- Refer to slides as reference for Assignment 6!  
(Every instruction format you'll need is in the following slides... we think...)



# AARCH64 Instruction Format

msb: bit 31



XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

lsb: bit 0



## Operation group

- Encoded in bits 25-28
- x101: Data processing – 3-register
- 100x: Data processing – immediate + register(s)
- 101x: Branch
- x1x0: Load/store



# AARCH64 Instruction Format

msb: bit 31



lsb: bit 0



wxsx 101x xxxr rrrr xxxx xxrr rrrr rrrr

## Op. Group: Data processing – 3-register

- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Second source register in bits 16-20
- First source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction



# AARCH64 Instruction Format

msb: bit 31



1000 1011 0000 0011 0000 0000 0100 0001

lsb: bit 0



Example: add x1, x2, x3

- opcode = add
- Instruction width in bit 31: 1 = 64-bit
- Whether to set condition flags in bit 29: no
- Second source register in bits 16-20: 3
- First source register in bits 5-9: 2
- Destination register in bits 0-4: 1
- Additional information about instruction: none



# AARCH64 Instruction Format

msb: bit 31



wxs1 00xx xxii iiiiiiii iirr rrrr rrrr

wxx1 0010 1xxi iiiiiiii iiii rrrr

lsb: bit 0



Op. Group: Data processing – immediate + register(s)

- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Immediate value in bits 10-21 for 2-register instructions, bits 5-20 for 1-register instructions
- Source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction





# AARCH64 Instruction Format

msb: bit 31



0111 0001 0000 0000 1010 1000 0100 0001

lsb: bit 0



Example: subs w1, w2, 42

- opcode: subtract immediate
- Instruction width in bit 31: 0 = 32-bit
- Whether to set condition flags in bit 29: yes
- Immediate value in bits 10-21:  $101010_b = 42$
- First source register in bits 5-9: 2
- Destination register in bits 0-4: 1
- Additional information about instruction: none

# AARCH64 Instruction Format

\*\*You may find this slide useful for A6



msb: bit 31



1101 0010 1000 0000 0000 0101 0100 0001

lsb: bit 0



Example: mov x1, 42

- opcode: move immediate
- Instruction width in bit 31: 1 = 64-bit
- Immediate value in bits 5-20:  $101010_b = 42$
- Destination register in bits 0-4: 1



# AARCH64 Instruction Format

msb: bit 31



xxx1 01ii iiiiiiii iiiiiiii iiiiiiii iiiiiiii

xxx1 01xx iiiiiiii iiiiiiii iiiiiiii iiiix cccc

lsb: bit 0



## Op. Group: Branch

- *Relative* address of branch target in bits 0-25 for unconditional branch (b) and function call (bl)
- *Relative* address of branch target in bits 5-23 for conditional branch
- Because all instructions are 32 bits long and are 4-byte aligned, relative addresses end in 00. Because this is invariable, we can omit those two bits from our representation. Doing so provides more range with the same number of bits!
- Type of conditional branch encoded in bits 0-3



# Displacement Discombobulation



msb: bit 31



xxx1 01ii iiiiiiii iiiiiiii iiiiiiii iiiiiiii iiiiiiii

lsb: bit 0



What is the range of the relative address?

- A. 0 – 64MB
- B. -32MB – +32MB
- C. 0 – +256MB
- D. -128MB – +128MB

D: 26 bits + 2 "chopped off" bits  
= 28 bits: 256MB.

2's complement splits half  
negative / half non-negative

# AARCH64 Instruction Format

\*\*You may find this slide useful for A6



msb: bit 31



0001 0111 1111 1111 1111 1111 1111 1101

lsb: bit 0



## Example: b someLabel

- This depends on where someLabel is relative to this instruction!  
For this example, someLabel is 3 instructions (12 bytes) *earlier*
- **opcode: unconditional branch**
- *Relative address in bits 0-25: 26-bit two's complement of  $11_b$ .  
Shift left by 2:  $1100_b = 12$ . So, offset is -12.*



# AARCH64 Instruction Format

msb: bit 31



1001 0111 1111 1111 1111 1111 1111 1101

lsb: bit 0



## Example: bl someLabel

- This depends on where someLabel is relative to this instruction!  
For this example, someLabel is 3 instructions (12 bytes) *earlier*
- **opcode: branch and link (function call)**
- **Relative address in bits 0-25: 26-bit two's complement of  $11_b$ .**  
Shift left by 2:  $1100_b = 12$ . So, offset is -12.



# AARCH64 Instruction Format

msb: bit 31



lsb: bit 0



0101 0100 0000 0000 0000 0000 0110 1101

Example: ble someLabel

- This depends on where someLabel is relative to this instruction!  
For this example, someLabel is 3 instructions (12 bytes) *later*
- **opcode: conditional branch**
- *Relative address in bits 5-23: 11<sub>b</sub>. Shift left by 2: 1100<sub>b</sub> = 12*
- **Conditional branch type in bits 0-3: LE**



# AARCH64 Instruction Format

msb: bit 31



lsb: bit 0



wwxx 1x0x xxxr rrrr xxxx xxrr rrrr rrrr

wwxx 1x0x xxii iiiiiiii iirr rrrr rrrr

## Op. Group: Load / store

- Instruction width in bits 30-31: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit, 11 = 64-bit
- For [Xn,Xm] addressing mode: second source register in bits 16-20
- For [Xn,offset] addressing mode: offset in bits 10-21, shifted left by 3 bits for 64-bit, 2 bits for 32-bit, 1 bit for 16-bit
- First source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction, e.g. scaled mode





# AARCH64 Instruction Format

msb: bit 31



1111 1000 0110 0010 0110 1000 0010 0000

lsb: bit 0



Example: ldr x0, [x1, x2]

- opcode: load, register+register
- Instruction width in bits 30-31: 11 = 64-bit
- Second source register in bits 16-20: 2
- First source register in bits 5-9: 1
- Destination register in bits 0-4: 0
- Additional information about instruction: no LSL



# AARCH64 Instruction Format

msb: bit 31



lsb: bit 0



1111 1001 0000 0000 0000 1111 1110 0000

Example: `str x0, [sp,24]`

- opcode: store, register+offset
- Instruction width in bits 30-31: 11 = 64-bit
- Offset value in bits 12-20:  $11_b$ , shifted left by 3 =  $11000_b = 24$
- “Source” (really destination!) register in bits 5-9: 31 = sp
- “Destination” (really source!) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from others: “source” and “destination” are flipped!

# AARCH64 Instruction Format

\*\*You may find this slide useful for A6



msb: bit 31



lsb: bit 0



0011 1001 0000 0000 0110 0011 1110 0000

Example: strb w0, [sp,24]

- opcode: store, register+offset
- Instruction width in bits 30-31: 00 = 8-bit
- Offset value in bits 12-20: 11000<sub>b</sub> (don't shift left!) = 24
- “Source” (really destination!) register in bits 5-9: 31 = sp
- “Destination” (really source!) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from others: “source” and “destination” are flipped!



# AARCH64 Instruction Format

msb: bit 31



0ii1 0000 iiiii iiiii iiiii iiiii iiir rrrr

lsb: bit 0



## ADR instruction

(Distinct from others w/ Op Group bits 100x)

- Specifies *relative* position of label (data location)
- 19 High-order bits of offset in bits 5-23
- 2 Low-order bits of offset in bits 29-30
- Destination register in bits 0-4

# AARCH64 Instruction Format

\*\*You may find this slide useful for A6



msb: bit 31



0101 0000 0000 0000 0000 0001 1001 0011

lsb: bit 0



Example: `adr x19, someLabel`

- This depends on where `someLabel` is relative to this instruction!  
For this example, `someLabel` is 50 bytes later
- **opcode: generate address**
- **19 High-order bits of offset in bits 5-23: 1100**
- **2 Low-order bits of offset in bits 29-30: 10**
- **Relative data location is  $110010_b = 50$  bytes after this instruction**
- **Destination register in bits 0-4: 19**