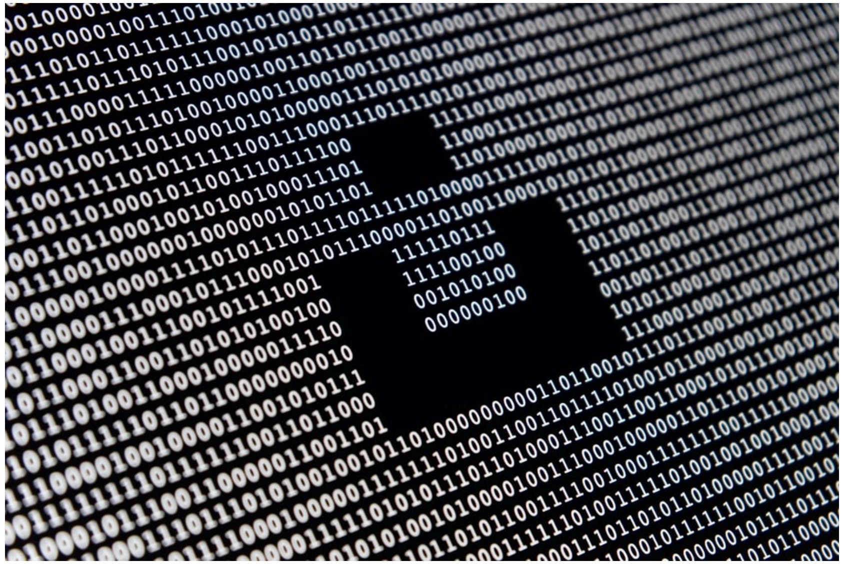
COS 217: Introduction to Programming Systems

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

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

(and, in the old days, dozens more)

@swimstaralex (previous slide)

Machine Language

Today we'll cover:

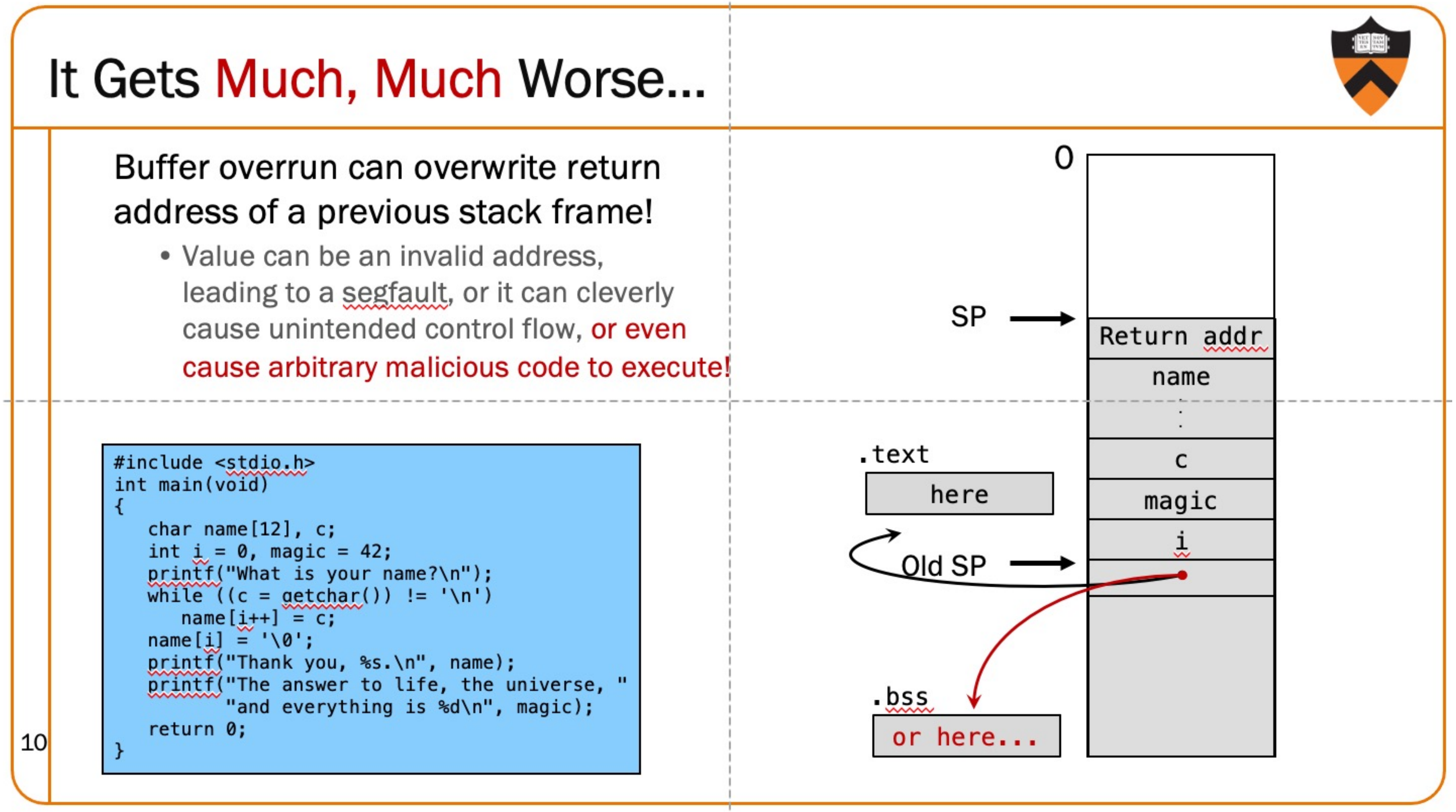
- A motivating example from Assignment 6: Buffer Overrun
- The AARCH64 machine language

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

- The assembly and linking processes

@olajidetunde

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;
}
```

Assignment 6: Attack the “Grader” Program

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

Unchecked write to buffer!

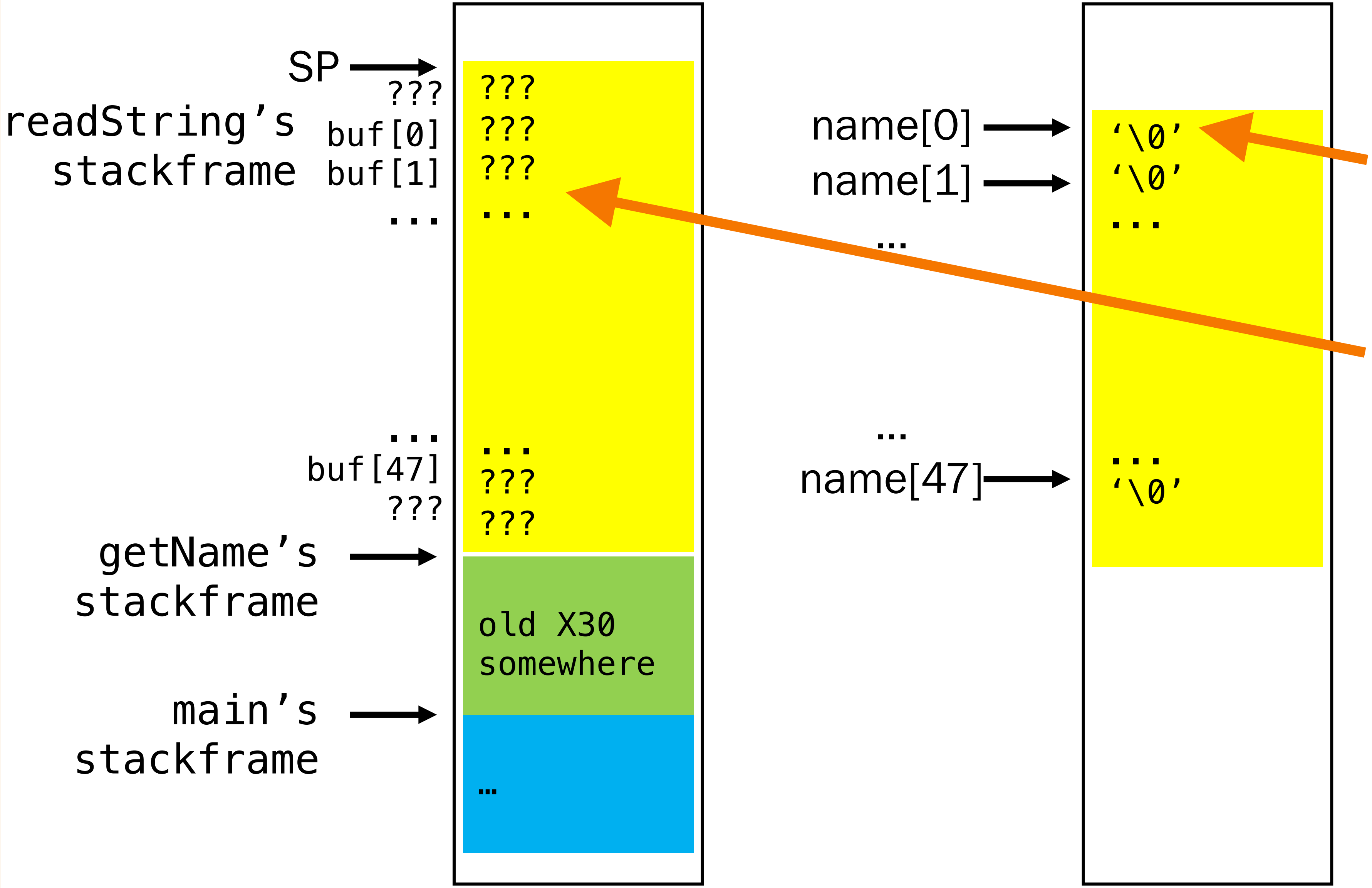
Opportunity to inject instructions into persistent memory!

```c
/* 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];
}
```

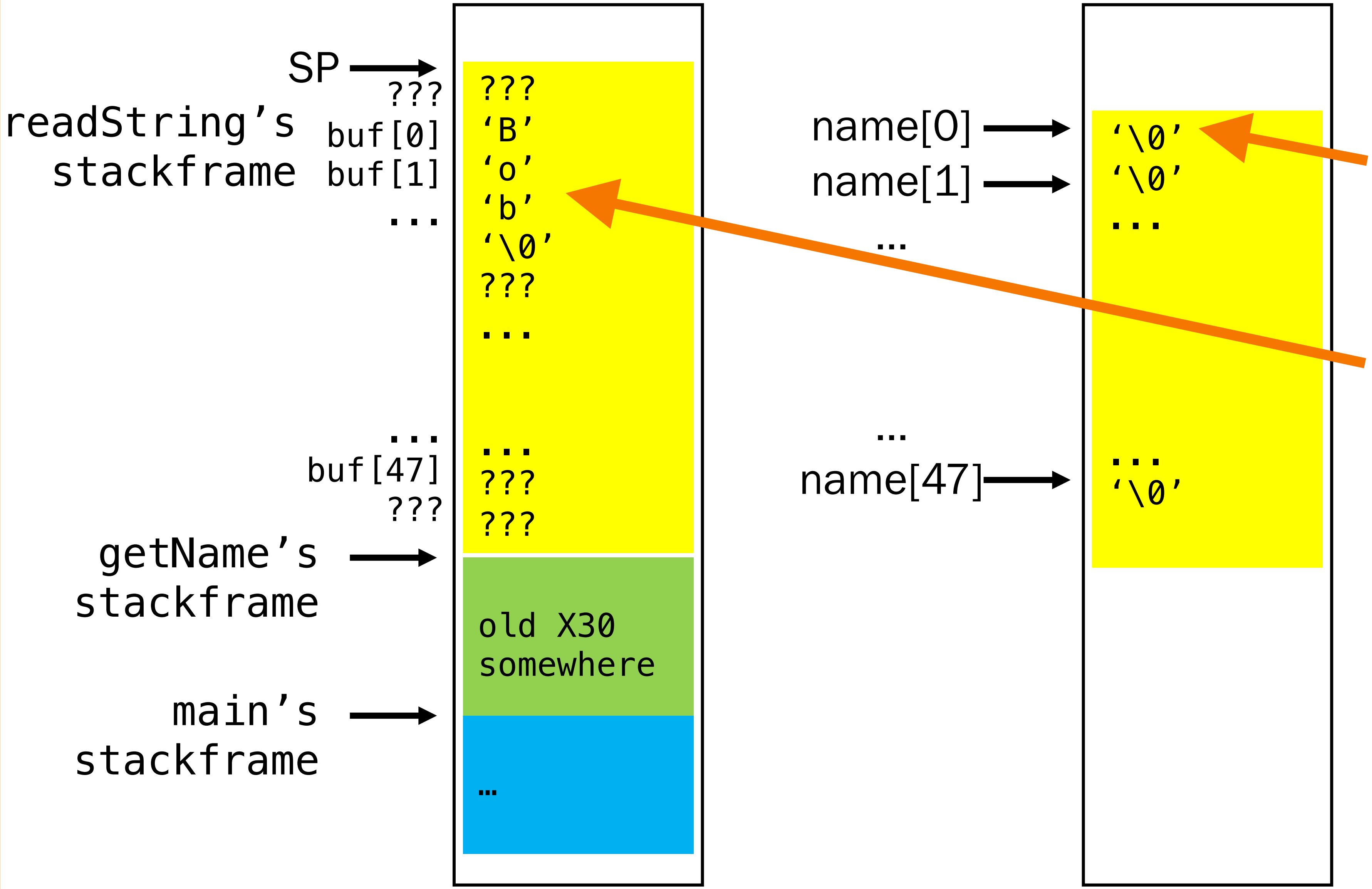
Memory Map of Stack and BSS Section



Initially, the name array in BSS is blank (all 0 bits).

Initially, the buf array in readString has garbage.

Memory Map of Stack and BSS Section



SP

readString's stackframe

???
buf[0]
buf[1]
...

...
buf[47]
???

getName's stackframe

main's stackframe

???
'B'
'o'
'b'
'\0'
???
...

...
???
???

old X30 somewhere

…

name[0]
name[1]
…

…
name[47]

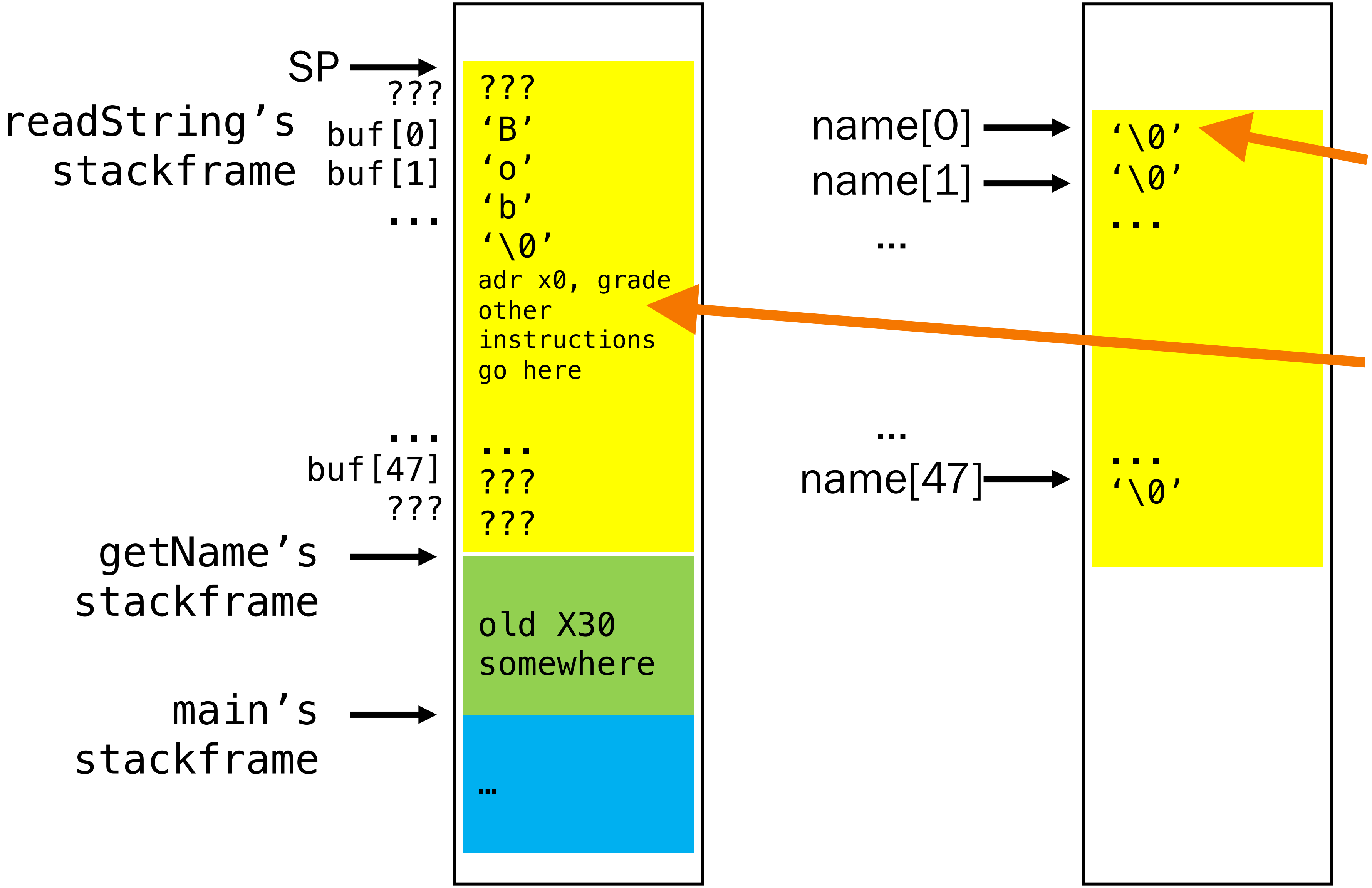
'\0'
'\0'
...

...
'\0'

(Nothing is copied to BSS until the loop filling buf finishes.)

You will put your name + '\0' into the buf array.

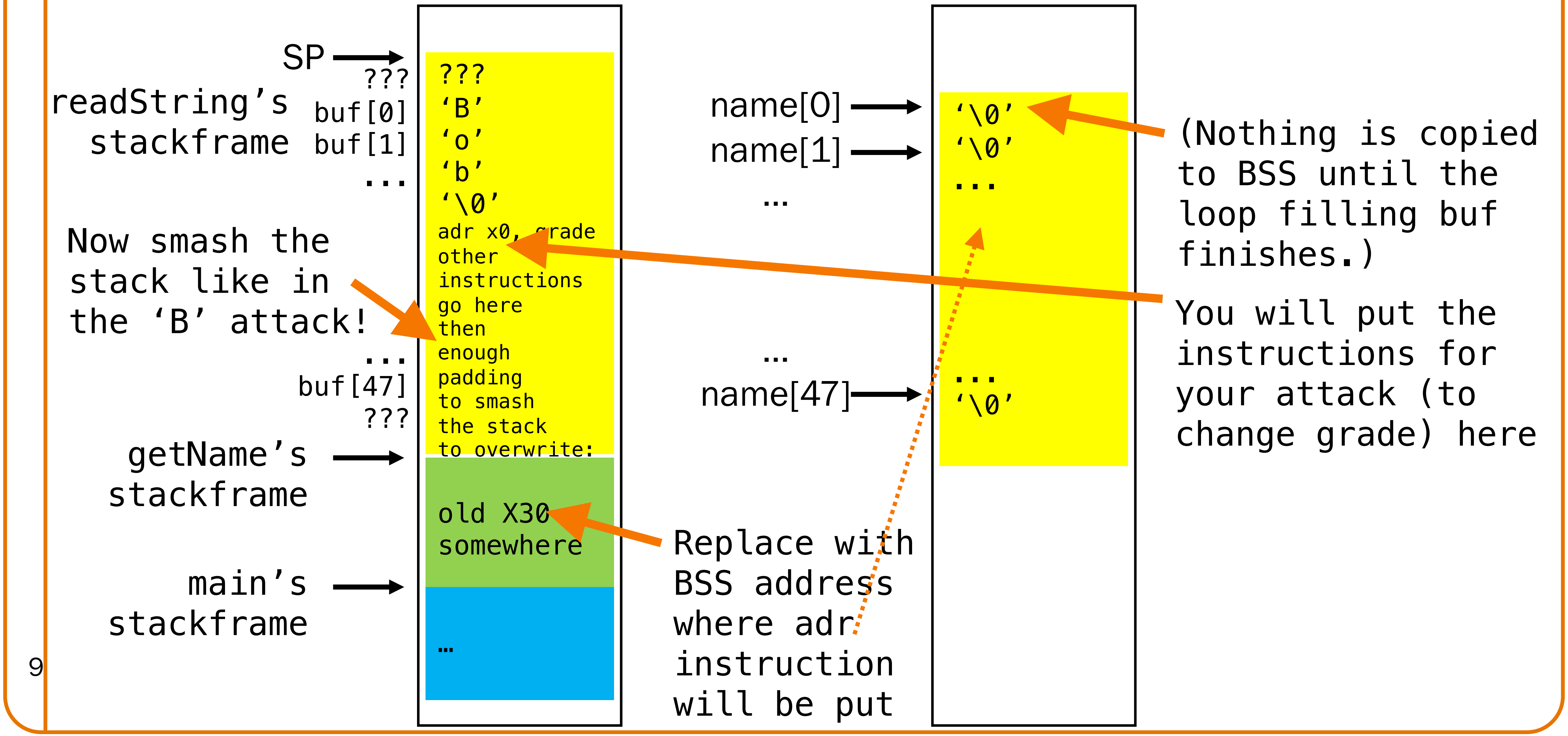
Memory Map of Stack and BSS Section



(Nothing is copied to BSS until the loop filling buf finishes.)

You will put the instructions for your attack (to change grade) here

Memory Map of Stack and BSS Section



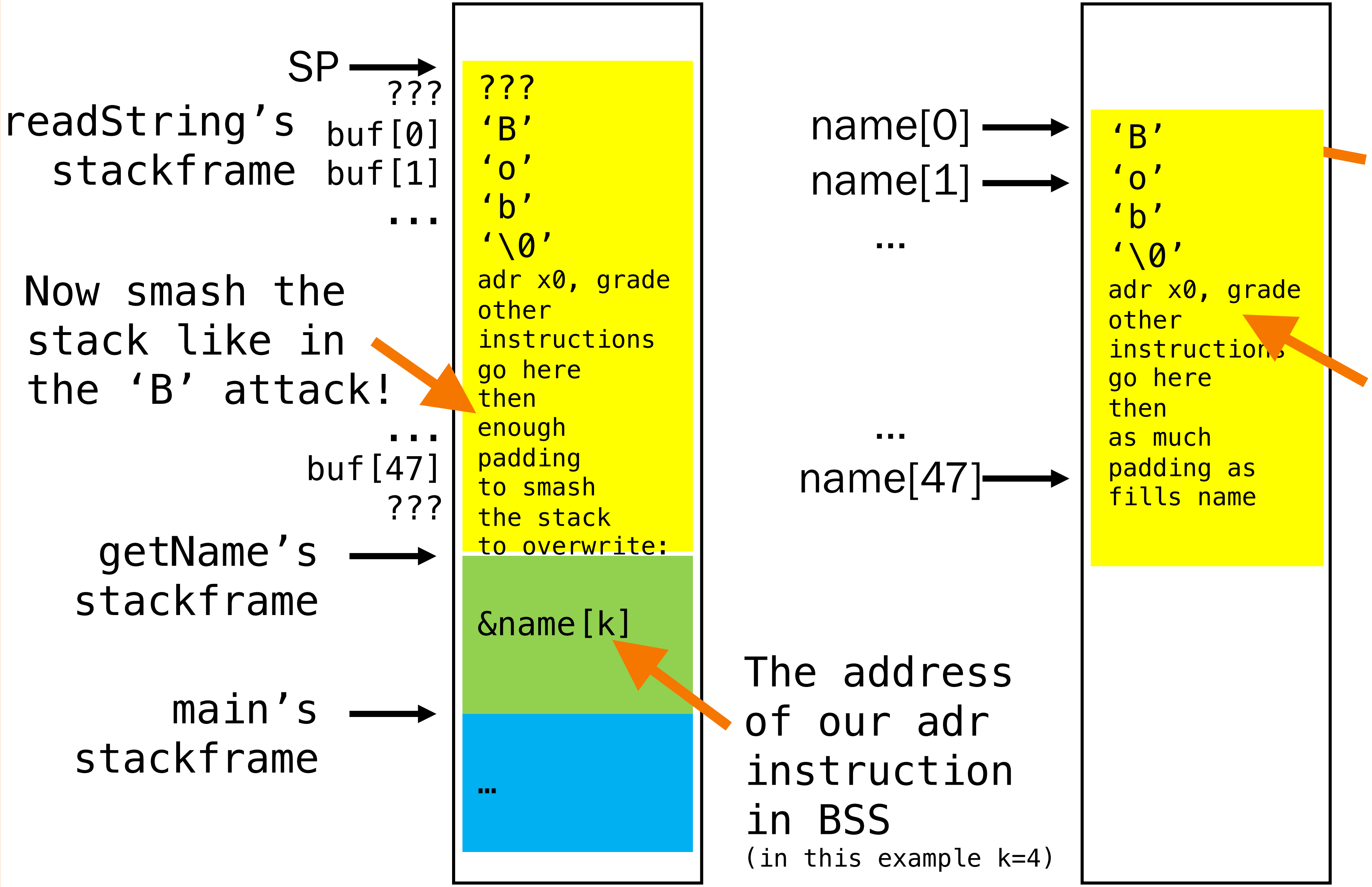
Now smash the stack like in the ‘B’ attack!

(Nothing is copied to BSS until the loop filling buf finishes.)

You will put the instructions for your attack (to change grade) here

Replace with BSS address where adr instruction will be put

Memory Map of Stack and BSS Section



SP

readString's stackframe

???

buf[0]

buf[1]

...

...

buf[47]

???

getName's stackframe

main's stackframe

???
'B'
'o'
'b'
'\0'
adr x0, grade
other
instructions
go here
then
enough
padding
to smash
the stack
to overwrite:

&name[k]

…

name[0]

name[1]

…

…

name[47]

'B'
'o'
'b'
'\0'
adr x0, grade
other
instructions
go here
then
as much
padding as
fills name

Now smash the stack like in the 'B' attack!

The address of our adr instruction in BSS

(in this example k=4)

(Nothing is copied to BSS until the loop filling buf finishes.)

How are these instructions represented in memory?

Machine language!

Agenda

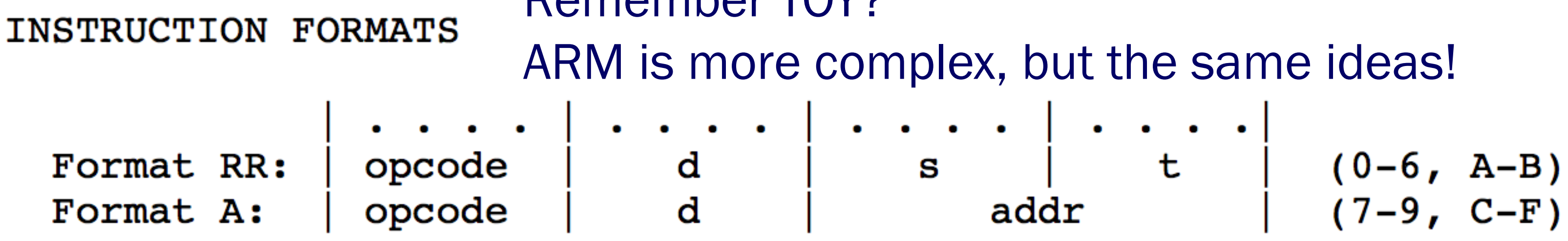
A6 “A” Attack

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

              | . . . . | . . . . | . . . . | . . . .|
  Format RR:  | opcode  |    d    |    s    |    t    |   (0-6, A-B)
  Format A:   | opcode  |    d    |       addr        |   (7-9, C-F)
```

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 ↓

lsb: bit 0 ↓

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
```

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 ↓ lsb: bit 0 ↓

```
1000 1011 0000 0011 0000 0000 0100 0001
```

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                                   lsb: bit 0
↓                                                      ↓
wxs1 00xx xxii iiii iiii iirr rrrr rrrr
wxx1 0010 1xxi iiii iiii iiii iiir rrrr
```

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 ↓

lsb: bit 0 ↓

```
0111 0001 0000 0000 1010 1000 0100 0001
```

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                            lsb: bit 0
↓                                               ↓
1101 0010 1000 0000 0000 0101 0100 0001
```

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 ↓ lsb: bit 0 ↓

```
xxx1 01ii iiii iiii iiii iiii iiii iiii
xxx1 01xx iiii iiii iiii iiii iiix cccc
```

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 ↓ … lsb: bit 0 ↓

```
xxx1 01ii iiii iiii iiii iiii iiii iiii
```

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

lsb: bit 0

```
0001 0111 1111 1111 1111 1111 1111 1101
```

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 ↓ … lsb: bit 0 ↓

```
1001 0111 1111 1111 1111 1111 1111 1101
```

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_b. Shift left by 2: 1100_b = 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 iiii iiii 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 ↓ lsb: bit 0 ↓

```
1111 1000 0110 0010 0110 1000 0010 0000
```

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_b (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                                                  lsb: bit 0
↓                                                                     ↓
0iii1 0000 iiii iiii iiii iiii iiir rrrr
```

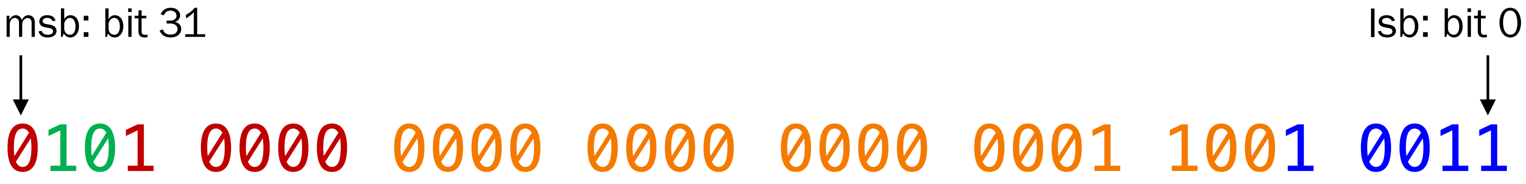
ADR instruction

(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

lsb: bit 0

```
0101 0000 0000 0000 0000 0001 1001 0011
```

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