Buffers & Direct Storage

Part 5

1

Buffers

Creating your own space

2

Buffers

- A *buffer* is any allocated block of memory that contains data
- This can hold anything:
  - text
  - image
  - file
  - etc.…

3

Buffers

- There are several assembly directives which will allocate space
- We have covered a few of them, but there are many – all with a specific purpose

4

A few directives that create space

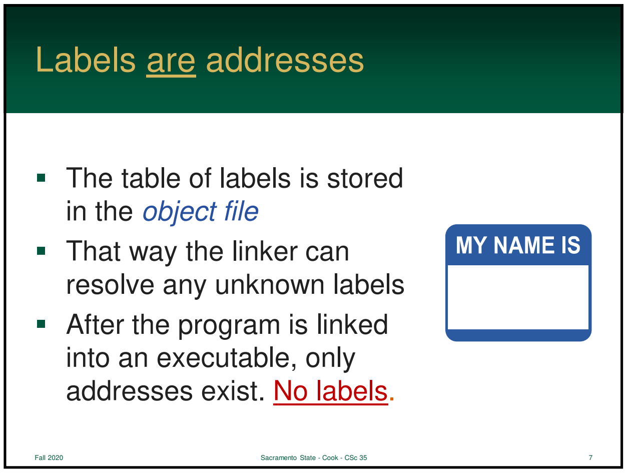
Directive	What it does
`.ascii`	Allocate enough space to store an ASCII string
`.quad`	Allocate 8-byte blocks with initial value(s)
`.byte`	Allocate byte(s) with initial value(s)
`.space`	Allocate any ***size*** of empty bytes (with initial values).

5

Labels are addresses

- Labels are used to keep track of memory locations
- They are stored, by the assembler, in a table
- Whenever a label is used in the program, the assembler substitutes the address

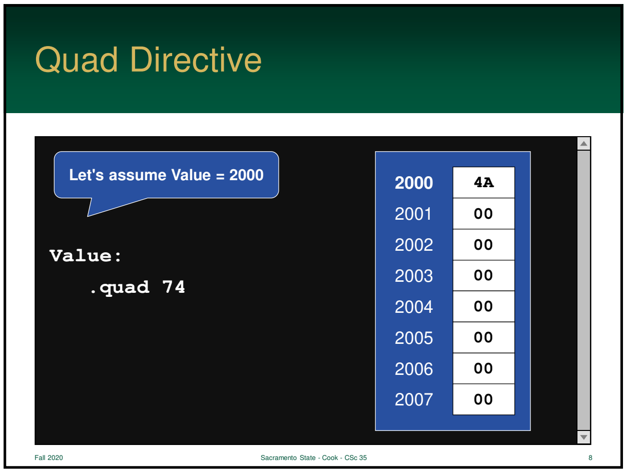
6



Labels are addresses

- The table of labels is stored in the *object file*
- That way the linker can resolve any unknown labels
- After the program is linked into an executable, only addresses exist. No labels.

7



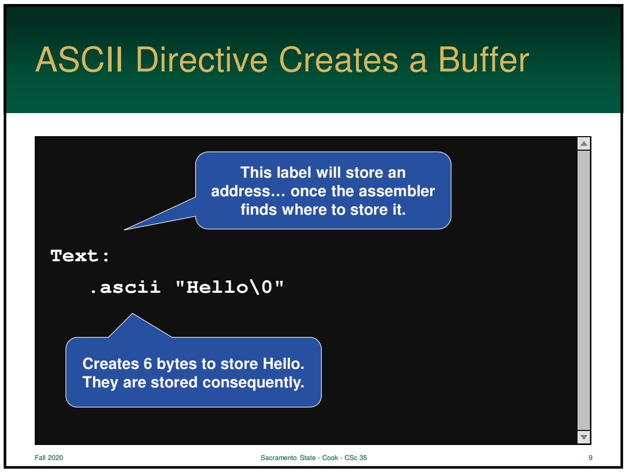
Quad Directive

Let's assume Value = 2000

```
Value:
    .quad 74
```

2000	4A
2001	00
2002	00
2003	00
2004	00
2005	00
2006	00
2007	00

8



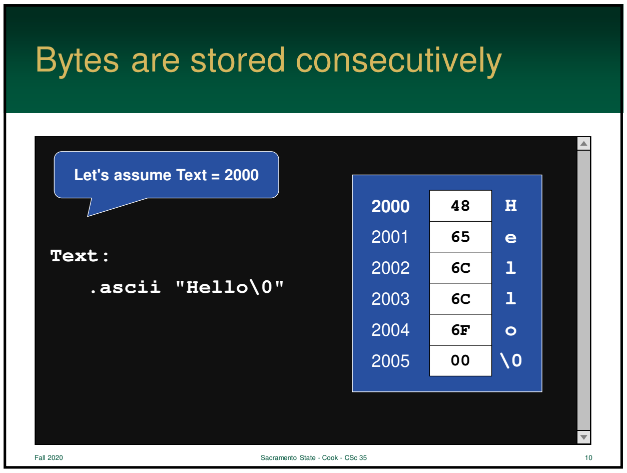
ASCII Directive Creates a Buffer

This label will store an address… once the assembler finds where to store it.

```
Text:
    .ascii "Hello\0"
```

Creates 6 bytes to store Hello. They are stored consequently.

9



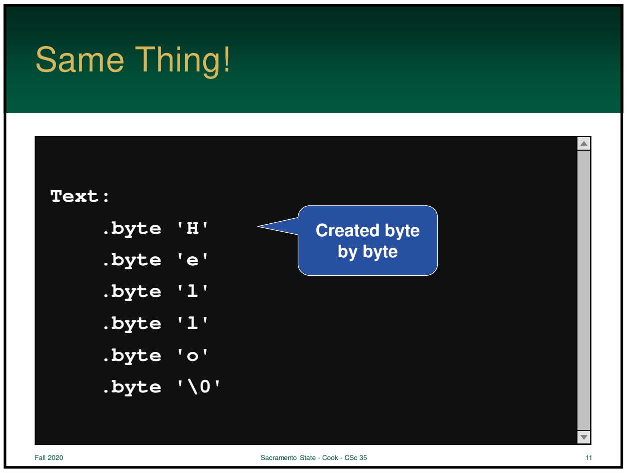
Bytes are stored consecutively

Let's assume Text = 2000

```
Text:
    .ascii "Hello\0"
```

2000	48	H
2001	65	e
2002	6C	l
2003	6C	l
2004	6F	o
2005	00	\0

10

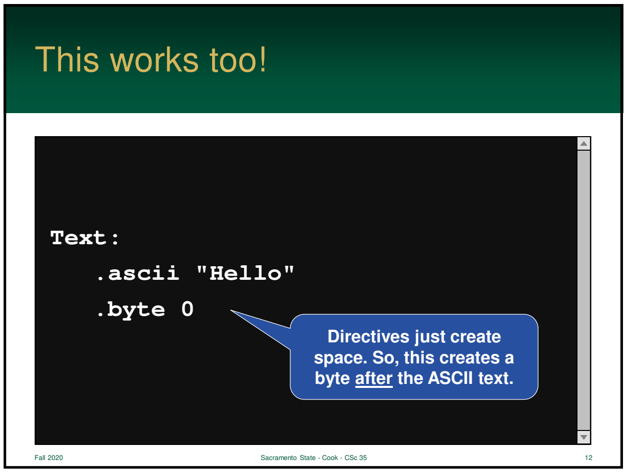


Same Thing!

```
Text:
     .byte 'H'
     .byte 'e'
     .byte 'l'
     .byte 'l'
     .byte 'o'
     .byte '\0'
```

Created byte by byte

11

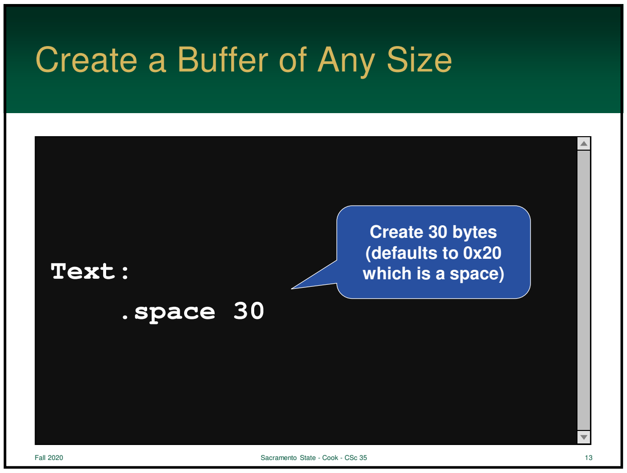


This works too!

```
Text:
    .ascii "Hello"
    .byte 0
```

Directives just create space. So, this creates a byte after the ASCII text.

12

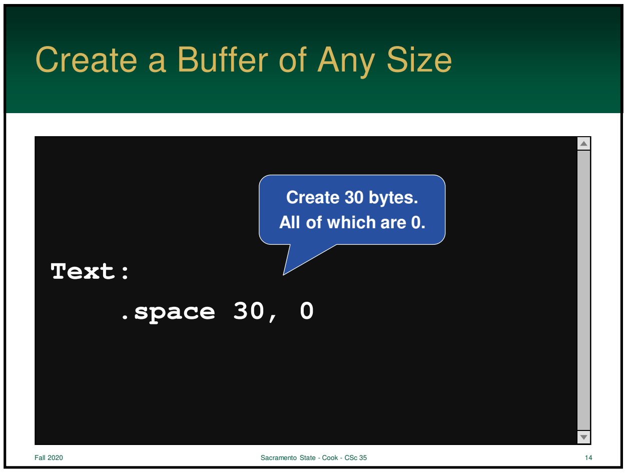


Create a Buffer of Any Size

```
Text:
    .space 30
```

Create 30 bytes (defaults to 0x20 which is a space)

13



Create a Buffer of Any Size

```
Text:
    .space 30, 0
```

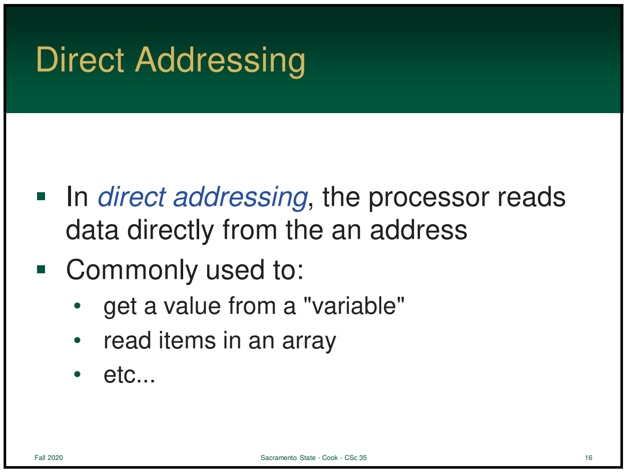
Create 30 bytes. All of which are 0.

14

Direct Addressing

Using memory… finally

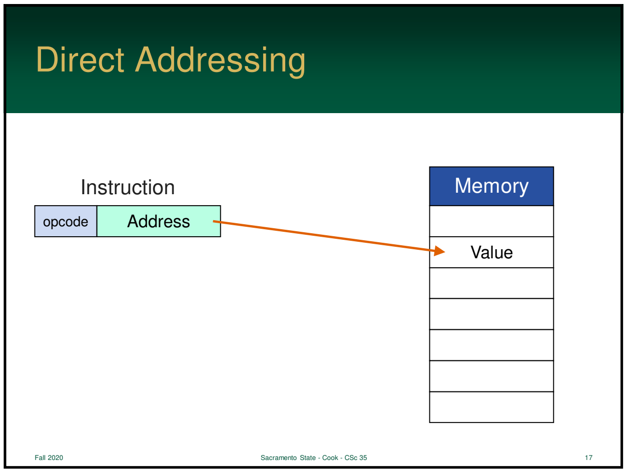
15



Direct Addressing

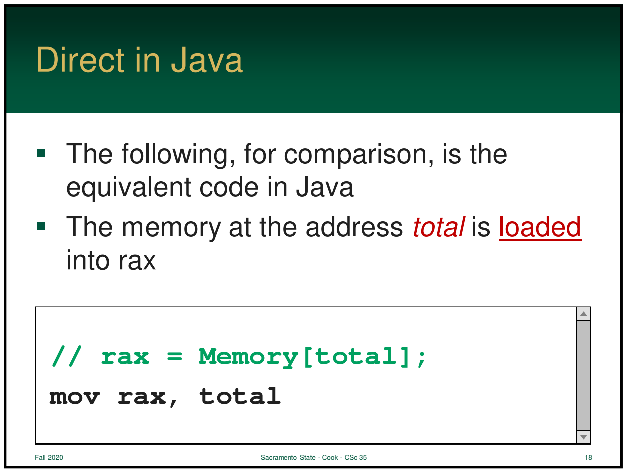
- In *direct addressing*, the processor reads data directly from the an address
- Commonly used to:
  - get a value from a "variable"
  - read items in an array
  - etc...

16



Direct Addressing

17

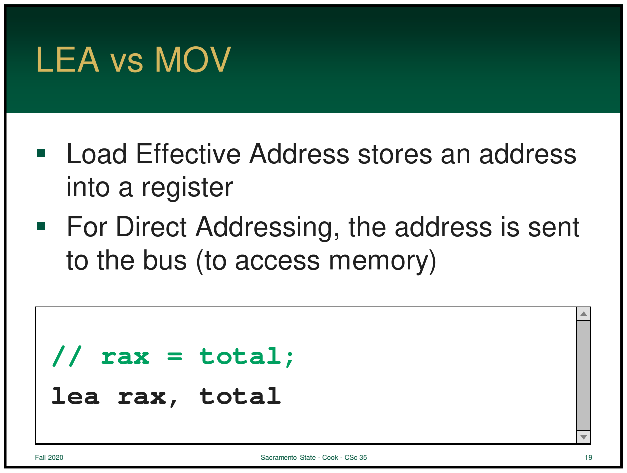


Direct in Java

- The following, for comparison, is the equivalent code in Java
- The memory at the address *total* is loaded into rax

```
// rax = Memory[total];
mov rax, total
```

18

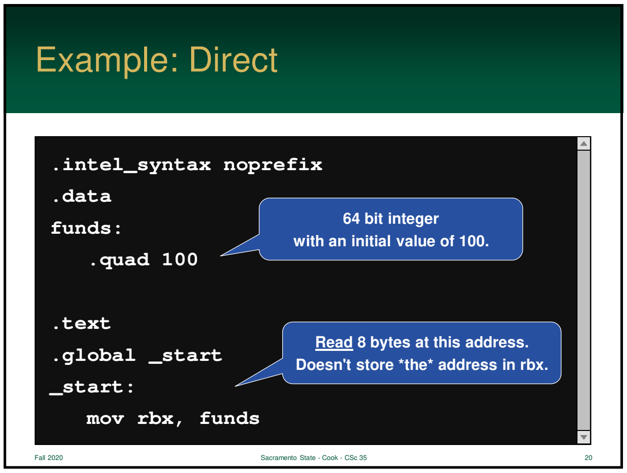


LEA vs MOV

- Load Effective Address stores an address into a register
- For Direct Addressing, the address is sent to the bus (to access memory)

```
// rax = total;
lea rax, total
```

19



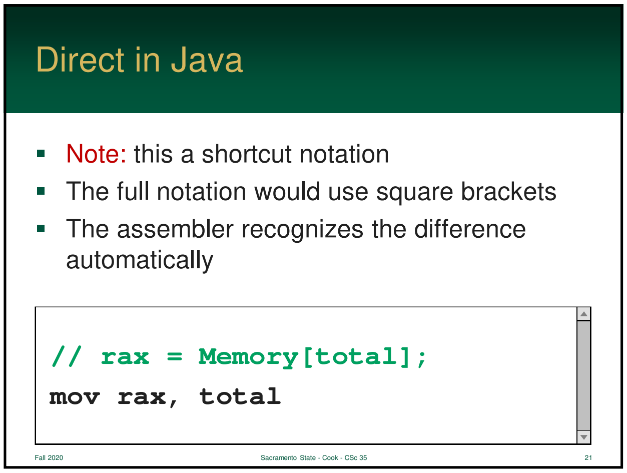
Example: Direct

```
.intel_syntax noprefix
.data
funds:
    .quad 100

.text
.global _start
_start:
    mov rbx, funds
```

64 bit integer with an initial value of 100.

Read 8 bytes at this address. Doesn't store *the* address in rbx.

20

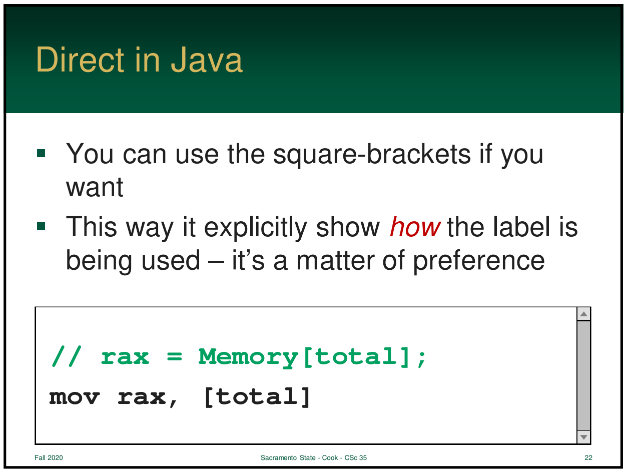


Direct in Java

- Note: this a shortcut notation
- The full notation would use square brackets
- The assembler recognizes the difference automatically

```
// rax = Memory[total];
mov rax, total
```

21

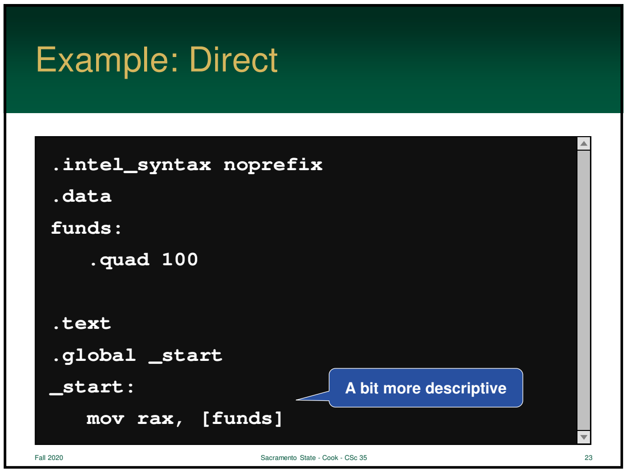


Direct in Java

- You can use the square-brackets if you want
- This way it explicitly show *how* the label is being used – it's a matter of preference

```
// rax = Memory[total];
mov rax, [total]
```

22

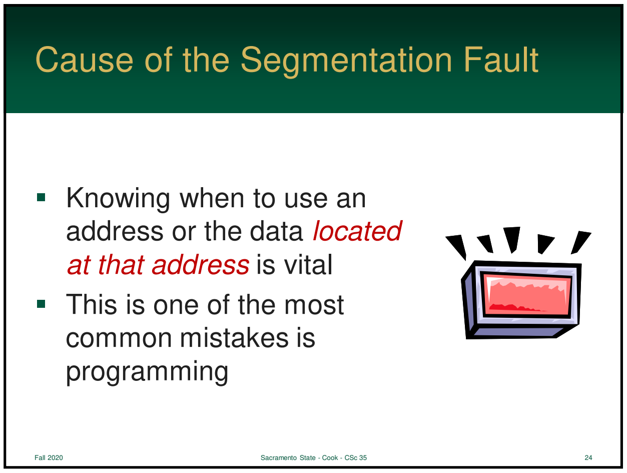


Example: Direct

```
.intel_syntax noprefix
.data
funds:
    .quad 100

.text
.global _start
_start:
    mov rax, [funds]
```

A bit more descriptive

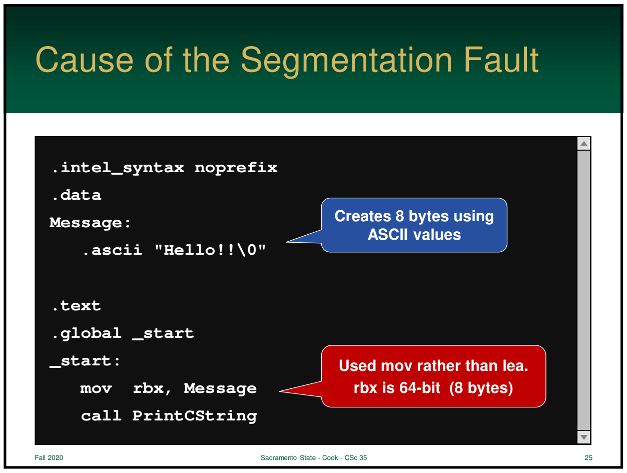
23



Cause of the Segmentation Fault

- Knowing when to use an address or the data *located at that address* is vital
- This is one of the most common mistakes is programming

24



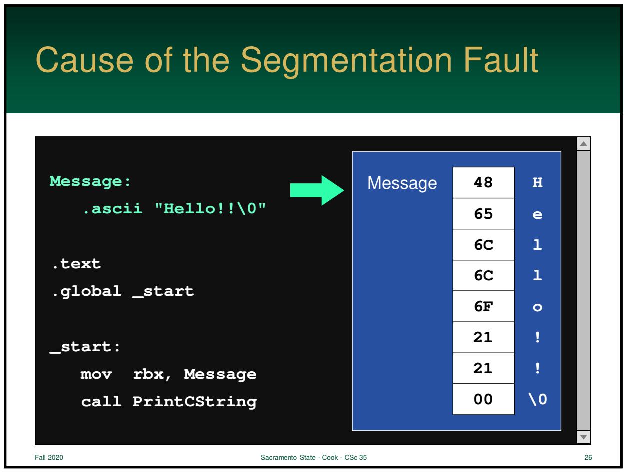
Cause of the Segmentation Fault

```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global _start
_start:
    mov  rbx, Message
    call PrintCString
```

Creates 8 bytes using ASCII values

Used mov rather than lea. rbx is 64-bit (8 bytes)

25

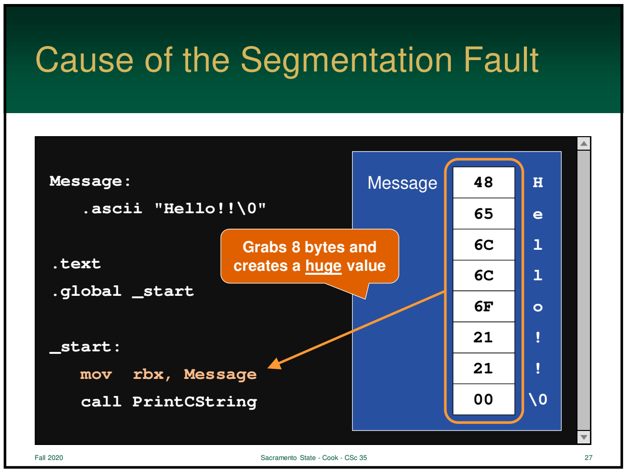


Cause of the Segmentation Fault

```
Message:
    .ascii "Hello!!\0"

.text
.global _start

_start:
    mov  rbx, Message
    call PrintCString
```

Message		
	48	H
	65	e
	6C	l
	6C	l
	6F	o
	21	!
	21	!
	00	\0

26



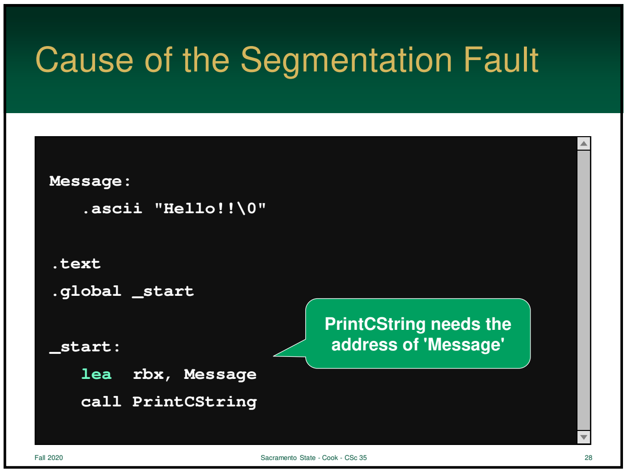
Cause of the Segmentation Fault

```
Message:
    .ascii "Hello!!\0"

.text
.global _start

_start:
    mov  rbx, Message
    call PrintCString
```

Grabs 8 bytes and creates a huge value

Message		
	48	H
	65	e
	6C	l
	6C	l
	6F	o
	21	!
	21	!
	00	\0

27

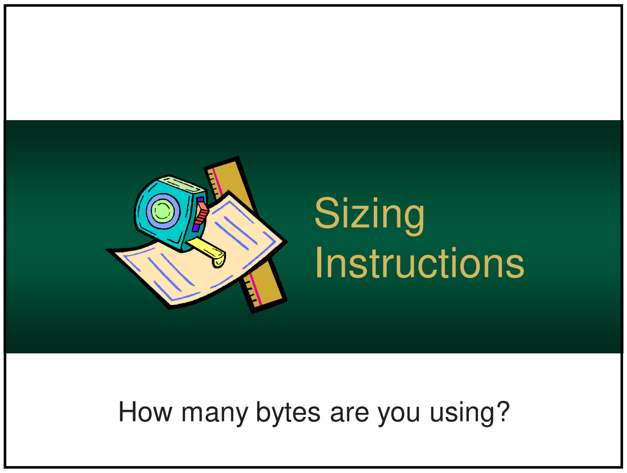


Cause of the Segmentation Fault

```
Message:
    .ascii "Hello!!\0"

.text
.global _start

_start:
    lea  rbx, Message
    call PrintCString
```

PrintCString needs the address of 'Message'

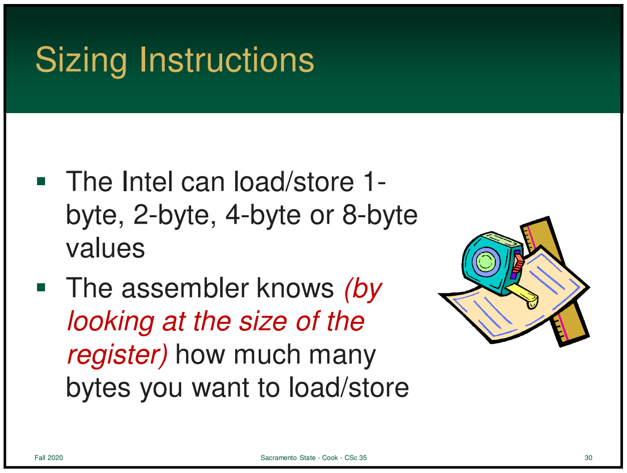
28



Sizing Instructions

How many bytes are you using?

29



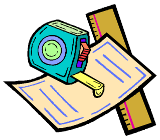
Sizing Instructions

- The Intel can load/store 1-byte, 2-byte, 4-byte or 8-byte values
- The assembler knows *(by looking at the size of the register)* how much many bytes you want to load/store

30

Sizing Instructions

- However, sometimes the number of bytes (1, 2, etc..) can't be determined
- In this case, the assembler will report an error
- … since it doesn't know how to encode the instruction



Example: How Many Bytes?

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global _start
_start:
    mov  total, 50
```

total is a target address. It doesn't have any implied size.

Example: How Many Bytes?

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global _start
_start:
    mov  total, 50
```

How many bytes is this? The value 50 can be stored in 1, 2, 4, or 8 bytes.

How Many Bytes?

- If it is not obvious to the assembler how many bytes you want to access, it will report *"ambiguous operand size"*
- To address this issue…
  - GAS assembly allows you places a single character after the instruction's mnemonic
  - this suffix will tell the assembler how many bytes will be accessed during the operation

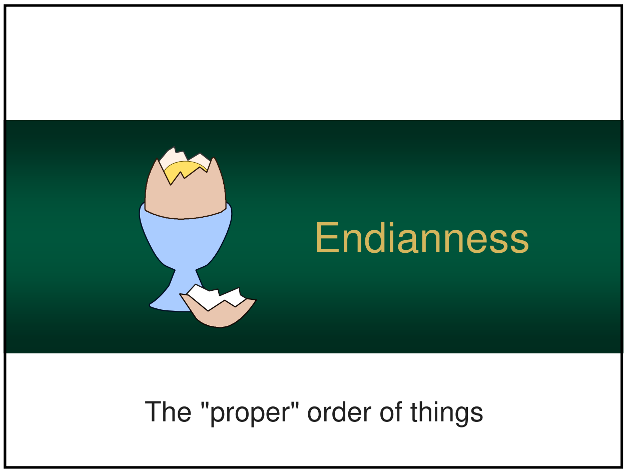
How Many Bytes

Suffix	Name	Size
b	byte	1 byte
s	short	2 bytes
l	long	4 bytes
q	quad	8 bytes

Example: Suffix Used

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global _start
_start:
    movq total, 50
```

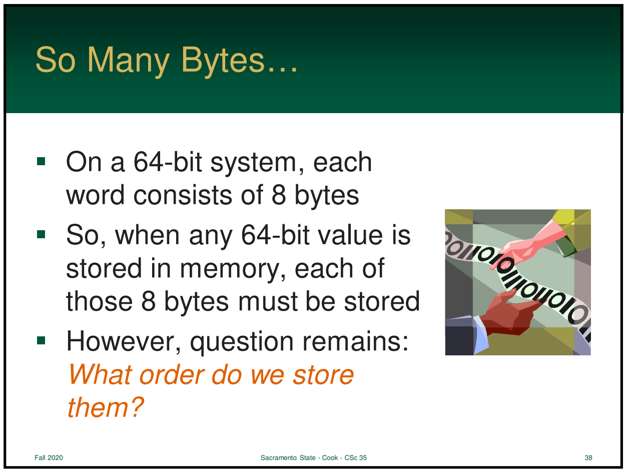
Now the assembler knows you mean "move quad".



Endianness

The "proper" order of things

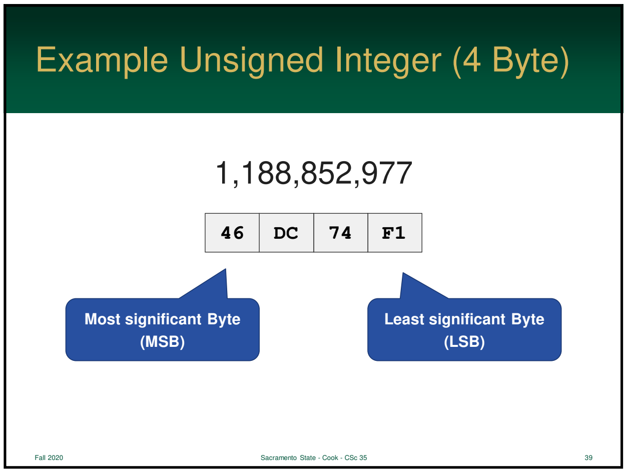
37



So Many Bytes…

- On a 64-bit system, each word consists of 8 bytes
- So, when any 64-bit value is stored in memory, each of those 8 bytes must be stored
- However, question remains: *What order do we store them?*

38



Example Unsigned Integer (4 Byte)

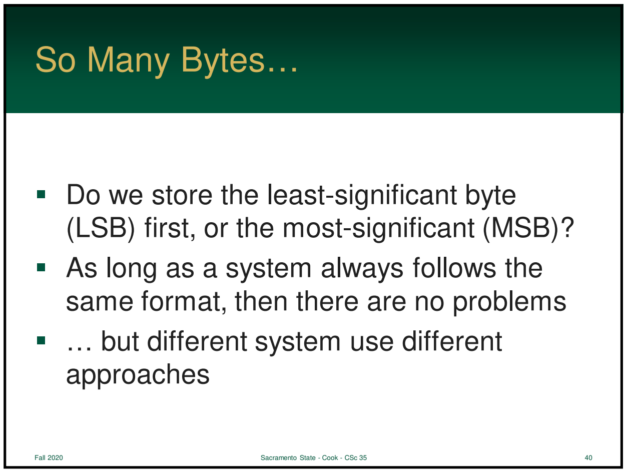
1,188,852,977

46	DC	74	F1

Most significant Byte (MSB)

Least significant Byte (LSB)

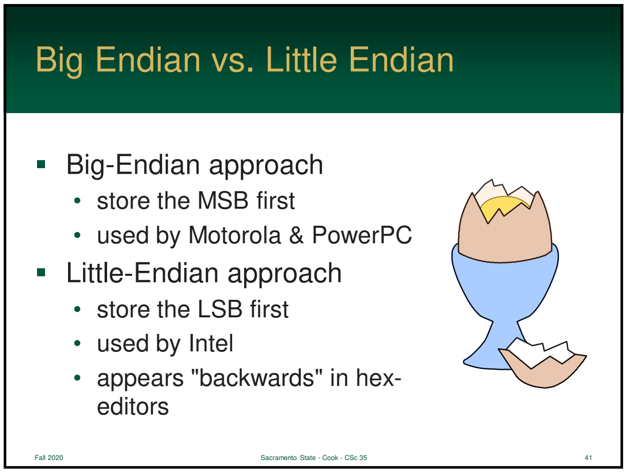
39



So Many Bytes…

- Do we store the least-significant byte (LSB) first, or the most-significant (MSB)?
- As long as a system always follows the same format, then there are no problems
- … but different system use different approaches

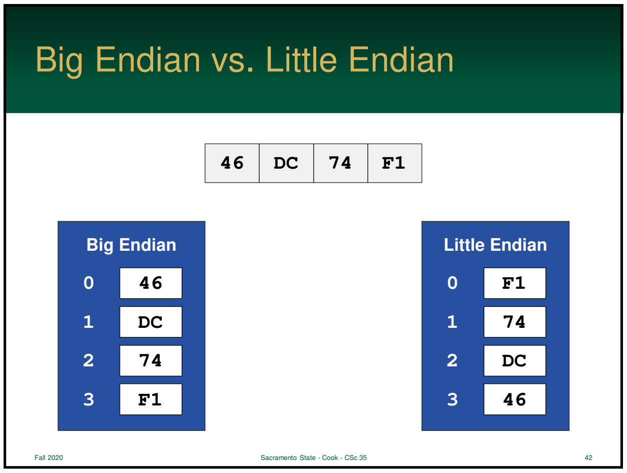
40



Big Endian vs. Little Endian

- Big-Endian approach
  - store the MSB first
  - used by Motorola & PowerPC
- Little-Endian approach
  - store the LSB first
  - used by Intel
  - appears "backwards" in hex-editors

41



Big Endian vs. Little Endian

46	DC	74	F1

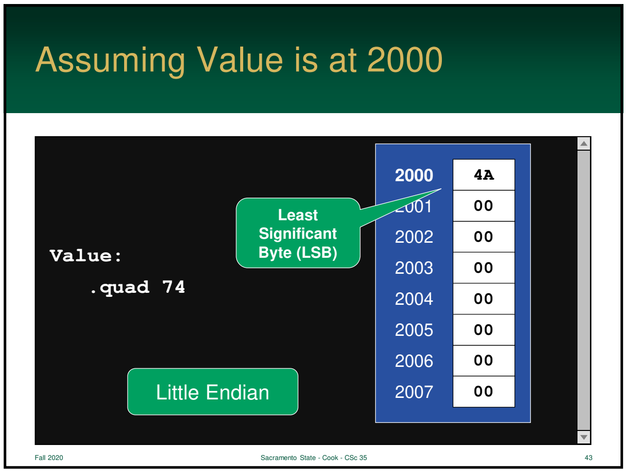
Big Endian

0	46
1	DC
2	74
3	F1

Little Endian

0	F1
1	74
2	DC
3	46

42

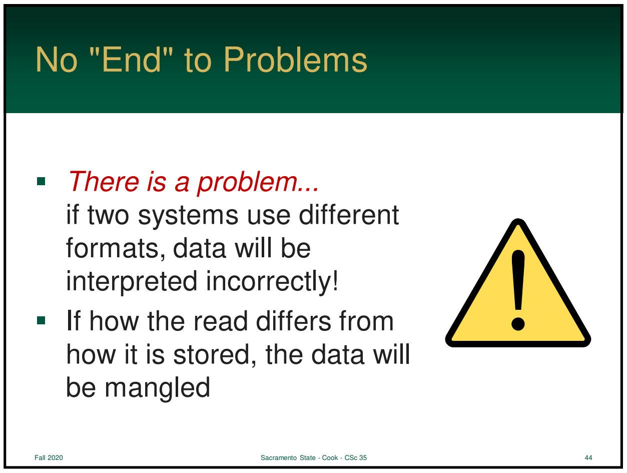


Assuming Value is at 2000

```
Value:
    .quad 74
```

2000	4A
2001	00
2002	00
2003	00
2004	00
2005	00
2006	00
2007	00

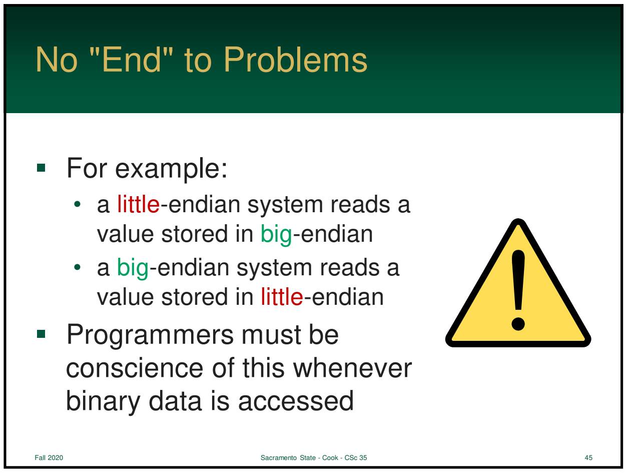
43



No "End" to Problems

- *There is a problem...* if two systems use different formats, data will be interpreted incorrectly!
- If how the read differs from how it is stored, the data will be mangled

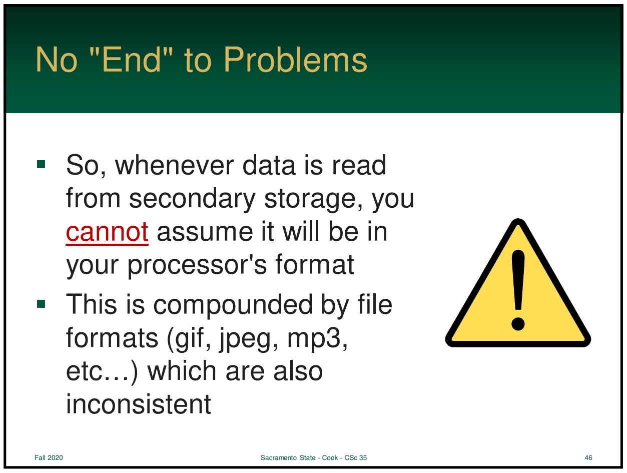
44



No "End" to Problems

- For example:
  - a little-endian system reads a value stored in big-endian
  - a big-endian system reads a value stored in little-endian
- Programmers must be conscience of this whenever binary data is accessed

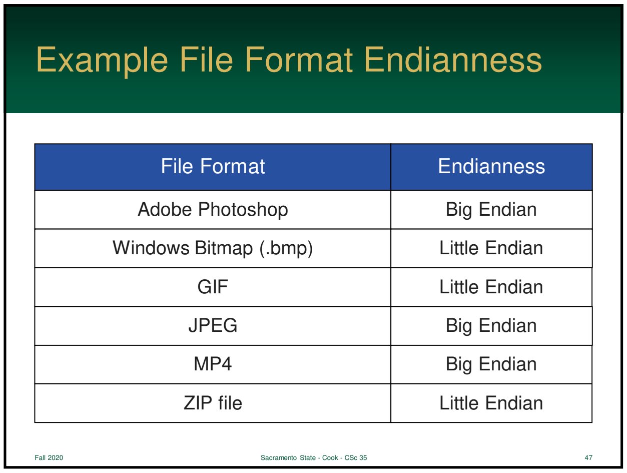
45



No "End" to Problems

- So, whenever data is read from secondary storage, you cannot assume it will be in your processor's format
- This is compounded by file formats (gif, jpeg, mp3, etc…) which are also inconsistent

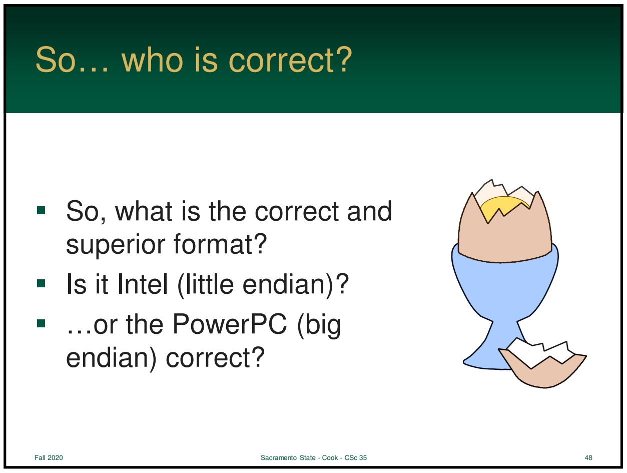
46



Example File Format Endianness

File Format	Endianness
Adobe Photoshop	Big Endian
Windows Bitmap (.bmp)	Little Endian
GIF	Little Endian
JPEG	Big Endian
MP4	Big Endian
ZIP file	Little Endian

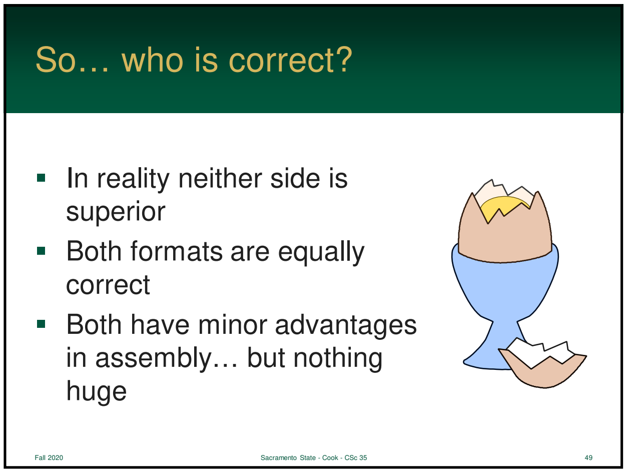
47



So… who is correct?

- So, what is the correct and superior format?
- Is it Intel (little endian)?
- …or the PowerPC (big endian) correct?

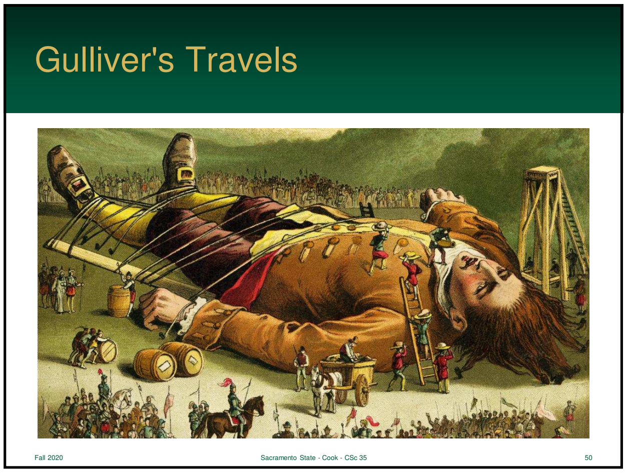
48



So… who is correct?

- In reality neither side is superior
- Both formats are equally correct
- Both have minor advantages in assembly… but nothing huge

49



Gulliver's Travels

50