Addressing

Part 6

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

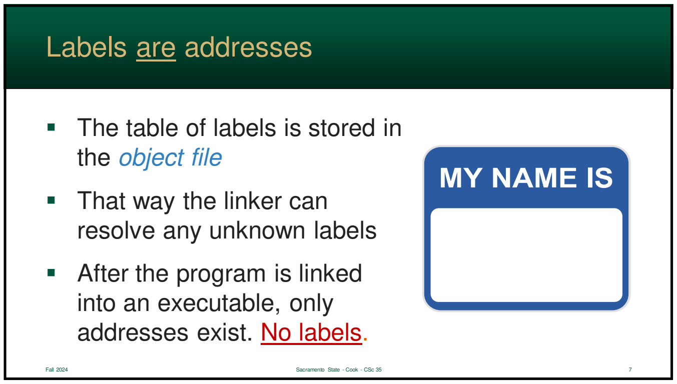
Directive	What it does
`.ascii`	Allocate enough space to store an ASCII string
`.quad`	Allocate 8-byte blocks with initial value(s)
`.byte`	Allocate byte(s) with initial value(s)
`.space`	Allocate any ***size*** of empty bytes (with initial values).

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

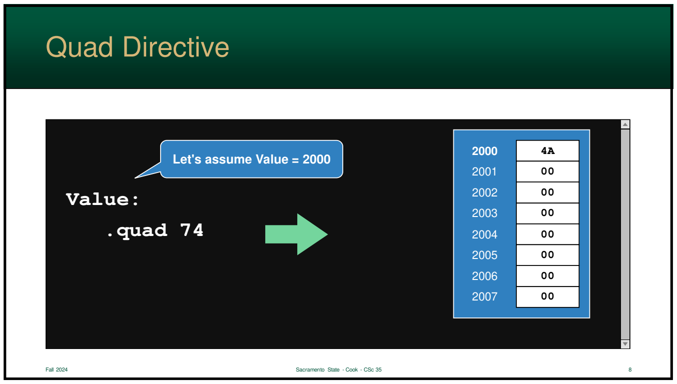
6



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



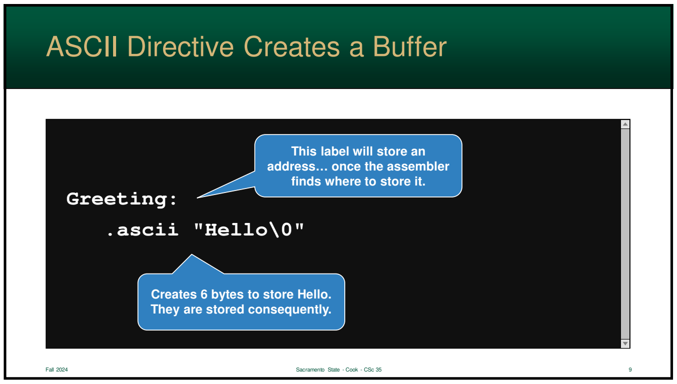
Quad Directive

```
Value:
    .quad 74
```

Let's assume Value = 2000

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

8



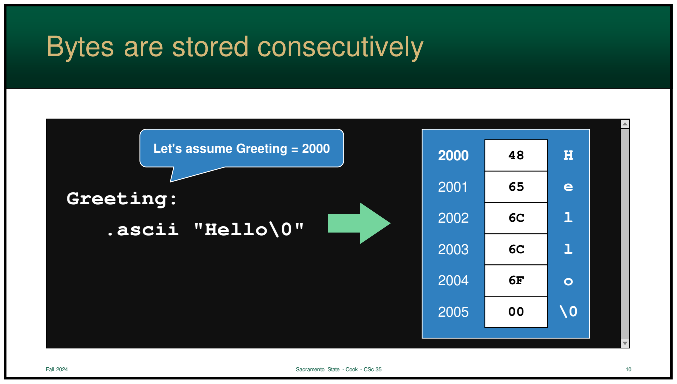
ASCII Directive Creates a Buffer

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

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

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

9



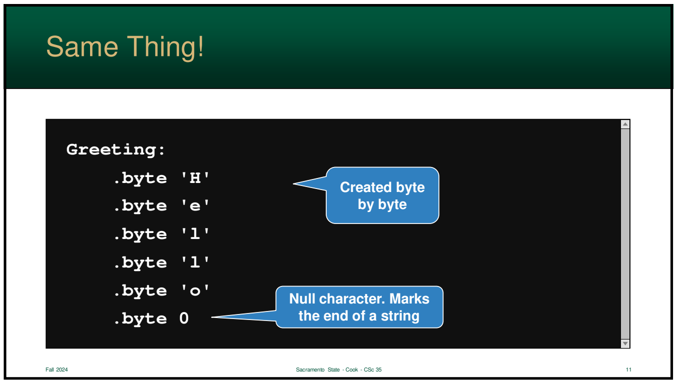
Bytes are stored consecutively

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

Let's assume Greeting = 2000

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

10



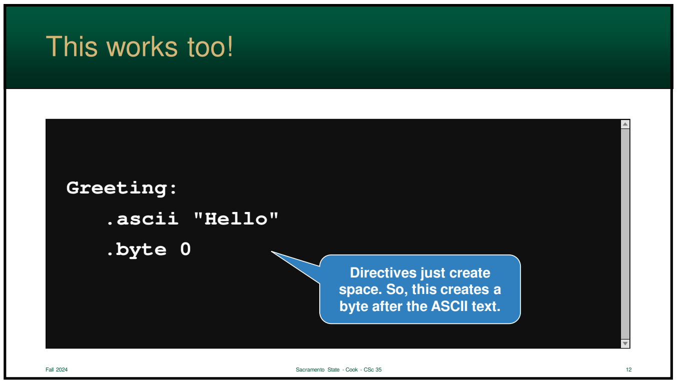
Same Thing!

```
Greeting:
    .byte 'H'
    .byte 'e'
    .byte 'l'
    .byte 'l'
    .byte 'o'
    .byte 0
```

Created byte by byte

Null character. Marks the end of a string

11



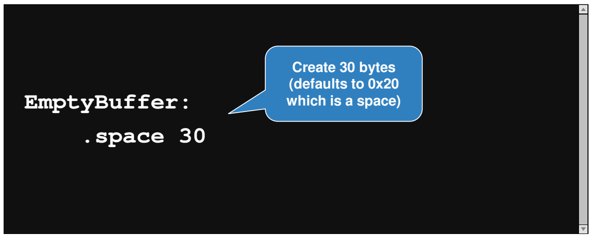
This works too!

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

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

12

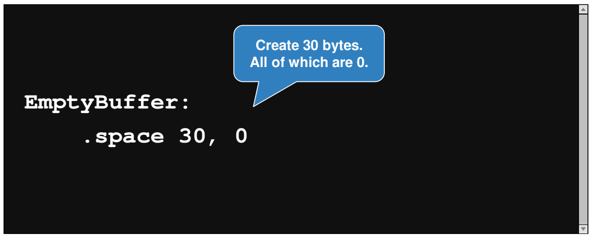
Create a Buffer of Any Size



```
EmptyBuffer:
     .space 30
```

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

Create a Buffer of Any Size



```
EmptyBuffer:
     .space 30, 0
```

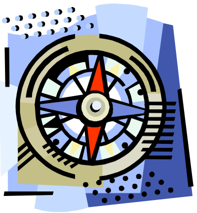
Create 30 bytes. All of which are 0.

Addressing Modes Basics

How to interact with memory

Addressing Modes

- Processor instructions often need to access memory to read values and store results
- So far, we have used registers to read and store single values
- However, we need to:
  - access items in an array
  - follow pointers
  - and more!

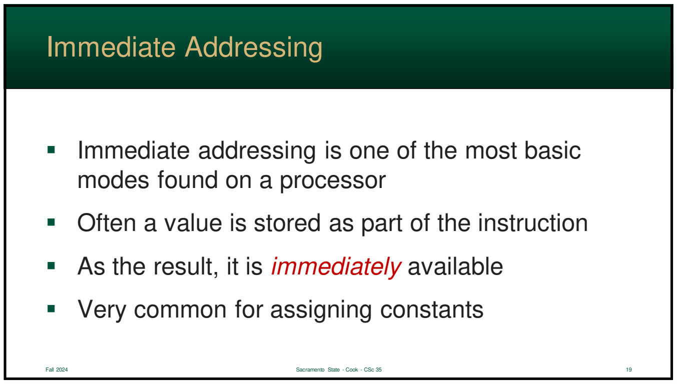


Addressing Modes

- *How* the processor can locate and read data is called an *addressing mode*
- Information combined from registers, immediates, etc… to create a target address
- Modes vary greatly between processors

4 Basic Addressing Modes

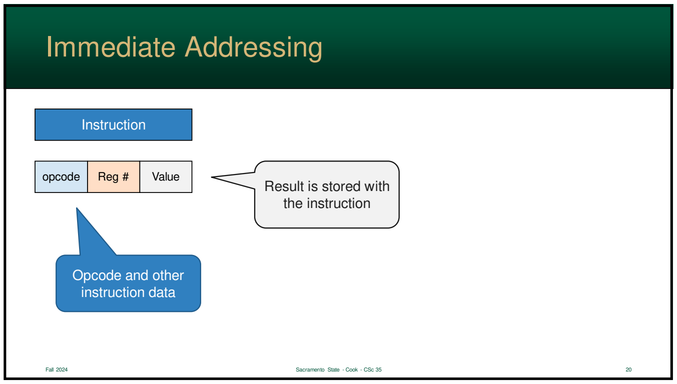
- Immediate Addressing
- Register Addressing
- Direct Addressing
- Indirect Addressing



Immediate Addressing

- Immediate addressing is one of the most basic modes found on a processor
- Often a value is stored as part of the instruction
- As the result, it is *immediately* available
- Very common for assigning constants

19

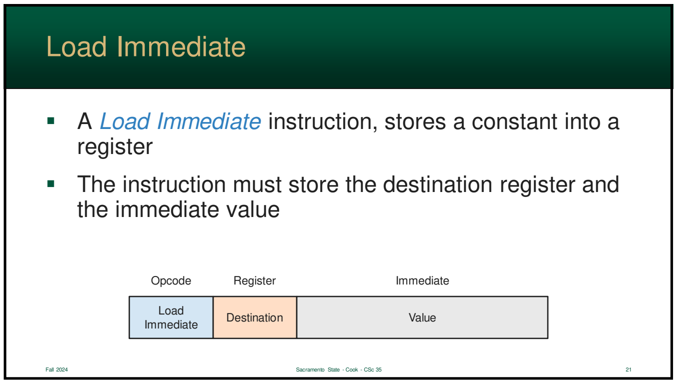


Immediate Addressing

Result is stored with the instruction

Opcode and other instruction data

20

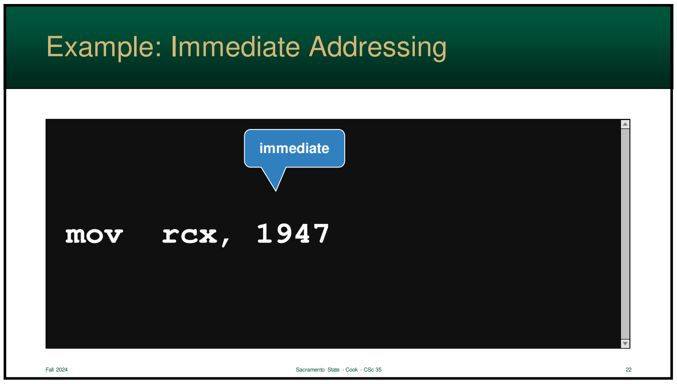


Load Immediate

- A *Load Immediate* instruction, stores a constant into a register
- The instruction must store the destination register and the immediate value

Opcode	Register	Immediate
Load Immediate	Destination	Value

21

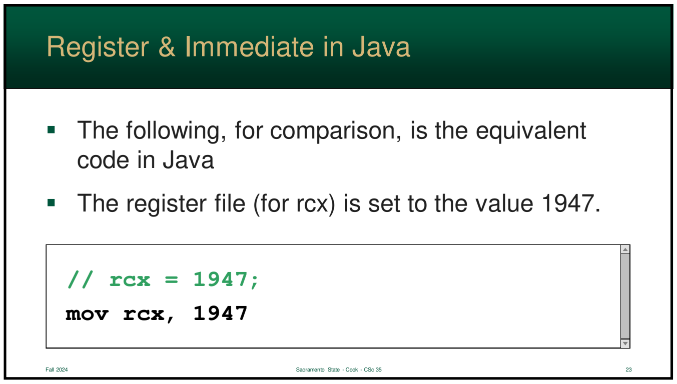


Example: Immediate Addressing

immediate

```
mov   rcx,  1947
```

22

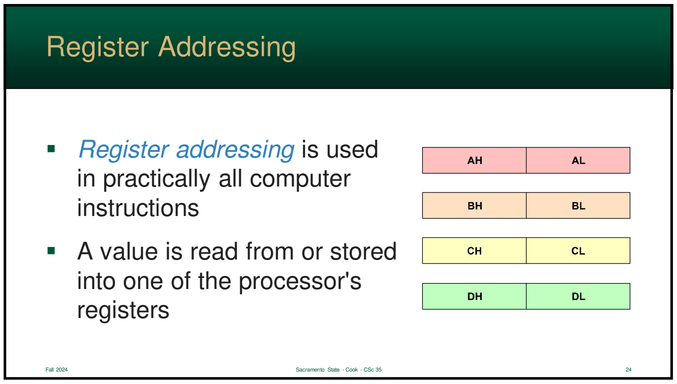


Register & Immediate in Java

- The following, for comparison, is the equivalent code in Java
- The register file (for rcx) is set to the value 1947.

```
// rcx = 1947;
mov rcx, 1947
```

23

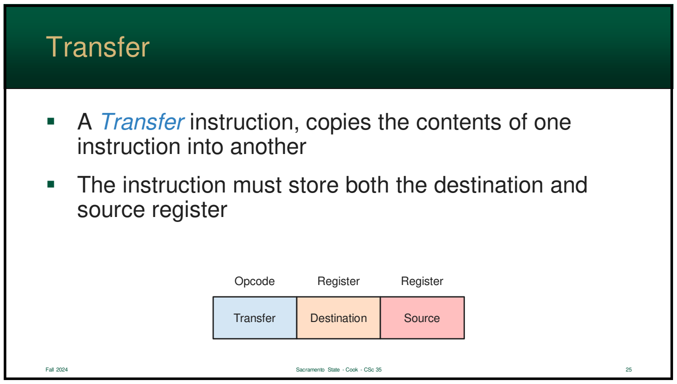


Register Addressing

- *Register addressing* is used in practically all computer instructions
- A value is read from or stored into one of the processor's registers

AH	AL
BH	BL
CH	CL
DH	DL

24

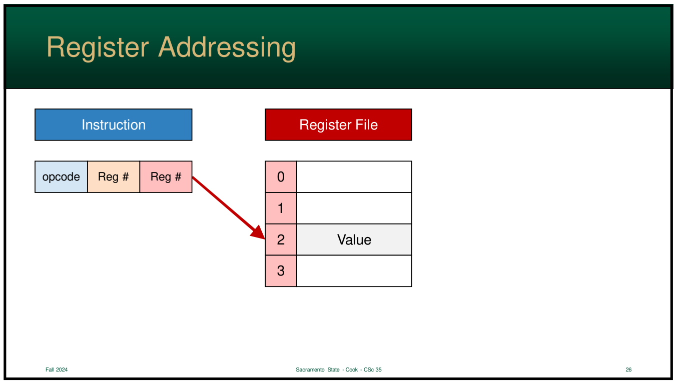


Transfer

- A *Transfer* instruction, copies the contents of one instruction into another
- The instruction must store both the destination and source register

Opcode	Register	Register
Transfer	Destination	Source

25



Register Addressing

Instruction

opcode	Reg #	Reg #

Register File

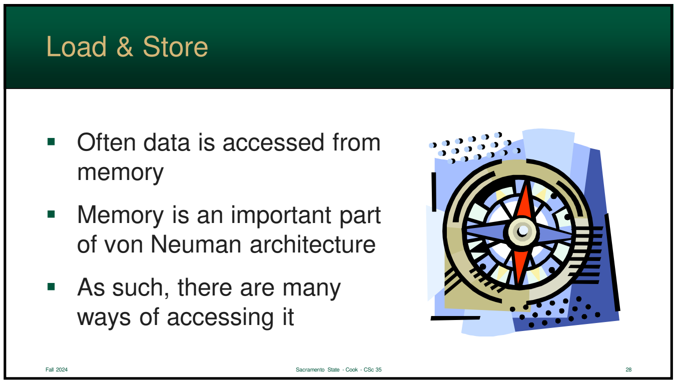
0	
1	
2	Value
3	

26

Load & Store

Saving and retrieving values

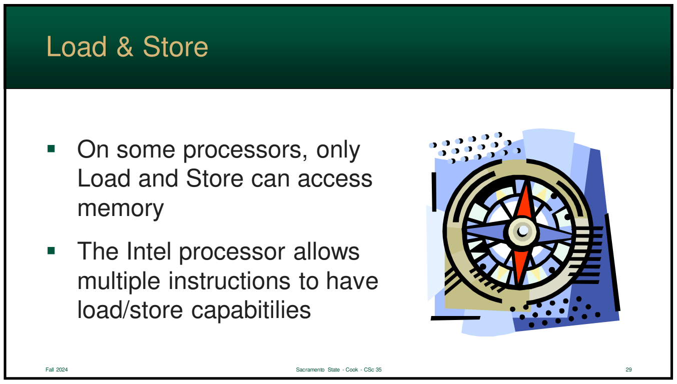
27



Load & Store

- Often data is accessed from memory
- Memory is an important part of von Neuman architecture
- As such, there are many ways of accessing it

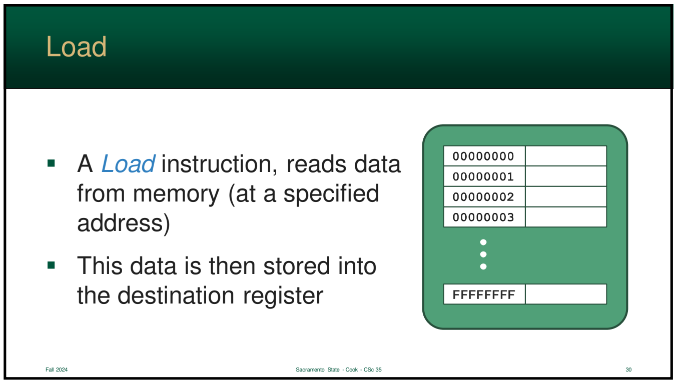
28



Load & Store

- On some processors, only Load and Store can access memory
- The Intel processor allows multiple instructions to have load/store capabitilies

29

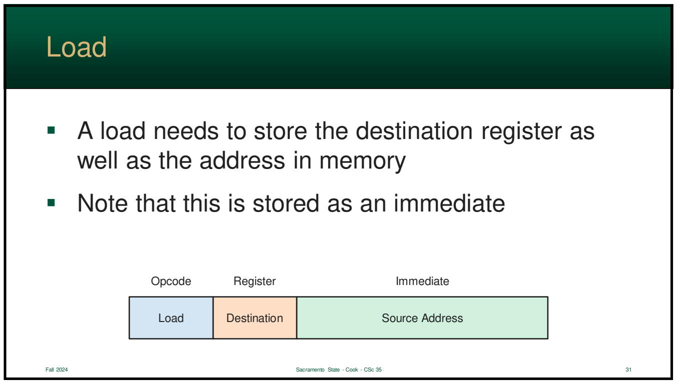


Load

- A *Load* instruction, reads data from memory (at a specified address)
- This data is then stored into the destination register

00000000	
00000001	
00000002	
00000003	
⋮	
FFFFFFFF	

30

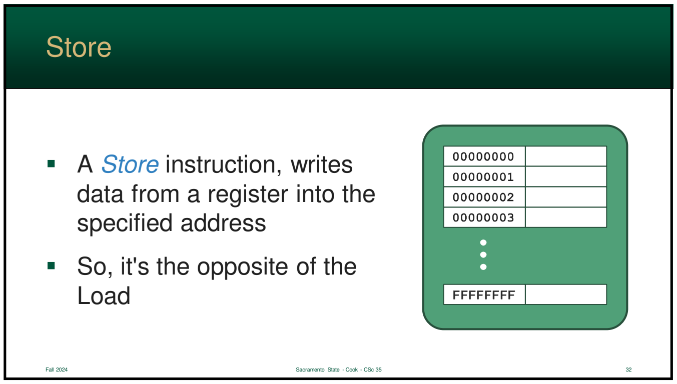


Load

- A load needs to store the destination register as well as the address in memory
- Note that this is stored as an immediate

Opcode	Register	Immediate
Load	Destination	Source Address

31

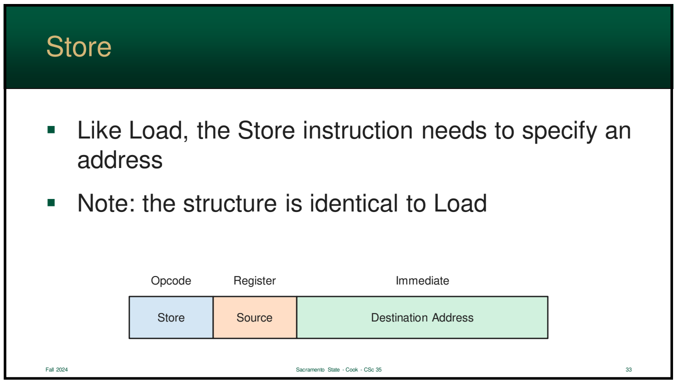


Store

- A *Store* instruction, writes data from a register into the specified address
- So, it's the opposite of the Load

00000000	
00000001	
00000002	
00000003	
⋮	
FFFFFFFF	

32

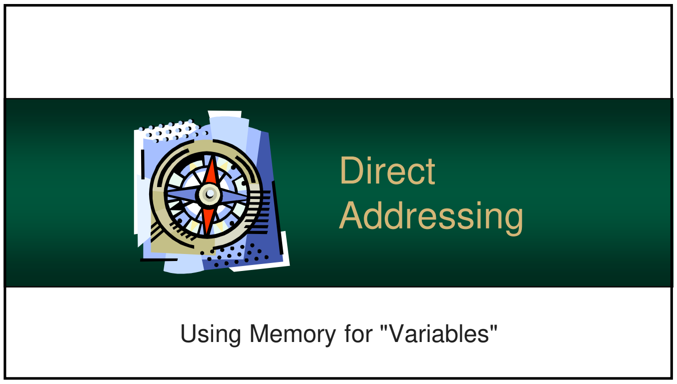


Store

- Like Load, the Store instruction needs to specify an address
- Note: the structure is identical to Load

Opcode	Register	Immediate
Store	Source	Destination Address

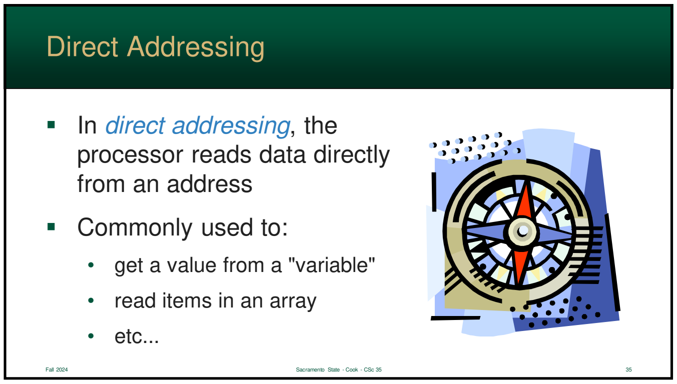
33



Direct Addressing

Using Memory for "Variables"

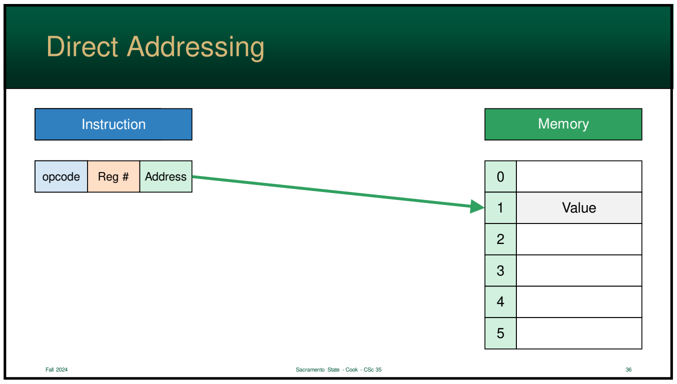
34



Direct Addressing

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

35



Direct Addressing

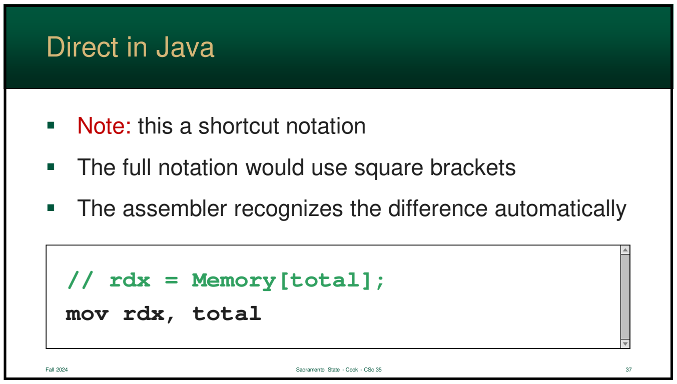
Instruction

opcode	Reg #	Address

Memory

0	
1	Value
2	
3	
4	
5	

36

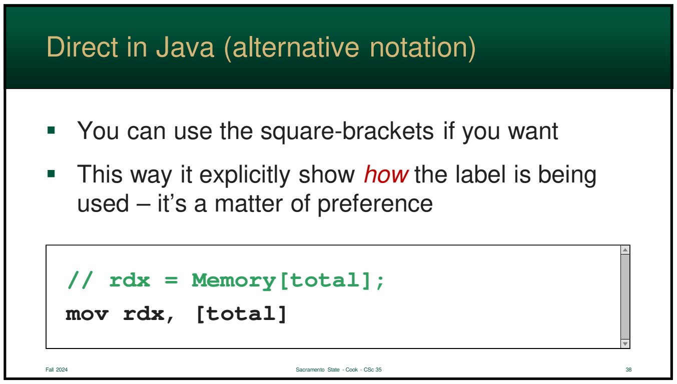


Direct in Java

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

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

37

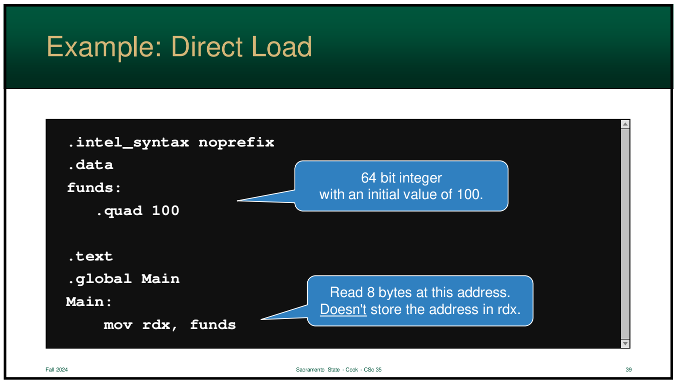


Direct in Java (alternative notation)

- 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

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

38



Example: Direct Load

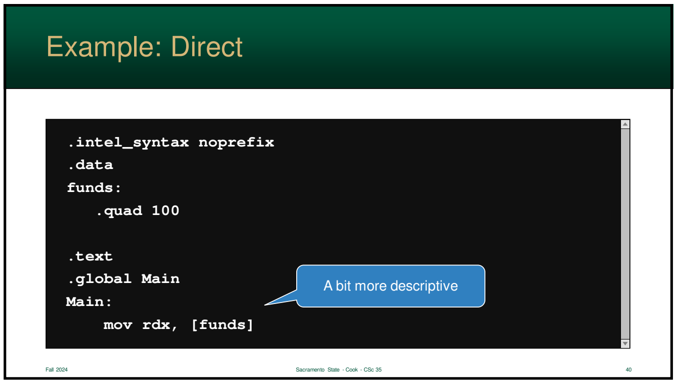
```
.intel_syntax noprefix
.data
funds:
   .quad 100

.text
.global Main
Main:
    mov rdx, funds
```

64 bit integer
with an initial value of 100.

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

39



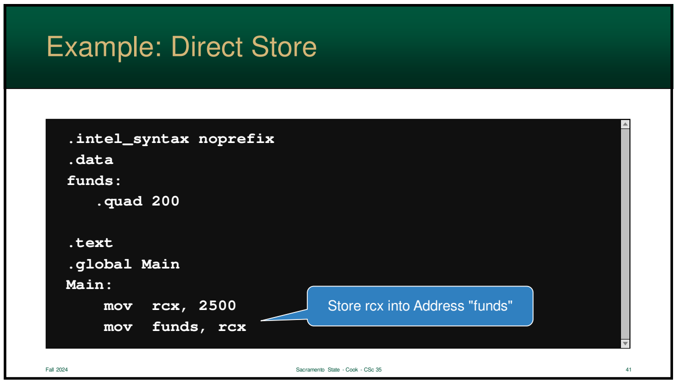
Example: Direct

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

.text
.global Main
Main:
    mov rdx, [funds]
```

A bit more descriptive

40



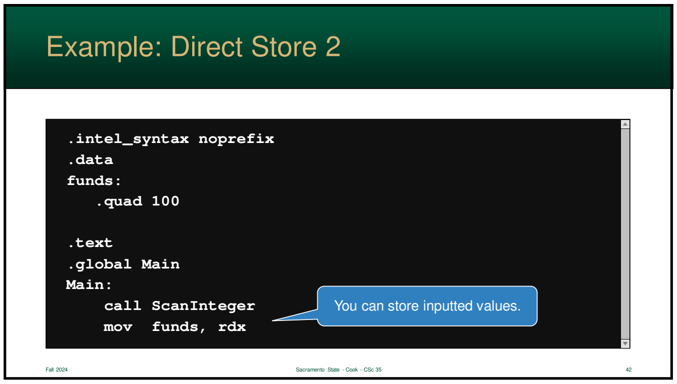
Example: Direct Store

```
.intel_syntax noprefix
.data
funds:
   .quad 200

.text
.global Main
Main:
    mov  rcx, 2500
    mov  funds, rcx
```

Store rcx into Address "funds"

41



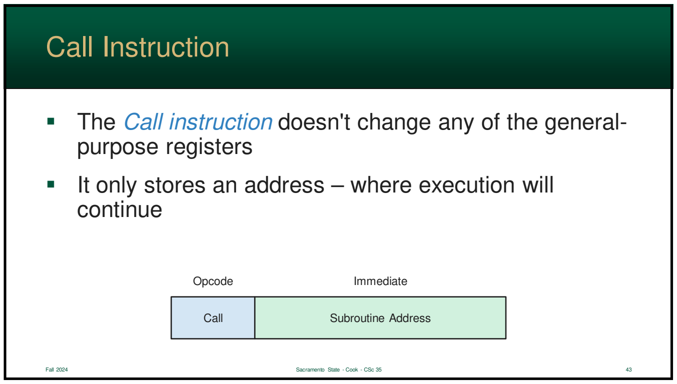
Example: Direct Store 2

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

.text
.global Main
Main:
    call ScanInteger
    mov  funds, rdx
```

You can store inputted values.

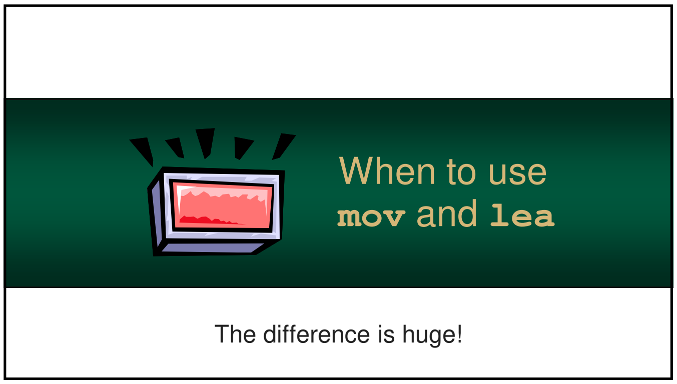
42



Call Instruction

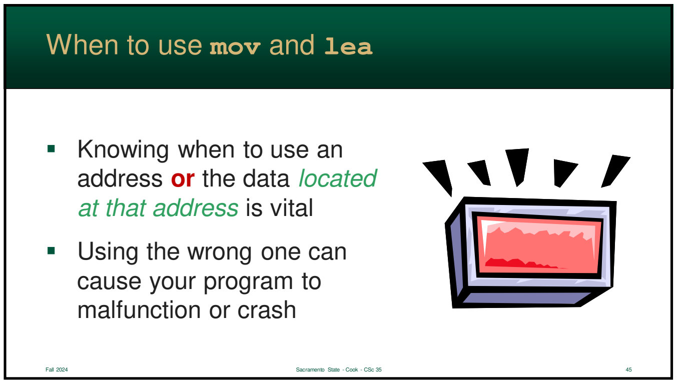
- The *Call instruction* doesn't change any of the general-purpose registers
- It only stores an address – where execution will continue

Opcode	Immediate
Call	Subroutine Address



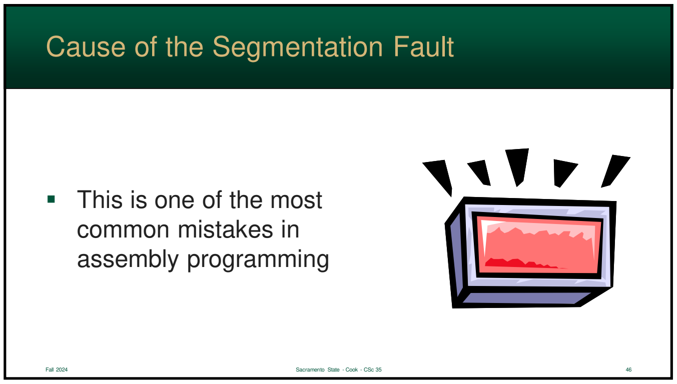
When to use **mov** and **lea**

The difference is huge!



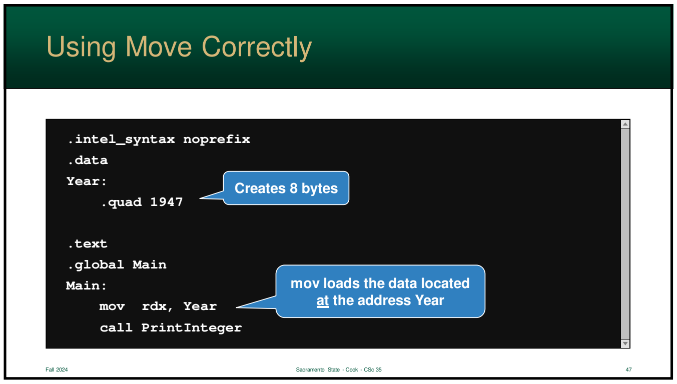
When to use **mov** and **lea**

- Knowing when to use an address **or** the data *located at that address* is vital
- Using the wrong one can cause your program to malfunction or crash



Cause of the Segmentation Fault

- This is one of the most common mistakes in assembly programming



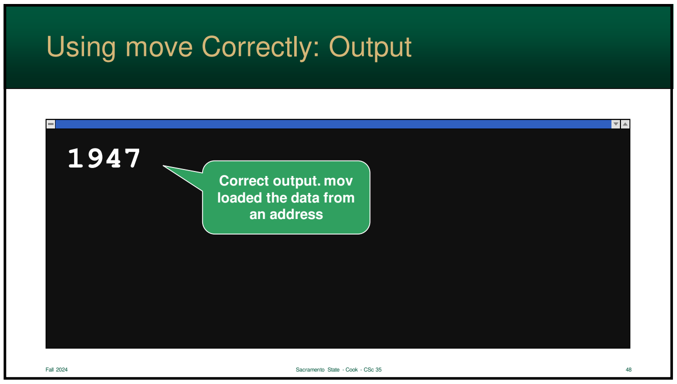
Using Move Correctly

```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global Main
Main:
    mov  rdx, Year
    call PrintInteger
```

Creates 8 bytes

mov loads the data located at the address Year

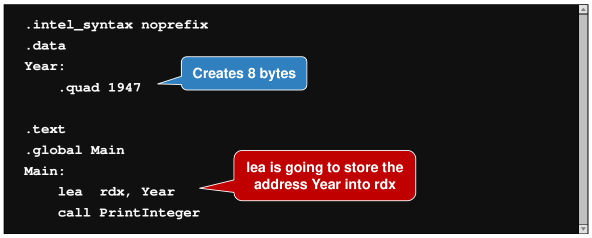


Using move Correctly: Output

```
1947
```

Correct output. mov loaded the data from an address

Using `lea` by accident



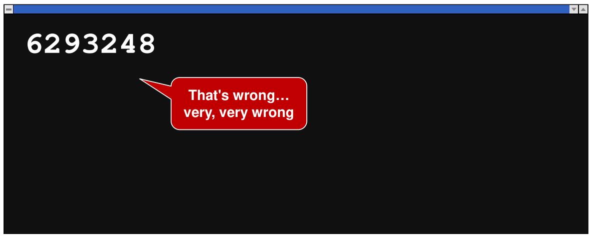
```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global Main
Main:
    lea  rdx, Year
    call PrintInteger
```

Creates 8 bytes

lea is going to store the address Year into rdx

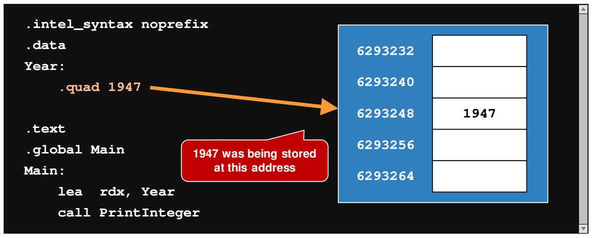
Using `lea` by accident



```
6293248
```

That's wrong… very, very wrong

Why it Failed



```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global Main
Main:
    lea  rdx, Year
    call PrintInteger
```

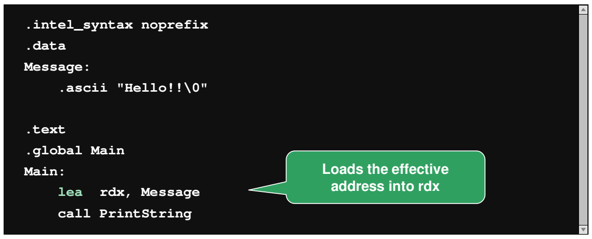
6293232	
6293240	
6293248	1947
6293256	
6293264	

1947 was being stored at this address

Sometimes, You Need the Address

- Of course, sometimes, you do need an address
- For example, PrintString
  - needs to know where the string is located so it can print a series of characters
  - so, it requires an address
  - `lea` is necessary

Using `lea` correctly

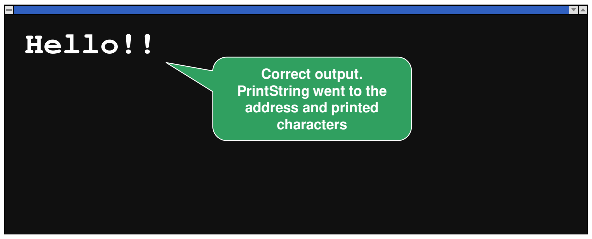


```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    lea  rdx, Message
    call PrintString
```

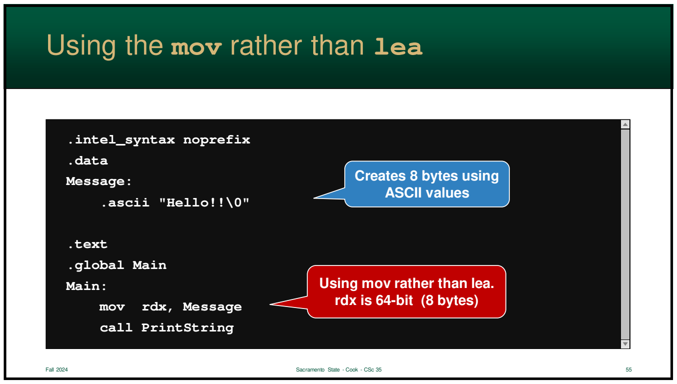
Loads the effective address into rdx

Using `lea` correctly: Output



```
Hello!!
```

Correct output. PrintString went to the address and printed characters



Using the **mov** rather than **lea**

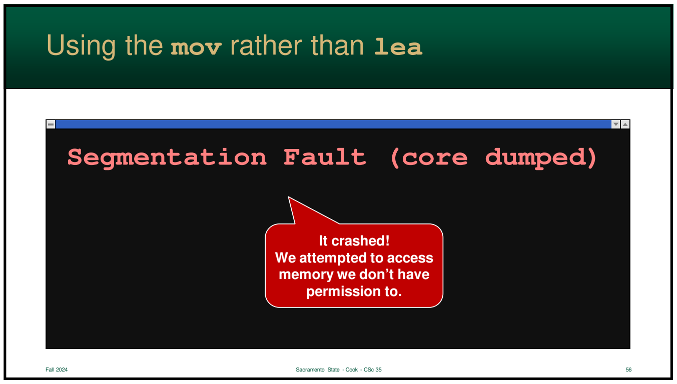
```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    mov  rdx, Message
    call PrintString
```

Creates 8 bytes using ASCII values

Using mov rather than lea. rdx is 64-bit (8 bytes)

55

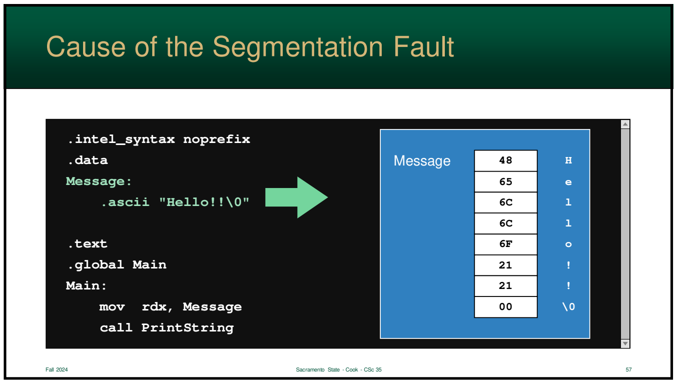


Using the **mov** rather than **lea**

```
Segmentation Fault (core dumped)
```

It crashed! We attempted to access memory we don't have permission to.

56



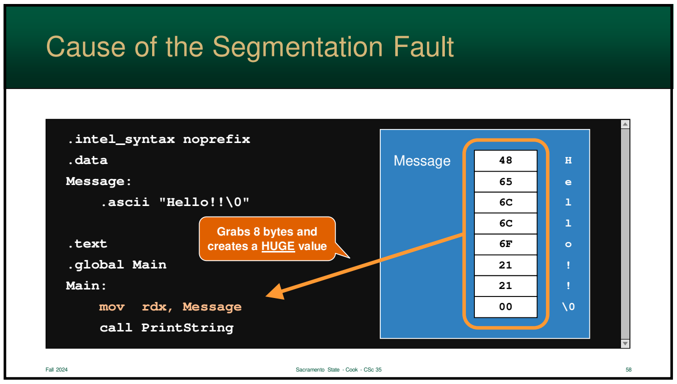
Cause of the Segmentation Fault

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

.text
.global Main
Main:
    mov  rdx, Message
    call PrintString
```

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

57



Cause of the Segmentation Fault

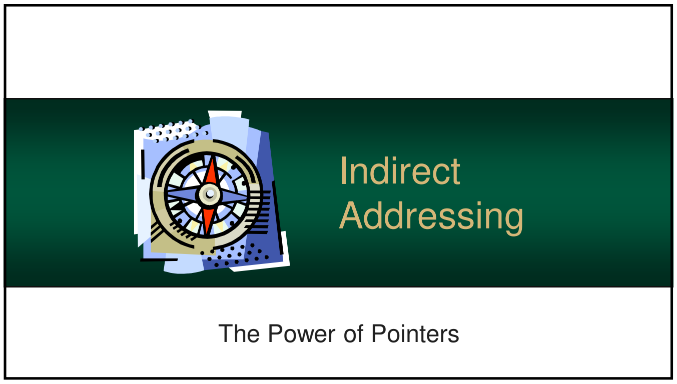
```
.intel_syntax noprefix
.data
Message:
    .ascii "Hello!!\0"

.text
.global Main
Main:
    mov  rdx, Message
    call PrintString
```

Grabs 8 bytes and creates a HUGE value

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

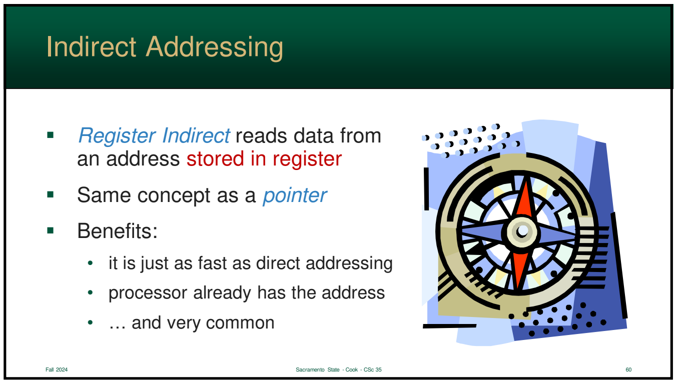
58



Indirect Addressing

The Power of Pointers

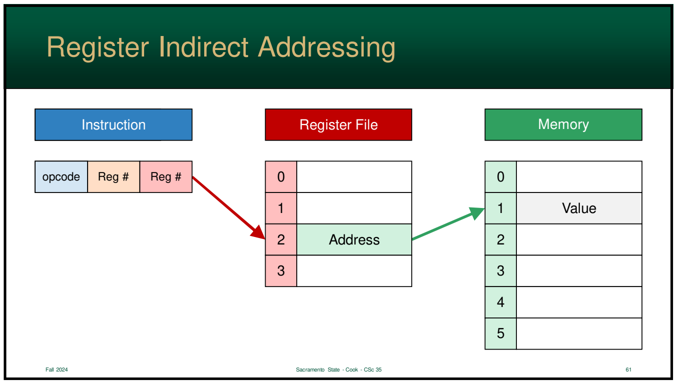
59



Indirect Addressing

- *Register Indirect* reads data from an address stored in register
- Same concept as a *pointer*
- Benefits:
  - it is just as fast as direct addressing
  - processor already has the address
  - … and very common

60



Register Indirect Addressing

Instruction

opcode	Reg #	Reg #

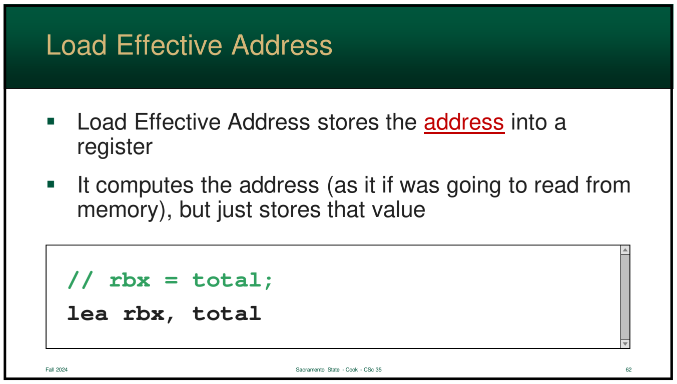
Register File

0	
1	
2	Address
3	

Memory

0	
1	Value
2	
3	
4	
5	

61

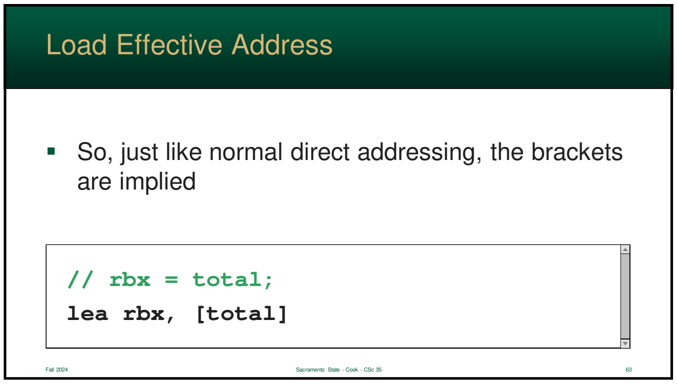


Load Effective Address

- Load Effective Address stores the address into a register
- It computes the address (as it if was going to read from memory), but just stores that value

```
// rbx = total;
lea rbx, total
```

62

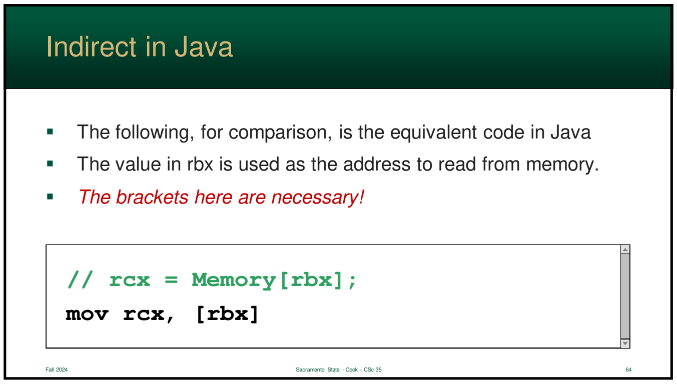


Load Effective Address

- So, just like normal direct addressing, the brackets are implied

```
// rbx = total;
lea rbx, [total]
```

63

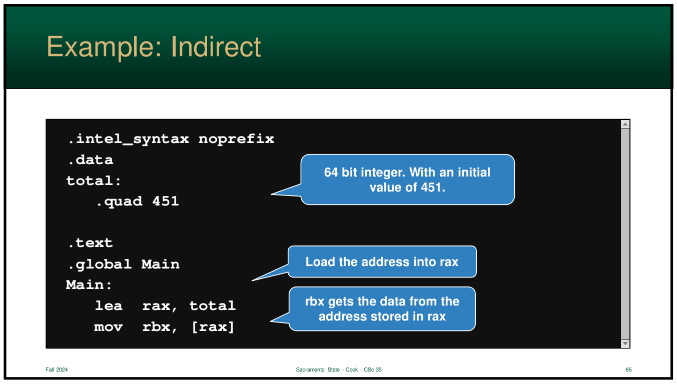


Indirect in Java

- The following, for comparison, is the equivalent code in Java
- The value in rbx is used as the address to read from memory.
- *The brackets here are necessary!*

```
// rcx = Memory[rbx];
mov rcx, [rbx]
```

64



Example: Indirect

```
.intel_syntax noprefix
.data
total:
    .quad 451

.text
.global Main
Main:
    lea  rax, total
    mov  rbx, [rax]
```

64 bit integer. With an initial value of 451.

Load the address into rax

rbx gets the data from the address stored in rax

65