



Buffers & Direct Storage

Part 5A

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....



Fall 2020

Sacramento State - Cook - CSc 35

3

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

Fall 2020

Sacramento State - Cook - CSc 35

4

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).

Fall 2020

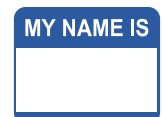
Sacramento State - Cook - CSc 35

5

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



Fall 2020

Sacramento State - Cook - CSc 35

6

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.

MY NAME IS



Fall 2020 Sacramento State - Cook - CSc 35 7

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

Fall 2020 Sacramento State - Cook - CSc 35 8

8

ASCII Directive Creates a Buffer

Text:

.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 consecutively.

Fall 2020 Sacramento State - Cook - CSc 35 9

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

Fall 2020 Sacramento State - Cook - CSc 35 10

10

Same Thing!

Text:

.byte 'H'
.byte 'e'
.byte 'l'
.byte 'l'
.byte 'o'
.byte '\0'

Created byte by byte

Fall 2020 Sacramento State - Cook - CSc 35 11

11

This works too!

Text:
.ascii "Hello"
.byte 0

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

Fall 2020 Sacramento State - Cook - CSc 35 12

12

Create a Buffer of Any Size

Text :

.space 30

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

Fall 2020

Sacramento State - Cook - CSc 35

13

Create a Buffer of Any Size

Text :

.space 30, 0

Create 30 bytes.
All of which are 0.

Fall 2020

Sacramento State - Cook - CSc 35

14



Direct Addressing

Using memory... finally

15

Direct Addressing

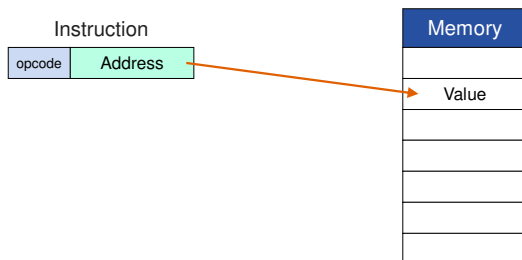
- In *direct addressing*, the processor reads data directly from the an address
- Commonly used to:
 - get a value from a "variable"
 - read items in an array
 - etc...

Fall 2020

Sacramento State - Cook - CSc 35

16

Direct Addressing



Fall 2020

Sacramento State - Cook - CSc 35

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
```

Fall 2020

Sacramento State - Cook - CSc 35

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
```

Fall 2020 Sacramento State - Cook - CSc 35

19

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.

Fall 2020 Sacramento State - Cook - CSc 35

20

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
```

Fall 2020 Sacramento State - Cook - CSc 35

21

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]
```

Fall 2020 Sacramento State - Cook - CSc 35

22

22

Example: Direct

```
.intel_syntax noprefix  
.data  
funds:  
    .quad 100  
  
.text  
.global _start  
_start:  
    mov rax, [funds]
```

A bit more descriptive

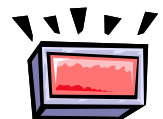
Fall 2020 Sacramento State - Cook - CSc 35

23

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



Fall 2020 Sacramento State - Cook - CSc 35

24

24

Cause of the Segmentation Fault

```
.intel_syntax noprefix
```

```
.data
```

```
Message:
```

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

Creates 8 bytes using
ASCII values

```
.text
```

```
.global _start
```

```
_start:
```

```
mov rbx, Message
```

```
call PrintCString
```

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

Fall 2020

Sacramento State - Cook - CSc 36

25

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

Fall 2020

Sacramento State - Cook - CSc 36

26

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

Fall 2020

Sacramento State - Cook - CSc 36

27

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'

Fall 2020

Sacramento State - Cook - CSc 36

28

28