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….

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

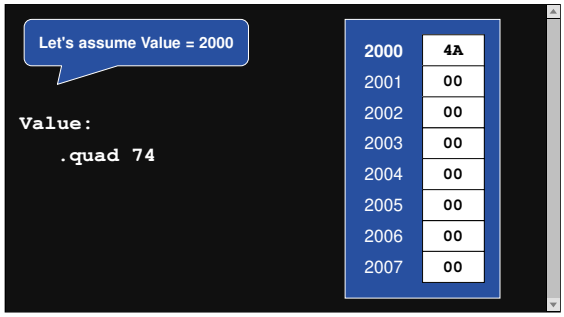
6

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.

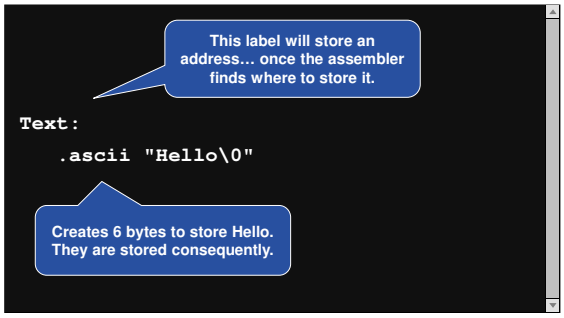
7

Quad Directive



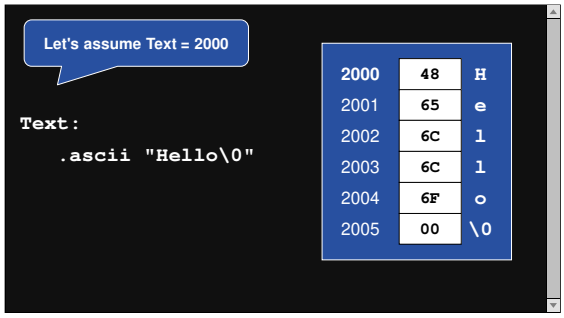
8

ASCII Directive Creates a Buffer



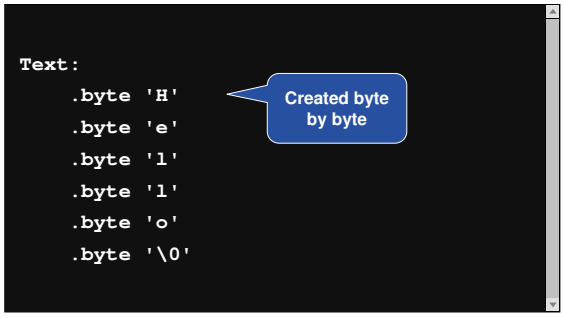
9

Bytes are stored consecutively



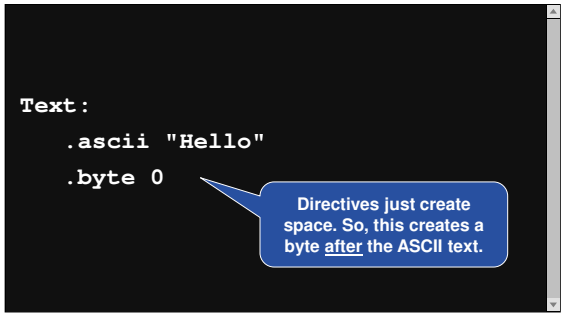
10

Same Thing!



11

This works too!



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

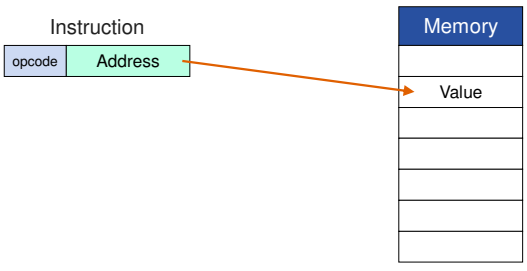
15

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

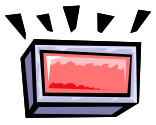
```
.intel_syntax noprefix
.data
funds:
   .quad 100

.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