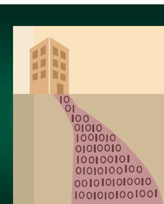
Part 6

Memory & Addressing

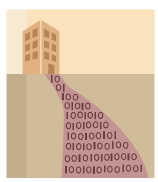


What is Memory?

Its… um…. I forgot….

Computer Memory

- Assembly offers you vast control over memory
- Understanding it…
  - is vital to becoming a great assembly programmer
  - and understanding computer architecture



What is Memory?

- Memory is essentially a long list of bytes
- Memory is sometimes referred to as *storage*
- This is because it stores both running programs and their related data

Memory	
0	01000100
1	01000011
2	01101111
3	01101111
4	01101011

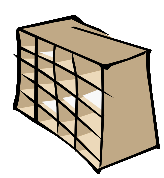
Memory Addresses

- Memory is divided into a storage locations that can hold 1 byte (8 bits) of data
- Each byte has an *address*
  - unique value that refers to that specific byte
  - used to locate the exact byte the processor wants

Memory	
0	01000100
1	01000011
2	01101111
3	01101111
4	01101011

Metaphor for Memory

- Think of memory as a *set of mailboxes*
- Each mailbox can contain a piece of data (byte)
- Each mailbox has a unique number



Metaphor for Memory

- ... or think of memory as a *group of boxes*
- Each box belongs to the same variable
- Each box has a unique number

Endianness

The "proper" order of things

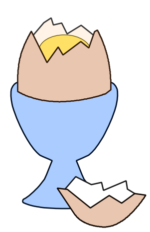
So Many Bytes…

- 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?*

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

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 editors

Example Unsigned Integer (4 Byte)

3,721,182,122

DD	CC	BB	AA

Most significant Byte

Least significant Byte

Big Endian vs. Little Endian

DD	CC	BB	AA

Big Endian	
0	DD
1	CC
2	BB
3	AA

Little Endian	
0	AA
1	BB
2	CC
3	DD

No "End" to Problems

- *There is a problem...* if two systems use different formats, data will be interpreted incorrectly!
- For example:
  - a little-endian system reads a value stored in big-endian
  - a big-endian system reads a value stored in little-endian

No "End" to Problems

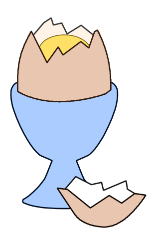
- So, whenever data is read from secondary storage, you cannot assume it will be in your processor's format
- This is compounded by file formats (gif, jpeg, mp3, etc…) which are also inconsistent

Endianness in File Formats

- Adobe Photoshop - Big Endian
- Windows Bitmap (.BMP) - Little Endian
- GIF - Little Endian
- JPEG - Big Endian

So… who is correct?

- Is the Intel x86 (little endian) or the PowerPC (big endian) correct?
- In reality…
  - neither side is correct
  - both formats are equally correct



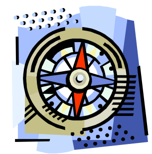
Gulliver's Travels

Addressing Modes

How to interact with memory

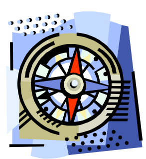
Addressing Modes

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



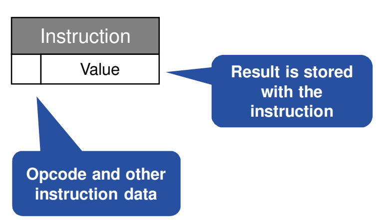
4 Basic Addressing Modes

1. Value stored in a register
2. Memory address specified in the instruction
3. Memory address pointed to by a register
4. Immediate (part of instruction after the opcode bits)

Immediate Addressing

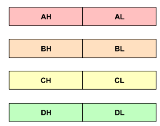
- Immediate addressing is one of the most basic modes found on a processor
- Often a value is stored as part of the instruction
- As the result, it is *immediately* available
- Very common for assigning constants

Immediate Addressing

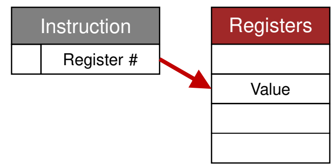


Register Addressing

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



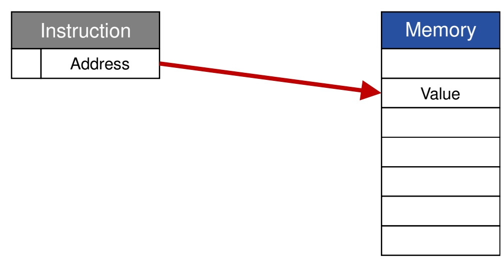
Register Addressing



Direct Addressing

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

Direct Addressing



Example: Immediate

$ means the immediate value

```
mov   $1,  %rax
```


Example: Direct

```
.data
Total:
    .quad 0

.text
.global _start
_start:
    mov   Total,  %rax
```

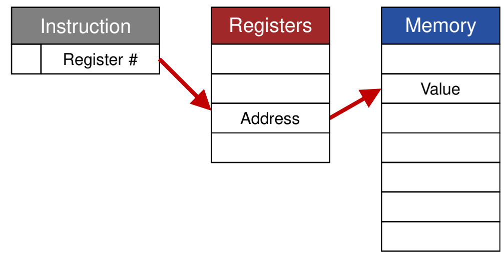
64 bit integer. With an initial value of 0.

No $. Get the 8 bytes at this address. Doesn't store *the* address in rax.

Register Indirect Addressing

- *Register Indirect* uses a register is used to store the address
- Same concept as a *pointer*
- Because the address is in a register…
  - processor does have to go to memory get it
  - it is just as fast as direct addressing
  - … and very common

Register Indirect Addressing



Relative Access

- In *relative access*, a value is added to a system register (e.g. program counter)
- Advantages:
  - instruction can just store the *difference* (in bytes) from the current instruction address
  - takes less storage than a full 64-bit address
  - it allows a program to be stored anywhere in memory – *and it will still work!*

Relative Addressing

- Often used in conditional jump statements
  - only need the to store the number of bytes to jump – either up or down
  - so, the instruction only stores the value to add to the program counter
  - practically all processors us this approach
- Also used to access local data – load/store

Behind the Scenes of Arrays

All the mystery is revealed!

Arrays

- Computers do not have an 'array' data type
- So, how do you have array variables?
- When you create an array…
  - you allocate a block of memory
  - each element (cell) is located sequentially in memory – one right after each other

Arrays

- Every byte in memory has an address
  - … as does every element in an array
  - to get an array cell, we merely need to *compute* the address
- The "index" and "scale" addressing features are designed for arrays
- … well, that and *any* block of memory

Array Math Example

- Start of our block of memory (buffer) is at address 2000
- The first array cell is at 2000
- Arrays consists of bytes…
  - the second is at 2001
  - the third is at 2002
  - the fourth at 2003
  - etc…

2000	H
2001	e
2002	l
2003	l
2004	o

Array Math Example – 32 bit

- However, what if we are storing 32-bit integers?
- A 32-bit integer takes 4 bytes in memory
- So, as a result, each cell will require 4 bytes of memory

Array Math Example – 32 bit

- First cell uses 2000… 2003
- Since each cell is 4 bytes…
  - the second is at 2004
  - the third is at 2008
  - the fourth at 2012
  - etc…

2000	F0A3
2004	042B
2008	C1F1
2012	0D0B
2016	9C2A

Array Math Example – 64 bit

- The case with 64-bit integers is exactly the same
- A 64-bit integer takes 8 bytes in memory
- So, as a result, each cell will require 8 bytes of memory

Array Math Example – 64 bit

- First cell uses 2000… 2007
- Since each cell is 4 bytes…
  - the second is at 2008
  - the third is at 20016
  - the fourth at 2024
  - etc…

2000	F0A3
2008	042B
2016	C1F1
2024	0D0B
2032	9C2A

Behind the Scenes…

- So, when an array element is read, internally, a mathematical equation is used
- It takes into account the start of the first cell, the array index, and the size of each element

```
Start of Buffer + (Index * Size)
```


Behind the Scenes…

- *This is why the C Programming Languages uses zero as the first array element*
- If zero is used with this formula, it gets the start of the buffer

```
Start of Buffer + (Index * Size)
```


Behind the Scenes…

- Java uses zero-indexing because C does
- … and C does so it can create efficient assembly!

```
Start of Buffer + (Index * Size)
```


Addressing on the x86

Grabbing any byte

Addressing on the x86

- The Intel x86 supports direct, indirect, indexing and scaling
- So, the Intel is very versatile in how it can access memory
- This is typical of CISC-ish architectures

Effective Addresses

- Using the addresses stored in memory, registers, etc… is useful in programs
- Often programs contain *groups* of data
  - fields in an abstract data type
  - cells in an array
  - entries in a large table etc…

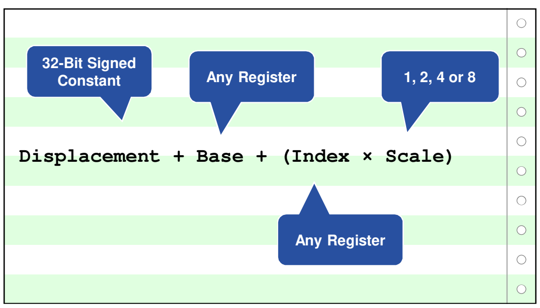
Effective Addresses

- Processors have the ability to create an *effective address* by combining data
- How it works:
  - starts with a base address
  - then adds a value (or values)
  - finally, uses this temporary value as the actual address

Terminology

- *Base-address* is the initial address
- *Displacement (aka offset)* is a constant (immediate) that is added to the address
- *Index* is a register added to the address
- *Scale* used to multiply the index before adding it to the address

x86 Effective Address Formula



Behind the Scenes…

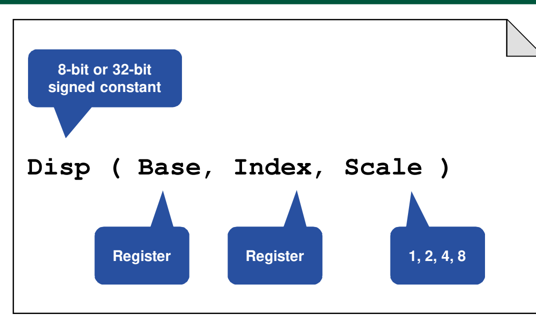
- But wait, doesn't that formula look familiar?
- The addressing term "scale" is basically equivalent to "size" in this example
- Addressing helps us use arrays!

```
Start of Buffer + (Index * Size)
```


Addressing Notation in Assembly

- The AT&T / GAS notation allows you to specify the full addressing
- The notation is a tad terse, and the alternative, Intel notation, is easier to read
- However…
  - you will get used to it quite quickly
  - look at what you can read already!

AT&T / GAS Operand Notation



AT&T / GAS Notation

Mode	Syntax	Example
Direct	`Address`	`mov address, %rdx`
Direct Indexed	`Address (Index)`	`mov address(%rax), %rdx`
Register Indirect	`(Register)`	`mov (%rax), %rdx`
Register Indirect Indexed	`(Register, Index)`	`mov (%rax, %rbx), %rdx`

Addressing Notation in Assembly

- When you write an assembly instruction…
  - you specify all 4 four addressing features
  - however, notation fills in the "missing" items
- For example: for direct addressing…
  - Displacement → Address of the data
  - Base → Not used
  - Index → Not used
  - Scale → 1, which is irrelevant without an Index

How Many Bytes

- When you store data into a register, the assembler knows *(by looking at the size of the register)* how much is going to be accessed
- However, when using addressing,
  - it sometimes is not obvious if you are accessing a byte, 2 bytes, etc…
  - this will cause a very cryptic error

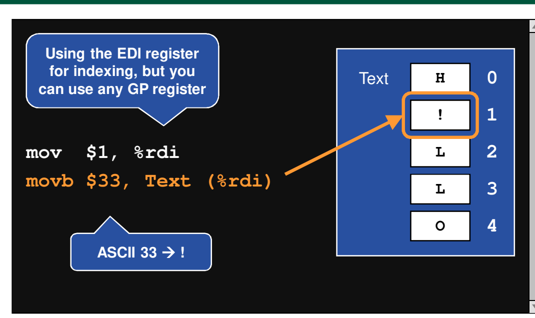
How Many Bytes

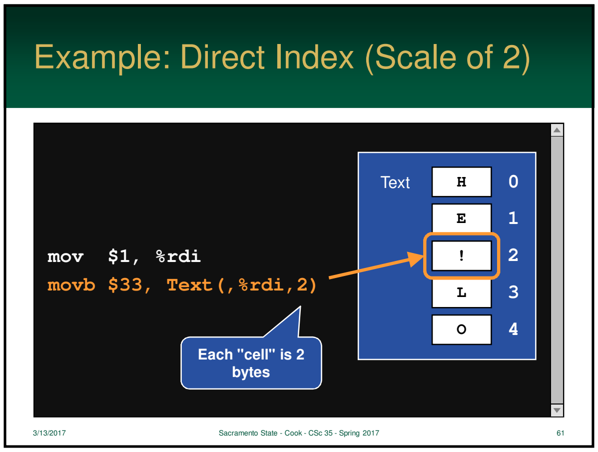
- To address this issue, AT&T/GAS notation places a single character after the instruction name
- This suffix will tell the assembler how many bytes will be accessed during the operation

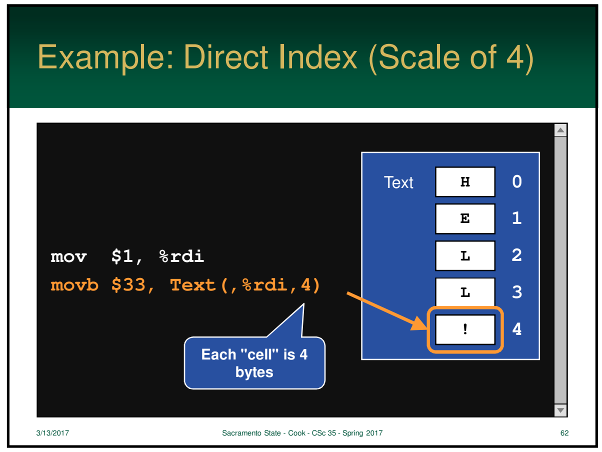
How Many Bytes

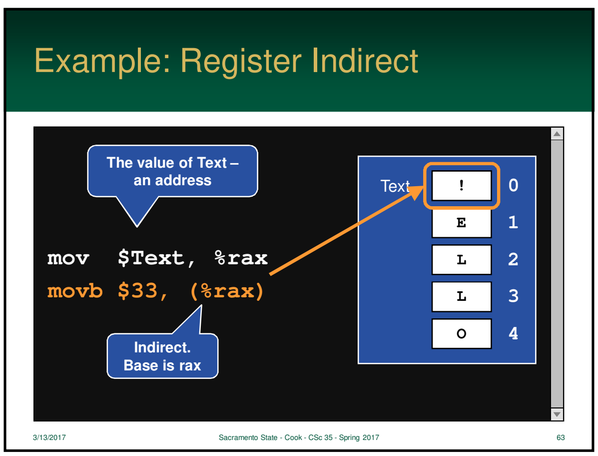
Suffix	Meaning
`b`	`byte`
`s`	`short (2 bytes)`
`l`	`long (4 bytes)`
`q`	`quad (8 bytes)`

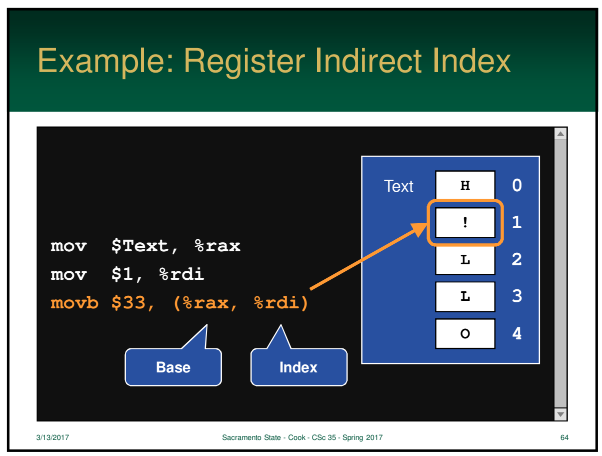
Example: Direct Index

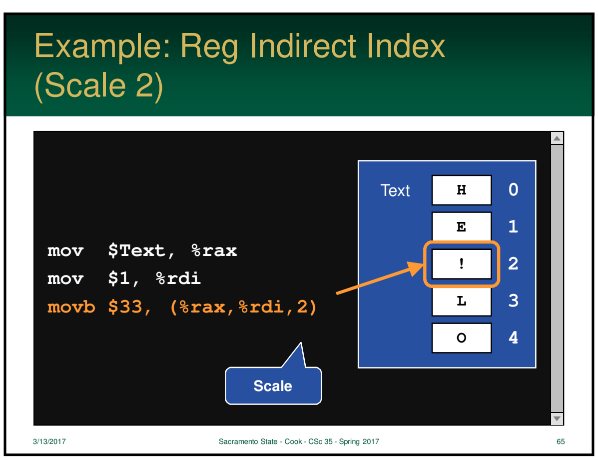


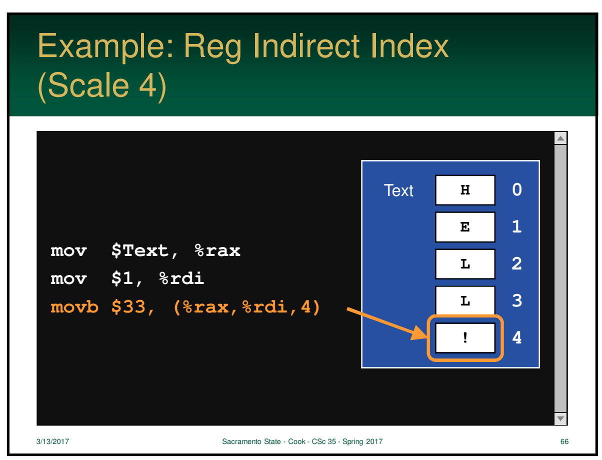

Example: Direct Index (Scale of 2)
Text
H
E
!
L
O
0
1
2
3
4
mov $1, %rdi
movb $33, Text(,%rdi,2)
Each "cell" is 2 bytes
3/13/2017
Sacramento State - Cook - CSc 35 - Spring 2017
61


Example: Direct Index (Scale of 4)
Text
H
E
L
L
!
0
1
2
3
4
mov $1, %rdi
movb $33, Text(,%rdi,4)
Each "cell" is 4 bytes
3/13/2017
Sacramento State - Cook - CSc 35 - Spring 2017
62


Example: Register Indirect
The value of Text – an address
Text
!
E
L
L
O
0
1
2
3
4
mov $Text, %rax
movb $33, (%rax)
Indirect. Base is rax
3/13/2017
Sacramento State - Cook - CSc 35 - Spring 2017
63


Example: Register Indirect Index
Text
H
!
L
L
O
0
1
2
3
4
mov $Text, %rax
mov $1, %rdi
movb $33, (%rax, %rdi)
Base
Index
3/13/2017
Sacramento State - Cook - CSc 35 - Spring 2017
64


Example: Reg Indirect Index (Scale 2)
Text
H
E
!
L
O
0
1
2
3
4
mov $Text, %rax
mov $1, %rdi
movb $33, (%rax,%rdi,2)
Scale
3/13/2017
Sacramento State - Cook - CSc 35 - Spring 2017
65

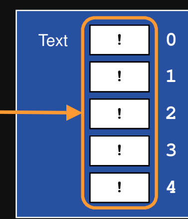

Example: Reg Indirect Index (Scale 4)
Text
H
E
L
L
!
0
1
2
3
4
mov $Text, %rax
mov $1, %rdi
movb $33, (%rax,%rdi,4)
3/13/2017
Sacramento State - Cook - CSc 35 - Spring 2017
66

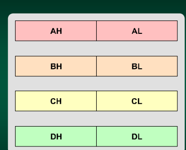
For Loop: 0 to 4

```
        mov   $0, %rdi

    Loop:
        cmp   $4, %rdi
        jg    End

        movb  $33, Text (%rdi)
        add   $1, %rdi
        jmp   Loop
    End:
```


For Loop: 0 to 4

```
mov   $0, %rdi

Loop:
   cmp   $4, %rdi
   jg    End

   movb   $33, Text (%rdi)
   add   $1, %rdi
   jmp   Loop
End:
```



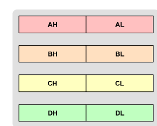


x86 Register Mastery

Choosing the right register (and mode)

x86 Register Mastery

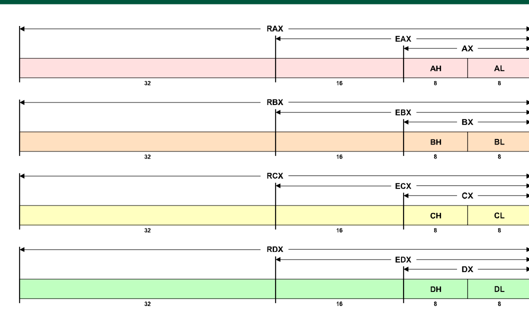
- x86 has 8-bit, 16-bit, 32-bit, and 64-bit registers
- They are different parts of the same register (e.g. ah, al, ax, rax are the "A" register)
- Using the correct one is vital to making your program work



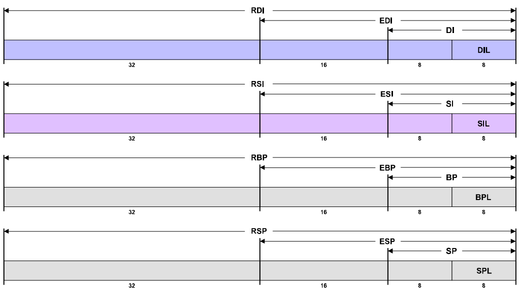
x86 Register Mastery

- When you load/store data, the register will grab as many bytes as it can store
- So…
  - 8-bit register will access 1 byte
  - 16-bit register will access 2 bytes
  - 32-bit register will access 4 bytes
  - 64-bit register will access 8 bytes
- Using the wrong size can cause problems

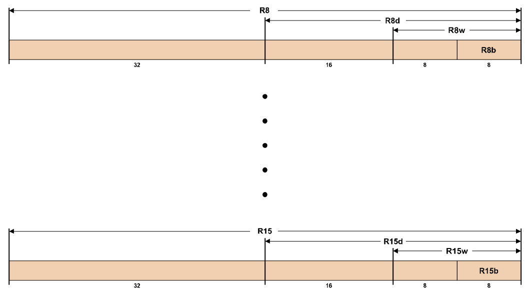
Expansion to 64-bit



Expansion to 64-bit



New 64-bit Registers



Example Program

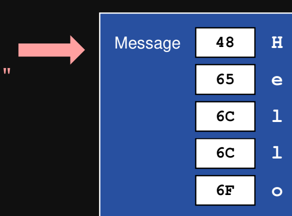
```
.data
Message:
    .ascii "Hello"

.text
.global _start

_start:
    mov  Message, %eax
```

Creates 5 bytes to store Hello

eax is 32-bit (4 bytes)

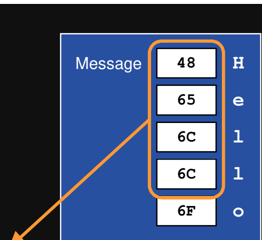
Example Program

```
.data
Message:
    .ascii "Hello"

.text
.global _start

_start:
    mov  Message, %eax
```



Example Program

```
.data
Message:
    .ascii "Hello"

.text
.global _start

_start:
    mov  Message, %eax
```

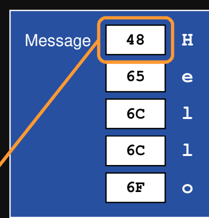


Example Program

- In that example, we used a 32-bit register (eax) to read from the address "Message".
- It grabbed 4 bytes!
- If we wanted to compare a single character to another using 32-bit registers…
  - it would fail – we grabbed too much!
  - it would also compare those extra characters

Example Program

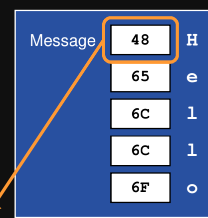
```
.data
Message:
    .ascii "Hello"

.text
.global _start

_start:
    mov  Message, %al
```



This works, but gives a warning…

```
.data
Message:
    .ascii "Hello"

.text
.global _start

_start:
    movb  Message, %rax
```

single byte



Buffer Overflow

With Great Power
Comes Great Responsibility

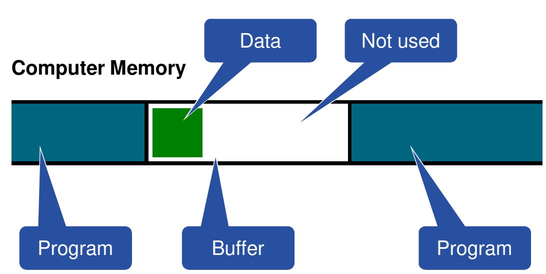
Buffer Overflow

- Operating systems protect programs from having their memory / code damaged by another program
- However…operating systems don't protect programs from damaging *themselves*

Buffers

- In memory, a running program's data is often stored next to its instructions
- Blocks of memory called *buffers* can store data (which can vary in size)
- Examples:
  - people's names
  - list of pet names
  - bytes in an image

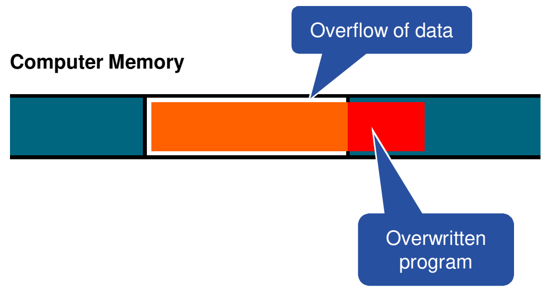
Buffer Overflow – How it Works

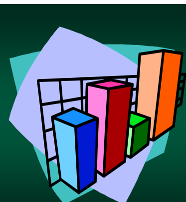


Buffer Overflow

- It is possible to store too much information – resulting in a *buffer overflow*
- The extra bytes will overwrite part of the running program – changing it!

Buffer Overflow – How it Works



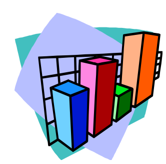


Tables

How to Organize Data

Data Tables

- In assembly, you have full control of memory
- You can take advantage of these to create tables
- They can contain any data – from integers, to characters, to addresses



ASCII Directive Creates a Table

```
Text:
    .ascii "Hello"
```

Creates 5 bytes to store Hello. They are stored consequently

Same Thing!

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

Created byte by byte

Accessing Each Cell

Use register to hold table index

```
mov   $1, %rdi
movb  Text(%rdi), %ah
```

Text		
	H	0
	E	1
	L	2
	L	3
	O	4

Tables of Integers

- Tables can contain anything!
- Often, they are used to store integers & addresses (8 bytes on a 64-bit system)
- Just make sure to use the scale feature!

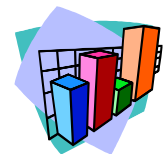


Table of Long Integers

```
Values:
     .quad 45
     .quad 35
     .quad 100
     .quad 25
     .quad 75
```

4 Bytes each

Table of Long Integers

```
Values:
     .quad 45
     .quad 35
     .quad 100
     .quad 25
     .quad 75
```

Values		
	45	0
	35	8
	100	16
	25	24
	75	32

Accessing Each Cell

Table index 1

```
mov  $1, %rdi
movl Values(,%rdi,8), %rax
```

Note the scale!

Values		
	45	0
	35	8
	100	16
	25	24
	75	32

Jump Table

- You can also jump to a value stored in a register
- … which means you can create a jump table!

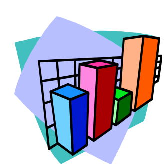


Table of Addresses

```
JumpTable:
    .quad ReadInt
    .quad PrintInt
```


Calling a Register (a tad odd)

```
mov  $1, %rdi
movl JumpTable(,%rdi,8), %rbx

call *%rbx
```

AT&T notation requires an asterisk