Memory

Part 5

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?*

Example Unsigned Integer (4 Byte)

1,188,852,977

46	DC	74	F1

Most significant Byte (MSB)

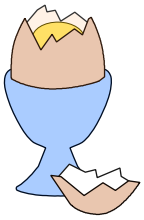
Least significant Byte (LSB)

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



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

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

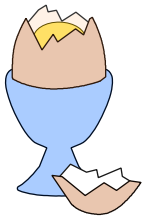
- So, whenever data is read from secondary storage, you cannot assume it will be in your processor's format
- This is compounded by file formats (gif, jpeg, mp3, etc…) which are also inconsistent

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

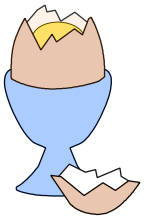
So… who is correct?

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



So… who is correct?

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



Gulliver's Travels

Buffers

Creating your own space

Buffers

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

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

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 an initial value(s)
`.byte`	Allocate byte(s) with an initial value(s)
`.space`	Allocate any ***size*** of empty bytes (with initial values).

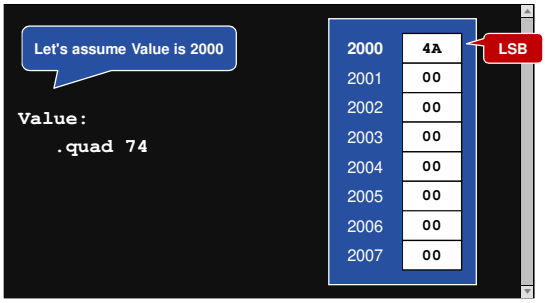
Labels are addresses

- 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

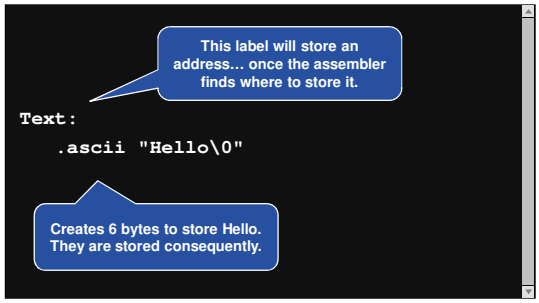
Labels are addresses

- 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.

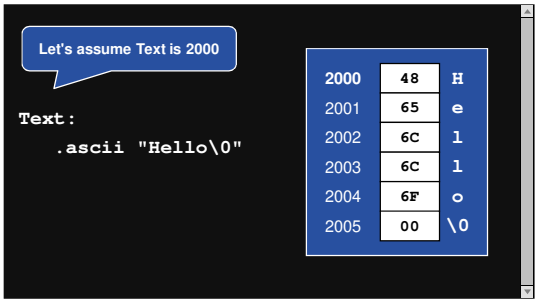
Quad Directive



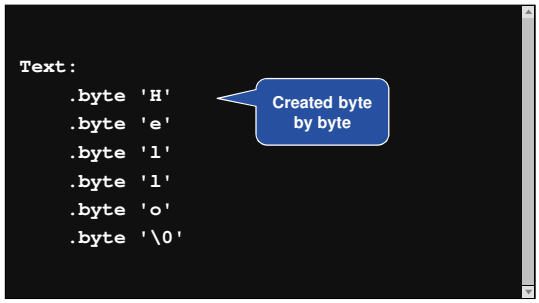
ASCII Directive Creates a Buffer



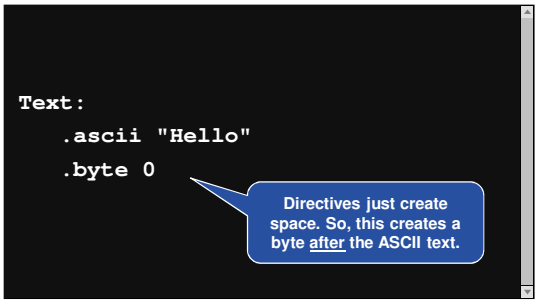
Bytes are stored consecutively



Same Thing!



This works too!



Create a Buffer of Any Size

```
Text:
     .space 30
```

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

Create a Buffer of Any Size

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

Create 30 bytes. All of which are 0.

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
- This is just like an array
- To get an array cell
  - we merely need to compute the address
  - we must also remember that some values take multiple bytes – *so there is math*

Array Math Example

- Let's again assume that our buffer starts 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 – 16 bit

- First cell uses 2000… 2001
- Since each cell is 2 bytes…
  - the second is at 2002
  - the third is at 2004
  - the fourth at 2006
  - etc…

2000	F0A3
2002	042B
2004	C1F1
2006	0D0B
2008	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 to 2007
- Second is at 2008
- Third is at 2016
- Fourth at 2024
- etc…

2000	446576696E20436F
2008	6F6B000000000000
2016	53616372616D656E
2024	746F205374617465
2032	4353433335000000

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 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

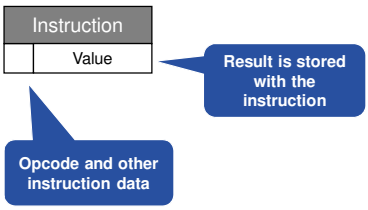
4 Basic Addressing Modes

1. Immediate Addressing
2. Register Addressing
3. Direct Addressing
4. Indirect Addressing

Immediate Addressing

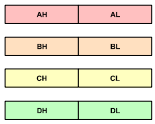
- Immediate addressing is one of the most basic modes found on a processor
- Often a value is stored as part of the instruction
- As the result, it is *immediately* available
- Very common for assigning constants

Immediate Addressing

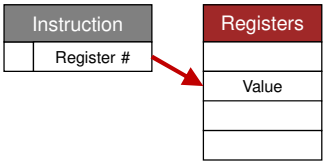


Register Addressing

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



Register Addressing



Register & Immediate in Java

- The following, for comparison, is the equivalent code in Java
- The register file (for rax) is set to the value 1.

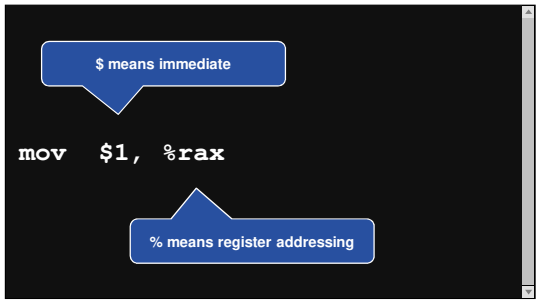
```
//mov $1, %rax
rax = 1;
```


Register & Immediate in Java

- This is the also the case with labels
- Remember: labels are addresses (numbers)

```
//mov $label, %rax
rax = label;
```

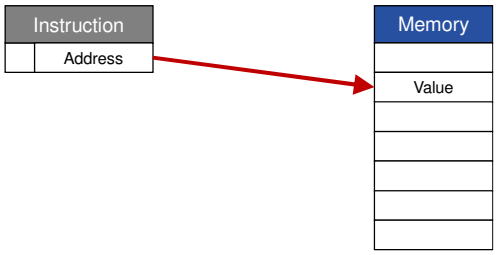

Example: Immediate & Register



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



Direct in Java

- The following, for comparison, is the equivalent code in Java
- The memory at the address *total* is stored into rax
- **Notice:** no $ in assembly

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


Example: Direct

```
.data
total:
   .quad 0

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

64 bit integer. With an initial value of 0.

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

Cause of the Segmentation Fault

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

.text
.global _start

_start:
   mov  Message, %rcx
```

Creates 8 bytes using ASCII values

rcx is 64-bit (8 bytes)

Cause of the Segmentation Fault

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

.text
.global _start

_start:
   mov  Message, %rcx
```

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

Cause of the Segmentation Fault

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

.text
.global _start

_start:
   mov  Message, %rcx
```

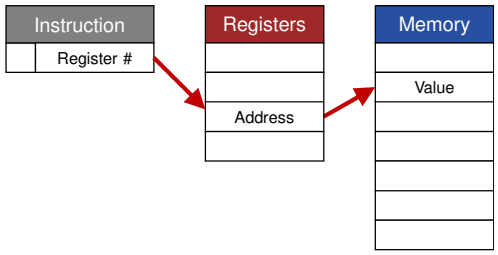
Huge value

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

Register Indirect Addressing

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

Register Indirect Addressing



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.

```
// mov (rbx), %rax
rax = Memory[rbx];
```


Example: Indirect

```
.data
total:
   .quad 0

.text
.global _start
_start:
   mov  $total, %rbx
   mov  (%rbx), %rax
```

64 bit integer. With an initial value of 0.

With the $, rbx gets the address of total

rax gets the data at the address stored in rbx

Relative Addressing

- In *relative addressing*, 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

Indexing on the x64

Grabbing any byte

Indexing on the x64

- The Intel x64 also 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…

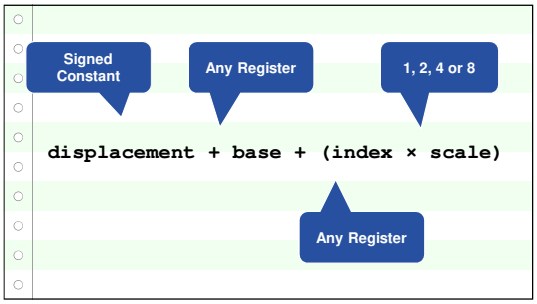
Effective Addresses

- 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

x64 Effective Address Formula



Behind the Scenes…

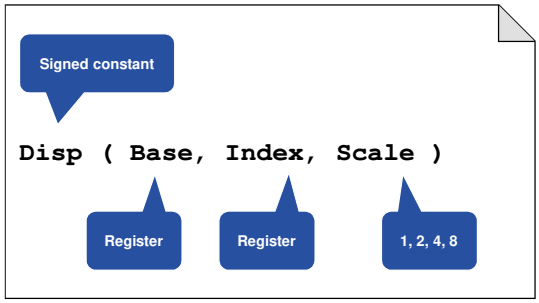
- But wait, doesn't that formula look familiar?
- The addressing term "scale" is basically equivalent to "size" in this example
- Addressing and arrays work together flawlessly

```
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	Java Equivalent
Immediate	`$value`	`value`
Register	`%reg`	`reg`
Direct	`address`	`Memory[address]`
Direct Indexed	`address(%reg)`	`Memory[address + %reg]`
Indirect	`(%reg)`	`Memory[%reg]`
Indirect Indexed	`(%reg, %reg)`	`Memory[%reg + %reg]`
Indirect Indexed Scaled	`(%reg, %reg, scale)`	`Memory[%reg + %reg × scale]`

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

Sizing Instructions

- 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, with addressing, sometimes the number of bytes (1, 2, etc..) can't be determined

How Many Bytes

- If it is not obvious to the assembler how many bytes you want to access, it will report a *very cryptic* error
- To address this issue…
  - AT&T/GAS notation allows you 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: Suffix Needed

```
.data
total:
   .quad 0

.text
.global _start
_start:
   movq  $42, total
```

The q suffix is needed. Neither $42 or total (an address) has a known size.

Indexing Examples

- The following examples use addressing modes modify an ASCII buffer
- Let's assume that the start of the buffer **Talk** is **5000**

Talk = 5000

5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	6F	o

Example: Direct Index

Using the RDI register for indexing, but you can use any GP register

```
mov   $1, %rdi
movb $33, Talk(%rdi)
```

ASCII 33 → !

5000	48	H
5001	33	!
5002	6C	l
5003	6C	l
5004	6F	o

Example: Direct Index (Scale 2)

```
mov   $1, %rdi
movb $33, Talk(,%rdi,2)
```

5000	48	H
5001	65	e
5002	33	!
5003	6C	l
5004	6F	o

Example: Direct Index (Scale 4)

```
mov   $1, %rdi
movb $33, Talk(,%rdi,4)
```

5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	33	!

Example: Register Indirect

The value of Text – an address

```
mov   $Talk, %rax
movb $33, (%rax)
```

Indirect. Base is rax

5000	33	!
5001	65	e
5002	6C	l
5003	6C	l
5004	6F	o

Example: Register Indirect Index

```
mov   $Talk, %rax
mov   $1, %rdi
movb $33, (%rax, %rdi)
```

Base Index

5000	48	H
5001	33	!
5002	6C	l
5003	6C	l
5004	6F	o

Ex: Register Indirect Index (Scale 2)

```
mov   $Talk, %rax
mov   $1, %rdi
movb $33, (%rax,%rdi,2)
```

Scale

5000	48	H
5001	65	e
5002	33	!
5003	6C	l
5004	6F	o

Ex: Register Indirect Index (Scale 4)

```
mov   $Talk, %rax
mov   $1, %rdi
movb $33, (%rax,%rdi,4)
```

5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	33	!

Buffer Overflow

With Great Power
Comes Great Responsibility

Buffer Overflow

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

Buffers & Programs

- In memory, a running program's data is often stored next to its instructions
- This means…
  - if the end of a buffer of exceeded, the program can be read/written
  - this is a common hacker technique to modify a program *while it is running!*

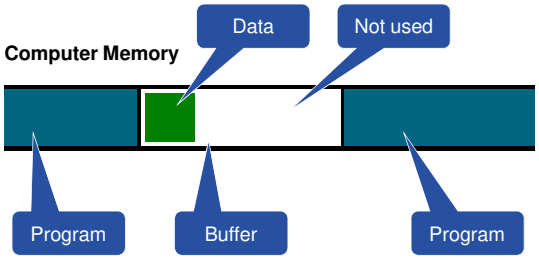
Example Program

```
.data
Kitty:
   .ascii "Cat"

Puppy:
   .ascii "Dog"

.text
.global _start
_start:
```

Kitty	43	C
	61	a
	74	t
Puppy	44	D
	6F	o
	67	g
_start		
	...	

Start of program

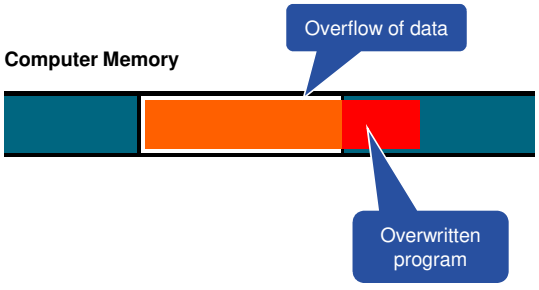
Buffer Overflow – How it Works



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



Bad Indexing

- It is possible to accidentally change data stored in the different buffers
- In assembly, you have full control over your allocated memory
- *With great power comes great responsibility*

Wrong Buffer Changed

```
.data
Kitty:
   .ascii "Cat\0"
Puppy:
   .ascii "Dog\0"

.text
.global _start
_start:
    mov  $4, %rdi
    movb $72, Kitty(%rdi)
```

4 bytes. Character indexes from 0 to 3

72 is ASCII 'H" In hex it's 48

Wrong Buffer Changed

```
.data
Kitty:
   .ascii "Cat\0"
Puppy:
   .ascii "Dog\0"

.text
.global _start
_start:
    mov  $4, %rdi
    movb $72, Kitty(%rdi)
```

Kitty	43	C
	61	a
	74	t
	00	
Puppy	44	D
	6F	o
	67	g
	00	

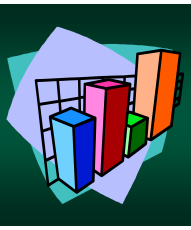
Wrong Buffer Changed

```
.data
Kitty:
   .ascii "Cat\0"
Puppy:
   .ascii "Dog\0"

.text
.global _start
_start:
    mov  $4, %rdi
    movb $72, Kitty(%rdi)
```

Kitty	43	C
	61	a
	74	t
	00	
Puppy	48	H
	6F	o
	67	g
	00	

Wrong Buffer Changed

Kitty	43	C
	61	a
	74	t
	00	
Puppy	48	H
	6F	o
	67	g
	00	



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

Accessing Each Cell

Use register to hold table index

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

	Text	
0	H	
1	E	
2	L	
3	L	
4	O	

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

```
Years:
    .quad 1776
    .quad 1783
    .quad 1846
    .quad 1850
    .quad 1947
```

8 Bytes each

Assuming Years is 6000

```
Years:
    .quad 1776
    .quad 1783
    .quad 1846
    .quad 1850
    .quad 1947
```

6000	1776
6008	1783
6016	1846
6024	1850
6032	1947

Assuming Years is 6000

Table index 1

```
mov  $1, %rdi
mov Years(,%rdi,8), %rax
```

Note the scale!

6000	1776
6008	1783
6016	1846
6024	1850
6032	1947