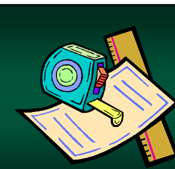
Memory Indexing

Part 5

1

Sizing Instructions

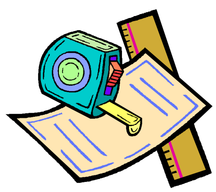


How many bytes are you using?

2

Sizing Instructions

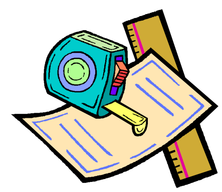
- The Intel can load/store 1-byte, 2-byte, 4-byte or 8-byte values
- Whenever a processor accesses memory, the instruction specifies how many bytes to access



3

Sizing Instructions

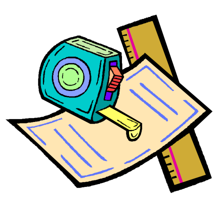
- The assembler will automatically fill this information in for you.
- *How?* If a register is used, the assembly can assume it by looking at size of the register



4

Sizing Instructions

- However, sometimes the number of bytes (1, 2, etc..) can't be determined
- In this case, the assembler will report an error
- … since it doesn't know how to encode the instruction



5

Example: How Many Bytes?

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global _start
_start:
    mov  total, 50
```

total is a target address. It doesn't have any implied size.

6

Example: How Many Bytes?

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global _start
_start:
    mov  total, 50
```

How many bytes is this? The value 50 can be stored in 1, 2, 4, or 8 bytes.

7

How Many Bytes?

- If the assembler can't infer how many bytes to access, it'll will report *"ambiguous operand size"*
- To address this issue…
  - GAS assembly allows you places a single character after the instruction's mnemonic
  - this suffix will tell the assembler how many bytes will be accessed during the operation

8

How Many Bytes

Suffix	Name	Size
b	byte	1 byte
s	short	2 bytes
l	long	4 bytes
q	quad	8 bytes

9

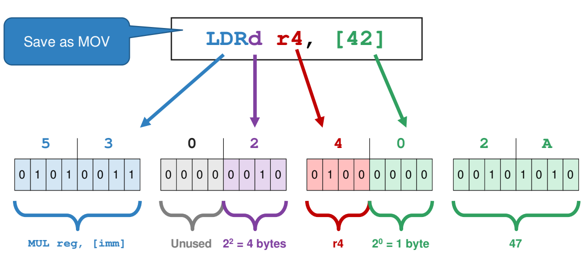
Example: Suffix Used

```
.intel_syntax noprefix
.data
total:
    .quad 0

.text
.global _start
_start:
    movq total, 50
```

Note the q. Now the assembler knows you mean "move quad".

10

Herky Load 4 Byte (32-bit) Example



11

Behind the Scenes of Arrays

All the mystery is revealed!

12

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 is located sequentially in memory – one right after each other

13

Arrays

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

14

Array Math Example

- Let's again assume that our buffer starts at address 2000
- The first array element is located at address 2000
- Arrays consists of bytes…
  - the second is 2001
  - the third is 2002
  - the fourth 2003
  - etc…

2000	H
2001	e
2002	l
2003	l
2004	o

15

Array Math Example – 16 bit

- First element uses 2000… 2001
- Since each array element is 2 bytes…
  - second address is 2002
  - third address is 2004
  - fourth address is 2006
  - etc…

2000	F0A3
2002	042B
2004	C1F1
2006	0D0B
2008	9C2A

16

Array Math Example – 64 bit

- First element uses 2000 to 2007
- Second address is 2008
- Third address is 2016
- Fourth address is 2024
- etc…

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

17

Behind the Scenes…

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

```
start address + (index × size)
```


18

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 address + (index × size)
```

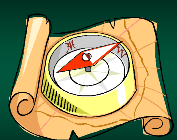

19

Behind the Scenes…

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

```
start address + (index × size)
```


20



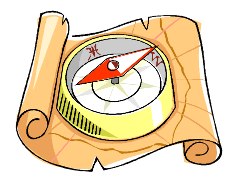
Indexing on the x64

Grabbing any byte

21

Indexing on the x64

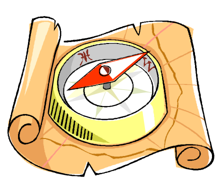
- The Intel x64 supports direct, indirect, indexing and scaling
- So, the Intel is very versatile in how it can access memory
- This is typical of CISC-ish architectures



22

Effective Addresses

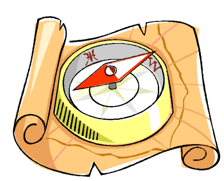
- Processors have the ability to create the *effective address* by combining data
- How it works:
  - starts with a base address
  - then adds a value (or values)
  - finally, uses this temporary value as the actual address



23

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
  - elements in an array
  - entries in a large table etc…



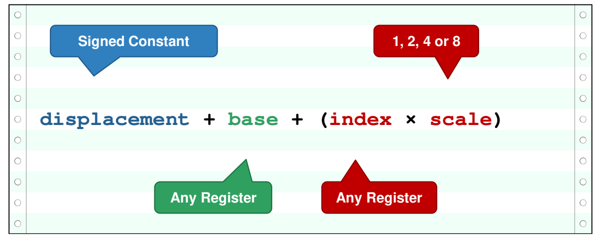
24

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

25

x64 Effective Address Formula



26

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 address + (index × size)
```

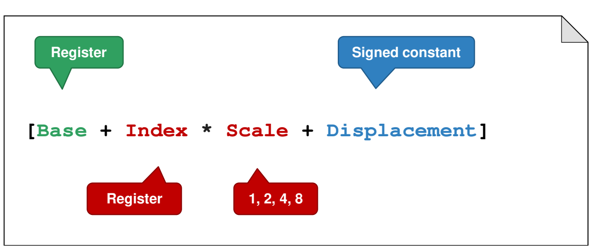

27

Addressing Notation in Assembly

- Intel Notation *(Microsoft actually created it)* allows you to specify the full equation
- The notation is very straight forward and mimics the equation used to compute the effective address
- Parts of the equation can be omitted, and the assembler will understand

28

Intel Notation



29

Notation (reg = register)

Mode	Syntax	Java Equivalent
Immediate	`value`	`value`
Register	`register`	`register`
Direct	`label`	`Memory[label]`
Direct Indexed	`[label + reg]`	`Memory[label + reg]`
Indirect	`[reg]`	`Memory[reg]`
Indirect Indexed	`[reg + reg]`	`Memory[reg + reg]`
Indirect Indexed Scale	`[reg + reg * scale]`	`Memory[reg + reg × scale]`

30

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, irrelevant without an Index

31

Indexing Examples

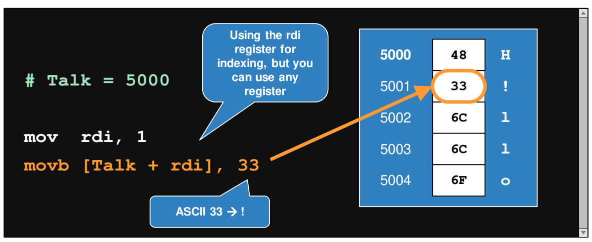
- The following examples use addressing modes modify an ASCII buffer
- Let's assume that the start of the buffer **Talk** is **5000**

Talk = 5000

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

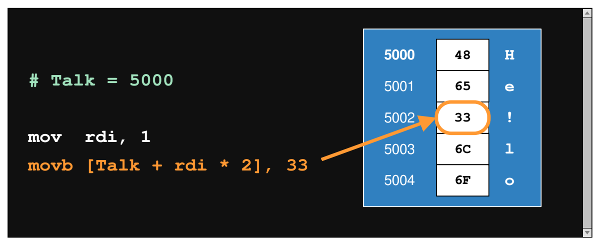
32

Example: Direct Index



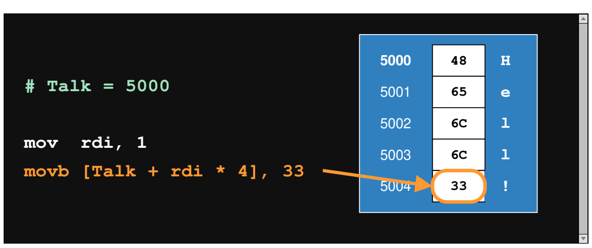
33

Example: Direct Index (Scale 2)



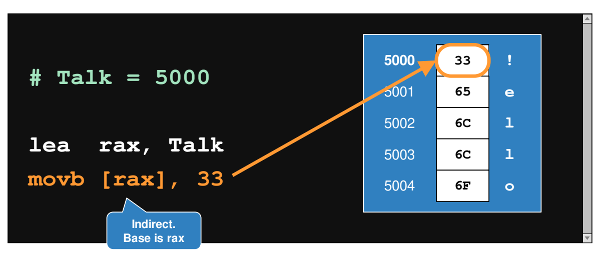
34

Example: Direct Index (Scale 4)



35

Example: Register Indirect



36

Example: Register Indirect Index

```
# Talk = 5000

lea   rax, Talk
mov   rdi, 1
movb [rax + rdi], 33
```

Base · Index

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

37

Ex: Register Indirect Index (Scale 2)

```
# Talk = 5000

lea   rax, Talk
mov   rdi, 1
movb [rax + rdi * 2], 33
```

Scale

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

38

Ex: Register Indirect Index (Scale 4)

```
# Talk = 5000

lea   rax, Talk
mov   rdi, 1
movb [rax + rdi * 4], 33
```

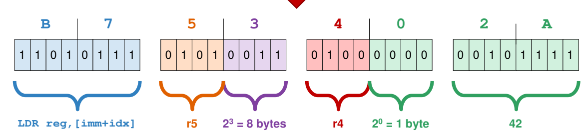
5000	48	H
5001	65	e
5002	6C	l
5003	6C	l
5004	33	!

39

Herky Direct Indexed Example

Assuming Name = 42

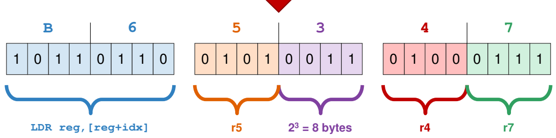
```
LDR r4, [Name + r5 * 8]
```



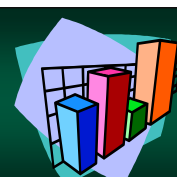
40

Herky Indirect Indexed Example

```
LDR r4, [r7 + r5 * 8]
```



41



Tables

How to Organize Data

42

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

43

Accessing Each element

Use register to hold table index

```
mov   rdi, 1
movb  ah, [Greet + rdi]
```

Greet

H	0
E	1
L	2
L	3
O	4

44

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!

45

Table of Long Integers

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

8 Bytes each

46

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

47

Assuming Years is 6000

Table index 1

```
mov  rdi, 1
mov  rcx, [Years + rdi * 8]
```

Note the scale!

6000	1776
6008	1783
6016	1846
6024	1850
6032	1947

48

Table of Addresses. Assume Names is 3000

```
Sutter:
    .ascii "John Sutter\0"

Marshal:
    .ascii "James Marshal\0"

Names:
    .quad Sutter
    .quad Marshal
```

3000	Sutter *(address)*
3008	Marshal *(address)*

49

Assuming Names is 3000

Note: mov is used. We want the data from the table (which is an address)

```
mov  rdi, 1
mov  rax, [Names + rdi * 8]

call PrintString
```

3000	Sutter *(address)*
3008	Marshal *(address)*

50

Buffer Overflow

With Great Power
Comes Great Responsibility

51

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*

52

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!*

53

Example Program

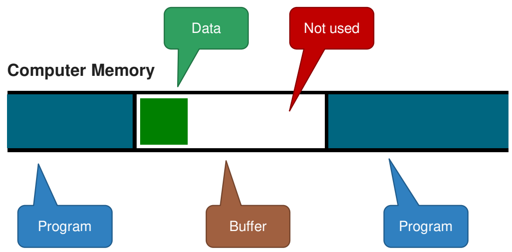
```
.data
Kitty:
    .ascii "Cat"

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

54

Buffer Overflow – How it Works



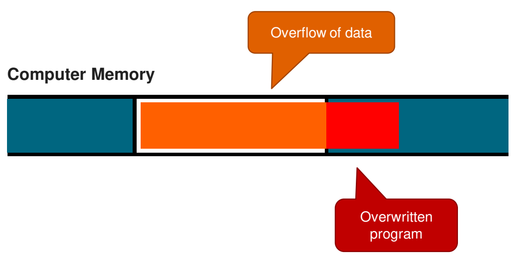
55

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!

56

Buffer Overflow – How it Works



57

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*

58

Wrong Buffer Changed

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

.text
.global _start
_start:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```

4 bytes. Character indexes from 0 to 3

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

59

Wrong Buffer Changed

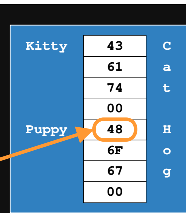
```
.intel_syntax noprefix
.data
Kitty:
    .ascii "Cat\0"
Puppy:
    .ascii "Dog\0"

.text
.global _start
_start:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```

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

60

Wrong Buffer Changed

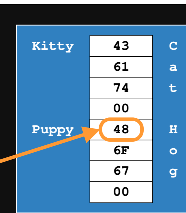
```
.intel_syntax noprefix
.data
Kitty:
    .ascii "Cat\0"
Puppy:
    .ascii "Dog\0"

.text
.global _start
_start:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```



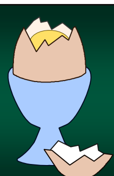
61

Wrong Buffer Changed

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

.text
.global _start
_start:
    mov  rdi, 4
    movb [Kitty + rdi], 72
```



62



Endianness

The "proper" order of things

63

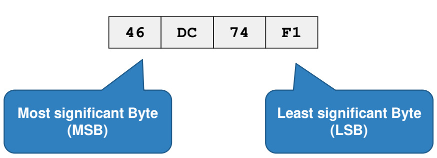
So Many Bytes…

- 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?*

64

Example Unsigned Integer (4 Byte)

1,188,852,977



65

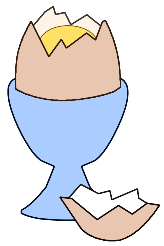
So Many Bytes…

- 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

66

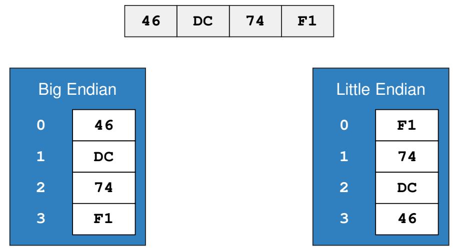
Big Endian vs. Little Endian

- Big-Endian approach
  - store the MSB first
  - used by Motorola & PowerPC
- Little-Endian approach
  - store the LSB first
  - used by Intel



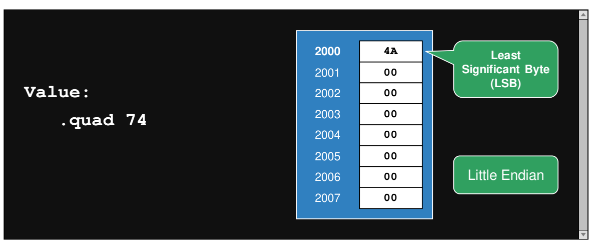
67

Big Endian vs. Little Endian



68

Assuming Value is located at 2000



69

No "End" to Problems

- *There is a problem...* if two systems use different formats, data will be interpreted incorrectly!
- If how the read differs from how it is stored, the data will be mangled

70

No "End" to Problems

- For example:
  - a little-endian system reads a value stored in big-endian
  - a big-endian system reads a value stored in little-endian
- Programmers must be conscience of this whenever binary data is accessed

71

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

72

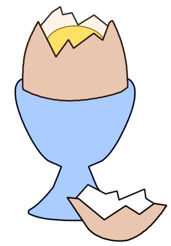
Example File Format Endianness

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

73

So… who is correct?

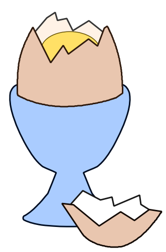
- So, what is the correct and superior format?
- Is it Intel (little endian)?
- …or the PowerPC (big endian) correct?



74

So… who is correct?

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



75

Gulliver's Travels

76