Addressing Modes

Part 8

1

Behind the Scenes of Arrays

All the mystery is revealed!

2

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

3

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*

4

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 at 2001
  - the third is at 2002
  - the fourth at 2003
  - etc…

2000	H
2001	e
2002	l
2003	l
2004	o

5

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

6

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

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)
```


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)
```


Behind the Scenes…

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

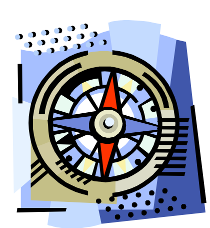
```
start address + (index × size)
```


Addressing Modes

How to interact with memory

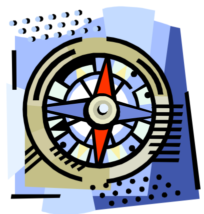
Addressing Modes

- 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

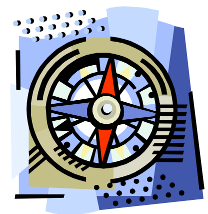
- *How* the processor can locate and read data from memory is called an *addressing mode*
- Information combined from registers, immediates, etc… to create a target address
- Modes vary greatly between processors



13

4 Basic Addressing Modes

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



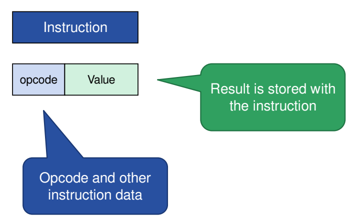
14

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

15

Immediate Addressing



16

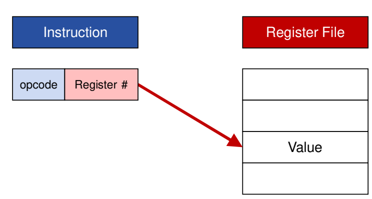
Register Addressing

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

AH	AL
BH	BL
CH	CL
DH	DL

17

Register Addressing



18

Register & Immediate in Java

- The following, for comparison, is the equivalent code in Java
- The register file (for rax) is set to the value 1947.

```
// rax = 1947;
mov rax, 1947
```


19

Example: Immediate & Register

```
mov   rax, 1947
```

immediate

register addressing

20

Register & Immediate in Java

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

```
// rax = label;
lea rax, label
```


21

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...

22

Direct Addressing

23

Register & Direct in Java

- Note the use of the instruction "move"
- The label (and address) will be used as an index into memory

```
// rax = Memory[label];
mov rax, label
```


24

Register & Direct in Java

- Optionally, you can put square brackets around the label
- This explicitly shows it is being used as an index

```
// rax = Memory[label];
mov rax, [label]
```


25

Example: Immediate & Register

Used as an index into memory

```
mov  rax, Total
```

register addressing

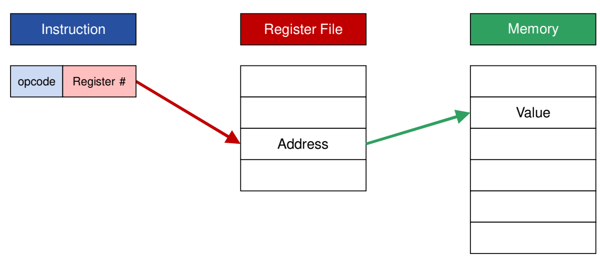
26

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

27

Register Indirect Addressing



28

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.
- *The brackets here are necessary!*

```
// rax = Memory[rbx];
mov rax, [rbx]
```


29

Example: Indirect

```
.intel_syntax noprefix
.data
total:
    .quad 451

.text
.global _start
_start:
    lea  rcx, total
    mov  rax, [rcx]
```

64 bit integer. With an initial value of 451.

Load the address into rcx

rax gets the data at the address stored in rcx

30

Relative Addressing

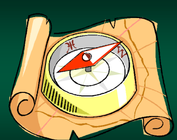
- In *relative addressing*, a value is added to a system register (e.g. program counter)
- Advantages:
  - instruction can just store the *difference* (in bytes) from the current instruction address
  - takes less storage than a full 64-bit address
  - it allows a program to be stored anywhere in memory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

31

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

32



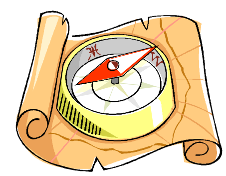
Indexing on the x64

Grabbing any byte

33

Indexing on the x64

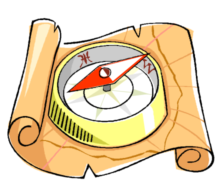
- The Intel x64 also supports direct, indirect, indexing and scaling
- So, the Intel is very versatile in how it can access memory
- This is typical of CISC-ish architectures



34

Effective Addresses

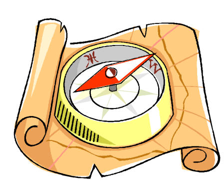
- Using the addresses stored in memory, registers, etc… is useful in programs
- Often programs contain *groups* of data
  - fields in an abstract data type
  - elements in an array
  - entries in a large table etc…



35

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



36

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

37

x64 Effective Address Formula

Signed Constant

1, 2, 4 or 8

```
displacement + base + (index × scale)
```

Any Register

Any Register

38

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)
```


39

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

40

Intel Notation

Register

Signed constant

```
[Base + Index * Scale + Displacement]
```

Register

1, 2, 4, 8

41

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]`

42

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

43

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

44

Example: Direct Index

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

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

ASCII 33 → !

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

45

Example: Direct Index (Scale 2)

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

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

46

Example: Direct Index (Scale 4)

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

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

47

Example: Register Indirect

The value of Text – an address

```
lea  rax, Talk
movb [rax], 33
```

Indirect. Base is rax

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

48

Example: Register Indirect Index

```
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

49

Ex: Register Indirect Index (Scale 2)

```
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

50

Ex: Register Indirect Index (Scale 4)

```
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	!

51

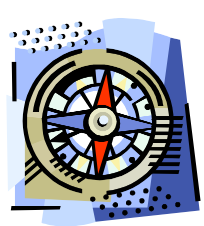
Addressing & Loops

They were made for each other … *literally*

52

Addressing & Loops

- When you use arrays in Java, often the index is a variable
- This allows you to use a For Loop to analyze very element in the array
- This is more common than you think in assembly



53

Addressing & Loops

- So, processors allow a register to be used as an index
- This allows you to:
  - copy strings (copying arrays)
  - search through a list
  - and much more…

54

For Loop: 0 to 4 - Before

```
.intel_syntax noprefix
.data
Greet:
    .ascii "HELLO"

.text
.global _start

_start:
```

Greet		
	H	0
	E	1
	L	2
	L	3
	O	4

55

For Loop: 0 to 4

```
    mov  rdi, 0

Loop:
    cmp  rdi, 4
    jg   End

    movb [Greet + rdi], 33
    add  rdi, 1
    jmp  Loop
End:
```

! character

Greet		
	H	0
	E	1
	L	2
	L	3
	O	4

56

For Loop: 0 to 4 - Before

```
mov   rdi, 0

Loop:
    cmp  rdi, 4
    jg   End

    movb [Greet + rdi], 33
    add  rdi, 1
    jmp  Loop
End:
```

Greet		
	H	0
	E	1
	L	2
	L	3
	O	4

57

For Loop: 0 to 4 - After

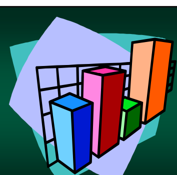
```
mov   rdi, 0

Loop:
    cmp  rdi, 4
    jg   End

    movb [Greet + rdi], 33
    add  rdi, 1
    jmp  Loop
End:
```

Greet		
	!	0
	!	1
	!	2
	!	3
	!	4

58



Tables

How to Organize Data

59

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

60

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

61

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!

62

Table of Long Integers

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

8 Bytes each

63

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

64

Assuming Years is 6000

Table index 1

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

Note the scale!

6000	1776
6008	1783
6016	1846
6024	1850
6032	1947

65

Buffer Overflow

With Great Power
Comes Great Responsibility

66

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*

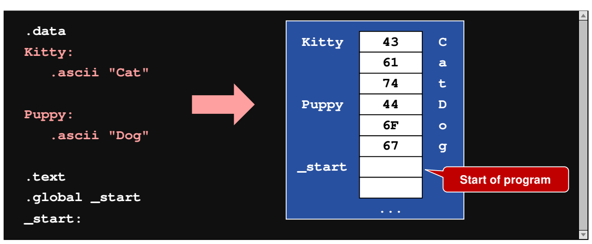
67

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!*

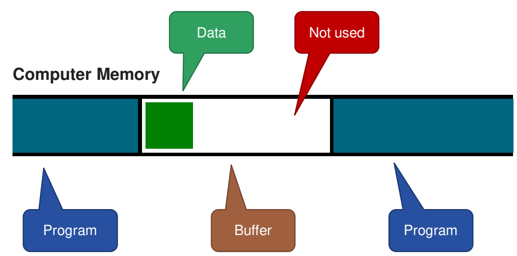
68

Example Program



69

Buffer Overflow – How it Works



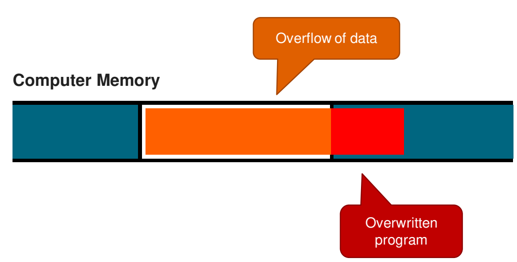
70

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!

71

Buffer Overflow – How it Works



72

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*

73

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

74

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	

75

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], 7
```

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

76