Memory

Part 4

1

von Neumann Architecture

The Information Superhighway

2

von Neumann Machine Architecture

- Modern computers are based on the design of John von Neumann
- His design greatly simplified the construction of (and use) computers

3

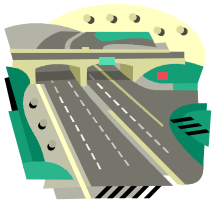
Some von Neumann Attributes

1. Programs are stored and executed in memory
2. Separation of processing from memory
3. Different system components communicate over a shared bus

4

The Bus

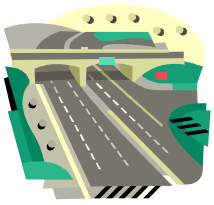
- Electronic pathway that transports data between components
- Think of it as a "highway"
  - data moves on shared paths
  - otherwise, the computer would be very complex



5

System Bus

- The information sent on the memory bus falls into 3 categories
- Three sets of signals
  - address bus
  - data bus
  - control bus



6

Address Bus

- Used by the processor to access a specific piece of data
- This "address" can be
  - a specific byte in memory
  - unique IO port
  - etc…
- The more bits it has, the more memory can be accessed

7

Address Bus Size Examples

- 8-bit → 2^8 = 256 bytes
- 16-bit → 2^{16} = 64 KB (65,536 bytes)
- 32-bit → 2^{32} = 4 GB (4,294,967,296 bytes)
- 64-bit → 2^{64} = 18 EB (18,446,744,073,709,551,616)

8

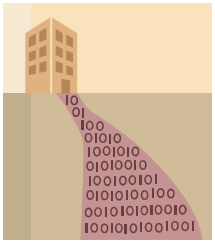
Historic Address Sizes

- Intel 8086
  - original 1982 IBM PC
  - 20-bit address bus (1 MB)
  - only 640 KB usable for programs
- MOS 6502 computers
  - Commodore 64, Apple II, Nintendo, etc…
  - 16-bit address bus (64 KB)

9

Data Bus

- The actual data travels over the *data bus*
- The number of bits that the processor uses – as its natural unit of data – is called a *word*



10

Data Bus

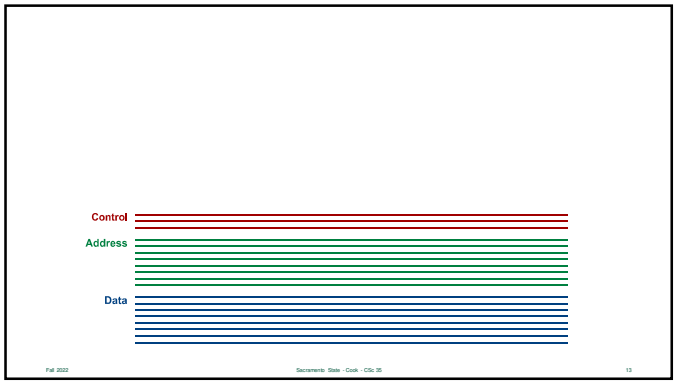
- Typically we define a system by word size
- Example:
  - 8-bit system uses 8 bit words
  - 16-bit system uses 16 bits (2 bytes) words
  - 32-bit system uses 32 bits (4 bytes) words
  - etc…

11

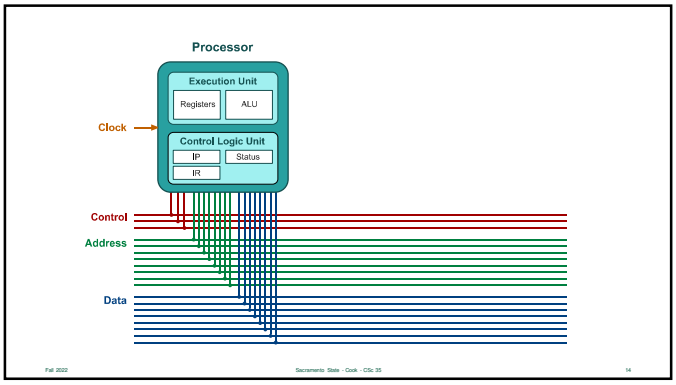
Control Bus

- The *control bus* controls the timing and synchronizes the subsystems
- Specifies what is happening
  - read data
  - write data
  - reset
  - etc…

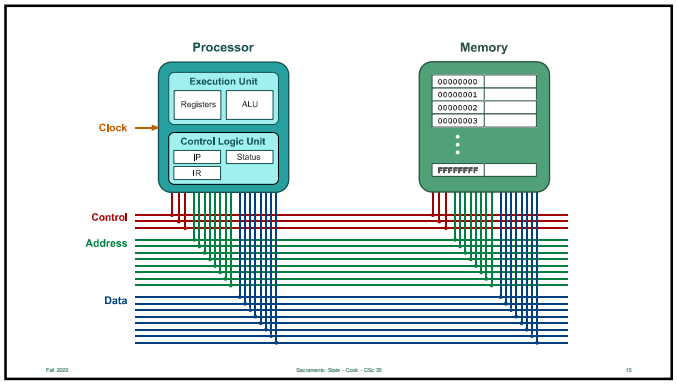
12



13



14



15

von Neumann Architecture Today

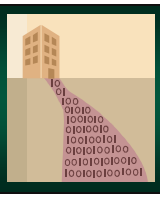
- Because of the emphasis on memory, most real-world systems use a modified version of his design
- In particular, they have a special high-speed bus between the processor and memory

16

von Neumann Architecture Today

- Think of it as a diamond-lane on a freeway
- … or as high-speed rail – which has a fixed source and destination and goes faster than the freeway

17



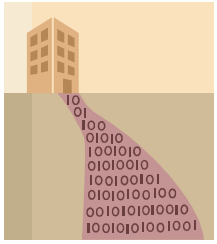
Accessing Data

Filling the registers

18

Accessing Data

- Processors use registers to hold data being computed
- So, how is data put into the registers to begin with?
- Data can come from two major sources



19

Immediates

- In programming, it is common to assign a constant to a variable
- As you can imagine, this will also be quite common with instructions

20

Immediates

- When a constant is stored as part of instruction, it is called an *immediate*
- Once the instruction is loaded by the processor, it is "immediately" available from the IR – hence, the name

21

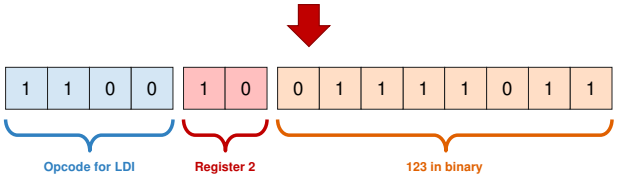
Load Immediate

- A *Load Immediate* instruction, stores a constant into a register
- The instruction must store the destination register and the immediate value

22

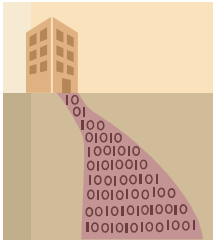
Load Immediate Example (not x86)

```
LDI r2, 123
```



23

Copying Data



- Processors have a number of instructions that can copy data
- Each has a unique name –*not surprising since each does something different*

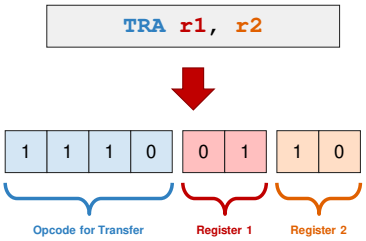
24

Transfer

- A *Transfer* instruction, copies the contents of one instruction into another
- The instruction must store both the destination and source register

25

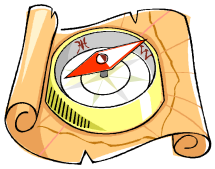
Transfer Example (not x86)



26

Effective Addresses

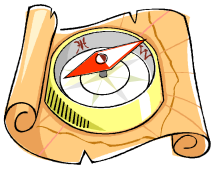
- Processors also have the ability to access memory
- Since memory is a massive array, the processor needs to know the address



27

Effective Addresses

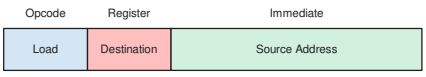
- The *effective address* is used to access memory
- Often it is created by combining multiple values
- … but we will cover that later in the semester



28

Load

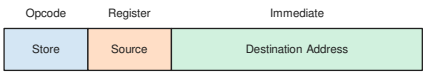
- A *Load* instruction, reads data from memory (at a specified address)
- This data is then stored into the destination register



29

Store

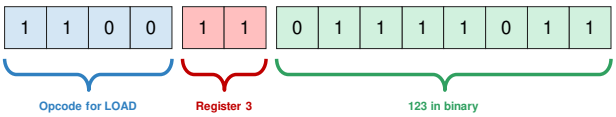
- A *Store* instruction, writes data from a register into the specified address
- Note: the structure is identical to Load



30

Load Example (not x86)

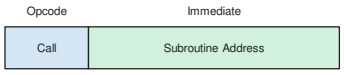
```
LDR r3, [123]
```



31

Call Instruction

- The *Call instruction* doesn't change any of the general-purpose registers
- It only stores an address – where execution will continue



32

Buffers

Creating your own space

33

Buffers

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

34

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

35

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).

36

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

37

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.

38

Quad Directive

```
Value:
    .quad 74
```

Let's assume Value = 2000

2000	4A
2001	00
2002	00
2003	00
2004	00
2005	00
2006	00
2007	00

39

ASCII Directive Creates a Buffer

```
Text:
    .ascii "Hello\0"
```

This label will store an address... once the assembler finds where to store it.

Creates 6 bytes to store Hello. They are stored consequently.

40

Bytes are stored consecutively

```
Text:
    .ascii "Hello\0"
```

Let's assume Text = 2000

2000	48	H
2001	65	e
2002	6C	l
2003	6C	l
2004	6F	o
2005	00	\0

41

Same Thing!

```
Text:
    .byte 'H'
    .byte 'e'
    .byte 'l'
    .byte 'l'
    .byte 'o'
    .byte 0
```

Created byte by byte

Null character. Marks the end of a string

42

This works too!

```
Text:
    .ascii "Hello"
    .byte 0
```

Directives just create space. So, this creates a byte after the ASCII text.

43

Create a Buffer of Any Size

```
Text:
    .space 30
```

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

44

Create a Buffer of Any Size

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

Create 30 bytes. All of which are 0.

45

Direct Addressing

Using memory… finally

46

Direct Addressing

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

47

Direct Addressing

48

Direct in Java

- The following, for comparison, is the equivalent in Java
- The memory, at the address total, is loaded into rdx

```
// rdx = Memory[total];
mov rdx, total
```


49

Example: Direct Load

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

.text
.global _start
_start:
    mov rdx, funds
```

64 bit integer with an initial value of 100.

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

50

Example: Direct Store

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

.text
.global _start
_start:
    mov rdx, 5000
    mov funds, rdx
```

Store rdx into Address "funds"

51

Example: Direct Store 2

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

.text
.global _start
_start:
    call ScanInt
    mov  funds, rdx
```

You can store inputted values.

52

Direct in Java

- Note: this a shortcut notation
- The full notation would use square brackets
- The assembler recognizes the difference automatically

```
// rdx = Memory[total];
mov rdx, total
```


53

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

```
// rdx = Memory[total];
mov rdx, [total]
```


54

Example: Direct

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

.text
.global _start
_start:
    mov rdx, [funds]
```

A bit more descriptive

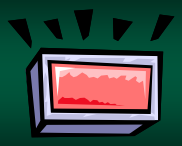
55

Load Effective Address

- Load Effective Address stores the actual address into a register
- It doesn't access memory

```
// rdx = total;
lea rdx, total
```


56



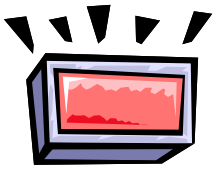
When to use `mov` and `lea`

The difference is huge!

57

When to use `mov` and `lea`

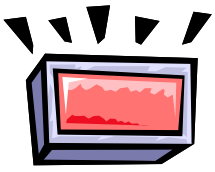
- Knowing when to use an address **or** the data *located at that address* is vital
- Using the wrong one can cause your program to malfunction or crash



58

Cause of the Segmentation Fault

- This is one of the most common mistakes in assembly programming



59

Using Move Correctly

```
.intel_syntax noprefix
.data
Year:
    .quad 1947

.text
.global _start
_start:
    mov  rdx, Year
    call PrintInt
```

Creates 8 bytes

mov loads the data located at the address Year

60

Using move Correctly: Output

```
1947
```

Correct output. mov loaded the data from an address

61

Using `lea` by accident

```
.intel_syntax noprefix
.data
Year:
   .quad 1947

.text
.global _start
_start:
   lea  rdx, Year
   call PrintInt
```

Creates 8 bytes

lea is going to store the address Year into rdx

62

Using `lea` by accident

```
6293248
```

That's wrong… very, very wrong

63

Why it Failed

```
.intel_syntax noprefix
.data
Year:
   .quad 1947

.text
.global _start
_start:
   lea  rdx, Year
   call PrintInt
```

6293232	
6293240	
6293248	1947
6293256	
6293264	

1947 was being stored at this address

64

Sometimes, You Need the Address

- Of course, sometimes, you do need an address
- For example, PrintZString
  - needs to know where the string is located so it can print a series of characters
  - so, it requires an address
  - `lea` is necessary

65

Using `lea` correctly

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

.text
.global _start
_start:
    lea  rdx, Message
    call PrintZString
```

Loads the effective address into rdx

66

Using `lea` correctly: Output

```
Hello!!
```

Correct output. PrintZString went to the address and printed characters

67

Cause of the Segmentation Fault

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

.text
.global _start
_start:
    mov  rdx, Message
    call PrintZString
```

Creates 8 bytes using ASCII values

Used mov rather than lea. rdx is 64-bit (8 bytes)

68

Cause of the Segmentation Fault

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

.text
.global _start
_start:
    mov  rdx, Message
    call PrintZString
```

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

69

Cause of the Segmentation Fault

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

.text
.global _start
_start:
    mov  rdx, Message
    call PrintZString
```

Grabs 8 bytes and creates a huge value

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

70

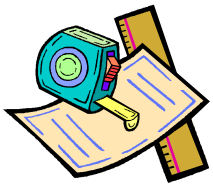
Sizing Instructions

How many bytes are you using?

71

Sizing Instructions

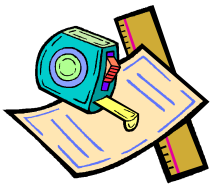
- The Intel can load/store 1-byte, 2-byte, 4-byte or 8-byte values
- The assembler knows *(by looking at the size of the register)* how much many bytes you want to load/store



72

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



73

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.

74

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.

75

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

76

How Many Bytes

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

77

Example: Suffix Used

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

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

Now the assembler knows you mean "move quad".

78

Endianness

The "proper" order of things

79

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?*

80

Example Unsigned Integer (4 Byte)

1,188,852,977

46	DC	74	F1

Most significant Byte (MSB)

Least significant Byte (LSB)

81

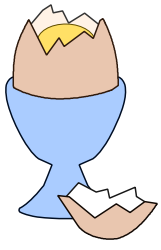
So Many Bytes…

- 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

82

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



83

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

84

Assuming Value is located at 2000

```
Value:
    .quad 74
```

2000	4A
2001	00
2002	00
2003	00
2004	00
2005	00
2006	00
2007	00

Least Significant Byte (LSB)

Little Endian

85

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

86

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

87

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

88

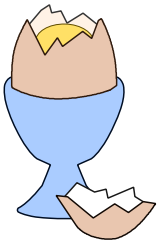
Example File Format Endianness

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

89

So… who is correct?

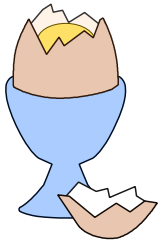
- So, what is the correct and superior format?
- Is it Intel (little endian)?
- …or the PowerPC (big endian) correct?



90

So… who is correct?

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



91

Gulliver's Travels

92