MIPS Memory

* Several slides from Harris and Harris

Memory

- We have already seen memory used for instructions. We now consider using memory to store other data as well
- Basic idea is the same: memory is very large array of bytes
- We can perform two operations on memory: load and store

Why memory?

- Registers
  - Only 32 registers
  - Very fast access
  - Good for commonly used values
- Memory
  - Much larger (several GB)
  - Much slower
  - Cannot perform operations directly on a value in memory

Using memory

- Cannot perform operations directly on a value in memory!
- Need to move value from memory into a register before performing operation
- Result of operation can then be written from register to memory
- Only access memory via reading (*load*) or writing (*store*)

Memory Layout

- Previously considered memory as byte-addressable array
- This is true, but not the whole story
- Typically want to access 4-bytes at a time (why?)

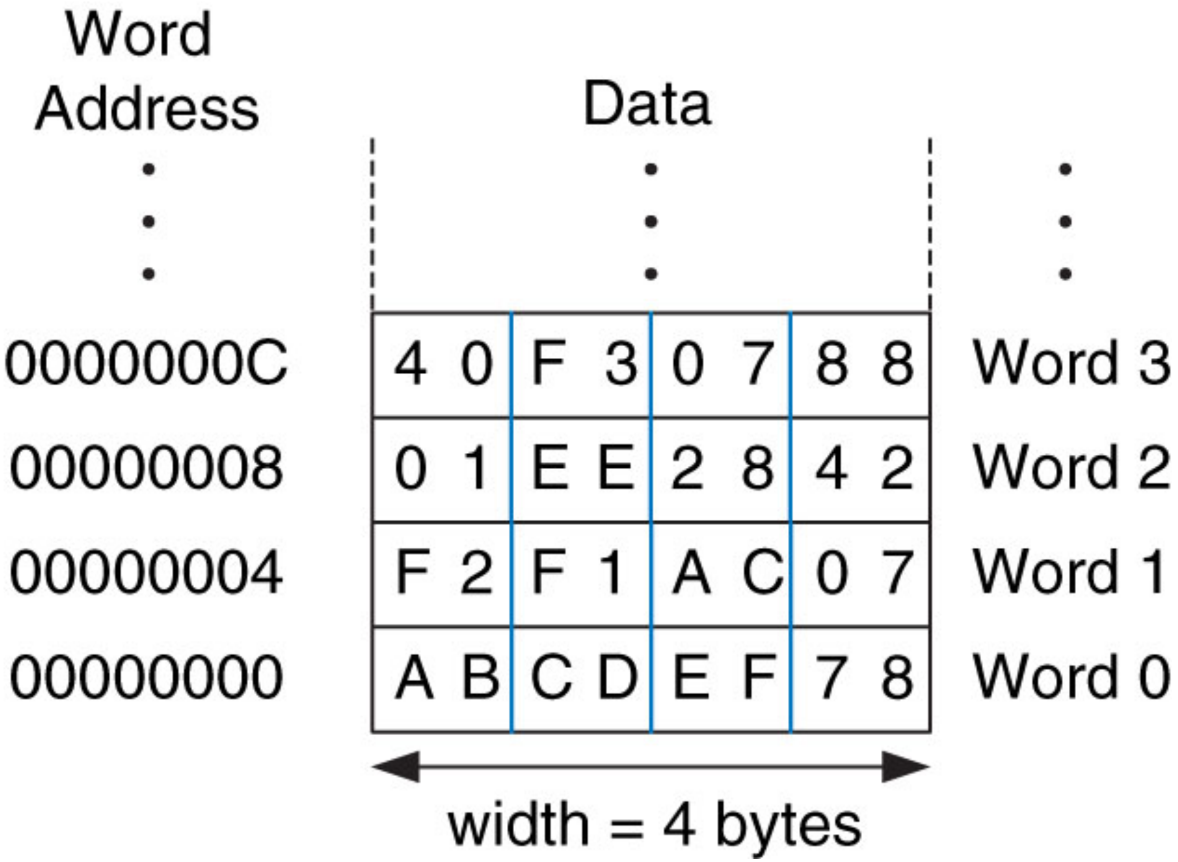
Memory Layout

- Previously considered memory as byte-addressable array
- This is true, but not the whole story
- Typically want to access 4-bytes at a time (why?)
  - MIPS is 32-bit (4-byte) architecture
  - We store data in 4-byte registers
  - ALU takes 4-byte operands
- We saw same idea with PC – grab entire instruction (4 bytes) at once

Memory Layout

- Memory is split into bytes, but also split into words
- We generally access as words
- Access words with `lw` and `sw` commands (*load word* and *store word*)
  - Use 1 byte with `lb` and `sb` (*load byte* and *store byte*)
- Words are still byte-addressable! Grab particular word by specifying first byte in that word

Memory


Word Address
Data
0000000C
4 0 F 3 0 7 8 8
Word 3
00000008
0 1 E E 2 8 4 2
Word 2
00000004
F 2 F 1 A C 0 7
Word 1
00000000
A B C D E F 7 8
Word 0
width = 4 bytes

Word alignment

- Calls to `ls` and `sw` must be **word-aligned**
- Nothing stops us from calling `lw` with `0x00000001` as argument, but MIPS will refuse because address is not multiple of 4
- Remember – words are addressed by location of first byte, *not* by "word number"

Using memory instructions

- Way we specify memory location to load/store will seem odd at first
- See later that it is designed to make common assembly tasks faster and easier
- Critically important to understand how memory addressing works

Reading Byte-Addressable Memory*

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

  ```
  lw $s0, 5($t1)
  ```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 5)

Any register may be used as base address

* Modified title

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after load

MIPS assembly code

```
lw $s3, 4($0)   # read word at address 4 into $s3
```

Word Address	Data				
⋮	⋮				⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

width = 4 bytes

Writing Byte-Addressable Memory

- **Example:** stores the value held in `$t7` into memory address 0x2C (44)

MIPS assembly code

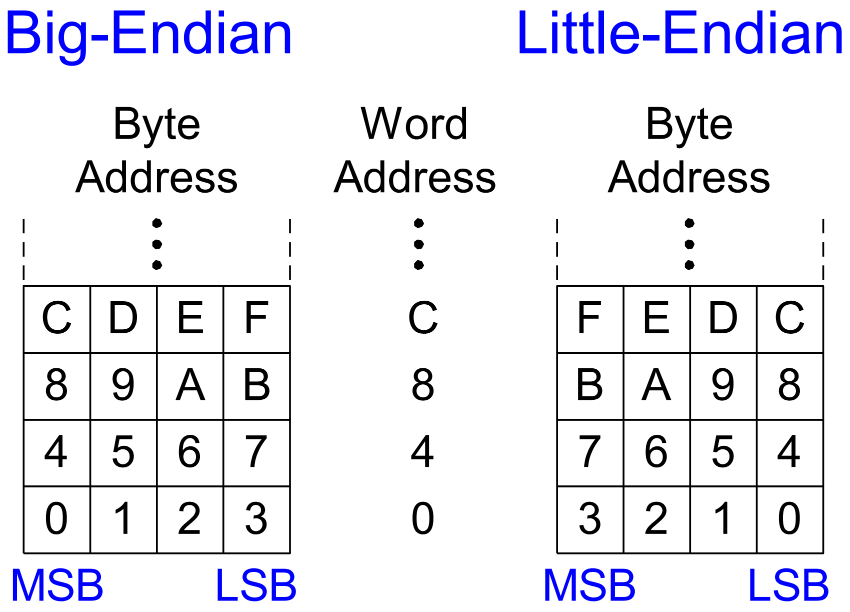
```
sw $t7, 44($0)   # write $t7 into address 44
```

Word Address	Data				
⋮	⋮				⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

width = 4 bytes

Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



Big-Endian & Little-Endian Memory

- Jonathan Swift’s *Gulliver’s Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn’t really matter which addressing type used – except when the two systems need to share data!

Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?
- In a little-endian system?

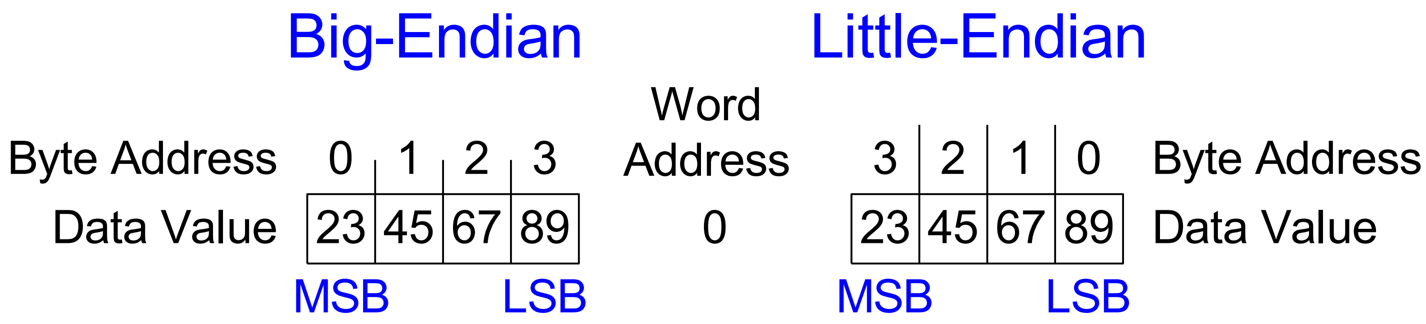
```
sw $t0, 0($0)
lb $s0, 1($0)
```

Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?
- In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Big-Endian & Little-Endian

Distinction matters only when working with individual bytes

`lb` and `sb` depend on endian-ness of system

`lw` and `sw` work the same regardless of endian-ness

- For example, most-significant byte of word always read into MSB of register when `lw` used