

# CSCI 210: Computer Architecture

## Lecture 5: MIPS

Stephen Checkoway

Oberlin College

Slides from Cynthia Taylor



# CS History: Nintendo 64



- Released in 1996
- Named after its 64-bit CPU
- Silicon Graphics Inc (SGI) adapted MIPS chips for supercomputers to use less power and be cheaper
- "At the heart of the system will be a version of the MIPS Multimedia Engine, a chip-set consisting of a 64-bit MIPS RISC microprocessor, a graphics co-processor chip and Application Specific Integrated Circuits" (SGI press release, 1993)
- "If it works at all, it could bring MIPS to levels of volume [SGI] never dreamed of." (Michael Slater, Microprocessor Report)
- Super Mario 64 features a rabbit named MIPS after the processor

# Review: Memory Instructions

- `lw $t0, 0($t1)`
  - `$t0 = Mem[$t1+0]`
  - Loads 4 bytes from `$t1`, `$t1+1`, `$t1+2`, and `$t1+3`
- `sw $t0, 4($t1)`
  - `Mem[$t1+4] = $t0`
  - Stores 4 bytes at `$t1+4`, `$t1+5`, `$t1+6`, and `$t1+7`
- These instructions are the cornerstones of our being able to go to and from memory

Review: If you have a pointer to address 1000 and you increment it by one to 1001. What does the new pointer point to, relative to the original pointer?

- A) The next word in memory
- B) The next byte in memory
- C) Either the next word or byte – depends on if you use that address for a load byte or load word
- D) Pointers are a high level construct – they don't make sense pointing to raw memory addresses.
- E) None of the above.

If a 4-byte word is in memory at address 4203084, what is the address of the next word in memory?

- A) 4203085
- B) 4203088
- C) 14203084
- D) It depends on the value of the words in memory
- E) Since a word is 4 bytes, it's not possible to have one at address 4203084

# Arrays

- Arrays are stored consecutively in memory
- The **base** address points to the first element in the array
- Accessing other elements in the array requires adding an **offset** to the base address
  - The offset to use is the index of the array element \* the size of one element

# Memory Operand Example 1

- C code:

```
g = h + A[8];
```

– g in \$s1, h in \$s2, base address of A in \$s3, A is an array of 4 byte ints

- Compiled MIPS code:

– Index 8 requires offset of 32

```
lw    $t0, 32($s3)
```

```
add   $s1, $s2, $t0
```

# Translate to MIPS

- C code: `g = h + A[5];`
  - `g` in `$s1`, `h` in `$s2`, base address of `A` in `$s3`.
  - `A` is an array of 4-byte ints

A. 

<pre>lw  \$t0, 5(\$s3) add \$s1, \$s2, \$t0</pre>
---

B. 

<pre>lw  \$t0, 20(\$s3) add \$s1, \$s2, \$t0</pre>
--

C. 

<pre>lw  \$t0, \$s5 add \$s1, \$s2, \$t0</pre>
--

D. 

<pre>lw  \$t0, \$s3 add \$s1, \$s2, \$t0</pre>
--



# Memory Operand Example 2

- C code:

`A[12] = h + A[8];`

– `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

– Index 8 requires offset of 32

```
lw    $t0, 32($s3)    # load word
```

```
add   $t0, $s2, $t0
```

```
sw    $t0, 48($s3)    # store word
```

When a 2-byte integer is stored in byte-addressed memory (occupying two consecutive bytes), is the most significant byte (MSB) stored in the lower address or the higher address?

A. High

1000	0000 1111
1001	0000 0000

= 15

B. Low

1000	0000 0000
1001	0000 1111

= 15

C. It Depends

# Byte ordering

- Big-endian: Most significant byte in lowest address
  - **MIPS**, Network byte order, Motorola 68000, PowerPC (usually), ...
- Little-endian: Most significant byte in highest address
  - Intel x86, x86-64, ARM (usually), ...
- Bi-endian: Switchable between big and little endian
  - ARM, PowerPC, Alpha, SPARC, ...
- Middle-endian/mixed-endian
  - Bytes not stored in either order, at least in some cases
  - Words stored one endian, bytes within words stored another endian
  - PDP-11 stored 16-bit words in little-endian order, but 32-bit double words as pairs of 16-bit words with the most significant word of the double word stored first (i.e., big-endian order)

Big-endian means most significant byte/digit/piece comes first, little-endian means least significant byte/digit/piece comes first. Mixed-endian means not in order.

Which row of the table correctly identifies the endianness of date formats?

	US (MM-DD-YYYY)	Most of the world (DD-MM-YYYY)	ISO 8601 (YYYY-MM-DD)
A	Little	Mixed	Big
B	Big	Little	Mixed
C	Mixed	Little	Big
D	Mixed	Big	Little
E	Little	Big	Mixed

# Questions about Memory?

# Immediate Operands

- Constant data specified in an instruction
  - `addi $s3, $s3, 4`
  - `li $t0, -25`      `# Pseudoinstruction: addi $t0, $zero, -25`
  - `ori $v0, $t8, 1`

# Subtract 2 from \$s0 and store in register \$s1

A. `addi $s0, $s1, -2`

B. `addi $s1, $s0, -2`

C. `subi $s0, $s1, 2`

D. `subi $s1, $s0, 2`

E. More than one of the above

# Pseudoinstructions

- `move dest, src`                     $\Rightarrow$  `add dest, $zero, src`
- `subi dest, src, imm`                 $\Rightarrow$  `addi dest, src, -imm`
- `li dest, imm`                         $\Rightarrow$  `addi dest, $zero, imm`
- More complicated expansions are possible when the immediate value is too large, MARS simulator will show you how it expands pseudoinstructions



# MIPS Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - **allow instructions to contain small immediate operands**

# Loading a large number into a register

- immediates are limited to 16 bits
  - -32768 to 32767 or 0 to 65535
- Numbers outside this range need to be loaded into registers before being used
- load upper immediate instruction sets the most-significant 16 bits of a register
  - `lui $t0, 0x1234`  
  `ori $t0, $t0, 0x5678`
- When li is given a value that's too large, the assembler expands it to lui/ori

# MIPS Questions?

# Why we need to learn binary (and other number systems)

- Fundamental to how your computer works
  - Will need a good grasp of binary to understand things like logical operations
  - Will need it a lot when we get to logic gates and how the CPU works
  - Will need to translate to binary to work out examples
- Need to understand it to understand many things like network protocols (IP addresses), bit masking, etc.

# Positional Notation

- The meaning of a digit depends on its position in a number.
- A number, written as the sequence of digits  $d_n d_{n-1} \dots d_2 d_1 d_0$  in base  $b$  represents the value

$$d_n * b^n + d_{n-1} * b^{n-1} + \dots + d_2 * b^2 + d_1 * b^1 + d_0 * b^0$$

# Consider 101

- In base 10, it represents the number 101 (one hundred one)
- In base 2,  $101_2 =$
- In base 8,  $101_8 =$

$$101_5 = ?$$

A. 26

B. 51

C. 126

D. 130

$$122_{-3}=?$$

A. 17

B. 5

C. 10

D. -30



# Reading

- Next lecture: Number Representation
  - Section 2.4
- Problem Set 1 – due Friday