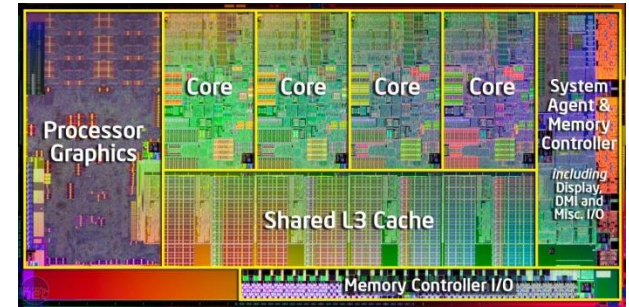




고려대학교
KOREA UNIVERSITY



COSE22, COMP212 Computer Architecture

Lecture 4. MIPS Instructions #2

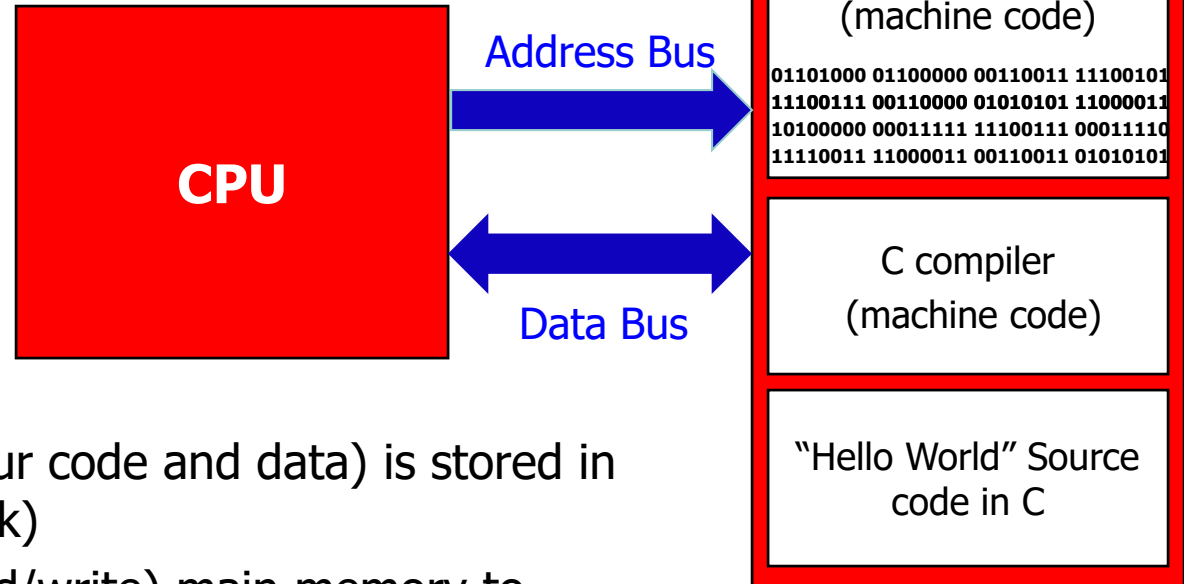
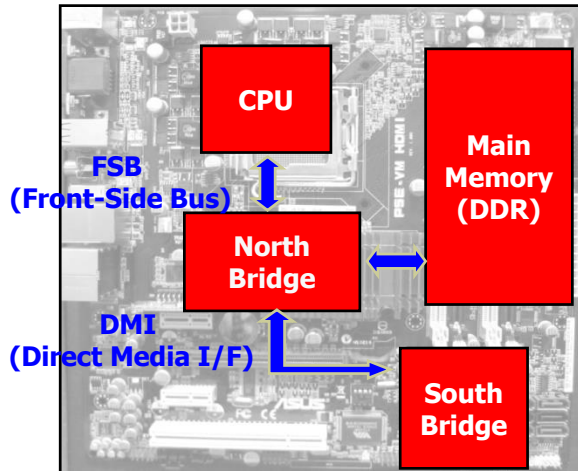
Memory Access (Load/Store) Instructions

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

Why Should CPU Access Memory?

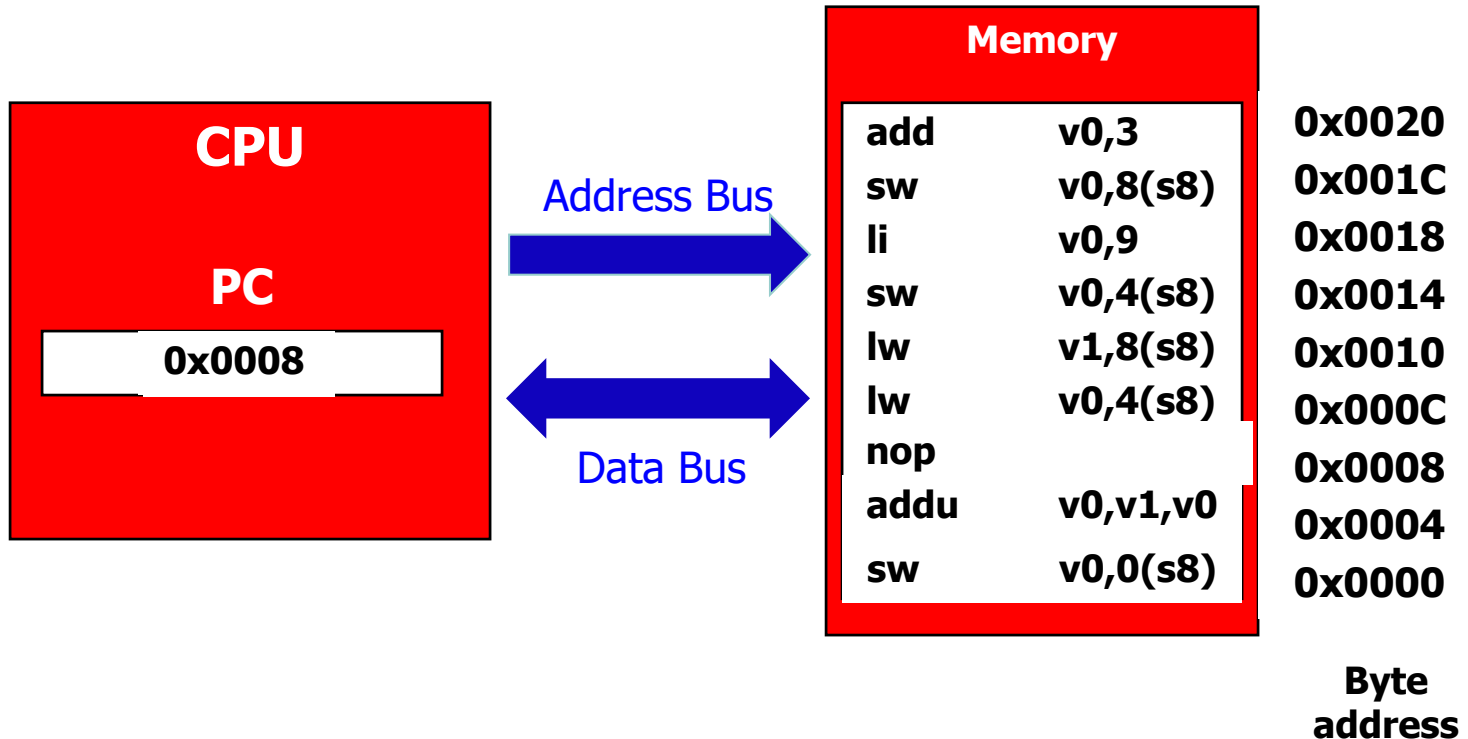


- Initially, everything (your code and data) is stored in main memory (hard-disk)
- CPU should access (read/write) main memory to execute your program
- So, 2 purposes
 - Instruction read:** CPU reads instructions from memory
 - Data read/write:** CPU reads data from memory and writes data to memory

Instruction Access (Read)

- How does CPU read instructions from main memory?
 - **Every CPU** has a special register that keeps track of the current instruction's address in execution
 - Many CPUs use the same term, **Program Counter (PC)**
 - An exception is x86, where the term IP (instruction pointer) is used
 - MIPS also has a 32-bit special register called **PC** inside the CPU
 - PC is initialized with a predefined address at reset
 - x86 fetches the first instruction from 0xFFFF_FFF0
 - ARM fetches the first instruction from 0x0000_0000
 - MIPS initializes PC to 0xBFC0_0000
 - Then, PC gets changed as (or after) each instruction gets executed

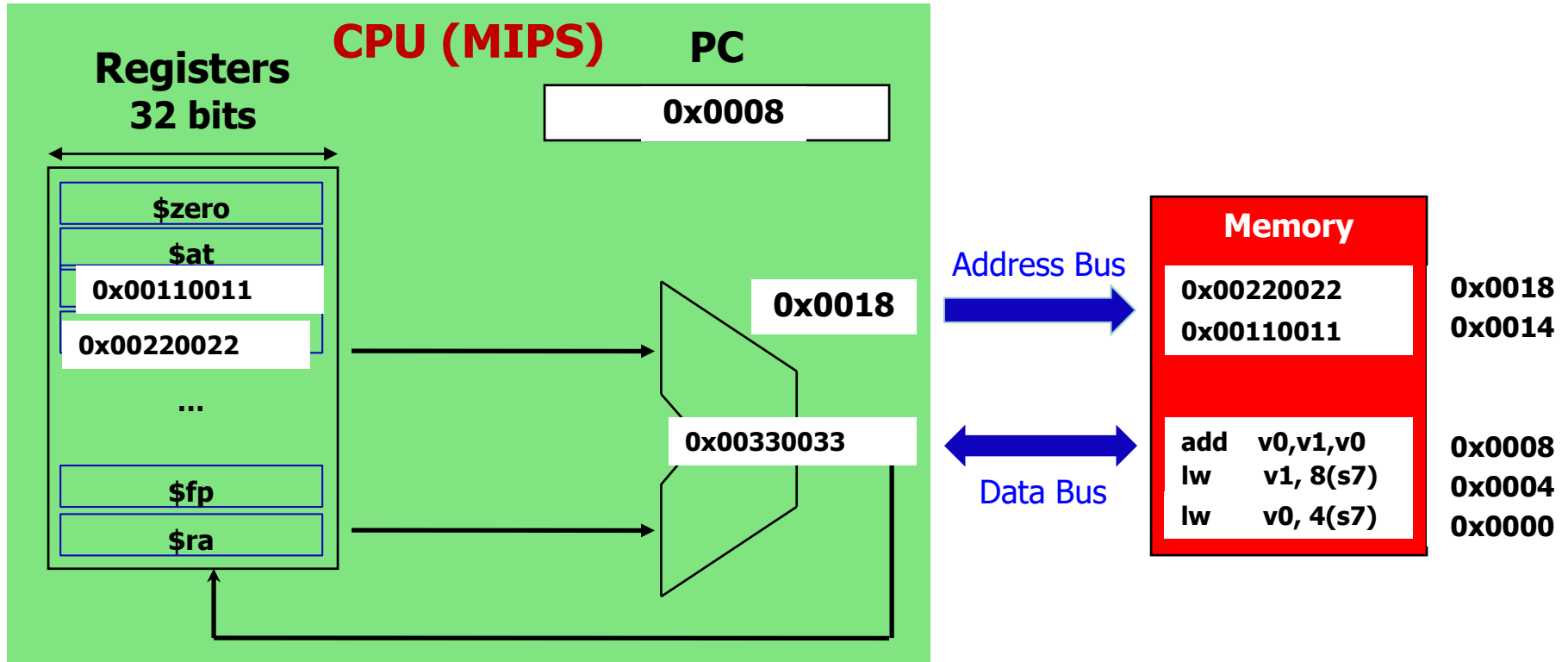
Instruction Access Illustration



Data Read/Write

- As mentioned, everything (your code and data) is stored in main memory
 - You define data to be used in your code (and you sometimes define complex data structures such as arrays and structs)
 - Examples: `int a, b, c;    int dummy [100];`
- CPU has a limited number of registers inside
 - There are many data in your program, much more than registers inside CPU can accommodate
 - In MIPS, there are 32 general-purpose registers inside the CPU
 - So, CPU is able to keep only a small amount of data in registers on the fly
- Thus, MIPS (and all other CPUs) must provide instructions that transfer data between memory and registers
 - Such instructions are called **memory access instructions**
 - To access data in memory, the data transfer instruction should supply the **memory address**

Data Transfer Illustration



- Assume that \$s7 contains 0x0000_0010

Word

- **Word** is a term for the natural unit of data used by a particular computer design
 - A word is simply a fixed-sized group of bits that are handled together
 - The number of bits in a word is an important characteristic of a computer architecture
 - The size of a word is reflected in many aspects of a computer's structure and operation
 - The majority of the registers in the computer are usually word-sized
- Modern computers usually have a word size of 32, or 64 bits
 - **The word size of MIPS is 32 bits (4 Bytes)**
 - In x86 case, the word size is still 16-bit because of the historical reason (backward compatibility) even though your computer (core2duo, corei7) is a 64-bit CPU based machine
- **Alignment restriction:** the memory address of a **word** must be on natural word boundaries (a multiple of 4 in MIPS-32)
 - For example, memory address of accessing a word should be 0x0000_0000, 0x0000_1004, 0x1234_567C etc

Memory Address

- Byte-address

Main Memory (64KB)

.....	0x000C
Byte	0x000B
Byte	0x000A
Byte	0x0009
Byte	0x0008
Byte	0x0007
Byte	0x0006
Byte	0x0005
Byte	0x0004
Byte	0x0003
Byte	0x0002
Byte	0x0001
Byte	0x0000

**Byte address
in hex**

- Word-address

Main Memory (64KB)

.....	0x000C	0x0003
Word (4 Bytes)	0x0008	0x0002
Word (4 Bytes)	0x0004	0x0001
Word (4 Bytes)	0x0000	0x0000

**Byte address Word address
in hex in hex**

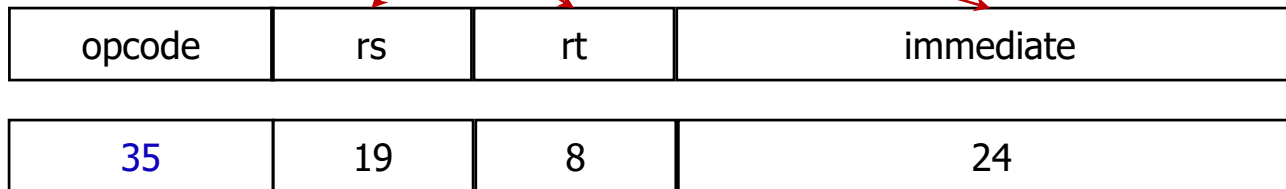
lw

- lw reads a word (32-bit) from memory and loads into a register
- I format Instruction

lw rt, address

- Example:

lw \$t0, 24 (\$s3) # load (read) **word** from memory
 # \$t0 <= [\$s3 + 24]



MIPS architect defines the opcode

binary 100011 10011 01000 00000 00000 011000
 1000**11** 100**11** 0**1**000 00000 00000 01**1**000

hexadecimal 0x8E68 0018

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

lw example

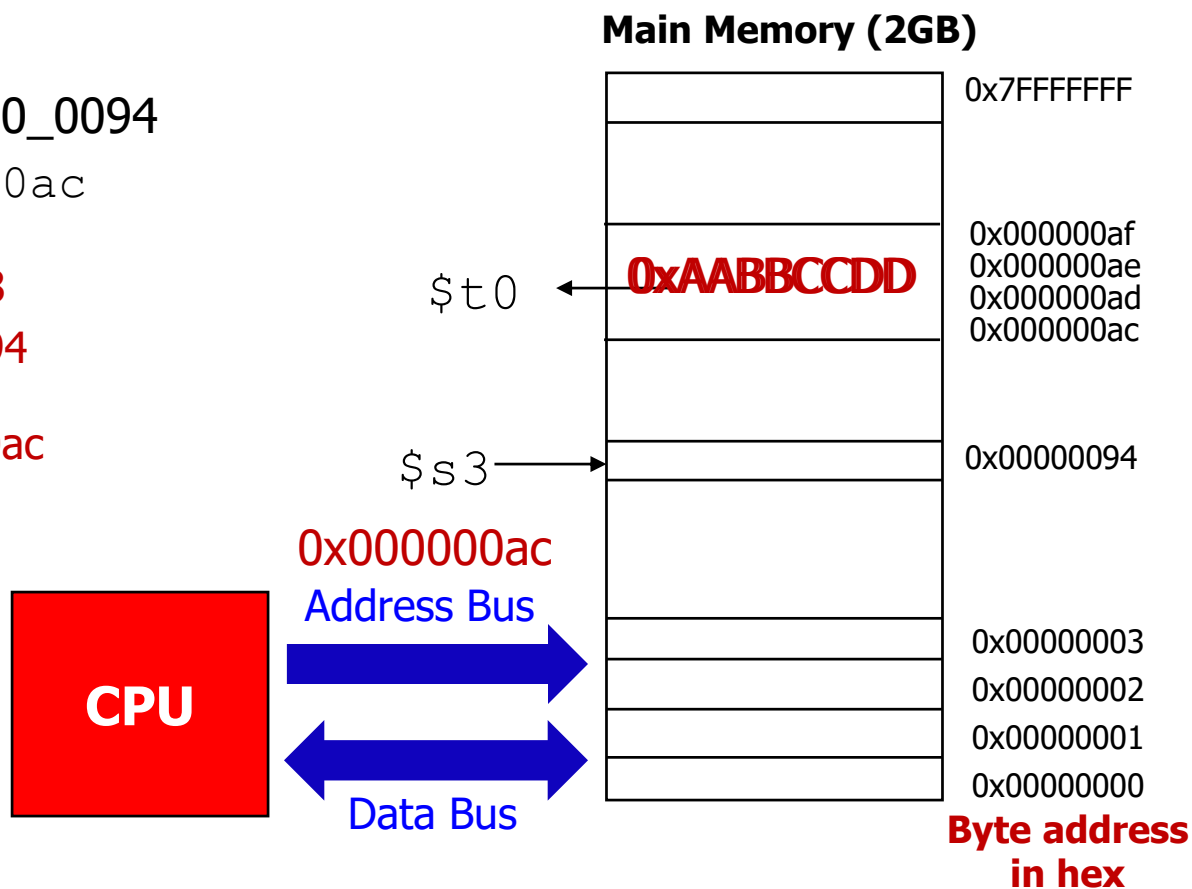
```
lw    $t0, 24($s3)    #load (read) word from memory
                        # $t0 <= [$s3 + 24]
```

Assume that \$s3 has 0x0000_0094

$\$s3 + 24_{10} = 0x0000_00ac$

0001 1000 // 24 = 0x18
+ ... 1001 0100 // \$s3 = 0x94

... 1010 1100 = 0x000_000ac

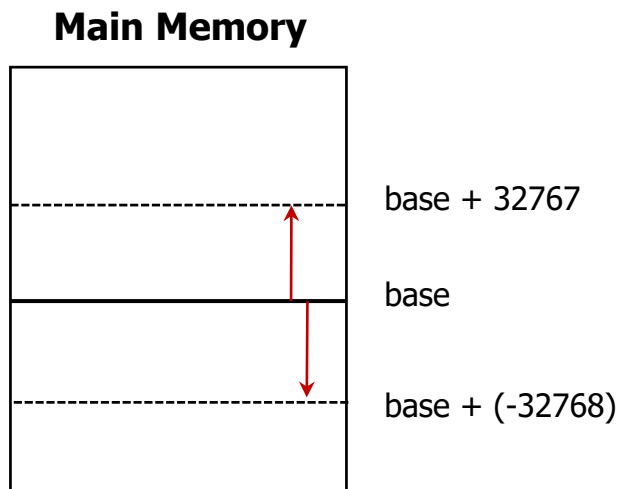


lw (Cont)

- `lw` is an I format instruction

opcode	rs	rt	immediate
--------	----	----	-----------

- The memory address (32 bit address) is formed by adding the contents of the **base address register** to the **offset**
 - In `lw $t0, 24($s3)`, the base is `$s3` and the offset is 24
 - The immediate specified in an instruction is a 16-bit 2's complement number in the range `[-32768, 32767]`



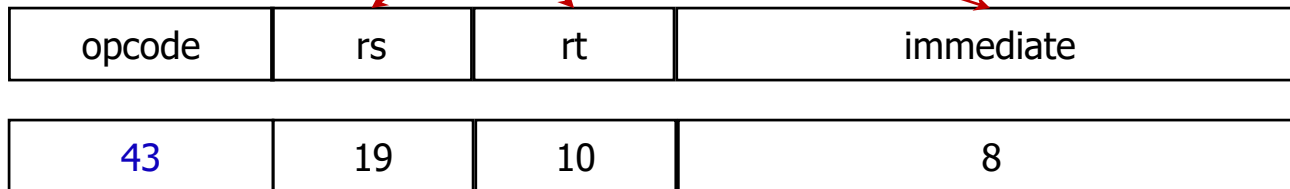
SW

- `sw` stores (writes) a word (32-bit) from a register to main memory
- I format instruction

`sw rt, address`

- Example:

`sw $t2, 8($s3) # store(write) word to memory`
`# [$s3 + 8] <= $t2`



MIPS architect defines the opcode

binary 101011 10011 01010 00000 00000 001000
 101011 10011 01010 00000 00000 001000

hexadecimal 0xAE6A 0008

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Loading and Storing Bytes

- Since bytes (8 bits) are so useful, most architectures provide capability of addressing and accessing individual **bytes** in memory
- Let's go over byte-addressable instructions in MIPS
 - `lb`, `lbu`, `sb`

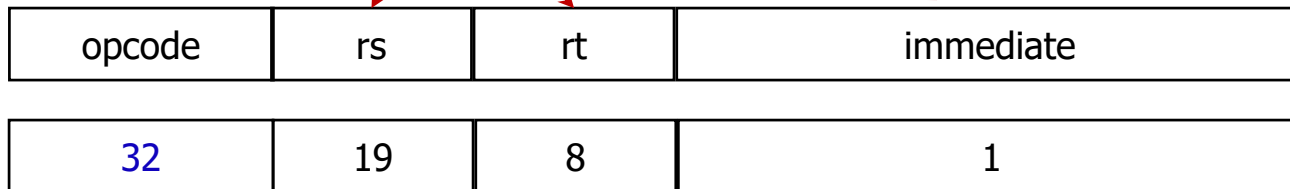
lb

- lb reads a byte (8-bit) from memory and loads into a register
- I format instruction

lb rt, address

- Example:

lb \$t0, 1(\$s3) # load (read) **byte** from memory
 # \$t0 <= [\$s3 + 1]



binary 100000 10011 01000 00000 00000 000001
 100000 10011 01000 00000 00000 000001

hexadecimal 0x8268 0001

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

1b - Where to Load and How?

```
lb      $t0, 1($s3)  # load (read) byte from memory
                        # $t0 <= [$s3 + 1]
```

- Byte is loaded into the LSB (Least Significant Byte) of the register `$t0` and sign-extended



- If you don't want to have it sign-extended, use the `lbu` (Load byte unsigned) instruction

```
lbu     $t0, 1($s3)  # load (read) byte from memory
                        # $t0 <= [$s3 + 1]
```

opcode	rs	rt	immediate
36	19	8	1

sb

- sb writes a byte (8-bit) to memory from a register
- I format instruction

sb rt, address

- Example:

sb (\$t0, -7(\$s3)) # store (write) **byte** to memory
 # [\$s3 + (-7)] <= \$t0

opcode	rs	rt	immediate
40	19	8	-7

binary 101000 10011 01000 11111 11111 111001

101000 10011 01000 11111 11111 111001

hexadecimal 0xA268 FFF9

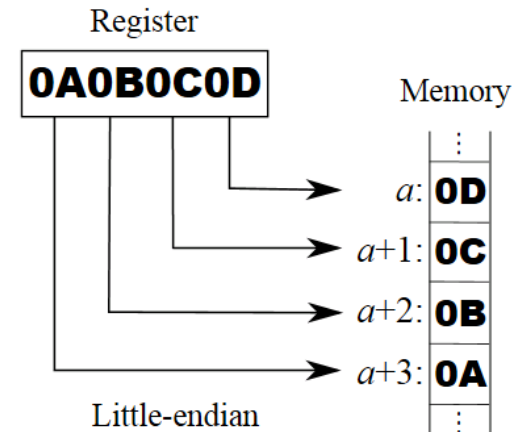
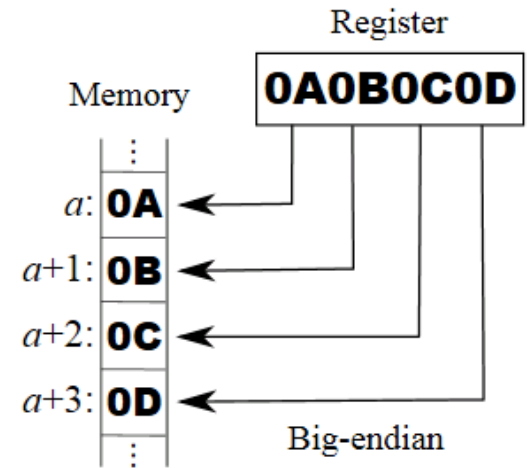
Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

sb - Where to Stored and from Where?

- `sb` takes the byte from LSB of a register and write it to a byte in memory
 - It does not change the other bits in a word in memory (no sign-extension!)

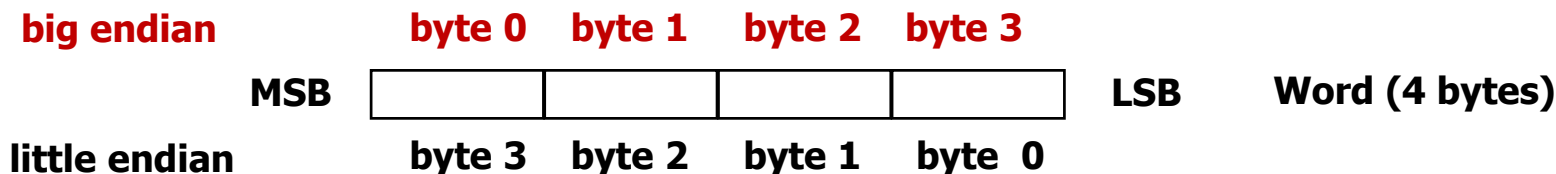
Endianness

- The terms **endian** and **endianness** refer to the convention used to interpret the bytes consisting of a data word (for example, 32-bit (4-byte))
- When reading or writing a byte in a data word, the order of the bytes stored in memory determines the interpretation of the byte



Endianness

- Byte addressable memories are organized in a **big-endian or little-endian** fashion. In both formats,
 - The most significant byte (MSB) of a word is on the left
 - The least significant byte (LSB) of a word is on the right
- **Endian**
 - In **big-endian** machines, **bytes** are numbered starting with byte 0 at MSB
 - Examples: IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
 - In **little-endian** machines, bytes are numbered starting with byte 0 at LSB
 - Examples: **Intel x86**, DEC Vax, DEC Alpha (Windows NT)

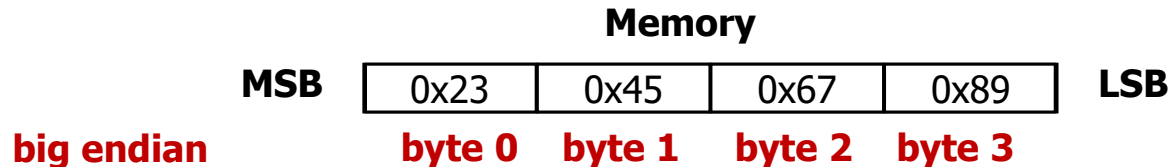


- The choice of Endianness is completely arbitrary, but leads to hassles when sharing data between big-endian and little-endian computers
 - Why so messy? Blame early computer designers!

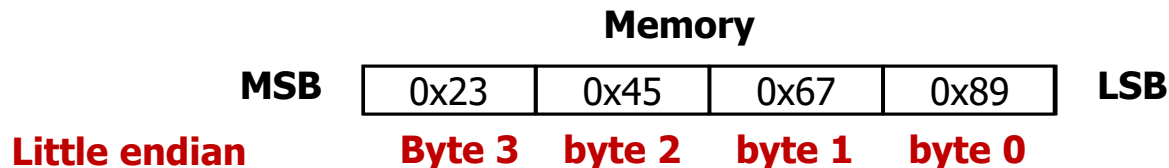
Endianness Example

- Suppose that `$s0` initially contains `0x23456789`. After running the following program on a big-endian machine, what value does `$s0` contain?

```
sw    $s0, 0($0) # [0 + $0] <= $s0
lb    $s0, 1($0) # $s0 <= [1 + $0]
```



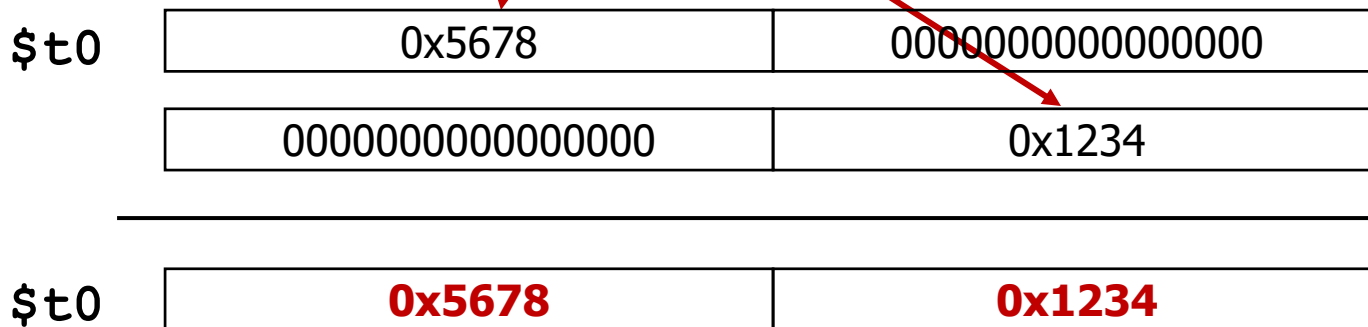
- How about in a little-endian machine?



How to load a 32-bit constant?

- How to load a 32-bit constant into a register?
 - Use `lw` instruction
 - Use 2 instructions (`lui`, `ori`)
 - `lui` (load upper immediate)
 - `ori` (or immediate)
- Example:

```
lui $t0, 0x5678
ori $t0, $t0, 0x1234
```



Pseudo Instructions

- MIPS defines pseudo instructions that are not actually part of the instruction set, but are commonly used by programmers and compilers (and assemblers)
 - So, pseudo instructions are provided for the programming convenience

Pseudo Instructions	Corresponding MIPS Instructions
<code>li \$s0, 0x1234AA77</code> (Load Immediate)	<code>lui \$s0, 0x1234</code> <code>ori \$s0, \$s0, 0xAA77</code>
<code>la \$s0, op1</code> (Load Address)	<code>lui ...</code> <code>ori ...</code>
<code>move \$s2, \$s1</code>	<code>add \$s2, \$s1, \$0</code>
<code>nop</code>	<code>sll \$0, \$0, 0</code>