

CS/ECE 5381/7381
Computer Architecture
Spring 2023

Dr. Manikas
Computer Science
Lecture 4: Jan. 26, 2023

LOCKDOWN BROWSER

- Notes from this week's announcement
- Our exams will use Lockdown Browser
- Exam 1 will be in three weeks (**Feb. 16**)
- Please install and test Lockdown Browser on your computer
- Use Practice Exam to verify set up and to get familiar with Lockdown Browser

Assignments

- Quiz 2 – due Sat., Jan. 28 (11:59 pm)
 - Covers concepts from Module 2 (this week)

Quiz 2 Details

- The quiz is open book and open notes.
- You are allowed 90 minutes to take this quiz.
- You are allowed 2 attempts to take this quiz - your highest score will be kept.
 - Note that some questions (e.g., fill in the blank) will need to be graded manually
- Quiz answers will be made available 24 hours after the quiz due date.

Instruction Set Principles

(Appendix A, Hennessy and Patterson)

Note: some course slides adopted from
publisher-provided material

Outline

- A.1 Introduction
- A.2 Classifying Instruction Set Architectures
- A.3 Memory Addressing
- A.4 Type and Size of Operands
- A.5 Operations in the Instruction Set
- A.6 Instructions for Control Flow
- A.7 Encoding an Instruction Set
- A.9 MIPS Architecture

Memory Addressing

- For any ISA class, must define how memory is addressed
 - Address interpretation
 - Address specification

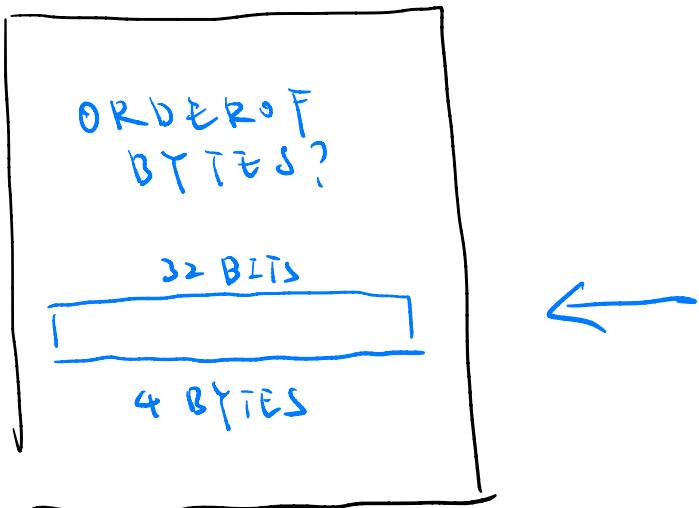
Interpreting Memory Addresses

- Memory size
 - $2^{10} = 1 \text{ K}$ (kilo)
 - $2^{20} = 1 \text{ M}$ (mega)
 - $2^{30} = 1 \text{ G}$ (giga)
 - $2^{40} = 1 \text{ T}$ (tera)
 - $2^{50} = 1 \text{ P}$ (peta)
- 1 B (byte) = 8 b (bits)

Word Size

- CPU word size = w
 - Typical: $w = 16$ or 32 bits
- Memory word size = s
 - Typical: $s = 8$ bits (*memory is byte-addressable*)
- Need to identify order of memory words in CPU word

PROCESSOR
(CPU)

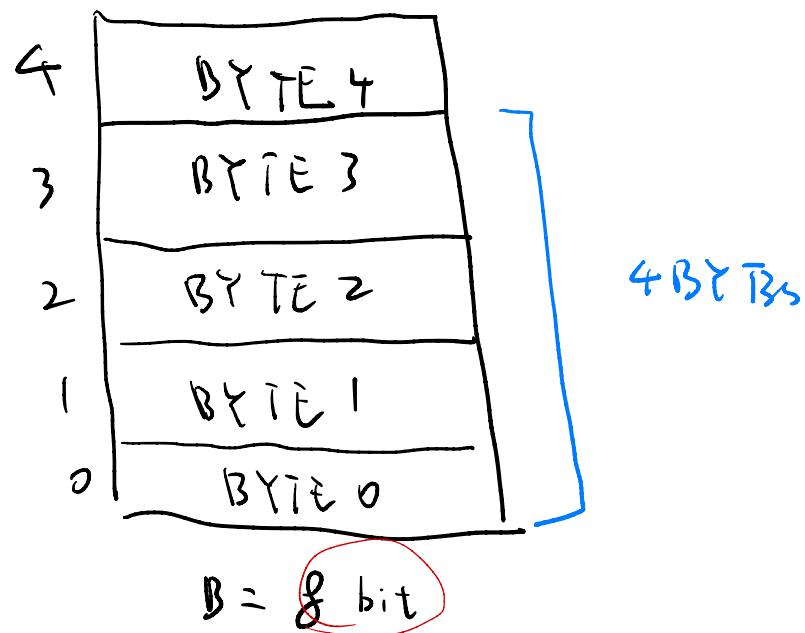


PROCESSOR

WORD SIZE = 32 BITS

MEMORY

16 GB



MEMORY WORD SIDE

8 bits = 1 byte

Something we have to define our architecture when we're designing our computers we have to come up with a standard and of course stick to it. when we write back to memory when I'm updating memory I got to make sure I keep the ordering correct.



Word-Ordering Schemes

- Big Endian *most significant byte*
 - Stores the MSB ("the big end") at the **lower** byte address and the LSB at the **higher** byte address
 - Little Endian *least significant byte*
 - Stores the LSB ("the little end") at the **lower** byte address and the MSB at the **higher** byte address
 - Example:
 - store ABCD1234H at word address location 0
 - AB is MSB, 34 is LSB
- hexadecimal 4 bytes . A B is the most significant byte.*
3 4 is the least significant byte.

Memory Ordering Example

most computer systems use big Endian.
Because it makes more sense : it's easier for us programmers trying to debug and read memory that's an order that we'd expect it.

Big Endian

Byte Address	0	1	2	3
Data	AB	CD	12	34

Little Endian

Byte Address	0	1	2	3
Data	34	12	CD	AB

ARM processor program that uses a little Endian

Addressing Modes

- What object is being addressed?
 - Memory location?
 - Specified by *effective address*
 - Register?
 - Specified by a *register number*
 - Constant?
 - Constant value contained in instruction

Register Mode

- All values in registers
- ADD R0, R1
 - $[Rx]$ = contents of register x
 - $[R0] \leftarrow [R0] + R[1]$

Immediate Mode

- For constants
- ADD R0, #3
 - $[R0] \leftarrow [R0] + 3$

Register Indirect Mode

间接

- Use a register value as a pointer to memory
- ADD R0, (R1)
 - $[R0] \leftarrow [R0] + \text{Mem}[[R1]]$
 - Where $\text{Mem}[x]$ is memory location x
 - Register R1 contains the value for x

Indexed Mode

- Add a value (index) to the memory address
- ADD R0, (R1+R2)
 - $[R0] \leftarrow [R0] + \text{Mem}[[R1] + [R2]]$
- Used in array addressing
 - R1 contains base array value
 - R2 contains index value

Outline

- A.1 Introduction
- A.2 Classifying Instruction Set Architectures
- A.3 Memory Addressing
- A.4 Type and Size of Operands
- A.5 Operations in the Instruction Set
- A.6 Instructions for Control Flow
- A.7 Encoding an Instruction Set
- A.9 MIPS Architecture

Type and Size of Operands

- Character 8 bits ASCII
- Half word (16 bits)
- Word (32 bits)
- Single-precision floating point (32 bits)
- Double-precision floating point (64 bits)

a lot of modern computers or 32-bit machines certainly the mips will study also the arm is a 32-bit machine, most modern processor 32-bit so sometimes they use half words versus 16 bits

OPERAND

TYPES

- SINGLE - PRECISION

FLOATING POINT

- DOUBLE - PRECISION

C / C++ / Java

float x, y;

SINGLE - PRECISION

double a, b;

DOUBLE - PRECISION

math.h → Functions are double

Outline

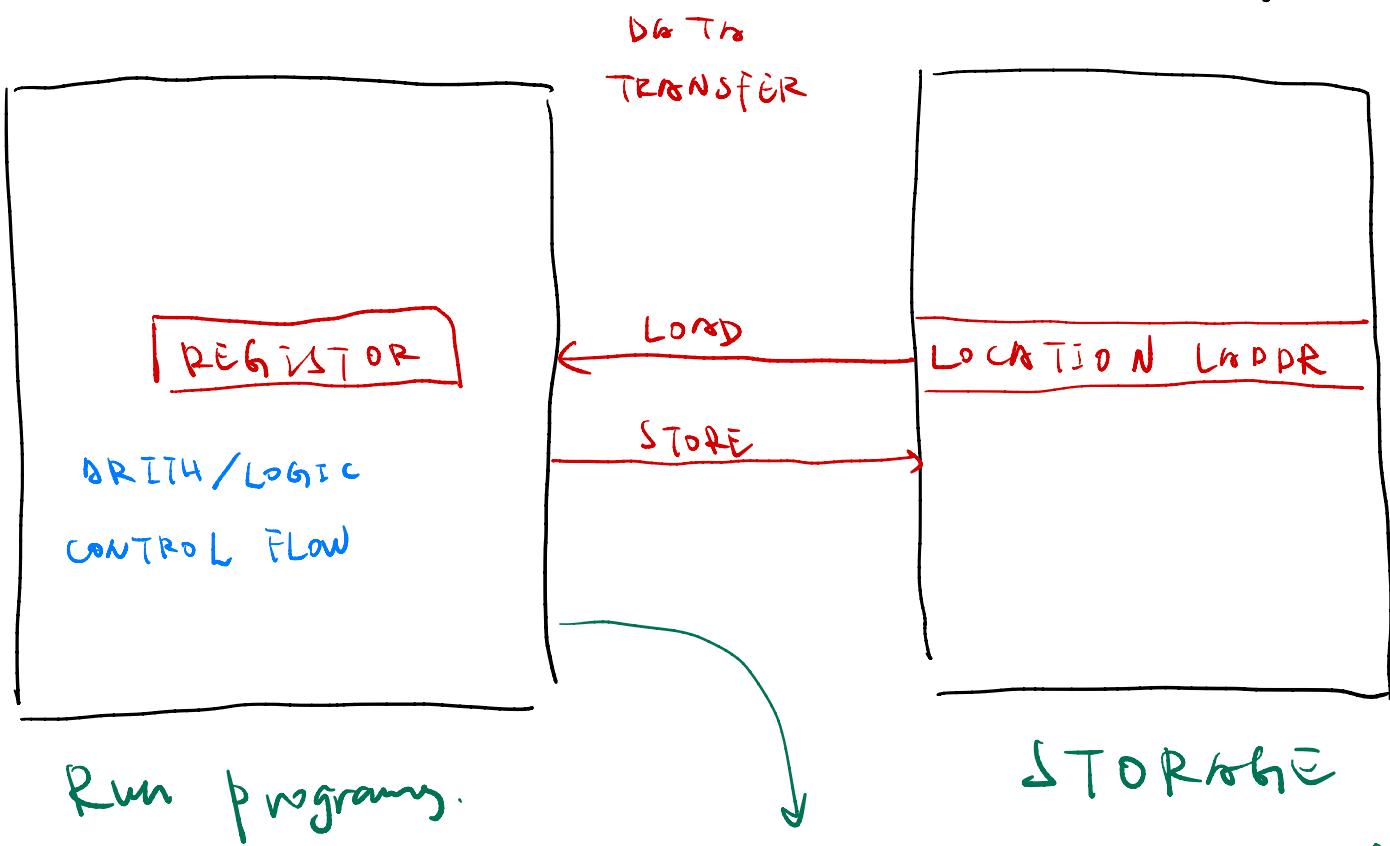
- A.1 Introduction
- A.2 Classifying Instruction Set Architectures
- A.3 Memory Addressing
- A.4 Type and Size of Operands
- A.5 Operations in the Instruction Set
- A.6 Instructions for Control Flow
- A.7 Encoding an Instruction Set
- A.9 MIPS Architecture

Operations in the Instruction Set

- Data Transfer (load/store)
- Arithmetic/Logic
 - Add, subtract, multiply, divide, negation (2's comp)
 - And, or, not (1's comp)
- Control flow (branch, jump)
- Other
 - System, graphics, etc.

Anytime I get on a new processor the first thing I do
see if I can add two and two
get four. That's my little test
there. Of course if that doesn't
work then I don't want to
use that processor.
anyone.
(quality control)

PROCESSOR



Memory

We'll look here to see how parallelism affects the memory systems, but this one has a relatively simpler job compared to the processor. The process is doing a lot of more work. The memory main thing is to get to make sure the processor could find it efficiently.

Storage

So this is doing a lot of work here, now throughout this course as we get to parallel processing. We're going to be looking spending a lot of time here. It's got

inside the processor of course this is to figure out how do I execute the code, how do instructions where we do the arithmetic and logic operations

Outline

- A.1 Introduction
- A.2 Classifying Instruction Set Architectures
- A.3 Memory Addressing
- A.4 Type and Size of Operands
- A.5 Operations in the Instruction Set
- A.6 Instructions for Control Flow
- A.7 Encoding an Instruction Set
- A.9 MIPS Architecture

Instructions for Control Flow

- Default instruction execution is sequential
- Control flow instructions allow instruction execution to go out of sequence
 - Conditional branches
 - Jumps (unconditional branches)
 - Procedure calls and returns

CONTROL FLOW - HIGH LEVEL

conditional BRANCH :

Condition



if ($x > y$) { do if $x > y$

} else {

else do + his

}

Condition

for ($i = 0$; $i < N$; $i++$) {

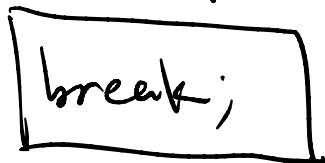
Do while i in range $(0, N-1)$

}

JUMPS (UNCONDITIONAL BRANCHES)

HIGH-LEVEL?

example



A hand-drawn rectangular box with a thin black border. Inside the box, the word "break;" is written in a simple, lowercase sans-serif font.

PROCEDURE CALLS AND RETURNS

(Function)

```
int sum, x, y;
```

```
:
```

```
sum = add(x, y);
```

call ↙

```
int add(int a, int b) {
```

```
    return (a+b);
```

RETURN

Control Flow Addressing

Ques)

- Destination address of next instruction?
 - PC-relative: displacement added to the program counter value
 - Dynamic address: register (or other location) contains destination address

Conditional Branch Options

- Branch based on condition code (CC)
 - Special test bits (zero/nonzero, positive/negative)
- Branch based on value in condition register
- Compare and branch
 - CJNE – compare and jump if not equal

```
        LOAD R0, 1      ; initialize loop counter
LOOP:   (loop code)
        ADD R0,1       ;increment loop counter
        CJNE R0, 10, LOOP ;repeat if loop counter < 10
        (next code)
```

CONDITION CODES?

if ($x = y$) \Leftrightarrow if " $x - y = 0$ "

if ZERO TEST BIT SET

so a lot of times

$x = y$.

for the condition because

they say well if I subtract

that at zero it means a equal.

So that's really what's happening on lots of processors.

Procedure Invocation Options

过程调用选项

- Procedures: functions, subroutines
- Procedure Call
 - Save desired values from calling program
 - Save current PC value (so procedure knows where to return)
- Procedure Return
 - Use saved PC value to return to calling program
 - Restore saved values

Outline

- A.1 Introduction
- A.2 Classifying Instruction Set Architectures
- A.3 Memory Addressing
- A.4 Type and Size of Operands
- A.5 Operations in the Instruction Set
- A.6 Instructions for Control Flow
- A.7 Encoding an Instruction Set
- A.9 MIPS Architecture

Encoding an Instruction Set

- How many bits for the instruction (16, 32)?
- How to allocate bits for instruction fields?
 - Op code – operation (load, add, branch)
 - Operand(s) – register/memory addresses
 - Constants (for immediate addressing)
 - Displacement (for indexing, branching)
- Fields may differ for different instructions

Outline

- A.1 Introduction
- A.2 Classifying Instruction Set Architectures
- A.3 Memory Addressing
- A.4 Type and Size of Operands
- A.5 Operations in the Instruction Set
- A.6 Instructions for Control Flow
- A.7 Encoding an Instruction Set
- A.9 MIPS Architecture

Outline

- A.9 MIPS Architecture
 - MIPS Instruction Set
 - MIPS Processor Design

The MIPS Instruction Set

- NOTE: Authors describe “RISC-V” architecture
 - We will use a specific version of this architecture: MIPS
 - Why use MIPS?
 - Widely used version
 - Used in prerequisite course CS 4381/ECE 3382
 - Many resources available
 - E.g. tools that we will use in this course
 - MIPS Reference Data sheet posted on Canvas
- Focus on this make easier*
- the MIPS is essentially a RISC-V architecture the author got a little more General.*

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination
- add a, b, c # a gets b + c
- All arithmetic operations have this form

Arithmetic Example

- C code:

$f = (g + h) - (i + j);$ *high level language*

- Compiled MIPS code:

*MIPS can only handle two of
these at a time.*

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

we're going to put these in temporary storage areas.

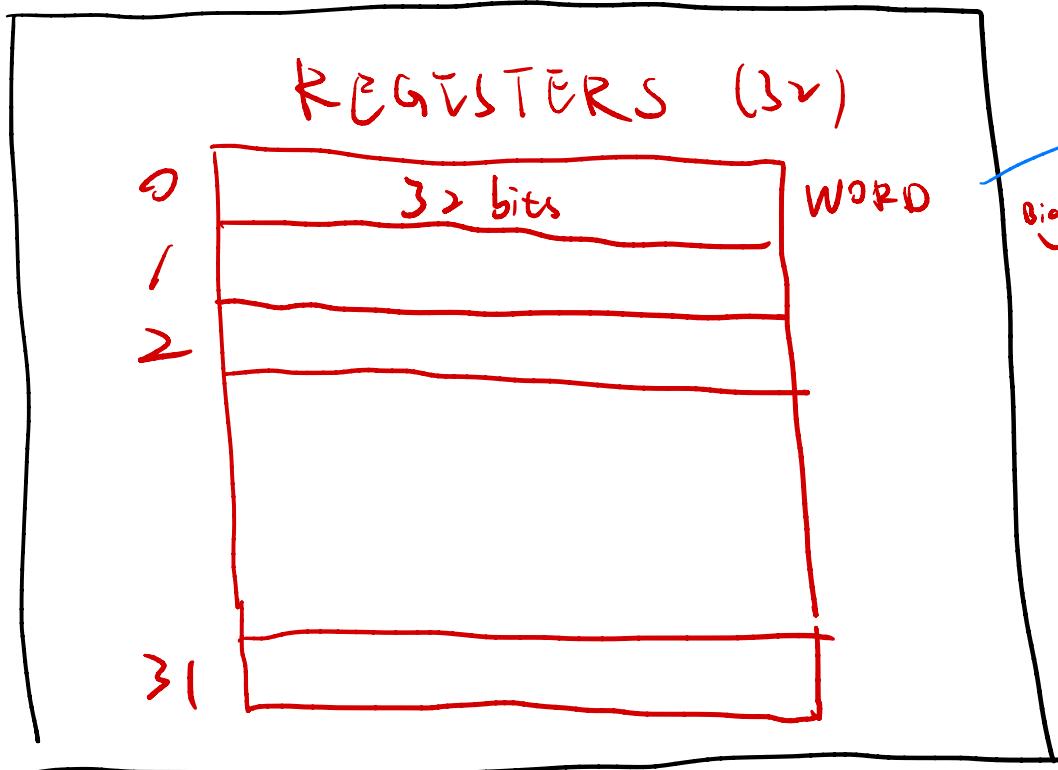
Register Operands

- Arithmetic instructions use register operands → 寄存器
- MIPS has a $32 \times 32\text{-bit}$ register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables

MIPS REGISTERS (REGISTER FILE)

INSIDE PROCESSOR

MIPS PROCESSOR



the mips word size is 32 bits
so each register has to hold
one word which is 32 bits
Big ENDIAN

Register Operand Example

- C code:

$f = (g + h) - (i + j);$

– f, ..., j in \$s0, ..., \$s4

- Compiled MIPS code:

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

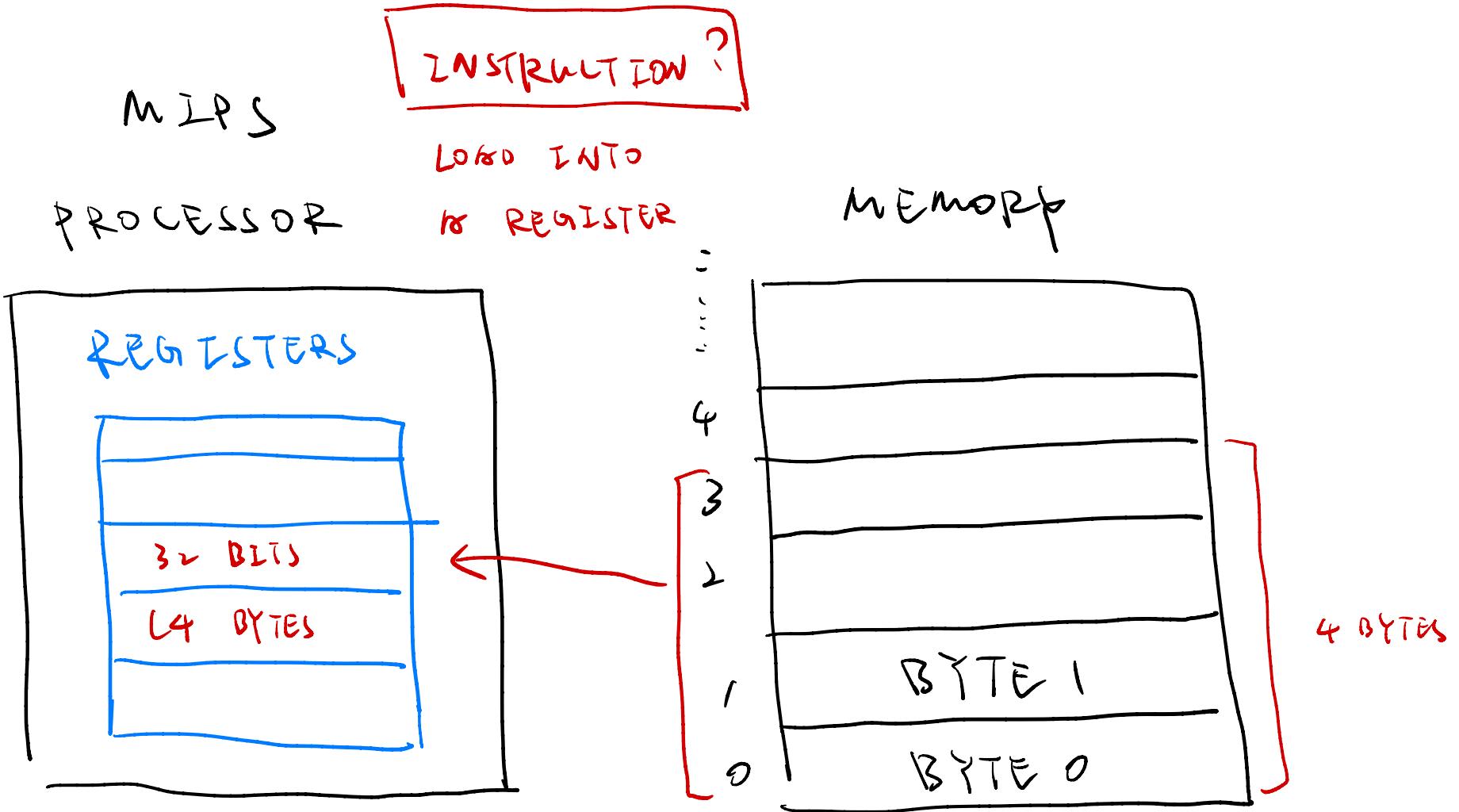
```
add $t1, $s3, $s4
```

```
sub $s0, $t0, $t1
```

Memory Operands

主記憶 . 漢字

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is Big Endian
 - Most-significant byte at least address of a word
 - c.f. Little Endian: least-significant byte at least address



STORE IN BIG-ENDIAN
 ORDER IN REGISTER

INFORMATION NEEDED ?

- LOCATION OF FIRST BYTE IN MEMORY
- LOCATION OF REGISTER IN PROCESSOR

WHERE WE WANT TO STORE DATA

Memory Operand Example 1

- C code:

g = h + A[8];

– g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

– Index 8 requires offset of 32

- 4 bytes per word

```
lw $t0, 32($s3)      # load word  
add $s1, $s2, $t0
```

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
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction

- Just use a negative constant

```
addi $s2, $s1, -1
```

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
`add $t2, $s1, $zero`

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$
- Example
 - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
 - 0 to $+4,294,967,295$

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
 - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
 - $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \overline{x} = 1111\dots111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000\ 0000\dots0010_2$
 - $-2 = 1111\ 1111\dots1101_2 + 1$
 $= 1111\ 1111\dots1110_2$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- In MIPS instruction set
 - addi: extend immediate value
 - lb, lh: extend loaded byte/halfword
 - beq, bne: extend the displacement
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110