



# Chapter2 – Part I

---

## Instructions: Language of the Computer

臺大電機系 吳安宇教授

2025/02/24 v1



# outline

---

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program



# Introduction

---

- To command a computer's hardware, you must speak its language.
  - The words of a machine's language are called "Instructions".
  - The vocabulary is called an "Instruction set".
  - Machine's languages are simpler and easier, once you learn one, you can pick up others easily.
  - Reason:
    - (1) Hardware and operating principle are similar.
    - (2) Basic operations (**add, mul, move**) are similar.
  - Purpose of this chapter: to learn "assembly language".

# Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - **Textual** representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded **instructions** and **data**

High-level  
language  
program  
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

Assembly  
language  
program  
(for MIPS)

```
swap:
    muli $2, $5, 4
    add  $2, $4, $2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler

Binary machine  
language  
program  
(for MIPS)

```
000000001010000100000000000011000
000000000000110000001100000100001
100011000110001000000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

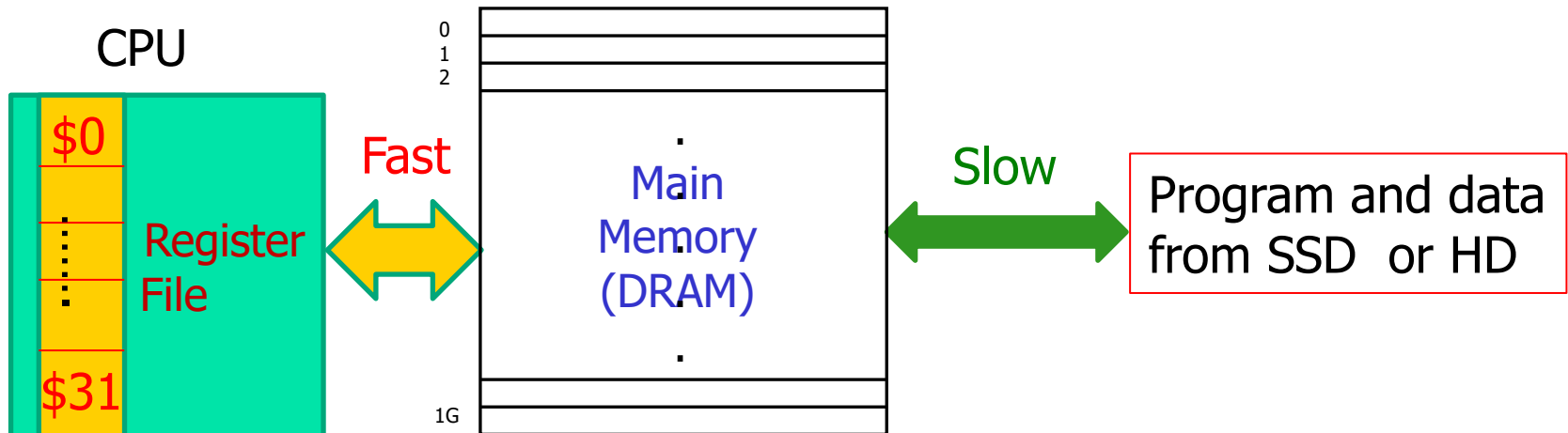
# MIPS RISC CPU (1/2): Storage

## MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
$2^{30}$ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Addressing space

Main memory (for data and program)  
(DRAM) –  $10^9$  (1G) words (word=32 bits=4 bytes)



# MIPS CPU (2/2): Inst. Set

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2   \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2   \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2   20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if ( $\$s1 == \$s2$ ) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ( $\$s1 \neq \$s2$ ) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if ( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if ( $\$s2 < 20$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if ( $\$s2 < 20$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$ ; go to 10000	For procedure call



# Outline

---

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program



# Operations of the computer hardware

- All instructions have **3 operands**
- Operand order is fixed (destination first)
  - Example:

In C language: **a = b + c**

In MIPS: **add** a, b, c

(we'll talk about registers later)

- The natural number of operands for an operation like addition is **three**...requiring every instruction to have **exactly three operands**, no more and no less, conforms to the philosophy of ***keeping the hardware as simple as possible!***





# Operations of the computer hardware

- Example1: add 4 numbers (b, c, d, e):  $a = b + c + d + e$

**add** a, b, c #  $a = b + c$

**add** a, a, d #  $a = a + d$  ( $a = b + c + d$ )

**add** a, a, e #  $a = a + e$  ( $a = b + c + d + e$ )

- It takes three instructions
- # stands for comment ( for human reader )
- Imply " simplicity favors regularity " (Principle 1)  
(Use three-operand instruction can handle all kinds of addition)
- Some CISC computers can "add 3 numbers" in one instruction.



# Operations of the computer hardware

## ■ Syntax

Category	instruction	example	meaning	comments
Arithmetic	add	add a,b,c	$a = b + c$	Always 3 <b>operands</b>
	subtract	sub a,b,c	$a = b - c$	

- Writing assembly programs (Example1):
  - In C program:  $a = b + c;$   
 $d = a - e;$   
*There are 5 variables (a,b,c,d,e).*
  - In assembly language: (produced by C compiler)  
 $\text{add } a, b, c$   
 $\text{sub } d, a, e$



# Example 2

- In C program:

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

- In assembly language:

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

-- Note that compiler "**created**" **t0** and **t1** to express the program in the restricted three-operands-per-instruction notation.



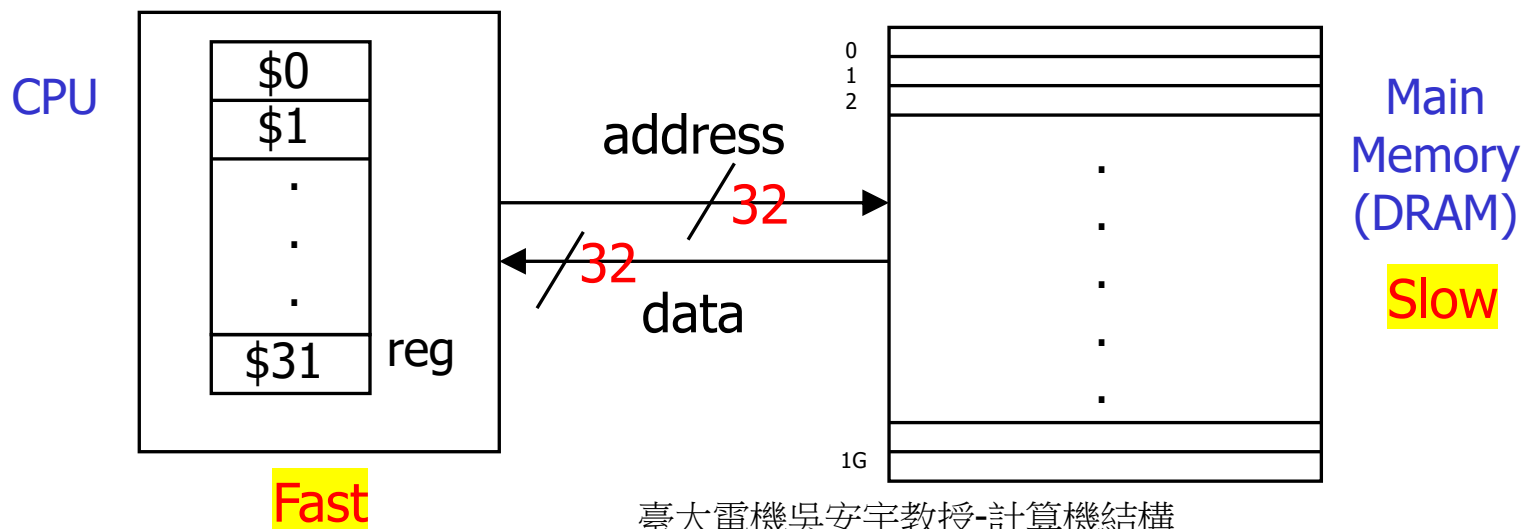
# Outline

---

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program

# Registers (inside CPU)

- Main (and the only) Storage within CPU.
- The size of a register in the MIPS architecture is **32-bit**.
- “**word**” = **32 bits** in MIPS.
- MIPS has 32 registers, using the notation **\$0**, **\$1**, ....., **\$31** to represent them.
- CPU (RISC) ONLY performs **arithmetic on registers**



# MIPS Register Convention

0	zero constant 0	16	s0 callee saves ... (caller can clobber)
1	at reserved for assembler	23	s7
2	v0 expression evaluation &	24	t8 temporary (cont'd)
3	v1 function results	25	t9
4	a0 arguments	26	k0 reserved for OS kernel
5	a1	27	k1
6	a2	28	gp Pointer to global area
7	a3	29	sp Stack pointer
8	t0 temporary: caller saves ... (callee can clobber)	30	fp frame pointer
15	t7	31	ra Return Address (HW)

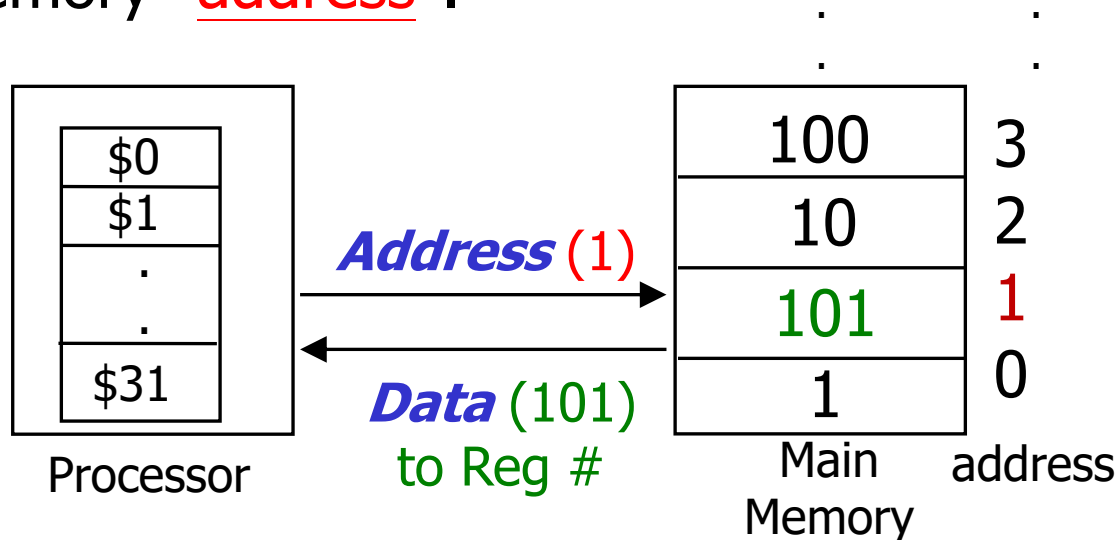


# Compiler → Assembly Language

- For now on, we use ***\$s0, \$s1***, .....for registers that correspond to **variables** in C and Java programs
- Use ***\$t0, \$t1***, .....for **temporary registers** needed to compile the program into MIPS instructions.
- Example:  $f = (g + h) - (i + j)$ 
  - Assignment: **(by compiler)**  
 $f = \$s0, g = \$s1, h = \$s2, i = \$s3, j = \$s4$
  - **Compiler** → assembly language  
 $\text{add } \$t0, \$s1, \$s2 \quad \# t0 = g+h$   
 $\text{add } \$t1, \$s3, \$s4 \quad \# t1 = i+j$   
 $\text{sub } \$s0, \$t0, \$t1 \quad \# f = (g+h)-(i+j)$

# Data transfer instructions

- CPU can store only a small amount of data in registers (\$0, \$1, ....., \$31).
- Arithmetic operations occur “ONLY” on registers in MIPS.
- MIPS needs instructions that transfer data between memory and registers.
- To access a word in memory, the instruction must supply the memory “address”.





# Variables and Array in Memory

In C language

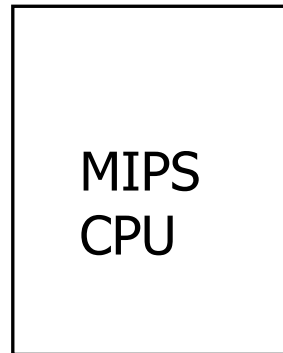
*Data:*

```
int a, b;  
int A[10], B[10];
```

*Program:*

```
a = -3  
b = 10  
A[0] = -8  
A[1] = 7
```

***a = 100***  
***b = 101***  
***A = 102 (base)***  
***B = 112 (base)***



Processor

Main  
Memory

Address

.	.	
-3	100	<b>a</b>
10	101	<b>b</b>
-8	102	<b>A[0]</b>
7	103	<b>A[1]</b>
A[2]	104	
A[3]	105	
.		
.		
A[9]	111	
B[0]	? 112	
B[1]	? 113	

Starting  
address  
of data



➡ **a, b, A[0], B[0] = address or data?**



# LW: Load Word from Memory

- Data transfer instruction that moves data from memory to a register.
- MIPS instruction: **lw** = load word
- Example:

Assume that A is an array of 100 words (**int A[100] in C**),  $g = \$s1$ ,  $h = \$s2$ , and the starting address (**base address**) of the array =  **$\$s3$** .

$g = h + A[8];$

The address of the array element is the **sum** of the **base of the Array A (stored in  $\$s3$ )** plus the number to select element 8.

**lw**     $\$t0, 8(\$s3)$                       #  $\$t0 \leftarrow A[8] = \text{MEM}[A+8]$

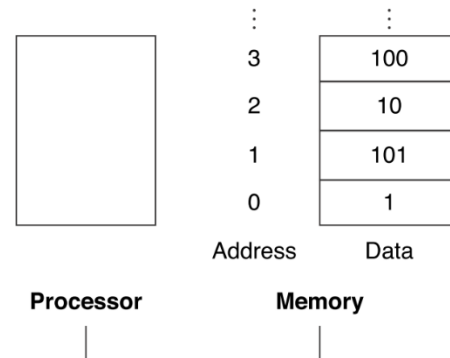
**add**    $\$s1, \$s2, \$t0$                     #  $g = h + A[8]$

- The constant in a data transfer instruction is called the offset.
- The register added to from the address is called the base register.

# Address format in MIPS

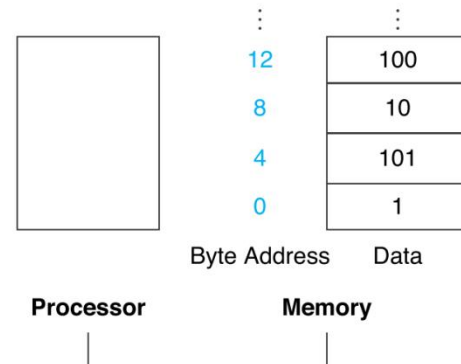
- 1 word (32bits) = **4 bytes**

- Memory unit is usually **"Byte (8 bits)"**



Byte = 8 bits

**FIGURE 2.2 Memory addresses and contents of memory at those locations.** If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word



Word =  
4 Bytes

**FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words.** The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

# SW: Store Word to Memory

- Data transfer instruction that moves data from a register to memory.
- MIPS instruction: **sw = store word**
- Example:

Assume that  $h = \$s2$ , and the Base address of **Array A** =  $\$s3$

$A[12] = h + A[8]; \rightarrow$  C Program

Compiler  $\rightarrow$  Assembly language

lw	\$t0, 32(\$s3)	# temp \$t0 = A[8]
add	\$t0, \$s2, \$t0	# temp \$t0 = h + A[8]
sw	\$t0, 48(\$s3)	# A[12] = h + A[8]





# Spilling registers

---

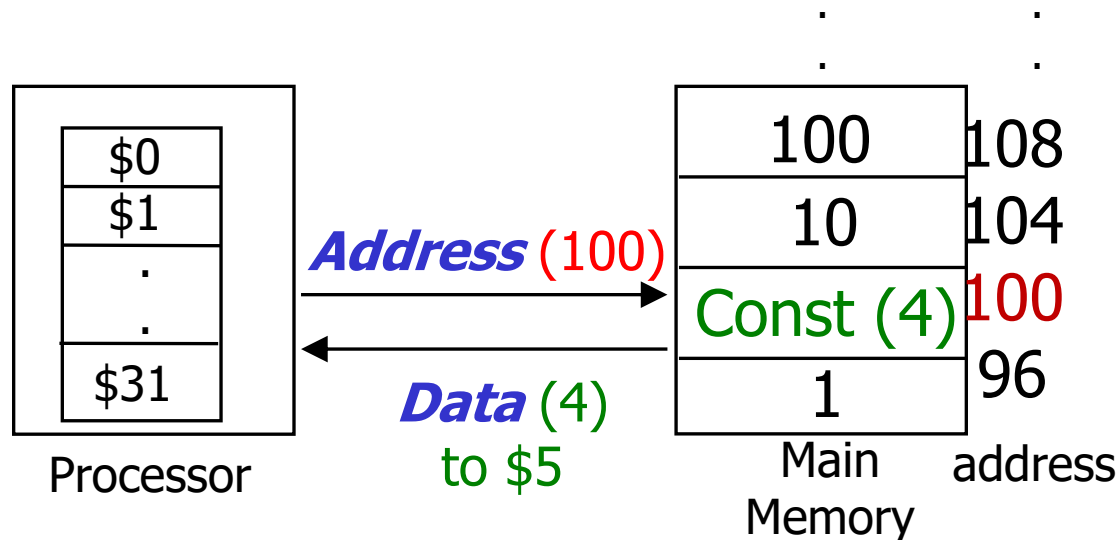
- Since registers are limited inside CPU (32 registers in MIPS CPU), MIPS compiler tries to keep **most frequently** used variables in registers and places the rest in **memory**
  - The process is called “**spilling registers**”.
- It's **compiler's** important job.
- Side Notes: 16 bits is called “**halfword**” in MIPS.

# Operands with Constant

- Assume that **AddrConstant4** is the memory address of the constant 4.
- $\text{MEM}[\text{AddrConstant4} + \$1] \leftarrow \text{Constant } 4$

**lw** \$t0 , AddrConstant4 (\$1)      # \$t0 = constant 4

**add** \$s3, \$s3, \$t0                      # \$s3 = \$s3 + \$t0 (=4)





# Operands with Constant

- **Constant** as an **Operand** (Immediate Addressing mode)
  - **addi** = add immediate # add an immediate constant

**addi** **\$s3**, **\$s3**, **4** # **\$s3** = **\$s3** + **4** (constant)

↓ ↑ equal

└─ **lw** **\$t0** , **AddrConstant4 (\$1)** # **\$t0** = constant 4

**add** **\$s3**, **\$s3**, **\$t0** # **\$s3** = **\$s3** + **\$t0** (**\$t0**=4)

Assume that **AddrConstant4** is the memory address of the constant 4.



# Outline

---

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers (skipped)
- 2.5 Representing Instructions in the Computer
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program



# Positive numbers (addr & data)

- $1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$   
 $= 8 + 2 + 1 = 11_{10}$
- In MIPS word ( = 32 bits)

0000	0000	.....	0000	1011
------	------	-------	------	------

- Bit\_31 = Leftmost bit = ***Most significant bit (MSB)***
- Bit\_0 = Rightmost bit = ***Least significant bit (LSB)***

We can represent positive integers (without negative number):

$$0 = 0000 \dots 0000_2$$

$$1 = 0000 \dots 0001_2$$

$$2 = 0000 \dots 0010_2$$

⋮  
⋮

$$2^{32}-1 = 4,294,967,295_{10} = 1111 \dots 1111_2$$



# 2's Complement Representation (data only)

MSB=sign bit

- 0: Positive number or zero
- 1: Negative number

## ■ Formula

Value in Base-10

$$= -2^{31} \times \text{bit31} + 2^{30} \times \text{bit30} + 2^{29} \times \text{bit29} + \dots + 2^0 \times \text{bit0}$$

## ■ Example

- $1111\ 1111\ \dots\ 1111\ 1100_2$   
 $= -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 = -4_{10}$

- Flip and plus one (negation)  
 $0000\ 0000\ \dots\ 0000\ 0011 + 1 = 4_{10}$

# Shortcut of Negation

- Negate  $2_{10}$ , and then check the result by negating  $-2_{10}$
- $2_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\ + \phantom{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ } 1_2 \\ \hline = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 \\ = -2_{10} \end{array}$$

Flip &  
Add 1

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \\ + \phantom{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ } 1_2 \\ \hline = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 \\ = 2_{10} \end{array}$$

Flip &  
Add 1



# Sign Extension

- Convert **16-bit** binary version of  $2_{10}$  and  $-2_{10}$  to **32-bit** binary numbers.
- $0000\ 0000\ 0000\ 0010_2 = 2_{10}$

Make 16 copies of the value in MSB. Then, make 2's complement

→ **0000 0000 0000 0000** **0000 0000 0000 0010**<sub>2</sub> =  $2_{10}$

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_2 \\ + \qquad \qquad \qquad 1_2 \\ \hline = \mathbf{1111\ 1111\ 1111\ 1110}_2 \\ = -2_{10} \end{array}$$

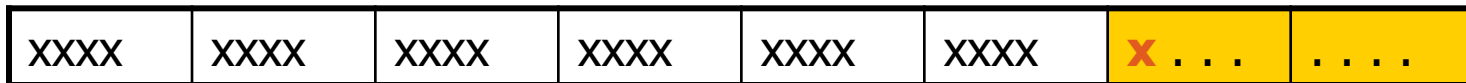
Make **16 copies** of the value in MSB (the sign bit)

→ **1111 1111 1111 1111** **1111 1111 1111 1110** =  $-2_{10}$

# Sign Extension

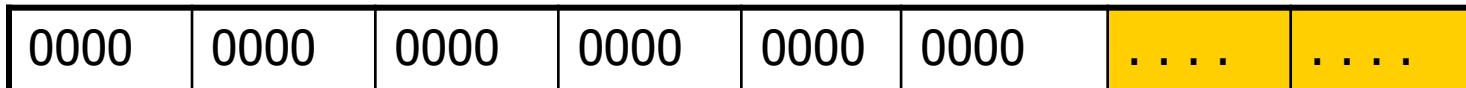
- **lb** (load byte from memory to 32-bit register)

Need Sign extension: fill with x's (x=0 or 1)



- **lbu** (load byte unsigned from memory to register)

fill with 0's



- **lh** (load half word)



- **lhu** (load half word unsigned)



# Summary

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$s1 = s2 + s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$s1 = s2 - s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$s1 = s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$s1 = \text{Memory}[s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[s2 + 20] = s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$s1 = \text{Memory}[s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$s1 = \text{Memory}[s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[s2 + 20] = s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$s1 = \text{Memory}[s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$s1 = \text{Memory}[s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[s2 + 20] = s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$s1 = \text{Memory}[s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[s2+20]=s1; s1=0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$s1 = s2 \& s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$s1 = s2   s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$s1 = \sim (s2   s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$s1 = s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$s1 = s2   20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$s1 = s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$s1 = s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if ( $s1 == s2$ ) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ( $s1 \neq s2$ ) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ( $s2 < s3$ ) $s1 = 1$ ; else $s1 = 0$	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if ( $s2 < s3$ ) $s1 = 1$ ; else $s1 = 0$	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if ( $s2 < 20$ ) $s1 = 1$ ; else $s1 = 0$	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if ( $s2 < 20$ ) $s1 = 1$ ; else $s1 = 0$	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4$ ; go to 10000	For procedure call



# Outline

---

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer (重要)
- 2.6 Logical Operations
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program





# Review of Number Systems (I)

- The hexadecimal-to-binary conversion table

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0 <sub>hex</sub>	0000 <sub>two</sub>	4 <sub>hex</sub>	0100 <sub>two</sub>	8 <sub>hex</sub>	1000 <sub>two</sub>	c <sub>hex</sub>	1100 <sub>two</sub>
1 <sub>hex</sub>	0001 <sub>two</sub>	5 <sub>hex</sub>	0101 <sub>two</sub>	9 <sub>hex</sub>	1001 <sub>two</sub>	d <sub>hex</sub>	1101 <sub>two</sub>
2 <sub>hex</sub>	0010 <sub>two</sub>	6 <sub>hex</sub>	0110 <sub>two</sub>	a <sub>hex</sub>	1010 <sub>two</sub>	e <sub>hex</sub>	1110 <sub>two</sub>
3 <sub>hex</sub>	0011 <sub>two</sub>	7 <sub>hex</sub>	0111 <sub>two</sub>	b <sub>hex</sub>	1011 <sub>two</sub>	f <sub>hex</sub>	1111 <sub>two</sub>

**FIGURE 2.4 The hexadecimal-binary conversion table.** Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

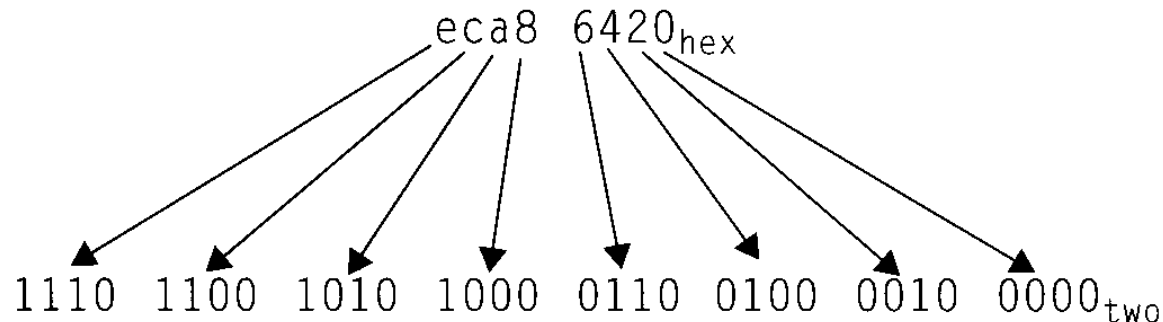
# Review of Number System (II)

- Example1:

$$\begin{array}{ccc} 123_{10} & = & 1111011_2 \\ \text{(base 10)} & & \text{(base 2)} \end{array}$$

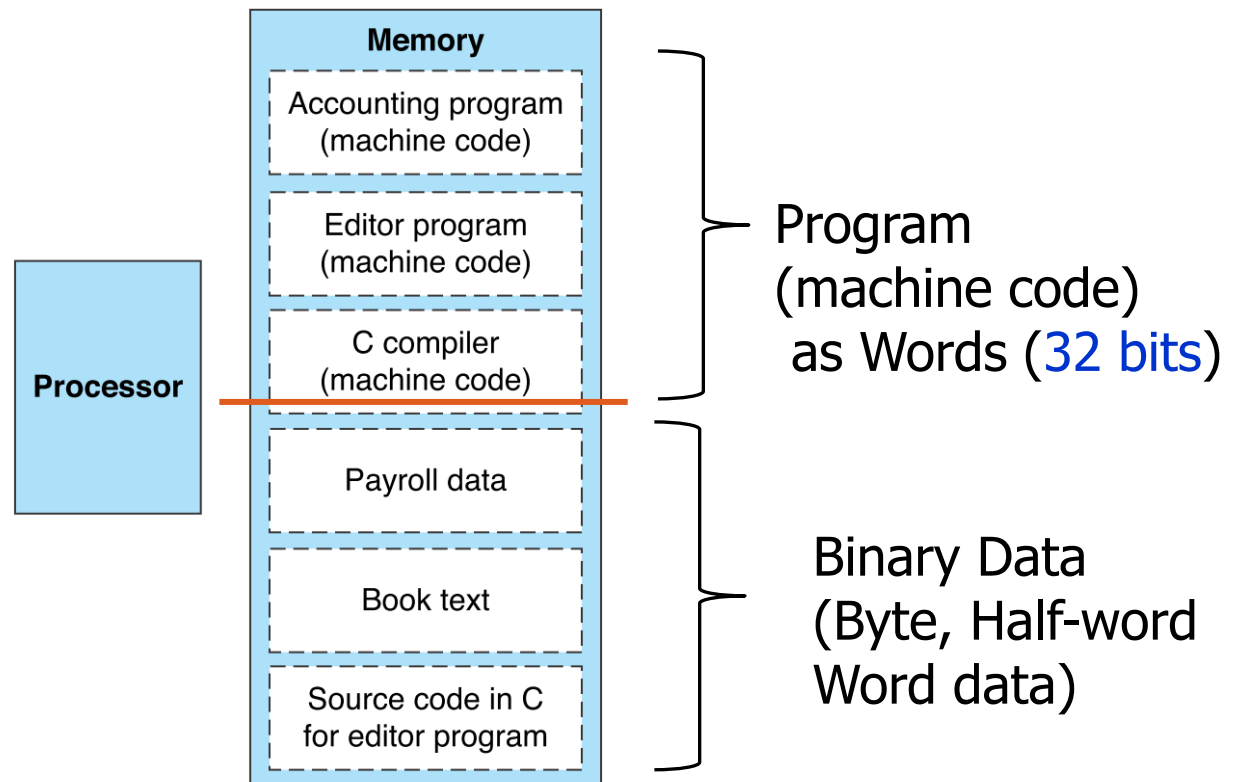
- Example2:

$$\begin{array}{ccc} \text{eca8 } 6420_{16} & = & 1110 \ 1100 \ 1010 \ 1000 \ 0110 \ 0100 \ 0010 \ 0000_2 \\ \text{(base 16)} & & \text{(base 2)} \end{array}$$



# "Stored-program" Concept

- Instructions are represented as (32-bit) numbers!!
- Programs can be stored in memory to be read or written just like numbers.
- Most important concept in computer history (*von Neumann architecture*)



# Review of MIPS register conventions

0	zero constant 0	16	s0 callee saves ... (caller can clobber)
1	at reserved for assembler	23	s7
2	v0 expression evaluation &	24	t8 temporary (cont'd)
3	v1 function results	25	t9
4	a0 arguments	26	k0 reserved for OS kernel
5	a1	27	k1
6	a2	28	gp Pointer to global area
7	a3	29	sp Stack pointer
8	t0 temporary: caller saves ... (callee can clobber)	30	fp frame pointer
15	t7	31	ra Return Address (HW)

# Representing Instructions in Binary Number

- MIPS instruction Example: `add $t0, $s1, $s2`

0	17	18	8	0	32
---	----	----	---	---	----

- The first and last fields (containing 0 and 32 in this case) in combination → **addition operation**
- The 2nd field (17 = \$s1) → the number of the register that is the first source operand of the addition operation.
- The 3rd field (18 = \$s2) → the number of the register that is the second source operand of the addition operation.
- The 4th field (8 = \$t0) → the number of the register that is to receive the sum.
- Binary numbers (for machine to use) → **Machine language (32 bits)**

000000	10001	10010	01000	00000	100000
<b>6 bits</b>	<b>5 bits</b>	<b>5 bits</b>	<b>5 bits</b>	<b>5 bits</b>	<b>6 bits</b>



# Instruction format in MIPS (R-type)

- R- type (register related)



- meaning:

- **op** : operation of the instruction
- **rs** : the first register **s**ource operand
- **rt** : the second register **s**ource operand
- **rd** : the register **d**estination operand
- **Shamt** : shift amount
- **funct** : function; this field selects the variant of the operation in the op field (e.g., **add**, **sub**)



# Review of Instruction Format

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

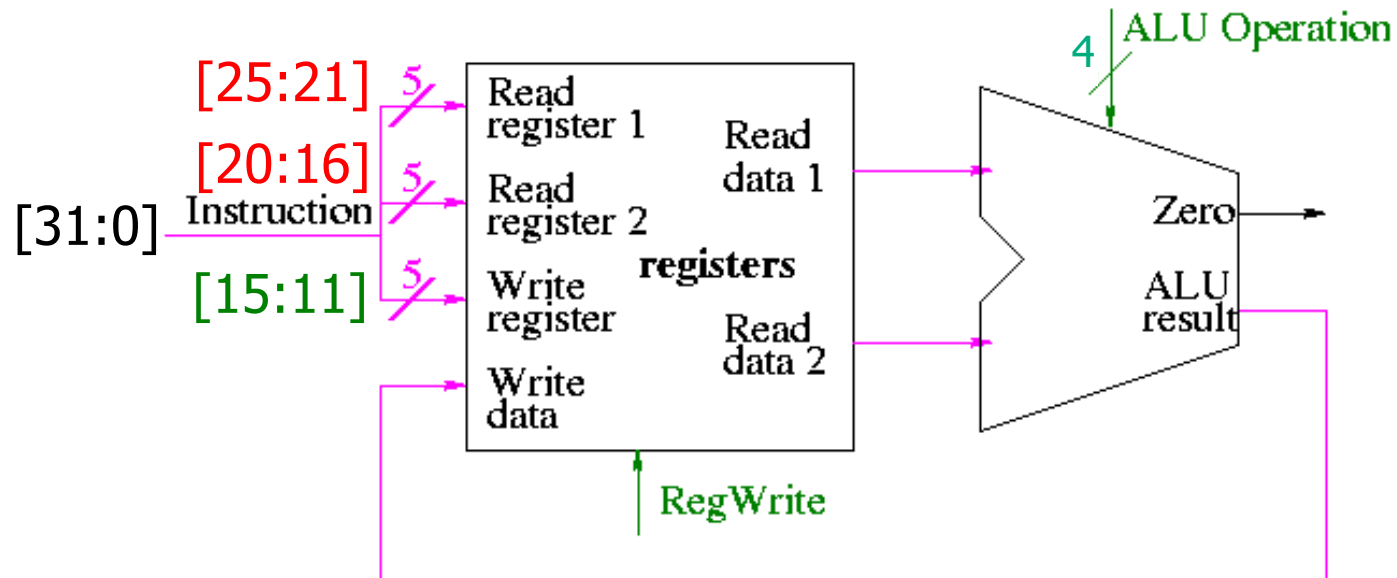
Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

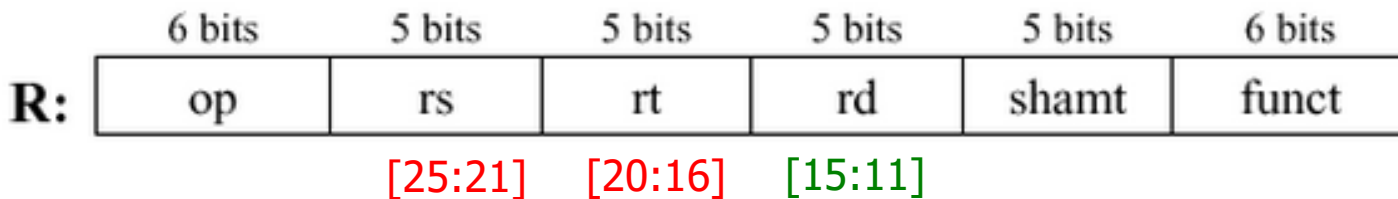
---

**FIGURE 4.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.**

# Datapath for R-type instructions



**add \$rd, \$rs, \$rt**





# Instruction format in MIPS (I-type)

- Example: `lw $t0, 32($s3)`

op	rs	rt	constant / address
35	19	8	32

- 19 (for \$s3) is placed in the **rs** field.
- 8 (for \$t0) is placed in the **rt** field.
- 32 (offset) is placed in the constant/address field.
- Note: In a load word instruction, **rt** field specifies the destination register, which receives the result of the load.

# Instruction format in MIPS (I-type)

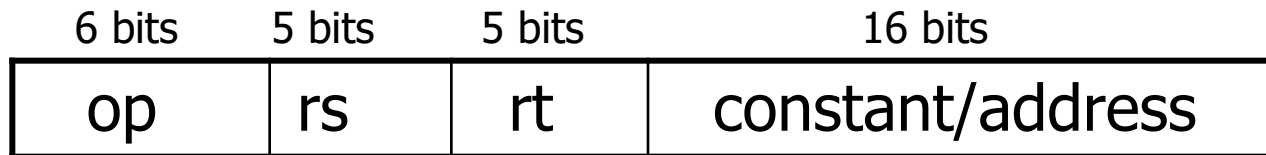
- Example: `addi $t0, $s3, 100`

op	rs	rt	constant / address
8	19	8	100

- 19 (for \$s3) is placed in the **rs** field.
- 8 (for \$t0) is placed in the **rt** field.
- 100 (constant) is placed in the constant/address field.
- Note: In a load word instruction, **rt** field specifies the destination register, which receives the result of the load.

# Limitation in I-type Inst. Format

- I- type (data transfer, or immediate mode)



- Meaning:

op : operation of the instruction

rs : the source register (usually the base register)

rt : the register to receive the result of the operation

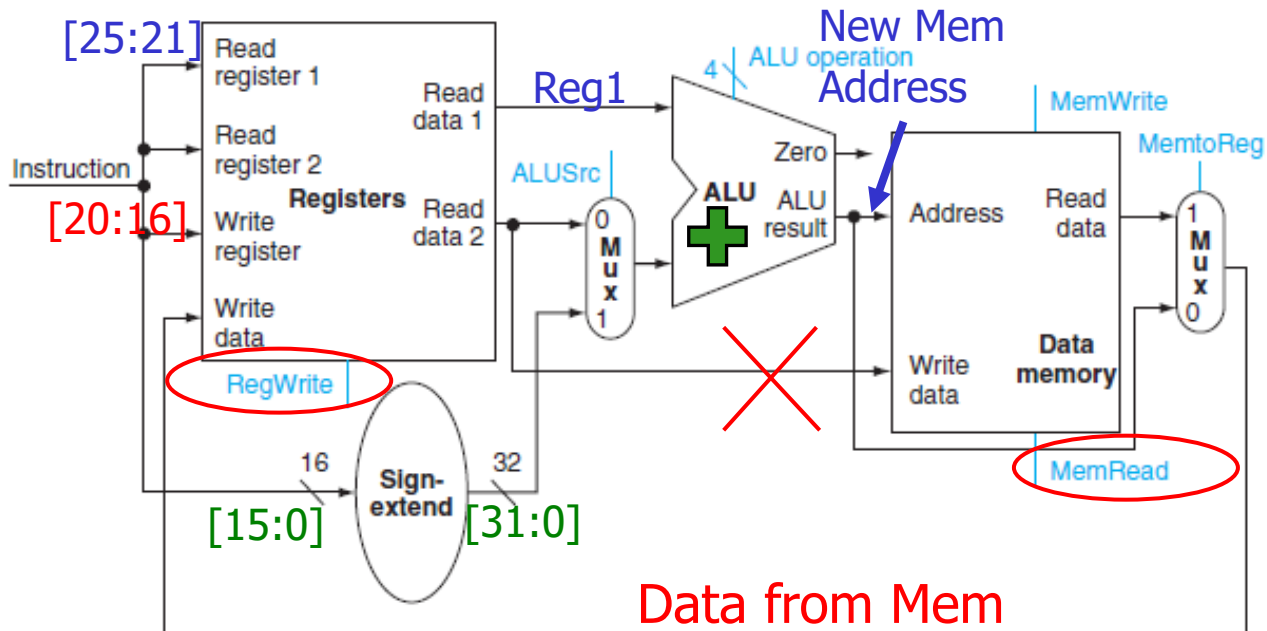
address : the offset from the base register

- Constant/address (16-bit):**

- The 16-bit address means a Load Word instruction can load any word within a region of  $\pm 2^{15}$  or 32768 **bytes** ( $\pm 2^{13}$  or 8192 **words**) of the address in the base register **rs**

- Similarly, the immediate is **limited to constants  $\leq \pm 2^{15}$**

# Datapath for **lw** instructions

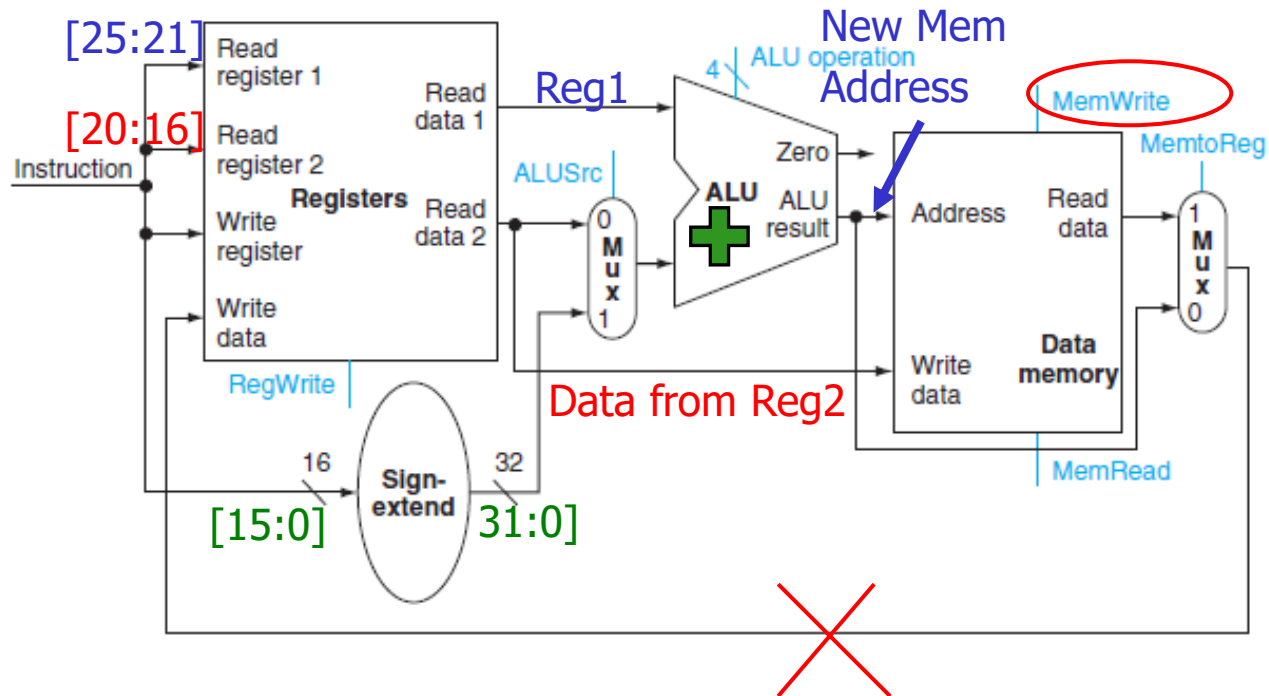


**lw \$rt, 100(\$rs)**

I:

op	rs	rt	address / immediate
	[25:21]	[20:16]	[15:0]

# Datapath for **sw** instructions



**sw** **\$rt**, **100(\$rs)**

I:

op	rs	rt	address / immediate
	[25:21]	[20:16]	[15:0]

# Instruction format in MIPS

- Two major classes of **instruction format** in MIPS

(1) R- type (register related) : add, sub

(2) I - type (data transfer) : lw, sw

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 <sub>ten</sub>	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 <sub>ten</sub>	n.a.
add immediate	I	8 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	address

- Design issue : keep format similar
- Use first field (**op**) to distinguish lower 16 bits
  - R-type* : three fields (rd, shamt, function)
  - I-type* : single field for 16-bit address/immediate data

# Representing Instructions in the Computer

- Example:

Assume  $h = \$s2$  and  $\$t1$  has the base of the array A.

$A[300] = h + A[300];$

- Compiler  $\rightarrow$  assembly language

```
lw  $t0, 1200($t1)    # temp $t0 = A[300]
add $t0, $s2, $t0      # temp $t0 = h + A[300]
sw  $t0, 1200($t1)    # A[300] = h + A[300]
```

- Assembler  $\rightarrow$  machine language (in decimal number)

op	rs	rt	(rd)	(shamt)	address/ (Function)
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		



# Representing Instructions in the Computer

- The base register **9 (\$t1)** is specified in the second field (rs).
- The destination register **8 (\$t0)** is specified in the third field (rt).
- The offset to select A[300] ( $1200 = 300 * 4$ ) is found in the **final field (constant/address)**.
- Final look in MIPS **machine language (32-bit width)**:

op	rs	rt	(rd)	(shamt)	address/ function
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		



# MIPS Machine Language

## MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

# Pseudo Instruction

- Note:

- op code:

- lw (35) : 1 0 0 0 1 1

- sw (43): 1 0 1 0 1 1 → to simplify the HW design

- Reg \$0 is always 0.

- Usage : Assembly language: `move $8, $18 # $8=$18`



**Assembler (system software)**

Machine language: `add $8, $0, $18 # $8=$18+0`

→ “move” is a “pseudo instruction” which simplifies our programming in writing Assembly language, **not implemented** in the HW design.

# Summary of MIPS Inst. Set & Format

## MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
2 <sup>30</sup> memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory

## MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format



# Outline

---

- 2.1 Introduction
- 2.2 Operations of the computer hardware
- 2.3 Operands of the computer hardware
- 2.4 Signed and Unsigned Numbers
- 2.5 Representing Instructions in the Computer (Week3)
- 2.6 Logical Operations (Week4)
- 2.7 Instructions for Making Decisions
- 2.8 Supporting Procedures in Computer Hardware
- 2.9 Communicating with People
- 2.10 MIPS Addressing for 32-bit Immediates and Addresses
- 2.11 Translating and Starting a Program