

**Module #2:**

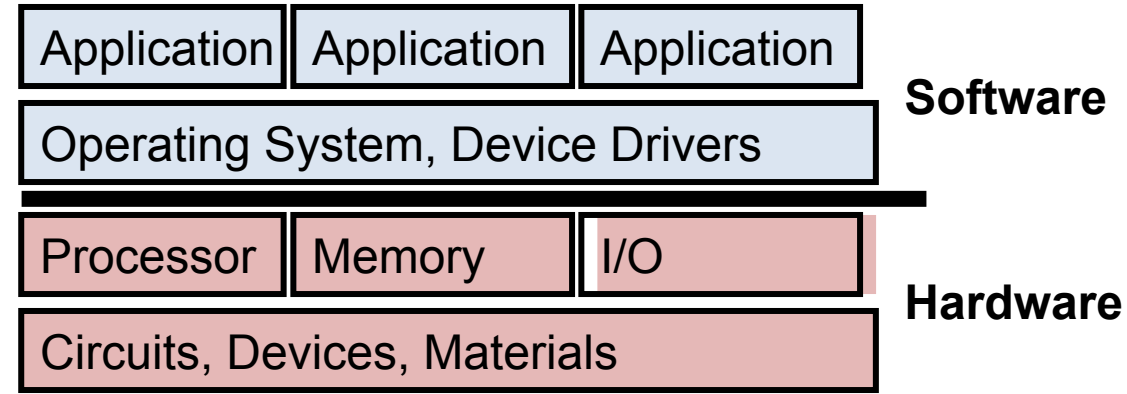
**Instruction Set Architecture Design**

# Objectives

<b>Identify</b>	ISA design approaches and tradeoffs
<b>Understand</b>	Understand machine language execution
<b>Develop</b>	Applications using MIPS assembly
<b>Understand</b>	data types and their representations
<b>Develop</b>	Applications using MIPS assembly

# Instruction Set Architecture

- The **ISA** defines how the CPU is controlled by the software.
- **SW Analogy:** ISA is similar to a library API that is used without understanding the implementation.
- **Common ISAs:** 80x86, ARM, MIPS



## Role of ISA

**Abstraction:** It provides an **abstraction** layer between hardware and software, enabling programmers to write code without needing to understand the intricacies of the underlying hardware



**Compatibility:** The ISA allows software to be compatible with different generations of processors that support the same ISA

# Instruction Set

- **Instruction Set**

- These instructions are the basic commands that tell the hardware what operations to perform, such as arithmetic operations, data movement, and control flow management.
- The instruction set defines the machine language, which is the lowest-level programming language directly understood by the hardware.

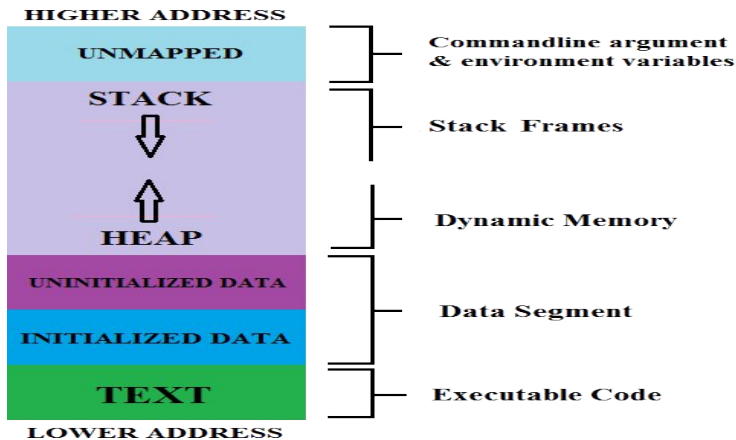
**Assembly Language** provides a more human-readable way to write programs that are closely tied to the instruction set of a particular CPU.

- Each assembly language command corresponds directly to a machine language instruction from the instruction set.
- Assembly language uses mnemonic codes (like `MOV`, `ADD`, `SUB`, etc.) to represent the binary machine instructions.

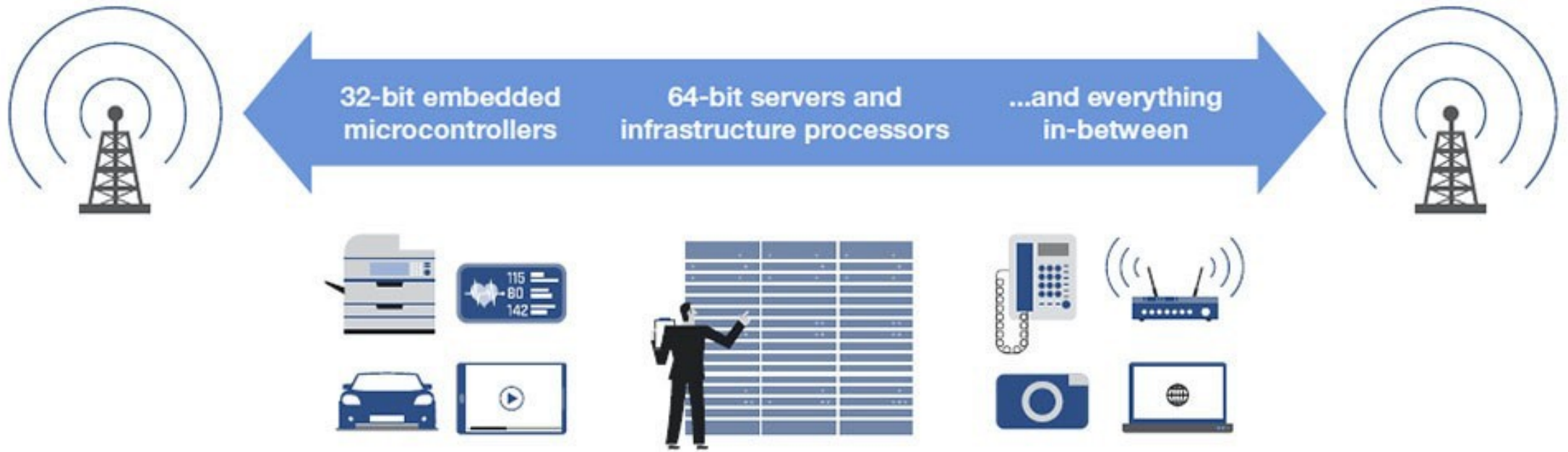
# ISA key components

- **Data Types:** supported by the CPU (**n-bit** integer?, float?, double? ....)
- **Registers:** The ISA specifies the number, size, and types of registers (small, fast CPU data storage locations)
- **Instruction Set** —> all operations that a particular CPU can execute (add, sub, mul, div, and, or, lw, lb, sw, sb, j, jal, beq, ....)
- **Instruction Formats:** the layout of the binary instructions, including fields for the operation code (opcode), operand(s), and other necessary information.

op	rs	rt	rd	shamt	funct
op	rs	rt	address/immediate		
op	target address				



- **Addressing Modes** define how the CPU access operands (register vs memory operands)
- **Memory Architecture:** how memory is organized and accessed, including memory addressing modes, stack management



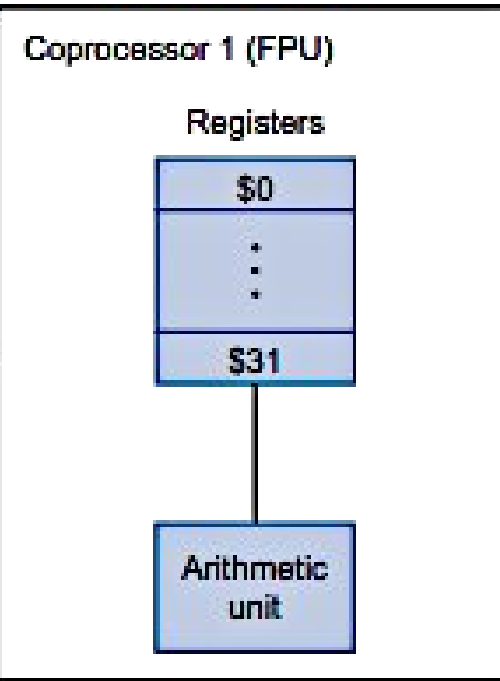
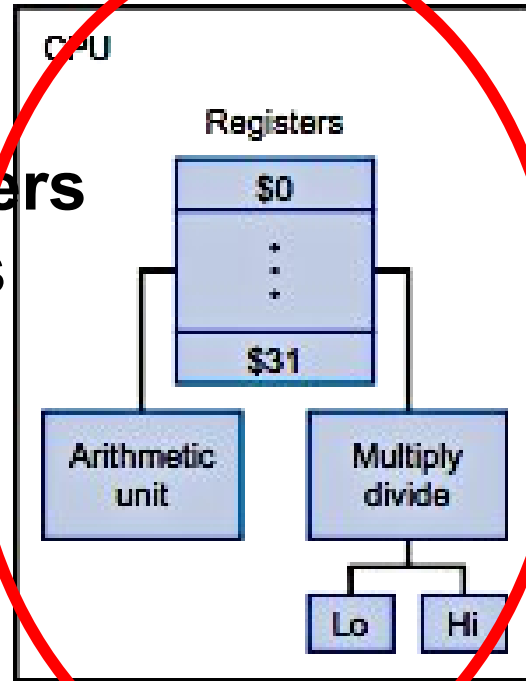
# MIPS Instruction Set Architecture



Sections 2.1-2.3

# MIPS Registers

**Main CPU registers**  
integer operations  
control registers



**Floating point registers**  
FP coprocessor



**Exception coprocessor**

# MIPS Main CPU Registers

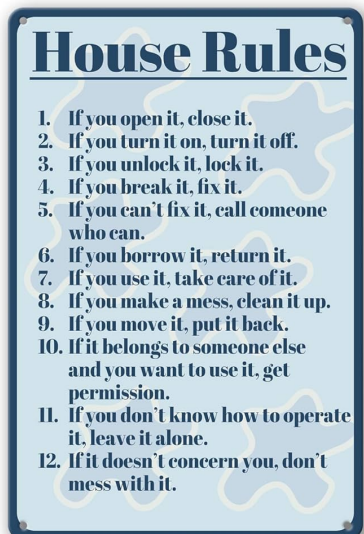
Name	Register number	Usage
\$zero	0	The constant value 0
\$v0-\$v1	2-3	Values for results and expression evaluation
\$a0-\$a3	4-7	Arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Saved
\$t8-\$t9	24-25	More temporaries
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address

Assembler notation  
\$a0 → \$4

32-bit data → “word”  
Default MIPS data unit

**32 × 32-bit**  
register file

**Registers are**  
**kind of your**  
**variables!**





# MIPS register functions

**General purpose registers:** \$t# and \$s# (18 registers) (some rules)

**Global Register (\$gp):** (Do NOT touch unless you are writing an OS)

Points to the location of global variables in memory.

**Stack Pointer (\$sp):** (some rules)

Points to the top of the stack (function calls)

**Return Address (\$ra):** (One rule - leave it alone and it will work fine!)

Holds the address of the instruction to return to after a function call.

**Frame Pointer (\$fp):** (Do NOT touch unless you are writing an OS)

Points to the base of the current function's stack frame.

**Argument Registers (\$a0-\$a3):** (some rules)

Used to pass arguments to functions

**Value Registers (\$v0-\$v1):** (some rules)

Used to return values from functions.



# Instruction Set

- Instruction sets of different processors perform similar functions
  - 1) *Arithmetic & logical operations*  
(Add, sub, mul, div, and, or, ..)
  - 2) *Memory and port transfers*  
move data between the CPU and other computer elements
  - 3) *Flow control* (j, beq, jal, ...)
- Each instruction involves **operands** that could be *processor-specific registers* and/or *memory content*

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   $15, 0($2)
  sw   $16, 4($2)
  jr   $31
```

Assembler

Binary machine  
language  
program  
(for MIPS)

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

# Instruction set design approaches

Basis	RISC	CISC
<b>Full Form</b>	RISC stands for <b>Reduced Instruction Set Computer</b> .	CISC stands for <b>Complex Instruction Set Computer</b>
<b>Type of Instruction</b>	RISC processors have simple instructions taking about one clock cycle.	CSIC processor has complex instructions that take up multiple clocks for execution.
<b>Instruction Set</b>	The instruction set is reduced i.e. it has only a <b>few</b> instructions in the instruction set. Many of these instructions are very <b>primitive</b> .	The instruction set has a variety of different instructions that can be used for complex operations.
<b>Execution Time</b>	In RISC Execution time is relatively small.	In CISC Execution time is very high.
<b>Examples</b>	The most common RISC microprocessors are Alpha, ARC, <b>ARM</b> , AVR, <b>MIPS</b> , PA-RISC, PIC, Power Architecture, and SPARC.	Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, <b>AMD</b> and <b>Intel</b> x86 CPUs.
<b>Average CPI</b>	The average CPI is 1.5 in RISC	The average CPI is in the range of 2 and 15
<b>Focus on</b>	Software Centric Design	Hardware Centric Design

# MIPS

## Instruction Set:

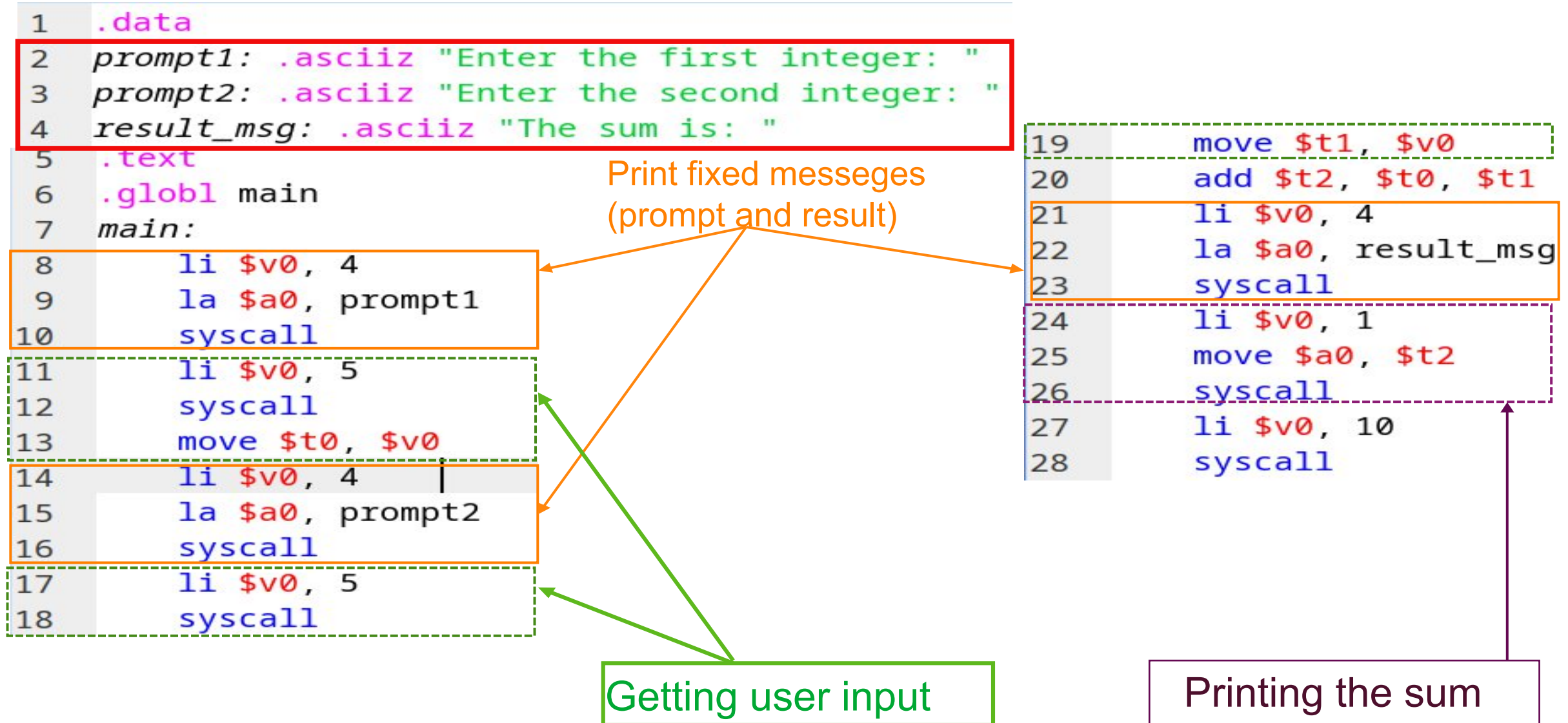
### An Overview

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

**Revisiting our toy code!**



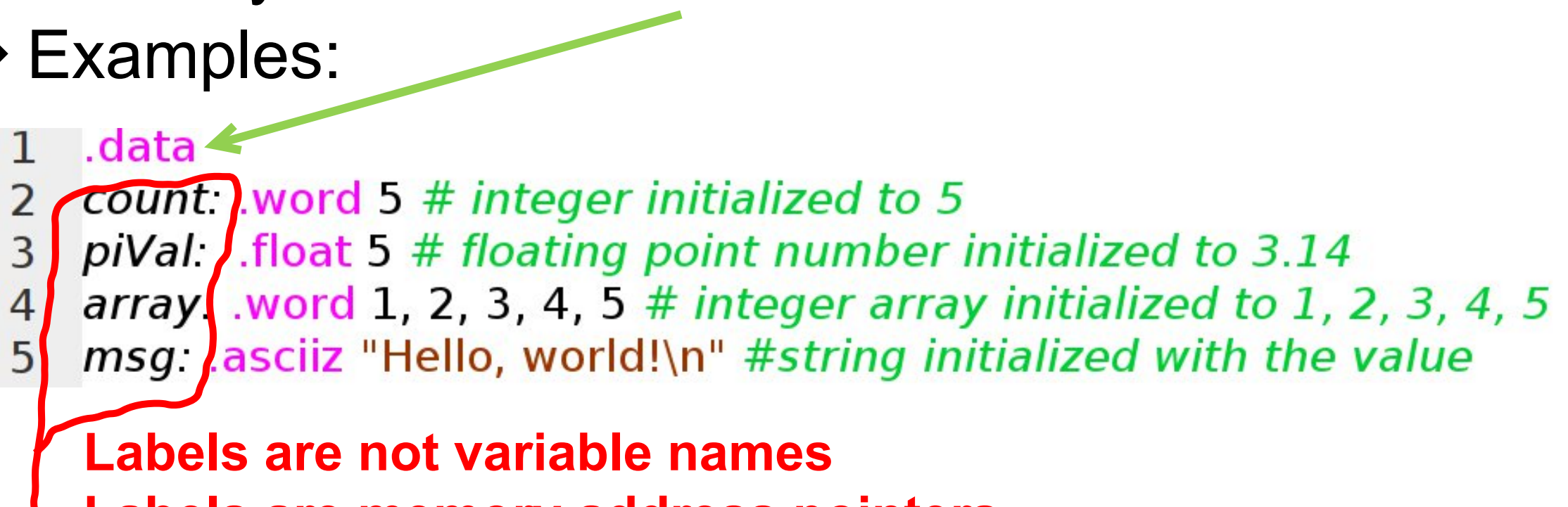
```
num1 = int(input("Enter the first integer: ")) num2 =  
int(input("Enter the second integer: ")) sum = num1 + num2  
print("The sum is:", sum) `
```



# Data Section

- ◆ is used to declare and initialize global variables.
- ◆ starts by **.data** assembler directive in the code
- ◆ Examples:

```
1 .data
2 count: .word 5 # integer initialized to 5
3 piVal: .float 5 # floating point number initialized to 3.14
4 array: .word 1, 2, 3, 4, 5 # integer array initialized to 1, 2, 3, 4, 5
5 msg: .asciiz "Hello, world!\n" #string initialized with the value
```



**Labels are not variable names**

**Labels are memory address pointers**

**Labels are intended to simplify writing and running assembly code.**

# Understanding Data Section

```
1 .data
2 count: .word 5 # integer initialized to 5
3 piVal: .float 5 # floating point number initialized to 3.14
4 array: .word 1, 2, 3, 4, 5 # integer array initialized to 1, 2, 3, 4, 5
5 msg: .asciiz "Hello, world!\n" #string initialized with the value
```

- **Line#1** (.data) instructs the assembler to store the following in the program data segment
- **Line #2** instructs the assembler to reserve 4 bytes and store the binary equivalent of 5 in these bytes
- **Line#3** instructs the assembler to reserve 4 bytes and stores the binary representation of floating number 5.0 in these bytes
- **Line#4** instructs the assembler to reserve 20 bytes and store the binary equivalent of 1,2,3,4,5 in these bytes (each in 4 bytes)
- **Line#5** instructs the assembler to reserve 20 bytes and store the binary representation of the ASCII code of the provide characters.



# Memory Content

Data Segment				
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00000005	0x40a00000	0x00000001	0x00000002

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	5	1084227584	1	2

DEMO

# I/O Syscalls in MIPS

- ◆ Any assembly will rely on OS code (device drivers) to deal with I/O operation. (Similar to *import os*)
- ◆ ***syscall*** instruction is used to execute such OS code.
- ◆ **syscall** needs procedure ID and may need procedure arguments
  1. syscall ID is stored in **\$v0**
  2. syscall arguments should be in **\$a0, \$a1, ..** before executing syscall instruction
  3. syscall outcome is typically stored in **\$v0** register

# Example#1: Syscall to print string

```
1  .data
2  prompt1: .asciiz "Enter first integer: "

12  li $v0, 4           # syscall code for print_string
13  la $a0, prompt1     # load address of prompt1 into $a0
14  syscall
```

- Line#12 —> loads **print string** syscall id **4** in **\$v0**
  - it uses load immediate (**li**) instruction
- Line #13 —> loads the address (**la**) of the string to be displayed in \$a0
  - after executing this instruction, the address of prompt1 will be stored in \$a0
- Everything is ready to execute the system call in line#14

## Example#2: get an integer from the user

```
17    li $v0, 5           # syscall code for read_int
18    syscall
19    move $t0, $v0       # move the first integer into $t0
```

- Line#17 —> loads read an integer syscall id (#5) in \$v0
- **No** arguments are needed here —> call syscall #5 in line 18
- After executing the syscall, the user input integer is stored in \$v0. So, Line 19 moves the integer from register \$v0 to register \$t0 for further processing.

# Q: How to know syscall details?

## A: Use help or references

Service	Code in \$v0	Arguments	Result
print integer	1	\$a0 = integer to print	
print float	2	\$f12 = float to print	
print double	3	\$f12 = double to print	
print string	4	\$a0 = address of null-terminated string to print	
read integer	5		\$v0 contains integer read
read float	6		\$f0 contains float read
read double	7		\$f0 contains double read
read string	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read	<i>See note below table</i>

# **MIPS Integer Arithmetic Instructions**

# Integer Arithmetic



- Processors may have different capabilities: **data unit** (e.g., 8-bit, 16-bit, 32-bit,..) and **types of operands** (integer only, any numbers)
- MIPS can process both **integer** and **floating-point** numbers
  - a **special coprocessor** is used for floating point operations
  - MIPS also has both **32-bit** and 64-bit architectures (but we only focus on 32-bit MIPS architecture)
- Integers** (-ve) .... 0 ..... (+ve)
- We consider **sign-bit representation** for negative numbers (simpler ALU HW)

	unsigned	Signed
Range	0 to $(2^n - 1)$ 32-bit (0 to +4,294,967,295)	$(-2^{n-1})$ to $(2^{n-1}-1)$ 32-bit (-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
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - **0**: 0000 0000 ... 0000
  - **-1**: 1111 1111 ... 1111
  - Most-positive: 0111 1111 ... 1111
  - Most-negative: 1000 0000 ... 0000

*Example*

$$10_{10} = 00\dots01010_2$$

$$-10_{10} = 11\dots10110_2$$



# MIPS Arithmetic Operations

- E.g., add, subtract
- Exactly three *operands*
  - Two sources and one destination

*Ins* \$result, \$operand1, \$operand2

*add* \$t0, \$S1, \$S2      #  $t0 = S1 + S2$

$a = b + c$

Assembly instruction operands are either *registers* or *memory operand*

*Reminder!*

MIPS Arithmetic instructions **ONLY** use register operands

Other processors may use memory or an implied register

# Key Integer MIPS Instructions

Instruction	RTL	Notes
add \$rd, \$rs, \$rt	$R[\$rd] \leftarrow R[\$rs] + R[\$rt]$	Exception on signed overflow
addu \$rd, \$rs, \$rt	$R[\$rd] \leftarrow R[\$rs] + R[\$rt]$	
sub \$rd, \$rs, \$rt	$R[\$rd] \leftarrow R[\$rs] - R[\$rt]$	Exception on signed overflow
subu \$rd, \$rs, \$rt	$R[\$rd] \leftarrow R[\$rs] - R[\$rt]$	
mtlo \$rs	$LO \leftarrow R[\$rs]$	
mult \$rs, \$rt	$\{HI, LO\} \leftarrow R[\$rs] * R[\$rt]$	Signed multiplication
multu \$rs, \$rt	$\{HI, LO\} \leftarrow R[\$rs] * R[\$rt]$	Unsigned multiplication
div \$rs, \$rt	$LO \leftarrow R[\$rs] / R[\$rt]$ $HI \leftarrow R[\$rs] \% R[\$rt]$	Signed division
divu \$rs, \$rt	$LO \leftarrow R[\$rs] / R[\$rt]$ $HI \leftarrow R[\$rs] \% R[\$rt]$	Unsigned division
mfhi \$rd	$R[\$rd] \leftarrow HI$	
mflo \$rd	$R[\$rd] \leftarrow LO$	

# Immediate Operands

- ***Constant data*** specified in an instruction

*addi* \$s3, \$s3, 4

<i>addi</i> \$rt, \$rs, imm	$R[\$rt] \leftarrow R[\$rs] + \text{SignExt}_{16b}(\text{imm})$	Exception on signed overflow
<i>addiu</i> \$rt, \$rs, imm	$R[\$rt] \leftarrow R[\$rs] + \text{SignExt}_{16b}(\text{imm})$	

- ***No subtract immediate*** instruction
  - Just use a negative constant (remember RISC)

*addi* \$s2, \$s1, -1


Guess what would  
be a common use  
for addi?



# The Zero Register

- MIPS register 0 (\$zero) is the constant 0
  - Hardwired (Cannot be overwritten)
- Useful for common operations

*add \$t2, \$s1, \$zero #no need for **move** instruction*  
*addi \$t2, \$zero, 5 #a new **load immediate (li)** instruct*  
*# canbe used for variable initializatic*



**Remember:**  
**MIPS is a RISC**  
**processor**

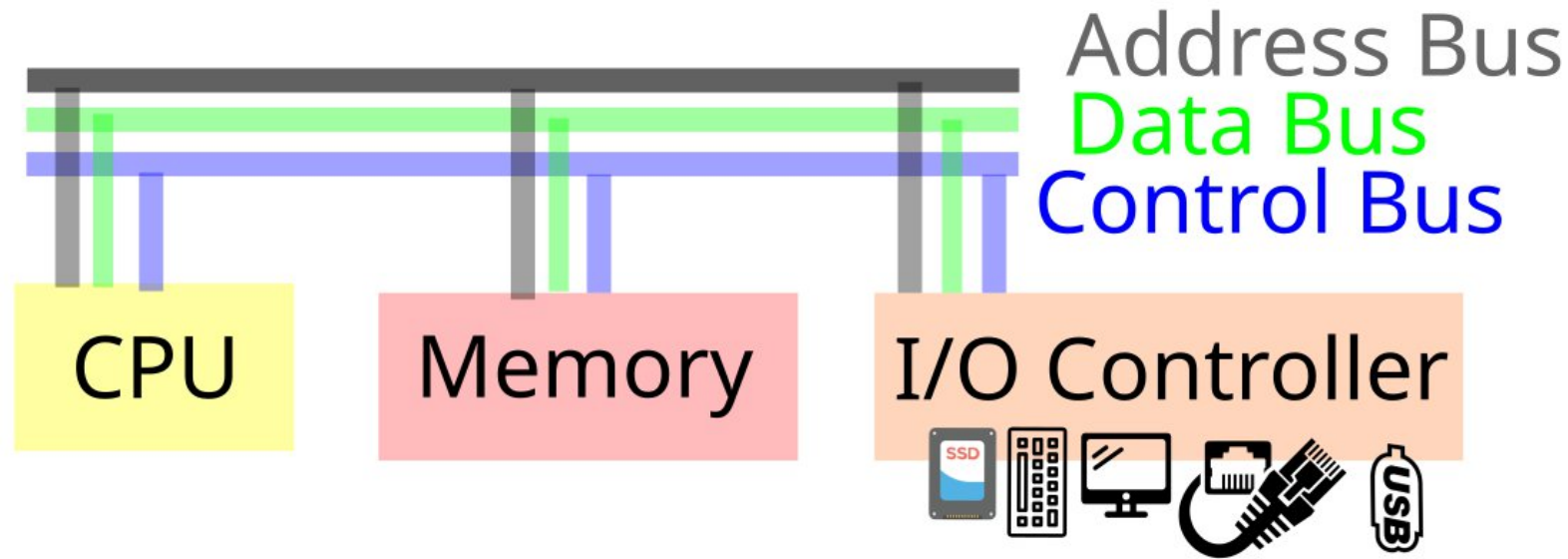
MIPS (similar to all processors) has a limited number of registers.

***How do we handle programs that process a large number of variables?***



# **Memory related instructions**

**(read from and write to memory)**



1. CPU puts the memory location address on the address bus
2. CPU activates control line to indicate whether it is a read or write operation
3. CPU puts (reads) data on (from) the data bus from (to) one of its registers

# Register vs. Memory Performance

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



# Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data

## *Data transfer instructions*

Load values from memory into registers  
Store result from register to memory

MIPS Memory is **byte-addressed**

→ you can read/write byte or larger data unit

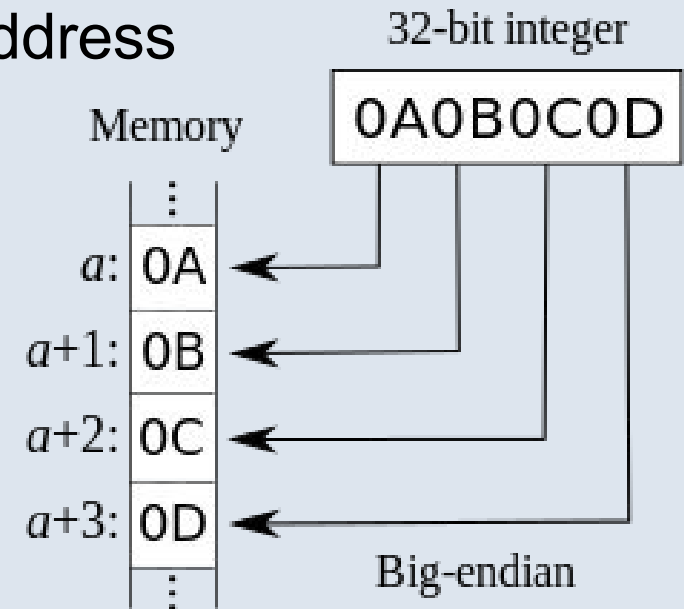
MIPS words are **aligned** in memory

- Word **can be only** accessible at an address that is a multiple of 4.
- Half-word address @ multiple of 2

MIPS is Big Endian

Most-significant byte at least address of a word

c.f. Little Endian: least-significant byte at least address



# Memory Operand Example 1

```
1 .data
2 A: .word 10, 20, 30, 40, 50
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x0000000a	0x00000014	0x0000001e	0x00000028	0x00000032

## ***base address:***

- a reference for array start
- 0x10010000 in this example
- typically obtained using load address (`la $s3, A`)

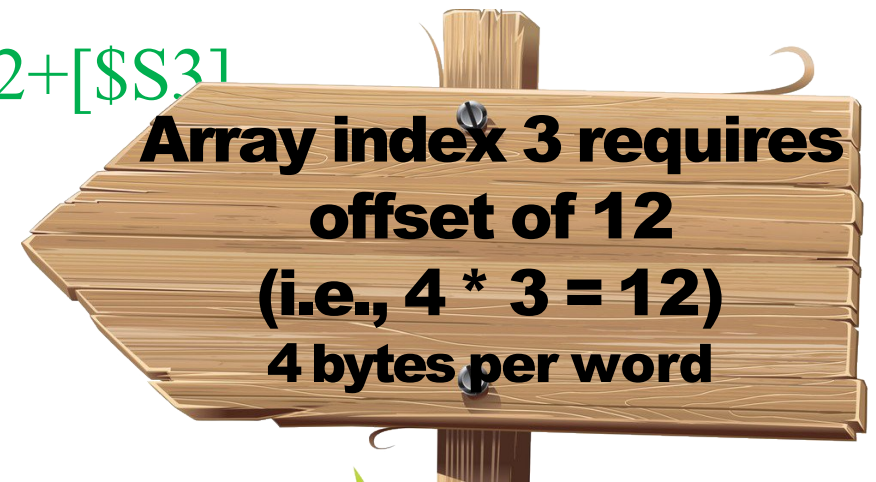
*A[] is an integer array*

`la $s3, A` # load base address

`lw $t0, 12($s3)` # load 4 bytes starting at 12+[\$s3]

offset: byte  
offset from the  
array start

***base register:***  
*address of first item  
in the array*



A program that loads an integer array of three elements from the memory **without** using loop and print the sum

```
3  .data
4  array: .word 5, 10, 15 # Integer array with three elements
5  .text
6  main:
7      # Load the three array elements into registers
8      lw $t0, array        # Load the first element into $t0
9      lw $t1, array+4      # Load the second element into $t1
10     lw $t2, array+8      # Load the third element into $t2
11     # Calculate the sum of the array elements
12     add $t3, $t0, $t1    # Add the first and second elements
13     add $t3, $t3, $t2    # Add the third element to the sum
14     # Print the sum
15     move $a0, $t3        # Move the sum to $a0 for printing
16     li $v0, 1            # Load the print integer syscall code
17     syscall
18     # Exit the program
19     li $v0, 10           # Load the exit syscall code
20     syscall
```

```

1  .data
2  array:      .word 5, 10, 15  # Integer array with three elements
3  .text
4  main:
5      la $t4, array  # Load the base address of the array into $t4
6      lw $t0, 0($t4)  # Load the first element into $t0
7      lw $t1, 4($t4)  # Load the second element into $t1
8      lw $t2, 8($t4)  # Load the third element into $t2
9      # Calculate the sum of the array elements
10     add $t3, $t0, $t1  # Add the first and second elements
11     add $t3, $t3, $t2  # Add the third element to the sum
12     # Print the sum
13     move $a0, $t3      # Move the sum to $a0 for printing
14     li $v0, 1          # Load the print integer syscall code
15     syscall
16     # Exit the program
17     li $v0, 10         # Load the exit syscall code
18     syscall

```

Same program  
using **standard lw**  
instruction

*lw \$t4, OFFSET(base)*

0x10001000

**5**

0x10001004

**10**

0x10001008

**15**



# What does every load instruction mean?

- ***la*** \$t4, label → loads memory address instruction of a label
  - After execution: \$t4=0x10001000 (address of the first array element)
- ***lw*** \$t4, array → loads memory content (4 bytes) starting at array label
  - After execution: \$t4=5 (first array element)
- ***li*** \$t4, 25 → loads \$t4 with the provided immediate
  - After execution: \$t4=25

# Key Memory MIPS Instructions

Instruction	RTL	Notes
<b>lb</b> \$rt, imm(\$rs)	$R[\$rt] \leftarrow \text{SignExt}_{8b}(\text{Mem}_{1B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm})))$	
<b>lh</b> \$rt, imm(\$rs)	$R[\$rt] \leftarrow \text{SignExt}_{16b}(\text{Mem}_{2B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm})))$	Computed address must be a multiple of 2
<b>lw</b> \$rt, imm(\$rs)	$R[\$rt] \leftarrow \text{Mem}_{4B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm}))$	Computed address must be a multiple of 4
<b>lbu</b> \$rt, imm(\$rs)	$R[\$rt] \leftarrow \{0 \times 24, \text{Mem}_{1B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm}))\}$	
<b>lhu</b> \$rt, imm(\$rs)	$R[\$rt] \leftarrow \{0 \times 16, \text{Mem}_{2B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm}))\}$	Computed address must be a multiple of 2
<b>sb</b> \$rt, imm(\$rs)	$\text{Mem}_{1B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm})) \leftarrow (R[\$rt])[7:0]$	
<b>sh</b> \$rt, imm(\$rs)	$\text{Mem}_{2B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm})) \leftarrow (R[\$rt])[15:0]$	Computed address must be a multiple of 2
<b>sw</b> \$rt, imm(\$rs)	$\text{Mem}_{4B}(R[\$rs] + \text{SignExt}_{16b}(\text{imm})) \leftarrow R[\$rt]$	Computed address must be a multiple of 4

## Signed Extension

- Needed when loading a byte, or half-word to MIPS 32-bit registers
  - **unsigned values:** extend with 0s
  - **Signed numbers:** replicate the sign bit to the left
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - -2: 1111 1110 => 1111 1111 1111 1110
- In MIPS instruction set
  - lb, lh: extend loaded byte/half-word in the register