

# Instruction Set Architecture, MIPS

# OUTLINES

- Instruction Set Architecture
- CISC (x86 Assembly) VS RISC (MIPS)
- Introduction to MIPS

# THE INSTRUCTION SET ARCHITECTURE (ISA)

- ISA is the set of instructions a computer can execute.
- All programs are combination of these instructions.
- An ISA defines everything a machine language programmer needs to know in order to program a computer.
- ISA defines a set of operations, their semantics, and rules for their use.

# WHY DO WE NEED TO STUDY DIFFERENT ARCHITECTURES?

- Every application is different, and hence requires different crucial factors that decide the architecture to use.
- Some may be performance dependent, some crucial with respect to reliability, some might require to reduce the cost.

# RISC / CISC

- Both are Instruction Set Architecture (ISA).
  - Emphasize on instructions; not the hardware.
- **CISC** = **C**omplex **I**nstruction **S**et **C**omputer
  - Slower
  - Fewer Instructions per program
  - E.g. x86
- **RISC** = **R**educed **I**nstruction **S**et **C**omputer.
  - Faster, simpler hardware.
  - More Instruction per program
  - E.g. MIPS

<https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscisc/>

# CISC

- The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible.
  - This is achieved by building processor hardware that is capable of understanding and executing a series of operations.
- One of the primary advantages of CISC is that the compiler has to do very little work to translate a high-level language statement into assembly.
- Because the length of the code is relatively short, very little RAM is required to store instructions.
- The emphasis is put on building complex instructions directly into the hardware.

# RISC

- RISC processors only use simple instructions that can be executed within one clock cycle.
- At first, this may seem like a much less efficient way of completing the operation.
  - Because there are more lines of code, more RAM is needed to store the assembly level instructions.
  - The compiler must also perform more work to convert a high-level language statement into code of this form.
- However, Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle command.
  - These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers.
- Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

# RISC

- RISC processors typically have a **load-store** architecture.
- This means there are two instructions for accessing memory:
  - I. A load (l) instruction to load data from memory and
  - II. a store (s) instruction to write data to memory.
- It also means that none of the other instructions can access memory directly.
- So, an instruction like "add this byte from memory to register 1" from a CISC instruction set would need two instructions in a load-store architecture: "load this byte from memory into register 2" and "add register 2 to register 1".



# RISC VS CISC

- Reasons for CISC:
  - Small memory
  - ASM programmers take full advantage of more complex instructions.
- With CISC, processor design complexity is the issue.
- Advantages of RISC:
  - Shorter design time
  - More general purpose registers, caches, pipelining
  - Greater Speed
  - Assembly doesn't need to closely match with HLL.

# MIPS

- **Microprocessor without Interlock Pipelined Stages.**
- **Developed by Stanford University in early 80s.**
  - Idea was to develop a processor whose architecture would represent the lowering of the compiler to the hardware level, as opposed to the raising of hardware to the software level.
- The early MIPS architectures were 32-bit, with 64-bit versions added later.

# MIPS BASIC

- **INSTRUCTIONS**

- 4 bytes (32 bits)
- 4 bytes aligned (they start at the addresses that are multiple of 4)

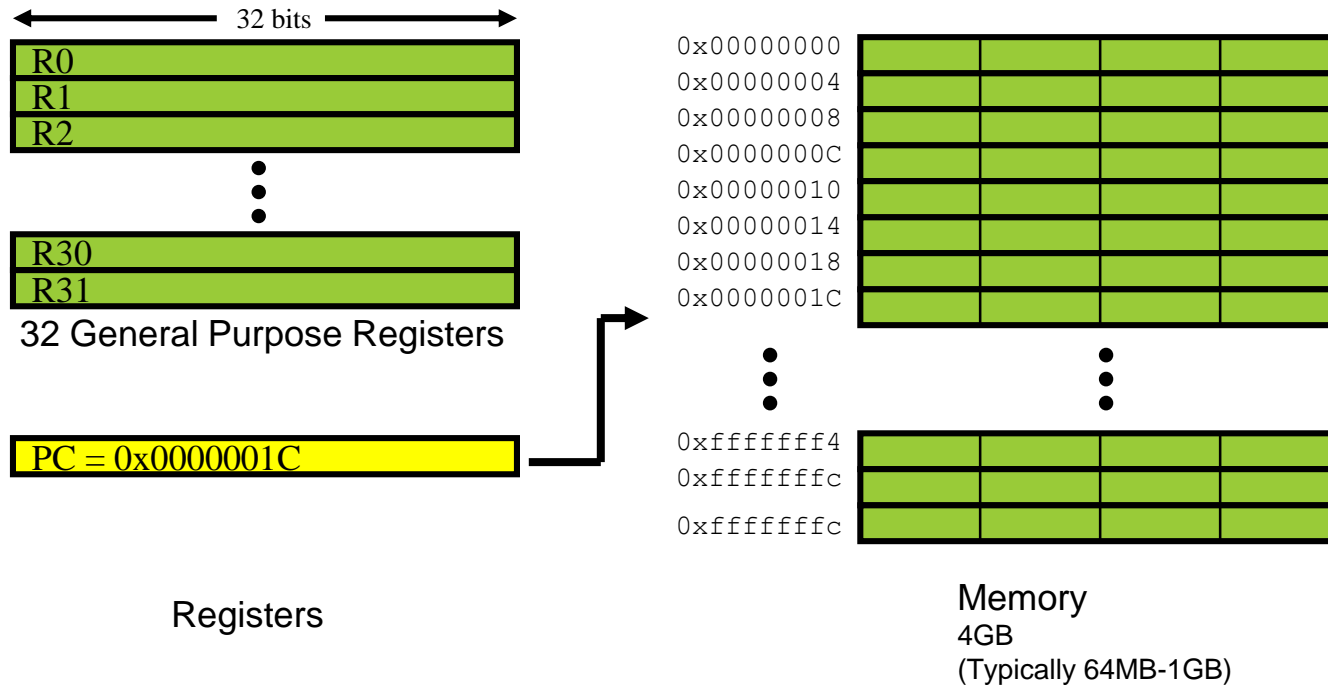
- **MEMORY DATA TYPES**

- BYTES: 8 bits
- Half Words: 16 bits
- Words: 32 Bits
- Double: 64 Bits
- Memory is denoted “M” (e.g. **M[000C]** is the byte at address 000C h)
- Floating-point must be of either word (32-bit) size or double word (64-bit) size.

- **REGISTERS**

- 32 4-byte registers in the **register file**: an array of processor registers..
- Denoted “R” (e.g. R[2] is register 2)

# MIPS32 REGISTERS AND MEMORY




Name	number	use	Callee saved
\$zero	0	zero	n/a
\$at	1	Assemble Temp	no
\$v0 - \$v1	2 - 3	return value	no
\$a0 - \$a3	4 - 7	arguments	no
\$t0 - \$t7	8 - 15	temporaries	no
\$s0 - \$s7	16 - 23	saved temporaries	yes
\$t8 - \$t9	24 - 25	temporaries	no
\$k0 - \$k1	26 - 27	Res. for OS	yes
\$gp	28	global ptr	yes
\$sp	29	stack ptr	yes
\$fp	30	frame ptr	yes
\$ra	31	return address	yes

# MIPS REGISTERS AND USAGE

Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	reserved for assembler
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporary registers
\$s0-\$s7	16-23	saved registers
\$t8-\$t9	24-25	more temporary registers
\$k0-\$k1	26-27	reserved for Operating System kernel
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Each register can be referred to by number or name.

- 
- All registers are the same.
    - Where a register is needed, any register will work
  - By convention we use them for particular tasks.
  - \$zero is the “zero register” which is always zero; writes to it have no effect.

# MISCELLANEOUS REGISTERS

In addition to the previously listed registers, there are some miscellaneous registers which are listed in the table:

Register Name	Register Usage
\$pc	Program counter
\$status or \$psw	Status Register
\$cause	Exception cause register
\$hi	Used for some multiple/divide operations
\$lo	

- The **\$pc** or program counter register points to the next instruction to be executed and is automatically updated by the CPU after instruction are executed.
- The **\$status** or status register, also called **\$psw**, is the processor status register and is updated after each instruction by the CPU.
- The **\$cause** or exception cause register is used by the CPU in the event of an exception or unexpected interruption in program control flow.
  - Examples of exceptions include division by 0, attempting to access an illegal memory address, or attempting to execute an invalid instruction (e.g., trying to execute a data item instead of code).
- The **\$hi** and **\$lo** registers are used by some specialized multiply and divide instructions.
  - For example, a multiple of two 32-bit values can generate a 64-bit result, which is stored in \$hi and \$lo (32-bits each or a total of 64-bits).



The number of available registers greatly influenced the instruction set architecture (ISA)

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003

# MEMORY

- Memory can be viewed as a series of bytes, one after another.
- Memory is byte addressable (each memory address holds one byte of information).
- To store a word, four bytes are required which use four memory addresses.
- The Least Significant Byte (LSB) is stored in the lowest memory address.
- The Most Significant Byte (MSB) is stored in the highest memory location.

For a word (32-bits), the MSB and LSB are allocated as shown below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSB																								LSB							

# MORE ABOUT MIPS MEMORY ORGANIZATION

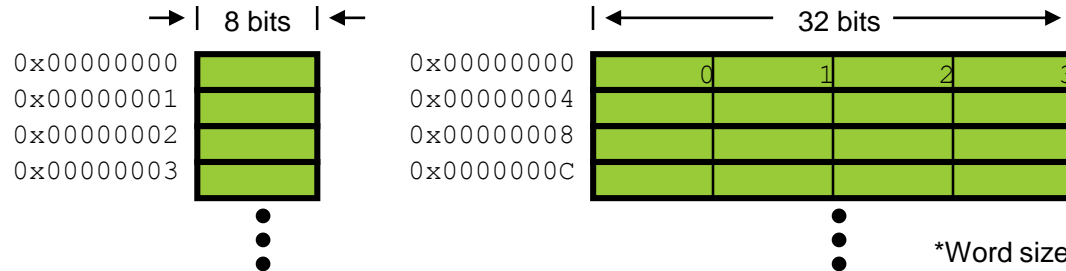
Two views of memory:

- $2^{32}$  bytes with addresses 0, 1, 2, ...,  $2^{32}-1$
- $2^{30}$  4-byte words\* with addresses 0, 4, 8, ...,  $2^{32}-4$

Both views use **byte** addresses

Not all architectures require this

Word address must be multiple of 4 (**aligned**)



\*Word sizes vary in other architectures



# BYTE ADDRESSABLE VS WORD ADDRESSABLE

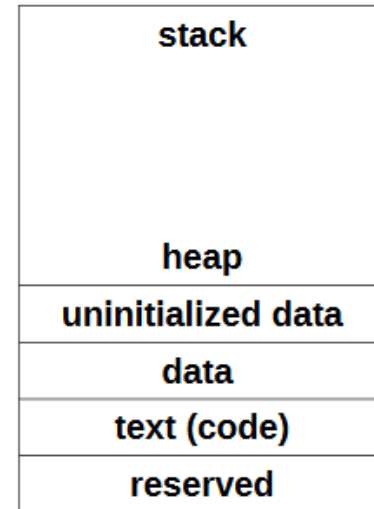
# MEMORY LAYOUT

high memory

The general memory layout for a program is as shown:

- The **reserved section** is not available to user programs (for **OS**).
- The **text (or code) section** is where the machine language (i.e., the 1's and 0's that represent the code) is stored.
- The **data section** is where the initialized data is stored.
  - This include declared variables that have been provided an initial value at assemble time.
- The **uninitialized data section** is where declared variables that have not been provided an initial value are stored.
  - If accessed before being set, the value will not be meaningful.
- The **heap** is where dynamically allocated data will be stored (if requested).
- The **stack** starts in high memory and grows downward.

low memory



# MIPS INSTRUCTION SET

- The instruction set consists of a variety of basic instructions, including:
- 21 arithmetic instructions (+, -, \*, /, %)
- 8 logic instructions (&, |, ~, XOR)
- 8 bit manipulation instructions
- 12 comparison instructions (>, <, =, >=, <=,  $\neg$ )
- 25 branch/jump instructions
- 15 load instructions
- 10 store instructions
- 8 move instructions
- 4 miscellaneous instructions

# LOADS AND STORES

Instruction name	Mnemonic	Format
Load Byte	LB	I
Load Halfword	LH	I
Load Word Left	LWL	I
Load Word	LW	I
Load Byte Unsigned	LBU	I
Load Halfword Unsigned	LHU	I
Load Word Right	LWR	I
Store Byte	SB	I
Store Halfword	SH	I
Store Word Left	SWL	I
Store Word	SW	I
Store Word Right	SWR	I

# ALU

Instruction name	Mnemonic	Format
Add	ADD	R
Add Unsigned	ADDU	R
Subtract	SUB	R
Subtract Unsigned	SUBU	R
And	AND	R
Or	OR	R
Exclusive Or	XOR	R
Nor	NOR	R
Set on Less Than	SLT	R
Set on Less Than Unsigned	SLTU	R
Add Immediate	ADDI	I
Add Immediate Unsigned	ADDIU	I
Set on Less Than Immediate	SLTI	I
Set on Less Than Immediate Unsigned	SLTIU	I
And Immediate	ANDI	I
Or Immediate	ORI	I
Exclusive Or Immediate	XORI	I
Load Upper Immediate	LUI	I



# SHIFTS

Instruction name	Mnemonic	Format
Shift Left Logical	SLL	R
Shift Right Logical	SRL	R
Shift Right Arithmetic	SRA	R
Shift Left Logical Variable	SLLV	R
Shift Right Logical Variable	SRLV	R
Shift Right Arithmetic Variable	SRAV	R

# MULT AND DIV

Instruction name	Mnemonic	Format
Move from HI	MFHI	R
Move to HI	MTHI	R
Move from LO	MFLO	R
Move to LO	MTLO	R
Multiply	MULT	R
Multiply Unsigned	MULTU	R
Divide	DIV	R
Divide Unsigned	DIVU	R

# JUMP AND BRANCH

Instruction name	Mnemonic	Format
Jump Register	JR	R
Jump and Link Register	JALR	R
Branch on Less Than Zero	BLTZ	I
Branch on Greater Than or Equal to Zero	BGEZ	I
Branch on Less Than Zero and Link	BLTZAL	I
Branch on Greater Than or Equal to Zero and Link	BGEZAL	I
Jump	J	J
Jump and Link	JAL	J
Branch on Equal	BEQ	I
Branch on Not Equal	BNE	I
Branch on Less Than or Equal to Zero	BLEZ	I
Branch on Greater Than Zero	BGTZ	I

**Instruction****Example****Meaning**

Add	Add \$1,\$2,\$3	$\$1 = \$2 + \$3$
Subtract	Sub \$1,\$2,\$3	$\$1 = \$2 - \$3$
Add immediate	Addi \$1,\$2,100	$\$1 = \$2 + 100$
Add unsigned	Addu \$1,\$2,\$3	$\$1 = \$2 + \$3$
Subtract unsigned	Subu \$1,\$2,\$3	$\$1 = \$2 - \$3$
Add immediate unsigned	Addiu \$1,\$2,100	$\$1 = \$2 + 100$
Multiply	mult \$2,\$3	Hi, lo = $\$2 * \$3$
Multiply unsigned	Multu \$2,\$3	Hi, lo = $\$2 * \$3$
Divide	Div \$2,\$3	Lo = $\$2 / \$3$ ; hi remainder
Divide unsigned	Divu \$2,\$3	Lo = $\$2 / \$3$ ; hi remainder

# MIPS INSTRUCTION TYPES

- **R-type instructions**, which perform arithmetic and logical operations on registers.
- **I-type instructions**, which deal with load/stores and immediate literal values, as well as branches.
- **J-type instructions**, which are used for jumps and function calls.

# MIPS INSTRUCTION TYPES

**Arithmetic & Logical** - manipulate data in registers

<code>add \$s1, \$s2, \$s3</code>	<code>\$s1 = \$s2 + \$s3</code>
<code>or \$s3, \$s4, \$s5</code>	<code>\$s3 = \$s4 OR \$s5</code>

**Data Transfer** - move register data to/from memory

<code>lw \$s1, 100(\$s2)</code>	<code>\$s1 = Memory[\$s2 + 100]</code>
<code>sw \$s1, 100(\$s2)</code>	<code>Memory[\$s2 + 100] = \$s1</code>

**Branch** - alter program flow

<code>beq \$s1, \$s2, 25</code>	<code>if (\$s1==\$s1) PC = PC + 4 + 4*25</code>
---------------------------------	---

# .DATA, .TEXT, .GLOBL DIRECTIVES

## **.DATA** directive

- Defines the **data segment** of a program containing data
- The program's variables should be defined under this directive
- Assembler will allocate and initialize the storage of variables

## **.TEXT** directive

- Defines the **code segment** of a program containing instructions

## **.GLOBL** directive

- Declares a symbol as **global**
- Global symbols can be referenced from other files
- We use this directive to declare *main* procedure of a program

# DATA DECLARATIONS

The data must be declared in the “.data” section.

All variables and constants are placed in this section.

Variable definitions must include the name, the data type, and the initial value for the variable.

In the definition, the variable name must be terminated with a ":".

The data type must be preceded with a "." (period).

The general format is:

```
<variableName>: .<dataType> <initialValue>
```

```
wVar1: .word 500000
```

```
wVar2: .word -100000
```

```
hVar1:  .half    5000
```

```
hVar2:  .half   -3000
```

The supported data types are as follows:

Declaration	
<b>.byte</b>	8-bit variable(s)
<b>.half</b>	16-bit variable(s)
<b>.word</b>	32-bit variable(s)
<b>.ascii</b>	ASCII string
<b>.asciiz</b>	NULL terminated ASCII string
<b>.float</b>	32 bit IEEE floating-point number
<b>.double</b>	64 bit IEEE floating-point number
<b>.space &lt;n&gt;</b>	<n> bytes of uninitialized memory



# STRING DATA DECLARATIONS

**Strings** are a series of contiguously defined byte-sized characters, typically terminated with a NULL byte (0x00).

Strings are defined with `.ascii` or `.asciiz` directives.

```
message: .asciiz "Hello World\n"
```

Define a string with multiple lines, the NULL termination would only be required on the final or last line.

For example:

```
message: .ascii "Line 1: Goodbye World\n"  
        .ascii "Line 2: So, long and thanks "  
        .ascii "for all the fish.\n"  
        .asciiz "Line 3: Game Over.\n"
```

# PROGRAM CODE

- The **code** must be preceded by the **".text "** directive.
- Naming a "main" procedure:
  - The ".globl name" and ".ent name" directives are required to define the name of the initial or main procedure.

The main procedure (as all procedures) should be terminated with the ".end <name>" directive.

```
# Text/code section
.text
.globl main
main:
...
...
.end main
```

# PSEUDO-INSTRUCTIONS VS BARE-INSTRUCTIONS

The MIPS instruction set is very small, so to do more complicated tasks we need to employ assembler macros called **pseudoinstructions**.

A pseudo-instruction is an instruction that the assembler will recognize but then convert into one or more bare-instructions.

In MIPS architecture, the assembly language includes a number of pseudo-instructions, for example, `blt`, `bgt`, `ble`, `neg`, `not`, `bge`, `li`, `la`, `move` etc.

`blt $8, $9, label`

# A branch if less than **pseudoinstruction** translates to the following bare-instructions

`blt $1, $8, $9`  
`bne $1, $0, label`

`li $8, 0x3BF20` # A load immediate value to a register translates to  
`lui $at, 0x0003`  
`ori $8, $at, 0xBF20`

A bare-instruction is an instruction that is executed by the CPU.

# NOTATIONAL CONVENTIONS

Operand Notation	Description
<b>Rdest</b>	Destination operand. Must be an integer register. Since it is a destination operand, the contents will be over written with the new result.
<b>Rsrc</b>	Source operand. Must be an integer register. Register value is unchanged after the instruction.
<b>Src</b>	Source operand. Must be an integer register or an integer immediate value. Value is unchanged after the instruction.
<b>FRdest</b>	Destination operand. Must be a floating-point register. Since it is a destination operand, the contents will be over written with the new result.
<b>FRsrc</b>	Source operand. Must be a floating-point register. Register value is unchanged after the instruction.
<b>Imm</b>	Immediate value.
<b>Mem</b>	Memory location. May be a variable name or an indirect reference (i.e., a memory address).

The general forms of the load and store instructions are as follows:

Instruction		Description
<b>l&lt;type&gt;</b>	<b>Rdest, mem</b>	Load value from memory location into destination register.
<b>li</b>	<b>Rdest, imm</b>	Load specified immediate value into destination register.
<b>la</b>	<b>Rdest, mem</b>	Load address of memory location into destination register.
<b>s&lt;type&gt;</b>	<b>Rsrc, mem</b>	Store contents of source register into memory location.

Assuming the following data declarations:

```
num:      .word      0
wnum:     .word      42
hnum:     .half      73
bnum:     .byte      7
wans:     .word      0
hans:     .half      0
bans:     .byte      0
```

To perform, the basic operations of:

```
num = 27
wans = wnum
hans = hnum
bans = bnum
```

The following instructions could be used:

```
li    $t0, 27
sw    $t0, num           # num = 27
lw    $t0, wnum
sw    $t0, wans          # wans = wnum
lh    $t1, hnum
sh    $t1, hans          # hans = hnum
lb    $t2, bnum
sb    $t2, bans          # bans = bnum
```

For the halfword and byte instructions, only the lower 16-bits are 8-bits are used.

Assuming the following data declarations:

```
wnum1:    .word    651
wnum2:    .word    42
wans1:    .word    0
wans2:    .word    0
wans3:    .word    0
hnum1:    .half    73
hnum2:    .half    15
hans:     .half    0
bnum1:    .byte    7

bnum2:    .byte    9
bans:     .byte    0
```

To perform, the basic operations of:

```
wans1 = wnum1 + wnum2
wans2 = wnum1 * wnum2
wans3 = wnum1 % wnum2
hans  = hnum1 * hnum2
bans  = bnum1 / bnum2
```

The following instructions could be used:

```
lw    $t0, wnum1
lw    $t1, wnum2
add   $t2, $t0, $t1
sw    $t2, wans1           # wans1 = wnum1 + wnum2
```

```
lw    $t0, wnum1
lw    $t1, wnum2
mul   $t2, $t0, $t1
sw    $t2, wans2           # wans2 = wnum1 * wnum2

lw    $t0, wnum1
lw    $t1, wnum2
rem   $t2, $t0, $t1
sw    $t2, wans3           # wans3 = wnum1 % wnum2

lh    $t0, hnum1
lh    $t1, hnum2
mul   $t2, $t0, $t1
sh    $t2, hans            # hans = hnum1 * hnum2

lb    $t0, bnum1
lb    $t1, bnum2
div   $t2, $t0, $t1
sb    $t2, bans            # bans = bnum1 / bnum2
```

# EXAMPLE PROGRAM: INTEGER ARITHMETIC

Program to compute the volume and surface area of a rectangular parallelepiped.  
The formulas for the volume and surface area are as follows:

$$\text{volume} = aSide * bSide * cSide$$

$$\text{surfaceArea} = 2(aSide * bSide + aSide * cSide + bSide * cSide)$$

This example main initializes the *a*, *b*, and *c* sides to arbitrary integer values.

```
# Example to compute the volume and surface area
# of a rectangular parallelepiped.

# -----
# Data Declarations
.data

aSide:      .word      73
bSide:      .word      14
cSide:      .word      16

volume:     .word      0
surfaceArea: .word      0
```



```
# -----  
# Text/code section  
  
.text  
.globl      main  
main:  
  
# -----  
# Load variables into registers.  
  
        lw    $t0, aSide  
        lw    $t1, bSide  
        lw    $t2, cSide  
  
# ----  
# Find volume of a rectangular paralllelpiped.  
#   volume = aSide * bSide * cSide  
  
        mul   $t3, $t0, $t1  
        mul   $t4, $t3, $t2  
        sw    $t4, volume
```

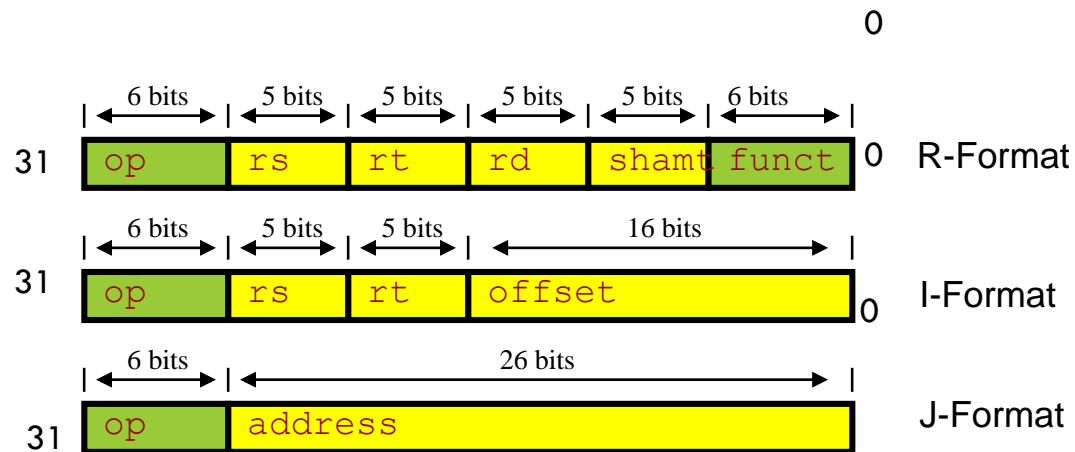


# MIPS INSTRUCTIONS

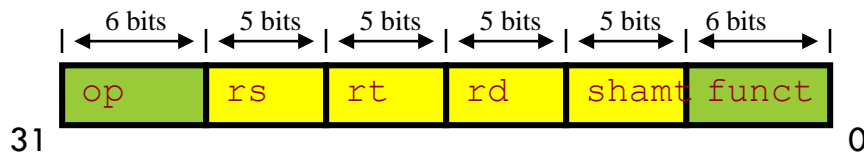
All instructions exactly 32 bits wide

Different formats for different purposes

Similarities in formats ease implementation



# ARITHMETIC & LOGICAL INSTRUCTIONS - BINARY REPRESENTATION



Used for arithmetic, logical, shift instructions

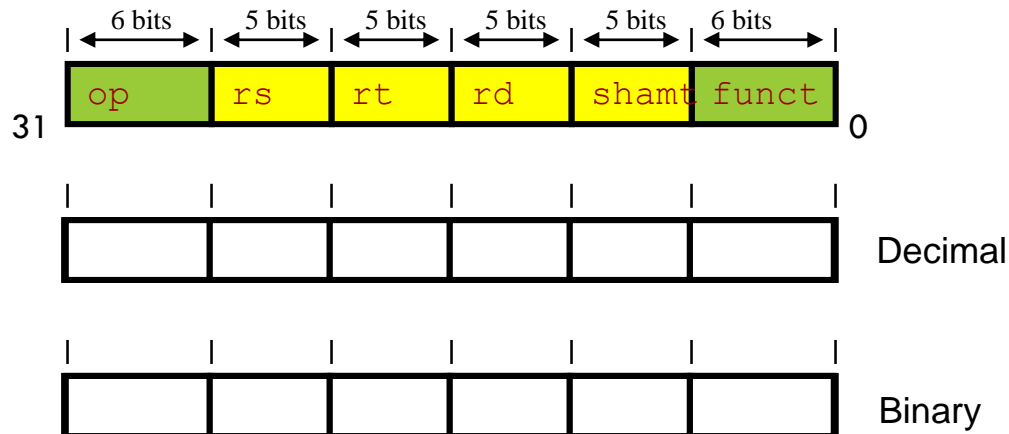
- **op**: Basic operation of the instruction (opcode)
- **rs**: first register source operand
- **rt**: second register source operand
- **rd**: register destination operand
- **shamt**: shift amount (more about this later)
- **funct**: function - specific type of operation, function code (identifies the specific R-format instruction) (6 bits)

Also called “**R-Format**” or “**R-Type**” Instructions

# ARITHMETIC & LOGICAL INSTRUCTIONS - BINARY REPRESENTATION EXAMPLE

Machine language for  
`add $8, $17, $18`

See reference card for `op`, `funct` values

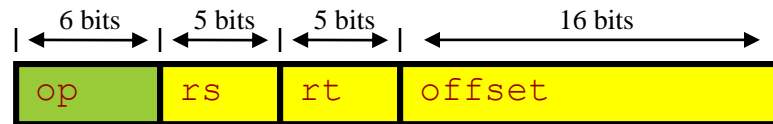


Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
		op	rs	rt	rd	shamt	funct	
add	R	0	2	3	1	0	32	add \$1, \$2, \$3
addu	R	0	2	3	1	0	33	addu \$1, \$2, \$3
sub	R	0	2	3	1	0	34	sub \$1, \$2, \$3
subu	R	0	2	3	1	0	35	subu \$1, \$2, \$3
and	R	0	2	3	1	0	36	and \$1, \$2, \$3
or	R	0	2	3	1	0	37	or \$1, \$2, \$3
nor	R	0	2	3	1	0	39	nor \$1, \$2, \$3
slt	R	0	2	3	1	0	42	slt \$1, \$2, \$3
sltu	R	0	2	3	1	0	43	sltu \$1, \$2, \$3
sll	R	0	0	2	1	10	0	sll \$1, \$2, 10
srl	R	0	0	2	1	10	2	srl \$1, \$2, 10
jr	R	0	31	0	0	0	8	jr \$31

add \$s0, \$s1, \$s2                      (registers 16, 17, 18)

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
000000	10001	10010	10000	00000	100000

# DATA TRANSFER INSTRUCTIONS - BINARY REPRESENTATION

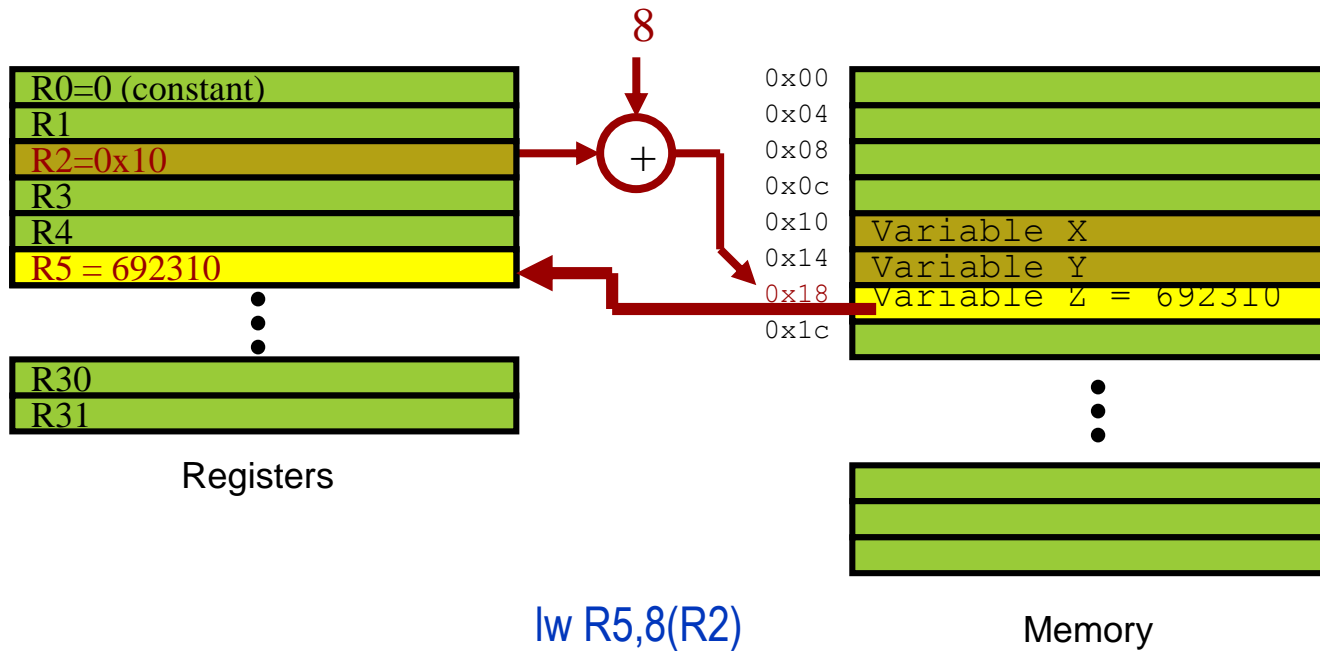


Used for load, store instructions

- **op**: Basic operation of the instruction (opcode)
- **rs**: first register source operand
- **rt**: second register source operand
- **offset**: 16-bit signed address offset (-32,768 to +32,767)

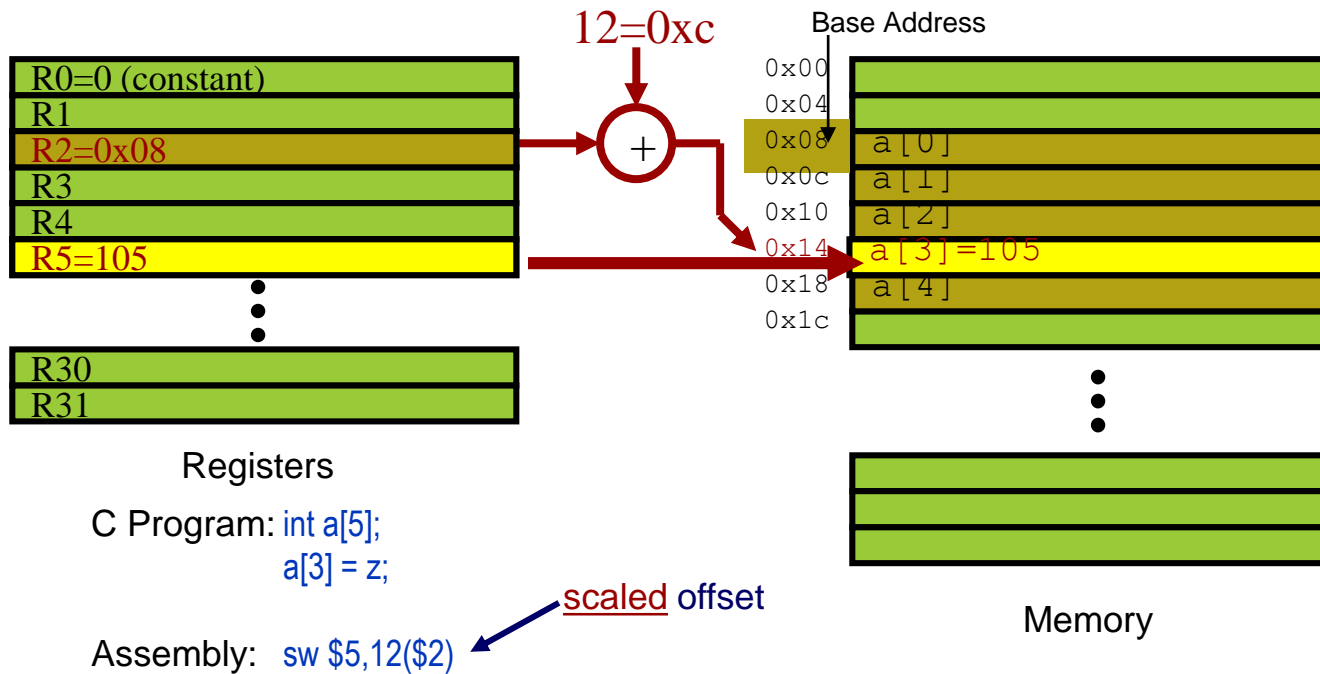
Also called “**I-Format**” or “**I-Type**” instructions

# EXAMPLE - LOADING A SIMPLE VARIABLE





# DATA TRANSFER EXAMPLE - ARRAY VARIABLE



Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
		op	rs	rt	immediate			
beq	I	4	1	2	25 (offset)			beq \$1, \$2, 100
bne	I	5	1	2	25 (offset)			bne \$1, \$2, 100
addi	I	8	2	1	100			addi \$1, \$2, 100
addiu	I	9	2	1	100			addiu \$1, \$2, 100
andi	I	12	2	1	100			andi \$1, \$2, 100
ori	I	13	2	1	100			ori \$1, \$2, 100
slti	I	10	2	1	100			slti \$1, \$2, 100
sltiu	I	11	2	1	100			sltiu \$1, \$2, 100
lui	I	15	0	1	100			lui \$1, 100
lw	I	35	2	1	100 (offset)			lw \$1, 100(\$2)
sw	I	43	2	1	100 (offset)			sw \$1, 100(\$2)

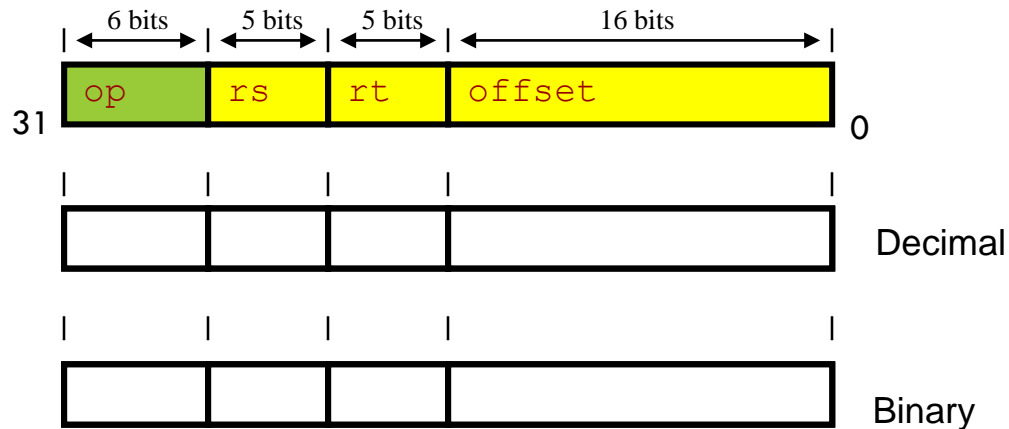
lw \$t0, 32(\$s3)      (registers 8 and 19)

op	rs	rt	immediate
35	19	8	32
100011	10011	01000	0000000000100000

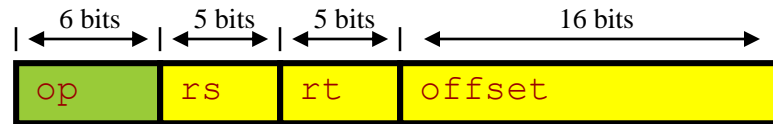
# I-FORMAT EXAMPLE

Machine language for

`lw $9, 1200($8) == lw $t1, 1200($t0)`



# BINARY REPRESENTATION - BRANCH



Branch instructions use **I-Format**

`offset` is added to PC when branch is taken

`beq r0, r1, offset`

has the effect:

Conversion to  
word offset

`if (r0==r1) pc = pc + 4 + (offset << 2)`  
`else pc = pc + 4;`

Offset is specified in instruction **words** (why?)

What is the range of the branch target addresses?

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
		op	address					
j	J	2	2500					j 10000
jal	J	3	2500					jal 10000

j LOOP

In the example above, if LOOP is at address 1028, then the value stored in the machine instruction would be 2

op	address
2	257
000010	00000000000000000000000000000001

# MIPS Assembly $\longleftrightarrow$ Machine Mappings

## Arithmetic Instructions

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
add	R	0	2	3	1	0	32	add \$1, \$2, \$3
addu	R	0	2	3	1	0	33	addu \$1, \$2, \$3
sub	R	0	2	3	1	0	34	sub \$1, \$2, \$3
subu	R	0	2	3	1	0	35	subu \$1, \$2, \$3
addi	I	8	2	1	100			addi \$1, \$2, 100
addiu	I	9	2	1	100			addiu \$1, \$2, 100

## Logical Instructions

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
and	R	0	2	3	1	0	36	and \$1, \$2, \$3
or	R	0	2	3	1	0	37	or \$1, \$2, \$3
nor	R	0	2	3	1	0	39	nor \$1, \$2, \$3
andi	I	12	2	1	100			andi \$1, \$2, 100
ori	I	13	2	1	100			ori \$1, \$2, 100
sll	R	0	0	2	1	10	0	sll \$1, \$2, 10
srl	R	0	0	2	1	10	2	srl \$1, \$2, 10

## Memory Access Instructions

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
lw	I	35	2	1	100			lw \$1, 100(\$2)
sw	I	43	2	1	100			sw \$1, 100(\$2)
lui	I	15	0	1	100			lui \$1, 100

## Branch-Related Instructions

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
beq	I	4	1	2	25			beq \$1, \$2, 100
bne	I	5	1	2	25			bne \$1, \$2, 100
slt	R	0	2	3	1	0	42	slt \$1, \$2, \$3
sltu	R	0	2	3	1	0	43	sltu \$1, \$2, \$3
slti	I	10	2	1	100			slti \$1, \$2, 100
sltiu	I	11	2	1	100			sltiu \$1, \$2, 100

## Jump Instructions

Name	Format	Layout						Example
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
jr	R	0	31	0	0	0	8	jr \$31
j	J	2	2500					j 10000
jal	J	3	2500					jal 10000



Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	$\text{Regs}[R1] \leftarrow_{64} \text{Mem}[30+\text{Regs}[R2]]$
LD R1,1000(R0)	Load double word	$\text{Regs}[R1] \leftarrow_{64} \text{Mem}[1000+0]$
LW R1,60(R2)	Load word	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[60+\text{Regs}[R2]])_0^{32} \text{ ## Mem}[60+\text{Regs}[R2]]$
LB R1,40(R3)	Load byte	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{56} \text{ ## Mem}[40+\text{Regs}[R3]]$
LBU R1,40(R3)	Load byte unsigned	$\text{Regs}[R1] \leftarrow_{64} 0^{56} \text{ ## Mem}[40+\text{Regs}[R3]]$
LH R1,40(R3)	Load half word	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{48} \text{ ## Mem}[40+\text{Regs}[R3]] \text{ ## Mem}[41+\text{Regs}[R3]]$
L.S F0,50(R3)	Load FP single	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R3]] \text{ ## } 0^{32}$
L.D F0,50(R2)	Load FP double	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R2]]$
SD R3,500(R4)	Store double word	$\text{Mem}[500+\text{Regs}[R4]] \leftarrow_{64} \text{Regs}[R3]$
SW R3,500(R4)	Store word	$\text{Mem}[500+\text{Regs}[R4]] \leftarrow_{32} \text{Regs}[R3]_{32..63}$
S.S F0,40(R3)	Store FP single	$\text{Mem}[40+\text{Regs}[R3]] \leftarrow_{32} \text{Regs}[F0]_{0..31}$
S.D F0,40(R3)	Store FP double	$\text{Mem}[40+\text{Regs}[R3]] \leftarrow_{64} \text{Regs}[F0]$
SH R3,502(R2)	Store half	$\text{Mem}[502+\text{Regs}[R2]] \leftarrow_{16} \text{Regs}[R3]_{48..63}$
SB R2,41(R3)	Store byte	$\text{Mem}[41+\text{Regs}[R3]] \leftarrow_8 \text{Regs}[R2]_{56..63}$

**Figure A.23** The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

